27 DEDUCTIVE DATABASES

For 'Is' and 'Is-Not' though with Rule and Line,
And 'Up-and-Down' by Logic I define,
Of all that one should care to fathom, I
Was never deep in anything but—Wine.

—Rubaiyat of Omar Khayyam, Translated by Edward Fitzgerald

Relational database management systems have been enormously successful for administrative data processing. In recent years, however, as people have tried to use database systems in increasingly complex applications, some important limitations of these systems have been exposed. For some applications, the query language and constraint definition capabilities have been found to be inadequate. As an example, some companies maintain a huge parts inventory database and frequently want to ask questions such as, "Are we running low on any parts needed to build a ZX600 sports car?" or, "What is the total component and assembly cost to build a ZX600 at today's part prices?" These queries cannot be expressed in SQL-92.

We begin this chapter by discussing queries that cannot be expressed in relational algebra or SQL and present a more powerful relational language called *Datalog*. Queries and views in SQL can be understood as **if–then** rules: "**If** some tuples exist in tables mentioned in the `FROM` clause that satisfy the conditions listed in the `WHERE` clause, **then** the tuple described in the `SELECT` clause is included in the answer." Datalog definitions retain this **if–then** reading, with the significant new feature that definitions can be *recursive*, that is, a table can be defined in terms of itself.

Evaluating Datalog queries poses some additional challenges, beyond those encountered in evaluating relational algebra queries, and we discuss some important implementation and optimization techniques that were developed to address these challenges. Interestingly, some of these techniques have been found to improve performance of even nonrecursive SQL queries and have therefore been implemented in several current relational DBMS products. Some systems, notably IBM's DB2 DBMS, support recursive queries and the SQL:1999 standard, the successor to the SQL-92 standard, requires support for recursive queries.

We concentrate on the main ideas behind recursive queries and briefly cover the SQL:1999 features that support these ideas. In Section 27.1, we introduce recursive

Recursion in SQL: The concepts discussed in this chapter are not included in the SQL-92 standard. However, the revised version of the SQL standard, SQL:1999, includes support for recursive queries and IBM's DB2 system already supports recursive queries as required in SQL:1999.

queries and Datalog notation through an example. We present the theoretical foundations for recursive queries, namely least fixpoints and least models, in Section 27.2. We discuss queries that involve the use of negation or set-difference in Section 27.3. Finally, we consider techniques for evaluating recursive queries efficiently in Section 27.4.

27.1 INTRODUCTION TO RECURSIVE QUERIES

We begin with a simple example that illustrates the limits of SQL-92 queries and the power of recursive definitions. Let Assembly be a relation with three fields *part*, *subpart*, and *qty*. An example instance of Assembly is shown in Figure 27.1. Each tuple in Assembly indicates how many copies of a particular subpart are contained in a given part. The first tuple indicates, for example, that a trike contains three wheels. The Assembly relation can be visualized as a tree, as shown in Figure 27.2. A tuple is shown as an edge going from the part to the subpart, with the *qty* value as the edge label.

part	*subpart*	*qty*
trike	wheel	3
trike	frame	1
frame	seat	1
frame	pedal	1
wheel	spoke	2
wheel	tire	1
tire	rim	1
tire	tube	1

Figure 27.1 An Instance of Assembly

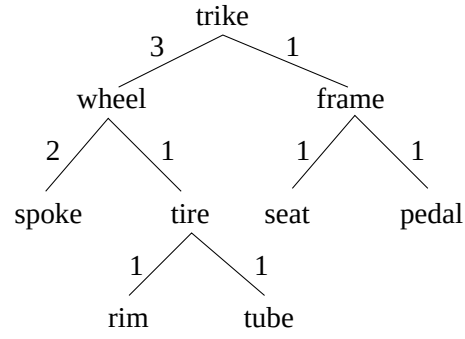

Figure 27.2 Assembly Instance Seen as a Tree

A natural question to ask is, "What are the components of a trike?" Rather surprisingly, this query is impossible to write in SQL-92. Of course, if we look at a given instance of the Assembly relation, we can write a 'query' that takes the union of the parts that are used in a trike. But such a query is not interesting—we want a query that identifies all components of a trike for *any* instance of Assembly, and such a query cannot be written in relational algebra or in SQL-92. Intuitively, the problem is that we are forced to join the Assembly relation with itself in order to recognize that *trike*

contains *spoke* and *tire*, that is, to go one level down the Assembly tree. For each additional level, we need an additional join; two joins are needed to recognize that *trike* contains *rim*, which is a subpart of *tire*. Thus, the number of joins needed to identify all subparts of *trike* depends on the height of the Assembly tree, that is, on the given instance of the Assembly relation. There is no relational algebra query that works for all instances; given any query, we can construct an instance whose height is greater than the number of joins in the query.

27.1.1 Datalog

We now define a relation called Components that identifies the components of every part. Consider the following **program**, or collection of **rules**:

```
Components(Part, Subpart) :-  Assembly(Part, Subpart, Qty).
Components(Part, Subpart) :-  Assembly(Part, Part2, Qty),
                              Components(Part2, Subpart).
```

These are rules in **Datalog**, a relational query language inspired by Prolog, the well-known logic programming language; indeed, the notation follows Prolog. The first rule should be read as follows:

For all values of Part, Subpart, and Qty,
if there is a tuple ⟨Part, Subpart, Qty⟩ in Assembly,
then there must be a tuple ⟨Part, Subpart⟩ in Components.

The second rule should be read as follows:

For all values of Part, Part2, Subpart, and Qty,
if there is a tuple ⟨Part, Part2, Qty⟩ in Assembly **and**
a tuple ⟨Part2, Subpart⟩ in Components,
then there must be a tuple ⟨Part, Subpart⟩ in Components.

The part to the right of the :- symbol is called the **body** of the rule, and the part to the left is called the **head** of the rule. The symbol :- denotes logical implication; if the tuples mentioned in the body exist in the database, it is implied that the tuple mentioned in the head of the rule must also be in the database. (Note that the body could be empty; in this case, the tuple mentioned in the head of the rule must be included in the database.) Therefore, if we are given a set of Assembly and Components tuples, each rule can be used to **infer**, or **deduce**, some new tuples that belong in Components. This is why database systems that support Datalog rules are often called **deductive database systems**.

Each rule is really a *template* for making inferences: by assigning constants to the variables that appear in a rule, we can infer specific Components tuples. For example,

by setting Part=*trike*, Subpart=*wheel*, and Qty=*3*, we can infer that $\langle trike, wheel \rangle$ is in Components. By considering each tuple in Assembly in turn, the first rule allows us to infer that the set of tuples obtained by taking the projection of Assembly onto its first two fields is in Components.

The second rule then allows us to combine previously discovered Components tuples with Assembly tuples to infer new Components tuples. We can apply the second rule by considering the cross-product of Assembly and (the current instance of) Components and assigning values to the variables in the rule for each row of the cross-product, one row at a time. Observe how the repeated use of the variable Part2 prevents certain rows of the cross-product from contributing any new tuples; in effect, it specifies an equality join condition on Assembly and Components. The tuples obtained by one application of this rule are shown in Figure 27.3. (In addition, Components contains the tuples obtained by applying the first rule; these are not shown.)

part	*subpart*
trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	rim
wheel	tube

Figure 27.3 Components Tuples Obtained by Applying the Second Rule Once

part	*subpart*
trike	spoke
trike	tire
trike	seat
trike	pedal
wheel	rim
wheel	tube
trike	rim
trike	tube

Figure 27.4 Components Tuples Obtained by Applying the Second Rule Twice

The tuples obtained by a second application of this rule are shown in Figure 27.4. Note that each tuple shown in Figure 27.3 is reinferred. Only the last two tuples are new.

Applying the second rule a third time does not generate any additional tuples. The set of Components tuples shown in Figure 27.4 includes all the tuples that can be inferred using the two Datalog rules defining Components and the given instance of Assembly. The components of a trike can now be obtained by selecting all Components tuples with the value *trike* in the first field.

Each application of a Datalog rule can be understood in terms of relational algebra. The first rule in our example program simply applies projection to the Assembly relation and adds the resulting tuples to the Components relation, which is initially empty. The second rule joins Assembly with Components and then does a projection. The result of each rule application is combined with the existing set of Components tuples using union.

The only Datalog operation that goes beyond relational algebra is the *repeated* application of the rules defining Components until no new tuples are generated. This repeated application of a set of rules is called the *fixpoint* operation, and we develop this idea further in the next section.

We conclude this section by rewriting the Datalog definition of Components in terms of extended SQL, using the syntax proposed in the SQL:1999 draft and currently supported in IBM's DB2 Version 2 DBMS:

```
WITH RECURSIVE Components(Part, Subpart) AS
        (SELECT A1.Part, A1.Subpart FROM Assembly A1)
        UNION
        (SELECT A2.Part, C1.Subpart
        FROM    Assembly A2, Components C1
        WHERE   A2.Subpart = C1.Part)

SELECT * FROM Components C2
```

The `WITH` clause introduces a relation that is part of a query definition; this relation is similar to a view, but the scope of a relation introduced using `WITH` is local to the query definition. The `RECURSIVE` keyword signals that the table (in our example, Components) is recursively defined. The structure of the definition closely parallels the Datalog rules. Incidentally, if we wanted to find the components of a particular part, for example, *trike*, we can simply replace the last line with the following:

```
SELECT * FROM Components C2
WHERE   C2.Part = 'trike'
```

27.2 THEORETICAL FOUNDATIONS

We classify the relations in a Datalog program as either output relations or input relations. **Output relations** are defined by rules (e.g., Components), and **input relations** have a set of tuples explicitly listed (e.g., Assembly). Given instances of the input relations, we must compute instances for the output relations. The meaning of a Datalog program is usually defined in two different ways, both of which essentially describe the relation instances for the output relations. Technically, a **query** is a selection over one of the output relations (e.g., all Components tuples C with $C.part$ = *trike*). However, the meaning of a query is clear once we understand how relation instances are associated with the output relations in a Datalog program.

The first approach to defining what a Datalog program means is called the *least model semantics* and gives users a way to understand the program without thinking about how the program is to be executed. That is, the semantics is *declarative*, like the semantics

of relational calculus, and not *operational* like relational algebra semantics. This is important because the presence of recursive rules makes it difficult to understand a program in terms of an evaluation strategy.

The second approach, called the *least fixpoint semantics*, gives a conceptual evaluation strategy to compute the desired relation instances. This serves as the basis for recursive query evaluation in a DBMS. More efficient evaluation strategies are used in an actual implementation, but their correctness is shown by demonstrating their equivalence to the least fixpoint approach. The fixpoint semantics is thus operational and plays a role analogous to that of relational algebra semantics for nonrecursive queries.

27.2.1 Least Model Semantics

We want users to be able to understand a Datalog program by understanding each rule independently of other rules, with the meaning: *If the body is true, the head is also true.* This intuitive reading of a rule suggests that given certain relation instances for the relation names that appear in the body of a rule, the relation instance for the relation mentioned in the head of the rule must contain a certain set of tuples. If a relation name R appears in the heads of several rules, the relation instance for R must satisfy the intuitive reading of all these rules. However, we do not want tuples to be included in the instance for R unless they are necessary to satisfy one of the rules defining R. That is, we only want to compute tuples for R that are supported by some rule for R.

To make these ideas precise, we need to introduce the concepts of models and least models. A **model** is a collection of relation instances, one instance for each relation in the program, that satisfies the following condition. For every rule in the program, whenever we replace each variable in the rule by a corresponding constant, the following holds:

> **If** every tuple in the body (obtained by our replacement of variables with constants) is in the corresponding relation instance,
>
> **Then** the tuple generated for the head (by the assignment of constants to variables that appear in the head) is also in the corresponding relation instance.

Observe that the instances for the input relations are given, and the definition of a model essentially restricts the instances for the output relations.

Consider the rule:

```
Components(Part, Subpart) :-  Assembly(Part, Part2, Qty),
```

```
                                    Components(Part2, Subpart).
```

Suppose that we replace the variable Part by the constant *wheel*, Part2 by *tire*, Qty by 1, and Subpart by *rim*:

```
Components(wheel, rim) :-     Assembly(wheel, tire, 1),
                              Components(tire, rim).
```

Let A be an instance of Assembly and C be an instance of Components. If A contains the tuple $\langle wheel, tire, 1\rangle$ and C contains the tuple $\langle tire, rim\rangle$, then C must also contain the tuple $\langle wheel, rim\rangle$ in order for the pair of instances A and C to be a model. Of course, the instances A and C must satisfy the inclusion requirement illustrated above for *every* assignment of constants to the variables in the rule: If the tuples in the rule body are in A and C, the tuple in the head must be in C.

As an example, the instance of Assembly shown in Figure 27.1 and the instance of Components shown in Figure 27.4 together form a model for the Components program.

Given the instance of Assembly shown in Figure 27.1, there is no justification for including the tuple $\langle spoke, pedal\rangle$ to the Components instance. Indeed, if we add this tuple to the components instance in Figure 27.4, we no longer have a model for our program, as the following instance of the recursive rule demonstrates, since $\langle wheel, pedal\rangle$ is not in the Components instance:

```
Components(wheel, pedal) :-   Assembly(wheel, spoke, 2),
                              Components(spoke, pedal).
```

However, by also adding the tuple $\langle wheel, pedal\rangle$ to the Components instance, we obtain another model of the Components program! Intuitively, this is unsatisfactory since there is no justification for adding the tuple $\langle spoke, pedal\rangle$ in the first place, given the tuples in the Assembly instance and the rules in the program.

We address this problem by using the concept of a least model. A **least model** of a program is a model M such that for every other model M2 of the same program, for each relation R in the program, the instance for R in M is contained in the instance of R in M2. The model formed by the instances of Assembly and Components shown in Figures 27.1 and 27.4 is the least model for the Components program with the given Assembly instance.

27.2.2 Safe Datalog Programs

Consider the following program:

```
Complex_Parts(Part) :-        Assembly(Part, Subpart, Qty), Qty > 2.
```

According to this rule, complex part is defined to be any part that has more than two copies of any one subpart. For each part mentioned in the Assembly relation, we can easily check if it is a complex part. In contrast, consider the following program:

```
Price_Parts(Part,Price) :-    Assembly(Part, Subpart, Qty), Qty > 2.
```

This variation seeks to associate a price with each complex part. However, the variable *Price* does not appear in the body of the rule. This means that an infinite number of tuples must be included in any model of this program! To see this, suppose that we replace the variable Part by the constant *trike*, SubPart by *wheel*, and Qty by 3. This gives us a version of the rule with the only remaining variable being *Price*:

```
Price_Parts(trike,Price) :-   Assembly(trike, wheel, 3), 3 > 2.
```

Now, any assignment of a constant to *Price* gives us a tuple to be included in the output relation Price_Parts. For example, replacing *Price* by 100 gives us the tuple Price_Parts(trike,100). If the least model of a program is not finite, for even one instance of its input relations, then we say the program is **unsafe**.

Database systems disallow unsafe programs by requiring that every variable in the head of a rule must also appear in the body. Such programs are said to be **range-restricted**, and every range-restricted Datalog program has a finite least model if the input relation instances are finite. In the rest of this chapter, we will assume that programs are range-restricted.

27.2.3 The Fixpoint Operator

A **fixpoint** of a function f is a value v such that the function applied to the value returns the same value, that is, $f(v) = v$. Consider a function that is applied to a set of values and also returns a set of values. For example, we can define *double* to be a function that multiplies every element of the input set by two, and *double+* to be *double* $\cup$ *identity*. Thus, *double*({1,2,5}) = {2,4,10}, and *double+*({1,2,5}) = {1,2,4,5,10}. The set of all even integers—which happens to be an infinite set!—is a fixpoint of the function *double+*. Another fixpoint of the function *double+* is the set of all integers. The first fixpoint (the set of all even integers) is *smaller* than the second fixpoint (the set of all integers) because it is contained in the latter.

The **least fixpoint** of a function is a fixpoint that is smaller than every other fixpoint of that function. In general it is not guaranteed that a function has a least fixpoint.

For example, there may be two fixpoints, neither of which is smaller than the other. (Does *double* have a least fixpoint? What is it?)

Now let us turn to functions over sets of tuples, in particular, functions defined using relational algebra expressions. The Components relation can be defined by an equation of the form:

$$Components \;=\; \pi_{1,5}(Assembly \bowtie_{2=1} Components) \;\cup\; \pi_{1,2}(Assembly)$$

This equation has the form

$$Components \;=\; f(Components, Assembly)$$

where the function f is defined using a relational algebra expression. For a given instance of the input relation Assembly, this can be simplified to:

$$Components \;=\; f(Components)$$

The least fixpoint of f is an instance of Components that satisfies the above equation. Clearly the projection of the first two fields of the tuples in the given instance of the input relation Assembly must be included in the (instance that is the) least fixpoint of Components. In addition, any tuple obtained by joining Components with Assembly and projecting the appropriate fields must also be in Components.

A little thought shows that the instance of Components that is the least fixpoint of f can be computed using repeated applications of the Datalog rules shown in the previous section. Indeed, applying the two Datalog rules is identical to evaluating the relational expression used in defining Components. If an application generates Components tuples that are not in the current instance of the Components relation, the current instance cannot be the fixpoint. Therefore, we add the new tuples to Components and evaluate the relational expression (equivalently, the two Datalog rules) again. This process is repeated until every tuple generated is already in the current instance of Components; at this point, we have reached a fixpoint. If Components is initialized to the empty set of tuples, intuitively we infer only tuples that are necessary by the definition of a fixpoint, and the fixpoint computed is the least fixpoint.

27.2.4 Least Model = Least Fixpoint

Does a Datalog program always have a least model? Or is it possible that there are two models, neither of which is contained in the other? Similarly, does every Datalog program have a least fixpoint? What is the relationship between the least model and the least fixpoint of a Datalog program?

As we noted earlier in this section, not every function has a least fixpoint. Fortunately, every function defined in terms of relational algebra expressions that do not contain set-difference is guaranteed to have a least fixpoint, and the least fixpoint can be computed

by repeatedly evaluating the function. This tells us that every Datalog program has a least fixpoint, and that the least fixpoint can be computed by repeatedly applying the rules of the program on the given instances of the input relations.

Further, every Datalog program is guaranteed to have a least model, and the least model is equal to the least fixpoint of the program! These results (whose proofs we will not discuss) provide the basis for Datalog query processing. Users can understand a program in terms of 'If the body is true, the head is also true,' thanks to the least model semantics. The DBMS can compute the answer by repeatedly applying the program rules, thanks to the least fixpoint semantics and the fact that the least model and the least fixpoint are identical.

Unfortunately, once set-difference is allowed in the body of a rule, there may no longer be a least model or a least fixpoint. We consider this point further in the next section.

27.3 RECURSIVE QUERIES WITH NEGATION

Consider the following rules:

```
Big(Part) :-    Assembly(Part, Subpart, Qty), Qty > 2,
                not Small(Part).
Small(Part) :-  Assembly(Part, Subpart, Qty), not Big(Part).
```

These two rules can be thought of as an attempt to divide parts (those that are mentioned in the first column of the Assembly table) into two classes, Big and Small. The first rule defines Big to be the set of parts that use at least three copies of some subpart and that are not classified as small parts. The second rule defines Small as the set of parts that are not classified as big parts.

If we apply these rules to the instance of Assembly shown in Figure 27.1, *trike* is the only part that uses at least three copies of some subpart. Should the tuple $\langle trike \rangle$ be in Big or Small? If we apply the first rule and then the second rule, this tuple is in Big. To apply the first rule, we consider the tuples in Assembly, choose those with Qty $>$ 2 (which is just $\langle trike \rangle$), discard those that are in the current instance of Small (both Big and Small are initially empty), and add the tuples that are left to Big. Therefore, an application of the first rule adds $\langle trike \rangle$ to Big. Proceeding similarly, we can see that if the second rule is applied before the first, $\langle trike \rangle$ is added to Small instead of Big!

This program has two fixpoints, neither of which is smaller than the other, as shown in Figure 27.5. The first fixpoint has a Big tuple that does not appear in the second fixpoint; therefore, it is not smaller than the second fixpoint. The second fixpoint has a Small tuple that does not appear in the first fixpoint; therefore, it is not smaller than

the first fixpoint. The order in which we apply the rules determines which fixpoint is computed, and this situation is very unsatisfactory. We want users to be able to understand their queries without thinking about exactly how the evaluation proceeds.

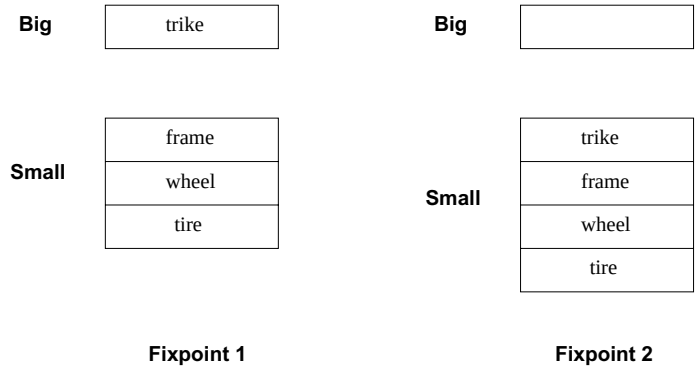

Figure 27.5 Two Fixpoints for the Big/Small Program

The root of the problem is the use of **not**. When we apply the first rule, some inferences are disallowed because of the presence of tuples in Small. Parts that satisfy the other conditions in the body of the rule are candidates for addition to Big, and we remove the parts in Small from this set of candidates. Thus, some inferences that are possible if Small is empty (as it is before the second rule is applied) are disallowed if Small contains tuples (generated by applying the second rule before the first rule). Here is the difficulty: If **not** is used, the addition of tuples to a relation can *disallow* the inference of other tuples. Without **not**, this situation can never arise; the addition of tuples to a relation can *never* disallow the inference of other tuples.

27.3.1 Range-Restriction and Negation

If rules are allowed to contain **not** in the body, the definition of range-restriction must be extended in order to ensure that all range-restricted programs are safe. If a relation appears in the body of a rule preceded by **not**, we call this a **negated occurrence**. Relation occurrences in the body that are not negated are called **positive occurrences**. A program is **range-restricted** if every variable in the head of the rule appears in some positive relation occurrence in the body.

27.3.2 Stratification

A widely used solution to the problem caused by negation, or the use of **not**, is to impose certain syntactic restrictions on programs. These restrictions can be easily checked, and programs that satisfy them have a natural meaning.

We say that a table T **depends on** a table S if some rule with T in the head contains S, or (recursively) contains a predicate that depends on S, in the body. A recursively defined predicate always depends on itself. For example, Big depends on Small (and on itself). Indeed, the tables Big and Small are **mutually recursive**, that is, the definition of Big depends on Small and vice versa. We say that a table T **depends negatively on** a table S if some rule with T in the head contains **not** S, or (recursively) contains a predicate that depends negatively on S, in the body.

Suppose that we classify the tables in a program into **strata** or **layers** as follows. The tables that do not depend on any other tables are in stratum 0. In our Big/Small example, Assembly is the only table in stratum 0. Next, we identify tables in stratum 1; these are tables that depend only on tables in stratum 0 or stratum 1 and depend negatively only on tables in stratum 0. Higher strata are similarly defined: The tables in stratum i are those that do not appear in lower strata, depend only on tables in stratum i or lower strata, and depend negatively only on tables in lower strata. A **stratified program** is a program whose tables can be classified into strata according to the above algorithm.

The Big/Small program is not stratified. Since Big and Small depend on each other, they must be in the same stratum. However, they depend negatively on each other, violating the requirement that a table can depend negatively only on tables in lower strata. Consider the following variant of the Big/Small program, in which the first rule has been modified:

```
Big2(Part) :- Assembly(Part, Subpart, Qty), Qty > 2.
Small2(Part) :- Assembly(Part, Subpart, Qty), not Big2(Part).
```

This program is stratified. Small2 depends on Big2 but Big2 does not depend on Small2. Assembly is in stratum 0, Big is in stratum 1, and Small2 is in stratum 2.

A stratified program is evaluated stratum-by-stratum, starting with stratum 0. To evaluate a stratum, we compute the fixpoint of all rules defining tables that belong to this stratum. When evaluating a stratum, any occurrence of **not** involves a table from a lower stratum, which has therefore been completely evaluated by now. The tuples in the negated table will still disallow some inferences, but the effect is completely deterministic, given the stratum-by-stratum evaluation. In the example, Big2 is computed before Small2 because it is in a lower stratum than Small2; $\langle trike \rangle$ is added to Big2. Next, when we compute Small2, we recognize that $\langle trike \rangle$ is not in Small2 because it is already in Big2.

Incidentally, observe that the stratified Big/Small program is not even recursive! If we replaced Assembly by Components, we would obtain a recursive, stratified program: Assembly would be in stratum 0, Components would be in stratum 1, Big2 would also be in stratum 1, and Small2 would be in stratum 2.

Intuition behind Stratification

Consider the stratified version of the Big/Small program. The rule defining Big2 forces us to add $\langle trike \rangle$ to Big2, and it is natural to assume that $\langle trike \rangle$ is the only tuple in Big2, because we have no supporting evidence for any other tuple being in Big2. The minimal fixpoint computed by stratified fixpoint evaluation is consistent with this intuition. However, there is another minimal fixpoint: We can place every part in Big2 and make Small2 be empty. While this assignment of tuples to relations seems unintuitive, it is nonetheless a minimal fixpoint!

The requirement that programs be stratified gives us a natural order for evaluating rules. When the rules are evaluated in this order, the result is a unique fixpoint that is one of the minimal fixpoints of the program. The fixpoint computed by the stratified fixpoint evaluation usually corresponds well to our intuitive reading of a stratified program, even if the program has more than one minimal fixpoint.

For nonstratified Datalog programs, it is harder to identify a natural model from among the alternative minimal models, especially when we consider that the meaning of a program must be clear even to users who do not have expertise in mathematical logic. Although there has been considerable research on identifying natural models for nonstratified programs, practical implementations of Datalog have concentrated on stratified programs.

Relational Algebra and Stratified Datalog

Every relational algebra query can be written as a range-restricted, stratified Datalog program. (Of course, not all Datalog programs can be expressed in relational algebra; for example, the Components program.) We sketch the translation from algebra to stratified Datalog by writing a Datalog program for each of the basic algebra operations, in terms of two example tables R and S, each with two fields:

Selection:	Result(Y) :- R(X,Y), X=*c*.
Projection:	Result(Y) :- R(X,Y).
Cross-product:	Result(X,Y,U,V) :- R(X,Y), S(U,V).
Set-difference:	Result(X,Y) :- R(X,Y), **not** S(U,V).
Union:	Result(X,Y) :- R(X,Y). Result(X,Y) :- S(X,Y).

We conclude our discussion of stratification by noting that the SQL:1999 draft requires programs to be stratified. The stratified Big/Small program is shown below in SQL:1999 notation, with a final additional selection on Big2:

```
WITH
```

```
Big2(Part) AS
        (SELECT   A1.Part FROM Assembly A1 WHERE Qty > 2)
Small2(Part) AS
        ((SELECT  A2.Part FROM Assembly A2)
        EXCEPT
        (SELECT   B1.Part from Big2 B1))

SELECT * FROM Big2 B2
```

27.3.3 Aggregate Operations

Datalog can be extended with SQL-style grouping and aggregation operations. Consider the following program:

```
NumParts(Part, SUM(⟨Qty⟩)) :- Assembly(Part, Subpart, Qty).
```

This program is equivalent to the SQL query:

```
SELECT    A.Part, SUM (A.Qty)
FROM      Assembly A
GROUP BY A.Part
```

The angular brackets $\langle \ldots \rangle$ notation was introduced in the LDL deductive system, one of the pioneering deductive database prototypes developed at MCC in the late 1980s. We use it to denote **multiset generation**, or the creation of multiset-values. In principle, the rule defining NumParts is evaluated by first creating the temporary relation shown in Figure 27.6. We create the temporary relation by sorting on the *part* attribute (which appears in the left side of the rule, along with the $\langle \ldots \rangle$ term) and collecting the multiset of *qty* values for each *part* value. We then apply the SUM aggregate to each multiset-value in the second column to obtain the answer, which is shown in Figure 27.7.

part	$\langle qty \rangle$
trike	{3,1}
frame	{1,1}
wheel	{2,1}
tire	{1,1}

Figure 27.6 Temporary Relation

part	SUM($\langle qty \rangle$)
trike	4
frame	2
wheel	3
tire	2

Figure 27.7 The Tuples in NumParts

The temporary relation shown in Figure 27.6 need not be materialized in order to compute NumParts—for example, SUM can be applied on-the-fly, or Assembly can simply

be sorted and aggregated as described in Section 12.7. However, we observe that several deductive database systems (e.g., LDL, Coral) in fact allowed the materialization of this temporary relation; the following program would do so:

```
TempReln(Part,⟨Qty⟩) :- Assembly(Part, Subpart, Qty).
```

The tuples in this relation are not in first-normal form, and there is no way to create this relation using SQL-92.

The use of aggregate operations leads to problems similar to those caused by **not**, and the idea of stratification can be applied to programs with aggregate operations as well. Consider the following program:

```
NumComps(Part, COUNT(⟨Subpart⟩) ) :- Components(Part, Subpart).
Components(Part, Subpart) :-  Assembly(Part, Subpart, Qty).
Components(Part, Subpart) :-  Assembly(Part, Part2, Qty),
                              Components(Part2, Subpart).
```

The idea is to count the number of subparts for each part; by aggregating over Components rather than Assembly, we can count subparts at any level in the hierarchy instead of just immediate subparts. The important point to note in this example is that we must wait until Components has been completely evaluated before we apply the NumComps rule. Otherwise, we obtain incomplete counts. This situation is analogous to the problem we faced with negation; we have to evaluate the negated relation completely before applying a rule that involves the use of **not**. If a program is stratified with respect to uses of $\langle \ldots \rangle$ as well as **not**, stratified fixpoint evaluation gives us meaningful results.

27.4 EFFICIENT EVALUATION OF RECURSIVE QUERIES

The evaluation of recursive queries has been widely studied. While all the problems of evaluating nonrecursive queries continue to be present, the newly introduced fixpoint operation creates additional difficulties. A straightforward approach to evaluating recursive queries is to compute the fixpoint by repeatedly applying the rules as illustrated in Section 27.1.1. One application of all the program rules is called an **iteration**; we perform as many iterations as necessary to reach the least fixpoint. This approach has two main disadvantages:

- **Repeated inferences:** As Figures 27.3 and 27.4 illustrate, inferences are repeated across iterations. That is, the same tuple is inferred repeatedly in *the same way*, that is, using the same rule and the same tuples for tables in the body of the rule.

- **Unnecessary inferences:** Suppose that we only want to find the components of a *wheel*. Computing the entire Components table is wasteful and does not take advantage of information in the query.

In this section we discuss how each of these difficulties can be overcome. We will consider only Datalog programs without negation.

27.4.1 Fixpoint Evaluation without Repeated Inferences

Computing the fixpoint by repeatedly applying all rules is called **Naive fixpoint evaluation**. Naive evaluation is guaranteed to compute the least fixpoint, but every application of a rule repeats all inferences made by earlier applications of this rule. We illustrate this point using the following rule:

```
Components(Part, Subpart) :-  Assembly(Part, Part2, Qty),
                              Components(Part2, Subpart).
```

When this rule is applied for the first time, after applying the first rule defining Components, the Components table contains the projection of Assembly on the first two fields. Using these Components tuples in the body of the rule, we generate the tuples shown in Figure 27.3. For example, the tuple $\langle wheel, rim \rangle$ is generated through the following inference:

```
Components(wheel, rim) :-     Assembly(wheel, tire, 1),
                              Components(tire, rim).
```

When this rule is applied a second time, the Components table contains the tuples shown in Figure 27.3, in addition to the tuples that it contained before the first application. Using the Components tuples shown in Figure 27.3 leads to new inferences, for example:

```
Components(trike, rim) :-     Assembly(trike, wheel, 3),
                              Components(wheel, rim).
```

However, every inference carried out in the first application of this rule is also repeated in the second application of the rule, since all the Assembly and Components tuples used in the first rule application are considered again. For example, the inference of $\langle wheel, rim \rangle$ shown above is repeated in the second application of this rule.

The solution to this repetition of inferences consists of remembering which inferences were carried out in earlier rule applications and not carrying them out again. It turns out that we can 'remember' previously executed inferences efficiently by simply keeping track of which Components tuples were generated for the first time in the most

recent application of the recursive rule. Suppose that we keep track by introducing a new relation called *delta_Components* and storing just the newly generated Components tuples in it. Now, we can use only the tuples in *delta_Components* in the next application of the recursive rule; any inference using other Components tuples should have been carried out in earlier rule applications.

This refinement of fixpoint evaluation is called **Seminaive fixpoint evaluation**. Let us trace Seminaive fixpoint evaluation on our example program. The first application of the recursive rule produces the Components tuples shown in Figure 27.3, just like Naive fixpoint evaluation, and these tuples are placed in *delta_Components*. In the second application, however, only *delta_Components* tuples are considered, which means that only the following inferences are carried out in the second application of the recursive rule:

```
Components(trike, rim) :-     Assembly(trike, wheel, 3),
                              delta_Components(wheel, rim).
Components(trike, tube) :-    Assembly(trike, wheel, 3),
                              delta_Components(wheel, tube).
```

Next, the bookkeeping relation *delta_Components* is updated to contain just these two Components tuples. In the third application of the recursive rule, only these two *delta_Components* tuples are considered, and thus no additional inferences can be made. The fixpoint of Components has been reached.

To implement Seminaive fixpoint evaluation for general Datalog programs, we apply all the recursive rules in a program together in an **iteration**. Iterative application of all recursive rules is repeated until no new tuples are generated in some iteration. To summarize how Seminaive fixpoint evaluation is carried out, there are two important differences with respect to Naive fixpoint evaluation:

- We maintain a *delta* version of every recursive predicate to keep track of the tuples generated for this predicate in the most recent iteration; for example, *delta_Components* for Components. The *delta* versions are updated at the end of each iteration.
- The original program rules are rewritten to ensure that every inference uses at least one *delta* tuple, that is, one tuple that was not known before the previous iteration. This property guarantees that the inference could not have been carried out in earlier iterations.

We will not discuss the details of Seminaive fixpoint evaluation, such as the algorithm for rewriting program rules to ensure the use of a *delta* tuple in each inference.

27.4.2 Pushing Selections to Avoid Irrelevant Inferences

Consider a nonrecursive view definition. If we want only those tuples in the view that satisfy an additional selection condition, the selection can be added to the plan as a final selection operation, and the relational algebra transformations for commuting selections with other relational operators allow us to 'push' the selection ahead of more expensive operations such as cross-products and joins. In effect, we are able to restrict the computation by utilizing selections in the query specification. The problem is more complicated for recursively defined queries.

We will use the following program as an example in this section:

```
SameLevel(S1, S2) :-  Assembly(P1, S1, Q1), Assembly(P1, S2, Q2).
SameLevel(S1, S2) :-  Assembly(P1, S1, Q1),
                      SameLevel(P1, P2), Assembly(P2, S2, Q2).
```

Consider the tree representation of Assembly tuples illustrated in Figure 27.2. There is a tuple $\langle S1, S2\rangle$ in SameLevel if there is a path from $S1$ to $S2$ that goes up a certain number of edges in the tree and then comes down the same number of edges.

Suppose that we want to find all SameLevel tuples with the first field equal to *spoke*. Since SameLevel tuples can be used to compute other SameLevel tuples, we cannot just compute those tuples with *spoke* in the first field. For example, the tuple $\langle wheel, frame\rangle$ in SameLevel allows us to infer a SameLevel tuple with *spoke* in the first field:

```
SameLevel(spoke, seat) :-  Assembly(wheel, spoke, 2),
                           SameLevel(wheel, frame),
                           Assembly(frame, seat, 1).
```

Intuitively, we have to compute all SameLevel tuples whose first field contains a value that is on the path from *spoke* to the root in Figure 27.2. Each such tuple has the potential to contribute to answers for the given query. On the other hand, computing the entire SameLevel table is wasteful; for example, the SameLevel tuple $\langle tire, seat\rangle$ cannot be used to infer any answer to the given query (or indeed, to infer any tuple that can in turn be used to infer an answer tuple). We can define a new table, which we will call Magic_SameLevel, such that each tuple in this table identifies a value m for which we have to compute all SameLevel tuples with m in the first column, in order to answer the given query:

```
Magic_SameLevel(P1) :-     Magic_SameLevel(S1), Assembly(P1, S1, Q1).
Magic_SameLevel(spoke) :- .
```

Consider the tuples in Magic_SameLevel. Obviously we have $\langle spoke\rangle$. Using this Magic_SameLevel tuple and the Assembly tuple $\langle wheel, spoke, 2\rangle$, we can infer that

the tuple $\langle wheel \rangle$ is in Magic_SameLevel. Using this tuple and the Assembly tuple $\langle trike, wheel, 3 \rangle$, we can infer that the tuple $\langle trike \rangle$ is in Magic_SameLevel. Thus, Magic_SameLevel contains each node that is on the path from *spoke* to the root in Figure 27.2. The Magic_SameLevel table can be used as a filter to restrict the computation:

```
SameLevel(S1, S2) :-  Magic_SameLevel(S1),
                      Assembly(P1, S1, Q1), Assembly(P2, S2, Q2).
SameLevel(S1, S2) :-  Magic_SameLevel(S1), Assembly(P1, S1, Q1),
                      SameLevel(P1, P2), Assembly(P2, S2, Q2).
```

These rules together with the rules defining Magic_SameLevel give us a program for computing all SameLevel tuples with *spoke* in the first column. Notice that the new program depends on the query constant *spoke* only in the second rule defining Magic_SameLevel. Thus, the program for computing all SameLevel tuples with *seat* in the first column, for instance, is identical except that the second Magic_SameLevel rule is:

```
Magic_SameLevel(seat) :- .
```

The number of inferences made using the 'Magic' program can be far fewer than the number of inferences made using the original program, depending on just how much the selection in the query restricts the computation.

The Magic Sets Algorithm

We illustrated the **Magic Sets** algorithm on the SameLevel program, which contains just one output relation and one recursive rule. The algorithm, however, can be applied to any Datalog program. The input to the algorithm consists of the program and a **query form**, which is a relation that we want to query plus the fields of the query relation for which a query will provide constants. The output of the algorithm is a rewritten program. When the rewritten program is evaluated, the constants in the query are used to restrict the computation.

The Magic Sets program rewriting algorithm can be summarized as follows:

1. **Add 'Magic' filters:** Modify each rule in the program by adding a 'Magic' condition to the body that acts as a filter on the set of tuples generated by this rule.
2. **Define the 'Magic' relations:** We must create new rules to define the 'Magic' relations. Intuitively, from each occurrence of an output relation R in the body of a program rule, we obtain a rule defining the relation Magic_R.

When a query is posed, we add the corresponding 'Magic' tuple to the rewritten program and evaluate the least fixpoint of the program.

We remark that the Magic Sets algorithm has turned out to be quite effective for computing correlated nested SQL queries, even if there is no recursion, and is used for this purpose in many commercial DBMSs even though these systems do not currently support recursive queries.

27.5 POINTS TO REVIEW

- It is not possible to write recursive rules in SQL-92, but SQL:1999 supports recursion. A *Datalog program* consists of a collection of *rules*. A rule consists of a *head* and a *body*. DBMSs that support Datalog are called *deductive database systems* since the rules are applied iteratively to *deduce* new tuples. **(Section 27.1)**

- Relations in Datalog are either defined by rules (*output relations*) or have tuples explicitly listed (*input relations*). The meaning of a Datalog program can be defined either through *least model semantics* or through *least fixpoint semantics*. Least model semantics is declarative. A *model* of a program is a collection of relations that is consistent with the input relations and the Datalog program. A model that is contained in every other model is called a *least model*. There is always a least model for a Datalog program without negation, and this model is defined to be the meaning of the program. Least fixpoint semantics is operational. A *fixpoint* of a function is a value v such that $f(v) = v$. The least fixpoint is a fixpoint that is smaller than every other fixpoint. If we consider Datalog programs without negation, every program has a least fixpoint and the least fixpoint is equal to the least model. **(Section 27.2)**

- We say that a table T *depends on* a table S if some rule with T in the head contains S, or (recursively) contains a predicate that depends on S, in the body. If a Datalog program contains **not**, it can have more than one least fixpoint. We can syntactically restrict ourselves to *stratified programs*, for which there is a least fixpoint (from among the many fixpoints that exist for the program) that corresponds closely to an intuitive reading of the program. In a stratified program, the relations can be classified into numbered layers called *strata* such that a relation in stratum k only depends on relations in strata less than k. Datalog can be extended with grouping and aggregation operations. Unrestricted use of aggregation can also result in programs with more than one least fixpoint, and we can again restrict ourselves to stratified programs to get natural query results. **(Section 27.3)**

- Straightforward evaluation of recursive queries by repeatedly applying the rules leads to *repeated inferences* (the same tuples are inferred repeatedly by the same rule) and unnecessary inferences (tuples that do not contribute to the desired

output of the query). We call one application of all rules using all tuples generated so far an *iteration*. Simple repeated application of the rules to all tuples in each iteration is also called *Naive fixpoint evaluation*. We can avoid repeated inferences using *Seminaive fixpoint evaluation*. Seminaive fixpoint evaluation only applies the rules to tuples that were newly generated in the previous iteration. To avoid unnecessary inferences, we can add filter relations and modify the Datalog program according to the *Magic Sets program rewriting algorithm*. **(Section 27.4)**

EXERCISES

Exercise 27.1 Consider the Flights relation:

Flights(*flno:* `integer`, *from:* `string`, *to:* `string`, *distance:* `integer`, *departs:* `time`, *arrives:* `time`)

Write the following queries in Datalog and SQL3 syntax:

1. Find the *flno* of all flights that depart from Madison.
2. Find the *flno* of all flights that leave Chicago after Flight 101 arrives in Chicago and no later than one hour after.
3. Find the *flno* of all flights that do not depart from Madison.
4. Find all cities reachable from Madison through a series of one or more connecting flights.
5. Find all cities reachable from Madison through a chain of one or more connecting flights, with no more than one hour spent on any connection. (That is, every connecting flight must depart within an hour of the arrival of the previous flight in the chain.)
6. Find the shortest time to fly from Madison to Madras, using a chain of one or more connecting flights.
7. Find the *flno* of all flights that do not depart from Madison or a city that is reachable from Madison through a chain of flights.

Exercise 27.2 Consider the definition of Components in Section 27.1.1. Suppose that the second rule is replaced by

```
Components(Part, Subpart) :-  Components(Part, Part2),
                              Components(Part2, Subpart).
```

1. If the modified program is evaluated on the Assembly relation in Figure 27.1, how many iterations does Naive fixpoint evaluation take, and what Components facts are generated in each iteration?
2. Extend the given instance of Assembly so that Naive fixpoint iteration takes two more iterations.
3. Write this program in SQL3 syntax, using the `WITH` clause.

4. Write a program in Datalog syntax to find the part with the most distinct subparts; if several parts have the same maximum number of subparts, your query should return all of these parts.
5. How would your answer to the previous part be changed if you also wanted to list the number of subparts for the part with the most distinct subparts?
6. Rewrite your answers to the previous two parts in SQL3 syntax.
7. Suppose that you want to find the part with the most subparts, taking into account the quantity of each subpart used in a part, how would you modify the Components program? (*Hint:* To write such a query you reason about the number of inferences of a fact. For this, you have to rely on SQL's maintaining as many copies of each fact as the number of inferences of that fact and take into account the properties of Seminaive evaluation.)

Exercise 27.3 Consider the definition of Components in Exercise 27.2. Suppose that the recursive rule is rewritten as follows for Seminaive fixpoint evaluation:

```
Components(Part, Subpart) :-  delta_Components(Part, Part2, Qty),
                              delta_Components(Part2, Subpart).
```

1. At the end of an iteration, what steps must be taken to update *delta*_Components to contain just the new tuples generated in this iteration? Can you suggest an index on Components that might help to make this faster?
2. Even if the *delta* relation is correctly updated, fixpoint evaluation using the preceding rule will not always produce all answers. Show an instance of Assembly that illustrates the problem.
3. Can you suggest a way to rewrite the recursive rule in terms of *delta*_Components so that Seminaive fixpoint evaluation always produces all answers and no inferences are repeated across iterations?
4. Show how your version of the rewritten program performs on the example instance of Assembly that you used to illustrate the problem with the given rewriting of the recursive rule.

Exercise 27.4 Consider the definition of SameLevel in Section 27.4.2 and the Assembly instance shown in Figure 27.1.

1. Rewrite the recursive rule for Seminaive fixpoint evaluation, and show how Seminaive evaluation proceeds.
2. Consider the rules defining the relation Magic, with *spoke* as the query constant. For Seminaive evaluation of the 'Magic' version of the SameLevel program, all tuples in Magic are computed first. Show how Seminaive evaluation of the Magic relation proceeds.
3. After the Magic relation is computed, it can be treated as a fixed database relation, just like Assembly, in the Seminaive fixpoint evaluation of the rules defining SameLevel in the 'Magic' version of the program. Rewrite the recursive rule for Seminaive evaluation and show how Seminaive evaluation of these rules proceeds.

BIBLIOGRAPHIC NOTES

The use of logic as a query language is discussed in several papers in [254, 466], which arose out of influential workshops. Good textbook discussions of deductive databases can be found in [656, 3, 122, 695, 438]. [535] is a recent survey article that provides an overview and covers the major prototypes in the area, including LDL [147], Glue-Nail! [478] and [180], EKS-V1 [666], Aditi [536], Coral [534], LOLA [705], and XSB [561].

The fixpoint semantics of logic programs (and deductive databases as a special case) is presented in [659], which also shows equivalence of the fixpoint semantics to a *least-model* semantics. The use of stratification to give a natural semantics to programs with negation was developed independently in [30, 131, 488, 660].

Efficient evaluation of deductive database queries has been widely studied, and [48] is a survey and comparison of several early techniques; [533] is a more recent survey. Seminaive fixpoint evaluation was independently proposed several times; a good treatment appears in [44]. The Magic Sets technique was proposed in [47] and was generalized to cover all deductive database queries without negation in [64]. The Alexander method [549] was independently developed and is equivalent to a variant of Magic Sets called *Supplementary Magic Sets* in [64]. [482] showed how Magic Sets offers significant performance benefits even for nonrecursive SQL queries. [586] describes a version of Magic Sets designed for SQL queries with correlation, and its implementation in the Starburst system (which led to its implementation in IBM's DB2 DBMS). [583] discusses how Magic Sets can be incorporated into a System R style cost-based optimization framework. The Magic Sets technique is extended to programs with stratified negation in [63, 43]. [102] compares Magic Sets with top-down evaluation strategies derived from Prolog.

[559] develops a program rewriting technique related to Magic Sets called *Magic Counting*. Other related methods that are not based on program rewriting but rather on run-time control strategies for evaluation include [191, 367, 664, 665]. The ideas in [191] have been developed further to design an *abstract machine* for logic program evaluation using tabling in [638] and [531]; this is the basis for the XSB system [561].