PART

F Finale

In this part, we consider four advanced topics. Two of them (incomplete information and dynamic aspects) have been studied for a while, but for some reason (perhaps their difficulty) they have never reached the maturity of more established areas such as dependency theory. Interest in the other two topics (complex values and object databases) is more recent, and our understanding of them is rudimentary. In all cases, no clear consensus has yet emerged. Our choice of material, as well as our presentation, are therefore unavoidably more subjective than in other parts of this book. However, the importance of these issues for practical systems, as well as the interesting theoretical issues they raise, led us to incorporate a discussion of them in this book.

In Chapter 19, we address the issue of incomplete information. In many database applications, the knowledge of the real world is incomplete. It is crucial to be able to handle such incompleteness and, in particular, to be able to ask queries and perform updates. Chapter 19 surveys various models of incomplete databases, research directions, and some results.

In Chapter 20, we present an extension of relations called complex values. These are obtained from atomic elements using tuple and set constructors. The richer structure allows us to overcome some limitations of the relational model in describing complex data. We generalize results obtained for the relational model; in particular, we present a calculus and an equivalent algebra.

Chapter 21 looks at another way to enrich the relational model by introducing a number of features borrowed and adapted from object-oriented programming, such as objects, classes, and inheritance. In particular, objects consist of a structural part (a data repository) and a behavioral part (pieces of code). Thus the extended framework encompasses behavior, a notion conspicuously absent from relational databases.

Chapter 22 deals with dynamic aspects. This is one of the less settled areas in databases, and it raises interesting and difficult questions. We skim through a variety of issues: languages and semantics for updates; updating views; updating incomplete information; and active and temporal databases.

A comprehensive vision of the four areas discussed in Part F is lacking. The reader should therefore keep in mind that some of the material presented is in flux, and its importance pertains more to the general flavor than the specific results.

19 Incomplete Information

Somebody: *What are we doing next?*
Alice: *Who are* we*? Who are* you*?*
Somebody: We *are you and the authors of the book, and* I *am one of them. This is an instance of incomplete information.*
Somebody: *It's not much, but we can still tell that* surely *one of us is Alice and that there are* possibly *up to three "Somebodies" speaking.*

In the previous parts, we have assumed that a database always records information that is completely known. Thus a database has consisted of a completely determined finite instance. In reality, we often must deal with incomplete information. This can be of many kinds. There can be missing information, as in "John bought a car but I don't know which one." In the case of John's car, the information exists but we do not have it. In other cases, some attributes may be relevant only to some tuples and irrelevant to others. Alice is single, so the spouse field is irrelevant in her case. Furthermore, some information may be imprecise: "Heather lives in a large and cheap apartment," where the values of *large* and *cheap* are fuzzy. Partial information may also arise when we cannot completely rely on the data because of possible inconsistencies (e.g., resulting from merging data from different sources).

As soon as we leave the realm of complete databases, most issues become much more intricate. To deal with the most general case, we need something resembling a theory of knowledge. In particular, this quickly leads to logics with modalities: Is it *certain* that John lives in Paris? Is it *possible* that he may? What is the *probability* that he does? Does John *know* that Alice is a good student? Does he *believe* so? etc.

The study of knowledge is a fascinating topic that is outside the scope of this book. Clearly, there is a trade-off between the expressivity of the model for incomplete information used and the difficulty of answering queries. From the database perspective, we are primarily concerned with identifying this trade-off and understanding the limits of what is feasible in this context. The purpose of this chapter is to make a brief foray into this topic. We limit ourselves mostly to models and results of a clear database nature. We consider simple forms of incompleteness represented by null values. The main problem we examine is how to answer queries on such databases. In relation to this, we argue that for a representation system of incomplete information to be adequate in the context of a query language, it must also be capable of representing *answers* to queries. This leads to a desirable closure property of representations of incomplete information with respect to query languages. We observe the increase of complexity resulting from the use of nulls.

We also consider briefly two approaches closer to knowledge bases. The first is based

on the introduction of disjunctions in deductive databases, which also leads to a form of incompleteness. The second is concerned with the use of modalities. We briefly mention the language KL, which permits us to talk about knowledge of the world.

19.1 Warm-Up

As we have seen, there are many possible kinds of incomplete information. In this section, we will focus on databases that partially specify the state of the world. Instead of completely identifying one state of the world, the database contents are compatible with many possible worlds. In this spirit, we define an *incomplete database* simply as a set of possible worlds (i.e., a set of instances). What is actually stored is a *representation* of an incomplete database. Choosing appropriate representations is a central issue.

We provide a mechanism for representing incomplete information using *null values*. The basic idea is to allow occurrences of variables in the tuples of the database. The different possible values of the variables yield the possible worlds.

The simplest model that we consider is the Codd table (introduced by Codd), or *table* for short. A table is a relation with constants and variables, in which no variable occurs twice. More precisely, let U be a finite set of attributes. A *table* T over U is a finite set of free tuples over U such that each variable occurs at most once. An example of a table is given in Fig. 19.1. The figure also illustrates an alternative representation (using @) that is more visual but that we do not adopt here because it is more difficult to generalize.

The preceding definition easily extends to database schemas. A database table $\mathbf{T}$ over a database schema $\mathbf{R}$ is a mapping over $\mathbf{R}$ such that for each R in $\mathbf{R}$, $\mathbf{T}(R)$ is a table over $sort(R)$. For this generalization, we assume that the sets of variables appearing in each table are *pairwise disjoint*. Relationships between the variables can be stated through

R	A	B	C
	0	1	x
	y	z	1
	2	0	v

Table T

R	A	B	C
	0	1	@
	@	@	1
	2	0	@

Alternative representation of T

R	A	B	C
	0	1	2
	2	0	1
	2	0	0

I_1

R	A	B	C
	0	1	2
	3	0	1
	2	0	5

I_2

R	A	B	C
	0	1	2
	2	0	1
	2	0	0

I_3

R	A	B	C
	0	1	1
	2	0	1

I_4

Figure 19.1: A table and examples of corresponding instances

global conditions (which we will introduce in the next section). In this section, we will focus on single tables, which illustrate well the main issues.

To specify the semantics of a table, we use the notion of valuation (see Chapter 4). The incomplete database represented by a table is defined as follows:

$$rep(T) = \{\nu(T) \mid \nu \text{ a valuation of the variables in } T\}.$$

Consider the table T in Fig. 19.1. Then $I_1, \ldots, I_4$ all belong to $rep(T)$ (i.e., are possible worlds).

The preceding definition assumes the Closed World Assumption (CWA) (see Chapter 2). This is because each tuple in an instance of $ref(T)$ must be justified by the presence of a particular free tuple in T. An alternative approach is to use the Open World Assumption (OWA). In that case, the incomplete database of T would include all instances that contain an instance of $rep(T)$. In general, the choice of CWA versus OWA does not substantially affect the results obtained for incomplete databases.

We now have a simple way of representing incomplete information. What next? Naturally, we wish to be able to query the incomplete database. Exactly what this means is not clear at this point. We next look at this issue and argue that the simple model of tables has serious shortcomings with respect to queries. This will naturally lead to an extension of tables that models more complicated situations.

Let us consider what querying an incomplete database might mean. Consider a table T and a query q. The table T represents a set of possible worlds $rep(T)$. For each $I \in rep(T)$, q would produce an answer $q(I)$. Therefore the set of possible answers of q is $q(rep(T))$. This is, again, an incomplete database. The answer to q should be a representation of this incomplete database.

More generally, consider some particular representation system (e.g., tables). Such a system involves a language for describing representations and a mapping *rep* that associates a set of instances with each representation. Suppose that we are interested in a particular query language $\mathcal{L}$ (e.g., relational algebra). We would always like to be capable of representing the result of a query in the same system. More precisely, for each representation T and query q, there should exist a computable representation $\overline{q}(T)$ such that

$$rep(\overline{q}(T)) = q(rep(T)).$$

In other words, $\overline{q}(T)$ represents the possible answers of q [i.e., $\{q(I) \mid I \in rep(T)\}$].

If some representation system τ has the property described for a query language $\mathcal{L}$, we will say that τ is a *strong representation system* for $\mathcal{L}$. Clearly, we are particularly interested in strong representation systems for relational algebra and we shall develop such a system later.

Let us now return to tables. Unfortunately, we quickly run into trouble when asking queries against them, as the following example shows.

EXAMPLE 19.1.1 Consider T of Fig. 19.1 and the algebraic query $\sigma_{A=3}(T)$. There is no table representing the possible answers to this query. A possible answer (e.g., for I_1) is the empty relation, whereas there are nonempty possible answers (e.g., for I_2). Suppose

that there exists a table T' representing the set of possible answers. Either T' is empty and $\sigma_{A=3}(I_2)$ is not in $rep(T')$; or T' is nonempty and the empty relation is not in $rep(T')$. This is a contradiction, so no such T' can exist.

The problem lies in the weakness of the representation system of tables; we will consider richer representation systems that lead to a complete representation system for all of relational algebra. An alternative approach is to be less demanding; we consider this next and present the notion of *weak* representation systems.

19.2 Weak Representation Systems

To relax our expectations, we will no longer require that the answer to a query be a representation of the set of all possible answers. Instead we will ask which are the tuples that are surely in the answer (i.e., that belong to *all* possible answers). (Similarly, we may ask for the tuples that are possibly in the answer (i.e., that belong to *some* possible answer). We make this more precise next.

For a table T and a query q, the set of sure facts, $sure(q, T)$, is defined as

$$sure(q, T) = \cap\{q(I) \mid I \in rep(T)\}.$$

Clearly, a tuple is in $sure(q, T)$ iff it is in the answer for every possible world. Observe that the sure tuples in a table T [i.e., the tuples in every possible world in $rep(T)$] can be computed easily by dropping all free tuples with variables. One could similarly define the set $poss(q, T)$ of possible facts.

One might be tempted to require of a weak system just the ability to represent the set of tuples surely in the answer. However, the definition requires some care due to the following subtlety. Suppose T is the table in Fig. 19.1 and q the query $\sigma_{A=2}(R)$, for which $sure(q, T) = \emptyset$. Consider now the query $q' = \pi_{AB}(R)$ and the query $q \circ q'$. Clearly, $q'(sure(q, T)) = \emptyset$; however, $sure(q'(q(rep(T))) = \{\langle 2, 0\rangle\}$. So $q \circ q'$ cannot be computed by first computing the tuples surely returned by q and then applying q'. This is rather unpleasant because generally it is desirable that the semantics of queries be compositional (i.e., the result of $q \circ q'$ should be obtained by applying q' to the result of q). The conclusion is that the answer to q should provide more information than just $sure(q, T)$; the incomplete database it specifies should be equivalent to $q(rep(T))$ with respect to its ability to compute the sure tuples of any query in the language applied to it. This notion of equivalence of two incomplete databases is formalized as follows.

If $\mathcal{L}$ is a query language, we will say that two incomplete databases $\mathcal{I}, \mathcal{J}$ are $\mathcal{L}$ equivalent, denoted $\mathcal{I} \equiv_{\mathcal{L}} \mathcal{J}$, if for each q in $\mathcal{L}$ we have

$$\cap\{q(I) \mid I \in \mathcal{I}\} = \cap\{q(I) \mid I \in \mathcal{J}\}.$$

In other words, the two incomplete databases are undistinguishable if all we can ask for is the set of sure tuples in answers to queries in $\mathcal{L}$.

We can now define weak representation systems. Suppose $\mathcal{L}$ is a query language. A

representation system is *weak* for $\mathcal{L}$ if for each representation T of an incomplete database, and each q in $\mathcal{L}$, there exists a representation denoted $\overline{q}(T)$ such that

$$rep(\overline{q}(T)) \equiv_{\mathcal{L}} q(rep(T)).$$

With the preceding definition, $\overline{q}(T)$ does not provide precisely $sure(q, T)$ for tables T. However, note that $sure(q, T)$ can be obtained at the end simply by eliminating from the answer all rows with occurrences of variables.

The next result indicates the power of tables as a weak representation system.

THEOREM 19.2.1 Tables form a weak representation system for selection-projection (SP) [i.e., relational algebra limited to selection (involving only equalities and inequalities) and projection]. If union or join are added, tables no longer form a weak representation system.

Crux It is easy to see that tables form a weak representation system for SP queries. Selections operate conservatively on tables. For example,

$$\overline{\sigma_{cond}}(T) = \{t \mid t \in T \text{ and } cond(\nu(t)) \text{ holds for all valuations } \nu \text{ of the variables in } t\}.$$

Projections operate like classical projections. For example, if T is again the table in Fig. 19.1, then

$$\overline{\sigma_{A=2}}(T) = \{\langle 2, 0, v\rangle\}$$

and

$$\overline{(\pi_{AB}(R) \circ \sigma_{A=2}(R))}(T) = \{\langle 2, 0\rangle\}.$$

Let us show that tables are no longer a weak representation system if join or union are added to SP. Consider join first. So the query language is now SPJ. Let T be the table

R	A	B	C
	a	x	c
	a'	x'	c'

where x, x' are variables and a, a', c, c' are constants.

Let $q = \pi_{AC}(R) \bowtie \pi_B(R)$. Suppose there is table W such that

$$rep(W) \equiv_{SPJ} q(rep(T)),$$

and consider the query $q' = \pi_{AC}(\pi_{AB}(R) \bowtie \pi_{BC}(R))$. Clearly, $sure(q \circ q', T)$ is

A	C
a	c
a'	c
a	c'
a'	c'

Therefore $sure(q', W)$ must be the same. Because $\langle a', c\rangle \in sure(q', W)$, for each valuation ν of variables in W there must exist tuples $u, v \in W$ such that $u(A) = a'$, $v(C) = c$, $\nu(u)(B) = \nu(v)(B)$. Let ν be a valuation such that $\nu(z) \neq \nu(y)$ for all variables z, y, $z \neq y$. If $u = v$, then $u(A) = a'$ and $u(C) = c$ so $\langle a', c\rangle \in sure(\pi_{AC}(R), W)$. This cannot be because, clearly, $\langle a', c\rangle \notin sure(\pi_{AC}(R), q(rep(T)))$. So, $u \neq v$. Because $\nu(u)(B) = \nu(v)(B)$ and W has no repeated variables, it follows that $u(B)$ and $v(B)$ equal some constant k. But then $\langle a', k\rangle \in sure(\pi_{AB}(R), W)$, which again cannot be because one can easily verify that $sure(\pi_{AB}(R), q(rep(T))) = \emptyset$.

The proof that tables do not provide a weak representation system for SPU follows similar lines. Just consider the table T

R	A	B
	x	b

and the query q outputting two relations: $\sigma_{A=a}(R)$ and $\sigma_{A\neq a}(R)$. It is easily seen that there is no pair of tables W_1, W_2 weakly representing $q(rep(T))$ with respect to SPU. To see this, consider the query $q' = \pi_B(W_1 \cup W_2)$. The details are left to the reader (Exercise 19.7). ■

Naive Tables

The previous result shows the limitations of tables, even as weak representation systems. As seen from the proof of Theorem 19.2.1, one problem is the lack of repeated variables. We next consider a first extension of tables that allows repetitions of variables. It will turn out that this will provide a weak representation system for a large subset of relational algebra.

A *naive* table is like a table except that variables may repeat. A naive table is shown in Fig. 19.2. Naive tables behave beautifully with respect to positive existential queries (i.e., conjunctive queries augmented with union). Recall that, in terms of the algebra, this is SPJU.

THEOREM 19.2.2 Naive tables form a weak representation system for positive relational algebra.

Crux Given a naive table T and a positive query q, the evaluation of $\overline{q}(T)$ is extremely simple. The variables are treated as distinct new constants. The standard evaluation of q is then performed on the table. Note that incomplete information yields no extra cost in this case. We leave it to the reader to verify that this works. ■

R	A	B	C
	0	1	x
	x	z	1
	2	0	v

Figure 19.2: A naive table

Naive tables yield a nice representation system for a rather large language. But the representation system is weak and the language does not cover all of relational algebra. We introduce in the next section a representation that is a strong system for relational algebra.

19.3 Conditional Tables

We have seen that Codd tables and naive tables are not rich enough to provide a strong representation system for relational algebra. To see what is missing, recall that when we attempt to represent the result of a selection on a table, we run into the problem that the presence or absence of certain tuples in a possible answer is conditioned by certain properties of the valuation. To capture this, we extend the representation with conditions on variables, which yields conditional tables. We will show that such tables form a strong representation system for relational algebra.

A *condition* is a *conjunct* of *equality atoms* of the form $x = y$, $x = c$ and of *inequality atoms* of the form $x \neq y$, $x \neq c$, where x and y are variables and c is a constant. Note that we only use conjuncts of atoms and that the Boolean *true* and *false* can be respectively encoded as atoms $x = x$ and $x \neq x$.

If formula Φ is a condition, we say that a valuation ν *satisfies* Φ if its assignment of constants to variables makes the formula true.

Conditions may be associated with table T in two ways: (1) A *global* condition Φ_T is associated with the entire table T; (2) a *local* condition φ_t is associated with one tuple t of table T. A *conditional table* (*c-table* for short) is a triple (T, Φ_T, φ), where

- T is a table,
- Φ_T is a global condition,
- φ is a mapping over T that associates a local condition φ_t with each tuple t of T.

A c-table is shown in Fig. 19.3. If we omit listing a condition, then it is by default the atom *true*. Note also that conditions Φ_T and φ_t for t in T may contain variables not appearing respectively in T or t.

For our purposes, the global conditions in c-tables could be distributed at the tuple level as local conditions. However, they are convenient as shorthand and when dependencies are considered.

For brevity, we usually refer to a c-table (T, Φ_T, φ) simply as T. A given c-table T represents a set of instances as follows (again adopting the CWA):

T'

A	B	
$x \neq 2, y \neq 2$		
0	1	$z = z$
1	x	$y = 0$
y	x	$x \neq y$

J_1

A	B
0	1
0	0

J_2

A	B
0	1
1	0

J_3

A	B
0	1

J_4

A	B
0	1
0	3

Figure 19.3: A c-table and some possible instances

$$rep(T) = \{I \mid \text{there is a valuation } \nu \text{ satisfying } \Phi_T \text{ such that relation } I \text{ consists exactly of those facts } \nu(t) \text{ for which } \nu \text{ satisfies } \varphi_t\}.$$

Consider the table T' in Fig. 19.3. Then J_1, J_2, J_3, J_4 are obtained by valuating x, y, z to (0,0,0), (0,1,0), (1,0,0), and (3,0,0), respectively.

The next example illustrates the considerable power of the local conditions of c-tables, including the ability to capture disjunctive information.

EXAMPLE 19.3.1 Suppose we know that Sally is taking math or computer science (CS) (but not both) and another course; Alice takes biology if Sally takes math, and math or physics (but not both) if Sally takes physics. This can be represented by the following c-table:

Student	*Course*	
	$(x \neq math) \wedge (x \neq CS)$	
Sally	*math*	$(z = 0)$
Sally	*CS*	$(z \neq 0)$
Sally	x	
Alice	*biology*	$(z = 0)$
Alice	*math*	$(x = physics) \wedge (t = 0)$
Alice	*physics*	$(x = physics) \wedge (t \neq 0)$

Observe that there may be several c-table representations for the same incomplete database. Two representations T, T' are said to be equivalent, denoted $T \equiv T'$, if $rep(T) = rep(T')$. Testing for equivalence of c-tables is not a trivial task. Just testing membership of an instance in $rep(T)$, apparently a simpler task, will be shown to be NP-complete. To test equivalence of two c-tables T and T', one must show that for each valuation ν of the variables in T there exists a valuation ν' for T' such that $\nu(T) = \nu'(T')$, and conversely. Fortunately, it can be shown that one need only consider valuations to a set C of constants containing all constants in T or T' and whose size is at most the number of variables in the two tables (Exercise 19.11). This shows that equivalence of c-tables is decidable.

In particular, finding a minimal representation can be hard. This may affect the computation of the result of a query in various ways: The complexity of computing the answer may depend on the representation of the input; and one may require the result to be somewhat compact (e.g., not to contain tuples with unsatisfiable local conditions).

It turns out that c-tables form a strong representation system for relational algebra.

THEOREM 19.3.2 For each c-table T over U and relational algebra query q over U, one can construct a c-table $\overline{q}(T)$ such that $rep(\overline{q}(T)) = q(rep(T))$.

Crux The proof is straightforward and is left as an exercise (Exercise 19.13). The example in Fig. 19.4 should clarify the construction.[1] For projection, it suffices to project the columns of the table. Selection is performed by adding new conjuncts to the local conditions. Union is represented by the union of the two tables (after making sure that they use distinct sets of variables) and choosing the appropriate local conditions. Join and intersection involve considering all pairs of tuples from the two tables. For difference, we consider a tuple in the first table and add a huge conjunct stating that it does not match any tuple from the second table (disjunctions may be used as shorthand; they can be simulated using new variables, as illustrated in Example 19.3.1). ■

To conclude this section, we consider (1) languages with recursion, and (2) dependencies. In both cases (and for related reasons) the aforementioned representation system behaves well. The presentation is by examples, but the formal results can be derived easily.

Languages with Recursion

Consider an incomplete database and a query involving fixpoint. For instance, consider the table in Fig. 19.5. The representation $\overline{tc}(T)$ of the answer to the transitive closure query tc is also given in the same figure. One can easily verify that

$$rep(\overline{tc}(T)) = tc(rep(T)).$$

This can be generalized to arbitrary languages with iteration. For example, consider a c-table T and a relational algebra query q that we want to iterate until a fixpoint is reached.

[1] The representations in the tables can be simplified; they are given in rough form to illustrate the proof technique.

T_1	B	C
	x	c

T_2	B	C	
	y	c	$(y = b)$
	z	w	

$\overline{\pi_B(T_2)}$	B	
	y	$(y = b)$
	z	

T_3	A	B
	a	y

$\overline{\sigma_{B=b}(T_1 \bowtie T_3)}$	A	B	C	
	a	y	c	$(y = x) \wedge (y = b)$

$\overline{T_1 \cup T_2}$	B	C	
	x	c	
	y	c	$(y = b)$
	z	w	

$\overline{T_1 \bowtie T_3}$	A	B	C	
	a	y	c	$(y = x)$

$\overline{T_1 - T_2}$	B	C	
	x	c	$(y \neq b) \wedge (x \neq z)$
	x	c	$(y \neq b) \wedge (w \neq c)$
	x	c	$(y = b) \wedge (x \neq b) \wedge (x \neq z)$
	x	c	$(y = b) \wedge (x \neq b) \wedge (w \neq c)$

Figure 19.4: Computing with c-tables

Then we can construct the sequence of c-tables:

$$\overline{q}(T), \overline{q}^2(T), \ldots, \overline{q}^i(T), \ldots.$$

Suppose now that q is a positive query. We are guaranteed to reach a fixpoint on every single complete instance. However, this does not a priori imply that the sequence of representations $\{\overline{q}^i(T)\}_{i>0}$ converges. Nonetheless, we can show that this is in fact the case. For some i,

$$rep(\overline{q}^i(T)) = rep(\overline{q}^{i+1}(T)).$$

T

A	B
a	b
x	c
c	d

$\overline{tc}(T)$

A	B	
a	b	
x	c	
c	d	
a	c	$x = b$
x	d	
c	c	$x = d$
a	d	$x = b$

Figure 19.5: Transitive closure of a table

(See Exercise 19.17.) It can also be shown easily that for such i, every $I \in rep(\overline{q}^i(T))$ is a fixpoint of q. The proof is by contradiction: Suppose there is $I \in rep(\overline{q}^i(T))$ such that $q(I) \neq I$, and consider one such I with a minimum number of tuples. Because $rep(\overline{q}^i(T)) = rep(\overline{q}^{i+1}(T))$, $I = q(J)$ for some $J \in rep(\overline{q}^i(T))$. Because q is positive, $J \subseteq I$; so because $q(I) \neq I$, $J \subset I$. This contradicts the minimality of I. So $\overline{q}^i(T))$ is indeed the desired answer.

Thus to find the table representing the result, it suffices to compute the sequence $\{\overline{q}^i(T)\}_{i>0}$ and stop when two consecutive tables are equivalent.

Dependencies

In Part B, we studied dependencies in the context of complete databases. We now reconsider dependencies in the context of incomplete information. Suppose we are given an incomplete database (i.e., a set $\mathcal{I}$ of complete databases) and are told, in addition, that some set Σ of dependencies is satisfied. The question arises: How should we interpret the combined information provided by $\mathcal{I}$ and by Σ?

The answer depends on our view of the information provided by an incomplete database. Dependencies should add to the information we have. But how do we compare incomplete databases with respect to information content? One common-sense approach, in line with our discussion so far, is that more information means reducing further the set of possible worlds. Thus an incomplete database $\mathcal{I}$ (i.e., a set of possible worlds) is more informative than $\mathcal{J}$ iff $\mathcal{I} \subset \mathcal{J}$. In this spirit, the natural use of dependencies would be to eliminate from $\mathcal{I}$ those possible worlds not satisfying Σ. This makes sense for egd's (and in particular fd's).

A different approach may be more natural in the context of tgd's. This approach stems from a relaxation of the CWA that is related to the OWA. Let $\mathcal{I}$ be an incomplete database, and let Σ be a set of dependencies. Recall that tgd's imply the presence of certain tuples based on the presence of other tuples. Suppose that for some $I \in \mathcal{I}$, a tuple t implied by a tgd in Σ is not present in I. Under the relaxation of the CWA, we conclude that t should be viewed as present in I, even though it is not represented explicitly. More generally, the chase (see Chapter 8), suitably generalized to operate on instances rather than tableaux,

I_1

A	B	C
a	b	c
a	b'	c'

I_2

A	B	C
e	f	g
e	f'	g'
e	f	g'
e	f'	g

I_3

A	B	C
a	b	c
g	b	h

J_1

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

J_2

A	B	C
e	f	g
e	f'	g'
e	f	g
e	f'	g

Figure 19.6: Incomplete databases and dependencies

can be used to complete the instance by adding all missing tuples implied by the tgd's in Σ. (See Exercise 19.18.)

In fact, the chase can be used for both egd's and tgd's. In contrast to tgd's, the effect of chasing with egd's (and, in particular, fd's) may be to eliminate possible worlds that violate them. Note that tuples added by tgd's may lead to violations of egd's. This suggests that an incomplete database $\mathcal{I}$ with a set Σ of dependencies represents

$$\{chase(I, \Sigma) \mid I \in \mathcal{I} \text{ and the chase of } I \text{ by } \Sigma \text{ succeeds}\}.$$

For example, consider Fig. 19.6, which shows the incomplete database $\mathcal{I} = \{I_1, I_2, I_3\}$. Under this perspective, the incorporation of the dependencies $\Sigma = \{A \twoheadrightarrow B, B \rightarrow A\}$ in this incomplete database leads to $\mathcal{J} = \{J_1, J_2\}$.

Suppose now that the incomplete database $\mathcal{I}$ is represented as a c-table T. Can the effect of a set Σ of full dependencies on T be represented by another c-table T'? The answer is yes, and T' is obtained by extending the chase to c-tables in the straightforward way. For example, a table T_1 and its completion T_2 by $\Sigma = \{A \twoheadrightarrow B, C \rightarrow D\}$ are given in Fig. 19.7. The reader might want to check that

$$chase_\Sigma(rep(T_1)) = rep(T_2).$$

T_1

A	B	C	D
a	b	c	d
x	e	y	g
a	b	c	z

T_2

A	B	C	D	
a	b	c	d	
x	e	y	g	
a	b	y	g	$(x = a)$
a	e	c	d	$(x = a)$

Figure 19.7: c-tables and dependencies

19.4 The Complexity of Nulls

Conditional tables may appear to be a minor variation from the original model of complete relational databases. However, we see next that the use of nulls easily leads to intractability. This painfully highlights the trade-off between modeling power and resources.

We consider some basic computational questions about incomplete information databases. Perhaps the simplest question is the *possibility* problem: "Given a set of possible worlds (specified, for instance, by a c-table) and a set of tuples, is there a possible world where these tuples are all true?" A second question is the *certainty* problem: "Given a set of possible worlds and a set of tuples, are these tuples all true in every possible world?" Natural variations of these problems involve queries: Is a given set of tuples possibly (or certainly) in the answer to query q?

Consider a (c-) table T, a query q, a relation I, and a tuple t. Some typical questions include the following:

(Membership) Is I a possible world for T [i.e., $I \in rep(T)$]?

(Possibility) Is t possible [i.e., $\exists I \in rep(T)(t \in I)$]?

(Certainty) Is t certain [i.e., $\forall I \in rep(T)(t \in I)$]?

(q-Membership) Is I a possible answer for q and T [i.e., $I \in q(rep(T))$]?

(q-Possibility) Is t possibly in the answer [i.e., $\exists I \in rep(T)(t \in q(I))$]?

(q-Certainty) Is t certainly in the answer [i.e., $\forall I \in rep(T)(t \in q(I))$]?

Finally we may consider the following generalizations of the q-membership problem:

(q-Containment) Is T contained in $q(T')$ [i.e., $rep(T) \subseteq q(rep(T'))$]?

(q, q'-Containment) Is $q(T)$ contained in $q'(T)$ [i.e., $rep(q(T)) \subseteq rep(q'(T))$]?

The crucial difference between complete and incomplete information is the large number of possible valuations for the latter case. Because of the finite number of variables in a set of c-tables, only a finite number of valuations are nonisomorphic (see Exercise 19.10). However, the number of such valuations may grow exponentially in the input size. By simple reasoning about all valuations and by guessing particular valuations, we have some easy upper bounds. For a query q that can be evaluated in polynomial time on complete databases, deciding whether $I \in q(rep(T))$, or whether I is a set of possible answers, can be answered in NP; checking whether $q(rep(T)) = \{I\}$, or if I is a set of certain tuples, is in co-NP.

To illustrate such complexity results, we demonstrate one lower bound concerning the q-membership problem for (Codd) tables.

Proposition 19.4.1 There exists a positive existential query q such that checking, given a table T and a complete instance I, whether $I \in q(rep(T))$ is NP-complete.

Proof The proof is by reduction of graph 3-colorability. For simplicity, we use a query mapping a two-relation database into another two-relation database. (An easy modification of the proof shows that the result also holds for databases with one relation. In particular,

increase the arity of the largest relation, and use constants in the extra column to encode several relations into this one.)

We will use (1) an input schema $\mathbf{R}$ with two relations R, S of arity 5 and 2, respectively; (2) an output schema $\mathbf{R}'$ with two relations R', S' of arity 3 and 1, respectively; and (3) a positive existential query q from $\mathbf{R}$ to $\mathbf{R}'$. The query q [returning, on each input $\mathbf{I}$ over $\mathbf{R}$, two relations $q_1(\mathbf{I})$ and $q_2(\mathbf{I})$ over R' and S'] is defined as follows:

$$\begin{aligned} q_1 = \{\langle x, z, z'\rangle \mid \exists y([\exists vw(R(x, y, v, w, z) \vee R(v, w, x, y, z))] \\ \wedge [\exists vw(R(x, y, v, w, z') \vee R(v, w, x, y, z'))])\} \\ q_2 = \{z \mid \exists xyvw(R(x, y, v, w, z) \wedge S(y, w))\}. \end{aligned}$$

For each input $G = (V, E)$ to the graph 3-colorability problem, we construct a table $\mathbf{T}$ over the input schema $\mathbf{R}$ and an instance $\mathbf{I}'$ over the output schema $\mathbf{R}'$, such that G is 3-colorable iff $\mathbf{I}' \in q(rep(\mathbf{T}))$.

Without loss of generality, assume that G has no self-loops and that E is a binary relation, where we list each edge once with an arbitrary orientation.

Let $V = \{a_i \mid i \in [1..n]\}$ and $E = \{(b_j, c_j) \mid j \in [1..m]\}$. Let $\{x_j \mid j \in [1..m]\}$ and $\{y_j \mid j \in [1..m]\}$ be two disjoint sets of distinct variables. Then $\mathbf{T}$ and $\mathbf{I}'$ are constructed as follows:

(a) $\mathbf{T}(R) = \{t_j \mid j \in [1..m]\}$, where t_j is the tuple $\langle b_j, x_j, c_j, y_j, j\rangle$;

(b) $\mathbf{T}(S) = \{\langle i, j\rangle \mid i, j \in \{1, 2, 3\}, i \neq j\}$;

(c) $\mathbf{I}'(R') = \{\langle a, j, k\rangle \mid a \in \{b_j, c_j\} \cap \{b_k, c_k\}$, where each (b, c) pair is an edge in $E\}$; and

(d) $\mathbf{I}'(S') = \{j \mid j \in [1..m]\}$.

Intuitively, for each tuple in $\mathbf{I}(R)$, the second column contains the color of the vertex in the first column, and the fourth column contains the color of the vertex in the third column. The edges are numbered in the fifth column. The role of query q_2 is to check whether this provides an assignment of the three colors $\{1, 2, 3\}$ to vertexes such that the colors of the endpoints of each edge are distinct. Indeed, q_2 returns the edges z for which the colors y, w of its endpoints are among $\{1, 2, 3\}$. So if $q(\mathbf{I})(S') = \mathbf{I}'(S')$, then all edges have color assignments among $\{1, 2, 3\}$ to their endpoints. Next query q_1 checks whether a vertex is assigned the same color consistently in all edges where it occurs. It returns the $\langle x, z, z'\rangle$, where x is a vertex, z and z' are edges, x occurs as an endpoint, and x has the same color assignment y in both z and z'. So if $q_1(\mathbf{I})(R') = \mathbf{I}'(R')$, it follows that the color assignment is consistent everywhere for all vertexes.

For example, consider the graph G given in Fig. 19.8; the corresponding $\mathbf{I}'$ and $\mathbf{T}$ are exhibited in Fig. 19.9. Suppose that f is a 3-coloring of G. Consider the valuation σ defined by $\sigma(x_j) = f(b_j)$ and $\sigma(y_j) = f(c_j)$ for all j. It is easily seen that $\mathbf{I}' = q(\sigma(T))$. Moreover, it is straightforward to show that G is 3-colorable iff $\mathbf{I}'$ is in $q(rep(\mathbf{T}))$. ■

1	2
2	3
3	4
4	1
3	1

Figure 19.8: Graph G

$\mathbf{T}(R)$					$\mathbf{T}(S)$		$\mathbf{I}'(R')$			$\mathbf{I}'(S')$
1	x_1	2	y_1	1	1	2	1	1	1	1
2	x_2	3	y_2	2	1	3	1	1	4	2
3	x_3	4	y_3	3	2	1	1	1	5	3
4	x_4	1	y_4	4	2	3	1	4	1	4
3	x_5	1	y_5	5	3	1	1	4	4	5
					3	2	1	4	5	
							2	1	1	
							2	1	2	
							2	2	1	
							2	2	2	
								⋮		
							4	3	3	
							4	3	4	
							4	4	3	
							4	4	4	

Figure 19.9: Encoding for the reduction of 3-colorability

19.5 Other Approaches

Incomplete information often arises naturally, even when the focus is on complete databases. For example, the information in a view is by nature incomplete, which in particular leads to problems when trying to update the view (as discussed in Chapter 22); and we already considered relations with nulls in the weak universal relations of Chapter 11.

In this section, we briefly present some other aspects of incomplete information. We consider some alternative kinds of null values; we look at disjunctive deductive databases; we mention a language that allows us to address directly in queries the issue of incompleteness; and we briefly mention several situations in which incomplete information arises naturally, even when the database itself is complete. An additional approach to representing incomplete information, which stems from using explicit logical theories, will be presented in connection with the view update problem in Chapter 22.

Other Nulls in Brief

So far we have focused on a specific kind of null value denoting values that are unknown. Other forms of nulls may be considered. We may consider, for instance, *nonexisting* nulls. For example, in the tuple representing a CEO, the field DirectManager has no meaning and therefore contains a nonexisting null. Nonexisting nulls are at the core of the weak universal model that we considered in Chapter 11.

It may also be the case that we do not know for a specific field if a value exists. For example, if the database ignores the marital status of a particular person, the spouse field is either unknown or nonexisting. It is possible to develop a formal treatment of such *no-information* nulls. An incomplete database consists of a set of sets of tuples, where each set of tuples is closed under projection. This closure under projection indicates that if a tuple is known to be true, the projections of this tuple (although less informative) are also known to be true. (The reader may want to try, as a nontrivial exercise, to define tables formally with such nulls and obtain a closure theorem analogous to Theorem 19.3.2.)

For each new form of null values, the game is to obtain some form of representation with clear semantics and try to obtain a closure theorem for some reasonable language (like we did for unknown nulls). In particular, we should focus on the most important algebraic operations for accessing data: projection and join. It is also possible to establish a lattice structure with the different kinds of nulls so that they can be used meaningfully in combination.

Disjunctive Deductive Databases

Disjunctive logic programming is an extension of standard logic programming with rules of the form

$$A_1 \vee \cdots \vee A_i \leftarrow B_1, \ldots, B_j, \neg C_1, \ldots, \neg C_k.$$

In datalog, the answer to a query is a set of valuations. For instance, the answer to a query $\leftarrow Q(x)$ is a set of constants a such that $Q(a)$ holds. In disjunctive deductive databases, an answer may also be a disjunction $Q(a) \vee Q(b)$.

Disjunctions give rise to new problems of semantics for logic programs. Although in datalog each program has a unique minimal model, this is no longer the case for datalog with disjunctions. For instance, consider the database consisting of a single statement $\{Q(a) \vee Q(b)\}$. Then there are clearly two minimal models: $\{Q(a)\}$ and $\{Q(b)\}$. This leads to semantics in terms of *sets of minimal models*, which can be viewed as incomplete databases. We can develop a fixpoint theory for disjunctive databases, extending naturally the fixpoint approach for datalog. To do this, we use an ordering over *sets of minimal interpretations* (i.e., sets $\mathcal{I}$ of instances such that there are no I, J in $\mathcal{I}$ with $I \subset J$).

DEFINITION 19.5.1 Let $\mathcal{I}$, $\mathcal{J}$ be sets of minimal interpretations. Then

$$\mathcal{J} \sqsubseteq \mathcal{I} \text{ iff } \forall I \in \mathcal{I}\ (\exists J \in \mathcal{J}\ (J \subseteq I)).$$

Consider the following immediate consequence operator. Let P be a datalog program with disjunctions, and let $\mathcal{I}$ be a set of minimal interpretations. A new set $\mathcal{J}$ of interpretations is obtained as follows. For each I in $\mathcal{I}$, $state_P(I)$ is the set of disjunctions of the form $A_1 \vee \cdots \vee A_i$ that are immediate consequences of some facts in I using P. Then $\mathcal{J}$ is the set of of instances J such that for some $I \in \mathcal{I}$, J is a model of $state_P(I)$ containing I. Clearly, $\mathcal{J}$ is not a set of minimal interpretations. The immediate consequence of $\mathcal{I}$, denoted $T_P(\mathcal{I})$, is the set of minimal interpretations in $\mathcal{J}$. Now consider the sequence

$$\mathcal{I}_0 = \emptyset$$
$$\mathcal{I}_i = T_P(\mathcal{I}_{i-1}).$$

It is easy to see that the sequence $\{\mathcal{I}_i\}_{i \geq 0}$ is nondecreasing with respect to the ordering $\sqsubseteq$, so it becomes constant at some point. The semantics of P is the limit of the sequence.

When negation is introduced, the situation, as usual, becomes more complicated. However, it is possible to extend semantics, such as stratified and well founded, to disjunctive deductive databases.

Overall, the major difficulty in handling disjunction is the combinatorial explosion it entails. For example, the fixpoint semantics of datalog with disjunctions may yield a set of interpretations exponential in the input.

Logical Databases and KL

The approach to null values adopted here is essentially a *semantic* approach, because the meaning of an incomplete database is a set of possible instances. One can also use a *syntactic*, proof-theoretic approach to modeling incomplete information. This is done by regarding the database as a set of sentences, which yields the *logical database* approach.

As discussed in Chapter 2, in addition to statements about the real world, logical databases consider the following:

1. *Uniqueness axioms*: State that distinct constants stand for distinct elements in the real world.
2. *Domain closure axiom*: Specify the universe of constants.
3. *Completion axiom*: Specify that no fact other than recorded holds.

Missing in both the semantic and syntactic approaches is the ability to make more refined statements about what the database knows. Such capabilities are particularly important in applications where the real world is slowly discovered through imprecise data. In such applications, it is general impossible to wait for a complete state to answer queries, and it is often desirable to provide the user with information about the current state of knowledge of the database.

To overcome such limitations, we may use languages with modalities. We briefly mention one such language: KL. The language KL permits us to distinguish explicitly between the real world and the knowledge the database has of it. It uses the particular modal symbol K. Intuitively, whereas the sentence φ states the truth of φ in the real world, $K\varphi$ states that the database knows that φ holds.

For instance, the fact that the database knows neither that Alice is a student nor that

she is not is expressed by the statement

$$\neg K\,Student(Alice) \wedge \neg K\,(\neg Student(Alice)).$$

The following KL statement says that there is a teacher who is unknown:

$$\exists x\,(Teacher(x) \wedge \neg K\,(Teacher(x))).$$

This language allows the database to reason and answer queries about its own knowledge of the world.

Incomplete Information in Complete Databases

Incomplete information often arises naturally even when the focus is on complete databases. The following are several situations that naturally yield incomplete information:

- *Views*: Although a view of a database is usually a complete database, the information it contains is incomplete relative to the whole database. For a user seeing the view, there are many possible underlying databases. So the view can be seen as a representation for the set of possible underlying databases. The incompleteness of the information in the view is the source of the difficulty in view updating (see Chapter 22).
- *Weak universal relations*: We have already seen how relations with nulls arise in the weak universal relations of Chapter 11.
- *Nondeterministic queries*: Recall from Chapter 17 that nondeterministic languages have several possible answers on a given input. Thus we can think of nondeterministic queries as producing as an answer a set of possible worlds (see also Exercise 19.20).
- *Semantics of negation*: As seen in Chapter 15, the well-founded semantics for datalog$^\neg$ involves 3-valued interpretations, where some facts are neither true nor false but unknown. Clearly, this is a form of incomplete information.

Bibliographic Notes

It was accepted early on that database systems should handle incomplete information [Cod75]. After some interesting initial work on the topic (e.g., [Cod75, Gra77, Cod79, Cod82, Vas79, Vas80, Bis81, Lip79, Lip81, Bis83]), the landmark paper [IL84] laid the formal groundwork for incomplete databases with nulls of the unknown kind and introduced the notion of representation system. That paper assumed the OWA, as opposed to the CWA that was assumed in this chapter. Since then, there has been considerable work on querying incomplete information databases. The focus of most of this work has been a search for the correct semantics for queries applied to incomplete information databases (e.g., [Gra84, Imi84, Zan84, AG85, Rei86, Var86b]).

Much of the material presented in this chapter is from [IL84] (although it was presented there assuming the OWA), and we refer the reader to it for a detailed treatment.

Tables form the central topic of the monograph [Gra91]. Examples in Section 19.1 are taken from there. The naive tables have been called "V-tables" and "e-tables" in [AG85, Gra84, IL84]. The c-tables with local conditions are from [IL84]; they were augmented with global conditions in [Gra84]. The fact that c-tables provide a strong representation system for relational algebra is shown in [IL84]. That this strong representation property extends to query languages with fixpoint on positive queries is reported in [Gra91]. Chasing is applied to c-tables in [Gra91].

There are two main observations in the literature on certainty semantics. The first observation follows from the results of [IL84] (based on c-tables) and [Rei86, Var86b] (based on logical databases). Namely, under particular syntactic restrictions on c-tables and using positive queries, the certainty question can be handled exactly as if one had a complete information database. The second observation deals with the negative effects of the many possible instantiations of the null values (e.g., [Var86b]).

Comprehensive data-complexity analysis of problems related to representing and querying databases with null values is provided in [IL84, Var86b, AKG91]. The program complexity of evaluation is higher by an exponential than the data complexity [Cos83, Var82a]. Such problems were first noted in [HLY80, MSY81] as part of the study of nulls in weak universal instances.

Early investigations suggesting the use of orderings in the spirit of denotational semantics for capturing incomplete information include [Vas79, Bis81]. The first paper to develop this approach is [BJO91], which focused on fd's and universal relations. This has spawned several papers, including an extension to complex objects (see Chapter 20) [BDW88, LL90], mvd's [Lib91], and bags [LW93b]. An important issue in this work concerns which power domain ordering is used (Hoare, Smyth, or Plotkin); see [BDW91, Gun92, LW93a].

The logical database approach has been largely influenced by the work of Reiter [Rei78, Rei84, Rei86] and by that of Vardi [Var86a, Var86b]. The extension of the fixpoint operator of logic programs to disjunctive logic programs is shown in [MR90]. Disjunctive logic programming is the topic of [LMR92]. A survey on deductive databases with disjunctions can be found in [FM92]. The complexity of datalog with disjunction is investigated in [EGM94].

A related but simpler approach to incomplete information is the use of "or-sets." As a simple example, a tuple $\langle Joe, \{20, 21\}\rangle$ might be used to indicate that Joe has age either 20 or 21. This approach is introduced in [INV91a, INV91b] in the context of complex objects; subsequent works include [Rou91, LW93a].

One will find in [Lev84b, Lev84a] entry points to the interesting world of knowledge bases (from the viewpoint of incompleteness of information), including the language KL. A related, active area of research, called reasoning about knowledge, extends modal operators to talk about the knowledge of several agents about facts in the world or about each other's knowledge. This may be useful in distributed databases, where sites may have different knowledge of the world. The semantics of such statements is in terms of an extension of the possible worlds semantics, based on Kripke structures. An introduction to reasoning about knowledge can be found in [Hal93, FHMV95].

Finally, nonapplicable nulls are studied in [LL86]; open nulls are studied in [GZ88]; and weak instances with nonapplicable nulls are studied in [AB87b].

Exercises

Exercise 19.1 Consider the c-table in Example 19.3.1. Give the c-tables for the answers to these queries: (1) Which students are taking Math? (2) Which students are not taking Math? (3) Which students are taking Biology? In each case, what are the sets of sure and possible tuples of the answer?

Exercise 19.2 Consider the c-table T' in Fig. 19.3. Show that each I in $rep(T')$ has two tuples. Is T' equivalent to some 2-tuple c-table?

Exercise 19.3 Consider the naive table in Fig. 19.2. In the weak representation system described in Section 19.1, compute the naive tables for the answers to the queries $\sigma_{A=C}(R)$, $\pi_{AB}(R) \bowtie \pi_{AC}(R)$. What are the tuples surely in the answers to these queries?

Exercise 19.4 A ternary c-table T represents a directed graph with blue, red, and yellow edges. The first two columns represent the edges and the last the colors. Some colors are unknown. The local conditions are used to enforce that a blue edge cannot follow a red one on a path. Give a datalog query q stating that there is a cycle with no two consecutive edges of the same color. Give c-tables such that (1) there is surely such a cycle; and (2) there may be one but it is not sure. In each case, compute the table strongly representing the answer to q.

Exercise 19.5 Let T be the Codd table in Fig. 19.1. Compute strong representations of the results of the following queries, using c-tables: (a) $\sigma_{A=3}(R)$; (b) $q_1 = \delta_{BC \to AB}(\pi_{BC}(R))$; (c) $q_1 \cup \pi_{AB}(R)$; (d) $q_1 \cap \pi_{AB}(R)$; (e) $q_1 - \pi_{AB}(R)$; (f) $q_1 \bowtie \pi_{BC}(R)$.

Exercise 19.6 Consider the c-table $T_4 = T_1 \cup T_2$ of Fig. 19.4. Compute a strong representation of the transitive closure of T_4.

Exercise 19.7 Complete the proof that Codd tables are not a weak representation system with respect to SPU, in Theorem 19.2.1.

Exercise 19.8 Example 19.1.1 shows that one cannot strongly represent the result of a selection on a table with another table. For which operations of relational algebra applied to tables is it possible to strongly represent the result?

Exercise 19.9 Prove that naive tables are not a weak representation system for relational algebra.

Exercise 19.10 Prove that, given a c-table T without constants, $rep(T)$ is the closure under isomorphism of a finite set of instances. Extend the result for the case with constants.

Exercise 19.11 Provide an algorithm for testing equivalence of c-tables.

★ **Exercise 19.12** Show that there exists a datalog query q such that, given a naive table T and a tuple t, testing whether t is possibly in the answer is NP-complete.

Exercise 19.13 Prove Theorem 19.3.2.

Exercise 19.14 Prove that for each c-table T_1 and each set of fd's and mvd's, there exists a table T_2 such that $chase_\Sigma(rep(T_1)) = rep(T_2)$. *Hint:* Use the chase on c-tables.

★ **Exercise 19.15** Show that there is a query q in polynomial time for which deciding, given I and a c-table T, (a) whether $I \in q(rep(T))$, or whether I is possible, are NP-complete; and (b) whether $q(rep(T)) \subseteq \{I\}$, or whether I is certain, are co-NP-complete.

Exercise 19.16 Give algorithms to compute, for a c-table T and a relational algebra query q, the set of tuples $sure(q, T)$ surely in the answer and the set of tuples $poss(q, T)$ possibly in the answer. What is the complexity of your algorithms?

Exercise 19.17 Let T be a c-table and q a positive existential query of the same arity as T. Show that the sequence $\overline{q}^i(T)$ converges [i.e., that for some i, $\overline{q}^i(T) \equiv \overline{q}^{i+1}(T)$]. *Hint:* Show that the sequence converges in at most m stages, where $m = max\{i \mid q^i(I) = q^{i+1}(I), I \in \mathcal{I}\}$ and where $\mathcal{I}$ is a finite set of relations representing the nonisomorphic instances in $rep(T)$.

Exercise 19.18 Describe how to generalize the technique of chasing by full dependencies to apply to instances rather than tableau. If an egd can be applied and calls for two distinct constants to be identified, then the chase ends in failure. Show that for instance I, if the chase of I by Σ succeeds, then $chase(I, \Sigma) \models \Sigma$.

Exercise 19.19 Show that for datalog programs with disjunctions in heads of rules, the sequence $\{\mathcal{I}_i\}_{i \geq 0}$ of Section 19.5 converges. What can be said about the limit in model-theoretic terms?

♠ **Exercise 19.20** [ASV90] There is an interesting connection between incomplete information and *nondeterminism*. Recall the nondeterministic query languages based on the *witness* operator W, in Chapter 17. One can think of nondeterministic queries as producing as an answer a set of possible worlds. In the spirit of the sure and possible answers to queries on incomplete databases, one can define for a nondeterministic query q the deterministic queries $sure(q)$ and $poss(q)$ as follows:

$$sure(q)(I) = \cap\{J \mid J \in q(I)\}$$
$$poss(q)(I) = \cup\{J \mid J \in q(I)\}$$

Consider the language $FO + W$, where a program consists of a finite sequence of assignment statements of the form $R := \varphi$, where φ is a relational algebra expression or an application of W to a relation. Let $sure(FO + W)$ denote all deterministic queries that can be written as $sure(q)$ for some $FO + W$ query q, and similarly for $poss(FO + W)$. Prove that

(a) $poss(FO + W) =$NP, and

(b) $sure(FO + W) =$co-NP.

20 Complex Values

Alice: *Complex values?*

Riccardo: *We could have used a different title: nested relations, complex objects, structured objects . . .*

Vittorio: *. . . N1NF, NFNF, NF^2, NF2, V-relation . . . I have seen all these names and others as well.*

Sergio: *In a nutshell, relations are nested within relations; something like Matriochka relations.*

Alice: *Oh, yes. I love Matriochkas.*

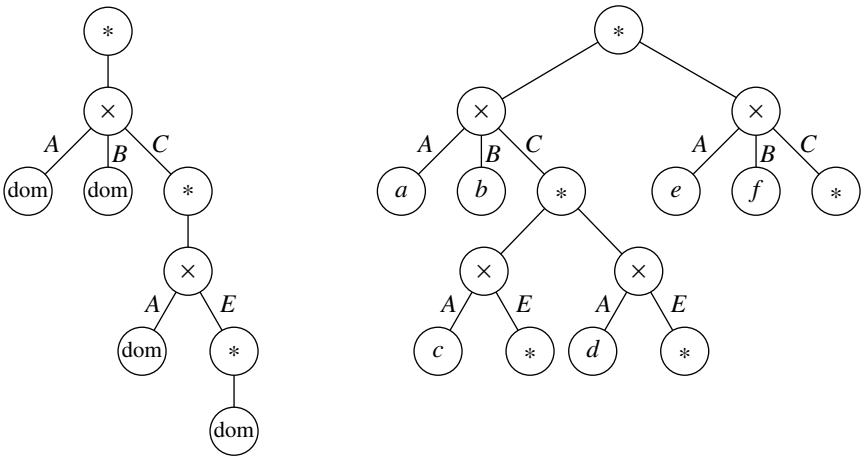
Although we praised the simplicity of the data structure in the relational model, this simplicity becomes a severe limitation when designing many practical database applications. To overcome this problem, the complex value model has been proposed as a significant extension of the relational one. This extension is the topic of this chapter.

Intuitively, complex values are relations in which the entries are not required to be atomic (as in the relational model) but are allowed to be themselves relations. The data structure in the relational model (the relation) can be viewed as the result of applying to atomic values two constructors: a *tuple constructor* to make tuples and a *set constructor* to make sets of tuples (relations). Complex values allow the application of the tuple and set constructor recursively. Thus they can be viewed as finite trees whose internal nodes indicate the use of the tuple and finite set constructors. Clearly, a relation is a special kind of complex value: a set of tuples of atomic values.

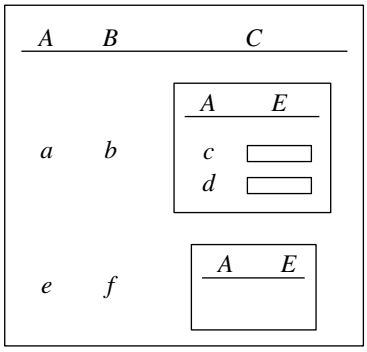
At the schema level, we will specify a set of complex *sorts* (or types). These indicate the structure of the data. At the instance level, sets of complex values corresponding to these sorts are provided. For example, we have the following:

Sort	*Complex Value*
dom	a
{**dom**}	$\{a, b, c\}$
$\langle A : \mathbf{dom}, B : \mathbf{dom}\rangle$	$\langle A : a, B : b\rangle$
$\{\langle A : \mathbf{dom}, B : \mathbf{dom}\rangle\}$	$\{\langle A : a, B : b\rangle, \langle A : b, B : a\rangle\}$
{{**dom**}}	$\{\{a, b\}, \{a\}, \{\,\}\}$

An example of a more involved complex value sort and of a value of that sort is shown in Fig. 20.1(a). The tuple constructor is denoted by $\times$ and the set constructor by $*$. An



(a) A sort and a value of that sort



(b) Another representation of the same value

Figure 20.1: Complex value

alternative representation more in the spirit of our representations of relations is shown in Fig. 20.1(b). Another complex value (for a **CINEMA** database) is shown in Fig. 20.2.

We will see that, whereas it is simple to add the tuple constructor to the traditional relational data model, the set constructor requires a number of interesting new ideas. There are similarities between this set construct and the set constructs used in general-purpose programming languages such as Setl.

In this chapter, we introduce complex values and present a many-sorted algebra and an equivalent calculus for complex values. The focus is on the use of the two constructors of complex values: tuples and (finite) sets. (Additional constructors, such as list, bags, and

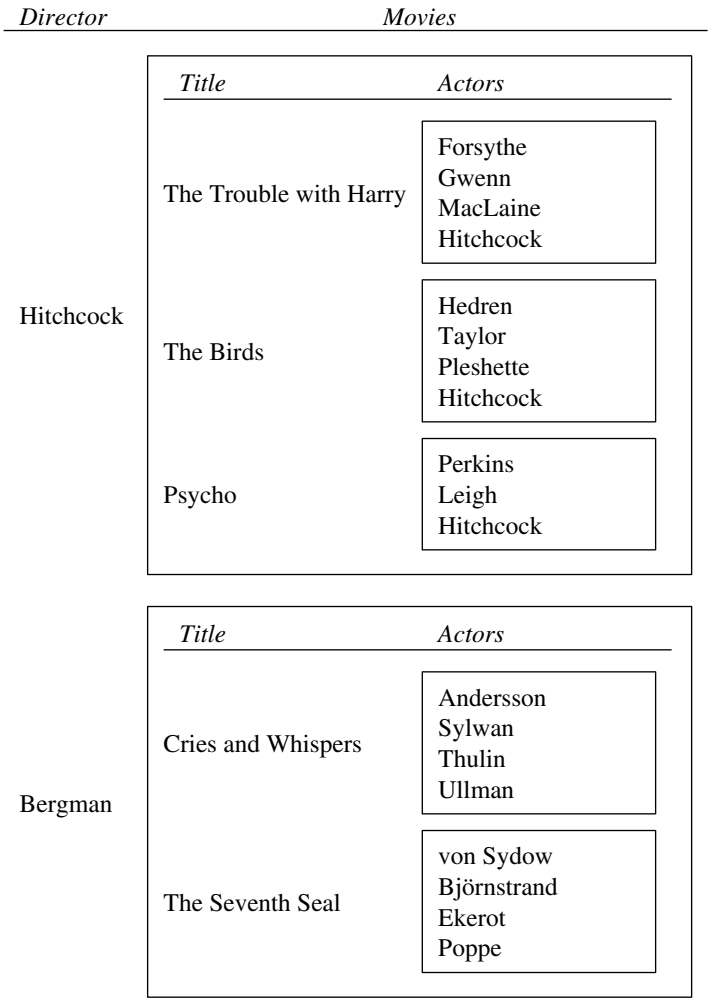


Figure 20.2: The **CINEMA** database revisited (with additional data shown)

union, have also been incorporated into complex values but are not studied here.) After introducing the algebra and calculus, we present examples of these interesting languages. We then comment on the issues of expressive power and complexity and describe equivalent languages with fixpoint operators, as well as languages in the deductive paradigm. Finally we briefly examine a subset of the commercial query language O_2SQL that provides an elegant SQL-style syntax for querying complex values.

The theory described in this chapter serves as a starting point for object-oriented databases, which are considered in Chapter 21. However, key features of the object-oriented paradigm, such as objects and inheritance, are still missing in the complex value framework and are left for Chapter 21.

20.1 Complex Value Databases

Like the relational model, we will use relation names in **relname**, attributes in **att**, and constants in **dom**. The sorts are more complex than for the relational model. Their abstract syntax is given by

$$\tau \;=\; \mathbf{dom} \;\mid\; \langle B_1 : \tau, \ldots, B_k : \tau \rangle \;\mid\; \{\tau\},$$

where $k \geq 0$ and $B_1, \ldots, B_k$ are distinct attributes. Intuitively, an element of **dom** is a constant; an element of $\langle B_1 : \tau_1, \ldots, B_k : \tau_k \rangle$ is a k-tuple with an element of sort τ_i in entry B_i for each i; and an element of sort $\{\tau\}$ is a finite set of elements of sort τ.

Formally, the set of values of sort τ (i.e., the interpretation of τ), denoted $[\![\tau]\!]$, is defined by

1. $[\![\mathbf{dom}]\!] = \mathbf{dom}$,
2. $[\![\{\tau\}]\!] = \{\{v_1, \ldots, v_j\} \mid j \geq 0, v_i \in [\![\tau]\!], i \in [1, j]\}$, and
3. $[\![\langle B_1 : \tau_1, \ldots, B_k : \tau_k \rangle]\!] = \{\langle B_1 : v_1, \ldots, B_k : v_k \rangle \mid v_j \in [\![\tau_j]\!], j \in [1, k]\}$.

An element of a sort is called a *complex value*. A complex value of the form $\langle B_1 : a_1, \ldots, B_k : a_k \rangle$ is said to be a *tuple*, whereas a complex value of the form $\{a_1, \ldots, a_j\}$ is a set.

Remark 20.1.1 For instance, consider the sort

$$\{\langle A : \mathbf{dom}, B : \mathbf{dom}, C : \{\langle A : \mathbf{dom}, E : \{\mathbf{dom}\}\rangle\}\rangle\}$$

and the value

$$\begin{array}{l} \{\ \langle A : a, B : b, C : \{\ \langle A : c, E : \{\}\rangle, \\ \qquad\qquad\qquad\qquad\ \ \langle A : d, E : \{\}\rangle\}\rangle, \\ \quad \langle A : e, B : f, C : \{\ \}\rangle\ \} \end{array}$$

of that sort. This is yet again the value of Fig. 20.1. It is customary to omit **dom** and for instance write this sort $\{\langle A, B, C : \{\langle A, E : \{\}\rangle\}\rangle\}$.

As mentioned earlier, each complex value and each sort can be viewed as a finite tree. Observe the tree representation. Outgoing edges from tuple vertexes are labeled; set vertexes have a single child in a sort and an arbitrary (but finite) number of children in a value.

Finally note that (because of the empty set) a complex value may belong to more than one sort. For instance, the value of Fig. 20.1 is also of sort

$$\{\langle A : \mathbf{dom}, B : \mathbf{dom}, C : \{\langle A : \mathbf{dom}, E : \{\{\mathbf{dom}\}\}\rangle\}\rangle\}.$$

Relational algebra deals with *sets* of tuples. Similarly, complex value algebra deals with sets of complex values. This motivates the following definition of sorted relation (this

definition is frequently a source of confusion):

A (complex value) *relation* of sort τ is a finite set of values of sort τ.

We use the term *relation* for complex value relation. When we consider the classical relational model, we sometimes use the phrase *flat relation* to distinguish it from complex value relation. It should be clear that the flat relations that we have studied are special cases of complex value relations.

We must be careful in distinguishing the sort of a complex value relation and the sort of the relation viewed as one complex value. For example, a complex value relation of sort $\langle A, B, C\rangle$ is a set of tuples over attributes ABC. At the same time, the entire relation can be viewed as one complex value of sort $\{\langle A, B, C\rangle\}$. There is no contradiction between these two ways of viewing a relation.

We now assume that the function *sort* (of Chapter 3) is from **relname** to the set of sorts. We also assume that for each sort, there is an infinite number of relations having that sort.

Note that the sort of a relation is not necessarily a tuple sort (it can be a set sort). Thus relations do not always have attributes at the top level. Such relations whose sort is a set are essentially unary relations without attribute names.

A *(complex value) schema* is a relation name; and a *(complex value) database schema* is a finite set of relation names. A *(complex value) relation* over relation name R is a finite set of values of sort $sort(R)$—that is, a finite subset of $[\![sort(R)]\!]$. A *(complex value database) instance* $\mathbf{I}$ of a schema $\mathbf{R}$ is a function from $\mathbf{R}$ such that for each R in $\mathbf{R}$, $\mathbf{I}(R)$ is a relation over R.

Example 20.1.2 To illustrate this definition, an instance $\mathbf{J}$ of $\{R_1, R_2, R_3\}$ where

$$sort(R_1) = sort(R_3) = \langle A : \mathbf{dom}, B : \{\langle A_1 : \mathbf{dom}, A_2 : \mathbf{dom}\rangle\}\rangle \text{ and}$$
$$sort(R_2) = \langle A : \mathbf{dom}, A_1 : \mathbf{dom}, A_2 : \mathbf{dom}\rangle$$

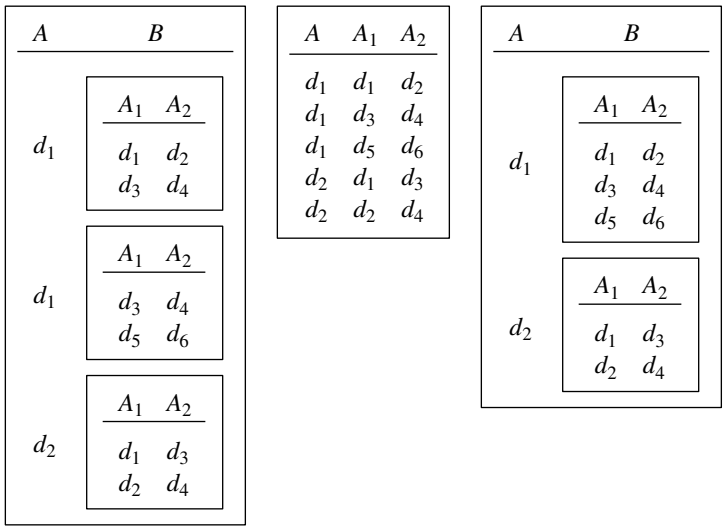
is shown in Fig. 20.3.

Variations

To conclude this section, we briefly mention some variations of the complex value model. The principal one that has been considered is the *nested relation model*. For nested relations, set and tuple constructors are required to alternate (i.e., set of sets and tuple with a tuple component are prohibited). For instance,

$$\tau_1 = \langle A, B, C : \{\langle D, E : \{\langle F, G\rangle\}\rangle\}\rangle \quad \text{and}$$
$$\tau_2 = \langle A, B, C : \{\langle E : \{\langle F, G\rangle\}\rangle\}\rangle$$

are nested relation sorts whereas



$\mathbf{J}(R_1)$ $\mathbf{J}(R_2)$ $\mathbf{J}(R_3)$

Figure 20.3: A database instance

$$\tau_3 = \langle A, B, C : \langle D, E : \{\langle F, G\rangle\}\rangle\rangle \quad \text{and}$$
$$\tau_4 = \langle A, B, C : \{\{\langle F, G\rangle\}\}\rangle$$

are not. (For τ_3, observe two adjacent tuple constructors; there are two set constructors for τ_4.)

The restriction imposed on the structure of nested relations is mostly cosmetic. A more fundamental constraint is imposed in so-called Verso-relations (V-relations).

As with nested relations, set and tuple constructors in V-relations are required to alternate. A relation is defined recursively to be a set of tuples, such that each component may itself be a relation but at least one of them must be atomic. The foregoing sort τ_1 would be acceptable for a V-relation whereas sort τ_2 would not because of the sort of tuples in the C component.

A further (more radical) assumption for V-relations is that for each set of tuples, the atomic attributes form a key. Observe that as a consequence, the cardinality of each set in a V-relation is bounded by a polynomial in the number of atomic elements occurring in the V-relation. This bound certainly does not apply for a relation of sort {**dom**} (a set of sets) or for a nested relation of sort

$$\langle A : \{\langle B : \mathbf{dom}\rangle\}\rangle,$$

which is also essentially a set of sets. The V-relations are therefore much more limited data structures. (See Exercise 20.1.) They can be viewed essentially as flat relational instances.

20.2 The Algebra

We now define a many-sorted algebra, denoted ALG^{cv} (for *c*omplex *v*alues). Like relational algebra, ALG^{cv} is a functional language based on a small set of operations. This section first presents a family of core operators of the algebra and then an extended family of operators that can be simulated by them. At the end of the section we introduce an important subset of ALG^{cv}, denoted ALG^{cv-}.

The Core of ALG^{cv}

Let $I, I_1, I_2, \ldots$ be relations of sort $\tau, \tau_1, \tau_2, \ldots$ respectively. It is important to keep in mind that a relation of sort τ is a *set* of values of sort τ.

Basic set operations: If $\tau_1 = \tau_2$, then $I_1 \cap I_2$, $I_1 \cup I_2$, $I_1 - I_2$, are relations of sort τ_1, and their values are defined in the obvious manner.

Tuple operations: If I is a relation of sort $\tau = \langle B_1 : \tau_1, \ldots, B_k : \tau_k \rangle$, then

- $\sigma_\gamma(I)$ is a relation of sort τ.
  The selection condition γ is (with obvious restrictions on sorts) of the form $B_i = d$, $B_i = B_j$, $B_i \in B_j$ or $B_i = B_j.C$, where d is a constant, and it is required in the last case that τ_j be a tuple sort with a C field. Then

$$\sigma_\gamma(I) = \{v \mid v \in I, v \models \gamma\},$$

  where $\models$ is defined by

  - $\langle \ldots, B_i : v_i, \ldots \rangle \models B_i = d$ if $v_i = d$,
  - $\langle \ldots, B_i : v_i, \ldots, B_j : v_j, \ldots \rangle \models B_i = B_j$ if $v_i = v_j$, and
  - $\langle \ldots, B_i : v_i, \ldots, B_j : v_j, \ldots \rangle \models B_i \in B_j$ if $v_i \in v_j$.
  - $\langle \ldots, B_i : v_i, \ldots, B_j : \langle \ldots, C : v_j, \ldots \rangle, \ldots \rangle \models B_i = B_j.C$ if $v_i = v_j$.

- $\pi_{B_1,\ldots,B_l}(I), l \leq k$ is a relation of sort $\langle B_1 : \tau_1, \ldots, B_l : \tau_l \rangle$ with

$$\pi_{B_1,\ldots,B_l}(I) = \{ \langle B_1 : v_1, \ldots, B_l : v_l \rangle \mid \exists v_{l+1}, \ldots, v_k (\langle B_1 : v_1, \ldots, B_k : v_k \rangle \in I)\}.$$

Constructive operations

- *powerset*(I) is a relation of sort $\{\tau\}$ and

$$powerset(I) = \{v \mid v \subseteq I\}.$$

- If $A_1, \ldots, A_n$ are distinct attributes, $tup\_create_{A_1 \ldots A_n}(I_1, \ldots, I_n)$ is of sort $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$, and

$$tup\_create_{A_1,\ldots,A_n}(I_1, \ldots, I_n) = \{\langle A_1 : v_1, \ldots, A_n : v_n \rangle \mid \forall i \ (v_i \in I_i)\}.$$

- *set_create*(I) is of sort $\{\tau\}$, and *set_create*$(I) = \{I\}$.

Destructive operations

- If $\tau = \{\tau'\}$, then *set_destroy*(I) is a relation of sort τ' and

$$set\_destroy(I) = \cup I = \{w \mid \exists v \in I, w \in v\}.$$

- If I is of sort $\langle A : \tau' \rangle$, *tup_destroy*$(I)$ is a relation of sort τ', and

$$tup\_destroy(I) = \{v \mid \langle A : v \rangle \in I\}.$$

We are now prepared to define the (core of the) language ALG^{cv}. Let $\mathbf{R}$ be a database schema. A query returns a set of values of the same sort. By analogy with relations, a query of sort τ returns a *set* of values of sort τ. ALG^{cv} queries and their answers are defined as follows. There are two base cases:

Base values: For each relation name R in $\mathbf{R}$, R is an algebraic query of sort *sort*(R). The answer to query R is $\mathbf{I}(R)$.

Constant values: For each element a, $\{a\}$ is a (constant) algebraic query of sort **dom**. The answer to query $\{a\}$ is simply $\{a\}$.

Other queries of ALG^{cv} are obtained as follows. If $q_1, q_2, \ldots$ are queries, γ is a selection condition, and $A_1, \ldots$ are attributes,

$q_1 \cap q_2$,	$q_1 \cup q_2$,	$q_1 - q_2$,
$\sigma_\gamma(q_1)$,	$\pi_{A_1,\ldots,A_k}(q_1)$,	$tup\_create_{A_1,\ldots,A_k}(q_1, \ldots, q_k)$,
$powerset(q_1)$,	$tup\_destroy(q_1)$,	$set\_destroy(q_1)$,
$set\_create(q_1)$		

are queries if the appropriate restrictions on the sorts apply. (Note that because of the sorting constraints, *tup_destroy* and *set_destroy* cannot both be applicable to a given q_1.) The sort of a query and its answer are defined in a straightforward manner.

To illustrate these definitions, we present two examples. We then consider other algebraic operators that are expressible in the algebra. In Section 20.4 we provide several more examples of algebraic queries.

Example 20.2.1 Consider the instance $\mathbf{J}$ of Fig. 20.3. Then one can find in Fig. 20.4

$$\begin{array}{ll} J_1 = [\sigma_{A=d_2}(R_1)](\mathbf{J}), & J_2 = \pi_B(J_1), \\ J_3 = tup\_destroy(J_2), & J_4 = set\_destroy(J_3), \\ J_5 = powerset(J_4), & J_6 = tup\_create_C(J_4). \end{array}$$

Also observe that

$$J_5 = [powerset(set\_destroy(tup\_destroy(\pi_B(\sigma_{A=d_2}(R_1)))))](\mathbf{J}).$$

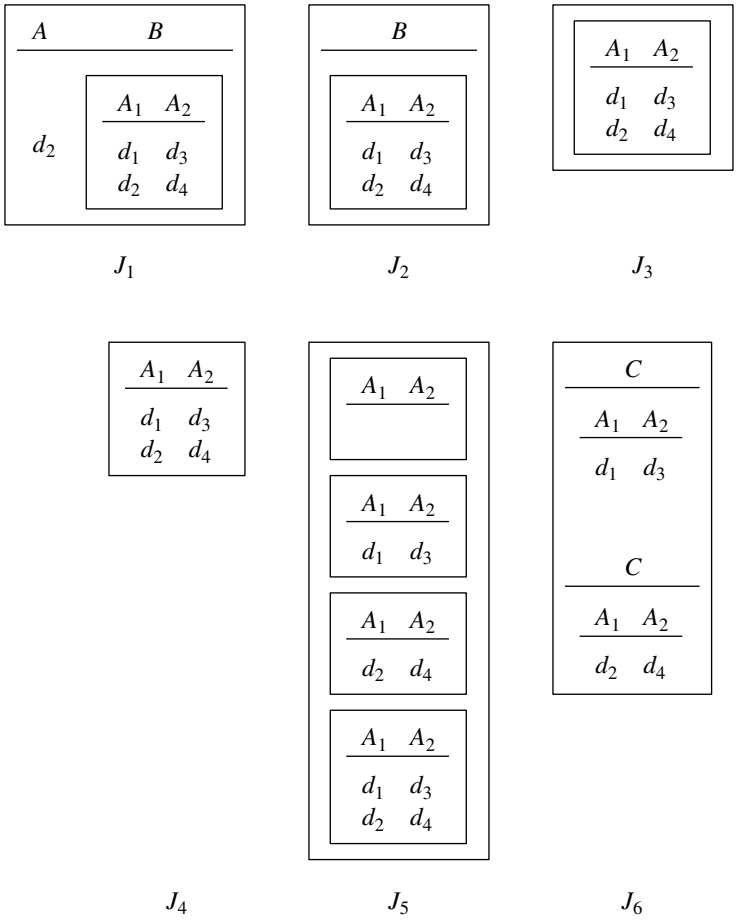


Figure 20.4: Algebraic operations

EXAMPLE 20.2.2 In this example, we illustrate the destruction and construction of a complex value. Consider the relation

$$I = \{\langle A : a, B : \{b, c\}, C : \langle A : d, B : \{e, f\}\rangle\rangle\}.$$

Then

$$\begin{aligned}
&[(\pi_A \circ \mathit{tup_destroy}) \\
&\quad \cup\ (\pi_B \circ \mathit{tup_destroy} \circ \mathit{set_destroy}) \\
&\quad \cup\ (\pi_C \circ \mathit{tup_destroy} \circ \pi_A \circ \mathit{tup_destroy}) \\
&\quad \cup\ (\pi_C \circ \mathit{tup_destroy} \circ \pi_B \circ \mathit{tup_destroy} \circ \mathit{set_destroy})](I) \\
&\quad = \{a, b, c, d, e, f\}.
\end{aligned}$$

We next reconstruct I from singleton sets:

$$I = tup\_create_{A,B,C}(\{a\}, set\_create(\{b\} \cup \{c\}),$$
$$tup\_create_{A,B}(\{d\}, set\_create(\{e\} \cup \{f\}))).$$

Additional Algebraic Operations

There are infinite possibilities in the choice of algebraic operations for complex values. We chose to incorporate in the core algebra only a few basic operations to simplify the formal presentation and the proof of the equivalence between the algebra and calculus. However, making the core *too* reduced would complicate that proof. (For example, the operator *set_create* can be expressed using the other operations but is convenient in the proof.) We now present several additional algebraic operations. It is important to note that all these operations can be expressed in complex value algebra. (In that sense, they can be viewed as macro operations.) Furthermore, all but the *nest* operator can be expressed without using the powerset operator.

We first generalize constant queries.

Complex constants: It is easy to see that the technique of Example 20.2.2 can be generalized. So instead of simply $\{a\}$ for a atomic, we use as constant queries arbitrary complex value sets.

We also generalize relational operations.

Renaming: Renaming can be computed using the other operations, as illustrated in Section 20.4 (which presents examples of queries).

Cross-product: For i in [1,2], let I_i be a relation of sort

$$\tau_i = \langle B_1^i : \tau_1^i, \ldots, B_{j_i}^i : \tau_{j_i}^i \rangle$$

and let the attribute sets in τ_1, τ_2 be disjoint. Then $I_1 \times I_2$ is the relation defined by

$$sort(I_1 \times I_2) = \langle B_1^1 : \tau_1^1, \ldots, B_{j_1}^1 : \tau_{j_1}^1, B_1^2 : \tau_1^2, \ldots, B_{j_2}^2 : \tau_{j_2}^2 \rangle$$

and

$$I_1 \times I_2 = \{ \langle B_1^1 : x_1^1, \ldots, B_{j_1}^1 : x_{j_1}^1, B_1^2 : x_1^2, \ldots, B_{j_2}^2 : x_{j_2}^2 \rangle \mid$$
$$\langle B_1^i : x_1^i, \ldots, B_{j_i}^i : x_{j_i}^i \rangle \in I_i \text{ for } i \in [1, 2] \}.$$

It is easy to simulate cross-product using the operations of the algebra. This is also illustrated in Section 20.4.

Join: This can be defined in the natural manner and can be simulated using cross-product, renaming, and selection.

It should now be clear that complex value algebra subsumes relational algebra when applied to flat relations. We also have new set-oriented operations.

N-ary set_create: We introduced *tup_create* as an n-ary operation. We also allow n-ary *set_create* with the meaning that

$$set\_create(I_1, \ldots, I_n) \equiv set\_create(I_1) \cup \cdots \cup set\_create(I_n).$$

Singleton: This operator transforms a set of values $\{a_1, \ldots, a_n\}$ into a set $\{\{a_1\}, \ldots, \{a_n\}\}$ of singletons.

Nest, unnest: Less primitive interesting operations such as *nest, unnest* can be considered. For example, for $\mathbf{J}$ of Fig. 20.3 we have

$$unnest_B(\mathbf{J}(R_1)) = \mathbf{J}(R_2) \qquad \text{and}$$
$$nest_{B=(A_1A_2)}(\mathbf{J}(R_2)) = \mathbf{J}(R_3).$$

More formally, suppose that we have R and S with sorts

$$sort(R) = \langle A_1 : \tau_1, \ldots, A_k : \tau_k, B : \{\langle A_{k+1} : \tau_{k+1}, \ldots, A_n : \tau_n\rangle\}\rangle$$
$$sort(S) = \langle A_1 : \tau_1, \ldots, A_k : \tau_k, A_{k+1} : \tau_{k+1}, \ldots, A_n : \tau_n\rangle.$$

Then for instances I of R and J of S, we have

$$unnest_B(I) = \{\langle A_1 : x_1, \ldots, A_n : x_n\rangle \mid \exists y$$
$$\langle A_1 : x_1, \ldots, A_k : x_k, B : y\rangle \in I \text{ and } \langle A_{k+1} : x_{k+1}, \ldots, A_n : x_n\rangle \in y\}$$
$$nest_{B=(A_{k+1},\ldots,A_n)}(J) = \{\langle A_1 : x_1, \ldots, A_k : x_k, B : y\rangle \mid$$
$$\emptyset \neq y = \{\langle A_{k+1} : x_{k+1}, \ldots, A_n : x_n\rangle \mid \langle A_1 : x_1, \ldots, A_n : x_n\rangle \in J\}\}.$$

Observe that

$$unnest_B(nest_{B=(A_1A_2)}(\mathbf{J}(R_2))) = \mathbf{J}(R_2).$$
$$nest_{B=(A_1A_2)}(unnest_B(\mathbf{J}(R_1))) \neq \mathbf{J}(R_1).$$

This is indeed not an isolated phenomenon. *Unnest* is in general the right inverse of *nest* ($nest_{B=\alpha} \circ unnest_B$ is the identity), whereas *unnest* is in general not information preserving (one-to-one) and so has no right inverse (see Exercise 20.8).

Relational projection and selection were filtering operations in the sense that intuitively they scan a set and keep only certain elements, possibly modifying them in a uniform way. The filters in complex value algebra are more general. Of course, we shall allow Boolean expressions in selection conditions. More interestingly, we also allow set comparators in addition to $\in$, such as $\ni$, $\subset$, $\subseteq$, $\supset$, $\supseteq$ and negations of these comparators (e.g., $\notin$). The inclusion comparator $\subseteq$ plays a special role in the calculus. We will see in Section 20.4 how to simulate selection with $\subseteq$.

Selection is a *predicative filter* in the sense that a predicate allows us to select some elements, leaving them unchanged. Other filters, such as projection, are *map filters*. They transform the elements. Clearly, one can combine both aspects and furthermore allow more complicated selection conditions or restructuring specifications. For instance, suppose I is

a set of tuples of sort

$$\langle A : \mathbf{dom}, B : \langle C : \langle E : \{\mathbf{dom}\}, E' : \mathbf{dom}\rangle, C' : \{\mathbf{dom}\}\rangle\rangle.$$

We could use an operation that first filters all the values matching the pattern

$$\langle A : x, B : \langle C : \langle E : y, E' : z\rangle, C' : \{x\}\rangle\rangle;$$

and then transforms them into

$$\langle A : (y \cup \{x\}), B : y, C : z\rangle.$$

This style of operations is standard in functional languages (e.g., *apply-to-all* in fp).

Remark 20.2.3 As mentioned earlier, all of the operations just introduced are expressible in ALG^{cv}. We might also consider an operation to *iterate* over the elements of a set in some order. Such an operation can be found in several systems. As we shall see in Section 20.6, iteration is essentially expressible within ALG^{cv}. On the other hand, an iteration that depends on a specific ordering of the underlying domain of elements cannot be simulated using ALG^{cv} unless the ordering is presented as part of the input. ■

In the following sections, we (informally) call *extended algebra* the algebra consisting of the operations of ALG^{cv} and allowing complex constants, renaming, cross-product, join, n-ary *set_create*, singleton, *nest*, and *unnest*.

An important subset of ALG^{cv}, denoted ALG^{cv-}, is formed from the core operators of ALG^{cv} by removing the *powerset* operator and adding the *nest* operator. As will be seen in Section 20.7, although the *nest* operator has the ability to construct sets, it is much weaker than *powerset*. When restricted to nested relations, the language ALG^{cv-} is usually called *nested relation algebra*.

20.3 The Calculus

The calculus is modeled after a standard, first-order, many-sorted calculus. However, as we shall see, calculus variables may denote sets, so the calculus will permit quantification over sets (something normally considered to be a second-order feature). For complex value calculus, the separation between first and second order (and higher order as well) is somewhat blurred. As with the algebra, we first present a core calculus and then extend it. The issues of domain independence and safety are also addressed.

For each sort, we assume the existence of a countably infinite set of variables of that sort. A variable is *atomic* if it ranges over the sort **dom**. Let **R** be a schema. A *term* is an atomic element, a variable, or an expression $x.A$, where x is a tuple variable and A is an attribute of x. We do not consider (yet) fancier terms. A *positive literal* is an expression of the form

$$R(t), \quad t = t', \quad t \in t', \quad \text{or} \quad t \subseteq t',$$

where $R \in \mathbf{R}$, t, t' are terms and the appropriate sort restrictions apply.[1] *Formulas* are defined from atomic formulas using the standard connectives and quantifiers: $\wedge, \vee, \neg, \forall, \exists$. A *query* is an expression $\{x \mid \varphi\}$, where formula φ has exactly one free variable (i.e. x). We sometimes denote it by $\varphi(x)$. The calculus is denoted CALC^{cv}.

The following example illustrates this calculus.

Example 20.3.1 Consider the schema and the instance of Fig. 20.3. We can verify that $\mathbf{J}(R_2)$ is the answer on instance $\mathbf{J}$ to the query

$$\begin{aligned}\{x \mid \exists y, z, z', u, v, w \;\; & (R_1(y) \wedge y.A = u \wedge y.B = z \\ & \wedge z' \in z \wedge z'.A_1 = v \wedge z'.A_2 = w \\ & \wedge x.A = u \wedge x.A_1 = v \wedge x.A_2 = w) \;\},\end{aligned}$$

where the sorts of the variables are as follows:

$$\begin{aligned} &sort(x) = \langle A, A_1, A_2\rangle, && sort(y) = \langle A, B : \{\langle A_1, A_2\rangle\}\rangle, \\ &sort(u) = sort(v) = sort(w) = \mathbf{dom}, && sort(z') = \langle A_1, A_2\rangle, \\ &sort(z) = \{\langle A_1, A_2\rangle\}. \end{aligned}$$

We could also have used an unsorted alphabet of variables and sorted them inside the formula, as in

$$\begin{aligned}\{x : \langle A, A_1, A_2\rangle \;\; \mid \;\; & \exists y : \langle A, B : \{\langle A_1, A_2\rangle\}\rangle, \\ & z : \{\langle A_1, A_2\rangle\}, z' : \langle A_1, A_2\rangle, \\ & u : \mathbf{dom}, v : \mathbf{dom}, w : \mathbf{dom} \\ & (R_1(y) \wedge y.A = u \wedge y.B = z \\ & \wedge z' \in z \wedge z'.A_1 = v \wedge z'.A_2 = w \\ & \wedge x.A = u \wedge x.A_1 = v \wedge x.A_2 = w) \;\}.\end{aligned}$$

The key difference with relational calculus is the presence of the predicates $\in$ and $\subseteq$, which are interpreted as the standard set membership and inclusion. Another difference (of a more cosmetic nature) is that we allow only one free variable in relation atoms and in query formulas. This comes from the stronger sorts: A variable may represent an n-tuple.

The *answer* to a query q on an instance $\mathbf{I}$, denoted $q(\mathbf{I})$, is defined as for the relational model. As in the relational case, we may define various interpretations, depending on the underlying domain of base values used. As with relational calculus, the basis for defining the semantics is the notion

$$\mathbf{I} \text{ satisfies } \varphi \text{ for } \nu \text{ relative to } \mathbf{d}.$$

---

[1] Strictly speaking, the symbols $=, \subseteq$ and $\in$ are also many sorted.

[Recall that ν is a valuation of the free variables of φ and $\mathbf{d}$ is an arbitrary set of elements containing $adom(\varphi, \mathbf{I})$.]

Consider the definition of this notion in Section 5.3. Cases (a) through (g) remain valid for the complex object calculus. We have to consider two supplementary cases. Recall that for equality, we had case (b):

(b) $\mathbf{I} \models_{\mathbf{d}} \varphi[\nu]$ if $\varphi = (s = s')$ and $\nu(s) = \nu(s')$.

In the same spirit, we add

(h-1) $\mathbf{I} \models_{\mathbf{d}} \varphi[\nu]$ if $\varphi = (s \in s')$ and $\nu(s) \in \nu(s')$

(h-2) $\mathbf{I} \models_{\mathbf{d}} \varphi[\nu]$ if $\varphi = (s \subseteq s')$ and $\nu(s) \subseteq \nu(s')$.

This formally states that $\in$ is interpreted as set membership and $\subseteq$ as set inclusion (in the same sense that as $=$ is interpreted as equality).

The issues surrounding domain independence for relational calculus also arise with CALC^{cv}. We develop a syntactic condition ensuring domain independence, but we also occasionally use an active domain interpretation.

Extensions

As in the case of the algebra, we now consider extensions of the calculus that can be simulated by the core syntax just given.

The standard abbreviations used for relational calculus, such as the logical connectives $\rightarrow$, $\leftarrow$, $\leftrightarrow$, can be incorporated into CALC^{cv}. Using these connectives, it is easy to see the nonminimality of the calculus: Each literal $x \subseteq y$ can be replaced by $\forall z(z \in x \rightarrow z \in y)$, where z is a fresh variable.

Arity In the core calculus, only relation atoms of the form $R(t)$ are permitted. Suppose that the sort of R is $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$ for some n. Then $R(u_1, \ldots, u_n)$ is a shorthand for

$$\exists y(R(y) \wedge y.A_1 = u_1 \wedge \cdots \wedge y.A_n = u_n),$$

where y is a new variable. In particular, if R_0 is a relation of sort $\langle\ \rangle$ $(n = 0)$, observe that the only value of that sort is the empty tuple. Thus a variable y of that sort has only one possible value, namely $\langle\ \rangle$. Thus for such y, we can use the following expression:

$$R_0(\,) \quad \text{for} \quad \exists y(R_0(y)).$$

Constructed Terms Next we allow constructed terms in the calculus such as

$$\{x, b\}, \quad x.A.C, \quad \langle B_1 : a, B_2 : y \rangle.$$

More formally, if $t_1, \ldots, t_k$ are terms and $B_1, \ldots, B_k$ are distinct attributes, then $\langle B_1 : t_1, \ldots, B_k : t_k \rangle$ is a term. Furthermore, if the t_i are of the same sort, $\{t_1, \ldots, t_k\}$ is a term;

and if t_1 is a tuple term with attribute C, then $t_1.C$ is a term. The sorts of terms are defined in the obvious way. Note that a term may have several sorts because of the empty set. (We ignore this issue here.)

The use of constructed terms can be viewed as syntactic sugaring. For instance, suppose that the term $\{a, y\}$ occurs in a formula ψ. Then ψ is equivalent to

$$\exists x(\psi' \wedge \forall z(z \in x \leftrightarrow (z = a \vee z = y))),$$

where ψ' is obtained from ψ by replacing the term $\{a, y\}$ by x (a fresh variable).

Complex Terms We can also view relations as terms. For instance, if R is a relation of sort $\langle A, B\rangle$, then R can be used in the language as a term of sort $\{\langle A, B\rangle\}$. We may then consider literals such as $x \in R$, which is equivalent to $R(x)$; or more complex ones such as $S \in T$, which essentially means

$$\exists y(T(y) \wedge \forall x(x \in y \leftrightarrow S(x))).$$

The previous extension is based on the fact that a relation (in our context) can be viewed as a complex value. This is again due to the stronger sort system. Now the answer to a query q is also a complex value. This suggests considering the use of queries as terms of the language. We consider this now: A query $q \equiv \{y \mid \psi(y)\}$ is a legal term that can be used in the calculus like any other term. More generally, we allow terms of the form

$$\{y \mid \psi(y, y_1, \ldots, y_n)\},$$

where the free variables of ψ are $y, y_1, \ldots, y_n$. Intuitively, we obtain queries by providing bindings for $y_1, \ldots, y_n$. We will call such an expression a *parameterized query* and denote it $q(y_1, \ldots, y_n)$ (where $y_1, \ldots, y_n$ are the parameters).

For instance, suppose that a formula $liked(x, y)$ computes the films y that person x liked; and another one $saw(x, y)$ computes those that x has seen. The set of persons who liked all the films that they saw is given by

$$\{\, x \mid \{y \mid liked(x, y)\} \subseteq \{y \mid saw(x, y)\} \,\}.$$

The following form of literals will play a particular role when we study safety for this calculus:

$$\begin{aligned} x &= \{y \mid \psi(y, y_1, \ldots, y_n)\}, \\ x' &\in \{y \mid \psi(y, y_1, \ldots, y_n)\}, \text{ and} \\ x'' &\subseteq \{y \mid \psi(y, y_1, \ldots, y_n)\}, \end{aligned}$$

where y is a free variable of ψ. Like the previous extensions, the parameterized queries can be viewed simply as syntactic sugaring. For instance, the three last formulas are, respectively, equivalent to

$$\forall y(y \in x \leftrightarrow \psi),$$
$$\exists y(x' = y \wedge \psi), \text{ and}$$
$$\forall y(y \in x'' \rightarrow \psi).$$

In the following sections, we (informally) call *extended calculus* the calculus consisting of CALC^{cv} extended with the abbreviations described earlier (such as constructed and complex terms and, notably, parameterized queries).

20.4 Examples

We illustrate the previous two sections with a series of examples. The queries in the examples apply to schema $\{R, S\}$ with

$$sort(R) = \langle A : \mathbf{dom}, A' : \mathbf{dom}\rangle,$$
$$sort(S) = \langle B : \mathbf{dom}, B' : \{\mathbf{dom}\}\rangle.$$

For each query, we give an algebraic and a calculus expression.

Example 20.4.1 The union of R and a set of two constant tuples is given by

$$\{r \mid R(r) \vee r = \langle A : 3, A' : 5\rangle \vee r = \langle A : 0, A' : 0\rangle\}$$

or

$$R \cup \{\langle A : 3, A' : 5\rangle, \langle A : 0, A' : 0\rangle\}.$$

Example 20.4.2 The selection of the tuples from S, where the first component is a member of the second component, is obtained with

$$\{s \mid S(s) \wedge s.B \in s.B'\} \quad \text{or} \quad \sigma_{B \in B'}(S).$$

Example 20.4.3 The (classical) cross-product of R and S is the result of

$$\{t \mid \exists r, s(R(r) \wedge S(s) \wedge t = \langle A : r.A, A' : r.A', B : s.B, B' : s.B'\rangle)\}$$

or

$$\pi_{AA'BB'}(\sigma_{A=A''.A}(\sigma_{A'=A''.A'}(\sigma_{B=B''.B}(\sigma_{B'=B''.B'}(q))))),$$

where q is

$$tup\_create_{AA'BB'A''B''}(tup\_destroy(\pi_A(R)),\ tup\_destroy(\pi_{A'}(R)),\ tup\_destroy(\pi_B(S)),\ tup\_destroy(\pi_{B'}(S)),\ R,\ S).$$

Example 20.4.4 The join of R and S on $A = B$. This query is the composition of the cross-product of Example 20.4.3, with a selection. In Example 20.4.3, let the formula describing the cross-product be φ_3 and let $(R \times S)$ be the algebraic expression. Then the $(A = B)$ join of R and S is expressed by

$$\{t \mid \varphi_3(t) \wedge t.A = t.B\} \quad \text{or} \quad \sigma_{A=B}(R \times S).$$

Example 20.4.5 The renaming of the attributes of R to A_1, A_2 is obtained in the calculus by

$$\{t \mid \exists r(R(r) \wedge t.A_1 = r.A \wedge t.A_2 = r.A')\}$$

with t of sort $\langle A_1 : \mathbf{dom}, A_2 : \mathbf{dom}\rangle$. In the algebra, it is given by

$$\pi_{A_1A_2}(\sigma_{A_0.A=A_1}(\sigma_{A_0.A'=A_2}(tup\_create_{A_0A_1A_2}(R, tup\_destroy(\pi_A(R)), tup\_destroy(\pi_{A'}(R)))))).$$

Example 20.4.6 Flattening S means producing a set of flat tuples, each of which contains the first component of a tuple of S and one of the elements of the second component. This is the unnest operation $unnest_{B'}(\cdot)$ in the extended algebra, or in the calculus

$$\{t \mid \exists s(S(s) \wedge t.B = s.B \wedge t.C \in s.B')\},$$

where t is of sort $\langle B, C\rangle$. In the core algebra, this is slightly more complicated. We first obtain the set of values occurring in the B' sets using

$$E_1 = tup\_create_C(set\_destroy(tup\_destroy(\pi_{B'}(S)))).$$

We can next compute $(E_1 \times S)$ (using the same technique as in Example 20.4.3). Then the desired query is given by

$$\pi_{BC}(\sigma_{C \in B'}(E_1 \times S)).$$

Flattening can be extended to sorts with arbitrary nesting depth.

Example 20.4.7 The next example is a selection using $\subseteq$. Consider a relation T of sort $\langle C : \{\mathbf{dom}\}, C' : \{\mathbf{dom}\}\rangle$. We want to express the query

$$\{t \mid T(t) \wedge t.C \subseteq t.C'\}$$

in the algebra. We do this in stages:

$$F_1 = \sigma_{C'' \in C}(T \times tup\_create_{C''}(set\_destroy(tup\_destroy(\pi_C(T))))),$$
$$F_2 = \sigma_{C'' \in C'}(F_1),$$
$$F_3 = F_1 - F_2,$$
$$F_4 = T - \pi_{CC'}(F_3).$$

Observe that

1. A tuple $\langle C : U, C' : V, C'' : u\rangle$ is in F_1 if $\langle C : U, C' : V\rangle$ is in T and u is in U.
2. A tuple $\langle C : U, C' : V, C'' : u\rangle$ is in F_2 if $\langle C : U, C' : V\rangle$ is in T and u is in U and V.
3. A tuple $\langle C : U, C' : V, C'' : u\rangle$ is in F_3 if $\langle C : U, C' : V\rangle$ is in T and u is in $U - V$.
4. A tuple $\langle C : U, C' : V\rangle$ is in F_4 if it is in T and there is no u in $U - V$ (i.e., $U \subseteq V$).

Example 20.4.8 This example illustrates the use of nesting and of sets. Consider the algebraic query

$$nest_{C=(A)} \circ nest_{C'=(A')} \circ \sigma_{C=C'} \circ unnest_C \circ unnest_{C'}(R).$$

It is expressed in the calculus by

$$\{\langle x, y\rangle \mid \exists u(x \in u \wedge y \in u \wedge u = \{x' \mid R(x', y)\} \wedge u = \{y' \mid \{x' \mid R(x', y')\} = u\})\}.$$

A consequence of Theorem 20.7.2 is that this query is expressible in relational calculus or algebra. It is a nontrivial exercise to obtain a relational query for it. (See Exercise 20.24.)

Example 20.4.9 Our last example highlights an important difference between the flat relational calculus and CALC^{cv}. As shown in Proposition 17.2.3, the flat calculus cannot express the transitive closure of a binary relation. In contrast, the following CALC^{cv} query does:

$$\{y \mid \forall x(closed(x) \wedge contains\_R(x) \rightarrow y \in x)\},$$

where

- $closed(x) \equiv$

$$\forall u, v, w(\langle A : u, A' : v\rangle \in x \wedge \langle A : v, A' : w\rangle \in x \rightarrow \langle A : u, A' : w\rangle \in x);$$

- $contains\_R(x) \equiv \forall z(R(z) \rightarrow z \in x)$;
- $sort(x) = \{sort(R)\}$, $sort(y) = sort(z) = sort(R)$; and $sort(u) = sort(v) = sort(w) = \mathbf{dom}$.

Intuitively, the formula specifies the set of pairs y such that y belongs to each binary relation x containing R and transitively closed. This construction will be revisited in Section 20.6.

20.5 Equivalence Theorems

This section presents three results that compare the complex value algebra and calculus. First we establish the equivalence of the algebra and the domain-independent calculus. Next we develop a syntactic safeness condition for the calculus and show that it does not reduce expressive power. Finally we develop a natural syntactic condition on CALC^{cv} that yields a subset equivalent to ALG^{cv-}.

Our first result is as follows:

THEOREM 20.5.1 The algebra and the domain independent calculus for complex values are equivalent.

In the sketch of the proof, we present a simulation of the core algebra by the extended calculus and the analogous simulation in the opposite direction. An important component of this proof—namely, that the extended algebra (calculus) is no stronger than the core algebra (calculus)—is left for the reader (see Exercises 20.6, 20.7, 20.8, 20.10, and 20.11).

From Algebra to Calculus

We now show that for each algebra query, there is a domain-independent calculus query equivalent to it.

Let q be a named algebra query. We construct a domain-independent query $\{x \mid \varphi_q\}$ equivalent to q. The formula φ_q is constructed by induction on subexpressions of q. For a subexpression E of q, we define φ_E as follows:

(a) E is R for some $R \in \mathbf{R}$: φ_E is $R(x)$.

(b) E is $\{a\}$: φ_E is $x = a$.

(c) E is $\sigma_\gamma(E_1)$: φ_E is $\varphi_{E_1}(x) \wedge \Gamma$, where Γ is

$$x.A_i = x.A_j \text{ if } \gamma \equiv A_i = A_j; \quad x.A_i = a \text{ if } \gamma \equiv A_i = a;$$
$$x.A_i \in x.A_j \text{ if } \gamma \equiv A_i \in A_j; \quad x.A_i = x.A_j.C \text{ if } \gamma \equiv A_i = A_j.C.$$

(d) E is $\pi_{A_{i_1},\ldots,A_{i_k}}(E_1)$: φ_E is

$$\exists y(x = \langle A_{i_1} : y.A_{i_1}, \ldots, A_{i_k} : y.A_{i_k}\rangle \wedge \varphi_{E_1}(y)).$$

(e) For the basic set operations, we have

$$\varphi_{E_1 \cap E_2}(x) = \varphi_{E_1}(x) \wedge \varphi_{E_2}(x),$$
$$\varphi_{E_1 \cup E_2}(x) = \varphi_{E_1}(x) \vee \varphi_{E_2}(x),$$
$$\varphi_{E_1 - E_2}(x) = \varphi_{E_1}(x) \wedge \neg\varphi_{E_2}(x).$$

(f) E is $powerset(E_1)$: φ_E is $x \subseteq \{y \mid \varphi_{E_1}(y)\}$.

(g) E is $set\_destroy(E_1)$: φ_E is $\exists y(x \in y \wedge \varphi_{E_1}(y))$.

(h) E is $tup\_destroy(E_1)$: φ_E is $\exists y(\langle A : x\rangle = y \wedge \varphi_{E_1}(y))$, where A is the name of the field (of y).

(i) E is $tup\_create_{A_1,\ldots,A_n}(E_1, \ldots, E_n)$: φ_E is

$$\exists y_1, \ldots, y_n(x = \langle A_1 : y_1, \ldots, A_n : y_n\rangle \wedge \varphi_{E_1}(y_1) \wedge \cdots \wedge \varphi_{E_n}(y_n)).$$

(j) E is $set\_create(E_1)$: $x = \{y \mid \varphi_{E_1}(y)\}$.

We leave the verification of this construction to the reader (see Exercise 20.13). The domain independence of the obtained calculus query follows from the fact that algebra queries are domain independent.

From Calculus to Algebra

We now show that for each domain-independent query, there is a named algebra query equivalent to it.

Let $q = \{x \mid \varphi\}$ be a domain-independent query over $\mathbf{R}$. As in the flat relational case, we assume without loss of generality that associated with each variable x occurring in q (and also variables used in the following proof) is a unique, distinct attribute A_x in **att**. We use the active domain interpretation for the query, denoted as before with a subscript *adom*.

The crux of the proof is to construct, for each subformula ψ of φ, an algebra formula E_ψ that has the property that for each input $\mathbf{I}$,

$$E_\psi(\mathbf{I}) = \{y \mid \exists x_1, \ldots, x_n(y = \langle A_{x_1} : x_1, \ldots, A_{x_n} : x_n\rangle \wedge \psi(x_1, \ldots, x_n))\}_{adom}(\mathbf{I}),$$

where $x_1, \ldots, x_n$ is a listing of $free(\psi)$.

This construction is accomplished in three stages.

Computing the Active Domain The first step is to construct an algebra query E_{adom} having sort **dom** such that on input instance $\mathbf{I}$, $E_{adom}(\mathbf{I}) = adom(q, \mathbf{I})$. The construction of E_{adom} is slightly more intricate than the similar construction for the relational case. We prove by induction that for each sort τ, there exists an algebra operation F_τ that maps a set I of values of sort τ to $adom(I)$. This induction was not necessary in the flat case because the base relations had fixed depth. For the base case (i.e., $\tau = \mathbf{dom}$), it suffices to use for F_τ an identity operation (e.g., $tup\_create_A \circ tup\_destroy$). For the induction, the following cases occur:

1. τ is $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$ for $n \geq 2$. Then F_τ is

$$F_{\langle A_1:\tau_1 \rangle}(\pi_{A_1}) \cup \cdots \cup F_{\langle A_n:\tau_n \rangle}(\pi_{A_n}).$$

2. τ is $\langle A_1 : \tau_1 \rangle$. Then F_τ is $F_{\tau_1}(\mathit{tup\_destroy})$.
3. τ is $\{\tau_1\}$. Then F_τ is $F_{\tau_1}(\mathit{set\_destroy})$.

Now consider the schema $\mathbf{R}$. Then for each R in $\mathbf{R}$, $F_{sort(R)}$ maps a relation I over R to $adom(I)$. Thus $adom(q, \mathbf{I})$ can be computed with the query

$$E_{adom} = F_{sort(R_1)}(R_1) \cup \cdots \cup F_{sort(R_m)}(R_m) \cup \{a_1\} \cup \cdots \cup \{a_p\},$$

where $R_1, \ldots, R_m$ is the list of relations in $\mathbf{R}$ and $a_1, \ldots, a_p$ is the list of elements occurring in q.

Constructing Complex Values In the second stage, we prove by induction that for each sort τ, there exists an algebra query G_τ that constructs the set of values I of sort τ such that $adom(I) \subseteq adom(q, \mathbf{I})$. For $\tau = \mathbf{dom}$, we can use E_{adom}. For the induction, two cases occur:

1. τ is $\langle A_1 : \tau_1, \ldots, A_n : \tau_n \rangle$. Then G_τ is $\mathit{tup\_create}_{A_1,\ldots,A_n}(G_{\tau_1}, \ldots, G_{\tau_n})$.
2. τ is $\{\tau_1\}$. Then G_τ is $\mathit{powerset}(G_{\tau_1})$.

Last Stage We now describe the last stage, an inductive construction of the queries E_ψ for subformulas ψ of φ. We assume without loss of generality that the logical connectives $\vee$ and $\forall$ do not occur in φ. The proof is similar to the analogous proof for the flat case. We also assume that relation atoms in φ do not contain constants or repeated variables. We only present the new case (the standard cases are left as Exercise 20.13). Let ψ be $x \in y$. Suppose that x is of sort τ, so y is of sort $\{\tau\}$. The set of values of sort τ (or $\{\tau\}$) within the active domain is returned by query G_τ, or $G_{\{\tau\}}$. The query

$$\sigma_{A_x \in A_y}(\mathit{tup\_create}_{A_x,A_y}(G_\tau, G_{\{\tau\}}))$$

returns the desired result.

Observe that with this construction, E_φ returns a set of tuples with a single attribute A_x. The query q is equivalent to $\mathit{tup\_destroy}(E_\varphi)$.

As we did for the relational model, we can define a variety of syntactic restrictions of the calculus that yield domain-independent queries. We consider such restrictions next.

Safe Queries

We now turn to the development of syntactic conditions, called safe range, that ensure domain independence. These conditions are reminiscent of those presented for relational calculus in Chapter 5. As we shall see, a variant of safe range, called strongly safe range, will yield a subset of CALC^{cv}, denoted CALC^{cv-}, that is equivalent to ALG^{cv-}.

We could define safe range on the core calculus. However, such a definition would be cumbersome. A much more elegant definition can be given using the extended calculus. In particular, we consider here the calculus augmented with (1) constructed terms and (2) parameterized queries.

Recall that intuitively, if a formula is safe range, then each variable is bounded, in the sense that it is restricted by the formula to lie within the active domain of the query or the input. We now define the notions of safe formulas and safe terms. To give these definitions, we define the set of *safe-range variables* of a formula using the following procedure, which returns either the symbol $\perp$ (which indicates that some quantified variable is not bounded) or the set of free variables that are bounded. In this discussion, we consider only formulas in which universal quantifiers do not occur.

In the following procedure, if several rules are applicable, the one returning the largest set of safe-range variables (which always exists) is chosen.

procedure *safe-range* (sr)

input: a calculus formula φ

output: a subset of the free variables of φ or $\perp$. (In the following, for each Z, $\perp \cup Z = \perp \cap Z = \perp - Z = Z - \perp = \perp$.)

begin
(*pred* is a predicate in $\{=, \in, \subseteq\}$)
if for some parameterized query $\{x \mid \psi\}$ occurring as a term in φ, $x \notin sr(\psi)$ **then return** $\perp$
case φ **of**

$R(t)$	: $sr(\varphi) = free(t)$;
$(t\ pred\ t' \wedge \psi)$	: **if** ψ is safe and $free(t') \subseteq free(\psi)$ **then** $sr(\varphi) = free(t) \cup free(\psi)$;
$t\ pred\ t'$	: **if** $free(t') = sr(t')$ **then** $sr(\varphi) = free(t') \cup free(t)$; **else** $sr(\varphi) = \emptyset$;
$\varphi_1 \wedge \varphi_2$	: $sr(\varphi) = sr(\varphi_1) \cup sr(\varphi_2)$;
$\varphi_1 \vee \varphi_2$	: $sr(\varphi) = sr(\varphi_1) \cap sr(\varphi_2)$;
$\neg\varphi_1$	: $sr(\varphi) = \emptyset$;
$\exists x \varphi_1$	: **if** $x \in sr(\varphi_1)$ **then** $sr(\varphi) = sr(\varphi_1) - \{x\}$ **else return** $\perp$

end;

We say that a formula φ is *safe* if $sr(\varphi) = free(\varphi)$; and a query q is safe if its associated formula is safe.

It is important to understand how new sets are created in a safe manner. The next example illustrates two essential techniques for such creation.

EXAMPLE 20.5.2 Let R be a relation of sort $\langle A, B\rangle$. The powerset of R can be obtained in a safe manner with the query

$$\{x \mid x \subseteq \{y \mid R(y)\}\}.$$

For $\{y \mid R(y)\}$ is clearly a safe query (by the first case). Now letting $t \equiv x$, $t' \equiv \{y \mid R(y)\}$, the formula is safe (by the third case).

Now consider the nesting of the B column of R. It is achieved by the following query:

$$\{x \mid x = \langle z, \{y \mid R(z, y)\}\rangle \wedge \exists y'(R(z, y'))\}.$$

Let $t \equiv x$, $t' \equiv \langle z, \{y \mid R(z, y)\}\rangle$ and $\psi \equiv \exists y'(R(z, y'))$. First note that $sr(R(z, y))$ contains y, so the parameterized query $\{y \mid R(z, y)\}$ can be used safely. Next the formula ψ is safe. Finally the only free variable in t' is z, which is also free in ψ. Thus x is safe range (by the second case) and the query is safe.

As detailed in Section 20.7, the complex value algebra and calculus can express mappings with complexity corresponding to arbitrarily many nestings of exponentiation. In contrast, as discussed in that section, the nested relation algebra ALG^{cv-}, which uses the nest operator but not powerset, has complexity in PTIME. Interestingly, there is a minor variation of the safe-range condition that yields a subset of the calculus equivalent to ALG^{cv-}. Specifically, a formula is *strongly safe range* if it is safe range and the inclusion predicate does not occur in it. In the previous example, the nesting is strongly safe range whereas *powerset* is not.

We now have the following:

THEOREM 20.5.3

(a) The safe-range calculus, the domain-independent calculus, and ALG^{cv} coincide.

(b) The strongly safe-range calculus and ALG^{cv-} coincide.

Crux Consider (a). By inspection of the construction in the proof that $\text{ALG}^{cv} \sqsubseteq \text{CALC}^{cv}$, each algebra query is equivalent to a safe-range calculus query. Clearly, each safe-range calculus query is a domain-independent calculus query. We have already shown that each domain-independent calculus query is an algebra query.

Now consider (b). Observe that in the proof that $\text{ALG}^{cv} \sqsubseteq \text{CALC}^{cv}$, $\subseteq$ is used only for powerset. Thus each query in ALG^{cv-} is a strongly safe-range query. Now consider a strongly safe-range query; we construct an equivalent algebra query. We cannot use the construction from the proof of the equivalence theorem, because *powerset* is crucial for constructing complex domains. However, we can show that this can be avoided using the ranges of variables. (See Exercise 20.16.) More precisely, the brute force construction of the domain of variables using powerset is replaced by a careful construction based on the strongly safe-range restriction. The remainder of the proof stays unchanged. ■

Because of part (b) of the previous result, we denote the strongly safe-range calculus by CALC^{cv-}.

20.6 Fixpoint and Deduction

Example 20.4.9 suggests that the complex value algebra and calculus can simulate iteration. In this section, we examine iteration in the spirit of both fixpoint queries and datalog. In both cases, they do not increase the expressive power of the algebra or calculus. However, they allow us to express certain queries more efficiently.

Fixpoint for Complex Values

Languages with fixpoint semantics were considered in the context of the relational model to overcome limitations of relational algebra and calculus. In particular, we observed that transitive closure cannot be computed in relational calculus. However, as shown by Example 20.4.9, transitive closure can be expressed in the complex value algebra and calculus. Although transitive closure can be expressed in that manner, the use of *powerset* seems unnecessarily expensive. More precisely, it can be shown that *any* query in the complex value algebra and calculus that expresses transitive closure uses exponential space (assuming the straightforward evaluation of the query). In other words, the blowup caused by the *powerset* operator cannot be avoided. On the other hand, a fixpoint construct allows us to express transitive closure in polynomial space (and time). It is thus natural to develop fixpoint extensions of the calculus and algebra.

We can provide inflationary and noninflationary extensions of the calculus with recursion. As in the relational case, an *inflationary fixpoint operator* μ_T^+ allows the iteration of a CALC^{cv} formula $\varphi(T)$ up to a fixpoint. This essentially permits the inductive definition of relations, using calculus formulas. The calculus CALC^{cv} augmented with the inflationary fixpoint operator is defined similarly to the flat case (Chapter 14) and yields $\text{CALC}^{cv} + \mu^+$. We only consider the inflationary fixpoint operator. (Exercise 20.19 explores the noninflationary version.)

Theorem 20.6.1 $\text{CALC}^{cv} + \mu^+$ is equivalent to ALG^{cv} and CALC^{cv}.

The proof of this theorem is left for Exercise 20.18. It involves simulating a fixpoint in a manner similar to Example 20.4.9.

Before leaving the fixpoint extension, we show how powerset can be computed by iterating a ALG^{cv-} formula to a fixpoint. (We will see later that powerset cannot be computed in ALG^{cv-} alone.)

Example 20.6.2 Consider a relation R of sort **dom** (i.e., a set of atomic elements). The powerset of R is computed by $\{x \mid \mu_T(\varphi(T))(x)\}$, where T is of sort $\{\textbf{dom}\}$ and

$$\varphi(T)(y) \equiv [y = \emptyset \vee \exists x', y'(R(x') \wedge T(y') \wedge y = y' \cup \{x'\}.]$$

This formula is in fact equivalent to a query in ALG^{cv-}. (See Exercise 20.15.) For example, suppose that R contains $\{2, 3, 4\}$. The iteration of φ yields

$$\begin{array}{lll} J_0 & = \emptyset & \\ J_1 & = \varphi(\emptyset) & = \{\emptyset\} \\ J_2 & = \varphi(J_1) & = J_1 \cup \{\{2\}, \{3\}, \{4\}\} \\ J_3 & = \varphi(J_2) & = J_2 \cup \{\{2, 3\}, \{2, 4\}, \{3, 4\}\} \\ J_4 & = \varphi(J_3) & = J_3 \cup \{\{2, 3, 4\}\}, \end{array}$$

and J_4 is a fixpoint and coincides with $powerset(\{2, 3, 4\})$.

Datalog for Complex Values

We now briefly consider an extension of datalog to incorporate complex values. The basic result is that the extension is equivalent to the complex value algebra and calculus. We also consider a special grouping construct, which can be used for set construction in this context.

In the datalog extension considered here, the predicates $\subseteq$ and $\in$ are permitted. A rule is *safe range* if each variable that appears in the head also appears in the body, and the body is safe (i.e., the conjunction of the literals of the body is a safe formula). We assume henceforth that rules are safe. Stratified negation will be used. The language is illustrated in the following example.

EXAMPLE 20.6.3 The input is a relation R of sort $\langle A, B : \{\langle C, C' \rangle\}\rangle$. Consider the query defining an *idb* relation T, which contains the tuples of R, with the B-component replaced by its transitive closure. Let us assume that we have a ternary relation *ins*, where $ins(w, y, z)$ is interpreted as "z is obtained by inserting w into y." We show later how to define this relation in the language. The program consists of the following rules:

$$(r1) \qquad S(x, y) \leftarrow R(x, y)$$

$$(r2) \qquad S(x, z) \leftarrow S(x, y), u \in y, v \in y, u.C' = v.C, ins(\langle u.C, v.C' \rangle, y, z)$$

$$(r3) \qquad S'(x, z) \leftarrow S(x, z), S(x, z'), z \subseteq z', z \neq z'$$

$$(r4) \qquad T(x, z) \leftarrow S(x, z), \neg S'(x, z).$$

The first two rules compute in S pairs corresponding to pairs from R, such that the second component of a pair contains the corresponding component from the pair in R and possibly additional elements derived by transitivity. Obviously, for each pair $\langle x, y \rangle$ of R, there is a pair $\langle x, z \rangle$ in S, such that z is the transitive closure of y, but there are other tuples as well. To answer the query, we need to select for each x the unique tuple $\langle x, z \rangle$ of S, where z is maximal.[2] The third rule puts into S' tuples $\langle x, z \rangle$ such that z is not maximal for that x. The last rule then selects those that are maximal, using negation.

[2] We assume, for simplicity, that the first column of R is a key. It is easy to change the rules for the case when this does not hold.

We now show the program that defines *ins* for some given sort τ (the variables are of sort $\{\tau\}$ except for w, which is of sort τ):

$$\begin{aligned}
&\mathit{super}(w, y, z) && \leftarrow w \in z, y \subseteq z\\
&\mathit{not\text{-}min\text{-}super}(w, y, z) && \leftarrow \mathit{super}(w, y, z), \mathit{super}(w, y, z'), z' \subseteq z, z' \neq z\\
&\mathit{ins}(w, y, z) && \leftarrow \mathit{super}(w, y, z), \neg \mathit{not\text{-}min\text{-}super}(w, y, z)
\end{aligned}$$

Note that the program is sort specific only through its dependence on the sorts of the variables. The same program computes *ins* for another sort τ', if we assume that the sort of w is τ' and that of the other variables is $\{\tau'\}$. Note also that the preceding program is not safe. To make it safe, we would have to use derived relations to range restrict the various variables.

We note that although we used $\subseteq$ in the example as a built-in predicate, it can be expressed using membership and stratified negation.

The proof of the next result is omitted but can be reconstructed reasonably easily using the technique of Example 20.6.3.

THEOREM 20.6.4 A query is expressible in datalog^{cv} with stratified negation if and only if it is expressible in CALC^{cv}.

The preceding language relies heavily on negation to specify the new sets. We could consider more set-oriented constructs. An example is the *grouping* construct, which is closely related to the algebraic nest operation. For instance, in the language $\mathcal{LDL}$, the rule:

$$S(x, \langle y \rangle) \leftarrow R(x, y)$$

groups in S, for each x, all the y's related to it in R (i.e., S is the result of the nesting of R on the second coordinate).

The grouping construct can be used to simulate negation. Consider a query q whose input consists of two unary relations R, S not containing some particular element a and that computes $R - S$. Query q can be answered by the following $\mathcal{LDL}$ program:

$$\begin{aligned}
&\mathit{Temp}(x, a) \leftarrow R(x)\\
&\mathit{Temp}(x, x) \leftarrow S(x)\\
&T(x, \langle y \rangle) \leftarrow \mathit{Temp}(x, y)\\
&\mathit{Res}(x) \leftarrow T(x, \{a\})
\end{aligned}$$

Note that for an x in $R - S$, we derive $T(x, \{a\})$; but for x in $R \cap S$, we derive $T(x, \{x, a\}) \neq T(x, \{a\})$ because a is not in R.

From the previous example, it is clear that programs with grouping need not be monotone. This gives rise to semantic problems similar to those of negation. One possiblity, adopted in $\mathcal{LDL}$, is to define the semantics of programs with grouping analogously to stratification for negation.

20.7 Expressive Power and Complexity

This section presents two results. First the expressive power and complexity of ALG^{cv}/CALC^{cv} is established—it is the family of queries computable in hyperexponential time. Second, we consider the expressive power of ALG^{cv-}/CALC^{cv-} (i.e., in algebraic terms the expressive power of permitting the *nest* operator, but not *powerset*). Surprisingly, we show that the *nest* operator can be eliminated from ALG^{cv-} queries with flat input/ouput.

Complex Value Languages and Elementary Queries

We now characterize the queries in ALG^{cv} in terms of the set of computable queries in a certain complexity class. First the notion of computable query is extended to the complex value model in the straightforward manner. The complexity class of interest is the class of *elementary queries*, defined next.

The *hyperexponential* functions hyp_i for i in N are defined by

1. $hyp_0(m) = m$; and
2. $hyp_{i+1}(m) = 2^{hyp_i(m)}$ for $i \geq 0$.

A query is an *elementary query* if it is a computable query and has hyperexponential time data complexity[3] w.r.t. the database size. By database size we mean the amount of space it takes to write the content of the database using some natural encoding. Note that, for complex value databases, size can be very different from cardinality. For example, the database could consist of a single but very large complex value.

It turns out that a query is in ALG^{cv}/CALC^{cv} iff it is an elementary query.

THEOREM 20.7.1 A query is in ALG^{cv}/CALC^{cv} iff it is an elementary query.

Crux It is trivial to see that each query in ALG^{cv}/CALC^{cv} is elementary. All operations can be evaluated in polynomial time in the size of their arguments except for powerset, which takes exponential time.

Conversely, let q be of complexity hyp_n. We show how to compute it in CALC^{cv}.

Suppose first that an enumeration of $adom(\mathbf{I})$ is provided in some binary relation *succ*. (We explain later how this is done.) We prove that q can then be computed in CALC$^{cv}+\mu^+$. Let $X^0 = adom(I)$ and for each i, $X^i = powerset(X^{i-1})$. Observe that for each X^i, we can provide an enumeration as follows: First *succ* provides the enumeration for X^0; and for each i, we define $V <_i U$ for U, V in X^i if there exists x in $U - V$ such that each element larger than x (under $<_{i-1}$) is in both or neither of U, V. Clearly, there exists a query in CALC$^{cv}+\mu^+$ that constructs X^n and a binary relation representing $<_n$.

Now we view each element of X^n as an atomic element. The input instance together with X^n and the enumeration can be seen as an ordered database with size the order of hyp_n. Query q is now polynomial in this new (much larger) instance. Finally we can easily

[3] We are concerned exclusively with the *data* complexity. Observe that when considering the union of hyperexponential complexities, time and space coincide.

extend to complex values the result from the flat case that CALC+μ^+ can express QPTIME on ordered databases (Theorem 17.4.2). Thus CALC^{cv}+μ^+ can also express all QPTIME queries on ordered complex value databases, so q can be computed in CALC^{cv}+μ^+ using $<_n$ on X_n. By Theorem 20.6.1, CALC^{cv}+μ^+ is equivalent to CALC^{cv}, so there exists a CALC^{cv} query φ computing q if an (arbitrary) enumeration of the active domain is given in some binary relation *succ*.

To conclude the proof, it remains to remove the restriction on the existence of an enumeration of the active domain. Let φ' be the formula obtained from φ by replacing

1. *succ* by some fresh variable y (the sort of y is set of pairs); and
2. each literal $succ(t, t')$ by $\langle t, t' \rangle \in y$.

Then q can be computed by

$$\exists y(\varphi' \wedge \psi).$$

where ψ is the CALC^{cv} formula stating that y is the representation in a binary relation of an enumeration of the active domain. (Observe that it is easy to state in CALC^{cv} that the content of a binary relation is an enumeration.) ■

On the Power of the *nest* Operator

The *set-height* of a complex sort is the maximum number of set constructors in any branch of the sort. We can exhibit hierarchies of classes of queries in CALC^{cv} based on the set-height of the sorts of variables used in the query. For example, consider all queries that take as input a flat relational schema and produce as output a flat relation. Then for each $n > 0$, the family of CALC^{cv} queries using variables that have sorts with set-height $\leq n$ is strictly weaker than the family of CALC^{cv} queries using variables that have sorts with set-height $\leq n + 1$. A similar hierarchy exists for ALG^{cv}, based on the sorts of intermediate types used. Intuitively, these results follow from the use of the powerset operator, which essentially provides an additional exponential amount of scratch paper for each additional level of set nesting.

The bottom of this hierarchy is simply relational calculus. Recall that ALG^{cv-} can use the *nest* operator but not the powerset operator. It is thus natural to ask, Where do ALG^{cv-}/CALC^{cv-} (assuming flat input and output) lie relative to the relational calculus and the first level of the hierarchy? Rather surprisingly, it turns out that the *nest* operator alone does not increase expressive power. Specifically, we show now that with flat input and output, ALG^{cv-}/CALC^{cv-} is equivalent to relational calculus.

THEOREM 20.7.2 Let φ be a CALC^{cv-}/ALG^{cv-} query over a relational database schema $\mathbf{R}$ with output of relational sort S. Then there exists a relational calculus query φ' equivalent to φ.

Crux The basic intuition underlying the proof is that with a flat input in CALC^{cv-} or ALG^{cv-}, each set constructed at an intermediate stage can be identified by a tuple of atomic

values. In terms of ALG^{cv-}, the intuitive reason for this is that sets can be created only in two ways:

- by *nest*, which builds a relation whose nonnested coordinates form a key for the nested one, and
- by *set_create*, which can build only singleton sets.

Thus all created sets can be identified using some flat key of bounded length. The sets can then be simulated in the computation by their flat representations. The proof consists of

- providing a careful construction of the flat representation of the sets created in the computation, which reflects the history of their creation; and
- constructing a new query, equivalent to the original one, that uses only the flat representations of sets.

The details of the proof are omitted. ■

Observe that an immediate consequence of the previous result is that transitive closure or powerset are *not* expressible in ALG^{cv-}.

REMARK 20.7.3 The previous results focus on relational queries. The same technique can be used for nonflat inputs. An arbitrary input $\mathbf{I}$ can be represented by a flat database $\mathbf{I}_f$ of size polynomial in the size of the input. Now an arbitrary ALG^{cv-} query on $\mathbf{I}$ can be simulated by a relational query on $\mathbf{I}_f$ to yield a flat database representing the result. Finally the complex object result is constructed in polynomial time. This shows in particular that ALG^{cv-} is in PTIME. ■

20.8 A Practical Query Language for Complex Values

We conclude our discussion of languages for complex values with a brief survey of a fragment of the query language O_2SQL supported by the commercial object-oriented database system O_2 (see Chapter 21). This fragment provides an elegant syntax for accessing and constructing deeply nested complex values, and it has been incorporated into a recent industrial standard for object-oriented databases.

For the first example we recall the query

(4.3) What are the address and phone number of the Le Champo?

Using the **CINEMA** database (Fig. 3.1), this query can be expressed in O_2SQL as

```
element select tuple ( t.address, t.phone )
        from  t in Location
        where t.name = "Le Champo"
```

The *select-from-where* clause has semantics analogous to those for SQL. Unlike SQL, the *select* part can specify an essentially arbitrary complex value, not just tuples. A *select-from-where* clause returns a set[4]; the keyword **element** here is a desetting operator that returns a runtime error if the set does not have exactly one element.

The next example illustrates how O_2SQL can work inside nested structures. Recall the complex value shown in Fig. 20.2, which represents a portion of the **CINEMA** database. Let the full complex value be named *Films*. The following query returns all movies for which the director does not participate as an actor.

```
select  m.Title
from    f in Films
        m in f.Movies
where f.Director not in select a
                          from  a in m.Actors
```

O_2SQL also provides a mechanism for collapsing nested sets. Again using the complex value *Films* of Fig. 20.2, the following gives the set of all directors that have not acted in any Hitchcock film.

```
select  f.Director
from    f in Films
where f.Director not in flatten select m.Actors
                                 from  g in Films
                                       m in g.Movies
                                 where g.Director = "Hitchcock"
```

Here the inner *select-from-where* clause returns a set of sets of actors. The keyword **flatten** has the effect of forming the union of these sets to yield a set of actors.

We conclude with an illustration of how O_2SQL can be used to construct a deeply nested complex value. The following query builds, from the complex value *Films* of Fig. 20.2, a complex value of the same type that holds information about all movies for which the director does not serve as an actor.

```
select tuple ( Director: f.Director,
               Movies: select tuple ( Title: m.Title,
                                      Actors: select a
                                              from a in m.Actors )
                       from m in f.Movies
                       where f.Director not in m.Actors )
from f in Films
```

[4] In the full language O_2SQL, a list or bag might also be returned; we do not discuss that here. Furthermore, we do not include the keyword **unique** in our queries, although technically it should be included to remove duplicates from answer sets.

Bibliographic Notes

The original proposal for generalizing the relational model to allow entries in relations to be sets is often attributed to Makinouchi [Mak77]. Our presentation is strongly influenced by [AB88]. An extensive coverage of the field can be found in [Hul87]. The nested relation model is studied in [JS82, TF86, RKS88]. The V-relation model is studied in [BRS82, AB86, Ver89], and the essentially equivalent partition normal form (PNF) nested relation model is studied in [RKS88]. The connection of the PNF nested relations with dependencies has also been studied (e.g., in [TF86, OY87]). References [DM86a, DM92] develop a *while*-like language that expresses all computable queries (in the sense of [CH80b]) over "directories"; these are database structures that are essentially equivalent to nested relations.

There have been many proposals of algebras. In general, the earlier ones have essentially the power of ALG^{cv-} (due to obvious complexity considerations). The powerset operation was first proposed for the Logical Data Model of [KV84, KV93b].

The calculus presented in this chapter is based on Jacobs's calculus [Jac82]. This original proposal allowed noncomputable queries [Var83]. We use in this chapter a computable version of that calculus that is also used (with minor variations) in [KV84, KV93b, AB88, RKS88, Hul87].

Parameterized queries are close to the commonly used mathematical concept of *set comprehension*.

The equivalence of the algebra and the calculus has been shown in [AB88]. An equivalence result for a more general model had been previously given in [KV84, KV93b]. The equivalence result is preserved with oracles. In particular, it is shown in [AB88] that if the algebra and the calculus are extended with an identical set of oracles (i.e., sorted functions that are evaluated externally), the equivalence result still holds.

The strongly safe-range calculus, and the equivalence of ALG^{cv-} and CALC^{cv-}, are based on [AB88].

The fact that transitive closure can be computed in the calculus was noted in [AB88]. The result that any algebra query computing transitive closure requires exponential space (with the straightforward evaluation model) was shown in [SP94]. The equivalence between the calculus and various rule-based languages is from [AB88]. In the rule-based paradigm, nesting can be expressed in many ways. A main difference between various proposals of logic programming with a set construct is in their approach to nesting: grouping in $\mathcal{LDL}$ [BNR$^+$87], data functions in COL [AG91], and a form of universal quantification in [Kup87]. In [Kup88], equivalence of various rule-based languages is proved. In [GG88], it is shown that various programming primitives are interchangeable: powerset, fixpoint, various iterators.

The correspondence between ALG^{cv}/CALC^{cv} queries and elementary queries is studied in [HS93, KV93a]. Hierarchies of classes of queries based on the level of set nesting are considered in [HS93, KV93a]. Related work is presented in [Lie89a]. Exact complexity characterizations are obtained with fixpoint, which is no longer redundant when the level of set nesting is bounded [GV91].

Theorem 20.7.2 is from [PG88], which uses a proof based on a strongly safe calculus.

The proof of Theorem 20.7.2 outlined in this chapter suggests a strong connection between ALG^{cv-} and the V-relation model.

Reference [BTBW92] introduces a rich family of languages for complex objects, extended to include lists and bags, that is based on structural recursion. One language in this family corresponds to the nested algebra presented in this chapter. Using this, an elegant family of generalizations of Theorem 20.7.2 is developed in [Won93].

An extension of complex values, called *formats* [HY84], includes a marked union construct in addition to tuple and finitary set. Abstract notions of relative information capacity are developed there; for example, it can be shown that two complex value types have equivalent information capacity iff they are isomorphic.

Exercises

Exercise 20.1 (V-relations) Consider the schema R of sort

$$\langle A, B : \{\langle C, D\rangle\}\rangle.$$

Furthermore, we impose the fd $A \to B$ (more precisely, the generalization of a functional dependency). (a) Prove that for each instance I of R, the size of I is bounded by a polynomial in $adom(I)$. (b) Show how the same information can be naturally represented using two flat relations. (One suffices with some coding.) (c) Formalize the notion of V-relation of Section 20.1 and generalize the results of (a) and (b).

Exercise 20.2 Consider a (flat) relation R of sort

$$name\ age\ address\ car\ child\_name\ child\_age$$

and the multivalued dependency $name\ age\ address \twoheadrightarrow car$. Prove that the same information can be stored in a complex value relation of sort

$$\langle name, age, address, cars : \{\mathbf{dom}\}, children : \{\langle child\_name, child\_age\rangle\}\rangle$$

Discuss the advantages of this alternative representation. (In particular, show that for the same data, the size of the instance in the second representation is smaller. Also consider update anomalies.)

Exercise 20.3 Consider the value

$$\{\ \langle A : a, B : \langle A : \{a, b\}, B : \langle A : a\rangle\rangle, C : \langle\ \rangle\rangle,$$
$$\langle A : a, B : \langle A : \{\}, B : \langle A : a\rangle\rangle, C : \langle\ \rangle\rangle\ \}.$$

Show how to construct it in the core algebra from $\{a\}$ and $\{b\}$.

Exercise 20.4 Prove that for each complex value relation I, there exists a constant query in the core algebra returning I.

Exercise 20.5 Let **R** be a database schema consisting of a relation R of sort

$$\langle A : \mathbf{dom},\ B : \langle A : \{\mathbf{dom}\},\ B : \langle A : \mathbf{dom}\rangle\rangle,\ C : \langle\ \rangle\rangle;$$

and let $\tau = \{\langle A : \mathbf{dom},\ B : \{\{\mathbf{dom}\}\}\rangle\}$.

(a) Give a query computing for each $\mathbf{I}$ over $\mathbf{R}$, $adom(\mathbf{I})$.

(b) Give a query computing the set of values J of sort τ such that $adom(J) \subseteq adom(\mathbf{I})$.

Exercise 20.6 Prove that *set_create* can be expressed using the other operations of the core algebra. *Hint:* Use *powerset*.

Exercise 20.7 Formally define the following operations: (a) renaming, (b) singleton, (c) cross-product, and (d) join. In each case, prove that the operation is expressible in ALG^{cv}. Which of these can be expressed without powerset?

Exercise 20.8 (Nest,unnest)

(a) Show that *nest* is expressible in ALG^{cv}.

(b) Show that *unnest* is expressible in ALG^{cv} without using the powerset operator.

(c) Prove that $unnest_A$ is a right inverse of $nest_{A=(A_1 \ldots A_k)}$ and that $unnest_A$ has no right inverse.

Exercise 20.9 (Map) The operation $map_{C,q}$ is applicable to relations of sort τ where τ is of the form $\{\langle C : \{\tau'\}, \ldots\rangle\}$ and q is a query over relations of sort τ'. For instance, let

$$I = \{\langle C : I_1, C' : J_1\rangle, \langle C : I_2, C' : J_2\rangle, \langle C : I_3, C' : J_3\rangle\}.$$

Then

$$map_{C,q}(I) = \{\langle C : q(I_1), C' : J_1\rangle, \langle C : q(I_2), C' : J_2\rangle, \langle C : q(I_3), C' : J_3\rangle\}.$$

(a) Give an example of *map* and show how the query of this example can be expressed in ALG^{cv}.

(b) Give a formal definition of *map* and prove that the addition of *map* does not change the expressive power of the algebra.

Exercise 20.10 Show how to express

$$\{x \mid \{y \mid liked(x, y)\} = \{y \mid saw(x, y)\}\}$$

in the core calculus.

Exercise 20.11 The calculus is extended by allowing terms of the form $z \cup z'$ and $z - z'$ for each set term z, z' of identical sort. Prove that this does not modify the expressive power of the language. More generally, consider introducing in the calculus terms of the form $q(t_1, \ldots, t_n)$, where q is an n-ary algebraic operation and the t_i are set terms of appropriate sort.

Exercise 20.12 Give five queries on the **CINEMA** database expressed in ALG^{cv}. Give the same queries in CALC^{cv}.

Exercise 20.13 Complete the proof that $\text{ALG}^{cv} \sqsubseteq \text{CALC}^{cv}$ for Theorem 20.5.1. Complete the proof of "Last Stage" for Theorem 20.5.1.

Exercise 20.14 This exercise elaborates the simulation of CALC^{cv} by ALG^{cv} presented in the proof of Theorem 20.5.1. In particular, give the details of

(a) the construction of E_{adom}

(b) the construction of G_τ for each τ

(c) the last stage of the construction.

Exercise 20.15 Show that the query in Example 20.6.2 is strongly safe range (e.g., give a query in ALG^{cv-} or CALC^{cv-} equivalent to it).

Exercise 20.16 Show that every strongly safe-range query is in ALG^{cv-} [one direction of (b) of Theorem 20.5.3].

Exercise 20.17 Sketch a program expressing the query *even* in $\text{CALC}^{cv}+\mu^+$.

Exercise 20.18 Prove that $\text{CALC}^{cv}+\mu^+ = \text{ALG}^{cv}$.

Exercise 20.19 Define a *while* language based on ALG^{cv}. Show that it does not have more power than ALG^{cv}.

Exercise 20.20 Consider a query q whose input consists of two relations *blue*, *red* of sort $\langle A, B\rangle$ (i.e., consists of two graphs). Query q returns a relation of sort $\langle A, B : \{\textbf{dom}\}\rangle$ with the following meaning. A tuple $\langle x, X\rangle$ is in the result if x is a vertex and X is the set of vertexes y such that there exists a path from x to y alternating blue and red edges. Prove in one line that q is expressible in ALG^{cv}. Show how to express q in some complex value language of this chapter.

Exercise 20.21 Generalize the construction of Example 20.6.2 to prove Theorem 20.6.1.

Exercise 20.22 Datalog with stratified negation was shown to be weaker than datalog with inflationary negation. Is the situation similar for datalog^{cv} with negation?

Exercise 20.23 Exhibit a query that is not expressible in CALC^{cv-} but is expressible in CALC^{cv}, and one that is not expressible in CALC^{cv}.

Exercise 20.24 Give a relational calculus formula or algebra expression for the query in Example 20.4.8.

★ **Exercise 20.25** Recall the language $while_N$ from Chapter 18. The language allows assignments of relational algebra expressions to relational variables, looping, and integer arithmetic. Let $while_N^{cv}$ be like $while_N$, except that the relational algebra expressions are in ALG^{cv}. Prove that $while_N^{cv}$ can express all queries from flat relations to flat relations.

21 Object Databases

Minkisi are complex objects clearly not the product of a momentary impulse. . . . To do justice to objects, a theory of them must be as complex as them.[1]

—Wyatt MacGaffey in *Astonishment and Power*

Alice: *What is a Minkisi?*
Sergio: *It is an African word that translates somewhat like "things that do things."*
Vittorio: *It is art, religion, and magic.*
Riccardo: *Oh, this sounds to me very object oriented!*

In this chapter, we provide a brief introduction to object-oriented databases (OODBs). A complete coverage of this new and exciting area is beyond the scope of this volume; we emphasize the new modeling features of OODBs and some of the preliminary theoretical research about them. On the one hand, we shall see that some of the most basic issues concerning OODBs, such as the design of query languages or the analysis of their expressive power, can be largely resolved using techniques already developed in connection with the relational and complex value models. On the other hand, the presence of new features (such as object identifiers) and methods brings about new questions and techniques.

As mentioned previously, the simplicity of the data structure in the relational model often hampers its use in many database applications. A relational representation can obscure the intention and intricate semantics of a complex data structure (e.g., for holding the design of a VLSI chip or an airplane wing). As we shall see, OODBs remedy this situation by borrowing a variety of data structuring constructs from the complex value model (Chapter 20) and from semantic data models (considered in Chapter 11). At a more fundamental level, the relational data model and all of the data models presented so far impose a sharp distinction between data storage and data processing: The DBMS provides data storage, but data processing is provided by a host programming language with a relatively simple language such as SQL embedded in it. OODBs permit the incorporation of behavioral portions of the overall data management application directly into the database schema, using methods in the sense of object-oriented programming languages.

This chapter begins with an informal presentation of the underlying constructs of OODBs. Next a formal definition for a particular OODB model is presented. Two directions of theoretical research into OODBs are then discussed. First a family of languages

[1] Reprinted with permission. Smithsonian Institution Press ©1993.

for data access is presented, with an emphasis on how the languages interact with the novel modeling constructs (of particular interest is the impact of generalizing the notion of complete query language to accommodate the presence of object identifiers, OIDs). Next two languages for methods are described. The first is an imperative language allowing us to specify methods with side effects.[2] The second language brings us to a functional perspective on methods and database languages and allows us to specify side-effect-free methods. In both cases, we present some results on type safety and expressive power. Checking type safety is generally undecidable; we identify a significant portion of the functional language, monadic method schemas, for which type safety is decidable. With respect to expressive power, the imperative language is complete in an extended sense formalized in this chapter. The functional language expresses precisely QPTIME on ordered inputs and so turns out to express the by-now-famous *fixpoint* queries. The chapter concludes with a brief survey of additional research issues raised by OODBs.

21.1 Informal Presentation

Object-oriented database models stem from a synthesis of three worlds: the complex value model, semantic database models, and object-oriented programming concepts. At the time of writing, there is not widespread agreement on a specific OODB model, nor even on what components are required to constitute an OODB model. In this section, we shall focus on seven important ingredients of OODB models:

1. objects and object identifiers;
2. complex values and types;
3. classes;
4. methods;
5. ISA hierarchies;
6. inheritance and dynamic binding;
7. encapsulation.

In this section, we describe and illustrate these interrelated notions informally; a more formal definition is presented in the following section. We will also briefly discuss alternatives.

As a running example for this discussion, we shall use the OODB schema specified in Fig. 21.1. This schema is closely related to the semantic data model schema of Fig. 11.1, which in turn is closely related to the **CINEMA** example of Chapter 3.

As discussed in Chapter 11, a significant shortcoming of the relational model is that it must use printable values, often called *keys*, to refer to entities or objects-in-the-world. As a simple example, suppose that the first and last names of a person are used as a key to identify that person. From a physical point of view, it is then cumbersome to refer to a person, because the many bytes of his or her name must be used. A more fundamental

[2] Methods are said to have side-effects if they cause updates to the database.

```
(* schema and base definitions *)

create schema PariscopeSchema ;
create base PariscopeBase;

(* class definitions *)

class Person
   type tuple ( name: string, citizenship: string, gender: string );
class Director inherit Person
   type tuple ( directs: set ( Movie ) );
class Actor inherit Person
   type tuple ( acts_in: { Movie },
         award: { tuple ( prize: string, year: integer ) } );
class Actor_Director inherit Director, Actor
class Movie
   type tuple ( title: string, actors: set ( Actor );
         director: Director );
class Theater
   type tuple ( name: string, address: string, phone: string );

(* name definitions *)

name Pariscope: set ( tuple ( theater: Theater, time: string, price: integer,
         movie: Movie ) );
name Persons_I_like: set ( Person );
name Actors_I_like, Actors_you_like: set ( Actor );
name My_favorite_director : Director

(* method definitions *)

method get_name in class Person : string
   { if (gender = "male")
         return "Mr." + self.name;
      else
         return "Ms." + self.name }

method get_name in class Director : string
   { return ( "Director" + self.name ) };

method get_name in class Actor_Director : string
   { return ( "Director" + self.name ) };

(* we assume here that '+' denotes a string concatenation operator *)
```

Figure 21.1: An OODB Schema

problem arises if the person changes his or her name (e.g., as the result of marriage). When performing this update, conceptually there is a break in the continuity in the representation of the person. Furthermore, care must be taken to update all tuples (typically arising in a number of different relations) that refer to this person, to reflect the change of name.

Following the spirit of semantic data models, OODB models permit the explicit representation of physical and conceptual objects through the use of object identifiers (OIDs). Conceptually, a unique OID is assigned to each object that is represented in the database, and this association between OID and object remains fixed, even as attributes of the object (such as name or age) change in value. The use of objects and OIDs permits OODBs to share information gracefully; a given object o is easily shared by many other objects simply by referencing the OID of o. This is especially important in the context of updates; for example, the name of a person object o need be changed in only one place even if o is shared by many parts of the database.

In an OODB, a complex value is associated with each object. This complex value may involve printables and/or OIDs (i.e., references to the same or other objects). For example, each object in the class *Movie* in Fig. 21.1 has an associated triple whose second coordinate contains a set of OIDs corresponding to actors. In this section, we focus on complex values constructed using the tuple and set construct. In practical OODB models, other constructs are also supported (including, for example, bags and lists). Some commercial OODBs are based on an extension of C++ that supports persistence; in these models essentially any C++ structure can serve as the value associated with an object.

Objects that have complex values with the same type may be grouped into classes, as happens in semantic data models. In the running example, these include *Person*, *Director*, and *Movie*. Classes also serve as a natural focal point for associating some of the behavioral (or procedural) components of a database application. This is accomplished by associating with each class a family of methods for that class. Methods might be simple (e.g., producing the name of a person) or arbitrarily complex (e.g., displaying a representation of an object to a graphical interface or performing a stress analysis of a proposed wing design). A method has a name, a signature, and an implementation. The name and signature serve as an external interface to the method. The implementation is typically written in a (possibly extended) programming language such as C or C++. The choice of implementation language is largely irrelevant and is generally not considered to be part of the data model.

As with semantic models, OODB models permit the organization of classes into a hierarchy based on what have been termed variously ISA, specialization, or class-subclass relationships. The term *hierarchy* is used loosely here: In many cases any directed acyclic graph (DAG) is permitted. In Fig. 21.1 the ISA hierarchy has *Director* and *Actor* as (immediate) specializations of *Person* and *Actor_Director* as a specialization of both *Director* and *Actor*. Following the tradition of object-oriented programming languages, a virtual class **any** is included that serves as the unique root of the ISA hierarchy.

In OODB models, there are two important implications of the statement that class c' is a subclass of c. First it is required that the complex value type associated with c' be a subtype (in the sense formally defined later) of the complex value type associated with c. Second it is required that if there is a method with name m associated with c, then there is also a method with name m associated with c'. In some cases, the implementation (i.e., the actual code) of m for c' is identical to that for c; in this case the code of m for c' need not

be explicitly specified because it is *inherited* from c. In other cases, the implementation of m for c' is different from that for c; in which case we say that the implementation of m for c' *overrides* the implementation of m for c. (See the different implementations for method *get_name* in Fig. 21.1.) The determination of what implementation is associated with a given method name and class is called *method resolution*. A method is invoked with respect to an object o, and the class to which o belongs determines which implementation is to be used. This policy is called *dynamic binding*. As we shall see, the interaction of method calls and dynamic binding in general makes type checking for OODB schemas undecidable. (It is undecidable to check whether such a schema would lead to a runtime type error; on the other hand, it is clearly possible to find decidable sufficient conditions that will guarantee that no such error can arise.)

In the particular OODB model presented here, both values (in the style of complex values) and objects are supported. For example, in Fig. 21.1 a persistent set of triples called *Pariscope* is supported (see also Fig. 11.1). The introduction of values not directly associated with OIDs is a departure from the tradition of object-oriented programming, and not all OODBs in the literature support it. However, in databases the use of explicit values often simplifies the design and use of a schema. Their presence also facilitates expressing queries in a declarative manner.

The important principle of encapsulation in object orientation stems from the field of abstract data types. Encapsulation is used to provide a sharp boundary between how information about objects is accessed by database users and how that information is actually stored and provided. The principle of encapsulation is most easily understood if we distinguish two categories of database use: *dba mode*, which refers to activities unique to database administrators (including primarily creating and modifying the database schema), and *user mode*, which refers to activities such as querying and updating the actual data in the database. Of course, some users may operate in both of these modes on different occasions. In general, application software is viewed as invoked from the user mode.

Encapsulation requires that when in user mode, a user can access or modify information about a given object only by means of the methods defined for that object; he or she cannot directly examine or modify the complex value or the methods associated with the object. In particular, then, essentially all application software can access objects only through their methods. This has two important implications. First, as long as the same set of methods is supported, the underlying implementation of object methods, and even of the complex value representation of objects, can be changed without having to modify any application software. Second, the methods of an object often provide a focused and abstracted interface to the object, thus making it simpler for programmers to work with the objects.

In object-oriented programming languages, it is typical to enforce encapsulation except in the special case of rewriting method implementations. In some OODB models, there is an important exception to this in connection with query languages. In particular, it is generally convenient to permit a query language to examine explicitly the complex values associated with objects.

The reader with no previous exposure to object-oriented languages may now be utterly overwhelmed by the terminology. It might be helpful at this point to scan through a book or manual about an object-oriented programming language such as C++, or an OODB such

as O_2 or ObjectStore. This will provide numerous examples and the overall methodology of object-oriented programming, which is beyond the scope of this book.

21.2 Formal Definition of an OODB Model

This section presents a formal definition of a particular OODB model, called the *generic OODB model*. (This model is strongly influenced by the IQL and O_2 models. Many features are shared by most other OODB models. While presenting the model, we also discuss different choices made in other models.) The presentation essentially follows the preceding informal one, beginning with definitions for the types and class hierarchy and then introducing methods. It concludes with definitions of OODB schema and instance.

Types and Class Hierarchy

The formal definitions of object, type, and class hierarchy are intertwined. An object consists of a pair (*identifier*, *value*). The identifiers are taken from a specific sort containing OIDs. The values are essentially standard complex values, except that OIDs may occur within them. Although some of the definitions on complex values and types are almost identical to those in Chapter 20, we include them here to make precise the differences from the object-oriented context. As we shall see, the class hierarchy obeys a natural restriction based on subtyping.

To start, we assume a number of atomic types and their pairwise disjoint corresponding domains: **integer**, **string**, **bool**, **float**. The set **dom** of atomic values is the (disjoint) union of these domains; as before, the elements of **dom** are called *constants*. We also assume an infinite set **obj** $= \{o_1, o_2, \ldots\}$ of *object identifiers* (OIDs), a set **class** of class names, and a set **att** of *attribute names*. A special constant nil represents the undefined (i.e., null) value.

Given a set O of OIDs, the family of *values* over O is defined so that

(a) nil, each element of **dom**, and each element of O are values over O; and

(b) if $v_1, \ldots, v_n$ are values over O, and $A_1, \ldots, A_n$ distinct attributes names, the tuple $[A_1 : v_1, \ldots, A_n : v_n]$ and the set $\{v_1, \ldots, v_n\}$ are values over O.

The set of all values over O is denoted **val**(O). An *object* is a pair (o, v), where o is an OID and v a value.

In general, object-oriented database models also include constructors other than tuple and set, such as list and bag; we do not consider them here.

Example 21.2.1 Letting *oid*7, *oid*22, etc. denote OIDs, some examples of values are as follows:

[*theater* : *oid*7, *time* : "16:45", *price* : 45, *movie* : *oid*22]

{"H. Andersson", "K. Sylwan", "I. Thulin", "L. Ullman"}

[*title* : "The Trouble with Harry", *director* : *oid*77,
actors : {*oid*81, *oid*198, *oid*265, *oid*77}]

An example of an object is

$$(oid22\ ,\ [title : \text{“The Trouble with Harry”},\ director : oid77,\ actors : \{oid81, oid198, oid265, oid77\}])$$

As discussed earlier, objects are grouped in classes. All objects in a class have complex values of the same type. The type corresponding to each class is specified by the OODB schema.

Types are defined with respect to a given set C of class names. The family of *types* over C is defined so that

1. **integer, string, bool, float,** are types;
2. the class names in C are types;
3. if τ is a type, then[3] $\{\tau\}$ is a (set) type;
4. if $\tau_1, \ldots, \tau_n$ are types and $A_1, \ldots, A_n$ distinct attribute names, then $[A_1 : \tau_1, \ldots, A_n : \tau_n]$ is a (tuple) type.

The set of types over C together with the special class name **any** are denoted **types**(C). (The special name **any** is a type but may not occur inside another type.) Observe the close resemblance with types used in the complex value model.

EXAMPLE 21.2.2 An example of a type over the classes of the schema in Fig. 21.1 is

$$[name : \textbf{string},\ citizenship : \textbf{string},\ gender : \textbf{string}]$$

One may want to give a name to this type (e.g., *Person_type*). Other examples of types (with names associated to them) include

$$\begin{aligned}
Director_type &= [name : \textbf{string},\ citizenship : \textbf{string},\ gender : \textbf{string},\\
&\qquad directs : \{Movie\}]\\
Theater_type &= [name : \textbf{string},\ address : \textbf{string},\ phone : \textbf{string}]\\
Pariscope_type &= [theater : Theater,\ time : \textbf{string},\ price : \textbf{integer},\ movie : Movie]\\
Movie_type &= [title : \textbf{string},\ actors : \{Actor\},\ director : Director]\\
Award_type &= [prize : \textbf{string},\ year : \textbf{integer}]
\end{aligned}$$

In an OODB schema we associate with each class c a type $\sigma(c)$, which dictates the type of objects in this class. In particular, for each object (o, v) in class c, v must have the exact structure described by $\sigma(c)$.

---

[3] In Fig. 21.1 we use keywords **set** and **tuple** as syntactic sugar when specifying the set and tuple constructors.

Recall from the informal description that an OODB schema includes an ISA hierarchy among the classes of the schema. The class hierarchy has three components: (1) a set of classes, (2) the types associated with these classes, and (3) a specification of the ISA relationships between the classes. Formally, a *class hierarchy* is a triple $(C, \sigma, \prec)$, where C is a finite set of class names, σ a mapping from C to **types**(C), and $\prec$ a partial order on C.

Informally, in a class hierarchy the type associated with a subclass should be a refinement of the type associated with its superclass. For example, a class *Student* is expected to refine the information on its superclass *Person* by providing additional attributes. To capture this notion, we use a subtyping relationship ($\leq$) that specifies when one type refines another.

Definition 21.2.3 Let $(C, \sigma, \prec)$ be a class hierarchy. The *subtyping relationship* on **types**(C) is the smallest partial order $\leq$ over **types**(C) satisfying the following conditions:

(a) if $c \prec c'$, then $c \leq c'$;

(b) if $\tau_i \leq \tau_i'$ for each $i \in [1, n]$ and $n \leq m$, then
$[A_1 : \tau_1, \ldots, A_n : \tau_n, \ldots, A_m : \tau_m] \leq [A_1 : \tau_1', \ldots, A_n : \tau_n']$;

(c) if $\tau \leq \tau'$, then $\{\tau\} \leq \{\tau'\}$; and

(d) for each τ, $\tau \leq$ **any** (i.e., **any** is the top of the hierarchy).

A class hierarchy $(C, \sigma, \prec)$ is *well formed* if for each pair c, c' of classes, $c \prec c'$ implies $\sigma(c) \leq \sigma(c')$.

By way of illustration, it is easily verified that

$$\mathit{Director\_type} \leq \mathit{Person\_type} \qquad \mathit{Director\_type} \not\leq \mathit{Movie\_type}.$$

Thus the schema obtained by adding the constraint $\mathit{Director} \prec \mathit{Movie}$ would not be well formed.

Henceforth we consider only well-formed class hierarchies.

Example 21.2.4 Consider the class hierarchy $(C, \sigma, \prec)$ of the schema of Fig. 21.1. The set of classes is

$$C = \{\mathit{Person}, \mathit{Director}, \mathit{Actor}, \mathit{Actor\_Director}, \mathit{Theater}, \mathit{Movie}\}$$

with $\mathit{Actor} \prec \mathit{Person}$, $\mathit{Director} \prec \mathit{Person}$, $\mathit{Actor\_Director} \prec \mathit{Director}$, $\mathit{Actor\_Director} \prec \mathit{Actor}$, and (referring to Example 21.2.2 for the definitions of *Person_type*, *Theater_type*, etc.)

$$
\begin{aligned}
\sigma(\mathit{Person}) &= \mathit{Person\_type},\\
\sigma(\mathit{Theater}) &= \mathit{Theater\_type},\\
\sigma(\mathit{Movie}) &= \mathit{Movie\_type},\\
\sigma(\mathit{Director}) &= \mathit{Director\_type},\\
\sigma(\mathit{Actor}) &= [\mathit{name} : \mathbf{string},\ \mathit{citizenship} : \mathbf{string},\\
&\quad\ \mathit{gender} : \mathbf{string},\ \mathit{acts\_in} : \{\mathit{Movie}\},\\
&\quad\ \mathit{award} : \{\mathit{Award\_type}\}]\\
\sigma(\mathit{Actor\_Director}) &= [\mathit{name} : \mathbf{string},\ \mathit{citizenship} : \mathbf{string},\\
&\quad\ \mathit{gender} : \mathbf{string},\ \mathit{acts\_in} : \{\mathit{Movie}\},\\
&\quad\ \mathit{award} : \{\mathit{Award\_type}\},\ \mathit{directs} : \{\mathit{Movie}\}]
\end{aligned}
$$

The use of type names here is purely syntactic. We would obtain the same schema if we replaced, for instance, *Person_type* with the value of this type.

Observe that $\sigma(\mathit{Director}) \leq \sigma(\mathit{Person})$ and $\sigma(\mathit{Actor}) \leq \sigma(\mathit{Person})$, etc.

The Structural Semantics of a Class Hierarchy

We now describe how values can be associated with the classes and types of a class hierarchy. Because the values in an OODB instance may include OIDs, the semantics of classes and types must be defined simultaneously. The basis for these definitions is the notion of OID assignment, which assigns a set of OIDs to each class.

DEFINITION 21.2.5 Let $(C, \sigma, \prec)$ be a (well-formed) class hierarchy. An *OID assignment* is a function π mapping each name in C to a disjoint finite set of OIDs. Given OID assignment π, the *disjoint extension* of c is $\pi(c)$, and the *extension* of c, denoted $\pi^*(c)$, is $\cup\{\pi(c') \mid c' \in C, c' \prec c\}$.

If π is an OID assignment, then $\pi^*(c') \subseteq \pi^*(c)$ whenever $c' \prec c$. This should be understood as a formalization of the fact that an object of a subclass c' may be viewed also as an object of a superclass c of c'. From the perspective of typing, this suggests that operations that are type correct for members of c are also type correct for members of c'.

Unlike the case for many semantic data models, the definition of OID assignment for OODB schemas implies that extensions of classes of an ISA hierarchy without common subclasses are necessarily disjoint. In particular, extensions of all leaf classes of the hierarchy are disjoint (see Exercise 21.2). This is a simplifying assumption that makes it easier to associate objects to classes. There is a unique class to whose disjoint extension each object belongs.

The semantics for types is now defined relative to a class hierarchy $(C, \sigma, \prec)$ and an OID assignment π. Let $O = \cup\{\pi(c) \mid c \in C\}$, and define $\pi(\mathbf{any}) = O$. The *disjoint interpretation* of a type τ, denoted $\mathit{dom}(\tau)$, is given by

(a) for each atomic type τ, $\mathit{dom}(\tau)$ is the usual interpretation of that type;

(b) $\mathit{dom}(\mathbf{any})$ is $\mathbf{val}(O)$;

(c) for each $c \in C$, $dom(c) = \pi^*(c) \cup \{nil\}$;

(d) $dom(\{\tau\}) = \{\{v_1, \ldots, v_n\} \mid n \geq 0,$ and $v_i \in dom(\tau), i \in [1, n]\}$; and

(e) $dom([A_1 : \tau_1, \ldots, A_k : \tau_k]) = \{[A_1 : v_1, \ldots, A_k : v_k] \mid v_i \in dom(\tau_i), i \in [1, k]\}$.

Remark 21.2.6 In the preceding interpretation, the type determines precisely the structure of a value of that type. It is interesting to replace (e) by

(e′) $$dom([A_1 : \tau_1, \ldots, A_k : \tau_k]) = \{[A_1 : v_1, \ldots, A_k : v_k, A_{k+1} : v_{k+1}, \ldots A_l : v_l] \mid v_i \in dom(\tau_i), i \in [1, k], v_j \in \mathbf{val}(O), j \in [k + 1, l]\}.$$

Under this alternative interpretation, for each τ, τ' in $\mathbf{types}(C)$, if $\tau' \leq \tau$ then $dom(\tau') \subseteq dom(\tau)$. This is why this is sometimes called the *domain-inclusion semantics*. From a data model viewpoint, this presents the disadvantage that in a correctly typed database instance, a tuple may have a field that is not even mentioned in the database schema. For this reason, we do not adopt the domain-inclusion semantics here. On the other hand, from a linguistic viewpoint it may be useful to adopt this more liberal semantics in languages to allow variables denoting tuples with more attributes than necessary. ■

Adding Behavior

The final ingredient of the generic OODB model is *methods*. A method has three components:

(a) a *name*

(b) a *signature*

(c) an *implementation* (or *body*).

There is no problem in specifying the names and signatures of methods in an OODB schema. To specify the implementation of methods, a language for methods is needed. We do not consider specific languages in the generic OODB model. Therefore only names and signatures of methods are specified at the schema level in this model. In Section 21.4, we shall consider several languages for methods and shall therefore be able to add the implementation of methods to the schema.

Without specifying the implementation of methods, the generic OODB model specifies their semantics (i.e., the effect of each method in the context of a given instance). This effect, which is a function over the domains of the types corresponding to the signature of the method, is therefore specified at the instance level.

We assume the existence of an infinite set **meth** of method names. Let $(C, \sigma, \prec)$ be a class hierarchy. For method name m, a *signature* of m is an expression of the form $m : c \times \tau_1 \times \cdots \times \tau_{n-1} \to \tau_n$, where c is a class name in C and each τ_i is a type over C. This signature is associated with the class c; we say that method m *applies* to objects of class c and to objects of classes that inherit m from c. It is common for the same method name to have different signatures in connection with different classes. (Some restrictions shall be specified later.) The notion of signature here generalizes the one typically found in

object-oriented programming languages, because we permit the τ_i's to be types rather than only classes.

It is easiest to describe the notions of overloading, method inheritance, and dynamic binding in terms of an example. Consider the methods defined in the schema of Fig. 21.1. All three share the name *get_name*. The signatures are given by

$$get\_name : Person \rightarrow \textbf{string}$$
$$get\_name : Director \rightarrow \textbf{string}$$
$$get\_name : Actor\_Director \rightarrow \textbf{string}$$

Note that *get_name* has different implementations for these classes; this is an example of *overloading* of a method name.

Recall that *Actor* is a subclass of *Person*. According to the informal discussion, if *get_name* applies to elements of *Person*, then it should also apply to members of *Actor*. Indeed, in the object-oriented paradigm, if a method m is defined for a class c but not for a subclass c' of c (and it is not defined anywhere else along a path from c' to c), then the definition of m for c' is *inherited* from c. In particular, the signature of m on c' is identical to the one of m for c, except that the first c is replaced by c'. The implementation of m for c' is identical to that for c. In the schema of Fig. 21.1, the signature of *get_name* for *Actor* is

$$get\_name : Actor \rightarrow \textbf{string}$$

and the implementation is identical to the one for *Person*. The determination of the correct method implementation to use for a given method name m and class c is called *method resolution*; the selected implementation is called the *resolution* of m for c.

Suppose that π is an OID assignment, that $oid25$ is in the extension $\pi^*(Person)$ of *Person*, and that *get_name* is called on $oid25$. What implementation of *get_name* will be used? In our OODB model we shall use *dynamic binding* (also called *late binding*, or *value-dependent binding*). This means that the specific implementation chosen for *get_name* on $oid25$ depends on the most specific class that $oid25$ belongs to, that is, the class c such that $oid25 \in \pi(c)$.

(An alternative to dynamic binding is *static binding*, or *context-dependent binding*. Under this discipline, the implementation used for *get_name* depends on the type associated with the variable holding $oid25$ at the point in program where *get_name* is invoked. This can be determined at compile time, and so static binding is generally much cheaper than dynamic binding. In the language C++, the default is static binding, but dynamic binding can be obtained by using the keyword **virtual** when specifying the method.)

Consider a call $m(o, v_1, \ldots, v_{n-1})$ to method m. This is often termed a *message*, and o is termed the *receiver*. As described here, the implementation of m associated with this message depends exclusively on the class of o. To emphasize the importance of the receiver for finding the actual implementation, in some languages the message is denoted $o \rightarrow m[v_1, \ldots v_{n-1}]$. In some object-oriented programming languages, such as CommonLoops (an object-oriented extension of LISP), the implementation depends on

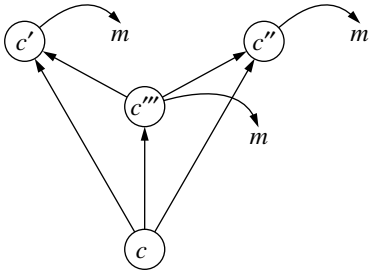


Figure 21.2: Unambiguous definition

all of the parameters of the call, not just the first. This is also the approach of the method schemas introduced in Section 21.4.

The set of methods applicable to an object is called the *interface* of the object. As noted in the informal description of OODB models, in most cases objects are accessed only via their interface; this philosophy is called *encapsulation*.

As part of an OODB schema, a set M of method signatures is associated to a class hierarchy $(C, \sigma, \prec)$. Note that a signature $m : c \times \tau_1 \times \cdots \times \tau_{n-1} \to \tau_n$ can be viewed as giving a particular meaning to m for class c, at least at a syntactic level. Because of inheritance, a meaning for method m need not be given explicitly for each class of C nor even for subclasses of a class for which m has been given a meaning. However, we make two restrictions on the family of method signatures: The set M is *well formed* if it obeys the following two rules:

Unambiguity: If c is a subclass of c' and c'' and there is a definition of m for c' and c'', then there is a definition of m for a subclass of c' and c'' that is either c itself, or a superclass of c. (See Fig. 21.2.)

Covariance[4]*:* If $m : c \times \tau_1 \times \cdots \times \tau_n \to \tau$ and $m : c' \times \tau'_1 \times \cdots \times \tau'_m \to \tau'$ are two definitions and $c \prec c'$, then $n = m$ for each i, $\tau_i \leq \tau'_i$ and $\tau \leq \tau'$.

The first rule prevents ambiguity resulting from the presence of two method implementations both applicable for the same object. A primary motivation for the second rule is intuitive: We expect the argument and result types of a method on a subclass to be more refined than those of the method on a superclass. This also simplifies the writing of type-correct programs, although type checking leads to difficulties even in the presence of the covariance assumption (see Section 21.4).

Database Schemas and Instances

We conclude this section by presenting the definitions of schemas and instances in the generic OODB model. An important subtlety here will be the role of OIDs in instances

[4] In type theory, *contravariance* is used instead. Contravariance is the proper notion when functions are passed as arguments, which is not the case here.

as placeholders; as will be seen, the specific OIDs present in an instance are essentially irrelevant.

As indicated earlier, a schema describes the structure of the data that is stored in a database, including the types associated with classes and the ISA hierarchy and the signature of methods (i.e., the interfaces provided for objects in each class).

In many practical OODBs, it has been found convenient to allow storage of complex values that are not associated with any objects and that can be accessed directly using some name. This also allows us to subsume gracefully the capabilities of value-based models, such as relations and complex values. It also facilitates writing queries. To reflect this feature, we allow a similar mechanism in schemas and instances. Thus schemas may include a set of value names with associated types. Instances assign values of appropriate type to the names. Method implementations, external programming languages, and query languages may all use these names (to refer to their current values) or a class name (to refer to the set of objects currently residing in that class). In this manner, named values and class names are analogous to relation names in the relational model and to complex value relation names in the complex value model.

In the schema of Fig. 21.1, examples of named values are *Pariscope* (holding a set of triples); *Persons_I_like*, *Actors_I_like*, and *Actors_you_like* (referring to sets of person objects and actor objects; and, finally, *My_favorite_director* (referring to an individual object as opposed to a set). These names can be used explicitly in method implementations and in external query and programming languages.

We now have the following:

DEFINITION 21.2.7 A *schema* is a 5-tuple $\mathbf{S} = (C, \sigma, \prec, M, G)$ where

- G is a set of *names* disjoint from C;
- σ is a mapping from $C \cup G$ to $\mathbf{types}(C)$;
- $(C, \sigma, \prec)$ is a well-formed class hierarchy[5]; and
- M is a well-formed set of method signatures for $(C, \sigma, \prec)$.

An instance of an OODB schema populates the classes with OIDs, assigns values to these OIDs, gives meaning to the other persistent names, and assigns semantics to method signatures. The semantics of method signatures are mappings describing their effect. From a practical viewpoint, the population of the classes, the values of objects, and the values of names are kept extensionally; whereas the semantics of the methods are specified by pieces of code (intensionally). However, we ignore the code of methods for the time being.

DEFINITION 21.2.8 An *instance* of schema $(C, \sigma, \prec, M, G)$ is a 4-tuple $\mathbf{I} = (\pi, \nu, \gamma, \mu)$, where

(a) π is an OID assignment (and let $O = \cup\{\pi(c) \mid c \in C\}$);

(b) ν maps each OID in O to a value in $\mathbf{val}(O)$ of correct type [i.e., for each c and $o \in \pi(c)$, $\nu(o) \in dom(\sigma(c))$];

[5] By abuse of notation, we use here and later σ instead of $\sigma|_C$.

(c) γ associates to each name in G of type τ a value in $dom(\tau)$;

(d) μ assigns semantics to method names in agreement with the method signatures in M. More specifically, for each signature $m : c \times \vec{\alpha} \to \tau$,

$$\mu(m : c \times \vec{\alpha} \to \tau) : dom(c \times \vec{\alpha}) \to dom(\tau);$$

that is, $\mu(m : c \times \vec{\alpha} \to \tau)$ is a partial function from $dom(c \times \vec{\alpha})$ to $dom(\tau)$.

Recall that a method m can occur with different signatures in the same schema. The mapping μ can assign different semantics to each signature of m. The function $\mu(m : c \times \vec{\alpha} \to \tau)$ is only relevant on objects associated with c and subclasses of c for which m is not redefined.

In the preceding definitions, the assignment of semantics to method signatures is included in the instance. As will be seen in Section 21.4, if method implementations are included in the schema, they induce the semantics of methods at the instance level (this is determined by the semantics of the particular programming language used in the implementation).

Intuitively, it is generally assumed that elements of the atomic domains have universally understood meaning. In contrast, the actual OIDs used in an instance are not relevant. They serve essentially as placeholders; it is only their relationship with other OIDs and constants that matters. This arises in the practical perspective in two ways. First, in most practical systems, OIDs cannot be explicitly created, examined, or manipulated. Second, in some object-oriented systems, the actual OIDs used in a physical instance may change over the course of time (e.g., as a result of garbage collection or reclustering of objects).

To capture this aspect of OIDs in the formal model, we introduce the notion of OID isomorphism. Two instances $\mathbf{I}$, $\mathbf{J}$ are *OID isomorphic*, denoted $\mathbf{I} \equiv_{OID} \mathbf{J}$, if there exists a bijection on $\mathbf{dom} \cup \mathbf{obj}$ that maps $\mathbf{obj}$ to $\mathbf{obj}$, is the identity on $\mathbf{dom}$, and transforms $\mathbf{I}$ into $\mathbf{J}$. To be precise, the term *object-oriented instance* should refer to an equivalence class under OID isomorphism of instances as defined earlier. However, it is usually more convenient to work with representatives of these equivalence classes, so we follow that convention here.

Remark 21.2.9 In the model just described, a class encompasses two aspects:

1. at the schema level, the class definition (its type and method signatures); and
2. at the instance level, the class extension (the set of objects currently in the class).

It has been argued that one should not associate explicit class extensions with classes. To see the disadvantage of class extensions, consider object deletion. To be removed from the database, an object has to be deleted *explicitly* from its class extension. This is not convenient in some cases. For instance, suppose that the database contains a class *Polygon* and polygons are used only in figures. When a polygon is no longer used in any figure of the current database, it is no longer of interest and should be deleted. We would like this deletion to be implicit. (Otherwise the user of the database would have to search all possible places in which a reference to a polygon may occur to be able to delete a polygon.)

To capture this, some OODBs use an integrity constraint, which states that

every object should be accessible from some named value.

This integrity constraint is enforced by an automatic deletion of all objects that become unreachable from the named values. In the polygon example, this approach would allow defining the class *Polygon*, thus specifying the structure and methods proper to polygons. However, the members of class *Polygon* would only be those polygons that are currently relevant. Relevance is determined by membership in (or accessibility from) the named values (e.g., *My-Figures, Your-Figures*) that refer to polygons. From a technical viewpoint, this involves techniques such as garbage collection.

In these OODBs, the set of objects in a class is not directly accessible. For this reason, the corresponding models are sometimes called models without class extension. Of course, it is always possible, given a schema, to compute the class extensions or to adapt object creation in a given class to maintain explicitly a named value containing that class extension. In these OODBs, the named values are also said to be roots of persistence, because the persistence of an object is dependent on its accessibility from these named values. ■

21.3 Languages for OODB Queries

This section briefly introduces several languages for querying OODBs. These queries are formulated against the database as a whole; unlike methods, they are not associated with specific classes. In the next section, we will consider languages intended to provide implementations for methods.

In describing the OODB query languages, we emphasize how OODB features are incorporated into them. The first language is an extension of the calculus for complex values, which incorporates such object-oriented components as OIDs, different notions of equality, and method calls. The second is an extension of the *while* language, initially introduced in Chapter 14. Of primary interest here is the introduction of techniques for creating new OIDs as part of a query. At this point we examine the notion of completeness for OODB access languages. We also briefly look at a language introducing a logic-based approach to object creation. Finally, we mention a practical language, O_2SQL. This is a variant of SQL for OODBs that provides elegant object-oriented features.

Although the languages discussed in this section do provide the ability to call methods and incorporate the results into the query processing and answer, we focus primarily on access to the extensional structural portion of the OODB. The intensional portion, provided by the methods, is considered in the following section. Also, we largely ignore the important issue of typing for queries and programs written in these languages. The issue of typing is considered, in the context of languages for methods, in the next section.

An Object-Oriented Calculus

The object-oriented calculus presented here is a straightforward generalization of the complex value calculus of Chapter 20, extended to incorporate objects, different notions of equality, and methods.

Let $(C, \sigma, \prec, M, G)$ be an OODB schema, and let us ignore the object-oriented features for a moment. Each name in G can be viewed as a complex value; it is straightforward to generalize the complex value calculus to operate on the values referred to by G. (The fact that in the complex value model all relations are sets whereas some names in G might refer to nonset values requires only a minor modification of the language.)

Let us now consider objects. OIDs may be viewed as elements of a specific sort. If viewed in isolation from their associated values, this suggests that the only primitive available for comparing OIDs is equality. Recall from the schema of Fig. 21.1 the names *Actors_I_like* and *Actors_you_like*. The query[6]

(21.1) $\exists x, y(x \in Actors\_I\_like \wedge y \in Actors\_you\_like \wedge x = y)$

asks whether there is an actor we both like. To obtain the names of such actors, we need to introduce dereferencing, a mechanism to obtain the value of an object. Dereferencing is denoted by $\uparrow$. The following query yields the names of actors we both like:

(21.2) $\{y \mid \exists x(x \in Actors\_I\_like \wedge x \in Actors\_you\_like \wedge x \uparrow .name = y)\}$

In the previous query, $x \uparrow$ denotes the value of x, in this case, a tuple with four fields. The dot notation (.) is used as before to obtain the value of specific fields.

In query (21.1), we tested two objects for equality, essentially testing whether they had the same OID. Although it does not increase the expressive power of the language, it is customary to introduce an alternative test for equality, called *value equality*. This tests whether the values of two objects are equal regardless of whether their OIDs are distinct. To illustrate, consider the three objects having *Actor_type*:

(*oid*50, [*name* : "Martin", *citizenship* : "French", *gender* : "male",
award : { }, *acts_in* : {*oid*33}])
(*oid*51, [*name* : "Martin", *citizenship* : "French", *gender* : "male",
award : { }, *acts_in* : {*oid*33}])
(*oid*52, [*name* : "Martin", *citizenship* : "French", *gender* : "male",
award : { }, *acts_in* : {*oid*34}])

Then *oid*50 and *oid*51 are value equal, whereas *oid*50 and *oid*52 are not. Yet another form of equality is *deep equality*. If *oid*33 and *oid*34 are value equal, then *oid*50 and *oid*52 are deep equal. Intuitively, two objects are deep equal if the (possibly infinite) trees obtained by recursively replacing each object by its value are equal. The infinite trees that we obtain are called the *expansions*. They present some regularity; they are regular trees (see Exercise 21.10).

The notion of deep equality highlights a major difference between value-based and object-based models. In a value-based model (such as the relational or complex value

[6] In this example, if *name* is a key for *Actor*, then one can easily obtain an equivalent query not using object equality; this may not be possible if there is no key for *Actor*.

models), the database can be thought of as a collection of (finite) trees. The connections between trees arise as a result of the contents of atomic fields. That is, they are implicit (e.g., the same string may appear twice). In the object-oriented world, a database instance can be thought of as graph. Paths in the database are more explicit. That is, one may view an (*oid*, *value*) pair as a form of logical pointer and a path as a sequence of pointer dereferencing.

This graph-based perspective leads naturally to a navigational form of data access (e.g., using a sequence such as $o \uparrow .director \uparrow .citizenship$ to find the citizenship of the director of a given movie object o). This has led some to view object-oriented models as less declarative than value-based models such as the relational model. This is inaccurate, because declarativeness is more a property of access languages than models. Indeed, the calculus for OODBs described here illustrates that a highly declarative language can be developed for the OODB model.

We conclude the discussion of the object-oriented calculus by incorporating methods. For this discussion, it is irrelevant how the methods are specified or evaluated; this evaluation is external to the query. The query simply uses the method invocations as oracles. Method resolution uses dynamic binding. The value of an expression of the form $m(t_1, \ldots, t_n)$ under a given variable assignment ν is obtained by evaluating (externally) the implementation of m for the class of $\nu(t_1)$ on input $\nu(t_1, \ldots, t_n)$. In this context, it is assumed that m has no side-effects. Although not defined formally here, the following illustrates the incorporation of methods into the calculus:

(21.3) $\{y \mid \exists x(x \in Persons\_I\_like \wedge y = get\_name(x))\}$

If the set *Persons_I_like* contains Bergman and Liv Ullman, the answer would be

{“Ms. Ullman”, “Liv Ullman”}

The use of method names within the calculus raises a number of interesting typing and safety issues that will not be addressed here.

Object Creation and Completeness

Relational queries take relational instances as input and produce relational instances as output. The preceding calculus fails to provide the analogous capability because the output of a calculus query is a set of values or objects. Two features are needed for a query language to produce the full-fledged structural portion of an object-oriented instance: the ability to create OIDs, and the ability to populate a family of named values (rather than producing a single set).

We first introduce an extension of the *while* language of Chapter 14 that incorporates both of these capabilities. This language leads naturally to a discussion of completeness of OODB access languages. After this we mention a second approach to object creation that stems from the perspective of logic programming.

The extension of *while* introduced here is denoted $while_{obj}$. It will create new OIDs in a manner reminiscent of how the language $while_{new}$ of Chapter 18 invented new constants.

The language $while_{obj}$ incorporates object-oriented features such as dereferencing and method calls, as in the calculus. To illustrate, we present a $while_{obj}$ program that collects all actors reachable from an actor I like—Liv Ullman. In this query, *v_movies* and *v_directors* serve as variables, and *reachable* serves as a new name that will hold the output.

reachable := $\{x \mid x \in Actors\_I\_like \wedge x \uparrow .name =$ "Liv Ullman"$\}$;
v_movies := { }; *v_directors* := { };
while change do
  begin
  reachable := *reachable* $\cup \{x \mid \exists y(y \in v\_movies \wedge x \in y \uparrow .actors)\}$;
  v_directors := *v_directors*
    $\cup \{x \mid \exists y(y \in v\_movies \wedge x \in y \uparrow .director)\}$;
  v_movies := *v_movies*
    $\cup\{x \mid \exists y(y \in reachable \wedge x \in y \uparrow .acts\_in)\}$
    $\cup\{x \mid \exists y(y \in v\_directors \wedge x \in y \uparrow .directs)\}$;
  end;

We now introduce object creation. The operator **new** works as follows. It takes as input a set of values (or objects) and produces one new OID for each value in the set. As a simple example, suppose that we want to objectify the quadruples in the named value *Pariscope* of the schema of Fig. 21.1. This may be accomplished with the commands

add_class Pariscope_obj
  type tuple (*theater* : *Theater*, *time* : **string**, *price* : **integer**, *movie* : *Movie*);
Pariscope_obj := **new**(*Pariscope*)

Of course, the **new** operator can be used in conjunction with arbitrary expressions that yield a set of values, not just a named value.

The **new** operator used here is closely related to the *new* operator of the language $while_{new}$ of Chapter 18. Given that $while_{obj}$ has iteration and the ability to create new OIDs, it is natural to ask about the expressive power of this language. To set the stage, we introduce the following analogue of the notion of (computable) query, which mimics the one of Chapter 18. The definition focuses on the structural portion of the OODB model; methods are excluded from consideration.

Definition 21.3.1 Let **R** and **S** be two OODB schemas with no method signatures. A *determinate query* is a relation Q from $inst(\mathbf{R})$ to $inst(\mathbf{S})$ such that

(a) Q is computable;
(b) (Genericity) if $\langle \mathbf{I}, \mathbf{J} \rangle \in Q$ and ρ is a one-to-one mapping on constants, then $\langle \rho(\mathbf{I}), \rho(\mathbf{J}) \rangle \in Q$;
(c) (Functionality) if $\langle \mathbf{I}, \mathbf{J} \rangle \in Q$, and $\langle \mathbf{I}, \mathbf{J}' \rangle \in Q$, then $\mathbf{J}$ and $\mathbf{J}'$ are OID isomorphic; and
(d) (Well defined) if $\langle \mathbf{I}, \mathbf{J} \rangle \in Q$ and $\langle \mathbf{I}', \mathbf{J}' \rangle$ is OID isomorphic to $\langle \mathbf{I}, \mathbf{J} \rangle$, then $\langle \mathbf{I}', \mathbf{J}' \rangle \in Q$.

A language is *determinate complete* (for OODBs) if it expresses exactly the determinate queries.

The essential difference between the preceding definition and the definition of determinate query in Chapter 18 is that here only OIDs can be created, not constants. Parts (c) and (d) of the definition ensure that a determinate query Q can be viewed as a *function* from OID equivalence classes of instances over $\mathbf{R}$ to OID equivalence classes of instances over $\mathbf{S}$. So OIDs serve two purposes here: (1) They are used to compute in the same way that invented values were used to break the polynomial space barrier; and (2) they are now essential components of the data structure and in particular of the result. With respect to (2), an important aspect is that we are not concerned with the actual value of the OIDs, which motivates the use of the equivalence relation. (Two results are viewed as identical if they are the same up to the renaming of the OIDs.)

Like $while_{new}$, $while_{obj}$ is not determinate complete. There is an elegant characterization of the determinate queries expressible in $while_{obj}$. This result, which we state next, uses a *local* characterization of input-output pairs of $while_{obj}$ programs. That characterization is in the spirit of the notion of BP-completeness, relating input-output pairs of relational calculus queries (see Exercise 16.11). For each input-output pair $\langle I, J\rangle$, the characterization of $while_{obj}$ queries requires a simple connection between the automorphism group of I and that of J. For an instance K, let $Aut(K)$ denote the set of automorphisms of K. For a pair of instances K, K', $Aut(\langle K, K'\rangle)$ denotes the bijections on $\mathbf{adom}(K \cup K')$ that are automorphisms of both K and K'.

THEOREM 21.3.2 A determinate query q is expressible in $while_{obj}$ iff for each input-output pair $\langle I, J\rangle$ in q there exists a mapping h from $Aut(I)$ to $Aut(\langle I, J\rangle)$ such that for each $\tau, \mu \in Aut(I)$,

(i) τ and $h(\tau)$ coincide on I;

(ii) $h(\tau \circ \mu) = h(\tau) \circ h(\mu)$; and

(iii) $h(\mathbf{id}_I) = \mathbf{id}_{\langle I,J\rangle}$.

The "only if" part of the theorem is proven by an extension of the trace technique developed in the proof of Theorem 18.2.5 (Exercise 21.14). The "if" part is considerably more complex and is based on a group-theoretic argument.

A mapping h just shown is called an *extension homomorphism* from $Aut(I)$ to $Aut(\langle I, J\rangle)$. To see an example of the usefulness of this characterization, consider the query q in Fig. 21.3. Recall that q was shown as not expressible in the language $while_{new}$ by Theorem 18.2.5. The language $while_{obj}$ is more powerful than $while_{new}$, so in principle it may be able to express that query. However, we show that this is not the case, so $while_{obj}$ is not determinate complete.

PROPOSITION 21.3.3 Query q (of Fig. 21.3) is not expressible in $while_{obj}$.

Proof Let $\langle I, J\rangle$ be the input-output pair of Fig. 21.3. The proof is by contradiction. Suppose there is a $while_{obj}$ query that produces J on input I. By Theorem 21.3.2, there is an extension homomorphism h from $Aut(I)$ to $Aut(\langle I, J\rangle)$. Let μ be the automorphism of I exchanging a and b. Note that $\mu^{-1} = \mu$, so $\mu \circ \mu = \mathbf{id}_I$. Consider $h(\mu)(\psi_0)$. Clearly, $h(\mu)(\psi_0) \in \{\psi_1, \psi_3\}$. Suppose $h(\mu)(\psi_0) = \psi_1$ (the other case is similar). Then clearly,

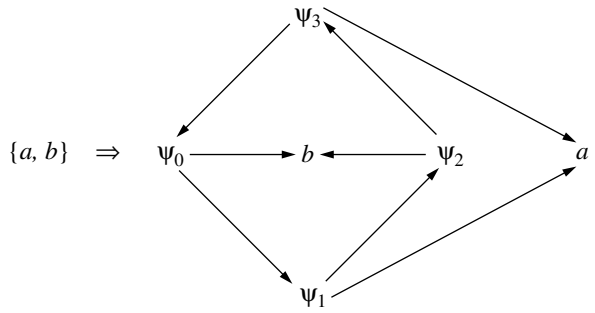


Figure 21.3: A query not expressible in $while_{obj}$

$h(\mu)(\psi_1) = \psi_2$. Consider now $h(\mu \circ \mu)(\psi_0)$. We have, on one hand,

$$\begin{aligned} h(\mu \circ \mu)(\psi_0) &= (h(\mu) \circ h(\mu))(\psi_0) \\ &= h(\mu)(\psi_1) \\ &= \psi_2 \end{aligned}$$

and on the other hand

$$\begin{aligned} h(\mu \circ \mu)(\psi_0) &= h(\mathbf{id}_I)(\psi_0) \\ &= \mathbf{id}_{\langle I,J\rangle}(\psi_0) \\ &= \psi_0, \end{aligned}$$

which is a contradiction because $\psi_0 \neq \psi_2$. So q is not expressible in $while_{obj}$. ■

It is possible to obtain a language expressing all determinate queries by adding to $while_{obj}$ a *choose* operator that allows the selection (nondeterministically but in a determinate manner) of one object out of a set of objects that are isomorphic (see Exercise 18.14). However, this is a highly complex construct because it requires the ability to check for isomorphism of graphs. The search for simpler, local constructs that yield a determinate-complete language is an active area of research.

A Logic-Based Approach to Object Creation

We now briefly introduce an alternative approach for creating OIDs that stems from the perspective of datalog and logic programming. Suppose that a new OID is to be created for each pair $\langle t, m\rangle$, where movie m is playing at theater t according to the current value of *Pariscope*. Consider the following dataloglike rule:

1. $create\_tm\_object(x, t, m) \leftarrow Pariscope(t, s, m)$

Note that x occurs in the rule head but not in the body, so the rule is not safe. Intuitively, we would like to attach semantics to this rule so that a new OID is associated to x for each

distinct pair of (t, m) values. Using the symbol $\exists!$ to mean "exists a unique," the following versions of (1) intuitively captures the semantics.

2. $\forall t \forall m \exists! x \forall s [create\_tm\_object(x, t, m) \leftarrow Pariscope(t, s, m)]$
3. $\forall t \forall m \exists! x [create\_tm\_object(x, t, m) \leftarrow \exists s (Pariscope(t, s, m))]$

This suggests that Skolem functions might be used. Specifically, let f_{tm} be a function symbol associated with the predicate *create_tm_object*. We rewrite (2) as

$$\forall t \forall m \forall s [create\_tm\_object(f_{tm}(t, m), t, m) \leftarrow Pariscope(t, s, m)]$$

or, leaving off the universal quantifiers as traditional in datalog,

4. $create\_tm\_object(f_{tm}(t, m), t, m) \leftarrow Pariscope(t, s, m)$

Under this approach, the Skolem terms resulting from rule (4) are to be interpreted as new, distinct OIDs. Under some formulations of the approach, syntactic objects such as $f_{tm}(oid7, oid22)$ (where $oid7$ is the OID of some theater and $oid22$ the OID of some movie) serve explicitly as OIDs. Under other formulations, such syntactic objects are viewed as placeholders during an intermediate stage of query evaluation and are (nondeterministically) replaced by distinct new OIDs in the final stage of query evaluation (see Exercise 21.13).

The latter approach to OID creation, incorporated into complex value datalog extended to include also OID dereferencing, yields a language equivalent to $while_{obj}$. As with $while_{obj}$, this language is not determinate complete.

A Practical Language for OODBs

We briefly illustrate some object-oriented features of the language O_2SQL, which was introduced in Section 20.8. Several examples are presented there, that show how O_2SQL can be used to access and construct deeply nested complex values. We now indicate how the use of objects and methods is incorporated into the language. It is interesting to note that methods and nested complex values are elegantly combined in this language, which has the appearance of SQL but is essentially based on the functional programming paradigm.

For this example, we again assume the complex value *Films* of Fig. 20.2, but we assume that *Age* is a method defined for the class *Person* (and thus for *Director*).

```
select  tuple (f.Director, f.Director.Age)
from    f in Films
where f.Director not in flatten select m.Actors
                                from  g in Films,
                                      m in g.Movies
                                where g.Director = "Hitchcock"
```

(Recall that here the inner *select-from-where* clause returns a set of sets of actors. The keyword **flatten** has the effect of forming the union of these sets to yield a set of actors.)

21.4 Languages for Methods

So far, we have used an abstraction of methods (their signature) and ignored their implementations. In this section, we present two abstract programming languages for specifying method implementations. Method implementations will be included in the specification of methods in OODB schemas. In studying these languages, we emphasize two important issues: type safety and expressive power. This focus largely motivates our choice of languages and the particular abstractions considered.

The first language is an imperative programming language. The second, method schemas, is representative of a functional style of database access. In the first language, we will gather a number of features present in practical object-oriented database languages (e.g., side-effect, iteration, conditionals). We will see that with these features, we get (as could be expected) completeness, and we pay the obvious price for it: the undecidability of many questions, such as type safety. With method schemas, we focus on the essence of inheritance and methods. We voluntarily consider a limited language. We see that the undecidability of type safety is a consequence of recursion in method calls. (We obtain decidability in the restricted case of monadic methods.) With respect to expressiveness, we present a surprising characterization of QPTIME in terms of a simple language with methods.

For both languages, we study type safety and expressive power. We begin by discussing briefly the meaning of these notions in our context, and then we present the two languages and the results.

An OODB schema **S** (with method implementations assigned to signatures) is type safe if for each instance **I** of **S** and each syntactically correct method call on **I**, the execution of this method does not result in a runtime type error (an illegal method call). When the imperative programming language is used in method implementations, type safety is undecidable. (It is possible, however, to obtain decidable sufficient conditions for type safety.) For method schemas, type safety remains undecidable. Surprisingly, type safety is decidable for monadic method schemas.

To evaluate the expressive power of OODB schemas using a particular language for method implementation, a common approach is to simulate relational queries and then ask what family of relational queries can be simulated. If OID creation is permitted, then all computable relational queries can be simulated using the imperative language. The expressive power of imperative methods without OID creation depends on the complex types permitted in OODB schemas. We also present a result for the expressive power of method schemas, showing that the family of method schemas using an ordered domain of atomic elements expresses exactly QPTIME.

A Model with Imperative Methods

To consider the issue of type safety in a general context, we present the *imperative (OODB) model*, which incorporates imperative method implementations. This model simplifies the OODB model presented earlier by assuming that the type of each class is a tuple of values and OIDs. However, a schema in this model will include an assignment of implementations to method signatures.

The syntax for method implementations is

par: $u_1, \ldots, u_n$;
var: $x_1, \ldots, x_l$;
body: $s_1; \ldots; s_q$;
return x_1

where the u_i's are parameters ($n \geq 1$), the x_j's are internal variables ($l \geq 1$), and for each $p \in [1, q]$, s_p is a statement of one of the following forms (where w, y, z range over parameters and internal variables):

Basic operations

(i) $w :=$ *self*.

(ii) $w :=$ *self.a* for some field name a.

(iii) $w := y$.

(iv) $w := m(y, \ldots, z)$, for some method name m.

(v) *self.a* $:= w$, for some field name a.

Class operations

(vi) $w :=$ **new**(c), where c is a class.

(vii) **delete**(c, w), where c is a class.

(viii) **for each** w **in** c **do** $s'_1; \ldots; s'_t$ **end**, where c is a class and $s'_1, \ldots, s'_t$ are statements having forms from this list.

Conditional

(ix) **if** $y \theta z$ **then** s, where θ is $=$ or $\neq$ and s is a statement having a form in this list except for the conditional.

It is assumed that all internal variables are initialized before used to some default value depending on their type. The intended semantics for the forms other than (viii) should be clear. (Here *clear* does not mean "easy to implement." In particular, object deletion is complex because all references to this object have to be deleted.) The looping construct executes for each element of the extension (not disjoint extension) of class c. The execution of the loop is viewed as nondeterministic, in the sense that the particular ordering used for the elements of c is not guaranteed by the implementation. In general, we focus on OODB schemas in which different orders of execution of the loops yield OID-equivalent results (note, however, that this property is undecidable, so it must be ensured by the programmer).

An imperative *schema* is a 6-tuple $\mathbf{S} = (C, \sigma, \prec, M, G, \mu)$, where $(C, \sigma, \prec, M, G)$ is a schema as before; where the range of σ is tuples of atomic and class types; and where μ is an assignment of implementations to signatures. The notion of *instance* for this model is defined in the natural fashion.

It is straightforward to develop operational semantics for this model, where the execution of a given method call might be *successful*, *nonterminating*, or *aborted* (as the result of a runtime type error) (Exercise 21.15a).

Type Safety in the Imperative Model There are two ways that a runtime type error can arise: (1) if the type of the result of an execution of method m does not lie within the type specified by the relevant method signature of m; or (2) if a method is called on a tuple of parameters that does not satisfy the domain part of the appropriate signature of m. We assume that the range of all method signatures is **any**, and thus we focus on case (2).

A schema **S** is *type safe* if for each instance over **S** and each $m(o, v_1, \ldots, v_n)$ method call that satisfies the signature of m associated with the class of o, execution of this call is either successful or nonterminating.

Given a Turing machine M, it is easy to develop a schema S in this model that can simulate the operation of M on a suitable encoding of an input tape (Exercise 21.15c). This shows that such schemas are computationally powerful and implies the usual undecidability results. With regard to type safety, it is easy to verify the following (Exercise 21.16):

Proposition 21.4.1 It is undecidable, given an imperative schema **S**, whether **S** is type safe. This remains true, even if in method implementations conditional statements and the **new** operator are prohibited and all methods are monadic (i.e., have only one argument).

A similar argument can be used to show that it is undecidable whether a given method terminates on all inputs. Finally, a method m' on class c' is *reachable* from method m on class c in OODB schema **S** if there is some instance **I** of **S** and some tuple $o, v_1, \ldots$ with o in c such that the execution of $m(o, v_1, \ldots)$ leads to a call of m' on some object in c'. Reachability is also undecidable for imperative schemas.

Expressive Power of the Imperative Model

As discussed earlier, we measure the expressive power of OODB schemas in terms of the relational queries they can simulate. A relational schema $\mathbf{R} = \{R_1, \ldots, R_n\}$ is *simulated by* an OODB schema **S** of this model if there are leaf classes $c_1, \ldots, c_n$ in **S**, where the number of attributes of c_i is the arity of R_i for $i \in [1, n]$ and where the type of each of these attributes is atomic. We focus on instances in which no null values appear for such attributes. Let **R** be a relational schema and **S** be an OODB schema that simulates **R**. An instance **I** of **R** is *simulated* by instance **J** of **S** if for each tuple $\vec{v} \in \mathbf{I}(R_i)$ there is exactly one object o in the extension of c_i such that the value associated with o is $\vec{v}$ and all other classes of **S** are empty. Following this spirit, it is straightforward to define what it means for a method call in schema **S** to simulate a relational query from **R** to relation schema R.

We consider only schema **S** for which different orders of evaluation of the looping construct yield the same final result (i.e., generic mappings). We now have the following (see Exercise 21.20):

Theorem 21.4.2 The family of generic queries corresponding to imperative schemas coincides with the family of all relational queries.

The preceding result relies on the presence of the **new** operator. It is natural to ask about the expressive power of imperative schemas that do not support **new**. As discussed in Exercise 21.21, the expressive power depends on the complex types permitted for objects.

Note also that imperative schemas can express all determinate queries. This uses the nondeterminism of the **for each** construct. Naturally, nondeterministic queries that are not determinate can also be expressed.

Method Schemas

We now present an abstract model for side-effect-free methods, called method schemas. In this model, we focus almost exclusively on methods and their implementations. Two kinds of methods are distinguished: base and composite. The base methods do not have implementations: Their semantics is specified explicitly at the instance level. The implementations of composite methods consist of a composition of other methods.

We now introduce method schemas. In the next definition, we make the simplifying assumption that there are no named values (only class names) in database schemas. In fact, data is only stored in base methods. In the following, $\sigma_{[\ ]}$ denotes the type assignment $\sigma_{[\ ]}(c) = [\ ]$ for every class c. Because the type assignment provides no information in method schemas (it is always $\sigma_{[\ ]}$), this assignment is not explicitly specified in the schemas.

Definition 21.4.3 A *method schema* is a 5-tuple $\mathbf{S} = (C, \prec, M_{base}, M_{comp}, \mu)$, where $(C, \sigma_{[\ ]}, \prec)$ is a well-formed class hierarchy, $M_{base} \cup M_{comp}$ is a well-formed set of method signatures for $(C, \sigma_{[\ ]}, \prec)$, and

- no method name occurs in both M_{base} and M_{comp};
- each method signature in M_{comp} is of the form $m : c_1, \ldots, c_n \rightarrow \mathbf{any}$ (method signatures for M_{base} are unrestricted, i.e., can have any class as range);
- μ is an assignment of implementations to the method signatures of M_{comp}, as follows: For a signature $m : c_1, \ldots, c_n \rightarrow \mathbf{any}$ in M_{comp}, $\mu(m : c_1, \ldots, c_n \rightarrow \mathbf{any})$ is a term obtained by composing methods in M_{base} and M_{comp}.

An example of an implementation for a method $m : c_1, c_2 \rightarrow \mathbf{any}$ is

$$m(x, y) \equiv m_1(m_2(x), m_1(x, y)).$$

The semantics of methods is defined in the obvious way. For instance, to compute $m(o, o')$, one computes first $o_1 = m_2(o)$ and then $o_2 = m_1(o, o')$; the result is $m_1(o_1, o_2)$. The range of composite methods is left unspecified (it is **any**) because it is determined by the domain and the method implementation as a composition of methods. Because the range of composite methods is always **any**, we will sometimes only specify their domain.

Let $\mathbf{S} = (C, \prec, M_{base}, M_{comp}, \mu)$ be a method schema. An *instance* of $\mathbf{S}$ is a pair $\mathbf{I} = (\pi, \nu)$, where π is an OID assignment for $(C, \prec)$ and where ν assigns a semantics to the base methods. Note the difference from the imperative schemas of the previous section, where π together with the method implementations was sufficient to determine the semantics of methods. In contrast, the semantics of the base methods must be specified in instances of method schemas.

Inheritance of method implementations for method schemas is defined slightly differently from that for the OODB model given earlier. Specifically, given an n-ary method m and invocation $m(o_1, \ldots, o_n)$, where o_i is in disjoint class c_i for $i \in [1, n]$, the implementation for m is inherited from the implementation of signature $m : c'_1, \ldots, c'_n \to c'$, where this is the unique signature that is pointwise least above $c_1, \ldots, c_n$. [Otherwise m is undefined on input $(o_1, \ldots, o_n)$.]

An important special case is when methods take just *one* argument. Method schemas using only such methods are called *monadic*. To emphasize the difference, unrestricted method schemas are sometimes called *polyadic*.

Example 21.4.4 Consider the following monadic method schema. The classes in the schema are

$$\begin{aligned} &\mathit{class}\ c \\ &\mathit{class}\ c' \prec c \end{aligned}$$

The base method signatures are

$$\begin{aligned} &\mathit{method}\ m_1 : c \to c' \\ &\mathit{method}\ m_2 : c \to c \\ &\mathit{method}\ m_2 : c' \to c' \\ &\mathit{method}\ m_3 : c' \to c \end{aligned}$$

The composite method definitions are

$$\begin{aligned} &\mathit{method}\ m : c = m_2(m_2(m_1(x))) \\ &\mathit{method}\ m' : c = m_3(m'(m_2(x))) \\ &\mathit{method}\ m' : c' = m_1(x) \end{aligned}$$

Note that m' is recursive and that calls to m' on elements in c' break the recursion.

Type Safety for Method Schemas As before, a method schema **S** is *type safe* if for each instance **I** of **S** no method call on **I** leads to a runtime type error.

The following example demonstrates that the schema of Example 21.4.4 is not type safe. Note how the interpretation ν for base methods can be viewed as an assignment of values for objects.

Example 21.4.5 Recall the method schema of Example 21.4.4. An instance of this is $\mathbf{I} = (\pi, \nu)$, where[7]

$$\begin{aligned} \pi(c) &= \{p, q\} \\ \pi(c') &= \{r\} \end{aligned}$$

---

[7] We write $\nu(m_1)(p)$ rather than $\nu(m_1, c)(p)$ to simplify the presentation.

and

$$\begin{array}{lll} \nu(m_1)(p) = r & \nu(m_2)(p) = q & \\ \nu(m_1)(q)l = \perp & \nu(m_2)(q) = r & \nu(m_3)(r) = p. \\ \nu(m_1)(r) = r & \nu(m_2)(r) = r & \end{array}$$

Consider the execution of $m(p)$. This calls for the computation of $m_2(m_2(m_1(p))) = m_2(m_2(r)) = r$. Thus the execution is successful. On the other hand, $m'(p)$ leads to a runtime type error: $m'(p) = m_3(m'(m_2(p))) = m_3(m'(q)) = m_3(m_3(m'(m_2(q)))) = m_3(m_3(m'(r))) = m_3(m_3(m_1(r))) = m_3(m_3(r)) = m_3(p)$, which is undefined and raises a runtime type error. Thus the schema is not type safe.

It turns out that type safety of method schemas permitting polyadic methods is undecidable (Exercise 21.19). Interestingly, type safety is decidable for monadic method schemas. We now sketch the proof of this result.

THEOREM 21.4.6 It is decidable in polynomial time whether a monadic method schema is type safe.

Crux Let $\mathbf{S} = (C, \prec, M_{base}, M_{comp}, \mu)$ be a monadic method schema. We construct a context-free grammar (see Chapter 2) that captures possible executions of a method call over all instances of $\mathbf{S}$. The grammar is $G_{\mathbf{S}} = (V_n, V_t, A, P)$, where the set V_t of terminals is the set of base method names (denoted N_{base}) along with the symbols $\{\langle error\rangle, \langle ignore\rangle\}$, and the set V_n of nonterminals includes start symbol A and

$$\{[c, m, c'] \mid c, c' \text{ are classes, and } m \text{ is a method name}\}$$

The set P of production rules includes

(i) $A \to [c, m, c']$, if m is a composite method name and it is defined at c or a superclass of c.

(ii) $[c, m, c'] \to \langle error\rangle$, if m is not defined at c or a superclass of c.

(iii) $[c, m, c'] \to m$, if m is a base method name, the resolution of m for c is $m : c_1 \to c_2$, and $c' \prec c_2$. (Note that $c' = c_2$ is just a particular case.)

(iv) $[c, m, c] \to \epsilon$, if m is a composite method name and the resolution of m for c is the identity mapping.

(v) $[c, m, c_n] \to [c, m_1, c_1][c_1, m_2, c_2] \ldots [c_{n-1}, m_n, c_n]$, if m is a composite method, m on c resolves to a method with implementation $m_n(m_{n-1}(\ldots(m_2(m_1(x)))\ldots))$, and $c_1, \ldots, c_n$ are arbitrary classes.

(vi) $[c, m, c'] \to \langle ignore\rangle$, for all classes c, c' and method names m.

Given a successful execution of a method call $m(o)$, it is easy to construct a word in $L(G_{\mathbf{S}})$ of the form $m_1 \ldots m_n$, where the m_i's list the sequence of base methods called during the execution. On the other hand, if the execution of $m(o)$ leads to a runtime error, a word of the form $m_1 \ldots m_i \langle error\rangle \ldots$ can be formed. The terminal $\langle ignore\rangle$ can be used in cases where

a nonterminal $[c, m, c']$ arises, such that m is a base method name and c' is outside the range of m for c. The productions of type (vi) are permitted for all nonterminals $[c, m, c']$, although they are needed only for some of them.

It can be shown that **S** is type safe iff

$$L(G_{\mathbf{S}}) \cap N^*_{base}\langle error\rangle V^*_t = \emptyset.$$

Because it can be tested if the intersection of a context-free language with a regular language is empty, the preceding provides an algorithm for checking type safety. However, a modification of the grammar $G_{\mathbf{S}}$ is needed to obtain the polynomial time test (see Exercise 21.18). ■

Expressive Power of Method Schemas We now argue that method schemas (with order) simulate precisely the relational queries in QPTIME. The object-oriented features are not central here: The same result can be shown for functional data models without such features.

As for imperative schemas, we show that method schemas can simulate relational queries. The encoding of these queries assumes an ordered domain, as is traditional in the world of functional programming.

A relational database is encoded as follows:

(a) a class **elem** contains objects representing the elements of the domain, and it has **zero** as a subclass containing a unique element, say 0;

(b) a function *pred*, which is included as a base method,[8] provides the predecessor function over **elem** ∪ **zero** [$pred(0)$ is, for instance, 0]; a base method 0 returns the least element and another base method N the largest object in **elem**;

(c) to have the Booleans, we think of 0 as the value *false* and all objects in **elem** as representations of *true*;

(d) an n-ary relation R is represented by an n-ary base method m_R of signature m_R : **elem**, ..., **elem** → **elem**, the characteristic function of R. [For a tuple t, $m_R(t)$ is *true* iff t is in R.]

Next we represent queries by composite methods. A query q is computed by method m_q if $m_q(t)$ is true (not in **zero**) iff t is in the answer to query q.

The following illustrates how to compute with this simple language.

EXAMPLE 21.4.7 Consider relation R with $R = \{R(1, 1), R(1, 2)\}$. The class **zero** is populated with the object 0 and the class **elem** with 1, 2. The base method *pred* is defined by $pred(2) = 1, pred(0) = pred(1) = 0$. The base method m_R is defined by $m_R(1, 1) = m_R(1, 2) = 1$ and $m_R(x, y) = 0$ otherwise.

---

[8] The function *pred* is a functional analog of the relation *succ*, which we have assumed is available in every ordered database (a successor function could also have been used).

Recall that each object in class **elem** is viewed as *true* and object 0 as *false*. We can code the Boolean function *and* as follows:

$$\begin{array}{lll} \text{for } x, y \text{ in} & \textbf{zero}, \textbf{zero} & and(x, y) \equiv 0 \\ \text{for } x, y \text{ in} & \textbf{elem}, \textbf{zero} & and(x, y) \equiv 0 \\ \text{for } x, y \text{ in} & \textbf{zero}, \textbf{elem} & and(x, y) \equiv 0 \\ \text{for } x, y \text{ in} & \textbf{elem}, \textbf{elem} & and(x, y) \equiv N. \end{array}$$

The other standard Boolean functions can be coded similarly. We can code the intersection between two binary relations R and S with $and(m_R(x, y), m_S(x, y))$. As a last example, the projection of a binary relation R over the first coordinate can be coded by a method $\pi_{R,1}$ defined by

$$\pi_{R,1} \equiv m(x, N),$$

where m is given by

$$\begin{array}{lll} \text{for } x, y \text{ in} & \textbf{elem}, \textbf{zero} & m(x, y) \equiv m_R(x, y) \\ \text{for } x, y \text{ in} & \textbf{elem}, \textbf{elem} & m(x, y) \equiv or(m_R(x, y), m(x, pred(y))). \end{array}$$

We now state the following:

THEOREM 21.4.8 Method schemas over ordered databases express exactly QPTIME.

Crux As indicated in the preceding example, we can construct composite methods for the Boolean operations *and*, *or*, and *not*. For each k, we can also construct k k-ary functions $pred^i_k$ for $i \in [1, k]$ that compute for each k tuple u the k components of the predecessor (in lexicographical ordering) of u. Indeed, we can simulate an arbitrary relational operation and more generally an arbitrary inflationary fixpoint. To see this, consider the transitive closure query. It is computed with a method *tc* defined (informally) as follows. Intuitively, a method $tc(x, y)$ asks, "Is $\langle x, y \rangle$ in the transitive closure?" Execution of $tc(x, y)$ first calls a method $m_1(x, y, N)$, whose intuitive meaning is "Is there a path of length N from x to y?" This will be computed by asking whether there is a path of length $N - 1$ (a recursive call to m_1), etc. This can be generalized to a construction that simulates an arbitrary inflationary fixpoint query. Because the underlying domain is ordered, we have captured all QPTIME queries. The converse follows from the fact that there are only polynomially many possible method calls in the context of a given instance, and each method call in this model can be answered in QPTIME. Moreover, loops in method calls can be detected in polynomial time; calls giving rise to loops are assumed to output some designated special value. (See Exercise 21.25.) ■

We have presented an object-oriented approach in the applicative programming style. There exists another important family of functional languages based on typed λ calculi. It is possible to consider database languages in this family as well. These calculi present

additional advantages, such as being able to treat functions as objects and to use higher-order functions (i.e., functions whose arguments are functions).

21.5 Further Issues for OODBs

As mentioned at the beginning of this chapter, the area of OODB is relatively young and active. Much research is needed to understand OODBs as well as we understand relational databases. A difficulty (and richness) is that there is still no well-accepted model. We conclude this chapter with a brief look at some current research issues for OODBs. These fall into two broad categories: advanced modeling features and dynamic aspects.

Advanced Modeling Features

This is not an exhaustive list of new features but a sample of some that are being studied:

Views: Views are intended to increase the flexibility of database systems, and it is natural to extend the notion of relational view to the OODB framework. However, unlike relational views, OODB views might redefine the behavior of objects in addition to restructuring their associated types. There are also significant issues raised by the presence of OIDs. For example, to maintain incrementally a materialized view with created OIDs, the linkage between the base data and the created OIDs must be maintained. Furthermore, if the view is virtual, then how should virtual OIDs be specified and manipulated?

Object roles: The same entity may be involved in several roles. For instance, a director may also be an actor. It is costly, if not infeasible, to forecast all cases in which this may happen. Although not as important in object-oriented programming, in OODBs it would be useful to permit the same object to live in several classes (a departure from the disjoint OID assignment from which we started) and at least conceptually maintain distinct repositories, one for each role. This feature is present in some semantic data models; in the object-oriented context, it raises a number of interesting typing issues.

Schema design: Schema design techniques (e.g., based on dependencies and normal forms) have emerged for the relational model (see Chapter 11). Although the richer model in the OODB provides greater flexibility in selecting a schema, there is a concomitant need for richer tools to facilitate schema design. The scope of schema design is enlarged in the OODB context because of the interaction of methods within a schema and application software for the schema.

Querying the schema: In many cases, information may be hidden in an OODB schema. Suppose, for example, that movies were assigned categories such as "drama," "western," "suspense," etc. In the relational model, this information would typically be represented using a new column in the *Movies* relation. A query such as "list all categories of movie that Bergman directed" is easily answered. In an OODB, the category information might be represented using different subclasses of the *Movie* class. Answering this query now requires the ability of the query language to return class names, a feature not present in most current systems.

Classification: A related problem concerns how, given an OODB schema, to classify new data for this schema. This may arise when constructing a view, when merging two databases, or when transforming a relational database into an OODB one by objectifying tuples. The issue of classification, also called taxonomic reasoning, has a long history in the field of knowledge representation in artificial intelligence, and some research in this direction has been performed for semantic and object-oriented databases.

Incorporating deductive capabilities: The logic-programming paradigm has offered a tremendous enhancement of the relational model by providing an elegant and (in many cases) intuitively appealing framework for expressing a broad family of queries. For the past several years, researchers have been developing hybrids of the logic-programming and object-oriented paradigms. Although it is very different in some ways (because the OO paradigm has fundamentally imperative aspects), the perspective of logic programming provides alternative approaches to data access and object creation.

Abstract data types: As mentioned earlier, OODB systems come equipped with several constructors, such as set, list, or bag. It is also interesting to be able to extend the language and the system with application-specific data types. This involves language and typing issues, such as how to gracefully incorporate access mechanisms for the new types into an existing language. It also involves system issues, such as how to introduce appropriate indexing techniques for the new type.

Dynamic Issues

The semantics of updates in relational systems is simple: Perform the update if the result complies with the dependencies of the schema. In an OODB, the issue is somewhat trickier. For instance, can we allow the deletion of an object if this object is referred to somewhere in the database (the dangling reference problem)? This is prohibited in some systems, whereas other systems will accept the deletion and just mark the object as dead. Semantically, this results in viewing all references to this object as nil.

Another issue is object migration. It is easy to modify the value of an object. But changing the status of an object is more complicated. For example, a person in the database may act in a movie and overnight be turned into an actor. In object-oriented programming languages, objects are often not allowed to change classes. Although such limitations also exist in most present OODBs, object migration is an important feature that is needed in many database applications. One approach, followed by some semantic data models, is to permit objects to be associated with multiple classes or roles and also permit them to migrate to different classes over time. This raises fundamental issues with regard to typing. For example, how do we treat a reference to the manager of a department (that should be of type *Employee*) when he or she leaves the company and is turned into a "normal" person?

Finally, as with the relational model, we need to consider evolution of the schema itself. The OODB context is much richer than the relational, because there are many more kinds of changes to consider: the class hierarchy, the type of a class, additions or deletions of methods, etc.

Bibliographic Notes

Collections of papers on object-oriented databases can be found in [BDK92, KL89, ZM90]. The main characteristics of object-oriented database systems are described in [ABD+89]. An influential discussion of some foundational issues around the OODB paradigm is [Bee90]. An important survey of subtyping and inheritance from the perspective of programming languages, including the notion of domain-inclusion semantics, is [CW85].

Object-oriented databases are, of course, closely related to object-oriented programming languages. The first of these is Smalltalk [GR83], and C++ [Str91] is fast becoming the most widely used object-oriented programming language. Several commercial OODBs are essentially persistent versions of C++. Several object-oriented extensions of Lisp have been proposed; the article [B+86] introduces a rich extension called CommonLoops and surveys several others.

There have been a number of approaches to provide a formal foundation [AK89, Bee90, HTY89, KLW93] for OODBs. We can also cite as precursors attempts to formalize semantic data models [AH87] and object-based models [KV84, HY84]. Recent graph-oriented models, although they do not stress object orientation, are similar in spirit (e.g., [GPG90]).

The generic OODB model used here is directly inspired from the IQL model [AK89] and that of O_2 [BDK92, LRV88]. The model and results on imperative method implementations are inspired by [HTY89, HS89a]. A similar model of imperative method implementation, which avoids nondeterminism and introduces a parallel execution model, is developed in [DV91]. Method schemas and Theorem 21.4.6 are from [AKRW92]; the functional perspective and Theorem 21.4.8 are from [HKR93].

OIDs have been part of many data models. For example, they are called *surrogates* in [Cod79], *l-values* in [KV93a], or *object identifiers* in [AH87]. The notion of object and the various forms of equalities among objects form the topic of [KC86]. Type inheritance and multiple inheritance are studied in [CW85, Car88].

Since [KV84], various languages for models with objects have been proposed in the various paradigms: calculus, algebra, rule based, or functional. Besides standard features found in database languages without objects, the new primitives are centered around object creation. Language-theoretic issues related to object creation were first considered in the context of IQL [AK89]. Object creation is an essential component of IQL and is the main reason for the completeness of the language. The need for a primitive in the style of copy elimination to obtain determinate completeness was first noticed in [AK89]. The IQL language is rule based with an inflationary fixpoint semantics in the style of datalog$^\neg$ of Chapter 14.

The logic-based perspective on object creation based on Skolem was first informally discussed in [Mai86] and refined variously in [CW89a, HY90, KLW93, KW89]. In particular, F-logic [KLW93] considers a different approach to inheritance. In our framework, the classification of objects is explicit; in particular, when an object is created, it is within an explicit class. In [KLW93], data organization is also specified by rules and thus may depend on the properties of the objects involved. For instance, reasoning about the hierarchy becomes part of the program.

Algebraic and imperative approaches to object creation are developed in [Day89].

Since then, object creation has been the center of much interesting research (e.g., [DV93, HS89b, HY92, VandBG92, VandBGAG92, VandB93]). The characterization of queries expressible in $while_{obj}$ (Theorem 21.3.2) is from [VandBG92]; this extends a previous result from [AP92]. The proof of Proposition 21.3.3 is also from [VandBGAG92]. In [VandBGAG92, VandB93], it is argued that the notion of determinate query may not be the most appropriate one for the object-based context, and alternative notions, such as semideterministic queries, are discussed. A tractable construct yielding a determinate-complete language is exhibited in [DV93]. However, the construct proposed there is global in nature and is involved. The search for simpler and more natural local constructs continues.

As mentioned earlier, the OODB calculus and algebra presented here are mostly variations of languages for non object-based models and, in particular, of the languages for complex values of Chapter 20. There have been several proposals of SQL extensions. In particular, as indicated in Section 21.3, O_2SQL [BCD89] retains the flavor of SQL but incorporates object orientation by adopting an elegant functional programming style. This approach has been advanced as a standard in [Cat94].

Functional approaches to databases have been considered rather early but attracted only modest interest in the past [BFN82, Shi81]. The functional approach has become more popular recently, both because of the success of object-oriented databases and due to recent results of complex objects and types emphasizing the functional models [BTBN92, BTBW92]. The use of a typed functional language similar to λ calculus as a formalism to express queries is adapted from [HKM93]. Characterizations of QPTIME in functional terms are from [HKM93, LM93]. The work in [AKRW92, HKM93, HKR93] provides interesting bridges between (object-oriented) databases and well-developed themes in computer science: applicative program schemas [Cou90, Gre75] and typed λ calculi [Chu41, Bar84, Bar63].

This chapter presented both imperative and functional perspectives on OODB methods. A different approach (based on rules and datalog with negation) has been used in [ALUW93] to provide semantics to a number of variations of schemas with methods. The connection between methods and rule-based languages is also considered in [DV91].

Views for OODBs are considered in [AB91, Day89, HY90, KKS92, KLW93]. The merging of OODBs is considered in [WHW90]. Incremental maintenance of materialized object-oriented views is considered in [Cha94]. The notion of object roles, or sharing objects between classes, is found in some semantic data models [AH87, HK87] and in recent research on OODBs [ABGO93, RS91]. A query language that incorporates access to an OODB schema is presented in [KKS92]. Classification has been central to the field of knowledge representation in artificial intelligence, based on the central notion of taxonomic reasoning (e.g., see [BGL85, MB92], which stem from the KL-ONE framework of [BS85]); this approach has been carried to the context of OODBs in, for example, [BB92, BS93, BBMR89, DD89]. Deductive object-oriented database is the topic of a conference (namely, the *Intl. Conf. on Deductive and Object-Oriented Databases*). Properties of object migration between classes in a hierarchy are studied in [DS91, SZ89, Su92].

Exercises

Exercise 21.1 Construct an instance for the schema of Fig. 21.1 that corresponds to the **CINEMA** instance of Chapter 3.

Exercise 21.2 Suppose that the class *Actor_Director* were removed from the schema of Fig. 21.1. Verify that in this case there is no OID assignment for the schema such that there is an actor who is also a director.

Exercise 21.3 Design an OODB schema for a bibliography database with articles, book chapters, etc. Use inheritance where possible.

Exercise 21.4 Exhibit a class hierarchy that is not well formed.

Exercise 21.5 Add methods to the schema of Fig. 21.1 so that the resulting family of methods violates rules *unambiguous* and *covariance*.

Exercise 21.6 Show that testing whether $\mathbf{I} \equiv_{OID} \mathbf{J}$ is in NP and at least at hard as the graph isomorphism problem (i.e., testing whether two graphs are isomorphic).

Exercise 21.7 Give an algorithm for testing value equality. What is the data complexity of your algorithm?

Exercise 21.8 In this exercise, we consider various forms of equality. Value equality as discussed in the text is denoted $=_1$. Two objects o, o' are 2-value equal, denoted $o =_2 o'$, if replacing each object in $\nu(o)$ and $\nu(o')$ by its value yields values that are equal. The relations $=_i$ for each i are defined similarly. Show that for each i, $=_{i+1}$ refines $=_i$. Let n be a positive integer. Give a schema and an instance over this schema such that for each i in $[1, n]$, $=_i$ and $=_{i+1}$ are different.

Exercise 21.9 Design a database schema to represent information about persons, including males and females with names and husbands and wives. Exhibit a cyclic instance of the schema and an object o that has an infinite expansion. Describe the infinite tree representing the expansion of o.

★ **Exercise 21.10** Consider a database instance $\mathbf{I}$ over a schema $\mathbf{S}$. For each o in $\mathbf{I}$, let $expand(o)$ be the (possibly infinite) tree obtained by replacing each object by its value recursively. Show that $expand(o)$ is a *regular* tree (i.e., that it has a finite number of distinct subtrees). Derive from this observation an algorithm for testing deep equality of objects.

Exercise 21.11 In this exercise, we consider the schema $\mathbf{S}$ with a single class c that has type $\sigma(c) = [A : c, B : \mathbf{string}]$. Exhibit an instance $\mathbf{I}$ over $\mathbf{S}$ and two distinct objects in $\mathbf{I}$ that have the same expansion. Exhibit two distinct instances over $\mathbf{S}$ with the same set of object expansions.

Exercise 21.12 Sketch an extension of the complex value algebra to provide an algebraic simulation of the calculus of Section 21.3. Give algebraic versions of the queries of that section.

♠ **Exercise 21.13** Recall the approach to creating OIDs by extending datalog to use Skolem function symbols. Consider the following programs:

$$
\begin{array}{ll}
T(f_1(x, y), x) \leftarrow S(x, y) & T(f_3(x, y), x) \leftarrow S(x, y) \\
T(f_2(x, y), x) \leftarrow S(x, y) & T(f_3(y, x), x) \leftarrow S(x, y) \\
T(f_1(x, y), y) \leftarrow S(x, y), S(y, x) & T(f_4(x, y), x) \leftarrow S(x, y), S(y, x) \\
P & Q
\end{array}
$$

(a) Two programs P_1, P_2 involving Skolem terms such as the foregoing are *exposed equivalent*, denoted $P_1 \sim_{exp} P_2$, if for each input instance $\mathbf{I}$ having no OIDs, $P_1(\mathbf{I}) = P_2(\mathbf{J})$. Show that $P \sim_{exp} Q$ does not hold.

(b) Following the ILOG languages [HY92], given an instance $\mathbf{J}$ possibly with Skolem terms, an *obscured* version of $\mathbf{J}$ is an instance $\mathbf{J}'$ obtained from $\mathbf{J}$ by replacing each distinct nonatomic Skolem term with a new OID, where multiple occurrences of a given Skolem term are replaced by the same OID. (Intuitively, this corresponds to hiding the history of how each OID was created.) Two programs P_1, P_2 are *obscured equivalent*, denoted $P_1 \sim_{obs} P_2$, if for each input instance $\mathbf{I}$ having no OIDs, if $\mathbf{J}_1$ is an obscured version of $P_1(\mathbf{I})$ and $\mathbf{J}_2$ is an obscured version of $P_2(\mathbf{I})$, then $\mathbf{J}_1 \equiv_{OID} \mathbf{J}_2$. Show that $P \sim_{obs} Q$.

(c) Let P and Q be two nonrecursive datalog programs, possibly with Skolem terms in rule heads. Prove that it is decidable whether $P \sim_{exp} Q$. *Hint:* Use the technique for testing containment of unions of conjunctive queries (see Chapter 4).

★(d) A nonrecursive datalog program with Skolem terms in rule heads has *isolated OID invention* if in each target relation at most one column can include nonatomic Skolem terms (OID). Give a decision procedure for testing whether two such programs are obscured equivalent. (Decidability of obscured equivalence of arbitrary nonrecursive datalog programs with Skolem terms in rule heads remains open.)

♠ **Exercise 21.14** [VandBGAG92] Prove the "only if" part of Theorem 21.3.2. *Hint:* Associate traces to new object id's, similar to the proof of Theorem 18.2.5. The extension homomorphism is obtained via the natural extension to traces of automorphisms of the input.

Exercise 21.15 [HTY89]

(a) Define an operational semantics for the imperative model introduced in Section 21.4.

(b) Describe how a method in this model can simulate a *while*loop of arbitrary length. *Hint:* Use a class c with associated type $\mathbf{tuple}(a : c, \ldots)$, and let $c' \prec c$. Construct the implementation of method m on c so that on input o if the loop is to continue, then it creates a new object o' in c, sets $o.a = o'$, and calls m on o'. To terminate the loop, create o' in c', and define m on c' appropriately.

(c) Show how the computation of a Turing machine can be simulated by this model.

Exercise 21.16 Prove Proposition 21.4.1. *Hint:* Use a reduction from the PCP problem, similar in spirit to the one used in the proof of Theorem 6.3.1. The effect of conditionals can be simulated by putting objects in different classes and using dynamic binding.

Exercise 21.17 Describe how monadic method schemas can be simulated in the imperative model.

Exercise 21.18 [AKRW92]

(a) Verify that the grammar $G_{\mathbf{S}}$ described in the proof of Theorem 21.4.6 has the stated property.

(b) How big is $G_{\mathbf{S}}$ in terms of $\mathbf{S}$?

(c) Find a variation of $G_{\mathbf{S}}$ that has size polynomial in the size of $\mathbf{S}$. *Hint:* Break production rules having form (v) into several rules, thereby reducing the overall size of the grammar.

(d) Complete the proof of the theorem.

Exercise 21.19 [AKRW92]

★ (a) Show that it is undecidable whether a polyadic method schema is type safe. *Hint:* You might use undecidability results for program schemas (see Bibliographic Notes), or you might use a reduction from the PCP.

★ (b) A schema is *recursion free* if there are no two methods m, m' such that m occurs in some code for m' and conversely. Show that type safety is decidable for recursion-free method schemas.

Exercise 21.20

(a) Complete the formal definition of an imperative schema simulating a relational query.

(b) Prove Theorem 21.4.2.

♠ **Exercise 21.21**

(a) Suppose that the imperative model were extended to include types for classes that have one level of the set construct (so tuple of set of tuple of atomic of class types is permitted) and that the looping construct is extended to the sets occurring in these types. Assume that the **new** command is not permitted. Prove that the family of relational queries that this model can simulate is QPSPACE. *Hint:* Intuitively, because the looping operates object at a time, it permits the construction of a nondeterministic ordering of the database.

(b) Suppose that n levels of set nesting are permitted in the types of classes. Show that this simulates $\text{QEXP}^{n-1}\text{SPACE}$.

Exercise 21.22

(a) Describe how the form of method inheritance used for polyadic method schemas can be simulated using the originally presented form of method inheritance, which is based only on the class of the first argument.

(b) Suppose that a base method m_R in an instance of a polyadic method schema is used to simulate an n-ary relation R. In a simulation of this situation by an instance of a conventional OODB schema, how many OIDs are present in the class on which m_R is simulated?

Exercise 21.23 Show how to encode or, not, and $equal$ using method schemas.

Exercise 21.24 Show how to encode $pred_k^i$ and the join operation using method schemas.

♠ **Exercise 21.25** [HKR93] Prove Theorem 21.4.8. *Hint:* Show first that method schemas can simulate relational algebra and then inflationary fixpoint. For the fixpoint, you might want to use $pred_k$. For the other direction, you might want to simulate method schemas over ordered databases by inflationary fixpoint.

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 $mod(C \rightarrow C')$, 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:

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

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:

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

and a slightly more complex one:

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

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:

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

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

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

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

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

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

$$\langle hire(emp1, sal1, dep1)\rangle \leftarrow$$
$$\langle +ES(emp1, sal1)\rangle(\langle +ED(emp1, dep1)\rangle(DA(dep1, avg1) \ \& \ avg1 < 50k)).$$

(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:

$$:- \quad assert(ES(john, 200)), assert(ED(john, toys)),$$
$$DA(toys, Avg1), Avg1 < 50.$$

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 $\mathbf{R}$ together with a finite set of parameterized IDM transactions over $\mathbf{R}$ is called an *IDM transactional schema*.

Example 22.2.1 Consider a database schema $\mathbf{R}$ 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 $\mathbf{T}$ 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 $\mathbf{T}$ 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:

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

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 $\mathbf{T}$ 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 $\mathbf{B}$ and view schema $\mathbf{V}$ 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 $\mathbf{R}$ 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

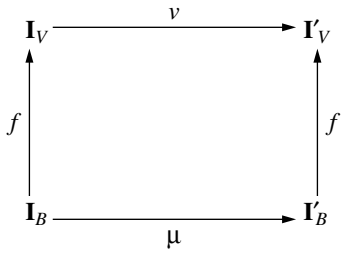


Figure 22.1: Relationship of views and updates

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:

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

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 $b' \in \mathbf{dom}$.

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

(d) Insert $\langle a', b'\rangle$ into R for some $b' \in \mathbf{dom}$, 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 $R : A \to B$, 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

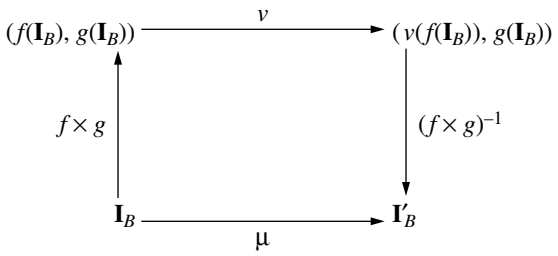


Figure 22.2: Properties of a g-translation μ of view update ν on view f

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

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

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 $\overline{t}(T)$ 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 $Mod(\mathbf{T})$. In our context, each model corresponds to a different possible instance. If $|Mod(\mathbf{T})| > 1$, then $\mathbf{T}$ 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 $\mathbf{T}$ (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 $\mathbf{T}$ is *model based*; it focuses on how proposed updates affect the elements of $Mod(\mathbf{T})$. Given an update u and instance $\mathbf{I}$, let $u(\mathbf{I})$ denote the set of possible instances that could result from applying u to $\mathbf{I}$. We use a set for the result to accommodate the case in which u itself involves incomplete information.

Now let $\mathbf{T}$ be a theory and u an update. Under the model-based approach, the result $u(\mathbf{T})$ of applying u to $\mathbf{T}$ 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 $\mathbf{T}$ (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 $\mathbf{T}$ 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 $\mathbf{I}$. 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 $A \rightarrow B$, 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 $\mathbf{T}$ 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,

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

$\mathbf{T}_1$ 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 $u = [insert\ \varphi]$, 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:

Suppliers[*Sname*, *Address*]
Prices[*Part*, *Sname*, *Price*]

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

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

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

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

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 $\mathbf{I}^{user} = apply(\mathbf{I}^{orig}, \Delta_0)$ and, more generally, $\mathbf{I}_{i+1} = apply(\mathbf{I}_i, \Delta_i)$ 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 $\Delta_0 \& \ldots \& \Delta_i = fail$, 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 $effect(r, \mathbf{I}, \Delta)$, 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

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

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

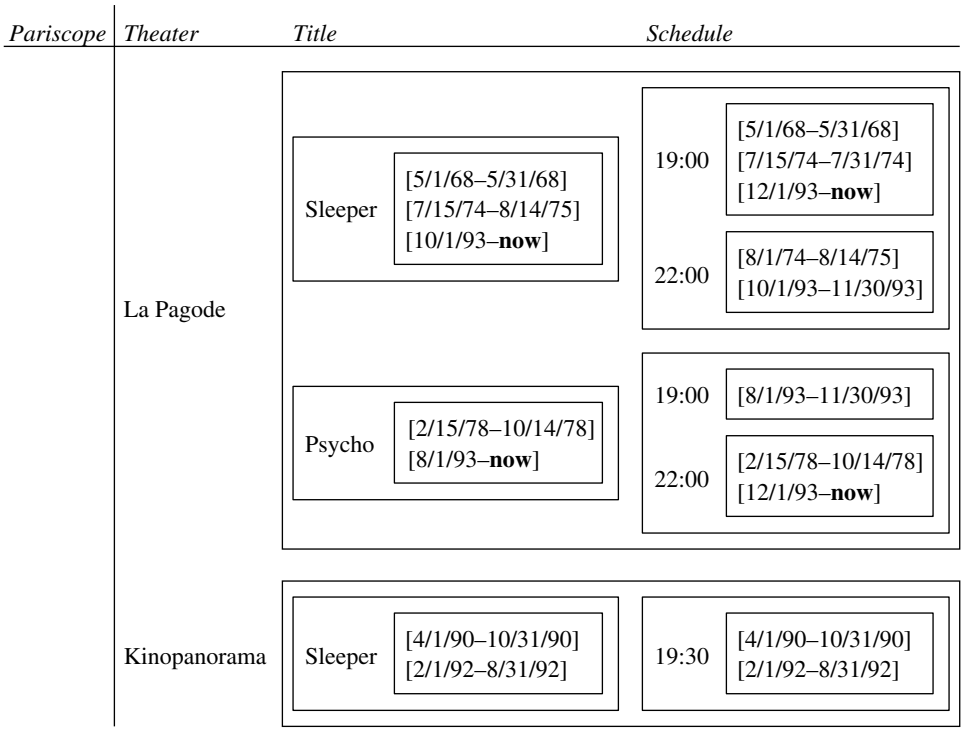


Figure 22.4: A representation of temporal information using attribute timestamps with nested relations

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:

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

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:

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

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

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

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

$$mod(C \to C')del(C') \equiv del(C)del(C')$$
$$ins(t)mod(C \to C') \equiv mod(C \to C')ins(t')$$
where t satisfies C 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 $\mathbf{T}$ is sound and complete with respect to Σ.

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

(c) Write a transactional schema $\mathbf{T}'$ that is sound and complete for Σ, such that whenever $\mathbf{I}$, $\mathbf{J}$ are in $Sat(\Sigma)$, there is a transformation from $\mathbf{I}$ to $\mathbf{J}$ 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 $\mathbf{T}$ and instance $\mathbf{I}$ 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 $\mathbf{T}$ 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 $\mathbf{T}$ needed to reach $\mathbf{I}$ 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 $\mathbf{T}$ 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 $\mathbf{T}$ over R.

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

♠ **Exercise 22.12** [AV89] Prove that it is undecidable whether $Gen(\mathbf{T})$ equals all instances over $\mathbf{R}$ for given IDM transaction schema $\mathbf{T}$ over $\mathbf{R}$. 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 $\mathbf{T}$ corresponding to a c-table and an update, describe how to change $\mathbf{T}$ 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 $\mathbf{T}$, 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 $\mathbf{T}$ be a closed theory, φ a sentence not in $\mathbf{T}$, and $\mathbf{T}'$ a closed theory that accomplishes [*insert* φ] for $\mathbf{T}$. 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 $\mathbf{I}$ 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 $Inst(\mathbf{B})$ 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 $g : Inst(\mathbf{I}) \to Inst(\mathbf{I})/\equiv$ so that $g(\mathbf{I})$ is the equivalence class of $\mathbf{I}$ under $\equiv$.