6 Static Analysis and Optimization

Alice:	*Do you guys mean* real *optimization?*
Riccardo:	*Well, most of the time it's local maneuvering.*
Vittorio:	*But sometimes we go beyond incremental reform . . .*
Sergio:	*. . . with provably global results.*

This chapter examines the conjunctive and first-order queries from the perspective of static analysis (in the sense of programming languages). It is shown that many properties of conjunctive queries (e.g., equivalence, containment) are decidable although they are not decidable for first-order queries. Static analysis techniques are also applied here in connection with query optimization (i.e., transforming queries expressed in a high-level, largely declarative language into equivalent queries or machine instruction programs that are arguably more efficient than a naive execution of the initial query).

To provide background, this chapter begins with a survey of practical optimization techniques for the conjunctive queries. The majority of practically oriented research and development on query optimization has been focused on variants of the conjunctive queries, possibly extended with arithmetic operators and comparators. Because of the myriad factors that play a role in query evaluation, most practically successful techniques rely heavily on heuristics.

Next the chapter presents the elegant and important *Homomorphism Theorem*, which characterizes containment and equivalence between conjunctive queries. This leads to the notion of tableau "minimization": For each tableau query there is a unique (up to isomorphism) equivalent tableau query with the smallest number of rows. This provides a theoretical notion of true optimality for conjunctive queries. It is also shown that deciding these properties and minimizing conjunctive queries is NP-complete in the size of the input queries.

Undecidability results are then presented for the first-order queries. Although related to undecidability results for conventional first-order logic, the proof techniques used here are necessarily different because all instances considered are finite by definition. The undecidability results imply that there is no hope of developing an algorithm that performs optimization of first-order queries that is complete. Only limited optimization of first-order queries involving difference is provided in most systems.

The chapter closes by returning to a specialized subset of the conjunctive queries based on *acyclic joins*. These have been shown to enjoy several interesting properties, some yielding insight into more efficient query processing.

Chapter 13 in Part D examines techniques for optimizing datalog queries.

6.1 Issues in Practical Query Optimization

Query optimization is one of the central topics of database systems. A myriad of factors play a role in this area, including storage and indexing techniques, page sizes and paging protocols, the underlying operating system, statistical properties of the stored data, statistical properties of anticipated queries and updates, implementations of specific operators, and the expressive power of the query languages used, to name a few. Query optimization can be performed at all levels of the three-level database architecture. At the physical level, this work focuses on, for example, access techniques, statistical properties of stored data, and buffer management. At a more logical level, algebraic equivalences are used to rewrite queries into forms that can be implemented more efficiently.

We begin now with a discussion of rudimentary considerations that affect query processing (including the usual cost measurements) and basic methods for accessing relations and implementing algebraic operators. Next an optimization approach based on algebraic equivalences is described; this is used to replace a given algebraic expression by an equivalent one that can typically be computed more quickly. This leads to the important notion of query evaluation plans and how they are used in modern systems to represent and choose among many alternative implementations of a query. We then examine intricate techniques for implementing multiway joins based on different orderings of binary joins and on join decomposition.

The discussion presented in this section only scratches the surface of the rich body of systems-oriented research and development on query optimizers, indicating only a handful of the most important factors that are involved. Nothing will be said about several factors, such as the impact of negation in queries, main-memory buffering strategies, and the implications of different environments (such as distributed, object oriented, real time, large main memory, and secondary memories other than conventional disks). In part due to the intricacy and number of interrelated factors involved, little of the fundamental theoretical research on query optimization has found its way into practice. As the field is maturing, salient aspects of query optimization are becoming isolated; this may provide some of the foothold needed for significant theoretical work to emerge and be applied.

The Physical Model

The usual assumption of relational databases is that the current database state is so large that it must be stored in secondary memory (e.g., on disk). Manipulation of the stored data, including the application of algebraic operators, requires making copies in primary memory of portions of the stored data and storing intermediate and final results again in secondary memory. By far the major time expense in query processing, for a single-processor system, is the number of disk pages that must be swapped in and out of primary memory. In the case of distributed systems, the communication costs typically dominate all others and become an important focus of optimization.

Viewed a little more abstractly, the physical level of relational query implementation involves three basic activities: (1) generating streams of tuples, (2) manipulating streams of tuples (e.g., to perform projections), and (3) combining streams of tuples (e.g., to perform joins, unions, and intersections). Indexing methods, including primarily B-trees and hash indexes, can be used to reduce significantly the size of some streams. Although not discussed here, it is important to consider the cost of maintaining indexes and clusterings as updates to the database occur.

Main-memory buffering techniques (including the partitioning of main memory into segments and paging policies such as deleting pages based on policies of least recent use and most recent use) can significantly impact the number of page I/Os used.

Speaking broadly, an *evaluation plan* (or *access plan*) for a query, a stored database state, and a collection of existing indexes and other data structures is a specification of a sequence of operations that will compute the answer to the query. The term evaluation plan is used most often to refer to specifications that are at a low physical level but may sometimes be used for higher-level specifications. As we shall see, query optimizers typically develop several evaluation plans and then choose one for execution.

Implementation of Algebraic Operators

To illustrate the basic building blocks from which evaluation plans are constructed, we now describe basic implementation techniques for some of the relational operators.

Selection can be realized in a straightforward manner by a scan of the argument relation and can thus be achieved in linear time. Access structures such as B-tree indexes or hash tables can be used to reduce the search time needed to find the selected tuples. In the case of selections with single tuple output, this permits evaluation within essentially constant time (e.g., two or three page fetches). For larger outputs, the selection may take two or three page fetches per output tuple; this can be improved significantly if the input relation is *clustered* (i.e., stored so that all tuples with a given attribute value are on the same or contiguous disk pages).

Projection is a bit more complex because it actually calls for two essentially different operations: *tuple rewriting* and *duplicate elimination*. The tuple rewriting is typically accomplished by bringing tuples into primary memory and then rewriting them with coordinate values permuted and removed as called for. This may yield a listing of tuples that contains duplicates. If a pure relational algebra projection is to be implemented, then these duplicates must be removed. One strategy for this involves sorting the list of tuples and then removing duplicates; this takes time on the order of $n \log n$. Another approach that is faster in some cases uses a hash function that incorporates all coordinate values of a tuple.

Because of the potential expense incurred by duplicate elimination, most practical relational languages permit duplicates in intermediate and final results. An explicit command (e.g., *distinct*) that calls for duplicate elimination is typically provided. Even in languages that support a pure algebra, it may be more efficient to leave duplicates in intermediate results and perform duplicate elimination once as a final step.

The equi-join is typically much more expensive than selection or projection because two relations are involved. The following naive *nested loop* implementation of $\bowtie_F$ will take time on the order of the product $n_1 \times n_2$ of the sizes of the input relations I_1, I_2:

```
$J := \emptyset$;
for each $u$ in $I_1$
    for each $v$ in $I_2$
        if $u$ and $v$ are joinable then $J := J \cup \{u \bowtie_F v\}$.
```

Typically this can be improved by using the *sort-merge* algorithm, which independently sorts both inputs according to the join attributes and then performs a simultaneous scan of both relations, outputting join tuples as discovered. This reduces the running time to the order of $\max(n_1 \log n_1 + n_2 \log n_2$, size of output).

In many cases a more efficient implementation of join can be accomplished by a variant of the foregoing nested loop algorithm that uses indexes. In particular, replace the inner loop by indexed retrievals to tuples of I_2 that match the tuple of I_1 under consideration. Assuming that a small number of tuples of I_2 match a given tuple of I_1, this computes the join in time proportional to the size of I_1. We shall consider implementations of multiway joins later in this section and again in Section 6.4. Additional techniques have been developed for implementing richer joins that include testing, e.g., relationships based on order ($\leq$).

Cross-product in isolation is perhaps the most expensive algebra operation: The output necessarily has size the product of the sizes of the two inputs. In practice this arises only rarely; it is much more common that selection conditions on the cross-product can be used to transform it into some form of join.

Query Trees and Query Rewriting

Alternative query evaluation plans are usually generated by rewriting (i.e., by local transformation rules). This can be viewed as a specialized case of program transformation. Two kinds of transformations are typically used in query optimization: one that maps from the higher-level language (e.g., the algebra) into the physical language, and others that stay within the same language but lead to alternative, equivalent implementations of a given construct.

We present shortly a family of rewriting rules that illustrates the general flavor of this component of query optimizers (see Fig. 6.2). Unlike true optimizers, however, the rules presented here focus exclusively on the algebra. Later we examine the larger issue of how rules such as these are used to find optimal and near-optimal evaluation plans.

We shall use the SPC algebra, generalized by permitting positive conjunctive selection and equi-join. A central concept used is that of *query tree*, which is essentially the parse tree of an algebraic expression. Consider again Query (4.4), expressed here as a rule:

$$\begin{gathered} ans(x_{th}, x_{ad}) \leftarrow Movies(x_{ti}, \text{“Bergman”}, x_{ac}),\ Pariscope(x_{th}, x_{ti}, x_s), \\ Location(x_{th}, x_{ad}, x_p). \end{gathered}$$

A naive translation into the generalized SPC algebra yields

$$q_1 = \pi_{4,8}\sigma_{2=\text{“Bergman”}}((Movies \bowtie_{1=2} Pariscope) \bowtie_{4=1} Location).$$

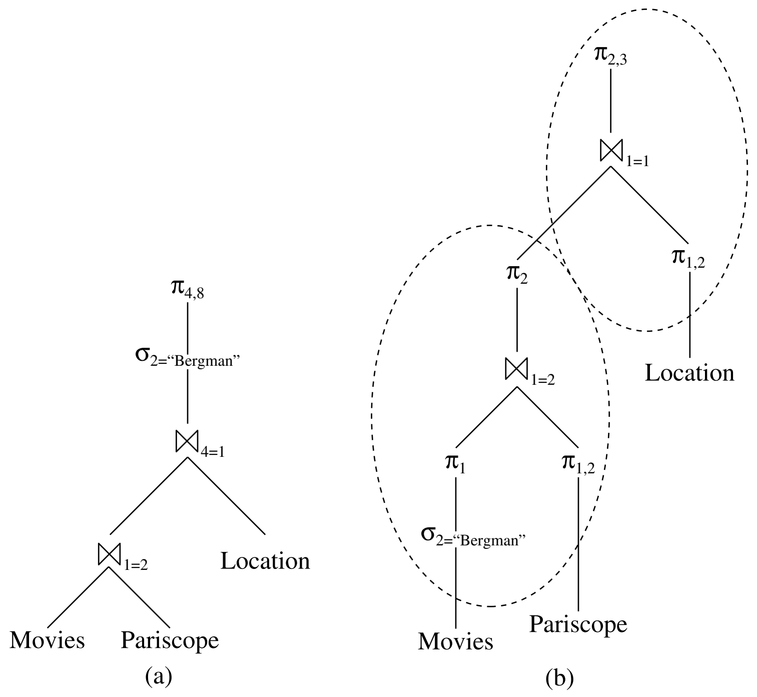


Figure 6.1: Two query trees for Query (4.4) from Chapter 4

The query tree of this expression is shown in Fig. 6.1(a).

To provide a rough idea of how evaluation costs might be estimated, suppose now that *Movies* has 10,000 tuples, with about 5 tuples per movie; *Pariscope* has about 200 tuples, and *Location* has about 100 tuples. Suppose further that in each relation there are about 50 tuples per page and that no indexes are available.

Under a naive evaluation of q_1, an intermediate result would be produced for each internal node of q_1's query tree. In this example, then, the join of *Movies* and *Pariscope* would produce about $200 \times 5 = 1000$ tuples, which (being about twice as wide as the input tuples) will occupy about 40 pages. The second equi-join will yield about 1000 tuples that fit 18 to a page, thus occupying about 55 pages. Assuming that there are four Bergman films playing in one or two theaters each, the final answer will contain about six tuples. The total number of page fetches performed here is about 206 for reading the input relations (assuming that no indexes are available) and 95 for working with the intermediate relations. Additional page fetches might be required by the join operations performed.

Consider now the query q_2 whose query tree is illustrated in Fig. 6.1(b). It is easily verified that this is equivalent to q_1. Intuitively, q_2 was formed from q_1 by "pushing" selections and projections as far "down" the tree as possible; this generally reduces the size of intermediate results and thus of computing with them.

In this example, assuming that all (i.e., about 20) of Bergman's films are in *Movies*, the selection on *Movies* will yield about 100 tuples; when projected these will fit onto a single page. Joining with *Pariscope* will yield about six tuples, and the final join with *Location* will again yield six tuples. Thus only one page is needed to hold the intermediate results constructed during this evaluation, a considerable savings over the 95 pages needed by the previous one.

It is often beneficial to combine several algebraic operators into a single implemented operation. As a general rule of thumb, it is typical to materialize the inputs of each equi-join. The equi-join itself and all unary operations directly above it in the query tree are performed before output. The dashed ovals of Fig. 6.1(b) illustrate a natural grouping that can be used for this tree. In practical systems, the implementation and grouping of operators is typically considered in much finer detail.

The use of different query trees and, more generally, different evaluation plans can yield dramatically different costs in the evaluation of equivalent queries. Does this mean that the user will have to be extremely careful in expressing queries? The beauty of query optimization is that the answer is a resounding no. The user may choose any representation of a query, and the system will be responsible for generating several equivalent evaluation plans and choosing the least expensive one. For this reason, even though the relational algebra is conceptually procedural, it is implemented as an essentially declarative language.

In the case of the algebra, the generation of evaluation plans is typically based on the existence of rules for transforming algebraic expressions into equivalent ones. We have already seen rewrite rules in the context of transforming SPC and SPJR expressions into normal form (see Propositions 4.4.2 and 4.4.6). A different set of rules is useful in the present context due to the focus on optimizing the execution time and space requirements.

In Fig. 6.2 we present a family of representative rewrite rules (three with inverses) that can be used for performing the transformations needed for optimization at the logical level. In these rules we view cross-product as a special case of equi-join in which the selection formula is empty. Because of their similarity to the rules used for the normal form results, several of the rules are shown only in abstract form; detailed formulation of these, as well as verification of their soundness, is left for the reader (see Exercise 6.1). We also include the following rule:

Simplify-identities: replace $\pi_{1,\ldots,arity(q)}q$ by q; replace $\sigma_{i=i}q$ by q; replace $q \times \{\langle\rangle\}$ by q; replace $q \times \{\}$ by $\{\}$; and replace $q \bowtie_{1=1\wedge\cdots\wedge arity(q)=arity(q)} q$ by q.

Generating and Choosing between Evaluation Plans

As suggested in Fig. 6.2, in most cases the transformations should be performed in a certain direction. For example, the fifth rule suggests that it is always desirable to push selections through joins. However, situations can arise in which pushing a selection through a join is in fact much more costly than performing it second (see Exercise 6.2). The broad variety of factors that influence the time needed to execute a given query evaluation plan make it virtually impossible to find an optimal one using purely analytic techniques. For this reason, modern optimizers typically adopt the following pragmatic strategy: (1) generate a possibly large number of alternative evaluation plans; (2) estimate the costs of executing

$$\begin{array}{lcl}
\sigma_F(\sigma_{F'}(q)) & \leftrightarrow & \sigma_{F\wedge F'}(q) \\
\pi_{\vec{j}}(\pi_{\vec{k}}(q)) & \leftrightarrow & \pi_{\vec{l}}(q) \\
\sigma_F(\pi_{\vec{l}}(q)) & \leftrightarrow & \pi_{\vec{l}}(\sigma_{F'}(q)) \\
q_1 \bowtie q_2 & \leftrightarrow & q_2 \bowtie q_1 \\
\sigma_F(q_1 \bowtie_G q_2) & \rightarrow & \sigma_F(q_1) \bowtie_G q_2 \\
\sigma_F(q_1 \bowtie_G q_2) & \rightarrow & q_1 \bowtie_G \sigma_{F'}(q_2) \\
\sigma_F(q_1 \bowtie_G q_2) & \rightarrow & q_1 \bowtie_{G'} q_2 \\
\pi_{\vec{l}}(q_1 \bowtie_G q_2) & \rightarrow & \pi_{\vec{l}}(q_1) \bowtie_{G'} q_2 \\
\pi_{\vec{l}}(q_1 \bowtie_G q_2) & \rightarrow & q_1 \bowtie_{G'} \pi_{\vec{k}}(q_2)
\end{array}$$

Figure 6.2: Rewriting rules for SPC algebra

them; and (3) select the one of lowest cost. The database system then executes the selected evaluation plan.

In early work, the transformation rules used and the method for evaluation plan generation were essentially intermixed. Motivated in part by the desire to make database systems extensible, more recent proposals have isolated the transformation rules from the algorithms for generating evaluation plans. This has the advantages of exposing the semantics of evaluation plan generation and making it easier to incorporate new kinds of information into the framework.

A representative system for generating evaluation plans was developed in connection with the Exodus database toolkit. In this system, techniques from AI are used and, a set of transformation rules is assumed. During processing, a set of partial evaluation plans is maintained along with a set of possible locations where rules can be applied. Heuristics are used to determine which transformation to apply next, so that an exhaustive search through all possible evaluation plans can be avoided while still having a good chance of finding an optimal or near-optimal evaluation plan. Several of the heuristics include weighting factors that can be tuned, either automatically or by the dba, to reflect experience gained while using the optimizer.

Early work on estimating the cost of evaluation plans was based essentially on "thought experiments" similar to those used earlier in this chapter. These analyses use factors including the size of relations, their expected statistical properties, selectivity factors of joins and selections, and existing indexes. In the context of large queries involving multiple joins, however, it is difficult if not impossible to predict the sizes of intermediate results based only on statistical properties. This provides one motivation for recent research on using random and background sampling to estimate the size of subquery answers, which can provide more reliable estimates of the overall cost of an evaluation plan.

Sideways Information Passing

We close this section by considering two practical approaches to implementing multiway joins as they arise in practical query languages.

Much of the early research on practical query optimization was performed in connection with the System R and INGRES systems. The basic building block of the query languages used in these systems (SQL and Quel, respectively) takes the form of "select-from-where" clauses or blocks. For example, as detailed further in Chapter 7, Query (4.4) can be expressed in SQL as

select *Theater, Address*
from *Movies*, *Location*, *Pariscope*
where *Director* = "Bergman"
  and *Movies.Title* = *Pariscope.Title*
  and *Pariscope.Theater* = *Location.Theater*.

This can be translated into the algebra as a join between the three relations of the **from** part, using join condition given by the **where** and projecting onto the columns mentioned in the **select**. Thus a typical select-from-where block can be expressed by an SPC query as

$$\pi_{\vec{j}}(\sigma_F(R_1 \times \cdots \times R_n)).$$

With such expressions, the System R query optimizer pushes selections that affect a single relation into the join and then considers evaluation plans based on *left-to-right* joins that have the form

$$(\ldots(R_{i_1} \bowtie R_{i_2}) \bowtie \cdots \bowtie R_{i_n})$$

using different orderings $R_{i_1}, \ldots, R_{i_n}$. We now present a heuristic based on "sideways information passing," which is used in the System R optimizer for eliminating some possible orderings from consideration. Interestingly, this heuristic has also played an important role in developing evaluation techniques for recursive datalog queries, as discussed in Chapter 13.

To describe the heuristic, we rewrite the preceding SPC query as a (generalized) rule that has the form

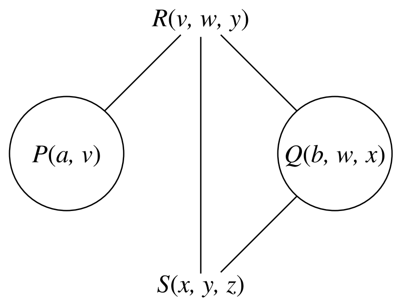
$$(*) \qquad ans(u) \leftarrow R_1(u_1), \ldots, R_n(u_n), C_1, \ldots, C_m,$$

where all equalities of the selection condition F are incorporated by using constants and equating variables in the free tuples $u_1, \ldots, u_n$, and the expressions $C_1, \ldots, C_m$ are conditions in the selection condition F not captured in that way. (This might include, e.g., inequalities and conditions based on order.) We shall call the $R_i(u_i)$'s *relation atoms* and the C_j's *constraint atoms*.

Example 6.1.1 Consider the rule

$$ans(z) \leftarrow P(a, v), Q(b, w, x), R(v, w, y), S(x, y, z), v \leq x,$$

where a, b denote constants. A common assumption in this case is that there are few values for v such that $P(a, v)$ is satisfied. This in turn suggests that there will be few triples (v, w, y) satisfying $P(a, v) \wedge R(v, w, y)$. Continuing by transitivity, then, we also expect there to be few 5-tuples (v, w, y, x, z) that satisfy the join of this with $S(x, y, z)$.



6-3.eps

Figure 6.3: A sip graph

More generally, the *sideways information passing graph*, or *sip graph*, of a rule ρ that has the form $(*)$ just shown has vertexes the set of relation atoms of a rule, and includes an undirected edge between atoms $R_i(u_i)$, $R_j(u_j)$ if u_i and u_j have at least one variable in common. Furthermore, each node with a constant appearing is specially marked. The sip graph for the rule of Example 6.1.1 is shown in Fig. 6.3.

Let us assume that the sip graph for a rule ρ is connected. In this case, a sideways information passing *strategy* (*sip strategy*) for ρ is an ordering $A_1, \ldots, A_n$ of the atoms in the rule, such that for each $j > 1$, either

(a) a constant occurs in A_j;

(b) A_j is a relational atom and there is at least one $i < j$ such that $\{A_i, A_j\}$ is an edge of the sip graph of (ρ); or

(c) A_j is a constraint atom and each variable occurring in A_j occurs in some atom A_i for $i < j$.

Example 6.1.2 A representative sample of the several sip strategies for the rule of Example 6.1.1 is as follows:

$$P(a, v), Q(b, w, x), v \le x, R(v, w, y), S(x, y, z)$$
$$P(a, v), R(v, w, y), S(x, y, z), v \le x, Q(b, w, x)$$
$$Q(b, w, x), R(v, w, y), P(a, v), S(x, y, z), v \le x.$$

A sip strategy for the case in which the sip graph of rule ρ is not connected is a set of sip strategies, one for each connected component of the sip graph. (Incorporation of constraint atoms whose variables lie in distinct components is left for the reader.) The System R optimizer focuses primarily on joins that have connected sip graphs, and it considers only those join orderings that correspond to sip strategies. In some cases a more efficient evaluation plan can be obtained if an arbitrary tree of binary joins is permitted; see Exercise 6.5. While generating sip strategies the System R optimizer also considers alternative implementations for the binary joins involved and records information about the orderings that the partial results would have if computed. An additional logical-level technique used in System R is illustrated in the following example.

Example 6.1.3 Let us consider again the rule

$$ans(z) \leftarrow P(a, v), R(v, w, y), S(x, y, z), v \leq x, Q(b, w, x).$$

Suppose that a left-to-right join is performed according to the sip strategy shown. At different intermediate stages certain variables can be "forgotten," because they are not used in the answer, nor are they used in subsequent joins. In particular, after the third atom the variable y can be projected out, after the fourth atom v can be projected out, and after the fifth atom w and x can be projected out. It is straightforward to formulate a general policy for when to project out unneeded variables (see Exercise 6.4).

Query Decomposition: Join Detachment and Tuple Substitution

We now briefly discuss the two main techniques used in the original INGRES system for evaluating join expressions. Both are based on decomposing multiway joins into smaller ones.

While again focusing on SPC queries of the form

$$\pi_{\vec{j}}(\sigma_F(R_1 \times \cdots \times R_n))$$

for this discussion, we use a slightly different notation. In particular, tuple variables rather than domain variables are used. We consider expressions of the form

$$(**) \qquad ans(s) \leftarrow R_1(s_1), \ldots, R_n(s_n), C_1, \ldots, C_m, T,$$

where $s, s_1, \ldots, s_n$ are tuple variables; $C_1, \ldots, C_n$ are Boolean conditions referring to coordinates of the variables $s_1, \ldots, s_n$ (e.g., $s_1.3 = s_4.1 \vee s_2.4 = a$); and T is a *target condition* that gives a value for each coordinate of the target variable s. It is generally assumed that none of $C_1, \ldots, C_n$ has $\wedge$ as its parent connective.

A condition C_j is called *single variable* if it refers to only one of the variables s_i. At any point in the processing it is possible to apply one or more single-variable conditions to some R_i, thereby constructing an intermediate relation R'_i that can be used in place of R_i. In the INGRES optimizer, this is typically combined with other steps.

Join detachment is useful for separating a query into two separate queries, where the second refers to the first. Consider a query that has the specialized form

$$(\dagger) \qquad \begin{aligned} ans(t) \leftarrow\ & P_1(p_1), \ldots, P_m(p_m), C_1, \ldots, C_k, T, \\ & Q(q), \\ & R_1(r_1), \ldots, R_n(r_n), D_1, \ldots, D_l, \end{aligned}$$

where conditions $C_1, \ldots, C_k, T$ refer only to variables $t, p_1, \ldots, p_m, q$ and $D_1, \ldots, D_l$ refer only to $q, r_1, \ldots, r_n$. It is easily verified that this is equivalent to the sequence

$$\begin{aligned} \mathit{temp}(q) &\leftarrow Q(q), R_1(r_1), \ldots, R_n(r_n), D_1, \ldots, D_l \\ \mathit{ans}(t) &\leftarrow P_1(p_1), \ldots, P_m(p_m), \mathit{temp}(q), C_1, \ldots, C_k, T. \end{aligned}$$

In this example, variable q acts as a "pivot" around which the detachment is performed. More general forms of join detachment can be developed in which a set of variables serves as the pivot (see Exercise 6.6).

Tuple substitution chooses one of the underlying relations R_j and breaks the n-variable join into a set of $(n-1)$-variable joins, one for each tuple in R_j. Consider again a query of form $(**)$ just shown. The tuple substitution of this on R_i is given by the "program"

```
**for each** $r$ **in**$R_i$ **do**
    $ans(s) \mathrel{+}\!\!\leftarrow R_1(s_1), \ldots, R_{i-1}(s_{i-1}), R_{i+1}(s_{i+1}), \ldots, R_n(s_n),$
            $(C_1, \ldots, C_m, T)[s_i/r].$
```

Here we use $+\!\!\leftarrow$ to indicate that *ans* is to accumulate the values stemming from all tuples r in (the value of) R_i; furthermore, r is substituted for s_i in all of the conditions.

There is an obvious trade-off here between reducing the number of variables in the join and the number of tuples in R_i. In the INGRES optimizer, each of the R_i's is considered as a candidate for forming the tuple substitution. During this process single-variable conditions may be applied to the R_i's to decrease their size.

6.2 Global Optimization

The techniques for creating evaluation plans presented in the previous section are essentially *local* in their operation: They focus on clusters of contiguous nodes in a query tree. In this section we develop an approach to the *global* optimization of conjunctive queries. This allows a transformation of an algebra query that removes several joins in a single step, a capability not provided by the techniques of the previous section. The global optimization technique is based on an elegant Homomorphism Theorem.

The Homomorphism Theorem

For two queries q_1, q_2 over the same schema $\mathbf{R}$, q_1 is *contained* in q_2, denoted $q_1 \subseteq q_2$, if for each $\mathbf{I}$ over $\mathbf{R}$, $q_1(\mathbf{I}) \subseteq q_2(\mathbf{I})$. Clearly, $q_1 \equiv q_2$ iff $q_1 \subseteq q_2$ and $q_2 \subseteq q_1$. The Homomorphism Theorem provides a characterization for containment and equivalence of conjunctive queries.

We focus here on the tableau formalism for conjunctive queries, although the rule-based formalism could be used equally well. In addition, although the results hold for tableau queries over database schemas involving more than one relation, the examples presented focus on queries over a single relation.

Recall the notion of *valuation*—a mapping from variables to constants extended to be the identity on constants and generalized to free tuples and tableaux in the natural fashion.

R	A	B
	x	y
	x	y

$q_0 = (T_0, \langle x, y\rangle)$

(a)

R	A	B
	x	y_1
	x_1	y_1
	x_1	y
	x	y

$q_1 = (T_1, \langle x, y\rangle)$

(b)

R	A	B
	x	y_1
	x_1	y_1
	x_1	y_2
	x_2	y_2
	x_2	y
	x	y

$q_2 = (T_2, \langle x, y\rangle)$

(c)

R	A	B
	x	y_1
	x_1	y
	x	y

$q_\omega = (T_\omega, \langle x, y\rangle)$

(d)

Figure 6.4: Tableau queries used to illustrate the Homomorphism Theorem

Valuations are used in the definition of the semantics of tableau queries. More generally, a *substitution* is a mapping from variables to variables and constants, which is extended to be the identity on constants and generalized to free tuples and tableaux in the natural fashion. As will be seen, substitutions play a central role in the Homomorphism Theorem.

We begin the discussion with two examples. The first presents several simple examples of the Homomorphism Theorem in action.

Example 6.2.1 Consider the four tableau queries shown in Fig. 6.4. By using the Homomorphism Theorem, it can be shown that $q_0 \subseteq q_1 \subseteq q_2 \subseteq q_\omega$.

To illustrate the flavor of the proof of the Homomorphism Theorem, we argue informally that $q_1 \subseteq q_2$. Note that there is substitution θ such that $\theta(T_2) \subseteq T_1$ and $\theta(\langle x, y\rangle) = \langle x, y\rangle$ [e.g., let $\theta(x_1) = \theta(x_2) = x_1$ and $\theta(y_1) = \theta(y_2) = y_1$]. Now suppose that I is an instance over AB and that $t \in q_1(I)$. Then there is a valuation ν such that $\nu(T_1) \subseteq I$ and $\nu(\langle x, y\rangle) = t$. It follows that $\theta \circ \nu$ is a valuation that embeds T_2 into I with $\theta \circ \nu(\langle x, y\rangle) = t$, whence $t \in q_2(I)$.

Intuitively, the existence of a substitution embedding the tableau of q_2 into the tableau of q_1 and mapping the summary of q_2 to the summary of q_1 implies that q_1 is *more restrictive* than q_2 (or more correctly, *no less restrictive* than q_2.) Surprisingly, the Homomorphism Theorem states that this is also a necessary condition for containment (i.e., if $q \subseteq q'$, then q is more restrictive than q' in this sense).

The second example illustrates a limitation of the techniques discussed in the previous section.

Example 6.2.2 Consider the two tableau queries shown in Fig. 6.5. It can be shown that $q \equiv q'$ but that q' cannot be obtained from q using the rewrite rules of the previous section (see Exercise 6.3) or the other optimization techniques presented there.

R	A	B
	x	x
	x	y_1
	y_1	y_2
	$\vdots$	$\vdots$
	y_{n-1}	y_n
	y_n	x
	x	

$q = (T, u)$

(a)

R	A	B
	x	x
	x	

$q' = (T', u)$

(b)

Figure 6.5: Pair of equivalent tableau queries

Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be two tableau queries over the same schema $\mathbf{R}$. A *homomorphism* from q' to q is a substitution θ such that $\theta(\mathbf{T}') \subseteq \mathbf{T}$ and $\theta(u') = u$.

Theorem 6.2.3 (Homomorphism Theorem) Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be tableau queries over the same schema $\mathbf{R}$. Then $q \subseteq q'$ iff there exists a homomorphism from $(\mathbf{T}', u')$ to $(\mathbf{T}, u)$.

Proof Suppose first that there exists a homomorphism θ from q' to q. Let $\mathbf{I}$ be an instance over $\mathbf{R}$. To see that $q(\mathbf{I}) \subseteq q'(\mathbf{I})$, suppose that $w \in q(\mathbf{I})$. Then there is a valuation ν that embeds $\mathbf{T}$ into $\mathbf{I}$ such that $\nu(u) = w$. It is clear that $\theta \circ \nu$ embeds $\mathbf{T}'$ into $\mathbf{I}$ and $\theta \circ \nu(u') = w$, whence $w \in q'(\mathbf{I})$ as desired.

For the opposite inclusion, suppose that $q \subseteq q'$ [i.e., that $(\mathbf{T}, u) \subseteq (\mathbf{T}', u')$]. Speaking intuitively, we complete the proof by applying both q and q' to the "instance" $\mathbf{T}$. Because q will yield the free tuple u, q' also yields u (i.e., there is an embedding θ of $\mathbf{T}'$ into $\mathbf{T}$ that maps u' to u). To make this argument formal, we construct an instance $\mathbf{I_T}$ that is isomorphic to $\mathbf{T}$.

Let V be the set of variables occurring in $\mathbf{T}$. For each $x \in V$, let a_x be a new distinct constant not occurring in $\mathbf{T}$ or $\mathbf{T}'$. Let μ be the valuation mapping each x to a_x, and let $\mathbf{I_T} = \mu(\mathbf{T})$. Because μ is a bijection from V to $\mu(V)$, and because $\mu(V)$ has empty intersection with the constants occurring in $\mathbf{T}$, the inverse μ^{-1} of μ is well defined on $adom(\mathbf{I_T})$.

It is clear that $\mu(u) \in q(\mathbf{I_T})$, and so by assumption, $\mu(u) \in q'(\mathbf{I_T})$. Thus there is a valuation ν that embeds $\mathbf{T}'$ into $\mathbf{I_T}$ such that $\nu(u') = \mu(u)$. It is now easily verified that $\nu \circ \mu^{-1}$ is a homomorphism from q' to q. ■

Permitting a slight abuse of notation, we have the following (see Exercise 6.8).

Corollary 6.2.4 For tableau queries $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$, $q \subseteq q'$ iff $u \in q'(\mathbf{T})$.

We also have

Corollary 6.2.5 Tableau queries q, q' over schema $\mathbf{R}$ are equivalent iff there are homomorphisms from q to q' and from q' to q.

In particular, if $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ are equivalent, then u and u' are identical up to one-one renaming of variables.

Only one direction of the preceding characterization holds if the underlying domain is finite (see Exercise 6.12). In addition, the direct generalization of the theorem to tableau queries with equality does not hold (see Exercise 6.9).

Query Optimization by Tableau Minimization

Although the Homomorphism Theorem yields a decision procedure for containment and equivalence between conjunctive queries, it does not immediately provide a mechanism, given a query q, to find an "optimal" query equivalent to q. The theorem is now applied to obtain just such a mechanism.

We note first that there are simple algorithms for translating tableau queries into (satisfiable) SPC queries and vice versa. More specifically, given a tableau query, the corresponding generalized SPC query has the form $\pi_{\vec{j}}(\sigma_F(R_1 \times \cdots \times R_k))$, where each component R_i corresponds to a distinct row of the tableau. For the opposite direction, one algorithm for translating SPC queries into tableau queries is first to translate into the normal form for generalized SPC queries and then into a tableau query. A more direct approach that inductively builds tableau queries corresponding to subexpressions of an SPC query can also be developed (see Exercise 4.18). Analogous remarks apply to SPJR queries.

The goal of the optimization presented here is to minimize the number of rows in the tableau. Because the number of rows in a tableau query is one more than the number of joins in the SPC (SPJR) query corresponding to that tableau (see Exercise 4.18c), the tableau minimization procedure provides a way to minimize the number of joins in SPC and SPJR queries.

Surprisingly, we show that an optimal tableau query equivalent to tableau query q can be obtained simply by eliminating some rows from the tableau of q.

We say that a tableau query $(\mathbf{T}, u)$ is *minimal* if there is no query $(\mathbf{S}, v)$ equivalent to $(\mathbf{T}, u)$ with $|\mathbf{S}| < |\mathbf{T}|$ (i.e., where $\mathbf{S}$ has strictly fewer rows than $\mathbf{T}$).

We can now demonstrate the following.

Theorem 6.2.6 Let $q = (\mathbf{T}, u)$ be a tableau query. Then there is a subset $\mathbf{T}'$ of $\mathbf{T}$ such that $q' = (\mathbf{T}', u)$ is a minimal tableau query and $q' \equiv q$.

Proof Let $(\mathbf{S}, v)$ be a minimal tableau that is equivalent to q. By Corollary 6.2.5, there are homomorphisms θ from q to $(\mathbf{S}, v)$ and λ from $(\mathbf{S}, v)$ to q. Let $\mathbf{T}' = \theta \circ \lambda(\mathbf{S})$. It is straightforward to verify that $(\mathbf{T}', u) \equiv q$ and $|\mathbf{T}'| \leq |\mathbf{S}|$. By minimality of $(\mathbf{S}, v)$, it follows that $|\mathbf{T}'| = |\mathbf{S}|$, and $(\mathbf{T}', u)$ is minimal. ■

Example 6.2.7 illustrates how one might minimize a tableau by hand.

R	A	B	C
u_1	x_2	y_1	z
u_2	x	y_1	z_1
u_3	x_1	y	z_1
u_4	x	y_2	z_2
u_5	x_2	y_2	z
u	x	y	z

Figure 6.6: The tableau (T, u)

Example 6.2.7 Let R be a relation schema of sort ABC and (T, u) the tableau over R in Fig. 6.6. To minimize (T, u), we wish to detect which rows of T can be eliminated. Consider u_1. Suppose there is a homomorphism θ from (T, u) onto itself that eliminates u_1 [i.e., $u_1 \notin \theta(T)$]. Because any homomorphism on (T, u) is the identity on u, $\theta(z) = z$. Thus $\theta(u_1)$ must be u_5. But then $\theta(y_1) = y_2$, and $\theta(u_2) \in \{u_4, u_5\}$. In particular, $\theta(z_1) \in \{z_2, z\}$. Because u_3 involves z_1, it follows that $\theta(u_3) \neq u_3$ and $\theta(y) \neq y$. But the last inequality is impossible because y is in u so $\theta(y) = y$. It follows that row u_1 cannot be eliminated and is in the minimal tableau. Similar arguments show that u_2 and u_3 cannot be eliminated. However, u_4 and u_5 can be eliminated using $\theta(y_2) = y_1, \theta(z_2) = z_1$ (and identity everywhere else). The preceding argument emphasizes the global nature of tableau minimization.

The preceding theorem suggests an improvement over the optimization strategies described in Section 6.1. Specifically, given a (satisfiable) conjunctive query q, the following steps can be used:

1. Translate q into a tableau query.
2. Minimize the number of rows in the tableau of this query.
3. Translate the result into a generalized SPC expression.
4. Apply the optimization techniques of Section 6.1.

As illustrated by Examples 6.2.2, 6.2.7, and 6.2.8, this approach has the advantage of performing global optimizations that typical query rewriting systems cannot achieve.

Example 6.2.8 Consider the relation schema R of sort ABC and the SPJR query q over R:

$$\pi_{AB}(\sigma_{B=5}(R)) \bowtie \pi_{BC}(\pi_{AB}(R) \bowtie \pi_{AC}(\sigma_{B=5}(R))).$$

R	A	B	C
	x	5	z_1
	x_1	5	z_2
	x_1	5	z
u	x	5	z

Figure 6.7: Tableau equivalent to q

The tableau (T, u) corresponding to it is that of Fig. 6.7. To minimize (T, u), we wish to find a homomorphism that "folds" T onto a subtableau with minimal number of rows. (If desired, this can be done in several stages, each of which eliminates one or more rows.) Note that the first row cannot be eliminated because every homomorphism is the identity on u and therefore on x. A similar observation holds for the third row. However, the second row can be eliminated using the homomorphism that maps z_2 to z and is the identity everywhere else. Thus the minimal tableau equivalent to (T, u) consists of the first and third rows of T. An SPJR query equivalent to the minimized tableau is

$$\pi_{AB}(\sigma_{B=5}(R)) \bowtie \pi_{BC}(\sigma_{B=5}(R)).$$

Thus the optimization procedure resulted in saving one join operation.

Before leaving minimal tableau queries, we present a result that describes a strong correspondence between equivalent minimal tableau queries. Two tableau queries $(\mathbf{T}, u)$, $(\mathbf{T}', u')$ are *isomorphic* if there is a one-one substitution θ that maps variables to variables such that $\theta((\mathbf{T}, u)) = (\mathbf{T}', u')$. In other words, (T, u) and (T', u') are the same up to renaming of variables. The proof of this result is left to the reader (see Exercise 6.11).

Proposition 6.2.9 Let $q = (\mathbf{T}, u)$ and $q' = (\mathbf{T}', u')$ be minimal and equivalent. Then q and q' are isomorphic.

Complexity of Tableau Decision Problems

The following theorem shows that determining containment and equivalence between tableau queries is NP-complete and tableau query minimization is NP-hard.

Theorem 6.2.10 The following problems, given tableau queries q, q', are NP-complete:

(a) Is $q \subseteq q'$?

(b) Is $q \equiv q'$?

(c) Suppose that the tableau of q is obtained by deleting free tuples of the tableau of q'. Is $q \equiv q'$ in this case?

These results remain true if q, q' are restricted to be single-relation typed tableau queries that have no constants.

Proof The proof is based on a reduction from the "exact cover" problem to the different tableau problems. The *exact cover* problem is to decide, given a set $X = \{x_1, \ldots, x_n\}$ and a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of subsets of X such that $\cup\mathcal{S} = X$, whether there is an exact cover of X by $\mathcal{S}$ (i.e., a subset $\mathcal{S}'$ of $\mathcal{S}$ such that each member of X occurs in exactly one member of $\mathcal{S}'$). The exact cover problem is known to be NP-complete.

We now sketch a polynomial transformation from instances $\mathcal{E} = (X, \mathcal{S})$ of the exact cover problem to pairs $q_\mathcal{E}, q'_\mathcal{E}$ of typed tableau queries. This construction is then applied in various ways to obtain the NP-completeness results. The construction is illustrated in Fig. 6.8.

Let $\mathcal{E} = (X, \mathcal{S})$ be an instance of the exact cover problem, where $X = \{x_1, \ldots, x_n\}$ and $\mathcal{S} = \{S_1, \ldots, S_m\}$. Let $A_1, \ldots, A_n, B_1, \ldots, B_m$ be a listing of distinct attributes, and let R be chosen to have this set as its sort. Both $q_\mathcal{E}$ and $q'_\mathcal{E}$ are over relation R, and both queries have as summary $t = \langle A_1 : a_1, \ldots, A_n : a_n\rangle$, where $a_1, \ldots, a_n$ are distinct variables.

Let $b_1, \ldots, b_m$ be an additional set of m distinct variables. The tableau $T_\mathcal{E}$ of $q_\mathcal{E}$ has n tuples, each corresponding to a different element of X. The tuple for x_i has a_i for attribute A_i; b_j for attribute B_j for each j such that $x_i \in S_j$; and a new, distinct variable for all other attributes.

Let $c_1, \ldots, c_m$ be an additional set of m distinct variables. The tableau $T'_\mathcal{E}$ of $q'_\mathcal{E}$ has m tuples, each corresponding to a different element of $\mathcal{S}$. The tuple for S_j has a_i for attribute A_i for each i such that $x_i \in S_j$; $c_{j'}$ for attribute $B_{j'}$ for each j' such that $j' \neq j$; and a new, distinct variable for all other attributes.

To illustrate the construction, let $\mathcal{E} = (X, \mathcal{S})$ be an instance of the exact cover problem, where $X = \{x_1, x_2, x_3, x_4\}$ and $\mathcal{S} = \{S_1, S_2, S_3\}$ where

$$\begin{aligned} S_1 &= \{x_1, x_3\} \\ S_2 &= \{x_2, x_3, x_4\} \\ S_3 &= \{x_2, x_4\}. \end{aligned}$$

The tableau queries q_ξ and q'_ξ corresponding to $(X, \mathcal{S})$ are shown in Fig. 6.8. (Here the blank entries indicate distinct, new variables.) Note that $\xi = (X, \mathcal{S})$ is satisfiable, and $q'_\xi \subseteq q_\xi$.

More generally, it is straightforward to verify that for a given instance $\xi = (X, \mathcal{S})$ of the exact cover problem, X has an exact cover in $\mathcal{S}$ iff $q'_\xi \subseteq q_\xi$. Verification of this, and of parts (b) and (c) of the theorem, is left for Exercise 6.16. ■

A subclass of the typed tableau queries for which containment and equivalence is decidable in polynomial time is considered in Exercise 6.21.

Although an NP-completeness result often suggests intractability, this conclusion may not be warranted in connection with the aforementioned result. The complexity there is measured relative to the size of the *query* rather than in terms of the underlying stored

R	A_1	A_2	A_3	A_4	B_1	B_2	B_3
	a_1				b_1		
		a_2				b_2	b_3
			a_3		b_1	b_2	
				a_4		b_2	b_3
	a_1	a_2	a_3	a_4			

q_ξ

(a)

R	A_1	A_2	A_3	A_4	B_1	B_2	B_3
	a_1		a_3			c_2	c_3
		a_2	a_3	a_4	c_1		c_3
		a_2		a_4	c_1	c_2	
	a_1	a_2	a_3	a_4			

q'_ξ

(b)

Figure 6.8: Tableau queries corresponding to an exact cover

data. Given an n-way join, the System R optimizer may potentially consider $n!$ evaluation strategies based on different orderings of the n relations; this may be exponential in the size of the query. In many cases, the search for a minimal tableau (or optimal left-to-right join) may be justified because the data is so much larger than the initial query. More generally, in Part D we shall examine both "data complexity" and "expression complexity," where the former focuses on complexity relative to the size of the data and the latter relative to the size of queries.

6.3 Static Analysis of the Relational Calculus

We now demonstrate that the decidability results for conjunctive queries demonstrated in the previous section do not hold when negation is incorporated (i.e., do not hold for the first-order queries). In particular, we present a general technique for proving the undecidability of problems involving static analysis of first-order queries and demonstrate the undecidability of three such problems.

We begin by focusing on the basic property of satisfiability. Recall that a query q is *satisfiable* if there is some input $\mathbf{I}$ such that $q(\mathbf{I})$ is nonempty. All conjunctive queries are satisfiable (Proposition 4.2.2), and if equality is incorporated then satisfiability is not guaranteed but it is decidable (Exercise 4.5). This no longer holds for the calculus.

To prove this result, we use a reduction of the Post Correspondence Problem (PCP) (see Chapter 2) to the satisfiability problem. The reduction is most easily described in terms of the calculus; of course, it can also be established using the algebras or nr-datalog$^\neg$.

At first glance, it would appear that the result follows trivially from the analogous result for first-order logic (i.e., the undecidability of satisfiability of first-order sentences). There is, however, an important difference. In conventional first-order logic (see Chapter 2), both finite and infinite interpretations are considered. Satisfiability of first-order sentences is co-recursively enumerable (co-r.e.) but not recursive. This follows from Gödel's Completeness Theorem. In contrast, in the context of first-order queries, only finite instances are considered legal. This brings us into the realm of finite model theory. As will be shown, satisfiability of first-order queries is recursively enumerable (r.e.) but not recursive. (We shall revisit the contrast between conventional first-order logic and the database perspective, i.e., finite model theory, in Chapters 9 and 10.)

Theorem 6.3.1 Satisfiability of relational calculus queries is r.e. but not recursive.

Proof To see that the problem is r.e., imagine a procedure that, when given query q over $\mathbf{R}$ as input, generates all instances $\mathbf{I}$ over $\mathbf{R}$ and tests $q(\mathbf{I}) = \emptyset$ until a nonempty answer is found.

To show that satisfiability is not recursive, we reduce the PCP to the satisfiability problem. In particular, we show that if there were an algorithm for solving satisfiability, then it could be used to construct an algorithm that solves the PCP.

Let $\mathcal{P} = (u_1, \ldots, u_n; v_1, \ldots, v_n)$ be an instance of the PCP (i.e., a pair of sequences of nonempty words over alphabet $\{0,1\}$). We describe now a (domain independent) calculus query $q_\mathcal{P} = \{\langle\rangle \mid \varphi_\mathcal{P}\}$ with the property that $q_\mathcal{P}$ is satisfiable iff $\mathcal{P}$ has a solution.

We shall use a relation schema $\mathbf{R}$ having relations *ENC(ODING)* with sort $[A, B, C, D, E]$ and *SYNCH(RONIZATION)* with sort $[F, G]$. The query $q_\mathcal{P}$ shall use constants $\{0, 1, \$, c_1, \ldots, c_n, d_1, \ldots, d_n\}$. (The use of multiple relations and constants is largely a convenience; the result can be demonstrated using a single ternary relation and no constants. See Exercise 6.19.)

To illustrate the construction of the algorithm, consider the following instance of the PCP:

$$u_1 = 011,\ u_2 = 011,\ u_3 = 0; \quad v_1 = 0,\ v_2 = 11,\ v_3 = 01100.$$

Note that $s = (1, 2, 3, 2)$ is a solution of this instance. That is,

$$u_1u_2u_3u_2 = 0110110011 = v_1v_2v_3v_2.$$

Figure 6.9 shows an input instance $\mathbf{I}_s$ over $\mathbf{R}$ which encodes this solution and satisfies the query $q_\mathcal{P}$ constructed shortly.

In the relation *ENC* of this figure, the first two columns form a *cycle*, so that the 10 tuples can be viewed as a sequence rather than a set. The third column holds a listing of the word $w = 0110110011$ that witnesses the solution to P; the fourth column describes which words of sequence $(u_1, \ldots, u_n)$ are used to obtain w; and the fifth column describes which words of sequence $(v_1, \ldots, v_n)$ are used. The relation *SYNCH* is used to synchronize the two representations of w by listing the pairs corresponding to the beginnings of new u-words and v-words.

The formula $\varphi_\mathcal{P}$ constructed now includes subformulas to test whether the various conditions just enumerated hold on an input instance. In particular,

$$\varphi = \varphi_{ENC\text{-}key} \wedge \varphi_{cycle} \wedge \ \varphi_{SYNCH\text{-}keys} \wedge \varphi_{u\text{-}encode} \wedge \varphi_{v\text{-}encode} \wedge \varphi_{u\text{-}v\text{-}synch},$$

where, speaking informally,

ENC	*A*	*B*	*C*	*D*	*E*
	\$	a_1	0	c_1	d_1
	a_1	a_2	1	c_1	d_2
	a_2	a_3	1	c_1	d_2
	a_3	a_4	0	c_2	d_3
	a_4	a_5	1	c_2	d_3
	a_5	a_6	1	c_2	d_3
	a_6	a_7	0	c_3	d_3
	a_7	a_8	0	c_2	d_3
	a_8	a_9	1	c_2	d_2
	a_9	\$	1	c_2	d_2

SYNCH	*F*	*G*
	\$	\$
	a_3	a_1
	a_6	a_3
	a_7	a_8

Figure 6.9: Encoding of a solution to PCP

$\varphi_{ENC\text{-}key}$: states that the first column of *ENC* is a *key*; that is, each value occurring in the *A* column occurs in exactly one tuple of *ENC*.

φ_{cycle}: states that constant \$ occurs in a cycle with length > 1 in the first two columns of *ENC*. (There may be other cycles, which can be ignored.)

$\varphi_{SYNCH\text{-}keys}$: states that both the first and second columns of *SYNCH* are keys.

$\varphi_{u\text{-}encode}$: states that for each value x occurring in the first column of *SYNCH*, if tuple $\langle x_1, y_1, z_1, c_i, d_{j_1}\rangle$ is in *ENC*, then there are at least $|u_i| - 1$ additional tuples in *ENC* "after" this tuple, all with value c_i in the fourth coordinate, and if these tuples are

$$\langle x_2, y_2, z_2, c_i, d_{j_2}\rangle, \ldots, \langle x_k, y_k, z_k, c_i, d_{j_k}\rangle$$

then $z_1 \ldots z_k = u_i$; none of $x_2, \ldots, x_k$ occurs in the first column of *SYNCH*; and if $y_k \neq \$$, then the *A* value "after" x_k occurs in the first column of *SYNCH*.

$\varphi_{v\text{-}encode}$: is analogous to $\varphi_{u\text{-}encode}$.

$\varphi_{u\text{-}v\text{-}synch}$: states that (1) $\langle \$, \$\rangle$ is in *SYNCH*; (2) if a tuple $\langle x, y\rangle$ is in *SYNCH*, then the associated u-word and v-word have the same index; and (3) if a tuple $\langle x, y\rangle$ is in *SYNCH*, and either x or y are not the "maximum" *A* value occurring in *F* or *G*, then there exists a tuple $\langle x', y'\rangle$ in *SYNCH*, where x' is the first *A* value "after" x occurring in *F* and y' is the first *A* value "after" y occurring in *G*. Finding the *A* values "after" x and y is done as in $\varphi_{u\text{-}encode}$.

The constructions of these formulas are relatively straightforward; we give two of them here and leave the others for the reader (see Exercise 6.19). In particular, we let

$$\psi(x, y) = \exists p, q, r\ ENC(x, y, p, q, r)$$

and set

$$\begin{aligned}\varphi_{cycle} = {} & \exists x(\psi(x, \$) \wedge \neg(x = \$)) \wedge \exists y(\psi(\$, y) \wedge \neg(y = \$)) \wedge \\ & \forall x((\exists y \psi(x, y)) \rightarrow (\exists z \psi(z, x))) \wedge \\ & \forall x((\exists y \psi(y, x)) \rightarrow (\exists z \psi(x, z))) \wedge \\ & \forall x, y_1, y_2(\psi(y_1, x) \wedge \psi(y_2, x) \rightarrow y_1 = y_2).\end{aligned}$$

If *ENC* satisfies $\varphi_{ENC-key} \wedge \varphi_{cycle}$, then the first two coordinates of *ENC* hold one or more disjoint cycles, exactly one of which contains the value \$.

Parts (1) and (2) of $\varphi_{u\text{-}v\text{-}synch}$ are realized by the formula

$$\begin{aligned}& SYNCH(\$, \$) \wedge \\ & \forall x, y(SYNCH(x, y) \rightarrow \\ & \quad \exists s, p, r, t, p', q((ENC(x, s, p, c_1, r) \wedge ENC(y, t, p', q, d_1)) \vee \\ & \qquad (ENC(x, s, p, c_2, r) \wedge ENC(y, t, p', q, d_2)) \vee \\ & \qquad \vdots \\ & \qquad (ENC(x, s, p, c_n, r) \wedge ENC(y, t, p', q, d_n)))).\end{aligned}$$

Verifying that the query $q_{\mathcal{P}}$ is satisfiable if and only if P has a solution is left to the reader (see Exercise 6.19). ■

The preceding theorem can be applied to derive other important undecidability results.

Corollary 6.3.2

(a) Equivalence and containment of relational calculus queries are co-r.e. and not recursive.

(b) Domain independence of a relational calculus query is co-r.e. and not recursive.

Proof It is easily verified that the two problems of part (a) and the problem of part (b) are co-r.e. (see Exercise 6.20). The proofs of undecidability are by reduction from the satisfiability problem. For equivalence, suppose that there were an algorithm for deciding equivalence between relational calculus queries. Then the satisfiability problem can be solved as follows: For each query $q = \{x_1, \ldots, x_n \mid \varphi\}$, this is unsatisfiable if and only if it is equivalent to the empty query $q^{\emptyset}$. This demonstrates that equivalence is not decidable. The undecidability of containment also follows from this.

For domain independence, let ψ be a sentence whose truth value depends on the underlying domain. Then $\{x_1, \ldots, x_n \mid \varphi \wedge \psi\}$ is domain independent if and only if φ is unsatisfiable. ■

The preceding techniques can also be used to show that "true" optimization cannot be performed for the first-order queries (see Exercise 6.20d).

6.4 Computing with Acyclic Joins

We now present a family of interesting theoretical results on the problem of computing the projection of a join. In the general case, if both the data set and the join expression are allowed to vary, then the time needed to evaluate such expressions appears to be exponential. The measure of complexity here is a combination of both "data" and "expression" complexity, and is somewhat non-standard; see Part D. Interestingly, there is a special class of joins, called *acyclic*, for which this evaluation is polynomial. A number of interesting properties of acyclic joins are also presented.

For this section we use the named perspective and focus exclusively on *flat project-join* queries of the form

$$q = \pi_X(R_1 \bowtie \cdots \bowtie R_n)$$

involving projection and natural join. For this discussion we assume that $\mathbf{R} = R_1, \ldots, R_n$ is a fixed database schema, and we use $\mathbf{I} = (I_1, \ldots, I_n)$ to refer to instances over it.

One of the historical motivations for studying this problem stems from the *pure universal relation assumption* (*pure URA*). An instance $\mathbf{I} = (I_1, \ldots, I_n)$ over schema $\mathbf{R}$ satisfies the pure URA if $\mathbf{I} = (\pi_{R_1}(I), \ldots, \pi_{R_n}(I))$ for some "universal" instance I over $\cup_{j=1}^{n} R_j$. If $\mathbf{I}$ satisfies the pure URA, then $\mathbf{I}$ can be stored, and queries against the corresponding instance I can be answered using joins of components in $\mathbf{I}$. The URA will be considered in more depth in Chapter 11.

Worst-Case Results

We begin with an example.

Example 6.4.1 Let $n > 0$ and consider the relations $R_i[A_iA_{i+1}]$, $i \in [1, n-1]$, as shown in Fig. 6.10(a). It is easily seen that the natural join of $R_1, \ldots, R_{n-1}$ is exponential in n and thus exponential in the size of the input query and data.

Now suppose that n is odd. Let R_n be as in Fig. 6.10(b), and consider the natural join of $R_1, \ldots, R_n$. This is empty. On the other hand, the join of any i of these for $i < n$ has size exponential in i. It follows that the algorithms of the System R and INGRES optimizers take time exponential in the size of the input and output to evaluate this query.

The following result implies that it is unlikely that there is an algorithm for computing projections of joins in time polynomial in the size of the query and the data.

Theorem 6.4.2 It is NP-complete to decide, given project-join expression q_0 over $\mathbf{R}$, instance $\mathbf{I}$ of $\mathbf{R}$, and tuple t, whether $t \in q_0(\mathbf{I})$. This remains true if q_0 and $\mathbf{I}$ are restricted so that $|q_0(\mathbf{I})| \leq 1$.

Proof The problem is easily seen to be in NP. For the converse, recall from Theorem 6.2.10(a) that the problem of tableau containment is NP-complete, even for single-

R_i	A_i	A_{i+1}
	0	a
	0	b
	1	a
	1	b
	a	0
	a	1
	b	0
	b	1

(a)

R_n	A_n	A_1
	0	a
	0	b
	1	a
	1	b
	a	0
	a	1
	b	0
	b	1

(b)

Figure 6.10: Relations to illustrate join sizes

relation typed tableaux having no constants. We reduce this to the current problem. Let $q = (T, u)$ and $q' = (T', u')$ be two typed constant-free tableau queries over the same relation schema. Recall from the Homomorphism Theorem that $q \subseteq q'$ iff there is a homomorphism of q' to q, which holds iff $u \in q'(T)$.

Assume that the sets of variables occurring in q and in q' are disjoint. Without loss of generality, we view each variable occurring in q to be a constant. For each variable x occurring in q', let A_x be a distinct attribute. For free tuple $v = (x_1, \ldots, x_n)$ in T', let I_v over $A_{x_1}, \ldots, A_{x_n}$ be a copy of T, where the i^{th} attribute is renamed to A_{x_i}. Letting $u' = \langle u'_1, \ldots, u'_m \rangle$, it is straightforward to verify that

$$q'(T) = \pi_{A_{u'_1},\ldots,A_{u'_m}}(\bowtie\{I_v \mid v \in T'\}).$$

In particular, $u \in q'(T)$ iff u is in this projected join.

To see the last sentence of the theorem, let $u = \langle u_1, \ldots, u_m \rangle$ and use the query

$$\pi_{A_{u'_1},\ldots,A_{u'_m}}(\bowtie\{I_v \mid v \in T'\} \bowtie \{\langle A_{u'_1} : u_1, \ldots, A_{u'_m} : u_m \rangle\}). \blacksquare$$

Theorem 6.2.10(a) considers complexity relative to the size of queries. As applied in the foregoing result, however, the queries of Theorem 6.2.10(a) form the basis for constructing a database instance $\{I_v \mid v \in T'\}$. In contrast with the earlier theorem, the preceding result suggests that computing projections of joins is intractable relative to the size of the query, the stored data, and the output.

Acyclic Joins

In Example 6.4.1, we may ask what is the fundamental difference between $R_1 \bowtie \cdots \bowtie R_{n-1}$ and $R_1 \bowtie \cdots \bowtie R_n$? One answer is that the relation schemas of the latter join form a cycle, whereas the relation schemas of the former do not.

We now develop a formal notion of acyclicity for joins and four properties equivalent to it. All of these are expressed most naturally in the context of the named perspective for the relational model. In addition, the notion of acyclicity is sometimes applied to database schemas $\mathbf{R}=\{R_1, \ldots, R_n\}$ because of the natural correspondence between the schema $\mathbf{R}$ and the join $R_1 \bowtie \cdots \bowtie R_n$.

We begin by describing four interesting properties that are equivalent to acyclicity. Let $\mathbf{R} = \{R_1, \ldots, R_n\}$ be a database schema, where each relation schema has a different sort. An instance $\mathbf{I}$ of $\mathbf{R}$ is said to be *pairwise consistent* if for each pair $j, k \in [1, n]$, $\pi_{R_j}(I_j \bowtie I_k) = I_j$. Intuitively, this means that no tuple of I_j is "dangling" or "lost" after joining with I_k. Instance $\mathbf{I}$ is *globally consistent* if for each $j \in [1, n]$, $\pi_{R_j}(\bowtie\mathbf{I}) = I_j$ (i.e., no tuple of I_j is dangling relative to the full join). Pairwise consistency can be checked in PTIME, but checking global consistency is NP-complete (Exercise 6.25). The first property that is equivalent to acyclicity is:

Property (1): Each instance $\mathbf{I}$ that is pairwise consistent is globally consistent.

Note that the instance for schema $\{R_1, \ldots, R_{n-1}\}$ of Example 6.4.1 is both pairwise and globally consistent, whereas the instance for $\{R_1, \ldots, R_n\}$ is pairwise but not globally consistent.

The second property we consider is motivated by query processing in a distributed environment. Suppose that each relation of $\mathbf{I}$ is stored at a different site, that the join $\bowtie\mathbf{I}$ is to be computed, and that communication costs are to be minimized. A very naive algorithm to compute the join is to send each of the I_j to a specific site and then form the join. In the general case this may cause the shipment of many unneeded tuples because they are dangling in the full join.

The *semi-join* operator can be used to alleviate this problem. Given instances I, J over R, S, then semi-join of I and J is

$$I \ltimes J = \pi_R(I \bowtie J).$$

It is easily verified that $I \bowtie J = (I \ltimes J) \bowtie J = (J \ltimes I) \bowtie I$. Furthermore there are many cases in which computing the join in one of these ways can reduce data transmission costs if I and J are at different nodes of a distributed database (see Exercise 6.24).

Suppose now that $\mathbf{R}$ satisfies Property (1). Given an instance $\mathbf{I}$ distributed across the network, one can imagine replacing each relation I_j by its semi-join with other relations of $\mathbf{I}$. If done cleverly, this might be done with communication cost polynomial in the size of $\mathbf{I}$, with the result of the replacements satisfying pairwise consistency. Given Property (1), all relations can now be shipped to a common site, safe in the knowledge that no dangling tuples have been shipped.

More generally, a *semi-join program* for $\mathbf{R}$ is a sequence of commands

$$\begin{aligned}
R_{i_1} &:= R_{i_1} \ltimes R_{j_1};\\
R_{i_2} &:= R_{i_2} \ltimes R_{j_2};\\
&\vdots\\
R_{i_p} &:= R_{i_p} \ltimes R_{j_p};
\end{aligned}$$

R_1	A	B	C
	0	3	2
	0	1	2
	3	1	2
	1	1	3

R_2	B	C	D	E
	3	2	1	0
	1	2	3	0
	1	3	1	0

R_3	B	C	D	G
	3	2	1	4
	1	2	3	2
	1	3	1	0
	1	3	1	1

R_4	C	D	E	F
	2	1	1	4
	2	3	0	1
	3	1	0	2
	3	1	0	3

Figure 6.11: Instance for Example 6.4.3

(In practice, the original values of R_{i_j} would not be overwritten; rather, a scratch copy would be made.) This is a *full reducer* for **R** if for each instance **I** over **R**, applying this program yields an instance **I**′ that is globally consistent.

Example 6.4.3 Let $\mathbf{R} = \{ABC, BCDE, BCDG, CDEF\} = \{R_1, R_2, R_3, R_4\}$ and consider the instance **I** of **R** shown in Fig. 6.11. **I** is not globally consistent; nor is it pairwise consistent.

A full reducer for this schema is

$$
\begin{aligned}
R_2 &:= R_2 \ltimes R_1;\\
R_2 &:= R_2 \ltimes R_4;\\
R_3 &:= R_3 \ltimes R_2;\\
R_2 &:= R_2 \ltimes R_3;\\
R_4 &:= R_4 \ltimes R_2;\\
R_1 &:= R_1 \ltimes R_2;
\end{aligned}
$$

Note that application of this program to **I** has the effect of removing the first tuple from each relation.

We can now state the second property:

Property (2): **R** has a full reducer.

It can be shown that the schema $\{R_1, \ldots, R_{n-1}\}$ of Example 6.4.1 has a full reducer, but $\{R_1, \ldots, R_n\}$ does not (see Exercise 6.26).

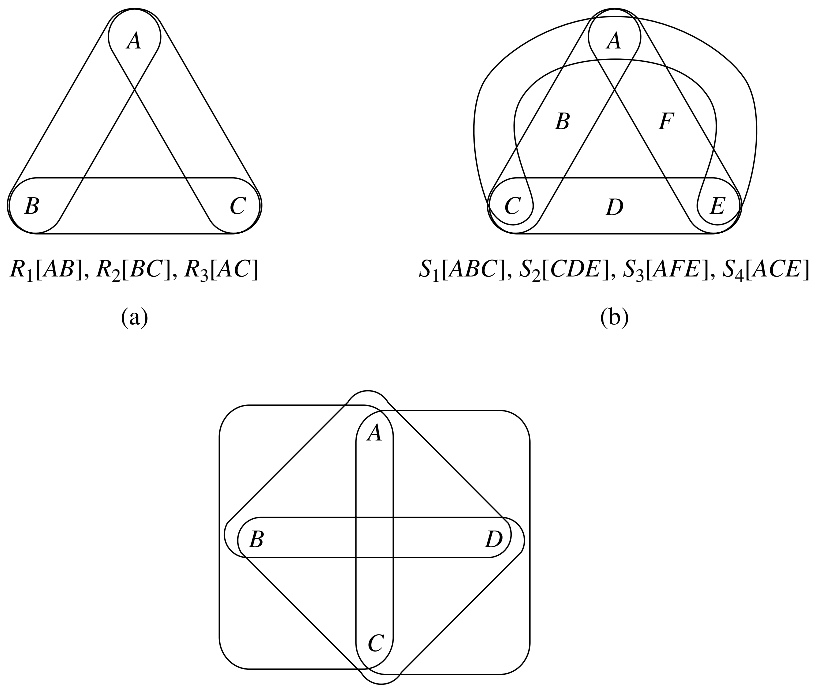
The next property provides a way to view a schema as a tree with certain properties. A *join tree* of a schema **R** is an undirected tree $T = (\mathbf{R}, E)$ such that

(i) each edge (R, R') is labeled by the set of attributes $R \cap R'$; and

(ii) for every pair R, R' of distinct nodes, for each $A \in R \cap R'$, each edge along the unique path between R and R' includes label A.

Property (3): **R** has a join tree.

For example, two join trees of the schema **R** of Figure 6.11 are $T_1 = (\mathbf{R}, \{(R_1, R_2), (R_2, R_3), (R_2, R_4)\})$ and $T_2 = (\mathbf{R}, \{(R_1, R_3), (R_3, R_2), (R_2, R_4)\})$. (The edge labels are not shown.)



$R_1[AB]$, $R_2[BC]$, $R_3[AC]$

(a)

$S_1[ABC]$, $S_2[CDE]$, $S_3[AFE]$, $S_4[ACE]$

(b)

$T_1[ABC]$, $T_2[BCD]$, $T_3[ABD]$, $T_4[ACD]$

(c)

Figure 6.12: Three schemas and their hypergraphs

The fourth property we consider focuses entirely on the database schema **R** and is based on a simple algorithm, called the *GYO algorithm*.[1] This is most easily described in terms of the hypergraph corresponding to **R**. A *hypergraph* is a pair $\mathcal{F} = (V, F)$, where V is a set of vertexes and F is family of distinct nonempty subsets of V, called *edges* (or *hyperedges*). The *hypergraph* of schema **R** is the pair $(U, \mathbf{R})$, where $U = \cup\mathbf{R}$. In what follows, we often refer to a database schema **R** as a hypergraph. Three schemas and their hypergraphs are shown in Fig. 6.12.

A hypergraph is *reduced* if there is no pair f, f' of distinct edges with f a proper subset of f'. The *reduction* of $\mathcal{F} = (V, F)$ is $(V, F - \{f \in F \mid \exists f' \in F \text{ with } f \subset f'\})$. Suppose that **R** is a schema and **I** over **R** satisfies the pure URA. If $R_j \subset R_k$, then $I_j =$

[1] This is so named in honor of M. Graham and the team C. T. Yu and M. Z. Ozsoyoglu, who independently came to essentially this algorithm.

$\pi_{R_j}(I_k)$, and thus I_j holds redundant information. It is thus natural in this context to assume that **R**, viewed as a hypergraph, is reduced.

An *ear* of hypergraph $\mathcal{F} = (V, F)$ is an edge $f \in F$ such that for some distinct $f' \in F$, no vertex of $f - f'$ is in any other edge or, equivalently, such that $f \cap (\cup(F - \{f\})) \subseteq f'$. In this case, f' is called a *witness* that f is an ear. As a special case, if there is an edge f of $\mathcal{F}$ that intersects no other edge, then f is also considered an ear.

For example, in the hypergraph of Fig. 6.12(b), edge ABC is an ear, with witness ACE. On the other hand, the hypergraph of Fig. 6.12(a) has no ears.

We now have

ALGORITHM 6.4.4 (GYO Algorithm)

Input: Hypergraph $\mathcal{F} = (V, F)$

Output: A hypergraph involving a subset of edges of $\mathcal{F}$

```
Do until $\mathcal{F}$ has no ears:
  1. Nondeterministically choose an ear $f$ of $\mathcal{F}$.
  2. Set $\mathcal{F} := (V', F - \{f\})$, where $V' = \cup(F - \{f\})$. ■
```

The output of the GYO algorithm is always reduced.

A hypergraph is *empty* if it is $(\emptyset, \emptyset)$. In Fig. 6.12, it is easily verified that the output of the GYO algorithm is empty for part (b), but that parts (a) and (c) have no ears and so equal their output under the algorithm. The output of the GYO algorithm is independent of the order of steps taken (see Exercise 6.28).

We now state the following:

Property (4): The output of the GYO algorithm on **R** is empty.

Speaking informally, Example 6.4.1 suggests that an absence of cycles yields Properties (1) to (4), whereas the presence of a cycle makes these properties fail. This led researchers in the late 1970s to search for a notion of acyclicity for hypergraphs that both generalized the usual notion of acyclicity for conventional undirected graphs and was equivalent to one or more of the aforementioned properties. For example, the conventional notion of hypergraph acyclicity from graph theory is due to C. Berge; but it turns out that this condition is necessary but not sufficient for the four properties (see Exercise 6.32).

We now define the notion of acyclicity that was found to be equivalent to the four aforementioned properties. Let $\mathcal{F} = (V, F)$ be a hypergraph. A *path* in $\mathcal{F}$ from vertex v to vertex v' is a sequence of $k \geq 1$ edges $f_1, \ldots, f_k$ such that

(i) $v \in f_1$;

(ii) $v' \in f_k$;

(iii) $f_i \cap f_{i+1} \neq \emptyset$ for $i \in [1, k-1]$.

Two vertexes are *connected* in $\mathcal{F}$ if there is a path between them. The notions of *connected* pair of edges, *connected component*, and *connected hypergraph* are now defined in the usual manner.

Now let $\mathcal{F} = (V, F)$ be a hypergraph, and $U \subseteq V$. The *restriction* of $\mathcal{F}$ to U, denoted $\mathcal{F}|_U$, is the result of forming the reduction of $(U, \{f \cap U \mid f \in F\} - \{\emptyset\})$.

Let $\mathcal{F} = (V, F)$ be a reduced hypergraph, let f, f' be distinct edges, and let $g = f \cap f'$. Then g is an *articulation set* of $\mathcal{F}$ if the number of connected components of $\mathcal{F}|_{V-g}$ is greater than the number of connected components of $\mathcal{F}$. (This generalizes the notion of articulation point for ordinary graphs.)

Finally, a reduced hypergraph $\mathcal{F} = (V, F)$ is *acyclic* if for each $U \subseteq V$, if $\mathcal{F}|_U$ is connected and has more than one edge then it has an articulation set; it is *cyclic* otherwise. A hypergraph is *acyclic* if its reduction is.

Note that if $\mathcal{F} = (V, F)$ is an acyclic hypergraph, then so is $\mathcal{F}|_U$ for each $U \subseteq V$.

Property (5): The hypergraph corresponding to **R** is acyclic.

We now present the theorem stating the equivalence of these five properties. Additional equivalent properties are presented in Exercise 6.31 and in Chapter 8, where the relationship of acyclicity with dependencies is explored.

Theorem 6.4.5 Properties (1) through (5) are equivalent.

Proof We sketch here arguments that (4) $\Rightarrow$ (2) $\Rightarrow$ (1) $\Rightarrow$ (5) $\Rightarrow$ (4). The equivalence of (3) and (4) is left as Exercise 6.30(a).

We assume in this proof that the hypergraphs considered are connected; generalization to the disconnected case is straightforward.

(4) $\Rightarrow$ (2): Suppose now that the output of the GYO algorithm on $\mathbf{R} = \{R_1, \ldots, R_n\}$ is empty. Let $S_1, \ldots, S_n$ be an ordering of **R** corresponding to a sequence of ear removals stemming from an execution of the GYO algorithm, and let T_i be a witness for S_i for $i \in [1, n-1]$. An induction on n ("from the inside out") shows that the following is a full reducer (see Exercise 6.30a):

$$\begin{aligned} T_1 &:= T_1 \ltimes S_1; \\ T_2 &:= T_2 \ltimes S_2; \\ &\vdots \\ T_{n-1} &:= T_{n-1} \ltimes S_{n-1}; \\ S_{n-1} &:= S_{n-1} \ltimes T_{n-1}; \\ &\vdots \\ S_2 &:= S_2 \ltimes T_2; \\ S_1 &:= S_1 \ltimes T_1; \end{aligned}$$

(2) $\Rightarrow$ (1): Suppose that **R** has a full reducer, and let **I** be a pairwise consistent instance of **R**. Application of the full reducer to **I** yields an instance **I**$'$ that is globally consistent. But by pairwise consistency, each step of the full reducer leaves **I** unchanged. It follows that **I** = **I**$'$ is globally consistent.

(1) $\Rightarrow$ (5): This is proved by contradiction. Suppose that there is a hypergraph that satisfies Property (1) but violates the definition of *acyclic*. Let $\mathbf{R} = \{R_1, \ldots, R_n\}$ be such a hypergraph where n is minimal among such hypergraphs and where the size of $U = \cup\mathbf{R}$ is minimal among such hypergraphs with n edges.

I	A_1	A_2	$\dots$	A_p	B_1	$\dots$	B_q
	1	0	$\dots$	0	1	$\dots$	1
	0	1	$\dots$	0	2	$\dots$	2
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	0	0	$\dots$	1	p	$\dots$	p

Figure 6.13: Instance for proof of Theorem 6.4.5

It follows easily from the minimality conditions that $\mathbf{R}$ is reduced. In addition, by minimality no vertex (attribute) in U is in only one edge (relation schema).

Consider now the schema $\mathbf{R}' = \{R_2 - R_1, \dots, R_n - R_1\}$. Two cases arise:

Case 1: $\mathbf{R}'$ is connected. Suppose that $R_1 = \{A_1, \dots, A_p\}$ and $U - R_1 = \{B_1 \dots, B_q\}$. Consider the instance I over U shown in Fig. 6.13. Define $\mathbf{I} = \{I_1, \dots, I_n\}$ so that

$$I_j = \pi_{R_j}(I) \text{ for } j \in [2, n], \text{ and}$$
$$I_1 = \pi_{R_1}(I) \cup \{\langle 0, 0, \dots, 0\rangle\}.$$

Using the facts that $\mathbf{R}'$ is connected and that each vertex of $\mathbf{R}$ occurs in at least two edges, it is straightforward to verify that $\mathbf{I}$ is pairwise consistent but not globally consistent, which is a contradiction (see Exercise 6.30b).

Case 2: $\mathbf{R}'$ is not connected. Choose a connected component of $\mathbf{R}'$ and let $\{S_1, \dots, S_k\}$ be the set of edges of $\mathbf{R} - \{R_1\}$ involved in that connected component. Let $S = \cup_{i=1}^{k} S_i$ and let $R_1' = R_1 \cap S$. Two subcases arise:

Subcase 2.a: $R_1' \subseteq S_j$ for some $j \in [1, k]$. If this holds, then $R_1' \cap S_j$ is an articulation set for $\mathbf{R}$, which is a contradiction (see Exercise 6.30b).

Subcase 2.b: $R_1' \not\subseteq S_j$ for each $j \in [1, k]$. In this case $\mathbf{R}'' = \{S_1, \dots, S_k, R_1'\}$ is a reduced hypergraph with fewer edges than $\mathbf{R}$. In addition, it can be verified that this hypergraph satisfies Property (1) (see Exercise 6.30b). By minimality of n, this implies that $\mathbf{R}''$ is acyclic. Because it is connected and has at least two edges, it has an articulation set. Two nested subcases arise:

Subcase 2.b.i: $S_i \cap S_j$ is an articulation pair for some i, j. We argue in this case that $S_i \cap S_j$ is an articulation pair for $\mathbf{R}$. To see this, let $x \in R_1' - (S_i \cap S_j)$ and let y be a vertex in some other component of $\mathbf{R}''|_{S-\{S_i \cap S_j\}}$. Suppose that $R_{i_1}, \dots, R_{i_l}$ is a path in $\mathbf{R}$ from y to x. Let R_{i_p} be the first edge in this path that is not in $\{S_1, \dots, S_k\}$. By the choice of $\{S_1, \dots, S_k\}$, $R_{i_p} = R_1$. It follows that there is a path from y to x in $\mathbf{R}''|_{S-\{S_i \cap S_j\}}$, which is a contradiction. We conclude that $\mathbf{R}$ has an articulation pair, contradicting the initial assumption in this proof.

Subcase 2.b.ii: $R_1' \cap S_i$ is an articulation pair for some i. In this case $R_1 \cap S_i$ is an articulation pair for $\mathbf{R}$ (see Exercise 6.30b), again yielding a contradiction to the initial assumption of the proof.

(5) ⇒ (4): We first show inductively that each connected reduced acyclic hypergraph $\mathcal{F}$ with at least two edges has at least two ears. For the case in which $\mathcal{F}$ has two edges, this result is immediate. Suppose now that $\mathcal{F} = (V, F)$ is connected, reduced, and acyclic, with $|F| > 2$. Let $h = f \cap f'$ be an articulation set of $\mathcal{F}$. Let $\mathcal{G}$ be a connected component of $\mathcal{F}|_{V-h}$. By the inductive hypothesis, this has at least two ears. Let g be an ear of $\mathcal{G}$ that is different from $f - h$ and different from $f' - h$. Let g' be an edge of $\mathcal{F}$ such that $g = g' - h$. It is easily verified that g' is an ear of $\mathcal{F}$ (see Exercise 6.30b). Because $\mathcal{F}|_{V-h}$ has more than two connected components, it follows that $\mathcal{F}$ has at least two ears.

Finally, suppose that $\mathcal{F} = (V, F)$ is acyclic. If there is only one edge, then the GYO algorithm yields the empty hypergraph. Suppose that it has more than one edge. If $\mathcal{F}$ is not reduced, the GYO algorithm can be applied to reduce it. If $\mathcal{F}$ is reduced, then by the preceding argument $\mathcal{F}$ has an ear, say f. Then a step of the algorithm can be applied to yield $\mathcal{F}|_{\cup(\mathcal{F}-\{f\})}$. This is again acyclic. An easy induction now yields the result. ■

Recall from Theorem 6.4.2 that computing projections of arbitrary joins is probably intractable if both query and data size are considered. The following shows that this is not the case when the join is acyclic.

Corollary 6.4.6 If **R** is acyclic, then for each instance **I** over **R**, the expression $\pi_X(\bowtie \mathbf{I})$ can be computed in time polynomial in the size of **IR**, the input, and the output.

Proof Because the computation for each connected component of **R** can be performed separately, we assume without loss of generality that **R** is connected. Let $\mathbf{R} = (R_1, \ldots, R_n)$ and $\mathbf{I} = (I_1, \ldots, I_n)$. First apply a full reducer to **I** to obtain $\mathbf{I}' = (I'_1, \ldots, I'_n)$. This takes time polynomial in the size of the query and the input; the result is globally consistent; and $\bowtie \mathbf{II} = \bowtie \mathbf{II}'$.

Because **R** is acyclic, by Theorem 6.4.5 there is a join tree T for **R**. Choose a root for T, say R_1. For each subtree T_k of T with root $R_k \neq R_1$, let $X_k = X \cap (\cup\{R \mid R \in T_k\})$, and $Z_k = R_k \cap$ (the parent of R_k). Let $J_k = I'_k$ for $k \in [1, n]$. Inductively remove nodes R_k and replace instances J_k from leaf to root of T as follows: Delete node R_k with parent R_m by replacing J_m with $J_m \bowtie \pi_{X_k Z_k} J_k$. A straightforward induction shows that immediately before nonleaf node R_k is deleted, then $J_k = \pi_{X_k R_k}(\bowtie_{R_l \in T_k} I'_l)$. It follows that at the end of this process the answer is $\pi_X J_1$ and that at each intermediate stage each instance J_k has size bounded by $|I'_k| \cdot |\pi_X(\bowtie \mathbf{II}_k)|$ (see Exercise 6.33). ■

Bibliographic Notes

An extensive discussion of issues in query optimization is presented in [Gra93]. Other references include [JK84a, KS91, Ull89b]. Query optimization for distributed databases is surveyed in [YC84]. Algorithms for binary joins are surveyed in [ME92].

The paper [SAC$^+$79] describes query optimization in System/R, including a discussion of generating and analyzing multiple evaluation plans and a thorough discussion of accessing tuples from a single relation, as from a projection and selection. System/R is the precursor of IBM's DB2 database management system. The optimizer for INGRES introduces query decomposition, including both join detachment and tuple substitution [WY76, SWKH76].

The use of semi-joins in query optimization was first introduced in INGRES [WY76, SWKH76] and used for distributed databases in [BC81, BG81]. Research on optimizing buffer management policies includes [FNS91, INSS92, NCS91, Sto81]. Other system optimizers include those for Exodus [GD87], distributed INGRES [ESW78], SDD-1 [BGW$^+$81], and the TI Open Object-Oriented Data Base [BMG93].

[B$^+$88] presents the unifying perspective that physical query implementation can be viewed as generation, manipulation, and merging of streams of tuples and develops a very flexible toolkit for constructing dbms's. A formal model incorporating streams, sets, and parallelism is presented in [PSV92].

The recent work [IK90] focuses on finding optimal and near-optimal evaluation plans for n-way joins, where n is in the hundreds, using simulated annealing and other techniques. Perhaps most interesting about this work are characterizations of the space of evaluation plans (e.g., properties of evaluation plan cost in relation to natural metrics on this space).

Early research on generation and selection of query evaluation plans is found in [SAC$^+$79, SWKH76]. Treatments that separate plan generation from transformation rules include [Fre87, GD87, Loh88]. More recent research has proposed mechanisms for generating *parameterized* evaluation plans; these can be generated at compile time but permit the incorporation of run-time information [GW89, INSS92]. An extensive listing of references to the literature on estimating the costs of evaluation plans is presented in [SLRD93]. This article introduces an estimation technique based on the computation of series functions that approximates the distribution of values and uses regression analysis to estimate the output sizes of select-join queries.

Many forward-chaining expert systems in AI also face the problem of evaluating what amounts to conjunctive queries. The most common technique for evaluating conjunctive queries in this context is based on a sequential generate-and-test algorithm. The paper [SG85] presents algorithms that yield optimal and near-optimal orderings under this approach to evaluation.

The technique of tableau query minimization was first developed in connection with database queries in [CM77], including the Homomorphism Theorem (Theorem 6.2.3) and Theorem 6.2.6. Theorem 6.2.10 is also due to [CM77]; the proofs sketched in the exercises are due to [SY80] and [ASU79b]. Refinements of this result (e.g., to subclasses of typed tableau queries) are presented in [ASU79b, ASU79a].

The notion of tableau homomorphism is a special case of the notion of *subsumption* used in resolution theorem proving [CL73]. That work focuses on clauses (i.e., disjunctions of positive and negative literals), and permits function symbols. A clause $C = (L_1 \vee \cdots \vee L_n)$ *subsumes* a clause $D = (M_1 \vee \cdots \vee M_k)$ if there is a substitution σ such that $C\sigma$ is a subclause of D. A generalized version of tableau minimization, called *condensation*, also arises in this connection. A condensation of a clause $C = (L_1 \vee \cdots \vee L_n)$ is a clause $C' = (L_{i_1} \vee \cdots \vee L_{i_m})$ with m minimal such that $C' = C\theta$ for some substitution θ. As observed in [Joy76], condensations are unique up to variable substitution.

Reference [SY80] studies restricted usage of difference with SPCU queries, for which several positive results can be obtained (e.g., decidability of containment; see Exercise 6.22).

The undecidability results for the relational calculus derive from results in [DiP69] (see also [Var81]). The assumption in this chapter that relations be finite is essential. For instance, the test for containment is co-r.e. in our context whereas it is r.e. when possibly infinite structures are considered. (This is by reduction to the validity of a formula in first-order predicate logic with equality using the Gödel Completeness Theorem.

The complexity of query languages is studied in [CH82, Var82a] and is considered in Part E of this volume.

As discussed in Chapter 7, practical query languages typically produce *bags* (also called multisets; i.e., collections whose members may occur more than once). The problem of containment and equivalence of conjunctive queries under the bag semantics is considered in [CV93]. It remains open whether containment is decidable, but it is Π_2^p-hard. On the other hand, two conjunctive queries are equivalent under the bag semantics iff they are isomorphic.

Acyclic joins enjoyed a flurry of activity in the database research community in the late 1970s and early 1980s. As noted in [Mal86], the same concept has been studied in the field of statistics, beginning with [Goo70, Hab70]. An early motivation for their study in databases stemmed from distributed query processing; the notions of join tree and full reducers are from [BC81, BG81]; see also [GS82, GS84, SS86]. The original GYO algorithm was developed in [YO79] and [Gra79]; we use here a variant due to [FMU82]. The notion of globally consistent is studied in [BR80, HLY80, Ris82, Var82b]; see also [Hul83]. Example 6.4.1 is taken from [Ull89b]. The paper [BFM$^+$81] introduced the notion of acyclicity presented here and observed the equivalence to acyclicity of several previously studied properties, including those of having a full reducer and pairwise consistency implying global consistency; this work is reported in journal form in [BFMY83].

A linear-time test for acyclicity is developed in [TY84]. Theorem 6.4.2 and Corollary 6.4.6 are due to [Yan81].

The notion of Berge acyclic is due to [Ber76a]. [Fag83] investigates several notions of acyclicity, including the notion studied in this chapter and Berge acyclicity. Further investigation of these alternative notions of acyclicity is presented in [ADM85, DM86b, GR86]. Early attempts to develop a notion of acyclic that captured desirable database characteristics include [Zan76, Gra79].

The relationship of acyclicity with dependencies is considered in Chapter 8.

Many variations of the universal relation assumption arose in the late 1970s and early 1980s. We return to this topic in Chapter 11; surveys of these notions include [AP82, Ull82a, MRW86].

Exercises

Exercise 6.1

(a) Give detailed definitions for the rewrite rules proposed in Section 6.1. In other words, provide the conditions under which they preserve equivalence.

(b) Give the step-by-step description of how the query tree of Fig. 6.1(a) can be transformed into the query tree of Fig. 6.1(b) using these rewrite rules.

Exercise 6.2 Consider the transformation $\sigma_F(q_1 \bowtie_G q_2) \to \sigma_F(q_1) \bowtie_G q_2$ of Fig. 6.2. Describe a query q and database instance for which applying this transformation yields a query whose direct implementation is dramatically more expensive than that of q.

Exercise 6.3

(a) Write generalized SPC queries equivalent to the two tableau queries of Example 6.2.2.

(b) Show that the optimization of this example cannot be achieved using the rewrite rules or multiway join techniques of System/R or INGRES discussed in Section 6.1.

(c) Generate an example analogous to that of Example 6.2.2 that shows that even for typed tableau queries, the rewrite rules of Section 6.1 cannot achieve the optimizations of the Homomorphism Theorem.

Exercise 6.4 Present an algorithm that identifies when variables can be projected out during a left-to-right join of a sip strategy.

Exercise 6.5 Describe a generalization of sip strategies that permits evaluation of multiway joins according to an arbitrary binary tree rather than using only left-to-right join processing. Give an example in which this yields an evaluation plan more efficient than any left-to-right join.

Exercise 6.6 Consider query expressions that have the form (†) mentioned in the discussion of join detachment in Section 6.1.

(a) Describe how the possibility of applying join detachment depends on how equalities are expressed in the conditions (e.g., Is there a difference between using conditions '$x.1 = y.1, y.1 = z.1$' versus '$x.1 = z.1, z.1 = y.1$'?). Describe a technique for eliminating this dependence.

(b) Develop a generalization of join detachment in which a set of variables serves as the pivot.

Exercise 6.7 [WY76]

(a) Describe some heuristics for choosing the atom $R_i(s_i)$ for forming a tuple substitution. These may be in the context of using tuple substitution and join detachment for the resulting subqueries, or they may be in a more general context.

(b) Develop a query optimization algorithm based on applying single-variable conditions, join detachment, and tuple substitution.

Exercise 6.8 Prove Corollary 6.2.4.

Exercise 6.9

(a) State the direct generalization of Theorem 6.2.3 for tableau queries with equality, and show that it does not hold.

(b) State and prove a correct generalization of Theorem 6.2.3 that handles tableau queries with equality.

Exercise 6.10 For queries q, q', write $q \subset q'$ to denote that $q \subseteq q'$ and $q \not\equiv q'$. The meaning of $q \supset q'$ is defined analogously.

(a) Exhibit an infinite set $\{q_0, q_1, q_2, \ldots\}$ of typed tableau queries involving no constants over a single relation with the property that $q_0 \subset q_1 \subset q_2 \subset \ldots$.

(b) Exhibit an infinite set $\{q'_0, q'_1, q'_2, \ldots\}$ of (possibly nontyped) tableau queries involving no constants over a single relation such that $q'_i \not\subseteq q'_j$ and $q'_j \not\subseteq q'_i$ for each pair $i \neq j$.

(c) Exhibit an infinite set $\{q''_0, q''_1, q''_2, \ldots\}$ of (possibly nontyped) tableau queries involving no constants over a single relation with the property that $q''_0 \supset q''_1 \supset q''_2 \supset \ldots$.

(d) Do parts (b) and (c) for typed tableau queries that may contain constants.

★(e) [FUMY83] Do parts (b) and (c) for typed tableau queries that contain no constants.

Exercise 6.11 [CM77] Prove Proposition 6.2.9.

Exercise 6.12

(a) Prove that if the underlying domain **dom** is finite, then only one direction of the statement of Theorem 6.2.3 holds.

(b) Let $n > 1$ be arbitrary. Exhibit a pair of tableau queries q, q' such that under the assumption that **dom** has n elements, $q \subseteq q'$, but there is no homomorphism from q' to q. In addition, do this using typed tableau queries.

(c) Show for arbitrary $n > 1$ that Theorem 6.2.6 and Proposition 6.2.9 do not hold if **dom** has n elements.

Exercise 6.13 Let R be a relation schema of sort ABC. For each of the following SPJR queries over R, construct an equivalent tableau (see Exercise 4.19), minimize the tableau, and construct from the minimized tableau an equivalent SPJR query with minimal number of joins.

(a) $\pi_{AC}[\pi_{AB}(R) \bowtie \pi_{BC}(R)] \bowtie \pi_A[\pi_{AC}(R) \bowtie \pi_{CB}(R)]$

(b) $\pi_{AC}[\pi_{AB}(R) \bowtie \pi_{BC}(R)] \bowtie \pi_{AB}(\sigma_{B=8}(R)) \bowtie \pi_{BC}(\sigma_{A=5}(R))$

(c) $\pi_{AB}(\sigma_{C=1}(R)) \bowtie \pi_{BC}(R) \bowtie \pi_{AB}[\sigma_{C=1}(\pi_{AC}(R)) \bowtie \pi_{CB}(R)]$

♠**Exercise 6.14** [SY80]

(a) Give a decision procedure for determining whether one union of tableaux query is contained in another one. *Hint:* Let the queries be $q = (\{\mathbf{T}_1, \ldots, \mathbf{T}_n\}, u)$ and $q' = (\{\mathbf{S}_1, \ldots, \mathbf{S}_m\}, v)$; and prove that $q \subseteq q'$ iff for each $i \in [1, n]$ there is some $j \in [1, m]$ such that $(\mathbf{T}_i, u) \subseteq (\mathbf{S}_j, v)$. (The case of queries equivalent to $q^{\emptyset}$ must be handled separately.)

A union of tableaux query $(\{\mathbf{T}_1, \ldots, \mathbf{T}_n\}, u)$ is *nonredundant* if there is no distinct pair i, j such that $(\mathbf{T}_i, u) \subseteq (\mathbf{T}_j, u)$.

(b) Prove that if $(\{\mathbf{T}_1, \ldots, \mathbf{T}_n\}, u)$ and $(\{\mathbf{S}_1, \ldots, \mathbf{S}_m\}, v)$ are nonredundant and equivalent, then $n = m$; for each $i \in [1, n]$ there is a $j \in [1, n]$ such that $(\mathbf{T}_i, u) \equiv (\mathbf{S}_j, v)$; and for each $j \in [1, n]$ there is a $i \in [1, n]$ such that $(\mathbf{S}_j, v) \equiv (\mathbf{T}_i, u)$.

(c) Prove that for each union of tableaux query q there is a unique (up to renaming) equivalent union of tableaux query that has a minimal total number of atoms.

Exercise 6.15 Exhibit a pair of typed restricted SPJ algebra queries q_1, q_2 over a relation R and having no constants, such that there is no conjunctive query equivalent to $q_1 \cup q_2$. *Hint:* Use tableau techniques.

♠ **Exercise 6.16** [SY80]

(a) Complete the proof of part (a) of Theorem 6.2.10.

(b) Prove parts (b) and (c) of that theorem. *Hint:* Given ξ and $q_\xi = (T_\xi, t)$ and $q'_\xi = (T'_\xi, t)$ as in the proof of part (a), set $q''_\xi = (T_\xi \cup T'_\xi, t)$. Show that ξ is satisfiable iff $q''_\xi \equiv q'_\xi$.

(c) Prove that it is NP-hard to determine, given a pair q, q' of typed tableau queries over the same relation schema, whether q is minimal and equivalent to q'. Conclude that optimizing conjunctive queries, in the sense of finding an equivalent with minimal number of atoms, is NP-hard.

Exercise 6.17 [ASU79b] Prove Theorem 6.2.10 using a reduction from 3-SAT (see Chapter 2) rather than from the exact cover problem.

Exercise 6.18 [ASU79b]

(a) Prove that determining containment between two typed SPJ queries of the form $\pi_X(\bowtie_{i=1}^n (\pi_{X_i} R))$ is NP-complete. *Hint:* Use Exercise 6.16.

(b) Prove that the problem of finding, given an SPJ query q of the form $\pi_X(\bowtie_{i=1}^n (\pi_{X_i} R))$, an SPJ query q' equivalent to q that has the minimal number of join operations among all such queries is NP-hard.

Exercise 6.19

(a) Complete the proof of Theorem 6.3.1.

(b) Describe how to modify that proof so that $q_{\mathcal{P}}$ uses no constants.

(c) Describe how to modify the proof so that no constants and only one ternary relation is used. *Hint:* Speaking intuitively, a tuple $t = \langle a_1, \ldots, a_5\rangle$ of *ENC* can be simulated as a set of tuples $\{\langle b_t, b_1, a_1\rangle, \ldots, \langle b_t, b_5, a_5\rangle\}$, where b_t is a value not used elsewhere and $b_1, \ldots, b_5$ are values established to serve as integers $1, \ldots, 5$.

(d) Describe how, given instance $\mathcal{P}$ of the PCP, to construct an nr-datalog$^\neg$ program that is satisfiable iff $\mathcal{P}$ has a solution.

Exercise 6.20 This exercise develops further undecidability results for the relational calculus.

(a) Prove that containment and equivalence of range-safe calculus queries are co-r.e.

(b) Prove that domain independence of calculus queries is co-r.e. *Hint:* Theorem 5.6.1 is useful here.

(c) Prove that containment of safe-range calculus queries is undecidable.

(d) Show that there is no algorithm that always halts and on input calculus query q gives an equivalent query q' of minimum length. Conclude that "complete" optimization of the relational calculus is impossible. *Hint:* If there were such an algorithm, then it would map each unsatisfiable query to a query with formula (of form) $\neg(a = b)$.

♠ **Exercise 6.21** [ASU79a, ASU79b] In a typed tableau query (T, u), a *summary variable* is a variable occurring in u. A *repeated nonsummary variable* for attribute A is a nonsummary variable in $\pi_A(T)$ that occurs more than once in T. A typed tableau query is *simple* if for each attribute A, there is a repeated nonsummary variable in $\pi_A(T)$, then no other constant or variable in $\pi_A(T)$ occurs more than once $\pi_A(T)$. Many natural typed restricted SPJ queries translate into simple tableau queries.

(a) Show that the tableau query over $R[ABCD]$ corresponding to

$$\pi_{AC}(\pi_{AB}(R) \bowtie \pi_{BC}(R)) \bowtie (\pi_{AB}(R) \bowtie \pi_{BD}(R))$$

is *not* simple.

(b) Exhibit a simple tableau query that is not the result of transforming a typed restricted SPJ query under the algorithm of Exercise 4.19.

(c) Prove that if (T, u) is simple, $T' \subseteq T$, and (T', u) is a tableau query, then (T', u) is simple.

(d) Develop an $O(n^4)$ algorithm that, on input a simple tableau query q, produces a minimal tableau query equivalent to q.

(e) Develop an $O(n^3)$ algorithm that, given simple tableau queries q, q', determines whether $q \equiv q'$.

(f) Prove that testing containment for simple tableau queries is NP-complete.

♠ **Exercise 6.22** [SY80] Characterize containment and equivalence between queries of the form $q_1 - q_2$, where q_1, q_2 are SPCU queries. *Hint:* First develop characterizations for the case in which q_1, q_2 are SPC queries.

Exercise 6.23 Recall from Exercise 5.9 that an arbitrary nonrecursive datalog$^\neg$ rule can be described as a difference $q_1 - q_2$, where q_1 is an SPC query and q_2 is an SPCU query.

(a) Show that Exercise 5.9 cannot be strengthened so that q_2 is an SPC query.

(b) Show that containment between pairs of nonrecursive datalog$^\neg$ rules is decidable. *Hint:* Use Exercise 6.22.

(c) Recall that for each nr-datalog program P with a single-relation target there is an equivalent nr-datalog program P' such that all rule heads have the same relation name (see Exercise 4.24). Prove that the analogous result does not hold for nr-datalog$^\neg$ programs.

Exercise 6.24

(a) Verify that $I \bowtie J = (I \ltimes J) \bowtie J$.

(b) Analyze the transmission costs incurred by the left-hand and right-hand sides of this equation, and describe conditions under which one is more efficient than the other.

Exercise 6.25 [HLY80] Prove that the problem of deciding, given instance $\mathbf{I}$ of database schema $\mathbf{R}$, whether $\mathbf{I}$ is globally consistent is NP-complete.

Exercise 6.26 Prove the following without using Theorem 6.4.5.

(a) The database schema $\mathbf{R} = \{AB, BC, CA\}$ has no full reducer.

(b) For arbitrary $n > 1$, the schema $\{R_1, \ldots, R_{n-1}\}$ of Example 6.4.1 has a full reducer.

(c) For arbitrary (odd or even) $n > 1$, the schema $\{R_1, \ldots, R_n\}$ of Example 6.4.1 has no full reducer.

Exercise 6.27

(a) Draw the hypergraph of the schema of Example 6.4.3.

(b) Draw the hypergraph of Fig. 6.12(b) in a fashion that suggests it to be acyclic.

Exercise 6.28 Prove that the output of Algorithm 6.4.4 is independent of the nondeterministic choices.

Exercise 6.29 As originally introduced, the GYO algorithm involved the following steps:

```
Nondeterministically perform either step,
        until neither can be applied
    1. If $v \in V$ is in exactly one edge $f \in F$
       then $\mathcal{F} := (V - \{v\}, (F - \{f\} \cup \{f - \{v\}\}) - \{\emptyset\})$.
    2. If $f \subseteq f'$ for distinct $f, f' \in F$,
       then $\mathcal{F} := (V, F - \{f\})$.
```

The result of applying the original GYO algorithm to a schema **R** is the *GYO reduction* of **R**.

(a) Prove that the original GYO algorithm yields the same output independent of the nondeterministic choices.

(b) [FMU82] Prove that Algorithm 6.4.4 given in the text yields the empty hypergraph on **R** iff the GYO reduction of **R** is the empty hypergraph.

Exercise 6.30 This exercise completes the proof of Theorem 6.4.5.

(a) [BG81] Prove that (3) $\Leftrightarrow$ (4).

(b) Complete the other parts of the proof.

Exercise 6.31 [BFMY83] **R** has the *running intersection property* if there is an ordering $R_1, \ldots, R_n$ of **R** such that for $2 \le i \le n$ there exists $j_i < i$ such that $R_i \cap (R_1 \cup \cdots \cup R_{i-1}) \subseteq R_{j_i}$. In other words, the intersection of each R_i with the union of the previous R_j's is contained in one of these. Prove that **R** has the running intersection property iff **R** is acyclic.

Exercise 6.32 [BFMY83] A *Berge cycle* in a hypergraph $\mathcal{F}$ is a sequence $(f_1, v_1, f_2, v_2, \ldots, f_n, v_n, f_{n+1})$ such that

(i) $v_1, \ldots, v_n$ are distinct vertexes of $\mathcal{F}$;

(ii) $f_1, \ldots, f_n$ are distinct edges of $\mathcal{F}$, and $f_{n+1} = f_1$;

(iii) $n \ge 2$; and

(iv) $v_i \in f_i \cap f_{i+1}$ for $i \in [1, n]$.

A hypergraph is *Berge cyclic* if it has a Berge cycle, and it is *Berge acyclic* otherwise.

(a) Prove that Berge acyclicity is necessary but not sufficient for acyclicity.

(b) Show that any hypergraph in which two edges have two nodes in common is Berge cyclic.

Exercise 6.33 [Yan81] Complete the proof of Corollary 6.4.6.