22 Dynamic Aspects

Alice:	*How come we've waited so long to talk about something so important?*
Riccardo:	*Talking about change is hard.*
Sergio:	*We're only starting to get a grip on it.*
Vittorio:	*And still have a long way to go.*

At a fundamental level, updating a database is essentially imperative programming. However, the persistence, size, and long life cycle of a database lead to perspectives somewhat different from those found in programming languages. In this chapter, we briefly examine some of these differences and sketch some of the directions that have been explored in this area. Although it is central to databases, this area has received far less attention from the theoretical research community than other topics addressed in this book. The discussion in this chapter is intended primarily to give an overview of the important issues raised concerning the dynamic aspects of databases. It therefore emphasizes examples and intuitions much more than results and proofs.

This chapter begins by examining database update languages, including a simple language that corresponds to the update capabilities of practical languages such as SQL, and more complex ones expressed within a logic-based framework. Next optimization and semantic properties of transactions built from simple update commands are considered, including a discussion of the interaction of transactions and static integrity constraints.

The impact of updates in richer contexts is then considered. In connection with views, we examine the issue of how to propagate updates incrementally from base data to views and the much more challenging issue of propagating an update on a view back to the base data. Next updates for incomplete information databases are considered. This includes both the conditional tables studied in Chapter 19 and more general frameworks in which databases are represented using logical theories.

The emerging field of active databases is then briefly presented. These incorporate mechanisms for automatically responding to changes in the environment or the database, and they often use a rule-based paradigm of specifying the responses.

This chapter concludes with a brief discussion of temporal databases, which support the explicit representation of the time dimension and thus historical information.

A broad area related to dynamic aspects of databases (namely, concurrency control) will not be addressed. This important area concerns mechanisms to increase the throughput of a database system by interleaving multiple transactions while guaranteeing that the semantics of the individual transactions is not lost.

22.1 Update Languages

Before embarking on a brief excursion into update languages, we should answer the following natural question: Why are update languages necessary? Could we not use query languages to specify updates?

The difference between query and update languages is subtle but important. To specify an update, we could indeed define the new database as the answer to a query posed against the old database. However, this misses an essential characteristic of updates: Most often, they involve small changes to the current database. Query languages are not naturally suited to speak explicitly about *change*. In contrast, update languages use as building blocks simple statements expressing change, such as insertions, deletions, and modifications of tuples in the database.

In this section, we outline several formal update languages and point to some theoretical issues that arise in this context.

Insert-Delete-Modify Transactions

We begin with a simple procedural language to specify insertions, deletions, and modifications. Most commercial relational systems provide at least these update capabilities.

To simplify the presentation, we suppose that the database consists of a single relation schema R. Everything can be extended to the multirelational case. An *insertion* is an expression $ins(t)$, where t is a tuple over $att(R)$. This inserts the tuple t into R. [We assume set-based semantics, under which $ins(t)$ has no effect if t is already present in R.] A deletion removes from R all tuples satisfying some stated set of conditions. More precisely, a *condition* is an (in)equality of the form $A = c$ or $A \neq c$, where $A \in att(R)$ and c is a constant. A *deletion* is an expression $del(C)$, where C is a finite set of conditions. This removes from R all tuples satisfying each condition in C. Finally, a *modification* is an expression $mod(C \to C')$, where C, C' are sets of conditions, with C' containing only equalities $A = c$. This selects all tuples in R satisfying C and then, for each such tuple and each $A = c$ in C', sets the value of A to c. An *update* over R is an insertion, deletion, or modification over R. An IDM transaction (for insert, delete, modify) over R is a finite sequence of updates over R. This is illustrated next.

Example 22.1.1 Consider the relation schema *Employee* with attributes *N (Name), D (Department), R (Rank).* The following IDM transaction fires the manager of the parts department, transfers the manager of the sales department to the parts department, and hires Moe as the new manager for the sales department:

$$del(\{D = parts, R = manager\});$$
$$mod(\{D = sales, R = manager\} \to \{D = parts\});$$
$$ins(Moe, sales, manager)$$

The same update can be expressed in SQL as follows:

delete from Employee

```
        where D = "parts" and R = "manager";
update  Employee
        set D = "parts"
        where D = "sales" and R = "manager";
insert  into Employee values ⟨ "Moe","sales","manager"⟩
```

As for queries, a question of central interest to update languages is optimization. To see how IDM transactions can be optimized, it is useful to understand when two such transactions are equivalent. It turns out that equivalence of IDM transactions has a sound and complete axiomatization. Following are some simple axioms:

$$\begin{aligned} &mod(C \rightarrow C'); del(C') &&\equiv del(C); del(C') \\ &ins(t); mod(C \rightarrow C') &&\equiv mod(C \rightarrow C'); ins(t') \\ &\text{where } t \text{ satisfies } C \text{ and } \{t'\} = mod(C \rightarrow C')(\{t\}) \end{aligned}$$

and a slightly more complex one:

$$\begin{aligned} &del(C_3); mod(C_1 \rightarrow C_3); mod(C_2 \rightarrow C_1); mod(C_3 \rightarrow C_2) \\ &\qquad \equiv del(C_3); mod(C_2 \rightarrow C_3); mod(C_1 \rightarrow C_2); mod(C_3 \rightarrow C_1), \end{aligned}$$

where C_1, C_2, C_3 are mutually exclusive sets of conditions.

We can define criteria for the optimization of IDM transactions along two main lines:

Syntactic: We can take into account the length of the transaction as well as the kind of operations involved (for example, it may be reasonable to assume that insertions are simpler than modifications).

Semantic: This can be based on the number of tuple operations actually performed when the transaction is applied.

Various definitions are possible based on the preceeding criteria. It can be shown that there exists a polynomial-time algorithm that optimizes IDM transactions, with respect to a reasonable definition based on syntactic and semantic criteria. The syntactic criteria involve the number of insertions, deletions, and modifications. The semantic criteria are based on the number of tuples touched at runtime by the transaction. We omit the details here.

EXAMPLE 22.1.2 Consider the IDM transaction over a relational schema R of sort AB:

$$\begin{aligned} &mod(\{A \neq 0, B = 1\} \rightarrow \{B = 2\}); ins(0, 1); ins(3, 2); \\ &mod(\{A = 0, B = 1\} \rightarrow \{B = 2\}); mod(\{A \neq 0, B = 0\} \rightarrow \{B = 1\}); \\ &mod(\{A = 0, B = 0\} \rightarrow \{B = 1\}); mod(\{A \neq 0, B = 2\} \rightarrow \{B = 0\}); \\ &mod(\{A = 0, B = 2\} \rightarrow \{B = 0\}); del(\{A \neq 0, B = 0\}). \end{aligned}$$

Assuming that insertions are less expensive than deletions, which are less expensive than modifications, an optimal IDM transaction equivalent to the foregoing is

$$\begin{aligned}&del(\{A \neq 0, B = 1\}); del(\{A \neq 0, B = 2\});\\ &mod(\{A = 0, B = 1\} \to \{B = 2\});\\ &mod(\{B = 0\} \to \{B = 1\});\\ &mod(\{A = 0, B = 2\} \to \{B = 0\});\\ &ins(0, 0).\end{aligned}$$

Thus the six modifications, one deletion, and two insertions of the original transaction were replaced by three modifications, two deletions, and one insertion.

Another approach to optimization is to turn some of the axioms of equivalence into simplification rules, as in

$$mod(C \to C'); del(C') \Rightarrow del(C); del(C').$$

It can be shown that such a set of simplification rules can be used to optimize a restricted set of IDM transactions that satisfy a syntactic acyclicity condition. For the other transactions, applications of the simplification rules yield a simpler, but not necessarily optimal, transaction. The simplification rules have the advantage that they are local and can be easily applied even online, whereas the complete optimization algorithm is global and has to know the entire transaction in advance.

Rule-Based Update Languages

The IDM transactions provide a simple update language of limited power. This can be extended in many ways. One possibility is to build another procedural language based on tuple insertions, deletions, and modifications, which includes relation variables and an iterative construct. Another, which we illustrate next, is to use a rule-based approach. For example, consider the language datalog$^{\neg\neg}$ described in Chapter 17, with its fixpoint semantics. Recall that rules allow for both positive and negative atoms in heads of rules; consistently with the fixpoint semantics, the positive atoms can be viewed as insertions of facts and the negative atoms as deletions of facts. For example, the following program removes all cycles of length one or two from the graph G:

$$\neg G(x, y) \leftarrow G(x, y), G(y, x).$$

In the usual fixpoint semantics, rules are fired in parallel with all possible instantiations for the variables. This yields a deterministic semantics. Some practical rule-based update languages take an alternative approach, which yields a nondeterministic semantics: The rules are fired one instantiation at a time. With this semantics, the preceeding program provides *some* orientation of the graph G. Note that generally there is no way to obtain an orientation of a graph deterministically, because a nondeterministic choice of edges to be removed may be needed.

A deterministic language expressing *all* updates can be obtained by extending datalog$^{\neg\neg}$ with the ability to invent new values, in the spirit of the language $while_{new}$

in Chapter 18. This can be done in the manner described in Exercise 18.22. The same language with nondeterministic semantics can be shown to express all nondeterministic updates.

The aforementioned languages yield a bottom-up evaluation procedure. The body of the rule is first checked, and then the actions in the head are executed. Another possibility is to adopt a top-down approach, in the spirit of the *assert* in Prolog. Here the actions to be taken are specified in rule bodies. A good example of this approach is provided by *Dynamic Logic Programming* (DLP). Interestingly, this language allows us to test hypothetical conditions of the form "Would φ hold if t was inserted?" This, and the connection of DLP with Prolog, is illustrated next.

Example 22.1.3 Consider a database schema with relations *ES* of sort *Emp,Sal* (employees and their salaries), *ED* of sort *Emp,Dep* (employees and their departments), and *DA* of sort *Dep,Avg* (average salary in each department).

Suppose that an update is intended to hire John in the toys department with a salary of $200K$, under the condition that the average salary of the department stays below $50K$. In the language DLP, this update is expressed by

$$\begin{array}{l} \langle hire(emp1, sal1, dep1)\rangle \leftarrow \\ \qquad \langle +ES(emp1, sal1)\rangle(\langle +ED(emp1, dep1)\rangle(DA(dep1, avg1) \ \ \& \ \ avg1 < 50k)). \end{array}$$

(Other rules are, of course, needed to define *DA*.) A call *hire*(John,200K,Toys) hires John in the toys department only if, after hiring him, the average salary of the department remains below $50K$. The $+$ symbol indicates an insertion. Here the conditions in parentheses should hold after the two insertions have been performed; if not, then the update is not realized. Testing a condition under the assumption of an update is a form of hypothetical reasoning.

It is interesting to contrast the semantics of DLP with that of Prolog. Consider the following Prolog program:

$$\begin{array}{ll} :- & assert(ES(john, 200)), assert(ED(john, toys)), \\ & \qquad DA(toys, Avg1), Avg1 < 50. \end{array}$$

In this program, the insertions into *ES* and *ED* will be performed even if the conditions are not satisfied afterward. (The reader familiar with Prolog is encouraged to write a program that has the desired semantics.)

A similar top-down approach to updates is adopted in Logical Data Language (LDL).

Updates concern not only instances of a fixed schema. Sometimes the schema itself needs to be changed (e.g., by adding an attribute). Some practical update languages include constructs for schema change. The main problem to be resolved is how the existing data can be fit to the new schema.

In deductive databases, some relations are defined using rules. Occasionally these definitions may have to be changed, leading to updates of the "rule base." There are languages that can be used to specify such updates.

22.2 Transactional Schemas

Typically, database systems restrict the kinds of updates that users can perform. There are three main ways of doing this:

(a) Specify constraints (say, fd's) that the database must satisfy and reject any update that leads to a violation of the constraints.

(b) Restrict the updates themselves by only allowing the use of a set of prespecified, valid updates.

(c) Permit users to request essentially arbitrary updates, but provide an automatic mechanism for detecting and repairing constraint violations.

Object-oriented databases essentially embrace option (b); updates are performed only by *methods* specified at the schema level, and it is assumed that these will not violate the constraints (see Chapter 21). Both options (a) and (b) are present in the relational model. Several commercial systems can recognize and abort on violation of simple constraints (typically key and simple inclusion dependencies). However, maintenance of more complex constraints is left to the application software. Option (c) is supported by the emerging field of active databases, which is discussed in the following section.

We now briefly explore some issues related to approach (b) in connection with the relational model. To illustrate the issues, we use simple procedures based on IDM transactions. The procedures we use are *parameterized IDM transactions*, obtained by allowing variables in addition to constants in conditions of IDM transactions. The variables are used as parameters. A database schema **R** together with a finite set of parameterized IDM transactions over **R** is called an *IDM transactional schema*.

Example 22.2.1 Consider a database schema **R** with two relations, *TA* (Teaching Assistant) of sort *Name*,*Course*, and *PHD* (Ph.D. student) of sort *Name, Address*. The following IDM-parameterized transactions allow the hiring and firing of TAs (subscripts indicate the relation to which each update applies):

$$\begin{aligned} hire(x, y, z) &= del_{TA}(Name = x);\ ins_{TA}(x, y) \\ &\qquad del_{PHD}(Name = x);\ ins_{PHD}(x, z) \\ fire(x) &= del_{TA}(Name = x) \end{aligned}$$

The pair $\mathbf{T} = \langle \mathbf{R}, \{hire, fire\}\rangle$ is an IDM transactional schema. Note in this simple example that once a name n is incorporated into the *PHD* relation, it can never be removed.

Clearly, we could similarly define transactional schemas in conjunction with any update language.

Suppose **T** is an IDM transactional schema. To apply the parameterized transactions, values must be supplied to the variables. A transaction obtained by replacing the variables of a parameterized transaction t in **T** by constants is a *call* to t. The only updates allowed by an IDM transactional schema are performed by calls to its parameterized transactions.

The set of instances that can be generated by such calls (starting from the empty instance) is denoted $Gen(\mathbf{T})$.

Transactional schemas offer an approach for constraint enforcement, essentially by preventing updates that violate them. So it is important to understand to what extent they can do so. First we need to clarify the issue. Suppose $\mathbf{T}$ is an IDM transactional schema and Σ is a set of constraints over a database schema $\mathbf{R}$; $Sat(\Sigma)$ denotes all instances over $\mathbf{R}$ satisfying Σ. If $\mathbf{T}$ is to replace Σ, we would expect the following properties to hold:

- *soundness* of $\mathbf{T}$ with respect to Σ: $Gen(\mathbf{T}) \subseteq Sat(\Sigma)$; and
- *completeness* of $\mathbf{T}$ with respect to Σ: $Gen(\mathbf{T}) \supseteq Sat(\Sigma)$.

Thus $\mathbf{T}$ is sound and complete with respect to Σ iff it generates precisely the instances satisfying Σ.

Example 22.2.2 Consider again the IDM transactional schema $\mathbf{T}$ in Example 22.2.1. Let Σ be the following constraints:

$$\begin{aligned} &TA : Name \to Course \\ &PHD : Name \to Address \\ &TA[Name] \subseteq PHD[Name] \end{aligned}$$

It is easily seen that $\mathbf{T}$ in Example 22.2.1 is sound and complete with respect to Σ. That is, $Gen(\mathbf{T}) = Sat(\Sigma)$ (Exercise 22.7).

This example also highlights a limitation in the notion of completeness: It can be seen that there are pairs $\mathbf{I}$ and $\mathbf{J}$ of instances in $Sat(\Sigma)$ where $\mathbf{I}$ cannot be transformed into $\mathbf{J}$ using $\mathbf{T}$. In other words, there are valid database states $\mathbf{I}$ and $\mathbf{J}$ such that when in state $\mathbf{I}$, $\mathbf{J}$ is never reachable. Such forbidden transitions are also a means of enriching the model, because we can view them as temporal constraints on the database evolution. We will return to temporal constraints later in this chapter.

Of course, the ability of transaction schemas to replace constraints depends on the update language used. For IDM transactional schemas, we can show the following (Exercise 22.8):

Theorem 22.2.3 For each database schema $\mathbf{R}$ and set Σ of fd's and acyclic inclusion dependencies over $\mathbf{R}$, there exists an IDM transactional schema $\mathbf{T}$ that is sound and complete with respect to Σ.

Thus IDM transactional schemas are capable of replacing a significant set of constraints. The kind of difficulty that arises with more general constraints is illustrated next.

Example 22.2.4 Consider a relation R of sort ABC and the following set Σ of constraints:

- the embedded join dependency

$$\forall xyzx'y'z'(R(xyz) \wedge R(x'y'z') \Rightarrow \exists z'' R(xy'z')),$$

- the functional dependency AB $\rightarrow$ C,
- the inclusion dependency R[A] $\subseteq$ R[C],
- the inclusion dependency R[B] $\subseteq$ R[A],
- the inclusion dependency R[A] $\subseteq$ R[B].

It is easy to check that, for each relation satisfying the constraints, the number of constants in the relation is a perfect square (n^2, $n \geq 0$). Thus there are unbounded gaps between instances in $Sat(\Sigma)$. There is no IDM transactional schema **T** such that $Sat(\Sigma) = Gen(\mathbf{T})$, because the gaps cannot be crossed using calls to parameterized transactions with a bounded number of parameters. Moreover, this problem is not specific to IDM transactional schemas; it arises with any language in which procedures can only introduce a bounded number of new constants into the database at each call.

Another natural question relating updates and constraints is, What about *checking* soundness and/or completeness of IDM transactional schemas with respect to given constraints? Even in the case of IDM transactional schemas, such questions are generally undecidable. There is one important exception: Soundness of IDM transactional schemas with respect to fd's is decidable. These questions are explored in Exercise 22.12.

22.3 Updating Views and Deductive Databases

We now turn to the impact of updates on views. Views are an important aspect of databases. The interplay between views and updates is intricate. We can mention in particular two important issues. One is the *view maintenance* problem: A view has been materialized and the problem is to maintain it incrementally when the database is updated. An important variation of this is in the context of deductive databases when the view consists of idb relations. The other is known as the *view update* problem: Given a view and an update against a view, the problem is to translate the update into a corresponding update against the base data. This section considers these two issues in turn.

View Maintenance

Suppose that a base schema **B** and view schema **V** are given along with a (total) view mapping $f : Inst(\mathbf{B}) \rightarrow Inst(\mathbf{V})$. Suppose further that a materialized view is to be maintained [i.e., whenever the base database holds an instance $\mathbf{I}_B$, then the view schema should be holding $f(\mathbf{I}_B)$].

For this discussion, an *update* for a schema **R** is considered to be a mapping from $Inst(\mathbf{R})$ to $Inst(\mathbf{R})$. If constraints are present, it is assumed that an update cannot map to instances violating the constraints. The updates considered here might be based on IDM transactions or might be more general. We shall often speak of "the" update μ that maps

$$\begin{tikzcd}
\mathbf{I}_V \arrow[r, "\nu"] & \mathbf{I}'_V \\
\mathbf{I}_B \arrow[u, "f"] \arrow[r, "\mu"'] & \mathbf{I}'_B \arrow[u, "f"']
\end{tikzcd}$$

Figure 22.1: Relationship of views and updates

instance $\mathbf{I}$ to instance $\mathbf{I}'$, and by this we shall mean the set of insertions and deletions that need to be made to $\mathbf{I}$ to obtain $\mathbf{I}'$.

Suppose that the base database $\mathbf{B}$ is holding $\mathbf{I}_B$ and that update μ maps this to $\mathbf{I}'_B$ (see Fig. 22.1). A naive way to keep the view up to date is to simply compute $f(\mathbf{I}'_B)$. However, $\mathbf{I}'_B$ is typically large relative to the difference between $\mathbf{I}_V$ and $\mathbf{I}'_V$. It is thus natural to search for more efficient ways to find the update ν that maps $\mathbf{I}_V$ to $\mathbf{I}'_V = f(\mu(\mathbf{I}_B))$. This is the *view maintenance problem*.

There are generally two main components to solutions of the view maintenance problem. The first involves developing algorithms to test whether an update to the base data can affect the view. Given such an algorithm, an update is said to be *irrelevant* if the algorithm certifies that the update cannot affect the view, and it is said to be *relevant* otherwise.

EXAMPLE 22.3.1 Let the base database schema be $\mathbf{B} = (R[AB], S[BC])$, and consider the following views:

$$\begin{aligned}
V_1 &= (R \bowtie \sigma_{C>50} S) \\
V_2 &= \pi_A R \\
V_3 &= R \bowtie S \\
V_4 &= \pi_{AC}(R \bowtie S).
\end{aligned}$$

Inserting $\langle b, 20\rangle$ into S cannot affect views V_1 or V_2. On the other hand, whether or not this insertion affects V_3 or V_4 depends on the data already present in the database.

Various algorithms have been developed for determining relevance with varying degrees of precision. A useful technique involves maintaining auxiliary information, as illustrated next.

EXAMPLE 22.3.2 Recall view V_2 of Example 22.3.1, and suppose that R currently holds

R	A	B
	a	20
	a	30
	a'	80

Deleting $\langle a, 20\rangle$ has no impact on the view, whereas deleting $\langle a', 80\rangle$ has the effect of deleting $\langle a'\rangle$ from the view. One way to monitor this is to maintain a count on the number of distinct ways that a value can arise; if this count ever reaches 0, then the value should be deleted from the view.

The other main component of solutions to the view maintenance problem concerns the development of *incremental evaluation algorithms*. This is closely related to the seminaive algorithm for evaluating datalog programs (see Chapter 13).

Example 22.3.3 Recall view V_3 from Example 22.3.1, and let Δ_R^+ and Δ_S^+ denote sets of tuples that are to be inserted into R and S, respectively. It is easily verified that

$$(R \cup \Delta_R^+) \bowtie (S \cup \Delta_S^+) = (R \bowtie S) \cup (R \bowtie \Delta_S^+) \cup (\Delta_R^+ \bowtie S) \cup (\Delta_R^+ \bowtie \Delta_S^+).$$

Thus the new join can be found by performing three (typically smaller) joins followed by some unions.

It is relatively straightforward to develop incremental evaluation expressions, such as in the preceeding example, for all of the relational algebra operators (see Exercise 22.13). In some cases, these expressions can be refined by using information about constraints, such as key and functional dependencies, on the base data.

Incremental Update of Deductive Views

The view maintenance problem has also been studied in connection with views constructed with (stratified) datalog$^{(\neg)}$. In general, the techniques used are analogous to those discussed earlier but are generalized to incorporate recursion. In the context of stratified datalog$^\neg$, various heuristics have been adapted from the field of belief revision for incrementally maintaining supports (i.e., auxiliary information that holds the justifications for the presence of a fact in the materialized output of the program).

An interesting research direction that has recently emerged focuses on the ability of first-order queries to express incremental updates on views defined using datalog. The framework for these problems is as follows. The base schema $\mathbf{B}$ and view schema $\mathbf{V}$ are as before, except that $\mathbf{V}$ contains only one relation and the view f is defined in terms of a datalog program P. A basic question is, Given P, is there a first-order query φ such that $\varphi(\mathbf{I}_B, \mathbf{I}_V, +R(t)) = P(\mathbf{I}_B \cup \{R(t)\})$ for each choice of $\mathbf{I}_B$, $\mathbf{I}_V = P(\mathbf{I}_B)$ and insertion $+R(t)$ where $R \in \mathbf{B}$? If this holds, then P is said to be *first-order incrementally definable* (FOID) (without auxiliary relations).

Example 22.3.4 Consider a binary relation $G[AB]$ and the usual datalog program P that computes the transitive closure of G in $T[AB]$. Suppose that I is an instance of G, and J is $P(I)$. Suppose that tuple $\langle a, b\rangle$ is inserted into I. Then a tuple $\langle a', b'\rangle$ will be inserted into J iff one of the following occurs:

(a) $a' = a$ and $b = b'$;

(b) $a' = a$ and $\langle b, b'\rangle \in J$;

(c) $\langle a', a\rangle \in J$ and $b = b'$; or

(d) $\langle a', a\rangle \in J$ and $\langle b, b'\rangle \in J$.

The preceeding conditions can clearly be specified by a first-order query. It easily follows that P is FOID (see Exercise 22.21).

Several variations of FOIDs have been studied. These include FOIDs with auxiliary relations (i.e., that permit the maintenance of derived relations not in the original datalog program) and FOIDs that support incremental updates for sets of insertions and/or deletions. FOIDs have been found for a number of restricted classes of datalog programs. However, it remains open whether there is a datalog program that is not FOID with auxiliary relations.

Basic Issues in View Update

The view update problem is essentially the inverse of the view maintenance problem. Referring again to Fig. 22.1, the problem now is, Given $\mathbf{I}_B$, $\mathbf{I}_V$, and update ν on $\mathbf{I}_V$, find an update μ so that the diagram commutes.

The first obvious problem here is the potential for ambiguity.

Example 22.3.5 Recall the view V_2 of Example 22.3.1. Suppose that the base value of R is $\{\langle a, b\rangle\}$ (and the base value of S is $\emptyset$). Thus the view holds $\{\langle a\rangle\}$. Now consider an update ν to the view that inserts $\langle a'\rangle$. Some possible choices for μ include

(a) Insert $\langle a', b\rangle$ into R.

(b) Insert $\langle a', b'\rangle$ into R for some $b' \in$ **dom**.

(c) Insert $\{\langle a', b'\rangle \mid b' \in X\}$ into R, where X is a finite subset of **dom**.

(d) Insert $\langle a', b'\rangle$ into R for some $b' \in$ **dom**, and replace $\langle a, b\rangle$ by $\langle a, b'\rangle$.

Possibility (d) seems undesirable, because it affects a tuple in a base relation that is, intuitively speaking, independent of the view update. Possibilities (a) and (b) seem more appealing than (c), but (c) cannot be ruled out. In any case, it is clear that there are a large number of updates μ that correspond to ν.

The fundamental problem, then, is how to select one update μ to the base data given that many possibilities may exist. One approach to resolving the ambiguity involves examining the intended semantics of the database and the view.

Example 22.3.6 Consider a schema *Employee*[*Name*, *Department*, *Team_position*], which records an employee's department and the position he or she plays in the corporate baseball league. It is assumed that *Name* is a key. The value "no" indicates that the employee does not play in the league. It is assumed that *Name* is a key. Consider the views defined by

$$\begin{aligned} Sales &= \sigma_{Department=\text{“Sales”}}(Employee) \\ Baseball &= \pi_{Employee,Team\_position}(\sigma_{Team\_position \neq \text{“no”}}(Employee)) \end{aligned}$$

Typically, if tuple ⟨"Joe", "Sales", "shortstop"⟩ is deleted from the *Sales* view, then this tuple should also be deleted from the underlying *Employee* relation. In contrast, if tuple ⟨"Joe", "shortstop"⟩ is deleted from the *Baseball* view, it is typically most natural to replace the underlying tuple ⟨"Joe", d, "shortstop"⟩ in *Employee* by ⟨"Joe", d, "no"⟩ (i.e., to remove Joe from the baseball league rather than forcing him out of the company).

As just illustrated, the correct translation of a view update can easily depend on the semantics associated with the view as well as the syntactic definition. Research in this area has developed notions of update translations that perform a *minimal* change to the underlying database. Algorithms that generate families of acceptable translations of views have been developed, so that the database administrator may choose at view definition time the most appropriate one.

Another issue in view update is that a requested update may not be permitted on the view, essentially because of constraints implicit to the view definition and algorithm for choosing translations of updates.

Example 22.3.7 Recall the view V_4 of Example 22.3.1, and suppose that the base data is

R	A	B
	a	20
	a'	20

S	B	C
	20	c
	20	c'

In this case the view contains $\{\langle a, c\rangle, \langle a, c'\rangle, \langle a', c\rangle, \langle a', c'\rangle\}$.

Suppose that the user requests that $\langle a, c\rangle$ be deleted. Typically, this deletion is mapped into one or more deletions against the base data. However, deleting $R(a, 20)$ results in a side-effect (namely, the deletion of $\langle a, c'\rangle$ from the view). Deletion of $S(20, c)$ also yields a side-effect.

Formal issues surrounding such side-effects of view updates are largely unexplored.

Complements of Views

We now turn to a more abstract formulation of the view update problem. Although it is relatively narrow, it provides an interesting perspective.

In this framework, a *view* over a base schema $\mathbf{B}$ is defined to be a (total) function f from $Inst(\mathbf{B})$ into some set. In practice this set is typically $Inst(\mathbf{V})$ for some view schema $\mathbf{V}$; however, this is not required for this development. [The proof of Theorem 22.3.10, which presents a completeness result, uses a view whose range is not $Inst(\mathbf{V})$ for any schema $\mathbf{V}$.] A binary relation $\leq$ on views is defined so that $f \leq g$ if for all base instances $\mathbf{I}$ and $\mathbf{I}'$, $g(\mathbf{I}) = g(\mathbf{I}')$ implies $f(\mathbf{I}) = f(\mathbf{I}')$. Intuitively, $f \leq g$ if g can distinguish more instances that f. For view f, let $\equiv_f$ be the equivalence relation on $Inst(\mathbf{B})$ defined by $\mathbf{I} \equiv_f \mathbf{I}'$ iff $f(\mathbf{I}) = f(\mathbf{I}')$. It is clear that $f \leq g$ iff $\equiv_g$ is a refinement of $\equiv_f$ and thus $\leq$ can be viewed as a partial order on the equivalence relations over $Inst(\mathbf{B})$.

Two views f, g are *equivalent*, denoted $f \equiv g$, if $f \leq g$ and $g \leq f$. This is an equivalence relation on views. In the following, the focus is primarily on the equivalence classes under $\equiv$. Let $\top$ denote the view that is simply the identity, and let $\bot$ denote a view that maps every base instance to $\emptyset$. It is clear that (the equivalence classes represented by) $\top$ and $\bot$ are the maximal and minimal elements of the partial order $\leq$. We use cross-product as a binary operator to create views: The product of views f and g is defined so that $(f \times g)(\mathbf{I}) = (f(\mathbf{I}), g(\mathbf{I}))$. View g is a *complement* of view f if $f \times g \equiv \top$. Intuitively, this means that the base relations can be completely identified if both f and g are available. Clearly, each view f has a trivial complement: $\top$.

Example 22.3.8 (a) Let $\mathbf{B} = \{R[ABC]\}$ along with the fd $R : A \rightarrow B$, and consider the view $f = \pi_{AB}R$. Let $g = \pi_{AC}R$. It follows from Proposition 8.2.2 that g is a complement of f.

(b) Let $\mathbf{B} = \{R[AB]\}$ and $f = \pi_A R$. As mentioned earlier, $\top$ is a complement of f. It turns out that there are other complements of f, but they cannot be expressed using the relational algebra (see Exercise 22.25).

(c) Let $\mathbf{B} = \{Employee(Name, Salary, Bonus, Total\_pay)\}$, with the constraints that *Name* is a key and that for each tuple $\langle n, s, b, t\rangle$ in *Employee* we have $s + b = t$. Consider the view $f = \pi_{Name,Salary}(Employee)$. Consider the views

$$g_1 = \pi_{Name,Bonus}(Employee)$$
$$g_2 = \pi_{Name,Total\_pay}(Employee).$$

Both g_1 and g_2 are complements of f.

Thus each view has at least one complement (namely, $\top$) and may have more than one minimal complement.

In some cases, complements can be used to resolve ambiguity in the view update problem in the following way. Suppose that view f has complement g, and suppose that $\mathbf{I}_V = f(\mathbf{I}_B)$ and update ν on $\mathbf{I}_V$ are given. An update μ is a *g-translation* of ν if $f(\mu(\mathbf{I}_B)) = \nu(f(\mathbf{I}_B))$ and $g(\mu(\mathbf{I}_B)) = g(\mathbf{I}_B)$ (see Fig. 22.2). Intuitively, a g-translation

$$\begin{tikzcd}
(f(\mathbf{I}_B), g(\mathbf{I}_B)) \arrow[r, "\nu"] & (\nu(f(\mathbf{I}_B)), g(\mathbf{I}_B)) \arrow[d, "(f\times g)^{-1}"] \\
\mathbf{I}_B \arrow[u, "f\times g"] \arrow[r, "\mu"'] & \mathbf{I}'_B
\end{tikzcd}$$

Figure 22.2: Properties of a g-translation μ of view update ν on view f

accomplishes the update but leaves $g(\mathbf{I}_B)$ fixed. By the properties of complements, for an update ν there is at most one g-translation of ν.

Example 22.3.9 (a) Recall the base schema $\{R[ABC]\}$, view f, and complement g of Example 22.3.8(a). Suppose that $\langle a, b\rangle$ is in the view, and consider the update ν on the view that modifies $\langle a, b\rangle$ to $\langle a, b'\rangle$. The update μ defined to modify all tuples $\langle a, b, c\rangle$ of R into $\langle a, b', c\rangle$ is a g-translation of ν. On the other hand, given an insertion or deletion ν to the view, there is no g-translation of ν.

(b) Recall the base schema, view f, and complementary views g_1 and g_2 of Example 22.3.8(c). Suppose that $\langle Joe, 200, 50, 250\rangle$ is in *Employee*. Consider the update ν that replaces $\langle Joe, 200\rangle$ by $\langle Joe, 210\rangle$ in the view. Consider the updates

$$\mu_1 = \text{replace } \langle Joe, 200, 50, 250\rangle \text{ by } \langle Joe, 210, 50, 260\rangle$$
$$\mu_2 = \text{replace } \langle Joe, 200, 50, 250\rangle \text{ by } \langle Joe, 210, 40, 250\rangle.$$

Then μ_1 is the g_1-translation of ν, and μ_2 is the g_2-translation of ν.

Finally, we state a result showing that a restricted class of view updates can be translated into base updates using complementary views. To this end, we focus on *updates* of a schema $\mathbf{R}$ that are total functions from $Inst(\mathbf{R})$ to $Inst(\mathbf{R})$. A family U of updates on $\mathbf{R}$ is said to be *complete* if

(a) it is closed under composition (i.e., if μ and μ' are in U, then so is $\mu \circ \mu'$);

(b) it is closed under inverse in the following sense: $\forall \mathbf{I} \in inst(\mathbf{R})\ \forall \mu \in U\ \exists \mu' \in U$ such that $\mu'(\mu(\mathbf{I})) = \mathbf{I}$.

Intuitively, condition (b) says that a user can always undo an update just made. It is certainly natural to focus on complete sets of updates.

Let base schema $\mathbf{B}$ and view f be given, and let U_f be a family of updates on the view. Let U_B denote the family of all updates on the base schema. A *translator* for U_f is a mapping $t : U_f \to U_B$ such that for each base instance $\mathbf{I}_B$ and update $\nu \in U_f$, $f(t(\nu)(\mathbf{I}_B)) = \nu(f(\mathbf{I}_B))$. Clearly, solving the view update problem consists of coming up with a translator.

If g is a complement for f, then a translator t is a g-*translator* if $t(\nu)$ is a g-translation of ν for each $\nu \in U_f$.

We can now state the following (see Exercise 22.26):

Theorem 22.3.10 Let base schema **B** and view f be given, and let U_f be a complete set of updates on the view. Suppose that t is a translator for U_f. Then there is a complement g of f such that t is a g-translator for U_f.

Thus to find a translator for a complete set of view updates, it is sufficient to specify an appropriate complementary view g and take the corresponding g-translator. The theorem says that one can find such g if a translator exists at all.

The preceeding framework provides an abstract, elegant perspective on the view update problem. Forming bridges to the more concrete frameworks in which views are defined by specific languages (e.g., relational algebra) remains largely unexplored.

22.4 Updating Incomplete Information

In a sense, an update to a view is an incompletely specified update whose completion must be determined or selected. In this section, we consider more general settings for studying updates and incomplete information.

First we return to the conditional tables of Chapter 19 and show a system for updating such databases. We then introduce formulations of incomplete information that use theories (i.e., sets of propositional or first-order sentences) to represent the (partial) knowledge about the world. Among other benefits, this approach offers an interesting alternative to resolving the view update problem. This section concludes by comparing these approaches to belief revision.

Updating Conditional Tables

The problems posed by updating a c-table are similar to those raised by queries. A representation T specifies a set of possible worlds $rep(T)$. Given an update u, the possible outcomes of the update are

$$u(rep(T)) = \{u(\mathbf{I}) \mid \mathbf{I} \in rep(T)\}.$$

As for queries, it is desirable to represent the result in the same representation system. If the representation system is always capable of representing the answer to any update in a language $\mathcal{L}$, it is a *strong representation system with respect to* $\mathcal{L}$.

Let us consider c-tables and simple insertions, deletions, and modifications, as in the language of IDM transactions. We know from Chapter 19 that c-tables form a strong representation system for relational algebra; and it is easily seen that IDM transactions can be expressed in the algebra (see Exercise 22.3). It follows that c-tables are a strong representation system for IDM transactions. In other words, for each c-table T and IDM transaction t, there exists a c-table $\bar{t}(T)$ such that $rep(\bar{t}(T)) = t(rep(T))$.

Example 22.4.1 Consider the c-table in Example 19.3.1. Insertions $ins(t)$ are straightforward: t is simply inserted in the table. Consider the deletion $d = del(\{Student = Sally, Course = Physics\})$. The c-table $\bar{t}(T)$ representing the result of the deletion is

Student	*Course*	
	$(x \neq Math) \wedge (x \neq CS)$	
Sally	*Math*	$(z = 0)$
Sally	*CS*	$(z \neq 0)$
Sally	x	$(x \neq Physics)$
Alice	*Biology*	$(z = 0)$
Alice	*Math*	$(x = Physics) \wedge (t = 0)$
Alice	*Physics*	$(x = Physics) \wedge (t \neq 0)$

Consider again the original c-table T in Example 19.3.1 and the modification

$$m = mod(\{Student = Sally, Course = Music\} \rightarrow \{Course = Physics\}).$$

The c-table $\overline{m}(T)$ representing the result of the modification is

Student	*Course*	
	$(x \neq Math) \wedge (x \neq CS)$	
Sally	*Math*	$(z = 0)$
Sally	*CS*	$(z \neq 0)$
Sally	*Physics*	$(x = Music)$
Sally	x	$(x \neq Music)$
Alice	*Biology*	$(z = 0)$
Alice	*Math*	$(x = Physics) \wedge (t = 0)$
Alice	*Physics*	$(x = Physics) \wedge (t \neq 0)$

In the context of incomplete information, it is natural to consider updates that themselves have partial information. For c-tables, it seems appropriate to define updates with the same kind of incomplete information, using tuples with variables subject to conditions. We can define extensions of insertions, deletions, and modifications in this manner. It can be shown that c-tables remain a strong representation system for such updates.

Representing Databases Using Logical Theories

Conditional tables provide a stylized, restricted framework for representing incomplete information and are closed under a certain class of updates. We now turn to more general frameworks for representing and updating incomplete information. These are based on representing databases as logical theories.

Given a logical theory **T** (i.e., set of sentences), the set of *models* of **T** is denoted by *Mod*(**T**). In our context, each model corresponds to a different possible instance. If $|Mod(\mathbf{T})| > 1$, then **T** can be viewed as representing incomplete information.

In general, these approaches use the *open world assumption* (OWA). Recall from Chapter 2 that under the closed world assumption (CWA), a fact is viewed as false unless it can be proved from explicitly stated facts or sentences. In contrast, under the OWA if a fact is not implied or contradicted by the underlying theory, then the fact may be true or false. As a simple example, consider the theory $\mathbf{T} = \{p\}$ over a language with two propositional constants p and q. Under the CWA, there is only one model of **T** (namely, $\{p\}$), but under the OWA, there are two models (namely, $\{p\}$ and $\{p, q\}$).

Model-Based Approaches to Updating Theories

One natural approach to updating a logical theory **T** is *model based*; it focuses on how proposed updates affect the elements of *Mod*(**T**). Given an update u and instance **I**, let $u(\mathbf{I})$ denote the set of possible instances that could result from applying u to **I**. We use a set for the result to accommodate the case in which u itself involves incomplete information.

Now let **T** be a theory and u an update. Under the model-based approach, the result $u(\mathbf{T})$ of applying u to **T** should be a theory $\mathbf{T}'$ such that

$$Mod(\mathbf{T}') = \cup\{u(\mathbf{I}) \mid \mathbf{I} \in Mod(\mathbf{T})\}.$$

Example 22.4.2

(a) Consider the theory $\mathbf{T} = \{p \wedge q\}$, where p and q are propositional constants, and the update [*insert* $\neg p$]. There is only one model of **T** (namely, $\{p, q\}$). If we take the meaning of *insert* $\neg p$ to be “make p false and leave other things unchanged,” then updating this model yields the single model $\{q\}$. Thus the result of applying [*insert* $\neg p$] to **T** yields the theory $\{q\}$.

(b) Consider $\mathbf{T}' = \{p \vee q\}$ and the update [*insert* $\neg p$]. The models of $\mathbf{T}'$ and the impact of the update are given by

$$\begin{aligned} \{p\} &\longmapsto \emptyset \\ \{q\} &\longmapsto \{q\} \\ \{p, q\} &\longmapsto \{q\}. \end{aligned}$$

Thus the result of applying the update to $\mathbf{T}'$ is $\{\neg p\}$.

The approach to updating c-tables presented earlier falls within the model-based paradigm (see Exercise 22.14). A family of richer model-based frameworks that supports null values and disjunctive updates has also been developed. An interesting dimension of variation in this approach concerns how permissive or restrictive a given update semantics is. This essentially amounts to considering how many models are associated with $u(\mathbf{I})$ for given update u and instance **I**. As a simple example, consider starting with an empty database $\mathbf{I}_\emptyset$ and the update [*insert* $(p \vee q)$]. Under a restrictive semantics, only $\{p\}$ and $\{q\}$

are in $u(\mathbf{I}_\emptyset)$, but under a permissive semantics, $\{p, q\}$ might also be included. The update semantics for c-tables given earlier is very permissive: All possible models corresponding to an update are included in the result.

Formula-Based Approaches to Updating Theories

Another approach to updating theories is to apply updates directly to the theories themselves. As we shall see, a disadvantage of this approach is that the same update may have a different effect on equivalent but distinct theories. On the other hand, this approach does allow us to assign priorities to different sentences (e.g., so that constraints are given higher priority than atomic facts).

We consider two forms of update: [*insert* φ] and [*delete* φ], where φ is a sentence (i.e., no free variables). Given theory $\mathbf{T}$, a theory $\mathbf{T}'$ *accomplishes* the update [*insert* φ] for $\mathbf{T}$ if $\varphi \in \mathbf{T}'$, and it *accomplishes* [*delete* φ] for $\mathbf{T}$ if[1] $\varphi \notin \mathbf{T}'^*$. Observe that there is a difference between [*insert* $\neg\varphi$] and [*delete* φ]: In the former case $\neg\varphi$ is true for all models of $\mathbf{T}'$, whereas in the latter case φ may hold in some model of $\mathbf{T}'$.

In general, we are interested in accomplishing an update for $\mathbf{T}$ with minimal impact on $\mathbf{T}$. Given theory $\mathbf{T}$, we define a partial order $\leq_\mathbf{T}$ on theories with respect to the degree of change from $\mathbf{T}$. In particular, we define $\mathbf{T}' \leq_\mathbf{T} \mathbf{T}''$ if $\mathbf{T} - \mathbf{T}' \subset \mathbf{T} - \mathbf{T}''$, or if $\mathbf{T} - \mathbf{T}' = \mathbf{T} - \mathbf{T}''$ and $\mathbf{T}' - \mathbf{T} \subseteq \mathbf{T}'' - \mathbf{T}$. Intuitively, $\mathbf{T}' \leq_\mathbf{T} \mathbf{T}''$ if $\mathbf{T}'$ has fewer deletions (from $\mathbf{T}$) than $\mathbf{T}''$, or both $\mathbf{T}'$ and $\mathbf{T}''$ have the same deletions but $\mathbf{T}'$ has no more insertions than $\mathbf{T}''$. (Exercise 22.16 considers the opposite ordering, where insertions are given priority over deletions.)

Intuitively, we are interested in theories $\mathbf{T}'$ that accomplish a given update u for $\mathbf{T}$ and are minimal under $\leq_\mathbf{T}$. We say that such theories $\mathbf{T}'$ accomplish u for $\mathbf{T}$ *minimally*. The following characterizes such theories (see Exercise 22.15):

Proposition 22.4.3 Let $\mathbf{T}$, $\mathbf{T}'$ be theories and φ a sentence. Then

(a) $\mathbf{T}'$ accomplishes [*delete* φ] for $\mathbf{T}$ minimally iff $\mathbf{T}'$ is a maximal subset of $\mathbf{T}$ that is consistent with $\neg\varphi$.

(b) $\mathbf{T}' \cup \varphi$ accomplishes [*insert* φ] for $\mathbf{T}$ minimally iff $\mathbf{T}'$ is a maximal subset of $\mathbf{T}$ that is consistent with φ.

Thus $\mathbf{T}'$ accomplishes [*delete* φ] for $\mathbf{T}$ minimally iff $\mathbf{T}' \cup \neg\varphi$ accomplishes [*insert* $\neg\varphi$] for $\mathbf{T}$ minimally.

The following example shows that equivalent but distinct theories can be affected differently by updates.

Example 22.4.4 (a) Consider the theory $\mathbf{T}_0 = \{p, q\}$ and the update [*insert* $\neg p$]. Then $\{\neg p, q\}$ is the unique minimal theory that accomplishes this update.

---

[1] For a theory $\mathbf{S}$, the (*logical*) *closure* of $\mathbf{S}$, denoted $\mathbf{S}^*$, is the set of all sentences implied by $\mathbf{S}$.

(b) Let $\mathbf{T}_1 = \{p \wedge q\}$ and consider $[insert\ \neg p]$. The unique minimal theory that accomplishes this update for $\mathbf{T}_1$ is $\{\neg p\}$ [i.e., $(\emptyset \cup \{\neg p\})$]. Note how this differs from the model-based update in Example 22.4.2(a).

A problem at this point is that, in general, there are several theories that minimally accomplish a given update. Thus an update to a theory may yield a set of theories, and so the framework is not closed under updates. Given a set $\mathbf{T}_1, \mathbf{T}_2, \ldots,$ we would like to find a theory $\mathbf{T}$ whose models are exactly the union of all models of the set of theories. In general, it is not clear that there is a theory that has this property. However, if there is only a finite number of theories that are possible answers, then we can use the *disjunction* operator $\bigvee$ defined by

$$\bigvee\{\mathbf{T}_i \mid i \in [1,n]\} = \{\tau_1 \vee \cdots \vee \tau_n \mid \tau_i \in \mathbf{T}_i \text{ for } i \in [1,n]\}.$$

It is easily verified that $Mod(\bigvee\{\mathbf{T}_i \mid i \in [1,n]\}) = \cup\{Mod(\mathbf{T}_i) \mid i \in [1,n]\}$. Of course, there is a great likelihood of a combinatorial explosion if the disjunction operator is applied repeatedly.

Assigning Priorities to Sentences

We now explore a mechanism for giving priority to some sentences in a theory over other sentences. Let $n \geq 0$ be fixed. A *tagged sentence* is a pair (i, φ), where $i \in [0, n]$ and φ is a sentence. A *tagged theory* is a set of tagged sentences. Given tagged theory $\mathbf{T}$ and $i \in [1,n]$, $\mathbf{T}_i$ denotes $\{\varphi \mid (i,\varphi) \in \mathbf{T}\}$.

The partial order for comparing theories is extended in the following natural fashion. Given tagged theories $\mathbf{T}$, $\mathbf{T}'$ and $\mathbf{T}''$, define $\mathbf{T}' \leq_{\mathbf{T}} \mathbf{T}''$ if for some $i \in [1,n]$ we have

$$\mathbf{T}_j - \mathbf{T}'_j = \mathbf{T}_j - \mathbf{T}''_j, \text{ for each } j \in [1, i-1]$$

and

$$\mathbf{T}_i - \mathbf{T}'_i \subset \mathbf{T}_i - \mathbf{T}''_i$$

or we have

$$\mathbf{T}_j - \mathbf{T}'_j = \mathbf{T}_j - \mathbf{T}''_j, \text{ for each } j \in [1, n]$$

and

$$\mathbf{T}' - \mathbf{T} \subset \mathbf{T}'' - \mathbf{T}.$$

Intuitively, $\mathbf{T}' \leq_{\mathbf{T}} \mathbf{T}''$ if the deletions of $\mathbf{T}'$ and $\mathbf{T}''$ agree up to some level i and then $\mathbf{T}'$ has fewer deletions at level i; or if the deletions match and $\mathbf{T}'$ has fewer insertions. In this manner, higher priority is given to the sentences having lower numbers.

Example 22.4.5 Consider a relation $R[ABC]$ that satisfies the functional dependency $A \to B$, and consider the instance

R	A	B	C
	a	b	c
	a	b	c'
	a'	b'	c''
	a''	b'	c'''

We now construct a tagged theory **T** to represent this situation and show how changing a B value of a tuple is accomplished.

We assume three tag values and describe the contents of $\mathbf{T}_0$, $\mathbf{T}_1$, and $\mathbf{T}_2$ in turn. $\mathbf{T}_0$ holds the functional dependency and the unique name axiom (see Chapter 2). That is,

$$\left\{\begin{array}{c} (0, \forall x, y, y', z, z'(R(x, y, z) \wedge R(x, y', z') \to y = y')), \\ (0, a \neq a'), (0, a \neq a''), \ldots, (0, a \neq b), \ldots, (0, c'' \neq c''') \end{array}\right\}$$

$\mathbf{T}_1$ holds the following existential sentences:

$$\left\{\begin{array}{c} (1, \exists x(R(a, x, c))), \\ (1, \exists x(R(a, x, c'))), \\ (1, \exists x(R(a', x, c''))), \\ (1, \exists x(R(a'', x, c'''))) \end{array}\right\}$$

Finally, $\mathbf{T}_2$ holds

$$\left\{\begin{array}{c} (2, R(a, b, c)), \\ (2, R(a, b, c')), \\ (2, R(a', b', c'')), \\ (2, R(a'', b', c''')) \end{array}\right\}$$

Consider now the update $u = [\mathit{insert}\ \varphi]$, where $\varphi = \exists y R(a, b'', y)$. Intuitively, this insertion should replace all $\langle a, b\rangle$ pairs occurring in $\pi_{AB}R$ by $\langle a, b''\rangle$. More formally, it is easy to verify that the unique tagged theory (up to choice of i) that accomplishes u is (see Exercise 22.17)

$$\{(i, \varphi)\} \cup \mathbf{T}_0 \cup \mathbf{T}_1 \cup \left\{\begin{array}{c} (2, R(a, b'', c)) \\ (2, R(a, b'', c')) \\ (2, R(a', b', c'')) \\ (2, R(a'', b', c''')) \end{array}\right\}$$

Thus the choice of sentences and tags included in the theory can influence the result of an update.

The approach of tagged theories can also be used to develop a framework for accomplishing view updates. The underlying database and the view are represented using a tagged theory, and highest priority is given to ensuring that the complement of the view remains fixed. Exercise 22.18 explores a simple example of this approach.

In the approach described here, a set of theories is combined using the disjunction operator. In this case, multiple deletions can lead to an exponential blowup in the size of the underlying theory, and performing insertions is NP-hard (see Exercise 22.19). This provided one motivation for developing a generalization of the approach, in which families of theories, called flocks, are used to represent a database with incomplete information.

Update versus Revision

The idea of representing knowledge using theories is not unique to the field of databases. The field of belief revision takes this approach and considers the issue of revising a knowledge base. Here we briefly compare the approaches to updating database theories described earlier with those found in belief revision.

A starting point for belief revision theory is the set of *rationality postulates* of Alchourrón, Gärdenfors, and Makinson, often referred to as the *AGM* postulates. These present a general family of guidelines for when a theory accomplishes a revision, and they include postulates such as

(R1) If $\mathbf{T}'$ accomplishes [*insert* φ] for $\mathbf{T}$, then $\mathbf{T}' \models \varphi$.

(R2) If φ is consistent with $\mathbf{T}$, then the result of [*insert* φ] on $\mathbf{T}$ should be equivalent to $\mathbf{T} \cup \{\varphi\}$.

(R3) If $\mathbf{T} \equiv \mathbf{T}'$ and $\varphi \equiv \varphi'$, then the result of [*insert* φ] on $\mathbf{T}$ is equivalent to the result of [*insert* φ'] on $\mathbf{T}'$.

(This is a partial listing of the eight AGM postulates.) Other postulates focus on maintaining satisfiability, relationships between the effects of different updates, and capturing some aspects of minimal change.

It is clear from postulate (R3) that the formula-based approaches to updating database theories do not qualify as belief revision systems. The relationship of the formula-based approaches and belief revision is largely unexplored.

A key difference between belief revision and the model-based approach to updating database theories stems from different perspectives on what a theory $\mathbf{T}$ is intended to represent. In the former context, $\mathbf{T}$ is viewed as a set of beliefs about the state of the world. If a new fact φ is to be inserted, this is a modification (and, it is hoped, improvement) of our knowledge about the state of the world, but the world itself is considered to remain unchanged. In contrast, in the model-based approaches, the theory $\mathbf{T}$ is used to identify a set of worlds that are possible given the limited information currently available. If a fact φ is inserted, this is understood to mean that the world itself has been modified. Thus $\mathbf{T}$ is modified to identify a different set of possible worlds.

Example 22.4.6 Suppose that the world of interest is a room with a table in it. There is an abacus and a (hand-held, electronic) calculator in the room. Let proposition a mean that the abacus is on the table, and let proposition c mean that the calculator is on the table. Finally, let $\mathbf{T}$ be $(a \wedge \neg c) \vee (\neg a \wedge c)$.

From the perspective of belief revision, $\mathbf{T}$ indicates that according to our current knowledge, either the abacus or the calculator is on the table, but not both. Suppose that we are informed that the calculator is on the table (i.e., [*insert* c]). This is viewed as additional knowledge about the unchanging world. Combining $\mathbf{T}$ with c, we obtain the new theory $\mathbf{T}_1 = ((a \wedge \neg c) \vee (\neg a \wedge c)) \wedge c \equiv (\neg a \wedge c)$. [Note that this outcome is required by postulate (R2).]

From the model-based perspective, $\mathbf{T}$ indicates that either the world is $\{a\}$ or it is $\{c\}$. The request [*insert* c] is understood to mean that the world has been modified so that c has become true. This can be envisioned in terms of having a robot enter the room and place the calculator on the table (if it isn't already there) without reporting on the status of anything except that the robot has been successful. As a result, the world $\{a\}$ is replaced by $\{a, c\}$, and the world $\{c\}$ is replaced by itself. The resulting theory is $\mathbf{T}_2 = c$ (which is interpreted under the OWA).

A set of postulates for updates, analogous to the AGM postulates for revision, has been developed. The postulate analogous to (R2) is

(U2) If $\mathbf{T}$ implies φ, then the result of [*insert* φ] on $\mathbf{T}$ should be equivalent to $\mathbf{T}$.

This is strictly weaker than (R2). Other postulates enforce the intuition that the effect of an update on a possible model is independent of the other possible models of a theory, maintaining satisfiability and relationships between the effects of different updates.

22.5 Active Databases

As we have seen, object orientation provides one paradigm for incorporating behavioral information into a database schema. This has the effect of separating a portion of the behavioral information from the application software and providing a more structured representation and organization for that portion. In this section, we briefly consider a second, essentially orthogonal, paradigm for separating a portion of the behavioral information from the application software. This emerging paradigm, called activeness, stems from a synthesis of techniques from databases, on the one hand, and expert systems and artificial intelligence, on the other.

Active databases generally support the automatic triggering of updates in response to internal or external events (e.g., a clock tick, a user-requested update, or a change in a sensor reading). In a manner reminiscent of expert systems, forward chaining of *rules* is generally used to accomplish the response. However, there are several differences between classical expert systems and active databases. At the conceptual and logical level, the differences are centered around the expressive power of rule conditions and the semantics of rule application. (Some active database systems, such as POSTGRES, also support a form of backward chaining or query rewriting; this is not considered here.)

Active databases have been shown to be useful in a variety of areas, including con-

Suppliers	*Sname*	*Address*
	The Depot	1210 Broadway
	Builder's Mart	100 Main

Prices	*Part*	*Sname*	*Price*
	nail	The Depot	.02
	bolt	The Depot	.05
	bolt	Builder's Mart	.04
	nut	Builder's Mart	.03

Figure 22.3: Sample instance for active database examples

straint maintenance, incremental update of materialized views, mapping view updates to the base data, and supporting database interoperability.

Rules and Rule Application

There are three distinguishing components in an active database: (1) a subsystem for monitoring events, (2) a set of rules, often called a *rule base*, and (3) a semantics for rule application, typically called an *execution model*.

Rules typically have the following so-called ECA form:

on ⟨*event*⟩ **if** ⟨*condition*⟩ **then** ⟨*action*⟩.

Depending on the system and application, the *event* may range over external phenomena and/or over internal events (such as a method call or inserting a tuple to a relation). Events may be atomic or *composite*, where these are built up from atomic events using, say, regular expressions or a process algebra. Events may be essentially Boolean or may return a tuple of values that indicate what triggered the event.

Conditions typically involve parameters passed in by the events, and the contents of the database. As will be described shortly, several systems permit conditions to look at more than one version of the database state (e.g., corresponding to the state before the event and the state after the event). In some systems, events are not explicitly specified; essentially any change to the database makes the event true and leads to testing of all rule conditions.

In principle, the *action* may be a call to an arbitrary routine. In many cases in relational systems, the action will involve a sequence of insertions, deletions, and modifications; and in object-oriented systems it will involve one or more method calls. Note that this may in turn trigger other rules.

The remainder of this discussion focuses on the relational model. A short example is given, followed by a brief discussion of execution models.

Example 22.5.1 Suppose that the *Inventory* database includes the following relations:

$$\begin{aligned} &\textit{Suppliers}[\textit{Sname}, \textit{Address}] \\ &\textit{Prices}[\textit{Part}, \textit{Sname}, \textit{Price}] \end{aligned}$$

Suppliers and the parts they supply are represented in *Suppliers* and *Prices*, respectively. It is assumed that *Sname* is a key of *Suppliers* and *Part*, *Sname* is a key of *Prices*. An example instance is shown in Fig. 22.3.

We now list some example rules. These rules are written in a pidgin language that uses tuple variables. The variable T ranges over sets of tuples and is used to pass them from the condition to the action. As detailed shortly, both (r1) considered in isolation and the set (r2.a) ... (r2.d) taken together can be used to enforce the inclusion dependency $Prices[Sname] \subseteq Suppliers[Sname]$.

```
(r1)    on true
        if $Prices(p)$ and $p.Sname \notin \pi_{Sname}(Suppliers)$
        then $Prices := Prices - \{p\}$

(r2.a)  on delete $Sname(s)$
        if $T := \sigma_{Sname=s.Sname}(Prices)$ is not empty
        then $Prices := Prices - T$

(r2.b)  on modify $Sname(s)$
        if $old(s).Sname \neq new(s).Sname$
            and $T = \sigma_{Sname=old(s).Sname}(Prices)$
        then set $p.Sname = new(s).Sname$
            for each $p$ in $Prices$
            where $p \in T$

(r2.c)  on insert $Prices(p)$
        if $\langle p.Sname \rangle \notin \pi_{Sname}(Suppliers)$
        then issue supplier_warning($p$)

(r2.d)  on modify $Prices(p)$
        if $\langle new(p).Sname \rangle \notin \pi_{Sname}(Suppliers)$
        then issue supplier_warning($new(p)$)
```

Consider rule (r1). If ever a state arises that violates the inclusion dependency, then the rule deletes violating tuples from the *Prices* relation. The event of (r1) is always true; in principle the database must check the condition whenever an update is made. It is easy to see in this case that such checking need only be done if the relations *Supplies* or *Prices* are updated, and so the event "**on** *Supplies* or *Prices* is updated" could be incorporated into (r1). Although this does not change the effect of the rule, it provides a hint to the system about how to implement it efficiently.

Rules (r2.a) ... (r2.d) form an alternative mechanism for enforcing the inclusion dependency. In this case, the cause of the dependency violation determines the reaction of the system. Here a deletion from (r2.a) or modification (r2.b) to *Suppliers* will result in deletions from or modifications to *Prices*. In (r2.b), variable s ranges over tuples that have been modified, $old(s)$ refers to the original value of the tuple, and $new(s)$ refers to the modified value. On the other hand, changes to *Prices* that cause a violation [rules (r2.c) and (r2.d)] call a procedure `supplier_warning`; this might abort the transaction and warn the user or dba of the constraint violation, or it might attempt to use heuristics to modify the offending *Sname* value.

Execution Models

Until now, we have considered rules essentially in isolation from each other. A fundamental issue concerns the choice of an execution model, which specifies how and when rules will be applied. As will be seen, a wide variety of execution models are possible. The true semantics of a rule base stems both from the rules themselves and from the execution model for applying them.

We assume for this discussion that there is only one user of the system, or that a concurrency control protocol is enforced that hides the effect of other users.

Suppose that a user transaction $t = c_1; \ldots ; c_n$ is issued, where each of the c_i's is an atomic command. In the absence of active database rules, application of t will yield a sequence

$$\mathbf{I}_0, \mathbf{I}_1, \ldots , \mathbf{I}_n$$

of database states, starting with the original state $\mathbf{I}_0$ and where each state $\mathbf{I}_{i+1}$ is the result of applying c_{i+1} to state $\mathbf{I}_i$. If rules are present, then a different sequence of states might arise.

One dimension of variation between execution models concerns when rules are fired. Under *immediate* firing, a rule is essentially fired as soon as its event and condition become true; under *deferred* firing, rule application is delayed until after the state $\mathbf{I}_n$ is reached; and under *concurrent* firing, a separate process is spawned for the rule action and is executed concurrently with other processes. In the most general execution models, each rule is assigned its own coupling mode (i.e., immediate, deferred, or concurrent), which may be further refined by associating a coupling mode between event and condition testing and between condition testing and action execution.

We now examine the semantics of immediate and deferred firing in more detail. We assume for this discussion that the event of each rule is simply *true*.

To illustrate immediate firing, suppose that a rule r with action $d_1; \ldots ; d_m$ is triggered (i.e., its condition has become true) in state $\mathbf{I}_1$ of the preceeding sequence of states. Then the sequence of databases states might start with

$$\mathbf{I}_0, \mathbf{I}_1, \mathbf{I}'_1, \mathbf{I}'_2, \ldots , \mathbf{I}'_m, \ldots ,$$

where $\mathbf{I}'_1$ is the result of applying d_1 to $\mathbf{I}_1$ and $\mathbf{I}'_{j+1}$ is the result of applying d_{j+1} to $\mathbf{I}'_j$. After $\mathbf{I}'_m$, the command c_2 would be applied. The semantics of intermediate rule firing is in fact more complex, for two reasons. First, another rule might be triggered during the execution of the action of the first triggered rule. In general, this calls for a recursive style of rule application, where the command sequences of each triggered rule are placed onto a stack. Second, several rules might be triggered at the same time. One approach in this case is to assume that the rules are ordered and that rules triggered simultaneously are considered in that order. Another approach is to fire simultaneously-triggered rules concurrently; essentially this has the effect of firing them in a nondeterministic order.

In the case of deferred firing, the full user transaction is completed before any rules are fired, and each rule action is executed in its entirety before another rule action is initiated. This gives rise to a sequence of states having the form

$$\mathbf{I}^{orig}, \mathbf{I}^{user}, \mathbf{I}_2, \mathbf{I}_3, \ldots, \mathbf{I}^{curr},$$

where now $\mathbf{I}^{orig}$ is the original state, $\mathbf{I}^{user}$ is the result of applying the user-requested transaction, and the states $\mathbf{I}_2, \mathbf{I}_3, \ldots, \mathbf{I}^{curr}$ are the results of applying the actions of fired rules. The sequence shown here might be extended if additional rules are to be fired.

Several intricacies arise. As before, the order of rule firing must be considered if multiple rules are triggered at a given state. Recall the (r2) rules of Example 22.5.1, whose events where based on transitions between some former state and some latter state. What states should be used? It is natural to use $\mathbf{I}^{curr}$ as the latter state. With regard to the former state, some systems advocate using $\mathbf{I}^{orig}$, whereas other systems support the use of one of the intermediate states (where the choice may depend on a complex condition).

Suppose that two rules r and r' are triggered at some state $\mathbf{I}^{curr} = \mathbf{I}_i$ and that r is fired first to reach state $\mathbf{I}_{i+1}$. The event and/or condition of r' may no longer be true. This raises the question, Should r' be fired? A consensus has not emerged in the literature.

As should be clear from the preceeding discussion, there is a wide variety of choices for execution models. A more subtle dimension of flexibility concerns the expressive power of rule events and conditions: In addition to accessing the current state, should they be able to access one or more previous ones? Several prototype active database systems have been implemented; each uses a different execution model, and several permit access to both current and previous states. It has been argued that different execution models may be appropriate for different applications. This has given rise to systems that include a choice of execution models and to languages that permit the specification of customized execution models. An open problem at the time this book was written is to develop a natural syntax that can be used to specify easily a broad range of execution models, including a substantial subset of those described in the literature.

The *while* languages studied in Part E can serve as the kernel of an active database. These languages do not use events; restrict rule actions to insertions, deletions, and value creation; and examine only the current state in a rule firing sequence. If value creation is supported, then these languages are complete for database mappings and so in some sense can simulate all active databases. However, richer rules and execution models permit the possibility of developing rule bases that enforce a desired set of policies in a more intuitive fashion than a *while* program.

An Execution Model That Reaches a Unique Fixpoint

It should be clear that whatever execution model and form for rules is selected, most questions about the behavior of an active database are undecidable. It is thus interesting to consider more restricted execution models that behave in predictable ways. We now present one such execution model, called the *accumulating* model; this forms a portion of the execution model of AP5, a main-memory active database system that has been used in research for over a decade.

To describe the accumulating execution model, we first introduce the notion of a *delta*. Let $\mathbf{R} = \{R_1, \ldots, R_n\}$ be a database schema. An *atomic update* over $\mathbf{R}$ is an expression of the form $+R_i(t)$ or $-R_i(t)$, where $i \in [1, n]$ and t is a tuple having the arity of R_i. A *delta* over $\mathbf{R}$ is a finite set of atomic updates over $\mathbf{R}$ that does not contain both $+R(t)$ and $-R(t)$ for any R and t or the special value *fail*. (Modifies could also be incorporated into deltas, but we do not consider that here.) A delta not containing the value *fail* is *consistent*. For delta Δ, we define

$$\Delta^+ = \{R(t) \mid +R(t) \in \Delta\}$$
$$\Delta^- = \{R(t) \mid -R(t) \in \Delta\}.$$

Given instance $\mathbf{I}$ and consistent delta Δ over $\mathbf{R}$, the result of *applying* Δ to $\mathbf{I}$ is

$$\mathit{apply}(\mathbf{I}, \Delta) = (\mathbf{I} \cup \Delta^+) - \Delta^- = (\mathbf{I} - \Delta^-) \cup \Delta^+.$$

Finally, the *merge* of two consistent deltas Δ_1, Δ_2 is defined by

$$\Delta_1 \& \Delta_2 = \begin{cases} \Delta_1 \cup \Delta_2 & \text{if this is consistent} \\ \mathit{fail} & \text{otherwise.} \end{cases}$$

The accumulating execution model uses deferred rule firing. Each rule action is viewed as producing a consistent delta. The user-requested transaction is also considered to be the delta Δ_0. Thus a sequence of states

$$\mathbf{I}^{orig} = \mathbf{I}_0, \mathbf{I}^{user} = \mathbf{I}_1, \mathbf{I}_2, \mathbf{I}_3, \ldots, \mathbf{I}^{curr}$$

is produced, where $\mathbf{I}^{user} = \mathit{apply}(\mathbf{I}^{orig}, \Delta_0)$ and, more generally, $\mathbf{I}_{i+1} = \mathit{apply}(\mathbf{I}_i, \Delta_i)$ for some Δ_i produced by a rule firing.

At this point the accumulating model is quite generic. We now restrict the model and develop some interesting theoretical properties. First we assume that rules have only conditions and actions (i.e., that the event part is always *true*). Second, as noted before, we assume that the action of each rule can be viewed as a delta. Furthermore, we assume that these deltas use only constants from $\mathbf{I}^{orig}$ (i.e., there is no invention of constants). Third we insist that for each $i \geq 0$, $\Delta_0 \& \ldots \& \Delta_i$ is consistent. More precisely, we modify the execution model so that if for some i we have $\Delta_0 \& \ldots \& \Delta_i = \mathit{fail}$, then the execution is aborted. For each $i \geq 0$, let $\Delta'_i = \Delta_0 \& \ldots \& \Delta_i$.

Suppose that we are now in state $\mathbf{I}^{curr}$ with delta Δ^{curr}. We assume that rule conditions can access only $\mathbf{I}^{orig}$ and Δ^{curr}. (If the rule conditions have the power of, for example, the relational calculus, this means they can in effect access $\mathbf{I}^{curr}$.) Given rule r, state $\mathbf{I}$, and delta Δ, the *effect* of r on $\mathbf{I}$ and Δ, denoted $\mathit{effect}(r, \mathbf{I}, \Delta)$, is the delta corresponding to the firing of r on $\mathbf{I}$ and Δ, if the condition of r is satisfied, and is $\emptyset$ otherwise.

Execution proceeds as follows. The sequence $\Delta'_0, \Delta'_1, \ldots$ is constructed sequentially. At the i^{th} step, if there is no rule whose condition is satisfied by $\mathbf{I}^{orig}$ and Δ'_i, then execution terminates successfully. Otherwise a rule r with condition satisfied by $\mathbf{I}^{orig}$ and Δ'_i is selected nondeterministically. If $\Delta'_i \& \mathit{effect}(r, \mathbf{I}^{orig}, \Delta'_i)$ is *fail*, then execution terminates with an abort; otherwise set $\Delta'_{i+1} = \Delta'_i \& \mathit{effect}(r, \mathbf{I}^{orig}, \Delta'_i)$ and continue.

A natural question at this point is, Will execution always terminate? It is easy to see that it does, because constants are not invented and the sequence of deltas being constructed is monotonically increasing under set containment.

It is also natural to ask, Does the order of rule firing affect the outcome? In general, the answer is yes. We now develop a semantic condition on rules that ensures independence of rule firing order. A rule r is *monotonic* if for each instance $\mathbf{I}$ and pair $\Delta_1 \subseteq \Delta_2$ of deltas, $\mathit{effect}(r, \mathbf{I}, \Delta_1) \subseteq \mathit{effect}(r, \mathbf{I}, \Delta_2)$. The following can now be shown (see Exercise 22.23):

Theorem 22.5.2 If each rule in a rule base is monotonic, then the outcome of the accumulating execution model on this rule base is independent of rule firing order.

Monitoring Events and Conditions

In Example 22.5.1, the events that triggered rules were primitive, in the sense that each one corresponded to an atomic occurrence of some phenomenon. There has been recent interest in developing languages for specifying and recognizing composite events, which might involve the occurrence of several primitive events. For example, composite event specification is supported by the ODE system, a recently released prototype object-oriented active database system. The ODE system supports a rich language for specifying composite events, which has essentially the power of regular expressions (see also Section 22.6 for examples of composite events specified by regular expressions). An implementation technique based on finite state automata has been developed for recognizing composite events specified in this language.

Other formalisms can also be used for specifying composite events (e.g., using Petri nets or temporal logics). There appears to be a trade-off between the expressiveness of triggers in rules and conditions. For example, some Petri-net-based languages for composite events can be simulated using additional relations and rules based on simple events. The details of such trade-offs are largely unexplored.

22.6 Temporal Databases and Constraints

Classical databases model *static* aspects of data. Thus the information in the database consists of data currently true in the world. However, in many applications, information about the history of data is just as important as static information. When history is taken into account, queries can ask about the evolution of data through time; and constraints may restrict the way changes occur. We briefly discuss these two aspects.

Temporal Databases

Suppose we are interested in a database over some schema $\mathbf{R}$. Thus we wish to model and query information about the content of the database through time. Conceptually, we can associate to each time t the state $\mathbf{I}_t$ of the database at time t. Thus the database appears as a sequence of states—*snapshots*—indexed by some time domain. Two basic questions come up immediately:

- What is the meaning of $\mathbf{I}_t$? Primarily two possible answers have been proposed. The first is that $\mathbf{I}_t$ represents the data that was true in the world at time t; this view of time is referred to as *valid time*. The second possibility is that time represents the moment when the information was recorded in the database; this is called *transaction time*.

  Clearly, using valid time requires including time as a first-class citizen in the data model. In many applications transaction time might be hidden and dealt with by the system; however, in time-critical applications, such as air-traffic control or monitoring a power plant, transaction time may be important and made explicit. A particular database may use valid time, transaction time, or both. In our discussion, we will consider valid time only.
- What is the time domain? This can be discrete (isomorphic to the integers), continuous (isomorphic to the reals), or dense and countable (isomorphic to the rationals). In databases, time is usually taken to be discrete, with some fixed granularity for the time unit. However, several distinct time domains with different granularities are often used (e.g., years, months, days, hours, etc.). The time domain is usually equipped with a total order and sometimes with arithmetic operations. A temporal variable **now** may be used to refer to the present time.

To query a temporal database, relational languages must be extended to take into account the time coordinate. To say that a tuple u is in relation R at time t, we could simply extend R with one temporal coordinate and write $R(u, t)$. Then we could use CALC or ALG on the extended relations. This is illustrated next.

Example 22.6.1 Consider the **CINEMA** database, indexed by a time domain consisting of dates of the form month/day/year. The query

"What were the movies shown at La Pagode in May, 1968?"

is expressed in CALC by

$$\{m \mid \exists s, t(\mathit{Pariscope}(\text{La Pagode}, m, s, t) \wedge 5/1/68 \leq t \leq 5/31/68)\}.$$

The query

"Since when has La Pagode been showing the current movie?"

is expressed by

$$\{t \mid \exists m[\exists s(\mathit{Pariscope}(\text{“La Pagode”}, m, s, \mathbf{now})) \wedge \\ \mathit{since}(t, m) \wedge \forall t''(\mathit{since}(t'', m) \rightarrow t \leq t'')]\},$$

where

$$since(t, m) = \forall t'[t \leq t' \leq \textbf{now} \rightarrow \exists s'(Pariscope(\text{“La Pagode”}, m, s', t'))].$$

Classical logics augmented with a temporal coordinate have been studied extensively, mostly geared toward specification and verification of concurrent programs. Such logics are usually referred to as *temporal logics*. There is a wealth of mathematical machinery developed around temporal logics; unfortunately, little of it seems to apply directly to databases.

Although the view of a temporal database as a sequence of instances is conceptually clean, it is extremely inefficient to represent a temporal database in this manner. In practice, this information is summarized in a single database in which data is timestamped to indicate the time of validity. The timestamps can be placed at the tuple level or at the attribute level. Typically, timestamps are unions of intervals of the temporal domain. Such representations naturally lead to nested structures, as in the nested relation, semantic, and object-oriented data models.

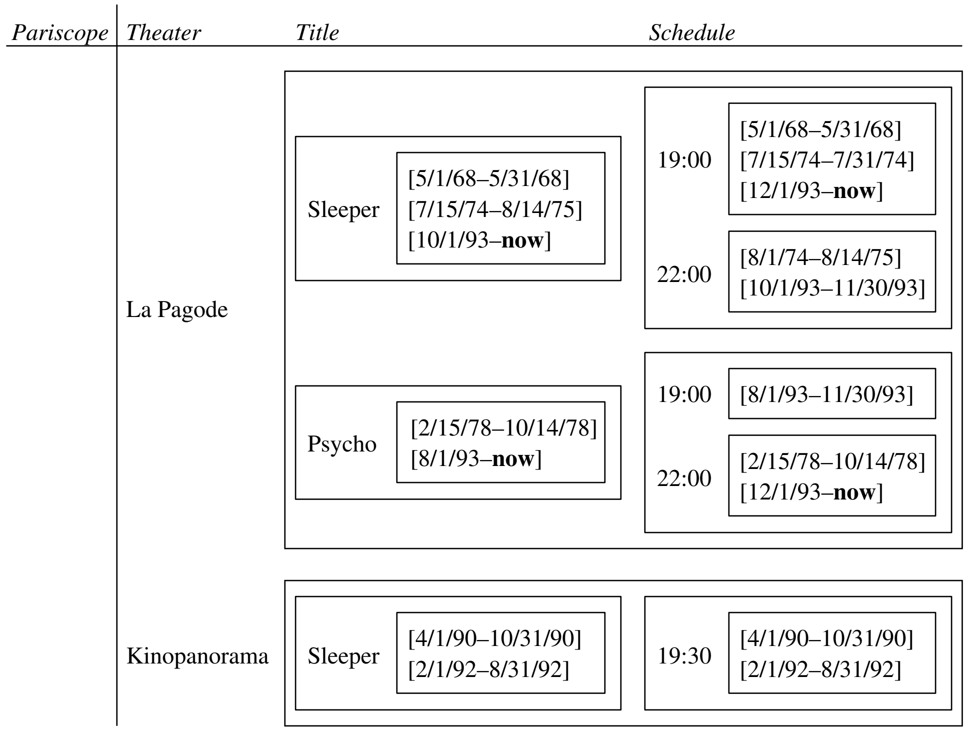
Example 22.6.2 Figure 22.4 is a representation of temporal information about *Pariscope* using attribute timestamps with nested relations. It would also be natural to represent this using a semantic or object-oriented model.

The same information can be represented by timestamping at the tuple level, as follows:

Pariscope	*Theater*	*Title*	*Schedule*	
	La Pagode	Sleeper	19:00	[5/1/68–5/31/68]
	La Pagode	Sleeper	19:00	[7/15/74–7/31/74]
	La Pagode	Sleeper	19:00	[12/1/93–**now**]
	La Pagode	Sleeper	22:00	[8/1/74–8/14/75]
	La Pagode	Sleeper	22:00	[10/1/93–11/30/93]
	La Pagode	Psycho	19:00	[8/1/93–11/30/93]
	La Pagode	Psycho	22:00	[2/15/78–10/14/78]
	La Pagode	Psycho	22:00	[12/1/93–**now**]
	Kinopanorama	Sleeper	19:30	[4/1/90–10/31/90]
	Kinopanorama	Sleeper	19:30	[2/1/92–8/31/92]

In this representation, the time intervals are more fragmented. This may have some drawbacks. For example, retrieving the information about when “Sleeper” was playing at La Pagode (using a selection and projection) yields time intervals that are more fragmented than needed. To obtain a more concise representation of the answer, we must merge some of these intervals.

Note also the difference between the timestamps and the attribute *Schedule*, which also conveys some temporal information. The value of *Schedule* is user defined, and the database may not know that this is temporal information. Thus from the point of view of



Pariscope	*Theater*	*Title*		*Schedule*	
	La Pagode	Sleeper	[5/1/68–5/31/68] [7/15/74–8/14/75] [10/1/93–**now**]	19:00	[5/1/68–5/31/68] [7/15/74–7/31/74] [12/1/93–**now**]
				22:00	[8/1/74–8/14/75] [10/1/93–11/30/93]
		Psycho	[2/15/78–10/14/78] [8/1/93–**now**]	19:00	[8/1/93–11/30/93]
				22:00	[2/15/78–10/14/78] [12/1/93–**now**]
	Kinopanorama	Sleeper	[4/1/90–10/31/90] [2/1/92–8/31/92]	19:30	[4/1/90–10/31/90] [2/1/92–8/31/92]

Figure 22.4: A representation of temporal information using attribute timestamps with nested relations

the temporal database, the value of *Schedule* is treated just like any other nontemporal value in the database.

Much of the research in temporal databases has been devoted to finding extensions of SQL and other relational languages suitable for temporal queries. Most proposals assume some representation based on tuple timestamping by intervals and introduce intuitive linguistic constructs to compare and manipulate these temporal intervals. Sometimes this is done without explicit reference to time, in the spirit of modal operators in temporal logic. One such operator is illustrated next.

Example 22.6.3 Several temporal extensions of SQL use a **when** clause to express a temporal condition. For example, consider the query on the **CINEMA** database:

"Find the pairs of theaters that have shown some movie at the same date and hour."

This can be expressed using the **when** clause as follows:

select $t_1.theater, t_2.theater$
from *Pariscope* t_1 t_2
where $t_1.title = t_2.title$ **and** $t_1.schedule = t_2.schedule$
when $t_1.interval$ **overlaps** $t_2.interval$

The **when** clause is true for tuples t_1, t_2 iff the intervals indicating their validity have nonempty intersection. Other Boolean tests on intervals include **before, after, during, follows, precedes**, etc., with the obvious semantics. The expressive power of such constructs is not always well elucidated in the literature, beyond the fact that they can clearly be expressed in CALC. A review of the many constructs proposed in the literature on temporal databases is beyond the scope of this book. For the time being, it appears that a single well-accepted temporal language is far from emerging, although there are several major prototypes.

Temporal Deductive Databases

An interesting recent development involves the use of deductive databases in the temporal framework, yielding temporal extensions of datalog. This can be used in two main ways.

- As a specification mechanism: Datalog-like rules allow the specification of some temporal databases in a concise fashion. In particular, this allows us to specify *infinite* temporal databases, with both past and future information.
- As a query mechanism: Rules can be used to express recursive temporal queries.

EXAMPLE 22.6.4 We first illustrate the use of rules in the specification of an infinite temporal database. The database holds information on a professor's schedule—more precisely, the times she meets her two Ph.D. students. The facts

$$\textit{meets-first}(\textit{Emma}, 0), \textit{follows}(\textit{Emma}, \textit{John}), \textit{follows}(\textit{John}, \textit{Emma})$$

say that the professor's first meeting is with Emma, and then John and Emma take turns. Consider the rules

$$\begin{aligned} \textit{meets}(x, t) &\leftarrow \textit{meets-first}(x, t) \\ \textit{meets}(y, t+1) &\leftarrow \textit{meets}(x, t), \textit{follows}(x, y) \end{aligned}$$

The rules define the following infinite sequence of facts providing the professor's schedule:

$$\begin{aligned} &\textit{meets}(\textit{Emma}, 0) \\ &\textit{meets}(\textit{John}, 1) \\ &\textit{meets}(\textit{Emma}, 2) \\ &\textit{meets}(\textit{John}, 3) \\ &\quad\vdots \end{aligned}$$

Another way to use temporal rules is for querying. Consider the query

"Find the times t such that La Pagode showed 'Sleeper' on date t and continued to show it at least until the Kinopanorama started showing it."

The answer (given in the unary relation *until*) is defined by the following stratified program:

$$\begin{aligned}
&\mathit{date}(x, y, t) &&\leftarrow \mathit{Pariscope}(x, y, s, t)\\
&\mathit{until}(t) &&\leftarrow \mathit{date}(\text{“Kinopanorama”}, \text{“Sleeper”}, t+1),\\
& &&\qquad \neg\, \mathit{date}(\text{“Kinopanorama”}, \text{“Sleeper”}, t),\\
& &&\qquad \mathit{date}(\text{“La Pagode”}, \text{“Sleeper”}, t)\\
&\mathit{until}(t) &&\leftarrow \mathit{date}(\text{“La Pagode”}, \text{“Sleeper”}, t), \mathit{until}(t+1)
\end{aligned}$$

The expressiveness of several datalog-like temporal languages and the complexity of query evaluation using such languages are active areas of research.

Temporal Constraints

Classical constraints in relational databases are static: They speak about properties of the data seen at some moment in time. This does not allow modeling the *behavior* of data. Temporal (or dynamic) constraints place restrictions on how the data changes in time. They can arise in the context of classical databases as well as in temporal databases. In temporal databases, we can specify restrictions on the sequence of time-indexed instances using temporal logics (extensions of CALC, or modal logics). These are essentially Boolean (yes/no) temporal queries. For example, we might require that "La Pagode" not be a first-run theater (i.e., every movie shown there must have been shown in some other theater at some earlier time). An important question is how to enforce such constraints efficiently. A step in this direction is suggested by the following example.

Example 22.6.5 Suppose that *Pariscope* is extended with a time domain ranging over days, as in Example 22.6.1. The constraint that "La Pagode" is not a first-run theater can be expressed in CALC as

$$\begin{aligned}
&\forall m, s, t\,(\mathit{Pariscope}(\text{“La Pagode”}, m, s, t)\\
&\qquad \rightarrow \exists x, s', t'(\mathit{Pariscope}(x, m, s', t') \wedge x \neq \text{“La Pagode”} \wedge t' < t))
\end{aligned}$$

A naive way to enforce this constraint involves maintaining the full history of the relation *Pariscope*; this would require unbounded storage. A more efficient way involves storing only the current value of *Pariscope* and maintaining a unary relation *Shown_Before*[*Title*], which holds all movie titles that have been shown in the past at a theater other than "La Pagode." Note that the size of *Shown_Before* is bounded by the number of titles that have occurred through the history of the database but is independent of how long the database has been in existence. (Of course, if a new title is introduced each day, then *Shown_Before* will have size comparable to the full history.)

A systematic approach has been developed to maintain temporal constraints in this fashion.

For classical databases, in which no history is kept, temporal constraints can only involve transitions from the current instance to the next; this gives rise to a subset of temporal constraints, called *transition* constraints

For instance, a transition constraint can state that “salaries do not decrease” or that “the new salary of an employee is determined by the old salary and the seniority.” Such transition constraints are by far the most common kind of temporal constraint considered for databases. We discuss some ways to specify transition constraints. Clearly, these can be stated using a temporal version of CALC that can refer to the previous and next state. A notion of identity similar to object identity is useful here; otherwise we may have difficulty speaking about the old and new versions of some tuple or entity. Such identity may be provided by a key, assuming that it does not change in time.

Besides CALC, transition constraints may be stated in various other ways, including

- pre- and postconditions associated with transitions;
- extensions of classical static constraints, such as dynamic fd’s;
- computational constraints on sequences of consecutive versions of tuples.

Restrictions on updates—say, by transactional schemas—also induce temporal constraints. For instance, consider again the transactional schema in Example 22.2.1. It can be verified that all possible sequences of instances obtained by calls to the transactions of that schema satisfy the temporal constraint:

“Nobody can be a PhD student without having been a TA at some point.”

The following less desirable temporal constraint is also satisfied:

“Once a PhD student, always a PhD student.”

Overall, the connection between canned updates and temporal constraints remains largely unexplored.

A related means of specifying temporal constraints is to identify a set of update events and impose restrictions on valid sequences of events. This can be done using regular expressions. For example, suppose that the events concerning an employee are

hire, **transfer**, **promote**, **raise**, **fire**, **retire**

The valid sequences of events are all prefixes of sequences specified by the regular expression

$$\textbf{hire}[(\textbf{transfer}) + (\textbf{promote} + \epsilon)(\textbf{raise})]^*[(\textbf{retire}) + (\textbf{fire})]$$

Thus an employee is first hired, receives some number of promotions and raises, may be transferred, and finally either retires or is fired. Everybody who is promoted must also receive a raise, but raises may be received even without promotion. Such constraints appear to be particularly well suited to object-oriented databases, in which events can naturally be associated with method invocations. Some active databases (Section 22.5) can also enforce constraints on sequences of events.

Bibliographic Notes

The properties of IDM transactions were formally studied in [AV88b]. The sound and complete axiomatization for IDM transactions is provided in [KV91]. The results on simplification rules are also presented there. The language datalog$^{\neg\neg}$ and other rule-based and imperative update languages are studied in [AV88c]. Dynamic Logic Programming is discussed in [MW88b]. In particular, Example 22.1.3 is from there. The language LDL, including its update capabilities, is presented in [NT89].

IDM transactional schemas are investigated in [AV89]. Transactional schemas based on more powerful languages are discussed in [AV87, AV88a]. Patterns of object migration in object-oriented databases are studied in [Su92], using results on IDM transactional schemas. A simple update language is shown there to express the family of migration patterns characterized by regular languages; richer families of patterns are obtained by permitting conditionals in this language.

One of the earliest works on the view maintenance problem is [BC79], which focuses on determining whether an update is relevant or not. References [KP81, HK89] study the maintenance of derived data in the context of semantic data models, and [SI84] studies the maintenance of a universal relation formed from an acyclic database family. Additional works that use the approach of incremental evaluation include [BLT86, GKM92, Pai84, QW91]. Heuristics for maintaining the materialized output of a stratified datalog$^{\neg}$ program are developed in [AP87b, Küc91]. A comprehensive approach, which handles views defined using the stratified datalog and aggregate operators, is developed in [GMS93]. Reference [Cha94] addresses the issue of incremental update to materialized views in the presence of OIDs.

Testing for relevance of updates in connection with view maintenance is related to the problem of incremental maintenance of integrity constraints. References [BBC80, HMN84] develop general techniques for this problem, and approaches for deductive databases include [BDM88, LST87, Nic82].

The issue of first-order incremental definability of datalog programs was first raised in [DS92] and [DS93]. Additional research in this area includes [DT92, DST94]. A more general perspective on these kinds of problems is presented in [PI94].

An informative survey of research on the view update problem is [FC85]. One practical approach to the view update problem is to consider the underlying database and the view to be abstract data types, with the updating operations predefined by the dba [SF78, RS79]. The other practical approach is to perform a careful analysis of the syntax and semantics of a view definition to determine a unique or a small set of update translation(s) that satisfy a family of natural properties. This approach is pioneered in [DB82] and further developed in [Kel85, Kel86]. Example 22.3.6 is inspired by [Kel86]. Reference [Kel82] considers the issue of unavoidable side-effects from view updates.

The discussion of view complements and Theorem 22.3.10 is from [BS81]. Reference

[CP84] studies complexity issues in this area; for example, in the context of projective views over a single relation possibly having functional dependencies, finding a minimal complement is NP-complete. Reference [KU84] examines some of the practical shortcomings of the approach based on complementary views.

The semantics of updates on incomplete databases is investigated in [AG85] and [Gra91].

The idea of representing a database as a logical theory, as opposed to a set of atomic facts, has roots in [Kow81, NG78, Rei84]. A survey of approaches to updating logical theories, which articulates the distinction between model-based and formula-based approaches, is [Win88]. Reference [Win86] develops a model-based approach for updating theories that extends the framework of [Rei84]. Complexity and expressiveness issues related to this approach are studied in [GMR92, Win86]. A model-based approach has recently been applied in connection with supporting object migration in object-oriented databases in [MMW94].

An early formula-based approach to updating is discussed in [Tod77]. This chapter's discussion of the formula-based approach is inspired largely by [FUV83]. The notion of using flocks (i.e., families of theories) to describe incomplete information databases is developed in [FKUV86]. Reference [Var85] investigates the complexity of querying databases that are logical theories and shows that even in restricted cases, the complexity of, for example, the relational calculus goes from LOGSPACE to CO-NP-complete.

References on belief revision include [AGM85], where the AGM postulates are developed, and [Gär88, Mak85]. The contrast between belief revision and knowledge update was articulated informally in [KW85] and formally in [KM91a], where postulates for updating theories under the model-based perspective were developed; see also [GMR92, KM91b]. The discussion in this chapter is inspired by [KM91a].

Active databases generally support the automatic triggering of updates as a response to user-requested or system-generated updates. Most active database systems (e.g., [CCCR$^+$90, Coh86, MD89, Han89, SKdM92, SJGP90, WF90]) use a paradigm of *rules* to specify the actions to be taken, in a manner reminiscent of expert systems.

Active databases and techniques have been shown to be useful for constraint maintenance [Mor83, CW90, CTF88], incremental update of materialized views [CW91], and database security [SJGP90]; and they hold the promise of providing a new family of solutions to the view and derived data update problem [CHM94] and issues in database interoperability [CW93, Cha94, Wie92]. Another functionality associated with some active databases is query rewriting [SJGP90], whereby a query q might be transformed into a related query q' before being executed.

As discussed in Section 22.5 (see also [HJ91b, HW92, Sto92]), each of the active database systems described in the literature uses a different approach for event specification and a different execution model. The execution models of several active database systems are specified using deltas, either implicitly or explicitly [Coh86, SKdM92, WF90]. The Heraclitus language [HJ91a, JH91, GHJ$^+$93] elevates deltas to be first-class citizens in a database programming language based on C and the relational model, thereby enabling the specification, and thus implementation, of a wide variety of execution models. Execution models that support immediate, deferred, and concurrent firing include [BM91, HLM88, MD89].

The accumulating execution model forms part of the semantics of the AP5 active database model [Coh86, Coh89] (see also [HJ91a]). Theorem 22.5.2 is from [ZH90], which goes on to present syntactic conditions on rules that ensure the Church-Rosser property for rule bases that are not necessarily monotonic.

An early investigation of composite events in connection with active databases is [DHL91]. Reference [GJS92c] describes the event specification language of the ODE active database system [GJ91]. Reference [GJS92b] presents the equivalence of ODE's composite event specification language and regular expressions, and [GJS92a] develops an implementation technique based on finite state automata for recognizing composite events in the case where parameters are omitted. Reference [GD94] uses an alternative formalism for composite events based on Petri nets and can support parameters.

A crucial issue with regard to efficient implementation of active databases is determining incrementally when a condition becomes true. Early work in this area is modeled after the RETE algorithm from expert systems [For82]. Enhancements of this technique biased toward active database applications include [WH92, Coh89]. Reference [CW90] describes a mechanism for analyzing rule conditions to infer triggers for them.

There is a vast amount of literature on temporal databases. The volume [TCG$^+$93] provides a survey of current research in the area. In particular, several temporal extensions of SQL can be found there. Bibliographies on temporal databases are provided in [Sno90, Soo91]. A survey of temporal database research, emphasizing theoretical aspects, is provided in [Cho94]. Deductive temporal databases are presented in [BCW93]. Example 22.6.4 is from [BCW93].

Specification of transition constraints by pre- and postconditions is studied in [CCF82, CF84]. Transition constraints based on a dynamic version of functional dependencies are investigated in [Via87], where the interaction between static and dynamic fd's is discussed. Constraints of a computational flavor on sequences of objects (*object histories*) are considered in [Gin93]. Temporal constraints specified by regular languages of events (where the events refer to object migration in object-oriented databases) are studied in [Su92]. References [Cho92a, LS87] develop the approach of "history-less" checking of temporal constraints, as illustrated in Example 22.6.5. This technique is applied to testing real-time temporal constraints in [Cho92b], providing one approach to monitoring complex events in an active database system.

Temporal databases are intimately related to temporal logic. Informative overviews of temporal logic can be found in [Eme91, Gal87].

A survey of dynamic aspects in databases is provided in [Abi88].

Exercises

Exercise 22.1 Show that there are updates expressible by IDM transactions that are not expressible by ID transactions (i.e., transactions with just insertions and deletions).

Exercise 22.2 Prove the soundness of the equivalence axioms

$$mod(C \to C')del(C') \equiv del(C)del(C')$$
$$ins(t)mod(C \to C') \quad \equiv mod(C \to C')ins(t')$$
$$\text{where } t \text{ satisfies } C \text{ and } \{t'\} = mod(C \to C')(\{t\})$$

and

$$del(C_3)mod(C_1 \to C_3)mod(C_2 \to C_1)mod(C_3 \to C_2)$$
$$\equiv del(C_3)mod(C_2 \to C_3)mod(C_1 \to C_2)mod(C_3 \to C_1),$$

where C_1, C_2, C_3 are mutually exclusive sets of conditions.

Exercise 22.3 Show that, for each IDM transaction, there exists a CALC query defining the same result but that the converse is false. Characterize the portion of CALC (or ALG) expressible by IDM transactions.

Exercise 22.4 [AV88b] Show that for every IDM transaction there exists an equivalent IDM transaction of the form $t_d; t_m; t_i$, where t_d is a sequence of deletions, t_m is a sequence of modifications, and t_i is a sequence of insertions.

♠ **Exercise 22.5** [VV92] Let $t_1, \ldots t_k$ be IDM transactions over the same relation R. A *schedule* s for $t_1, \ldots, t_k$ is an interleaving of the updates in the t_i's, such that the updates of each t_i occur in s in the same order as in t_i. The schedule s is *serializable* if it is equivalent to $t_{\sigma(1)} \ldots t_{\sigma(k)}$ for some permutation σ of $\{1, \ldots, k\}$.

(a) Prove that checking whether a schedule s for a set of IDM transactions $t_1, \ldots, t_k$ is serializable is NP-complete with respect to the size of s.

(b) Show that checking the serializability of a schedule can be done in polynomial time if the transactions contain no modifications.

♠ **Exercise 22.6** [KV90a] Suppose m boxes $B_1, \ldots, B_m$ are given. Initially, each box B_i is either empty or contains some balls. Balls can be moved among boxes by any sequence of *moves*, $m(B_j, B_k)$, each of which consists of putting the entire contents of box B_j into box B_k. Suppose that the balls must be redistributed among boxes according to a given mapping f from boxes to boxes [$f(B_j) = B_k$ means that the contents of box B_j must wind up in box B_k after the redistribution].

(a) Show that redistribution according to a given mapping f cannot always be accomplished by a sequence of moves. If it can, the mapping f is called *realizable*. Characterize realizable redistribution mappings.

(b) A parallel schedule of moves is a partially ordered set of moves $(M,\leq)$ such that incomparable moves commute. (Thus incomparable moves are independent and can be executed in parallel.) A parallel schedule takes time t if the depth of the partial order is t. Show that the problem of testing if a parallel schedule of moves accomplishes the redistribution in minimal time (according to a realizable redistribution mapping) is NP-complete with respect to m.

(c) Show that testing if a parallel schedule accomplishes the redistribution in time within *one* unit from the minimal time can be done in time polynomial in m.

(d) What is the connection between moving balls and IDM transactions?

Exercise 22.7 Recall the transaction schema **T** of Example 22.2.1 and the set Σ of constraints in Example 22.2.2.

(a) Prove that **T** is sound and complete with respect to Σ.

(b) Exhibit instances **I** and **J** in $Sat(\Sigma)$, where **I** cannot be transformed into **J** using **T**.

(c) Write a transactional schema $\mathbf{T}'$ that is sound and complete for Σ, such that whenever **I**, **J** are in $Sat(\Sigma)$, there is a transformation from **I** to **J** using $\mathbf{T}'$. (Do not use a $\mathbf{T}'$ that completely empties the database to make a change involving only one student.)

Exercise 22.8 [AV89] Prove Theorem 22.2.3.

Exercise 22.9 Prove the statements in Example 22.2.4.

♠ **Exercise 22.10** [AV89]

(a) Prove that it is undecidable whether $\mathbf{I} \in Gen(\mathbf{T})$ for given IDM transactional schema **T** and instance **I** over a database schema. *Hint:* Reduce the question of whether $w \in L(M)$ for a word w and Turing machine M to the preceeding problem.

(b) Show that (a) becomes decidable if **T** is an ID transactional schema (no modifications). *Hint:* For $\mathbf{I} \in Gen(\mathbf{T})$, find a bound on the number of calls to transactions in **T** needed to reach **I** and on the number of constants used in these calls.

(c) Prove that it is undecidable whether $Gen(\mathbf{T}) = Gen(\mathbf{T}')$ for given IDM transactional schemas **T** and $\mathbf{T}'$.

♠ **Exercise 22.11** [AV89]

(a) Show that there is a relation schema R and a join dependency g over R such that $Sat(\{g\}) \neq Gen(\mathbf{T})$ for each IDM transactional schema **T** over R.

(b) Prove that there is a database schema **R** and a set Σ of inclusion dependencies over **R**, such that $Sat(\Sigma) \neq Gen(\mathbf{T})$ for each IDM transactional schema **T** over **R**.

♠ **Exercise 22.12** [AV89] Prove that it is undecidable whether $Gen(\mathbf{T})$ equals all instances over **R** for given IDM transaction schema **T** over **R**. What does this say about the decidability of soundness and completeness of IDM transaction schemas with respect to sets of constraints?

Exercise 22.13 [QW91] Develop expressions for incremental evaluation of the relational algebra operators, analogous to the expression for join in Example 22.3.3. Consider both insertions and deletions from the base relations.

Exercise 22.14 Recast c-tables in terms of first-order theories. Observe that the approach to updating c-tables is model based. Given a theory **T** corresponding to a c-table and an update, describe how to change **T** in accordance with the update. *Hint:* To represent c-tables using a theory, you will need to use variations of the equality, extension, unique name, and closure axioms mentioned at the end of Chapter 2.

Exercise 22.15 Prove Proposition 22.4.3.

Exercise 22.16 [FUV83] Given theory **T**, define $\mathbf{T}' \preceq_{\mathbf{T}} \mathbf{T}''$ if $\mathbf{T}' - \mathbf{T} \subset \mathbf{T}'' - \mathbf{T}$, or if $\mathbf{T}' - \mathbf{T} = \mathbf{T}'' - \mathbf{T}$ and $\mathbf{T} - \mathbf{T}' \subseteq \mathbf{T} - \mathbf{T}''$. Thus $\preceq_{\mathbf{T}}$ is like $\leq_{\mathbf{T}}$, except that insertions are given priority over deletions.

Let **T** be a closed theory, φ a sentence not in **T**, and $\mathbf{T}'$ a closed theory that accomplishes $[insert\ \varphi]$ for **T**. Show that $\{\varphi\}^* \preceq_{\mathbf{T}} \mathbf{T}'$.

Exercise 22.17 [FUV83] Verify the claim of Example 22.4.5.

Exercise 22.18 [FUV83] Let $R[ABC]$ be a relation schema with functional dependency $A \rightarrow B$, and let **I** be the instance of Example 22.4.5.

Consider the view f over $S[AB]$ defined by $\pi_{AB}(R)$. A complement of this view is $\pi_{AC}(R)$. The idea of keeping this complement unchanged while updating the view is captured by the sentences

$$\left\{ \begin{array}{c} \exists x(R(a, x, c)), \\ \exists x(R(a, x, c')) \\ \exists x(R(a', x, c'')) \\ \exists x(R(a'', x, c''')) \end{array} \right\}$$

Let $\mathbf{T}_0$ be that set of sentences. Let $\mathbf{T}_1$ include the functional dependency and the unique name axioms. Finally, let $\mathbf{T}_2$ include the four atoms of $\mathbf{I}$. Verify that there is a unique tagged theory that accomplishes the view update [*insert* $S(a, b'')$] with minimal change.

Exercise 22.19 [FUV83] Show that under the formula-based approach to updating theories presented in Section 22.4,

(a) A sequence of deletions can lead to an exponential blowup in the size of the theory.

(b) Determining the result of an insertion is NP-hard.

Exercise 22.20 [DT92, DS93] Give a formal definition of FOID and of FOID with auxiliary relations. Include the cases in which sets of insertions and/or deletions are permitted.

♠ **Exercise 22.21** [DT92]

(a) Verify the claim of Example 22.3.4, that the transitive closure query is FOID.

(b) Consider the datalog program

$$\begin{aligned} R(z) &\leftarrow R(x), S(x, y, z) \\ R(z) &\leftarrow R(y), S(x, y, z) \\ R(x) &\leftarrow T(x) \end{aligned}$$

An intuitive interpretation of this is that the variables range over nodes in a graph, and the predicate $S(a, b, c)$ indicates that nodes a and b are connected by an *or*-gate to node c. The relation R contains all nodes that have value *true*, assuming that the nodes in the input relation T are initially set to *true*.

Prove that there is a FOID with auxiliary relations for R. *Hint:* Define a new derived relation Q that holds paths of nodes with value *true*.

(c) Prove that there is no FOID without auxiliary relations for R.

★ (d) A *regular chain* program consists of a finite set of chain rules of the form

$$R(x, z) \leftarrow R_1(x, y_1), R_2(y_1, y_2), \ldots, R_n(y_{n-1}, z),$$

where the only idb predicate occurring in the body (if any) is R_n. Show that each regular chain program is FOID with auxiliary relations. In particular, describe an algorithm that produces, for each regular chain program defining a predicate R, a first-order query with auxiliary relations that incrementally evaluates the program.

Exercise 22.22 Specify in detail an active database execution model based on immediate rule firing.

Exercise 22.23 [ZH90] Recall the accumulating execution model for active databases.

(a) Exhibit a rule base for which the outcome of execution depends on the order of rule firing.

(b) Prove Theorem 22.5.2.

★ **Exercise 22.24** [HJ91a] Recall that in the accumulating semantics, rule conditions can access $\mathbf{I}^{orig}$ and Δ^{curr}. Consider an alternative semantics that differs from the accumulating semantics only in that the rule conditions can access only $\mathbf{I}^{orig}$ and $\mathbf{I}^{curr}$. Suppose that rule conditions have the expressive power of the relational calculus (and in the case of the accumulating semantics, the ability to access the sets $\Delta_R^+ = \{R(t) \mid +R(t) \in \Delta\}$ and $\Delta_R^- = \{R(t) \mid -R(t) \in \Delta\}$). Show that the accumulating semantics is more expressive than the alternative semantics. *Hint:* It is possible that Δ^{curr} may have "redundant" elements, e.g., an update $+R(t)$, where $R(t) \in \mathbf{I}^{orig}$. Such redundant elements are not accessible to the alternative semantics.

Exercise 22.25 Consider a base schema $\mathbf{B} = \{R[AB]\}$ and a view $f = \pi_A R$, as in Example 22.3.8(b).

(a) Describe a complement g of f that is not equivalent to $\top$.

(b) Show that each complement g of f expressible in the relational algebra is equivalent to $\top$.

Exercise 22.26 [BS81] Prove Theorem 22.3.10. *Hint:* Consider the equivalence relation on $\mathit{Inst}(\mathbf{B})$ defined by $\mathbf{I} \equiv \mathbf{I}'$ iff $\exists$ update $\nu \in U_f$ such that $\mathbf{I}' = t(\nu)(\mathbf{I})$. Now define the mapping $g : \mathit{Inst}(\mathbf{I}) \rightarrow \mathit{Inst}(\mathbf{I})/\equiv$ so that $g(\mathbf{I})$ is the equivalence class of $\mathbf{I}$ under $\equiv$.