■ *CHAPTER 8* ■

Missing Information

The purpose of this chapter is to clarify and summarize the way missing information is treated in Version 2 of the relational model. The clarification places heavy emphasis on the semantic aspects of missing information. The systematic approach of RM/V1 has been extended in RM/V2 to deal with the inapplicability of certain properties to some objects. Once again, this treatment is independent of the data type of the missing information. This extension does not invalidate any part of RM/V1.

In RM/V2 the approach to manipulating information from which values may be missing represents my current thinking about this problem. I do not feel this part of the relational model rests on such a solid theoretical foundation as the other parts. However, I do think that this approach represents a considerable improvement over the prerelational methods that amounted to leaving it up to application programmers to solve it in many different and specialized ways (even within a single installation).

8.1 ■ Introduction to Missing Information

In Section 2 of my paper on the extended model RM/T [Codd 1979], I included an account of *how* the basic relational model RM/V1 represents and handles missing information, but gave very little emphasis to *why* that

approach was adopted. Included in that discussion of the manipulation of missing information was an account of the three-valued logic proposed for determining the possibilities if some of the missing information were conceptually and temporarily replaced by known values.

Criticisms have been fired at the three-valued logic approach of RM/V1. In its place, the critics propose, in effect, a return to the "good old days" when, for each column permitted to have missing information, the database administrator or some suitably authorized user is forced to select a specific value from the particular domain on which the column is defined to denote the fact that information in that column is missing.

The case for a logic having more than two truth-values is discussed in Sections 8.9 and 8.10. Criticisms are answered in Chapter 9.

In this chapter, the representation and handling of missing information are described according to the way such information is treated in RM/V2. This approach provides a stronger semantic underpinning than any non-relational approach. One of the relatively new extensions in RM/V2 is the treatment of a property that is generally applicable to a class of objects, but inapplicable to certain members of that class. One example of such a property is the name of the spouse of each employee, when there may be a significant number of employees who are not married, and therefore have no spouse. Another example is the sales commission earned to-date in an EMPLOYEE relation, which deals with both salespeople and non-salespeople.

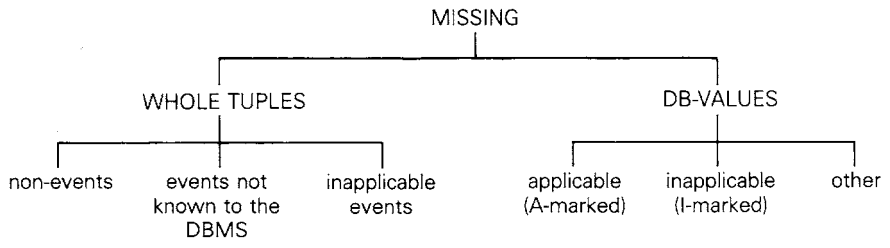
It is the meaning of the data that determines whether a database value is missing-but-applicable or missing-and-inapplicable. Thus, it is the database designer who should establish for each column of each relation whether missing values of each type are permitted or prohibited in that column. Sometimes a missing and inapplicable status can be derived from data found elsewhere in the database. For example, in the salesperson example, it is quite likely that the EMPLOYEE relation contains as another column the job type within the company or institution. From this datum, the DBMS can determine whether an elementary database value (db-value for brevity) is permitted or prohibited in the sales-commission column. This topic is discussed further in Chapters 13 and 14, which deal with integrity constraints.

The two types of missing information that are defined in Section 8.2 and stressed in this chapter are missing but applicable (denoted A) and missing and inapplicable (denoted I). Section 8.19 deals with operators that generate marks, and briefly discusses the relationship between (1) A and I applied to whole rows and (2) A and I applied to components of a row.

Figure 8.1 illustrates the kinds of information that can be missing. Later in this chapter, a description of how RM/V2 handles these two types of missing information, types A and I, is given.

The various technical criticisms of the RM/V1 approach to missing information that have recently come to my attention are discussed in detail in Chapter 9. That discussion includes some strong technical arguments against one proposed alternative, the scheme of "default values."

Figure 8.1 **Classification of Missing Information in a Relational DBMS**



8.2 ■ Definitions

In logic and in algebra, when the value or possible values of an item are unknown, a named variable is assigned to the item and it is usually called an *unknown*. Distinct items with unknown values are assigned variables with distinct names. Thus, a formula in logic or in algebra may involve several variables, and a common task in solving a problem is to find the values of these variables using a collection of equations.

In database management, the same approach *could* be followed. Thus, if a database contained information about employees and projects, *each occurrence* of an unknown birthdate of an employee and of an unknown start-date for a project could be recorded as a distinctly named variable.

Under certain circumstances, the DBMS might be able to deduce equality or inequality between two distinctly named variables, or to deduce certain other constraints on the variables. It would rarely be possible, however, for the DBMS to deduce the actual values of these variables. Instead, most of the unknown or missing-and-applicable items are eventually supplied by users in the form of late-arriving input or not-yet-completed calculation. On the other hand, those items marked missing-and-inapplicable behave more like unknowable items than like unknowns.

Note that in the few cases where it is conceivably possible for the DBMS to deduce actual values for missing information, the cost of such deduction is likely to be too expensive relative to the actual benefit. In the present version of the relational model, the potential complexities of using the variables of algebra or logic for missing information are avoided. Moreover, compared with pre-relational approaches, the RM/V2 approach continues to place more burden on the system and less on the application programmers and terminal users.

I have made *no* claim, now or in the past, that the relational approach to missing information places *no* burden at all on users. Any attempt, however, to put missing information on a systematic basis (i.e., an attempt

that is uniformly applied to data whatever its type) will necessarily entail a learning burden. It is important that this learning burden should pay off in terms of a safer and more reliable treatment of databases, one that will strengthen the retention of database integrity.

As noted earlier, the term "elementary database value" is written as db-value for brevity. This term means any value that a single column may have in any relation. Except for certain special functions, a db-value is *atomic* in the context of the relational model. The term "datum" would have been preferred, but its plural, "data" has very broad use.

With regard to missing information, two questions seem dominant:

1. What kind of information is missing?
2. What is the main reason for its being missing?

In the relational approach, Question 1, regarding the kind of information, can be interpreted as a question concerning structural context: is the missing information a whole row, a component atomic value (a db-value) of a row, or a combination of these atomic values? There appears to be no need to consider the consequences of an entire relation being missing because a database necessarily models just a micro-world.

Moreover, the "main reason" in Question 2 can be interpreted to mean, "Is the information missing simply because its present value is unknown to the users, but that value is *applicable* and can be entered whenever it happens to be forthcoming? Or is it missing because it represents a property that is *inapplicable* to the particular object represented by the row involved?" Figure 8.1 summarizes the classification by kind (the structural context) and by reason.

In a certain row of a relation that describes the capabilities of suppliers, it may be recorded that supplier s3 is capable of supplying part p5, but the current price for this part is missing. This is an example of a missing-but-applicable value. In those rows of the EMPLOYEE relation that describe employees who are not legally married, the name of the employee's spouse is missing. This is an example of a missing-and-inapplicable value.

Note that, although there are many other ways to classify missing information, only the missing-but-applicable and missing-and-inapplicable types appear to justify general support by the DBMS at this time. In this context, "general" means independent of the particular column and of its domain or extended data type. Since DBMS users and even designers are not yet accustomed to using techniques of this generality for handling missing information, gradual introduction appears appropriate.

A basic principle of the relational approach to missing information is that of recording *the fact that a db-value is missing* by means of a *mark* (originally called a *null* or *null value*). There is nothing imprecise about a mark: a db-value is either present or absent in a column of a relation in the database.

The semantics of the fact that a db-value is missing are *not* the same as the semantics of the db-value itself. The former fact applies to any db-value, no matter what its type. The latter fact has semantics depending heavily on the domain (or extended data type) from which the column draws its values.

Like a variable, a mark is a placeholder. It does not, however, conform to the other accepted property of a variable—namely, that semantically distinct missing values are represented by distinctly named variables.

We begin with a definition of the *missing-but-applicable value mark* (for brevity, an A-mark). This mark is treated *neither* as a value *nor* as a variable by the DBMS, although it may be treated as a special kind of value by the host language. Consider an immediate property P of objects of type Z in a database. Normally P has a specific value in each and every row of that relation that provides the immediate properties of type Z objects. Suppose that, represented in the database, there is an object z of type Z and that, at this time, the value of P for this object is unknown. Then, P would be assigned an A-mark in the database, provided P is considered to be applicable to the object z. In the example introduced previously, Z is the capability of a supplier, z is the combination of supplier s3 and part p5, and P is the price that s3 charges for p5.

On the other hand, suppose that property P is inapplicable to the particular object z. Then P would be assigned an *inapplicable-value mark* (for brevity, an I-mark) in the row representing z. Thus, in the P column, each row contains a value for P *or* an A-mark *or* an I-mark. In the example cited previously, Z describes employees, z is any unmarried employee, and P is the name of the employee's spouse. Two more examples follow:

1. If an employee has a *missing-but-applicable* present salary, his or her record would have an A-mark in the salary column.
2. If an employee has an *inapplicable* sales commission (such an employee does not sell any products at this time), his or her record would have an I-mark in the commission column.

Sometimes the occurrence of an I-mark in one component of a row is based on data that occurs in other components of that row. In Example 2, both the job category and sales commission may be a component of each row of the EMPLOYEE relation. In that case, it is quite likely that inapplicability of the sales commission can be derived from the job category.

Why are these items now called "marks" rather than "values," "null values," or "nulls"? Four reasons follow:

1. The DBMS does not treat marks as if they were values.
2. There are now two kinds of marks, where there was previously just one kind of null.
3. Some host languages deal with objects called "nulls" that are quite different in meaning from database marks.

4. "Marked" and "unmarked" are better adjectives in English than are "nulled," "un-nulled," and "nullified."

To pursue the first reason, a mark in a numeric column (a column that normally has numeric values) cannot be arithmetically incremented or decremented by the DBMS, whereas the numeric values that are present can be subjected to such operators. To be more specific, if x denotes a db-value, A denotes an A-mark, and I denotes an I-mark, this is the effect of the arithmetic operator **addition:**

x + x = 2x	x + A = A	A + x = A
A + A = A	A + I = I	I + A = I
I + I = I	x + I = I	I + x = I

A similar table holds for the three arithmetic operators **minus, times,** and **divide** (except that when both arguments are db-values the result is what one would expect from ordinary arithmetic). Similarly, a mark that appears in a character-string column (one that normally has character-string values) cannot have a second character-string concatenated with that mark by the DBMS, in contrast to the character-string values that are present in the database. A table similar to the one just given for addition also holds for concatenation. These remarks can be summarized as follows:

> If I-marks are placed in the top class, A-marks in the second class, and all db-values in the third class, the combination (arithmetic or otherwise) of any two items is an item of whichever class is the higher of the two operands.

How, then, can these marks appear in a column that normally contains values? Present hardware is of little help: it fails to support any special treatment of marks as distinct from values. For the same reason, present host languages, such as COBOL and PL/1, are also of little help.

In the relational approach, one way to support marks by software is to assign a single extra byte to any column that is allowed to have applicable or inapplicable marks. This approach, adopted for A-marks in the IBM mainframe and mid-range relational DBMS products (DB2 and SQL/DS), appears to be fundamentally sound, although in these products some of the manipulative actions on marks should be cleaned up. Incidentally, criticisms of the way missing information is handled by SQL should not be interpreted or presented as criticisms of the relational model. Moreover, criticisms of SQL's treatment of missing information do not justify abandoning database nulls or marks.

The principal feature of RM/V2 pertaining to the way missing information is perceived by users is Feature RS-13 (see Chapter 2).

8.3 ■ Primary Keys and Foreign Keys of Base Relations

An important rule for relational databases is that, to maintain integrity, information about an unidentified (or inadequately identified) object is *never* recorded in these databases—a sharp contrast to non-relational databases. Thus, the declaration of exactly one primary key for each base relation is mandatory; it is not an optional feature. Moreover, the primary-key attribute is not permitted to include marks of either type (see Section 8.6). The pertinent RM/V2 feature is called *entity integrity* (Feature RI-3 in Chapter 13).

As an aside, the mere fact that such marks are prohibited from appearing in a column does not of itself make that column the primary-key attribute of a base relation. It is required that the catalog include an explicit declaration of the primary key of each base relation (see Feature RC-3 in Chapter 15).

A foreign key consists of one or more columns drawing its values from the domain (simple or composite), upon which at least one primary key is defined. In the case of composite foreign keys, it is possible that some, perhaps all, of the component values of a foreign key value are allowed to be A-marked (missing-but-applicable). This case needs special attention. Those *components of such a foreign key value that are unmarked should adhere to the referential-integrity constraint.* This detail is not supported in many of today's DBMS products, even when the vendors claim that their products support referential integrity.

I strongly recommend that database administrators or users consider very carefully the question of whether to permit or prohibit A-marks in foreign-key columns, and also that they document how and why that decision was made. Sometimes there will be a strong business case for prohibiting missing information altogether in foreign-key columns. However, reasons are presented in some detail in Section 4.3 for choosing to permit A-marks in these columns (see Features RB-33 and RB-34, the primary-key update operators). On the other hand, I-marks must be prohibited in all foreign-key columns in the entire database, because such a mark contradicts the foreign-key concept.

8.4 ■ Rows Containing A-marks and/or I-marks

According to Feature RI-12 in Chapter 13, any row containing nothing but A-marks and/or I-marks can and should be discarded by the DBMS from the relation in which it appears, no matter what the type of the relation. Such a row would be illegal in a *base relation,* because of the entity-integrity rule (see Section 8.6). Such a row does not bear information in any *derived relation,* whether it be a view, a query, a snapshot, or even an updated relation.

An external symbol is needed for the marks in several cases. Whenever such a symbol is needed, the following are suggested:

Type of Mark	A-mark	I-mark
External symbol	— or ??	!!

8.5 ■ Manipulation of Missing Information

Feature RM-10 in Chapter 12 is the RM/V2 feature of most importance with regard to manipulation of missing information. It calls for an approach that is uniform and systematic across the entire database. In particular, the approach is applicable to missing database values only, and must be independent of the data type of the missing information.

Features RM-11 and RM-12 in Chapter 12 specify the actions of arithmetic operators and concatenation on A-marked and I-marked values, respectively. Feature RJ-3 in Chapter 11 is an indicator that is turned on whenever a relational command encounters a missing db-value.

8.6 ■ Integrity Rules

There are two integrity rules that apply to every relational database:

1. Type E, *entity integrity*. No component of a primary key is allowed to have a missing value of any type. No component of a foreign key is allowed to have an I-marked value (missing-and-inapplicable).
2. Type R, *referential integrity*. For each distinct, unmarked foreign-key value in a relational database, there must exist in the database an equal value of a primary key from the same domain. If the foreign key is composite, those components that are themselves foreign keys and unmarked must exist in the database as components of at least one primary-key value drawn from the same domain.

A single instance of referential integrity is an example of an inclusion dependency. In the case of referential integrity, the set of distinct values in a foreign key must be a subset of the set of primary-key values drawn from the same domain. Casanova, Fagin, and Papadimitriou (1984) report interesting relationships between inclusion dependencies and functional dependencies.

It is important to observe that the entity-integrity and referential-integrity rules specify a *state* of integrity, not what *action* is to be taken by the system if an attempt is made to violate either rule. In the case of referential integrity, the DBMS should support *at least* three options:

1. refuse the command;
2. cascade the updates or deletes on the primary key values to all foreign keys defined on the same domain; **or**
3. replace each corresponding foreign-key value with an A-mark.

The DBMS rejects any attempts by users to replace each corresponding foreign-key value by an I-mark, since that would violate the second part of entity integrity.

This required choice of violation responses is the reason that the referential-integrity constraint should be supported in a general manner similar to that for user-defined integrity constraints (see Chapter 14), where a general choice of actions is also needed.

Finally, it should be possible for the DBA or any suitably authorized user to define additional special-purpose integrity constraints and the action to be taken if there is an attempted violation. These constraints are specific to the particular database involved.

If, as is usual, several columns take their values from a common domain, marks may occur in some of these columns and not in others. For example, a primary-key column is not permitted to contain any occurrences of either kind of mark, whereas (at the DBA's discretion) corresponding foreign-key columns (on the same domain) may be permitted to contain occurrences of the A-mark. Thus, declarations concerning whether a mark is permitted in or prohibited from a column should normally be associated with that column, not with the corresponding domain from which it draws its values.

8.7 ■ Updating A-marks and I-marks

A-marks and I-marks are treated differently from one another with regard to updating. This difference stems from the fact that an A-mark indicates that a value is at present unknown, whereas an I-mark indicates that a value is in some sense unknowable, given the present state of the micro-world being modeled.

An A-mark in column C may be replaced by any db-value that complies with the domain constraints and column constraints declared for column C; this replacement may be carried out by any user authorized to make updates in column C. Similarly, any db-value in a column for which marks are permitted can be replaced by an A-mark.

An I-mark in column C may be replaced by an A-mark or by any actual value, provided that:

- the DBMS finds that the user has the necessary extra authorization; **and**
- pertinent integrity constraints are satisfied.

See Section 14.3 for more details.

In Figure 8.2, "*" means that extra authorization is needed as required by Feature RA-9 (see Chapter 18). The authorization mechanism requires that the user who replaces any database value or A-mark by the I-mark must have special authorization for this action. Such authorization is also required for any change from an I-mark in the reverse direction.

The I-mark is strictly stronger than the A-mark. Any user who is authorized to update values in a column is thereby permitted to change any active value into an A-marked value, or vice versa. However, changing any non-missing value *directly* into an I-marked value or vice versa, requires special authorization (enforced by the DBMS), because that would be a direct attempt to violate the meaning of an I-mark.

In the examples already cited, changing a price from missing-and-applicable to a specified value is no threat to the integrity of the database. On the other hand, changing an employee's sales commission from missing-but-inapplicable to a specified value could damage database integrity. Thus, an I-mark is treated as if it were an integrity constraint of a special kind—namely, one applied to selected objects rather than selected object types.

8.8 ■ Application of Equality

What does it mean to assert that one missing-but-applicable value equals another? Is it appropriate to speak of the equality of two inapplicable values? In other words, under what circumstances does equality make sense? The RM/V2 position is that there are two kinds of equality of marks to be considered: (1) *semantic equality,* in which the meaning participates heavily, and (2) *symbolic (or formal) equality,* in which the meaning is ignored.

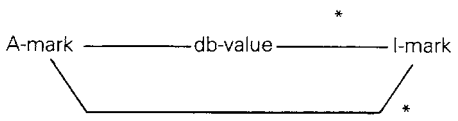
With regard to semantic equality, a factor that must be taken into account is how applicable and inapplicable values are expected to be used. Their uses are quite different in nature. In fact, the truth-value of

A-mark = I-mark

is FALSE with respect to both types of equality.

How about equality between any two occurrences of the *same type of mark?* Since the symbol is the same in both cases, the two occurrences are symbolically equal. The question of semantic equality, however, needs more detailed investigation, and is discussed in the next section.

Figure 8.2 **State Diagram Specifying Permitted Updates**



8.8.1 Missing-but-Applicable Information

Missing-but-applicable information presents the opportunity to ask what might be true if one or more missing values were to be temporarily replaced by actual values. Frequently, "what-if" databases must be developed and manipulated separately from the so-called operational databases. This occurs because the former represent *what might be the case* if certain events were to take place in the future (in the business or in its environment), while the latter represent *reality*. Accordingly, updates in the "what-if" databases must be regarded as representing conceptual actions (analytical, planning, or projecting into the future). An important advantage of the A-marks is that some of the analysis can be carried out directly on the operational data without making any conceptual updates.

Suppose a database includes information about employees, including each employee's birthdate. Suppose also that birthdate is one of the immediate, single-valued properties of an employee that is allowed to be temporarily missing for one or more employees. It is quite possible that when an employee's birthdate is unknown, the actual value of this date may (eventually) prove to be *any* date that lies within the range of employee ages permitted by law and by company policy. Such a range of dates would be specified as a formula based on a variable representing the date of the current day. This formula would be included in the catalog declarations for the specific column (quite likely) or in the catalog declarations for the domain from which this column draws its values (less likely).

In this case, the set of possible values is quite large. In general, however, *whether the set of possible values for a property is large or small,* there must be at least two possibilities—otherwise, the property's value would be known. It would therefore be a mistake to expect the value TRUE when evaluating a logical condition that involves semantically comparing either one missing-but-applicable value with another or one such missing value with a known or specified value. For example, what is the truth value of the inequality

BIRTHDATE > 66-1-1

for a missing birthdate? It is clearly neither TRUE nor FALSE. Instead, it can be said to be MAYBE (meaning maybe true and maybe false; the DBMS does not know which holds). When focusing on the domain of truth-values, the logical truth-value MAYBE can be thought of as a value-oriented counterpart for the A-mark.

When represented by A-marks, two missing values possess marks that match one another symbolically, but not necessarily semantically. As time advances, the database is subjected to commands that modify the data. Thus, users and/or programs may eventually replace these two marks by different values.

8.8.2 Inapplicable Information

A natural subsequent question is, "Must the systematic treatment of inapplicable values cause an additional extension of the underlying three-valued logic to a four-valued logic?" Such an extension is logically necessary, and it now seems appropriate to introduce it as part of the relational model.

At first glance, it appears to make sense to handle equality between two inapplicable-value marks just like equality between two actual values. Note, however, that I-marks are neither values nor placeholders for values. They mean *unknowable* rather than *unknown.* Thus, within the condition part of a relational-language statement, whenever an I-mark is equated to an actual value, an A-mark, or an I-mark, the truth-value of such a condition is always taken to be MAYBE of type INAPPLICABLE.

8.9 ■ The Three-Valued Logic of RM/V1

A database retrieval may, of course, include several conditions like BIRTHDATE > 66-1-1, and the conditions may be combined in many different logical combinations, including the logical connectives AND, OR, NOT and the quantifiers UNIVERSAL and EXISTENTIAL. (See the explanation at the end of Section 4.2, dealing with relational division, and the works listed in the predicate logic part of the Reference section.)

Suppose, as an example, that a second immediate property recorded for each employee is the present salary of that employee. Suppose also that this column is allowed to have missing db-values. How does the DBMS deal with a query involving the combination of conditions

(BIRTHDATE > 66-1-1) $\vee$ (SALARY < 20,000),

where either the birthdate condition or the salary condition or both may evaluate to MAYBE? Clearly, the DBMS must know the truth-value of MAYBE or TRUE, TRUE or MAYBE, and MAYBE or MAYBE.

From this it can be seen that *there is a clear need in any systematic treatment of missing values to extend the underlying two-valued predicate logic to at least three-valued predicate logic.*

In the following truth-tables for the three-valued logic of RM/V1, P and Q denote propositions, each of which may have any one of the following truth-values:

t for true *or* m for maybe *or* f for false.

The truth values t, m, f are actual values, and should not be confused with marked values or the MAYBE qualifiers (see Table 8.1).

In the relational model, the universal and existential quantifiers are applied over finite sets only. Thus, the universal quantifier behaves like the logic operator AND, and the existential quantifier behaves like OR. Both

Table 8.1 **The Truth Tables of Three-Valued Logic**

P	not P
t	f
m	m
f	t

P ∨ Q	Q = t	Q = m	Q = f
P = t	t	t	t
P = m	t	m	m
P = f	t	m	f

P ∧ Q	Q = t	Q = m	Q = f
P = t	t	m	f
P = m	m	m	f
P = f	f	f	f

operators are extended to apply the specified condition to each and every member of the pertinent set.

When an entire condition based on three-valued, first-order predicate logic is evaluated, the result can be any one of the three possibilities TRUE, MAYBE, or FALSE. If such a condition is part of a query that does not include the MAYBE option, the result consists of all the cases in which this condition evaluates to TRUE, and no others.

If in this query the keyword MAYBE is applied to the whole condition, then the result consists of all the cases in which this condition evaluates to MAYBE, and no others. This qualifier is used only for exploring possibilities; special authorization would be necessary for a user to incorporate it in one of his or her programs or in a terminal interaction.

One problem of which DBMS designers and users should be aware is that in rare instances the condition part of a query may be a tautology. In other words, it may have the value TRUE no matter what data is in the pertinent columns and no matter what data is missing. An example is the following condition pertaining to employees (where B denotes BIRTHDATE):

$$(B < 66\text{-}1\text{-}1) \vee (B = 66\text{-}1\text{-}1) \vee (B > 66\text{-}1\text{-}1).$$

However, if the DBMS were to apply three-valued logic to each term and it encountered a marked value in the birthdate column, each of the terms in this query condition would receive the truth-value MAYBE. MAYBE OR MAYBE yields the truth-value MAYBE. Thus, the condition as a whole evaluates to MAYBE, which is incorrect, but not traumatically incorrect.

There are two options:

1. warn users not to use tautologies as conditions in their relational-language statements (tautologies waste the computer's resources);
2. develop a DBMS that examines all conditions not in excess of some clearly specified complexity, and determines whether each condition is a tautology or not.

Naturally, in this latter case, it would be necessary to place some limitation on the complexity of each and every query, because with predicate logic the general problem is unsolvable. In my opinion, option 1 is good enough for now because this is not a burning issue.

8.10 ■ The Four-Valued Logic of RM/V2

Consider an example of a combination (either by AND, or by OR) of two logical conditions used in selecting employees:

(birthdate > 50-1-1) AND/OR (commission > 1000).

Suppose that, for a particular employee, the first condition evaluates to the truth-value missing-and-applicable and the second to missing-and-inapplicable. What is the truth-value of the whole condition?

Clearly, the truth tables of four-valued logic must be examined. In the following tables (Table 8.2), t stands for true, f for false, i for *missing-and-inapplicable*, and a for *missing-and-applicable*. Note that t, a, i, f are actual values, and should not be confused with marked values.

Note that we obtain the truth tables of the three-valued logic by replacing i by m *and* a by m (where m simply stands for missing, and the reason for anything being missing is ignored). It should be clear that four-valued logic is more precise but more complicated than three-valued logic. Four-valued logic is selected as an integral part of RM/V2 (see Feature RM-10 in Chapter 12).

If a DBMS vendor feels that the extra complexity of four-valued logic is not justifiable at this time, the external specifications of its DBMS product should permit expansion at a later time from three-valued to four-valued logic support without affecting users' investment in application programming, or with only a minimal impact. If four-valued logic is built into a DBMS product initially or as an extension, either it should agree with the four-valued logic just described, or its departures should be defended in writing from a technical and practical standpoint.

Table 8.2 **The Truth Tables of Four-Valued Logic**

P	not P
t	f
a	a
i	i
f	t

P	P ∨ Q	Q: t	Q: a	Q: i	Q: f
t	t	t	t	t	t
a	a	t	a	a	a
i	i	t	a	i	f
f	f	t	a	f	f

P	P ∧ Q	Q: t	Q: a	Q: i	Q: f
t	t	t	a	i	f
a	a	a	a	i	f
i	i	i	i	i	f
f	f	f	f	f	f

A repetition of the warning about multi-valued logics that I included in [Codd 1986a, 1987a] may be appropriate here. Such logics can yield the truth-value MAYBE for an expression that happens to be TRUE because it happens to be a tautology. For example, find the employees whose birth year is 1940 or prior to 1940 or after 1940. Every employee should be assigned the value TRUE for this condition, even if his or her birth year happens to be missing! This warning applies to other multi-valued logics. It may be necessary in the future for DBMS products to be equipped with detection algorithms for simple tautologies of this kind.

8.11 ■ Selects, Equi-joins, Inequality Joins, and Relational Division

The manner in which algebraic **selects, equi-joins, inequality joins,** and **relational division** treat A-marks and I-marks is determined by the semantic treatment of equality described in Section 8.8. An **inequality join** is a special kind of join using the inequality comparator NOT EQUAL TO.

Thus, whenever an **equi-join** involves comparing two items for equality, and either just one of them is a mark or both of them are marks of the same type (both A or both I), the pertinent rows are glued together *if and only if* the MAYBE qualifier has been specified.

Suppose that a query Q does not include the MAYBE qualifier. Then, executing Q delivers only those cases in which the condition part of Q evaluates to TRUE. To obtain a result from this query that includes all the TRUE cases and all the MAYBE cases, it is necessary to apply the **union** operator: Q **union** (Q MAYBE).

8.12 ■ Ordering of Values and Marks

Ordering should be handled in a manner similar to that described for equality. There are two kinds of ordering to be considered: *semantic ordering* and *symbolic ordering*. The semantic version applies when using a less-than condition or a greater-than condition in a statement of a relational data sublanguage. The symbolic version applies when using the ORDER BY clause (e.g., to determine how a report is to be ordered). Let us consider symbolic ordering first.

The present ordering as implemented in DB2 in the ORDER BY clause of SQL involves nulls (i.e., A-marks) representing missing-and-applicable values. (The case of inapplicable values is not yet handled at all by the language SQL or by the DB2 system.) DB2 places nulls at the high end of the value-ordering scale. In order to be compatible with this ordering, A-marks are placed at the high end, immediately after values. On top of A-marks are the new I-marks. This is the *symbolic (or formal) ordering.*

Note that inapplicable information in a particular column *could* be supported in extended SQL by requiring the DBA or a suitably authorized user to declare a value from the pertinent domain as the one to represent such inapplicability for a given column. I definitely advocate, however, that this approach *not* be taken. For one thing, this approach would implicitly and potentially define as many different orderings for the missing db-values relative to the existing db-values as there are columns that are allowed to have missing db-values. Instead, I believe it is more systematic and more uniform across different data types to use a special mark (the I-mark) because these marks are not database values.

Now let us consider the *semantic ordering*. The truth-value of each of the following expressions is MAYBE (not TRUE)

(db-value < mark) , (mark < db-value) , (mark < mark)

for any type of mark and any db-value. The same applies to these expressions if the symbol "<" is replaced by the symbol ">". If such an expression involves either one or two occurrences of marks, the truth-value of the expression is MAYBE.

There has been some criticism that the symbolic ordering of marks relative to values runs counter to the semantic ordering and the application of three- or four-valued logic. I fail to see any problem, however, because the use of truth-valued conditions involving ordering when applying a relational data sublanguage is at a higher level of abstraction than the use of the ordering of marks relative to values in the ORDER BY clause of a relational command (see Feature RQ-7 in Chapter 10).

8.13 ■ Joins Involving Value-ordering

Joins involving the comparators

LESS THAN OR EQUAL TO
LESS THAN
GREATER THAN OR EQUAL TO
GREATER THAN

treat applicable and inapplicable marks as determined by the usual orderings of db-values and the semantic ordering of marks defined in the immediately preceding section. If the MAYBE qualifier does not accompany the request for a **join,** then (as usual) only those items are generated for each of which the entire pertinent condition has the truth-value TRUE. On the other hand, if the MAYBE qualifier is applied to the entire condition part in the command, only those items are generated for each of which the entire pertinent condition has the truth-value MAYBE.

Consider a database that includes two relations T1 and T2 that describe events of type 1 and type 2, respectively; the description includes the date of occurrence of each event. Suppose that a request is made that involves pairing off events of type 1 with events of type 2, provided the type 1 event occurs before the type 2 event. Such a pairing activity can be expressed in terms of a **join** of T1 on the date of the type 1 event with T2 on the date of the type 2 event using the comparator <.

For the sake of simplicity, suppose that the extensions of T1 and T2 are as follows:

T1

E#	EDATE	. . .
e1	88-02-14	
e2	86-03-22	
e3	—	(A-marked)
e4	88-12-27	

T2

V#	VDATE	. . .
v1	86-03-13	
v2	89-01-27	
v3	87-08-19	

Also suppose that two requests are as follows:

T ← T1 [EDATE < VDATE] T2
T″ ← T1 [EDATE < VDATE] T2 MAYBE.

Then, the derived relations T and T″ are as follows:

T

E#	EDATE	V#	VDATE
e1	88-02-14	v2	89-01-27
e2	86-03-22	v2	89-01-27
e2	86-03-22	v3	87-08-19
e4	88-12-27	v2	89-01-27

T″

E#	EDATE	V#	VDATE
e3	—	v1	86-03-13
e3	—	v2	89-01-27
e3	—	v3	87-08-19

8.14 ■ Scalar Functions Applied to Marked Arguments

In this context, a *scalar function* is a function that transforms scalar arguments into a scalar result. Consider the effect of such a function when one or more of its arguments is marked.

In general, if the strongest mark on one of its arguments is I, then the scalar result is I-marked. If, on the other hand, the strongest mark is A, then the scalar result is A-marked.

For example, let @ denote any one of the arithmetic operators +, −, ×, /, and let z denote an unmarked scalar argument. Then:

z @ a = a z @ i = i a @ z = a i @ z = i
a @ a = a a @ i = i i @ a = i i @ i = i

The functions NEGATION, OR, and AND are not exceptions to this general rule because of the distinction (noted in Section 8.9) between the truth values a and i, on the one hand, and marked truth values (A-marked and I-marked), on the other.

8.15 ■ Criticisms of Arithmetic on Marked Values

Occasionally one must be careful when asking a computer to carry out ordinary arithmetic: One should not request it to divide any number by zero. For example, if each customer can make several partial payments instead of one lump sum, one might ask a question such as "What is the average payment made by each customer?" This seemingly innocent question can cause the machine to complain every time a customer is encountered who has made no payments at all, because it is then being asked to divide zero by zero.

The same kind of remark holds for arithmetic operations upon values from columns in which the value-inapplicable mark may occur. Suppose the employee relation contains two columns:

1. the total salary earned to date;
2. the commission earned to date.

Suppose also that for some employees the commission is marked inapplicable. Consider the request "What is the total income earned to date for each employee?", where total income is equal to salary (always applicable) plus commission (if applicable). If this is carelessly expressed as salary plus commission, the answer for any employee who has an inapplicable commission is inapplicable, using the table for addition shown on page 174. What is needed in this case is that the amount added for commission should be zero when the commission is applicable.

The first reaction often heard is that, instead of the commission being marked inapplicable, it should be set to zero. One problem this action would cause is that the DBMS might not (and very probably would not) be able to distinguish between a case in which commission was inapplicable and a case in which it was applicable but the employee had actually earned zero commission to date. Moreover, in this case virtually every integrity constraint that involved the inapplicable state for a value in the commission column could not be expressed.

A second claim often heard is that the addition table is incorrect, and the entry x + I = I should be replaced by x + I = x. While this might be appropriate for this example, consider a second example. Suppose one requested Q1, the average commission earned by employees, when one really intended to request Q2, the average commission earned by those employees entitled to earn commissions. If there do exist some employees who are entitled to earn commission, and if the cases of commission being inapplicable are each represented by the value zero, execution of Q1 delivers an incorrect result without any alarm from the computer. If, however, the cases of inapplicability are represented by I-marks, then execution of Q1 delivers an I-mark and that will alert the user to his or her folly.

8.16 ■ Application of Statistical Functions

In applying a statistical function to the db-values in one or more columns of a relation, it is desirable to be able to specify how A-marks and I-marks are to be treated by this function, if it should encounter either type of mark. A practical approach is to support two temporary replacements: one for the A-mark occurrences and another for the I-mark occurrences.

A convenient way of expressing the replacement action is by means of two separate, single-argument functions: AR (which stands for A-mark replacement) and IR (I-mark replacement). In each case, the single argument is a scalar constant or a scalar function that operates upon other component values in the row being examined and delivers a scalar value. The pertinent features of RM/V2 are RQ-4 and RQ-5 (see Chapter 10).

The function or qualifier specifying the replacement is called the *substitution qualifier.* This qualifier is applicable to every kind of statistical function. However, if the statistical function has two or more arguments and these are applied to two or more columns, the specified replacement action must apply to all of these attributes.

An example of practical use of this qualifier is the calculation of a salary budget for each department based on the present salary of each member of a department. If a few salaries are missing (and therefore A-marked), one may wish to compute the total for each department by requesting that each A-mark occurrence be replaced temporarily by the maximum salary of those persons known by the DBMS to be members of that department.

In certain special cases, the two replacements may be values equal to one another. In certain other special cases, one or both of the replacements may be a mark identical to the mark being replaced. The need for this case is determined by the default action for omitted substitution qualifiers being specified as "ignore marks of the corresponding type."

Note that the state of the database is not changed by the execution of any one of these statistical functions alone. In other words, the substitutions replacing marks by values or by marks are in effect during, and only during, the execution of the pertinent statistical function. One advantage of making these substitutions temporary is that certain kinds of possibilities can be investigated without setting up a separate "what-if" database.

If the substitution qualifier is omitted altogether from a statistical function request, the DBMS would assume that *only the unmarked values should contribute to the result.* On the other hand, if the existence of any occurrence of a mark of type q in the operand is to yield an A-mark as the result, there must be a qualifier in the command requesting that marks of type q be detected, but not modified in the temporary substitution sense (i.e., a mark of type q should be replaced by itself).

Notice that any replacement action specified in this way is a replacement of a marked *argument* of the function, not of any *result* the function might deliver. Moreover, the specified scalar replacement(s) must be values be-

longing to the domain from which that column draws its values, and must comply with any additional constraints that have been declared for that specific column.

Finally, any specific replacement action applies to just one of the columns cited in the retrieval or update command. Of course, several of the columns cited may be subject to replacement actions. In general, if N columns are cited in a relational command, there may be as many as $2N$ replacement actions specified in that command, two for each distinct citation.

Thus, a single pair of occurrences of replacement qualifiers (one for I-marked values, one for A-marked values) for each command is generally inadequate. It should be replaced by a pair of replacement qualifiers for each column cited in any pertinent command. The syntax must allow one pair to be specified for each statistical function cited, and unambiguously associate that pair with the pertinent function.

Little has been said about the results generated by the scalar and aggregate functions discussed in this section and Section 8.17. Feature RF-8 in Chapter 19 specifies that marked values are not generated by scalar and aggregate functions when acting upon unmarked arguments.

8.17 ■ Application of Statistical Functions to Empty Sets

This issue, raised by critics, is not directly related to the subject of missing values. Nevertheless, I touched upon it in Section 2.5 of [Codd 1986a] because SQL happens to generate null as the result of applying certain statistical functions (such as AVERAGE) to an empty set. Since the null of SQL was introduced to denote the fact that a db-value is unknown, it is an unwise choice now to mean something entirely different—namely, that an arithmetic result is undefined. This topic is clarified in more detail here, but only with respect to the relational model, not SQL.

Section 9.5 deals with the case of applying a statistical function to a collection of sets, some of which are empty and some non-empty. We must treat the extreme case where all the sets are empty (even if there is only one set in the collection of sets), and in such a way that all these cases behave in a consistent way. As a first step, an initial value of zero must be established immediately before the evaluation of the pertinent function against the specified sets.

If the empty-set qualifier is omitted from a command, each occurrence of an empty set is ignored. In addition to the value returned, however, there must be a trigger (known as the *empty trigger*) that is turned on whenever at least one set encountered in the execution of this command is empty.

Suppose that the *value* returned, whenever a statistical function is applied to a single empty set, is the initial value just cited, that is, zero. A special case needs careful attention to avoid misinterpretation of the value returned. Whenever (1) a statistical function is cited in a command, (2) this function

(for example, AVERAGE) happens to require dividing by the number of elements in the pertinent set, and (3) the value returned by the function is zero for one or more of the sets, then it is normally necessary to examine each of these sets for its possible emptiness. Such an examination would distinguish the empty-set case from the case in which the statistical function happened to generate zero from elements actually encountered in the set. The reader should remember that the burden of this extra examination arises from ordinary integer arithmetic, in which dividing by zero is unacceptable. The burden of this extra examination is therefore *not* a consequence of the relational model.

Moreover, the relational model is consistent with elementary arithmetic. Every DBMS based on that model should also be consistent with elementary arithmetic.

Marks in the relational model are intended to represent the fact that information (more precisely, a db-value) is missing, and should be sharply distinguished from the case in which the value of a function (such as arithmetic division) is undefined.

8.18 ■ Removal of Duplicate Rows

Unfortunately, in many present releases of relational DBMS products, derived relations (often loosely called tables) are corrupted by leaving duplicate rows in them unless the user appends an explicit qualifier requesting that these duplicate rows be removed. A common example of this problem occurs in a corrupted projection that does not happen to include the primary key of the operand relation.

Although it is possible in SQL for the user to specify explicitly that all but one occurrence of any duplicate rows be removed, the user can choose to retain duplicate rows because he or she is unaware of the consequences. Users should not be burdened with this choice, and the DBMS optimizer should not be impaired by permitting duplicate rows under any circumstances.

If two or more rows happen to contain the same actual values and no marks (applicable or inapplicable), the removal of duplicate rows is obvious. The interesting case for this chapter is that in which some of the values are missing.

In this case, suppose that a typical pair of row components for which equality is to be tested is $< x,y >$. Then, it seems reasonable to assert that two rows are duplicates of one another, if one of the following conditions is satisfied by every pair tested:

1. x and y are actual values and $x = y$, **or**
2. one of the pair is marked and the other is not, **or**
3. both x and y are marked, and the marks are *symbolically equal* (i.e., both values are A-marked or both are I-marked),

and if condition 1 is satisfied by at least one tested pair of components. When duplicate rows are discovered by the DBMS, it should remove all but one occurrence of the duplicate rows.

As an example, consider a relation EMP that identifies and describes employees:

EMP	(EMP#	ENAME	DEPT#	SALARY	H_CITY)
	E107	Rook	D12	10,000	Wimborne
	E912	Knight	— A	12,000	Poole
	E239	Knight	— A	12,000	Poole
	E575	Pawn	D12	— A	Poole
	E123	King	D01	15,000	Portland
	E224	Bishop	— A	— A	Weymouth

"— A" denotes a missing-and-applicable value. Suppose the relation E is derived from EMP by true projection (not the corrupted variety) onto DEPT# and SALARY:

E ← EMP [DEPT#, SALARY].

Before and after the removal of duplicate rows and empty rows, the following information is derived (the user sees only the AFTER version):

BEFORE → AFTER

E'	(DEPT#	SALARY)
	D12	10,000
*	— A	12,000
**	— A	12,000
	D12	— A
	D01	15,000
***	— A	— A

E	(DEPT#	SALARY)	
	D12	10,000	
	— A	12,000	#
	D12	— A	##
	D01	15,000	

The rows labeled "*" and "**" are treated as duplicates of one another because the corresponding components in these rows constitute the following:

- pairs of equal db-values, and *there exists at least one pair of this type;*
- pairs in which a db-value is accompanied by a missing value.

The row labeled "***" is removed because it consists of nothing but missing values. Note that the rows marked "#" and "##" are not treated as duplicates because of the lack of at least one pair of corresponding components that have equal values.

There has been some criticism of the fact that this scheme for removal of duplicate rows does not conform to the *semantic notions of equality* described in Section 8.8. I fail to see any problem, however, because the

semantic notions of equality are applicable at a higher level of abstraction than the symbolic equality involved in removal of duplicate rows.

8.19 ■ Operator-generated Marks

The introduction of a new column C for a selected base relation R is achieved by appending to the catalog a description of this column. This causes the DBMS to record in R itself an A-mark in column C for each row in R. It does not make sense to record the I-mark because it is senseless to assert that every value in a column is unknowable—in that case, why have the column at all?

The operators **outer join** and **outer union** are capable of generating derived relations in which some of the columns have one or more missing db-values. Which type of mark should the DBMS generate? It seems reasonable to generate A-marks only. If a suitably authorized user believes that I-marks are needed instead, he or she will have to replace some A-marks by I-marks.

Note that A-marks are weaker than I-marks in that a user requires no special authorization beyond the usual update authorization if he or she wishes to update an A-mark into a db-value (see Section 8.7).

Moreover, it is easier for the user to delve into "what-if" kinds of interactions on the relation wherever A-marks occur, since in these cases he or she need not get special authorization beyond that for querying.

The question has been raised of why those operators that are capable of generating new marks in the result create only A-marks, never I-marks. (In this context, "new marks" mean marks not simply copied into the result from one or other of the operands.) The answer is that A-marks are preferred because they are the weaker and more flexible of the two types. Hence, they are more readily changed by users, without needing any special mark-type-change authorization.

8.20 ■ Some Necessary Language Changes

Here several minor language aspects are covered together, even though most are discussed elsewhere in this book. An example is the use of the MAYBE qualifier on a condition, whenever only those items are needed for which this condition evaluates to MAYBE. Note that, in order to support this qualifier, the DBMS must be able to handle either three-valued or four-valued logic (including the truth tables).

Moreover, if the items X are needed for which the condition K evaluates to either TRUE or MAYBE, then a command such as

(X where K) ∪ (X where K MAYBE)

should be used. Further, if the DBMS supports four-valued logic, then two additional qualifiers, MAYBE_A and MAYBE_I, are needed to specify for a given truth-valued expression which of the truth-values a and i are to replace the truth-value t as the truth-value that qualifies values to be retrieved. MAYBE_A means maybe true, maybe false, but certainly applicable, while MAYBE_I means neither true nor false, but inapplicable.

Also note that, because the MAYBE qualifiers apply to conditions that may involve negation, OR, AND, the existential quantifier, and the universal quantifier, they require that the DBMS handle four-valued logic internally, and *not* put that burden on users, as does the present version of SQL. The MAYBE qualifiers are described in Features RQ-1–RQ-3 in Chapter 10.

In the following discussion, changes in language are expressed as changes to SQL. It should be a simple matter to adapt them to any other reasonably complete, relational data sublanguage.

It is necessary to be able to refer to marks that are similar to the way SQL presently refers to nulls, except that the user should be allowed to distinguish between the two types of marks when he or she wishes to do so.

One user-friendly solution that is *not* being advocated is to introduce the clauses shown in Table 8.3 to refer to the presence and absence of an A-mark, an I-mark, or either type of mark (in case the user does not care which type of mark is involved). These clauses are not part of RM/V2 because the MAYBE_A, MAYBE_I, and MAYBE qualifiers of Features RQ-1–RQ-3 are more powerful. The SQL clauses IS NULL and IS NOT NULL should be abandoned swiftly.

Here is an example using the clauses listed in Table 8.3:

1. find the employees who are eligible to receive sales commissions; **and**
2. find the employees who are ineligible to receive sales commissions.

In many companies, query 1 is much more likely than query 2, because usually only a minority of employees are eligible for such commissions. In pseudo-SQL, appropriate statements for these queries would be as follows:

1. SELECT serial_number FROM employees
   WHERE commission IS NOT I-MARKED
2. SELECT serial_number FROM employees
   WHERE commission IS I-MARKED

Table 8.3 **Possible Clauses for Simple Conditions**

Type	Presence of Mark	Absence of Mark
A-mark	Is A-marked	Is not A-marked
I-mark	Is-I-marked	Is not I-marked
Either	Is missing	Is not missing

Additional needs are the substitution qualifiers AR and IR, when applying a statistical function to any column in which marks may occur (see Section 8.16), and the empty set qualifier ESR, when applying a statistical function to a collection of sets, some of which may be empty (see Section 9.5).

8.21 ■ Normalization

The concepts and rules of functional dependence, multi-valued dependence, and join dependence were developed without considering missing db-values. Early papers on functional dependence were [Codd 1971b and 1971c]. A comparatively recent paper on these dependencies is [Beeri, Fagin, and Howard 1977].

All the normal forms based on these dependencies were also developed without considering missing db-values. Does the possible presence of marks in some columns (each mark indicating the fact that a db-value is missing) undermine all these concepts, and theorems based on them? Fortunately, the answer is no: a mark is not itself a db-value. More specifically, a mark in column C is semantically different from the db-values in C. Thus, the normalization concepts *do not* apply and *should not* be applied globally to those combinations of columns and rows containing marks. Instead, they should be applied as follows:

- the normalization concepts should be applied to a conceptual version of the database in which rows containing missing-but-applicable information in the pertinent columns have been removed;
- these concepts should also be applied when any attempt is made to replace a mark by a db-value.

When an attempt is made to insert a new row into a relation and a certain component db-value is missing, it is pointless for the system to base acceptance or rejection of this row on whether the missing db-value *does meet or might meet or fails to meet* certain integrity constraints based on a dependence in which the pertinent column is involved. The proper time for the system to make this determination is when an attempt is made to replace the pertinent mark by an actual db-value.

One might be tempted to treat I-marks differently from A-marks. One or more users, however, may be authorized to replace an I-mark by a db-value. Thus, all marks should be treated alike, regardless of type, in the matter of testing any dependence constraint, whether it be functional, multi-valued, join, or inclusion. For every row that contains a mark in the column or columns being tested, the DBMS should wait until an attempt is made to replace the marked item(s) by an actual db-value.

A fully relational DBMS should have the capability of storing (in its catalog) statements defining the various kinds of dependencies—including

the functional, multi-valued, join, and inclusion types of dependencies—as they apply to the particular database being managed.

A program should also be available to deduce all the dependencies that are a consequence of those supplied by the DBA or other suitably authorized users, so that, when attempting to change a mark into a db-value, no dependence that is logically implied by others is overlooked by the DBMS. Further, such a DBMS should be able to check the database against one or more of these integrity constraints whenever necessary, and without explicit invocation by an application program. In general, such checking is likely to be necessary whenever a mark is replaced by a db-value.

Exercises

8.1 Does the relational model use a specially reserved numeric value to represent the fact that a numeric database value is missing? Does it use a specially reserved character-string to represent the fact that a character-string database value is missing? Give reasons for your answers.

8.2 In RM/V2, what are the two main reasons for values being missing from a database? Is the NULL term in SQL capable of distinguishing between these reasons? If your answer is yes, explain how this would be accomplished by means of two examples.

8.3 The comment is frequently made that "nulls are a headache; who needs them?" Take a position on nulls (marked values) and three- or four-valued logic versus special values reserved by users to mean that a value is missing. Now defend that position from the viewpoints of technical soundness and usability by the community of users.

8.4 Do the three-valued logic in RM/V1 and the four-valued logic in RM/V2 preserve the commutativity of AND and OR?

8.5 Supply the truth table for four-valued logic.

8.6 The MAYBE qualifiers apply to (1) the whole condition part of a query, (2) a truth-valued expression, and (3) part of a query that refers to just a single column. Which of these represents the generality of scope most accurately? Give one example of the use of the MAYBE_A qualifier, and one example of the use of the MAYBE_I qualifier.

8.7 Which of the three alternatives in Exercise 8.6 represents the generality of scope of the IS NULL phrase in present versions of the language SQL?

8.8 You are applying a statistical function to a column that is allowed to contain missing values. You want each missing value to be ignored. How is that accomplished?

8.9 You are applying a statistical function to a numeric column that is allowed to contain missing values. You want each A-marked value to be temporarily replaced by 399, and each I-marked value by 0. How is that accomplished?

8.10 The function AVERAGE is being applied to a numeric column in relation S. S happens to be empty. What kind of result is delivered by the relational model? By SQL?

■ *CHAPTER 9* ■

Response to Technical Criticisms Regarding Missing Information

There has been some justified technical criticism of the treatment of missing information in the data sublanguage SQL. Some of this criticism has been directed by mistake at the relational model [Codd 1986a, 1987c].

As explained in Chapter 1, it is important to distinguish between technical criticisms of the model, on the one hand, and of the implementations and products based on that model, on the other. With respect to the treatment of missing information, technical criticisms have strayed across the boundary without proper justification. I shall discuss criticisms that have appeared in recent technical articles, along with a counter-proposal called the *default value scheme* (for brevity, DV). I devote an entire chapter to dealing with these criticisms for two main reasons. First, the way the relational model deals with missing data appears to be one of its least understood parts. Second, discussion of these criticisms may help readers understand the approach and why it was adopted.

9.1 ■ The Value-oriented Misinterpretation

The representation in IBM relational DBMS products of *missing database values* in any column by means of an extra byte seems correct. In the IBM manuals, the corresponding marks are sometimes called *nulls* and sometimes called *null values*. Few of the ways in which these products process missing information, however, conform to Version 1 of the relational model. There are numerous cases in which the processing of nulls (more specifically,

A-marks) in the IBM product DB2 is non-systematic. These cases, however, should not be construed as criticisms of the relational model itself.

Any approach to the treatment of missing information should consider *what it means* for a db-value to be missing, including how such occurrences should be processed. A basic principle in the relational model is that the treatment of all aspects of shared data in databases is not just a representation issue. There are always other considerations which the DBMS must handle, especially the approach to manipulating the data and the preservation of database integrity.

It is quite inappropriate to leave these considerations to be handled by users in a variety of ways, and buried in a variety of application programs. This principle applies just as forcefully to missing information. Thus, *missing information is not just a representation issue.*

9.2 ■ The Alleged Counter-intuitive Nature

Consider the following examples: (1) suppliers in London and (2) suppliers not in London, where the CITY column for suppliers is allowed to have missing-but-applicable indicators (A-marks). One criticism of the relational model is that it requires the user to make the distinction between (1) "suppliers *known to the system* not to be in London" and (2) "suppliers not in London."

It has been asserted that this distinction is subtle and likely to mystify the user. Subtlety, however (like beauty), is in the eyes of the beholder. What is more important is that a user, in failing to make this very distinction, may cause serious errors, errors that could have serious consequences for the user's business.

In order to comment on the *default value scheme* (DV) [Date, 1986] for representing missing information, it is necessary to describe that approach first. In this scheme, if items of data are allowed to be missing in a column C, it is left to one or more users to declare that a particular value in C denotes the fact that a datum is missing in C. There is no constraint that all columns, in which missing values are permitted, must use the same representation of the fact that a value is missing. Moreover, the user who declares the "default value" for column C is expected to embody in his or her application program the method by which any missing values in column C are to be handled.

To return to the discussion of "suppliers known to the system not to be in London" and "suppliers not in London," this distinction may well be judged subtle by some users. However, even if the default value scheme were adopted, this would not prevent or help prevent the occurrence of the type of error in which the user fails to make this distinction.

Let us look at the example in more detail, demonstrating how the DV scheme constitutes a non-solution to the problem. Consider a relation S identifying suppliers and describing their immediate, single-valued properties. Let one of these properties be the city in which the supplier is based. A sample snapshot follows:

S	(S#	SNAME	CITY	. . .)
	s1	JONES	LONDON	. . .
	s2	SMITH	BRISTOL	. . .
	s3	DUPONT	v	. . .
	s4	EIFFEL	PARIS	. . .
	s5	GRID	v	. . .

"v" denotes a character string, declared in the catalog to be the "default value" for the column CITY in the relation S, which in the DV scheme means "unknown" or "missing" for this column only. Note that v may be *any* character string that does not represent the actual name of any existing city (e.g., "???" or "XXX").

Now consider these two queries:

Q1: Find the suppliers in London

Q2: Find the suppliers NOT in London

If these queries are represented in a relational language (such as ALPHA [Codd 1971a], SQL [IBM 1988], or QUEL [Relational Technology 1988]), ignoring the occurrences of v, and therefore ignoring the occurrences of missing db-values, the answer to Q1 would be s1. This answer is correct only if interpreted as those suppliers *known by the system* to be in London, since at any later time an occurrence of v may be updated to "LONDON." Q2 would similarly yield the set (s2, s3, s4, s5), which is *definitely incorrect* when interpreted as the suppliers known by the system not to be in London, and *potentially incorrect* when interpreted as the suppliers actually not in London.

Thus, the user of a DBMS equipped with the DV scheme *must* take into account whether a column is allowed to contain missing values, shaping the query accordingly, and differentiating in his or her thinking among (1) what is known to the system, (2) what is actually a fact, and (3) what could be the case. This requirement of the DV approach to missing information forces the user to make the very same distinctions for which the relational approach to missing information has been criticized.

The burden on the user of having to make these distinctions is not removed by having him or her formulate the query as "find the suppliers not based in London and not based in '???'. " In fact, the burden arises because the problem of dealing with missing information correctly is just not a simple problem.

The claim that the DV approach "avoids all the difficulties associated with the null value scheme" [Date, 1986] is clearly incorrect. I would characterize the DV scheme as an approach that is likely to entice the naive user and whose claimed simplicity is quite likely to trap the unwary and give rise to serious mistakes.

Finally, consider the idea that a notion should be rejected because it is "counter-intuitive." This is a type of criticism that I cannot accept as technical in nature, precisely because it is too subjective with respect to a person and the culture and era in which he or she lives. One example should suffice, although there are many.

At least 10 centuries ago, very few people were concerned with making long voyages, whether over land or sea. Most people therefore had no cause to consider that the earth might not be flat, and that the shortest distance from A to B on the surface of the earth is not a straight line, but instead an arc of a great circle. Thus, when the scientific proposal was made that the earth is spherical, most people considered the proposal extremely counter-intuitive. Today, however, it would be quite difficult to find anybody who considers this idea to be counter-intuitive. Thus, if an idea appears to be counter-intuitive, it is not necessarily wrong. Similarly, if an idea is appealingly intuitive, it is not necessarily right.

9.3 ■ The Alleged Breakdown of Normalization in the Relational Model

In attempting to show that the relational model runs into difficulties with normalization, the critics cite an example of a base relation R (A,B,C) satisfying the functional dependence A→B, for which it is not assumed that A is the primary key, so that A can be permitted to have missing db-values. The critics assert that serious problems are bound to arise if R contains a row (?,b1,c1) and an attempt is made to insert another row (?,b2,c2), where b1 and b2 are not equal, and ? denotes an A-mark. They assert that *either* the two nulls must be considered distinct from one another *or* the second row must be rejected because "it might violate the dependency" [Date, 1986] when the null is replaced by an actual value.

The critics seem to have rejected without supplying a reason a third option, which is the one adopted in the relational model. That is, whenever the A component of a row is missing (or becomes missing), the functional dependence A → B is not enforced by the DBMS for this row until an attempt is made to replace the mark (null) in column A by an actual db-value. In fact, if the proposed DV scheme were adopted, this third option would not be available, because a null or missing value is treated in the DV scheme as just another database value. Hence, in the DV scheme the functional-dependence constraint must be enforced upon first entry of the row, and this gives rise to the possibility that a row might be erroneously rejected by the DBMS.

The critics also assert that, if the second row is not rejected upon attempted entry, "we are forced to admit that we do not have a functional dependency" [Date, 1986] of B on A. This is clearly one more instance of a value-oriented misinterpretation.

The claim that the normalization procedure breaks down is false. It should be clear that, because nulls—or, as they are now called, marks—

are *not* database values, the rules of functional and multi-valued dependence do not apply to them. Instead, they apply to all unmarked db-values.

With the DV scheme, the normalization procedure does break down, precisely because missing information is treated as database values. This is one more reason why I contend that the DV scheme is not an acceptable solution to the problem of handling missing information in databases.

The following example, more practical and less symbolic, is intended to illustrate the absence of any effect of missing values on normalization. The example is a slightly modified version of one presented in [Codd 1971b].

The relation EMP identifies and describes employees. Three of its columns are shown:

E# Employee serial number (the primary key)
D# Department serial number
CT Contract type

In this company, a department is assigned to exactly one type of contract. It will be convenient to refer to this as Rule 1. One consequence of Rule 1 is that the EMP relation is not in third-normal form. The following functional dependencies are applicable:

$$E\# \rightarrow D\# \rightarrow CT.$$

Note that the department D# to which an employee E# is assigned is an immediate property of an employee, while the contract type CT is an immediate property of the department. In the column CT. the values g and n appear. They denote two types of contracts, government and non-government, respectively.

EMP	(E#	. . .	D#	CT)
	e1	. . .	d5	g
	e2	. . .	??	g
	e3	. . .	d2	n
	e4	. . .	d3	n
	e5	. . .	d2	n
	e6	. . .	??	n
	e7	. . .	d8	g

In this example, the two department numbers that are missing must be distinct, because of Rule 1 and the fact that the contract types in column CT of these two rows are distinct. The problems associated with checking functional dependency where there are missing values can be avoided completely by postponing the checking of compliance of each row with the functional dependence D# → CT until the attempted update of the missing department serial number (if any) to a non-missing value.

9.4 ■ Implementation Anomalies

In general, the present state of relational DBMS products in regard to the representation and handling of missing information is far from satisfactory. For IBM products based on the language SQL, the main problem is the way missing information is handled, rather than its representation. In non-IBM products, even the representation aspects have gone astray. In several of these products, the DBMS designer has misinterpreted nulls as db-values (see Section 9.1).

In at least one well-known (and otherwise sound) DBMS product [Relational Technology 1988], zero was chosen as the value to indicate missing information in all numeric columns. I consider the number zero to be far too valuable in its normal role in all kinds of business activities—for example, as a real number representing the actual quantity of a part in stock or the actual quantity of currency owed by one or more customers. Therefore, I do not consider zero to be an acceptable value to be reserved by a DBMS for denoting a missing numeric db-value. In fact, in the context of computer-supported database management, it is unacceptable to reserve *any* specific numeric value or character-string value to denote the fact that a db-value is missing.

9.5 ■ Application of Statistical Functions

An example involving the sum of an empty set of numbers is sometimes used to show that SQL encounters difficulties by unconditionally yielding the SQL null as the result. While I agree with this example when interpreted solely as an SQL blunder, I do not agree with the use of it as an example that justifies outright rejection of the relational approach to missing information.

If the sum were to yield zero unconditionally as the result, there would be the problem that the average of an empty set of numbers would not be the sum divided by the count (the number of elements in the set). This problem arises, however, because 0/0 is normally taken to be undefined in elementary mathematics; this difficulty does *not* stem from the relational approach to missing information. When taking averages, it is necessary for programmers and users to provide special treatment for the case in which the divisor (i.e., the number of elements in the set) is zero, because zero exhibits a unique behavior when it is used as a divisor. Incidentally, I fail to see how the DV scheme provides any solution or simplification for this problem.

I consider this problem to be quite separable from the question of how to deal with missing information. Nevertheless, the approach taken to this problem in the relational model can be illustrated by taking the example of generating the total salaries earned by each department, where each total is computed as the sum of the salaries earned by each employee assigned

to the pertinent department. Let us assume that a few departments exist that have zero employees assigned to them at this time.

In applying statistical functions, there are two important alternative methods of handling occurrences of empty sets.

1. Each occurrence of an empty set is ignored (i.e., passed over).
2. A value, specified by the user, is taken as the *result* for each occurrence of an empty set.

Note the difference between these two actions if the statistical function happens to be the AVERAGE, and if the value selected in the second approach is zero. The first action omits the departments that have zero members, while the second generates zero *for each such department.*

The empty set qualifier, a function ESR with a single argument, say x, causes each occurrence of an empty set to yield the result x and does not affect the result obtained from each non-empty set. Omission of this qualifier altogether causes each occurrence of an empty set to be ignored.

9.6 ■ Interface to Host Languages

Host languages do not include support specifically aimed at the semantics of the fact that information in databases may be missing. This means there is bound to be an interface problem, whatever approach is taken in the data sublanguage. If the approach taken on the database side of the interface is uniform and systematic (independent of data type), the interface is likely to be simpler than an approach like the DV scheme, which requires database users to keep inventing, column by column, their own techniques for dealing with this problem—not to mention the burden of communicating their inventions to other users of the database.

The relational approach therefore has a strong advantage in this area over the DV scheme. In this chapter and in the relational model itself, one major concession has been made to reduce user confusion about the host-language interface—namely, a change from the terms "null" and "null value" to the term "mark" for the indicator that designates the fact that a db-value is missing.

9.7 ■ Problems Encountered in the Default-Value Approach

There are six main problems with the DV approach:

1. The DV approach does not appear to provide any tools for the *handling* of missing information, but merely provides a means for *representing* the fact that something is missing.

2. The representation proposed is by means of a db-value, which forces the testing of functional dependencies and other kinds of dependencies at the time data is entered—the wrong time if a missing value is involved.
3. The representation of the fact that a db-value is missing is not only dependent on the data type of each pertinent column, but can even vary across columns having a common data type. All of this presents a severe burden in thinking and in inter-personal communication for the DBA, end users, and programmers.
4. The numerous and varied techniques for *handling* missing data will be buried in the application programs, and it is highly doubtful that they will be uniform or systematic, or even documented adequately.
5. Each missing db-value is treated as if it were just another db-value (i.e., the DV approach ignores the semantics and suffers from the value-oriented misinterpretation).
6. The DV approach is a step backward from the relational model to an ad hoc, unsystematic approach frequently adopted in the pre-relational era.

In the case of item 5, numerous specific consequences were cited earlier in this chapter, along with the ensuing penalties.

It is also important to realize that, whatever the approach, the DBA, application programmers, and end users *must cope with the semantics of the fact that some db-values for some columns may be missing.* Because the DV scheme offers no tools for handling missing information in a uniform, systematic way, users are forced to invent a variety of ad hoc, unsystematic ways, over which the DBMS cannot exert any real integrity control. Finally, research in this area is still being pursued, and I make no claim that the relational model, as it now stands, treats missing information in a way that is unsurpassable. Any replacement, however, must be shown to be technically superior.

9.8 ■ A Legitimate Use of Default Values

Suppose that a bank has a central database that includes information on all of its customer accounts. When a branch of the bank enters a new account, the information is inserted into the database from a terminal located in that branch. If the person making the entry omits the branch code (which identifies the branch), the system could assume with reasonable safety that the branch is identified by the particular terminal used for the entry.

In this example, a default value is being used for the branch code, and, during the entry of a new account, it is the system that computes an appropriate value for this code, and then inserts that code into the database along with the rest of the account information from the terminal. It is important to realize that at no time is the branch code actually missing from

its database context: a record that describes a customer account in detail. Therefore, this example is clearly distinguishable from a case of information missing in the database.

Use of the word "system" in the last two paragraphs is intentionally vague. It should not be interpreted as meaning the DBMS. No specific feature of RM/V2 supports this kind of use of default values, which can be handled adequately by terminal support that is programmed by users—provided, of course, that the system, of which the DBMS is part, can make the identification of a requesting terminal available along with each request.

9.9 ■ Concluding Remarks

In the past, I have intentionally included in the relational model a systematic treatment of missing-but-applicable information (information that is temporarily unknown). In RM/V2, I am now adding similar treatment for missing-and-inapplicable information (information that is unknowable). The whole treatment of missing information is intended to remove from database administrators and users the burden of solving this problem in highly specialized, and often inadequate, ways.

I make no claim that this systematic treatment is either intuitive or counter-intuitive. Neither do I claim that users who have used the old style with non-relational DBMS will be able to avoid the burden of learning the new approach. I also do not claim that missing-but-applicable information, inapplicable information, and the derivation of deductions therefrom are thoroughly understood yet.

Finally, I do not claim that RM/V2 handles the "half-missing" case, in which a specific and precise value is unknown, but either a small range of possible values is known, or else there is a high probability that the missing value is one of a very few values. This case may not be ignored in RM/V3, but it is now more urgent for DBMS products to handle the cases for which RM/V2 provides support. In any event, it will be necessary to show that the extra machinery (hardware and/or software) needed to support the "half-missing" case is going to pay its way.

The old-style approach used values that were specially earmarked by users to represent missing information (and misrepresent the semantics). The earmarking and the invention of manipulation techniques were likely to be different for each different column and were a significant burden on database administrators and users. The reader will undoubtedly agree that the present scheme in the relational model is far more systematic than the old-style approach, and moves more of the burden of handling missing information from the users to the DBMS.

Version 1 of the relational model was defined precisely in [Codd 1968–1979]. One of the great advantages of the relational approach is the unparalleled power of its treatment of integrity. It is high time for vendors and users to place more emphasis on the introduction and retention of database

integrity, and consequently invest more effort in learning the systematic treatment of missing information as described in this book.

Exercises

9.1 What is the default-value approach, and in what ways is it unsatisfactory for representing and handling the treatment of missing values? State five undesirable properties.

9.2 In what circumstances is it appropriate to store default values in a relational database? Has this anything to do with values that are missing from the database? If so, what is the connection?

9.3 Suppose that functional dependencies and multi-valued dependencies that are applicable to a certain database are stored in the catalog as DBA-defined integrity constraints. How does the relational model cope with these constraints when numerous columns are allowed to have missing values?

9.4 How does RM/V2 cope with the application of an aggregate function to an empty set? What does SQL deliver as the result? Which of these actions makes sense? Explain your answer.

9.5 Provide a legitimate case in which a default value (one not supplied by the user) should be stored in a database? Does the DV scheme support this type of default value? How is this case related to the representation and handling of missing information?

■ CHAPTER 10 ■

Qualifiers

A qualifier is an expression that can be used in a command to alter some aspect of the execution of that command. In the context of this book, the commands of interest are relational commands; during execution of these commands, the focus of interest is the effect of the qualifiers on database management. Features RQ-1–RQ-13 (the 13 qualifiers) are discussed in this section.

Normally, when a truth-valued condition in a relational command is evaluated, the specified combination of target values is extracted from the database, if and only if the complete condition evaluates to TRUE (abbreviated *t*). The first three features, RQ-1–RQ-3, are used to change the qualifying truth-value from *t* to one of the MAYBE truth values (*a* or *i*) for whatever scope of the condition is embraced by the MAYBE qualifier. Table 10.1 exhibits for each pertinent feature which truth-value becomes the qualifying truth-value in place of *t*.

Table 10.1 **Qualifiers and Truth Values**

F		Q
	TRUE	*t*
RQ-1	A-MAYBE	*a*
RQ-2	I-MAYBE	*i*
RQ-3	MAYBE	both *a* and *i*

Table 10.2 **Qualifiers**

Feature	Qualifier	Pertinent Context
RQ-1	A-MAYBE	Condition part of any operator
RQ-2	I-MAYBE	. . . . .
RQ-3	MAYBE	. . . . .
RQ-4	AR(x)	Any column in any request
RQ-5	IR(x)	. . . . .
RQ-6	ESR(x)	Any command
RQ-7	ORDER BY	Any retrieval command
RQ-8	ONCE ONLY	Any inner join
RQ-9	DOMAIN CHECK OVERRIDE	Any operator involving inter-column comparisons
RQ-10	EXCLUDE SIBLINGS	Operators involving PKs
RQ-11	DEGREE OF DUPLICATION	Any retrieval operator
RQ-12	SAVE	Any relational assignment
RQ-13	VALUE	Any command that can generate marked values

The absence of a TRUE or MAYBE qualifier indicates the TRUE case by default.

Features RQ-4 and RQ-5 deal with the temporary replacement of missing occurrences of values by specified values, where "temporary" means during execution of the pertinent command only. Feature RQ-4 provides for temporary replacement of A-marked values, while RQ-5 provides for temporary replacement of I-marked values. The need for these features is explained in Chapter 8.

Feature RQ-6 deals with the handling of empty sets. Feature RQ-7 imposes ordering upon the rows of the resulting relation, while RQ-8 forces each tuple from each operand to be used once if possible, or else not at all, in executing a **join.** Feature RQ-9 is the qualifier that suppresses the checking of domains when a relational command involves using one or more pairs of columns as comparands.

Feature RQ-10 pertains to controlling the propagation of certain operators (such as **update** and **delete**) to the sibling values of a given primary-key value. RQ-11 requests the DBMS to append to each row a count of the degree of *potential duplication* of that row in the result of a **projection** or **union,** if duplicate rows had not been prohibited in the relational model. RQ-12 requests the DBMS to save the relation formed as a result of executing a command. Table 10.2 provides a list of all the qualifiers and the context in which they are applicable.

10.1 ■ The 13 Qualifiers

In the context of a command that can generate marked values, RQ-13 causes a specified value to be inserted instead of these marked values. The three qualifiers in Features RQ-1–RQ-3, all based on four-valued logic, are concerned with extracting values when the condition part of a query has a truth-value other than TRUE or FALSE. A relation EMP that identifies and describes employees is used to illustrate the effect of these qualifiers:

	EMP (EMP#	ENAME	DEPT#	SALARY	BONUS)
1	E107	Rook	D12	10,000	— I
2	E912	Knight	D05	— A	2,000
3	E239	Knight	D03	12,000	1,800
4	E575	Pawn	D12	— A	— I
5	E123	King	D01	15,000	— A
6	E224	Bishop	D03	— I	2,500

"A" denotes an A-mark, and the corresponding value is missing and applicable. "I" denotes an I-mark, and the corresponding value is missing and inapplicable. The row numbers are purely expository.

The query to be applied in each case is as follows: retrieve the employees whose salary exceeds 11,000 and whose bonus is less than 4,000. This query can be expressed using the Boolean extension of the **select** operator:

$$Q \leftarrow EMP\ [(SALARY > 11{,}000) \wedge (BONUS < 4{,}000)].$$

Without any qualifier, query Q selects row 3 of the EMP relation.

RQ-1 The MAYBE_A Qualifier

This qualifier, based on four-valued logic, can be applied to any truth-valued expression in an RL command. The DBMS focuses on those items for which this expression has the truth-value *a* (which denotes MAYBE-AND-APPLICABLE). For example, if the MAYBE_A qualifier is applied to the whole condition, then the DBMS yields as the final result just those items for which the whole condition has the truth-value *a*.

Do not confuse the MAYBE_A qualifier with the truth-value *a* or with an A-marked value. The query Q qualified by MAYBE_A selects rows 2 and 5 of the EMP relation.

RQ-2 The MAYBE_I Qualifier

This qualifier, based on four-value logic, can be applied to any truth-valued expression in an RL command. The DBMS focuses on those items for which this expression has the truth-value *i* (which denotes maybe and inapplicable). For example, if the MAYBE_I qualifier is applied to the whole condition, then the DBMS yields as the final result just those items for which the whole condition has the truth-value *i*.

Do not confuse the qualifier MAYBE_I with the truth value *i* or with an I-marked value. The query Q qualified by MAYBE_I selects rows 4 and 6 of the EMP relation. It does not select row 1, since the non-missing salary is less than 11,000.

RQ-3 The MAYBE Qualifier

This qualifier, based on four-valued logic, can be applied to any truth-valued expression in an RL command. The DBMS focuses on those items for which this expression has the truth-value *a* or *i* (either applicable or inapplicable). For example, if the MAYBE qualifier is applied to the whole condition, then the DBMS yields as the final result just those items for which the whole condition has the truth-value *a* or *i*.

Do not confuse the qualifier MAYBE with the truth-values *a* or *i*, or with marked values (an A-marked value or an I-marked value). The query Q qualified by MAYBE selects rows 2, 4, 5, and 6 of the EMP relation.

If the DBMS supports features RQ-1–RQ-3 fully, it must support four-valued logic under the covers. For more details on four-valued logic, see Chapter 8.

RQ-4, RQ-5 Temporary Replacement of Missing Database Values

In applying statistical functions to database values in one or more columns of an R-table, missing occurrences of such values can be temporarily replaced (during the execution of the function only) by applying the qualifier AR(x), which replaces A-marked values by x (in the case of Feature RQ-4) or the qualifier IR(x), which replaces I-marked values by x (in the case of Feature RQ-5).

RQ-6 Temporary Replacement of Empty Relation(s)

The qualifier ESR(x) appended to an RL expression causes each empty relation encountered *as an argument* during the execution of that expression to be replaced by the set whose only element is x, provided x is type-compatible with the pertinent relation (normally, of course, x is a tuple).

10.1.1 Ordering Imposed on Retrieved Data

The ability of a DBMS to deliver derived data in any specified order that is based on values within the result should be understood from two points of view: (1) the terminal users, and (2) the application programmers.

Frequently, a terminal user must see the data retrieved from a relational database in some specific sequence. Moreover, an application programmer may be faced with the task of taking the retrieved data, an entire derived relation, and interfacing that data to a host language that, for computing reasons, is able to process no more than a single record at a time. Such a programmer is likely to require that the retrieved data be ordered in some specific sequence.

The relational model does not permit the programmer to take advantage of the sequence in which the data in the database happens to be stored. Two important reasons for this are as follows:

1. the DBA may alter the way data in the database is stored at any time to improve performance and to cope with changes in traffic on the database;
2. the program should work correctly on a system of different design (even if it is supplied by another vendor), and a different design may not support precisely the same representations of data in storage;

RQ-7 The ORDER BY Qualifier

An ORDER BY clause consists of the following:

- the ORDER BY qualifier;
- names for those columns of the operands whose values are to act as the ordering basis;
- a symbol ASC or DESC indicating whether the ordering is to be by ascending values or descending values.

An ORDER BY clause can be appended to a relational command that retrieves data. The DBMS then delivers the data in the order specified.

One option available to the user is to base the ordering on the values occurring in a simple or composite column of one of the operands or of the resulting relation. If the ordering in the result is not represented (redundantly) by values in one or more columns of the result, the DBMS must warn the user of that fact (see Feature RJ-10 in Chapter 11).

If the ordering is based on character strings, a collating sequence is used that is declared by name only, if standard, or by name and extension, if non-standard.

If the comparator < is inapplicable to the extended data type of any of the columns upon which the ordering is based, the DBMS applies the comparator to the corresponding basic data type. (See Chapter 3 for the distinction between basic data type and extended data type.)

As just noted, the DBMS must warn the user when the ordering in the result is not represented (redundantly) by values in one or more columns of the result. This warning is required because a user might expect to be able to use *all* of the information in the result for further interrogation. It is important to remember that the relational operators are incapable of exploiting any information that is not represented by values in R-tables. This *incapability* is neither an accident nor an oversight. It is intended to keep the relational operators from becoming overly complicated in handling simple tasks.

Consider the example of the relation C, which identifies and describes the capabilities of suppliers. This relation is intended to provide information concerning which suppliers can supply which kinds of parts. Examples of properties that are applicable to capabilities are as follows: speed of delivery of parts ordered, the minimum package size adopted by the supplier as the unit of delivery, and the price of this unit delivered.

S#	Supplier serial number
P#	Part serial number
SPEED	Number of business days to deliver
QP	Quantity of parts
UNIT_QP	Minimum package
MONEY	U.S. currency
PRICE	Price in U.S dollars of minimum package.

There are five domains: S#, P#, TIME, QP, and MONEY.

C

S#	P#	SPEED	UNIT_QP	PRICE
s1	p1	5	100	10
s1	p2	7	100	20
s1	p6	12	10	600
s2	p3	5	50	37
s2	p5	8	100	15
s3	p6	15	10	700

s4	p2	10	100	15
s4	p5	15	5	300
s5	p6	10	5	350

Suppose there is an urgent need for fast delivery of certain parts. The following request tabulates triples consisting of the serial numbers of suppliers, the serial numbers of parts, and the speed of delivery; all the triples are ordered by part serial number (major participant) and speed (minor participant):

C[S#, P#, SPEED] ORDER BY (P#, SPEED).

Upon receipt of this request, the DBMS delivers the following result:

C	(S#	P#	SPEED)	
	s1	p1	5	p1
	s1	p2	7	p2
	s4	p2	10	p3
	s2	p3	5	
	s2	p5	8	p5
	s4	p5	15	
	s5	p6	10	p6
	s1	p6	12	
	s3	p6	15	

Note that the option of requesting the ordering to be based on values in a column of the result allows the values computed according to a specified function to be used as the ordering basis. For example, suppose that a relation EMP containing information about employees is being interrogated, and that EMP has:

- a column containing the present salary of each employee; **and**
- a column containing the department number of each employee.

Consider this query: find the department number together with the total salary earned by all employees assigned to that department. It must be possible to display the result ordered by these total salaries.

10.1.2 The ONCE Qualifier and Its Effect upon Theta-joins

The **inner** and **outer T-joins** were introduced in Section 5.7. An interesting property of the **T-joins** is that each tuple of the operands participates at most once in the result. It is entirely possible that some of the operands' tuples do not participate at all in mid-sequence—neither at the early end nor at the late end.

RQ-8 The ONCE ONLY Qualifier (abbreviated ONCE)

When attached to a request for an **inner T-join**, the qualifier ONCE converts this request into a special **inner T-join**, in which every tuple of the operands participates exactly once with few exceptions. The exceptions can occur at the early end, the late end, or both ends of the sequence, where early and late are based on values of date and/or time in the comparand columns. Similarly, the qualifier ONCE converts an **outer T-join** into a special **outer T-join**, in which every tuple of the operands participates exactly once without exception.

How is this full participation realized? Consider the example that was introduced in Section 5.7. The sample operands are relations S and T:

S (P	A)
k1	4
k2	6
k3	12
k4	18
k5	20

T (Q	B)
m1	3
m2	5
m3	9
m4	11
m5	13
m6	15

The result of taking the **inner T-join** based on < of S on A with T on B (omitting the qualifier ONCE) is the relation V, assuming that

$V \leftarrow S [[A < B]] T.$

V (P	A	B	Q)
k1	4	5	m2
k2	6	9	m3
k3	12	13	m5

Notice that tuples < k4, 18 > and < k5, 20 > of relation S and tuples < m1, 3 >, < m4, 11 >, and < m6, 15 > of relation T did not participate at all in the result.

If the qualifier ONCE is attached, however, the result is generated by first sorting each relation by increasing time (S is sorted by A, T is sorted by B). The next step is to start at the tuple in S with the earliest time, namely < k1, 4 >. This tuple is coupled with the earliest tuple in T, namely < m1, 3 >. To do this and still comply with the comparand <, the time component of the tuple < m1, 3 > from T is incremented by the least integer amount (i.e., 2) to make it satisfy the LESS THAN condition. When coupled, the resulting tuple is < k1, 4, 5, m1 >. Note that I have reversed the last two components for expository reasons.

The next step is to couple < k2, 6 > from S with < m2, 5 > from T by incrementing the time component of the row from T by the least amount (again, 2) to make it comply with the LESS THAN condition. The resulting tuple is < k2, 6, 7, m2 >. These steps are repeated and yield first < k3, 12, 13, m3 >, then < k4, 18, 19, m4 >, then < k5, 20, 21, m5 >, and finally the tuple < m6, 15 > from T is left as a non-participant. Notice that the necessary increments are not constant. The resulting relation is

W (P, A, B, Q)

P	A	B	Q
k1	4	5	m1
k2	6	7	m2
k3	12	13	m3
k4	18	19	m4
k5	20	21	m5

If, instead of the inner T-join, the symmetric outer T-join had been used with the ONCE qualifier, the result would have been

W (P, A, B, Q)

P	A	B	Q
k1	4	5	m1
k2	6	7	m2
k3	12	13	m3
k4	18	19	m4
k5	20	21	m5
—	—	22	m6

Note that the time component of the last tuple < m6, 15 > from T has been incremented by the least integer amount (i.e., 7) that will make it greater than the largest time component in the rest of the result.

Note that the operands S and T remain in the database unchanged. Thus, it would be incorrect to regard this operation as an update of either S or T. The user may need to bear in mind, however, that the values in the result are not necessarily drawn from the database without change. Whatever changes take place are certainly not the result of applying a simple transformation uniformly across all the values in a column.

RQ-9 The DOMAIN CHECK OVERRIDE (DCO) Qualifier

If specifically authorized, use of the qualifier DCO in a command permits values to be compared during the execution of the command that are drawn from *any pair* of distinct domains in the entire database. The qualifier may, however, be accompanied by the name of a unary relation containing a specific list of the names of domains. The effect of this list is to request the DBMS to permit comparing

activity that involves pairs of distinct domains, only when the names of those domains appear in the list.

A user who is authorized to use qualifier RQ-9 across the entire database is endowed with tremendous power for doing good or evil. This is why I would recommend that the authorization for domain check override normally be confined to a short, specified list of domains only, and even then only for a short time and specific trouble analysis.

For more detail, see Features RJ-6 in Chapter 11, RM-14 in Chapter 12, and RA-9 in Chapter 18. Use of this qualifier would seldom be authorized, and then for only a short time. The principal use is for detective work in trying to determine how a portion of the database lost its integrity. For example, if the domains for a particular database happen to include a part serial number domain and a supplier serial number domain, and they happen to have identical basic data types (both character strings of length 12, say), one might wish to ask which of these semantically distinct serial numbers happens to be identical to one another when viewed simply as character strings.

RQ-10 The EXCLUDE SIBLINGS Qualifier

In some of the manipulative operators (Features RB-33–RB-34, RB-36–RB-37), the primary key of some base relation is either specified directly or is indirectly involved, and certain action is to be taken on the siblings of this primary key. This action on the siblings is thwarted if the EXCLUDE SIBLINGS qualifier is attached to the command.

See Section 4.3 for definitions and details.

RQ-11 The Appended DEGREE OF DUPLICATION (DOD) Qualifier

Assume that the DOD qualifier is appended to the projection of a single relation or to the **union** of two union-compatible relations. For each row in the result, the DBMS calculates the number of occurrences of that row if duplicate rows had been permitted in the result. This count is appended to each row in the actual result as an extra component. Thus, the result is a relation with an extra column, which is called the *DOD column* here.

This qualifier enables the DBA to grant a user access to enough information from the database for him or her to make correct statistical analysis, without

granting access to some primary keys that happen to be sensitive. See Chapter 18 for further information on this topic, as well as an alternative approach that can be taken by the DBA and DBMS.

Examples of **projection** with the DOD qualifier appear next. The operand is as follows:

R (K A B C D . . .)

k1	a1	b1	c1	d1	. . .
k2	a1	b1	c1	d2	. . .
k3	a1	b1	c1	d2	. . .
k4	a1	b1	c2	d3	. . .
k5	a2	b1	c1	d3	. . .
k6	a2	b2	c2	d4	. . .

The results are as follows:

R [A B C] DOD

a1	b1	c1	3
a1	b1	c2	1
a2	b1	c1	1
a2	b2	c2	1

R [A B] DOD

a1	b1	4
a2	b1	1
a2	b2	1

R [B C] DOD

b1	c1	4
b1	c2	1
b2	c2	1

R [A] DOD

a1	4
a2	2

R [B] DOD

b1	5
b2	1

R [C] DOD

c1	4
c2	2

R [D] DOD

d1	1
d2	2
d3	2
d4	1

Clearly, any DOD **projection** of R that includes the primary key K of R will have as many rows as R does, and in each row the DOD component will be exactly one. Note also that if relation R happens to be empty, every projection of R is empty, whether or not the pertinent projection is DOD-qualified.

An example of **union** with the DOD qualifier appears next. The operands are as follows:

S (A B C)

a1	b1	c1
a2	b2	c2

T (A B C)

a2	b2	c2
a3	b2	c2

The result is as follows:

S UNION T (A B C DOD)

a1	b1	c1	1
a2	b2	c2	2
a3	b2	c2	1

Note that 1 and 2 are the only two possible values for the DOD component of the DOD-qualified **union** of any two union-compatible relations. Note also that, if both operands happen to be empty, the result is empty whether the **union** is DOD-qualified or not.

Implementation of Feature RQ-11 in an index-based DBMS is quite easy and cheap—most of the code needed for RQ-11 must be developed to support the CREATE INDEX command in any case (see Feature RE-14 in Chapter 7).

Each of the statistical functions built into the DBMS should have two flavors: one that treats each row as it occurs (just once, ignoring any DOD component, if such exists); the other that treats each row as if it occurred *n* times, where *n* is the DOD component of that row (see Chapter 19 for details).

RQ-12 The SAVE Qualifier

The SAVE qualifier may be attached to any relational assignment (see Feature RB-30 in Chapter 4). Let T be the relation formed by this assignment. The SAVE qualifier requests the DBMS to store the description of T in the catalog, and to save T as if it were part of the database.

If the SAVE qualifier is omitted, and if T still exists at the end of the interactive session or after the pertinent application program is executed, then the DBMS drops T. Thus, the SAVE qualifier causes a relation to be saved for shared use. Omitting the SAVE qualifier restricts the pertinent relation to private and temporary use—as far as the DBMS is concerned. A user must have the necessary authorization to make a copy of a base or derived relation to be saved for private use (outside the control of the DBMS). He or she may then issue an EXTRACT command (see Feature RE-19 in Chapter 7).

RQ-13 The VALUE Qualifier

When this qualifier is attached along with a value v to a command or expression that (1) creates a new column in a relation (base or derived) and (2) would normally fill this column with marked values, it causes v to be inserted in this column instead of each of the marked values.

This qualifier can be used with the advanced operator RZ-2 to extend a relation per another relation or with the DBA command RE-10 to append a named column to the description and to the extension of a base relation.

Exercises

10.1 Which qualifiers in RM/V2 support the extraction of data for which the truth value of the whole condition is
   1. a = unknown due to a missing-but-applicable datum?
   2. i = unknown due to a missing-and-inapplicable datum, and
   3. either a or i.

10.2 Describe the effect of appending the AR qualifier with 912 as its argument to a statistical function applied to a numeric column.

10.3 Describe the effect of appending the IR qualifier with the character string "??" as its argument to a statistical function applied to a character-string column.

10.4 What is the DBMS required to do if the user includes the ORDER BY qualifier in his or her request, and the ordering is not represented redundantly by values in the result?

10.5 With what kind of operators can the ONCE qualifier be attached to one or both of the operands? What is the effect of this qualifier?

10.6 How is a regular **join** on $<$ with the ONCE qualifier related to a **T-join** on $<$?

10.7 Describe the domain checking that is inhibited by the domain check override.

10.8 What action is excluded on what primary keys when the EXCLUDE SIBLINGS qualifier is used?

■ CHAPTER 11 ■

Indicators

An *indicator* is a side effect of executing a relational command. If turned on during the execution of a relational command, it indicates the occurrence of an exceptional condition pertaining to the relational result, an intermediate result, or one of the arguments. Note that indicators are best implemented as return codes, preferably with explanatory comments. There is likely to be a need for more indicators than the 14 listed in Table 11.1.

Each of the 14 indicators cited in Features RJ-1–RJ-14 is turned off at the beginning of the execution of each type of RL command that is capable of turning such an indicator on, so that its state at any time reflects the outcome of the most recently executed command of this type.

Of course, the language RL must permit use of these indicators as part or all of a condition expressed within an immediately following RL command. To support this, each of the 14 indicators (except RJ-11–RJ-14) comes in pairs, say u and v, because it is necessary to distinguish between the use of an indicator as an argument in a command and its use as a result of that command.

During execution of an RL command, the indicator u is used to remember the indication from the immediately previous RL command, and the indicator v is prepared to accept the indication from execution of the current RL command. During execution of the immediately following RL command, the roles of u and v are reversed. All a user must know is that, when an indicator (no matter what its type) is tested in the condition part of an RL request, it reflects the execution of the immediately preceding RL command.

Table 11.1 **Indicators**

Feature	Type	Indicator	Pertinent Features
RJ-1	Result	Empty relation	All operators
RJ-2	Argument	Empty divisor	**Relational division** (RB-27)
RJ-3	Result	Missing information	All operators
RJ-4	Argument	Non-existing argument	RE-2, RE-3, RE-5, RE-6, RE-8, RE-9, RE-12
RJ-5	Argument	Domain not declared	Drop domain (RE-4)
RJ-6	Argument	Domain check error	**Selects, joins, relational division** (RB-27)
RJ-7	Argument	Column still exists	Alter domain (RE-3)
RJ-8	Argument	Duplicate row	Loading
RJ-9	Argument	Duplicate primary key	Loading
RJ-10	Result	Non-redundant ordering	ORDER BY qualifier (RQ-7)
RJ-11	Result	Catalog block	Blocks to alter database description (RM-7)
RJ-12	Result	View not tuple-insertible	Create view
RJ-13	Result	Tuple-component not updatable	Create view
RJ-14	Result	View not tuple-deletable	Create view

The term "immediately preceding RL command" means the RL command that the DBMS encountered as the immediately preceding one from the particular terminal or program, whichever is pertinent. The terms "preceding" and "following" apply in this case to RL commands only, not to any host-language commands in between the RL commands.

As can be seen from Table 11.1, most of the indicators are *argument indicators*. This means that, when turned on, they reflect an exceptional condition that applies to one of the arguments of a command.

Two of the six result indicators (Features RJ-1 and RJ-2) are intended to relieve users of the burden of detailed (and possibly programmed) examination for emptiness in one or more relations, and for the possibility that one or more cases of missing information were encountered during the execution of the command.

11.1 ■ Indicators Other than the View-Defining Indicators

RJ-1 Empty-relation Indicator (Result Indicator)

When the result of any retrieval or manipulative command expressed in RL is generated, an *empty-relation indicator* is turned on whenever the final result happens to be (or to include) an empty relation.

RJ-2 Empty Divisor Indicator (Argument Indicator)

If (1) a command in RL is about to be executed, (2) it involves relational division, and (3) the divisor relation happens to be empty, then an empty-divisor indicator is turned on, and the result generated by the DBMS from the division is the dividend with the divisor columns removed.

The empty-divisor indicator is set to zero at the beginning of the execution of each RL command. The indicator is set to one (and remains set to one) whenever within any command a **relational division** is encountered for which the divisor is empty. Therefore, upon completion of the execution of a single command of RL, if the empty-divisor indicator is in state one, this indicates that an empty divisor was encountered in one or more of the **relational divisions** within that command. This indicator is most helpful when the divisor happens to be an intermediate result, one that exists for only a short time during the execution of a more comprehensive command.

The relation that results from a **relational division** by an empty set is just what one would expect from the corresponding expression in predicate logic involving the universal quantifier. For example, suppose the database includes an R-table indicating in each row that supplier S# *can supply* part P#. If the user is finding the suppliers, each of whom can supply every one of a list of parts, and if that list happens to be empty, then every supplier recorded in the CAN_SUPPLY relation qualifies.

RJ-3 Missing-information Indicator (Result Indicator)

Whenever, during the execution of any retrieval or manipulative command expressed in RL, the DBMS encounters a database value declared to be missing, the missing-information indicator is turned on.

RJ-4 Non-existing Argument Indicator (Argument Indicator)

The DBMS is unable to find an argument in accordance with the name specified in the command being executed. Execution of the command is aborted.

RJ-5 Domain-not-declared Indicator (Argument Indicator)

An attempt has been made to execute a CREATE R-TABLE command in which a column draws its values from domain D, and the DBMS finds that domain D has not been declared. Execution of the command is aborted.

RJ-6 Domain-check-error Indicator (Argument Indicator)

An operator has been requested that involves comparing values from two columns. The DBMS discovers that (1) the columns cited do not draw their values from a common domain, and (2) the user has not specified DOMAIN-CHECK-OVERRIDE in his or her command. The domain-check-error indicator is turned on, and execution of the command is aborted.

This feature protects the database from damage by those users who happen to make errors in formulating **selects, joins,** and **divides.** Such errors are quite likely when a naive or tired user is trying to exploit a powerful relational language. For more detail, see Feature RQ-9 in Chapter 10, Feature RM-14 in Chapter 12, and Feature RA-9 in Chapter 18.

The following feature is effective when the DBA or some other suitably authorized user attempts to drop a domain from the catalog without making sure that there no longer exists in the database a column that draws its values from that domain.

RJ-7 Domain Not Droppable, Column Still Exists Indicator (Argument Indicator)

An attempt has been made to execute a DROP DOMAIN command, but a column still exists that draws its values from that

domain. When this indicator is turned on, the DBMS aborts the DROP DOMAIN command.

See Features RE-3 and RE-6 in Chapter 7 for more detail.

RJ-8 Duplicate-row Indicator (Argument Indicator)

When loading data from a non-relational source into the base R-tables of a relational database, the DBMS examines the data to see whether duplicate rows occur. If so, the duplicate-row indicator is turned on.

For both Feature RJ-8 and Feature RJ-9, it is assumed that the description of each base R-table is in the catalog before the loading is started. This applies even if the base R-table is empty before the loading. Of course, the description includes an identification of which column(s) constitute the primary key.

The duplicate-row indicator is not intended for use during any manipulative operations that are totally within the relational model. Such operations never generate duplicate rows.

RJ-9 Duplicate-primary-key Indicator (Argument Indicator)

When loading data from a non-relational source into a base R-table of a relational database, the DBMS examines the data to see whether there are duplicate occurrences of primary key values. If so, the duplicate-primary-key indicator is turned on.

Features RJ-10 and RJ-11, which follow, are also described in Section 4.3 in the context of the insert operator RB-31.

RJ-10 Non-redundant Ordering Indicator (Result Indicator)

As noted in Feature RQ-7 in Chapter 10, the ORDER BY qualifier can generate a result in which tuples are ordered according to information not included in the result. When this occurs, the non-redundant-ordering indicator (NRO) is turned on.

RJ-11 Catalog Block Indicator (Result Indicator)

This indicator indicates that a catalog block of commands is being executed. It is turned on by a BEGIN CAT command only and turned off by an END CAT command only. Thus, it stays on throughout the execution of a catalog block.

During the execution of a catalog block, the DBMS uses this indicator to suppress cascading action that would occur if the indicator had been off. See the DROP R-TABLE command (Feature RE-9 in Chapter 7) for an example of a command in which the cascading action is dependent on the state of the catalog-block indicator.

11.2 ■ The View-Defining Indicators

At the beginning of execution of a CREATE VIEW command, the indicators RJ-13, RJ-14, RJ-15 are all turned off. The DBMS then examines the updatability of the declared view using algorithm VU-1 or some stronger algorithm (see Chapter 17). Each of the three indicators is either left off or turned on, whichever accurately reflects the extent of updatability of the view.

RJ-13 View Not Tuple-insertible

From the view definition contained in a CREATE VIEW command, the DBMS has inferred that the view is not tuple-insertible.

RJ-14 View Not Component-updatable

From the view definition contained in a CREATE VIEW command, the DBMS has inferred that at least one component of every tuple in the view is not updatable.

Of course, as part of the execution of a CREATE VIEW command, the DBMS determines for each component (virtual column) whether it is or is not updatable. This information is stored in the catalog (a bit for each component), so that it is not necessary to recompute any of it when any update is requested on this view.

RJ-15 View Not Tuple-deletable

From the view definition contained in a CREATE VIEW command, the DBMS has inferred that the view is not tuple-deletable.

For more information on view updatability, see Chapter 17.

Exercises

11.1 Describe the empty-relation indicator and the empty-divisor indicator. In what ways is each of these indicators useful?

11.2 Upon completion of the execution of an RL command, it is found that the missing-information indicator has been turned on. What does this mean?

11.3 What are the domain-oriented indicators, and what is their purpose?

11.4 A file is being loaded into a relational database from a non-relational source. Explain the two indicators that may be turned on, and indicate two distinct forms of undesirable redundancy.

11.5 A user is trying to drop a domain. The DBMS refuses, and the column still exists indicator is turned on. What does this mean?

11.6 What is the purpose of the non-redundant ordering indicator?

11.7 Consider a request for tuple insertion, component update, or tuple deletion acting on a view. Is it at request time or at view-definition time that the DBMS determines whether the request can be honored while maintaining integrity? Justify the timing adopted in the relational model.

■ CHAPTER 12 ■

Query and Manipulation

The features described in this chapter concern the general properties and capabilities of the relational language, not its specific features, and certainly not its syntax.

12.1 ■ Power-oriented Features

RM-1 Guaranteed Access

Each and every datum (atomic value) stored in a relational database is guaranteed to be logically accessible by resorting to a combination of R-table name, primary-key value, and column name. (This feature is Rule 2 in the 1985 set.)

The access path supporting this feature cannot be canceled. Most other access paths, however, are purely performance-oriented, and can be both introduced and canceled. Both Feature RM-1 and Feature RM-3 are needed in order to support ad hoc query without pre-defined access paths.

Clearly, each datum in a relational database can be accessed in a rich variety (possibly thousands) of logically distinct ways. It is important, however, to have at least one means of access, independent of the specific relational database, that is guaranteed—because most computer-oriented

concepts (such as scanning successive addresses) have been deliberately omitted from the relational model.

Note that the guaranteed-access feature represents an associative-addressing scheme that is unique to the relational model. It does not depend at all on the usual computer-oriented addressing. Moreover, like the original relational model, it does not require any associative-addressing hardware, even though the need for such hardware was once frequently claimed by opponents of the relational model.

The primary-key concept, however, is an essential part of Feature RM-1. Feature RS-8 requires each base relation to have a declared primary key (see Chapter 2). Feature RM-1 is one more reason why the primary key of each base relation should be supported by every relational DBMS, and why its declaration by the DBA should be mandatory for every base relation.

RM-2 Parsable Relational Data Sublanguage

There is at least one relational language (denoted RL in this book) supported in the DBMS (not in an optional additional software package) such that (1) RL statements must be capable of being represented as parsable character strings, and can therefore be written or typed by a programmer, (2) for each manipulative operation, each and every operand is a relation, and (3) for each manipulative operation, each generated result is a relation with the result indicators (see Chapter 11) acting as a possible source of additional information.

A few vendors strongly promote interaction by programmers based on multiple-choice questions generated by the DBMS. This is claimed to be an alternative to writing programs, but I believe that the claim is insufficiently substantiated. This unproven claim is one important reason for requiring statements that are parsable character strings. Three additional reasons for this requirement are as follows:

1. it facilitates program maintenance;
2. the language is then in a form suitable for formal analysis;
3. the language may represent a standard for interfacing the DBMS to software packages on top (e.g., application development tools and expert systems).

Interactive tools that are claimed to make written or typed programs obsolete do not yet appear to support the maintenance requirement adequately. Moreover, these tools are badly in need of a published abstract model.

RM-3 Power of the Relational Language

Excluding consideration of general logical inference, RL as a language has the full power of four-valued, first-order predicate logic [Pospesel 1976, Stoll 1961, Suppes 1967, Church 1958].

The DBMS is capable of applying this power to all the following tasks:

- retrieval (database description, contents, and audit log);
- view definition;
- insertion;
- update;
- deletion;
- handling missing information (independent of data type);
- integrity constraints and authorization constraints.

If the DBMS is claimed to be able to handle distributed data, it can handle the following task:

- Distributed database management with distribution independence, including automatic decomposition of commands by the DBMS and automatic recomposition of results by the DBMS (see Features RP-4 and RP-5 in Chapter 20).

This last task represents a target to which the DBMS should apply the relational language RL, rather than a reason to extend RL. In other words, RL is scarcely affected by the need to support the management of distributed databases.

Of course, if the DBMS is to handle all the tasks specified in Feature RM-3, it would need additional capabilities beyond predicate logic. Clearly, to take just two examples, functions and arithmetic operators may also be needed. We emphasize the *predicate logic* because it is vital as a source of power of the relational language, and will remain so until another logic as powerful and rigorous is developed, which could take another two millennia.

RM-4 High-level Insert, Update, and Delete

The relational language RL supports **retrieval, insert, update,** and **delete** at a uniformly high, set level (multiple-records-at-a-time). (This Feature is Rule 7 in the 1985 set.)

This requirement gives the system much more scope in optimizing the efficiency of its execution-time actions. It allows the system to determine

which access paths to exploit to obtain the most efficient code. It can also be extremely important in obtaining efficient handling of transactions across a distributed database. In this case, users would prefer that communication costs are saved by avoiding the necessity of transmitting a separate request for each record obtained from remote sites.

If a product supports *retrieval only* at this high level, it is not a relational DBMS, but merely a system that includes a relational retrieval subsystem.

RM-5 Operational Closure

RL is mathematically closed with respect to the relational operators it supports.

This means that the retrieval and manipulative operators that can be invoked by statements in RL are incapable (and must remain incapable) of generating a result that is neither a relation nor a set of relations (although the indicators mentioned in Chapter 11 may provide additional output). The following misinterpretations are common:

- that RL *cannot* be expanded to support more operators than are currently part of the relational model;
- that an operator must *not* generate a relation that lacks a primary key.

Regarding the second misinterpretation, it is worth noting that, when a relation is generated that does not have a primary key, it always has a weak identifier. Note also that, as mentioned in Chapter 1 and elsewhere, in RM/V2 no relation, whether base or derived, is allowed to have duplicate rows.

Feature RM-5 is as necessary in database management as arithmetic closure is in accounting. When applying addition, subtraction, and multiplication to numbers, the accountant knows that the result is always a number. Therefore, it is always possible to continue the process and use a result from one activity as an argument for another. Similarly, when a user interrogates a relational database, the result is always a relation. Thus, it is always possible to continue the process and use a result from one activity as an argument for another. This feature makes it possible for users to employ interrogation in a detective style.

12.2 ■ Blocking Commands

If a DBMS is to be more than a simple query system, it must support the transaction concept. The precise definition accepted today is due to the System R team at IBM Research. (I believe that a principal contributor to this definition was Jim Gray.)

A *transaction* is a logical unit of work that transforms a consistent state of the database into a consistent state, without necessarily preserving consistency at all intermediate points. All actions within this logical unit of work must succeed, or else none of them must succeed. This atomicity of the unit of work must be applicable even though a sequence of several commands is normally involved in specifying the work to be done within that unit.

Within the logical unit of work, the DBMS may build up numerous changes for the database: entirely new rows to be inserted, rows that have been updated, or deletions. Usually these changes are accumulated in a cache memory until a command to commit the changes (usually called COMMIT) is received. Then, all the changes are recorded in the database.

If a failure occurs in either the hardware or the software during execution of the transaction, it is the responsibility of the DBMS to ensure that none of the changes accumulated in the cache memory is recorded in the database. If the program discovers some irregularity, such as an attempt to divide by zero, it issues a ROLLBACK command to abort the transaction and cause none of the changes to be committed to the database.

Three commands are necessary (the System R terms are adopted):

1. BEGIN signals the DBMS that a transaction is about to begin.
2. COMMIT signals the DBMS that a transaction has been completed in a normal manner, and that therefore all the changes to the database generated by this transaction can be committed to the database.
3. ROLLBACK signals the DBMS that none of the changes to the database generated so far in the execution of this transaction is to be committed.

RM-6 Transaction Block

> The BEGIN and COMMIT commands identify the beginning and ending of a block of commands. At least one of the commands within a block must be expressed in the relational language; the others may be expressed in either the relational language or in the host language or in both. Such a block constitutes a transaction if, during its execution, either all parts succeed or none succeeds. ROLLBACK signals the DBMS (1) to terminate this execution of the transaction requested by the program, and (2) to avoid committing to the database any of the changes already developed during this execution of the transaction."

The following practical example illustrates the need for this feature. Suppose that a customer requests a bank to transfer $1,000 from his or her checking account to his or her savings account. Such a transfer is normally programmed so that the first action is an attempted withdrawal of $1,000 from the checking account (which incidentally checks to see whether the

balance in the checking account is sufficient for the withdrawal to be made). If the first action is successful, the second action is to deposit this amount in the savings account.

Suppose that immediately after the withdrawal succeeds, a hardware failure occurs and the computer is taken off-line. Then, the corresponding deposit has not been put into effect. If the withdrawal has been recorded in the database and the deposit has not been recorded, the customer has lost $1,000. Therefore, such a transfer of funds from one account to another should be treated as a logical unit of work (i.e., as a transaction) so that all of it succeeds, or none of it succeeds.

RM-7 Blocks to Simplify Altering the Database Description

An RL command—labeled CAT here—signals the DBMS that the immediately following commands are all RL commands (i.e., no host language occurs in the package) and that each of these commands deals with changes in the catalog. The sequence of commands is ended by the RL command END CAT. In executing a block of RL commands defined in this way, the DBMS postpones certain actions until the command END CAT is encountered. The postponed actions include cascading actions normally associated with dropping base R-tables and views, plus the application of certain I-timed integrity constraints. Immediately before encountering END CAT, the DBMS cancels that part of the postponed cascading of view elimination, which has become unnecessary by END CAT time. It also cancels cascading of deletion for those authorization assertions that are still meaningful.

This feature concerns reducing or eliminating cascading actions on the catalog that result from changes in the catalog requested by isolated (unblocked) RL commands. Examples of such actions follow:

- dropping every view whose definition depends on an R-table (base or view) when that table is being dropped;
- dropping all authorization commands that refer to an R-table when that table is being dropped.

In many cases, Feature RM-7 enables the DBMS to eliminate or drastically reduce cascading action (such as the dropping of all views defined on a relation R when R is dropped) that results from unblocked commands that request changes in the catalog.

The main purpose of this feature is to relieve the DBA and any other suitably authorized user from the burden of having to redefine or redeclare all the items dropped in cascading action, when it would be necessary to

restore many of them by hand as soon as one or more replacement R-tables are created. A second purpose is to permit the DBMS to optimize its treatment of the block of catalog commands as a whole.

For example, the addition in the catalog of a new column to the description of a base R-table can involve a complete scan of all rows of that R-table to alter the way each row is stored. If two columns are added to a single R-table by means of two consecutive, but unblocked, catalog commands, the DBMS will make two complete scans of all rows of that R-table. On the other hand, if these two commands are placed within a catalog block, then the DBMS can handle both columns by means of just one scan. If the pertinent R-table is large, the gain in performance could be significant.

12.3 ■ Modes of Execution

RM-8 Dynamic Mode

The DBMS supports the following kinds of changes dynamically—that is, without bringing activity on the regular data to a halt, without changing the source coding of any application programs, and without any off-line recompiling of any source RL statements:

1. creating new and dropping old domains, R-tables, and columns for already-declared R-tables;
2. creating new and dropping old representations in storage for parts of the database;
3. creating and dropping performance-oriented access paths;
4. changing the authorization data in the catalog;
5. changing declarations in the catalog (e.g., data types, user-defined functions, integrity constraints).

The relational approach is intentionally highly dynamic. In contrast to non-relational DBMS, it should rarely be necessary to bring the database activity to a halt for any reason.

RM-9 Triple Mode

The same language, RL, can be used in three distinct ways. First, RL can be used interactively at terminals. Second, statements in RL can be incorporated into application programs. Third, statements in RL can be combined to specify the action to be taken in case of attempted violation of an integrity constraint (see Chapters 13 and 14).

Generally speaking, adherence to this feature enables an application programmer to develop and debug the database statements separately from the remainder of the program in which these statements occur.

12.4 ■ Manipulation of Missing Information

RM-10 Four-valued Logic: Truth Tables

The DBMS evaluates all truth-valued expressions using the four-valued logic defined by the truth tables that follow:

P	not P
t	f
a	a
i	i
f	t

P ∨ Q		Q			
		t	a	i	f
	t	t	t	t	t
P	a	t	a	a	a
	i	t	a	i	f
	f	t	a	f	f

P ∧ Q		Q			
		t	a	i	f
	t	t	a	i	f
P	a	a	a	i	f
	i	i	i	i	f
	f	f	f	f	f

In these tables, t stands for TRUE, a for MISSING AND APPLICABLE, i for MISSING AND INAPPLICABLE, and f for FALSE. Note that t, f, i, and a are actual values, and should not be regarded as marked values. Because the sets retrieved and manipulated in database management are all finite, the existential and universal quantifiers can be treated as iterated OR and iterated AND, respectively.

Evaluation of a truth-valued expression according to this logic is executed by the DBMS without assistance from the user. This does not mean that users should be unaware of four-valued logic, but they need not be continuously concerned with the details.

RM/V1 involved only three-valued logic; no distinction was made on the basis of reasons why information might be missing. Such a distinction, however, is made by RM/V2. (For details, see Chapter 8.)

RM-11 Missing Information: Manipulation

Throughout the database, missing database values are manipulated by the DBMS uniformly and systematically, and, in particular, independent of data type.

Users should be able to exploit the full expressive power of the

four-valued predicate logic in RL. In particular, the MAYBE qualifier should be applicable to any truth-valued expression, whether a complete logical condition or just part of such a condition.

(This feature is part of Rule 3 in the 1985 set; see Feature RS-13 in Chapter 2 for the representation part.)

RL should permit the MAYBE qualifier to be applied either to the whole condition or to part of it. When a view is cited within a query command, the DBMS must replace the name of the view by its definition. As a result, the expanded query can contain a condition that originates partly from the original query and partly from the view definition. Exactly one of these, the command or the definition, could have a MAYBE qualifier attached to all of its condition, while the other has no such qualifier. Thus, in the expanded command, the MAYBE qualifier applies to no more than part of the condition. It is, however, appropriate to remember that any part of a condition to which the MAYBE qualifier is attached must be a truth-valued expression.

RM-12 Arithmetic Operators: Effect of Missing Values

A marked database value in a numeric column cannot be arithmetically incremented or decremented by the DBMS, whereas the unmarked values can be subjected to such operators.

If x denotes a numeric database value, A denotes an A-mark, and I denotes an I-mark,

x + x = 2x	x + A = A	A + x = A
A + A = A	A + I = I	I + A = I
I + I = I	x + I = I	I + x = I

A similar table holds for the three arithmetic operators minus, times, and divide. When both arguments are unmarked database values, however, the result is what would be expected from ordinary arithmetic.

RM-13 Concatenation: Effect of Marked Values

A marked value in a character string column cannot be subjected to concatenation with any other string by the DBMS, whereas the unmarked values can.

Let $\wedge$ denote the concatenation operator and x an unmarked character string. Using the symbols A, I as in Feature RM-12,

$$x \wedge x = xx \quad x \wedge A = A \quad A \wedge x = A$$
$$A \wedge A = A \quad A \wedge I = I \quad I \wedge A = I$$
$$I \wedge I = I \quad x \wedge I = I \quad I \wedge x = I$$

12.5 ■ Safety Features

RM-14 Domain-constrained Operators and DOMAIN CHECK OVERRIDE

Those relational operators that involve comparison of database values are normally constrained to compare pairs of values if and only if both are drawn from the same domain (and therefore have the same extended data type).

There should seldom be any need to override this constraint. However, if the need does arise, the qualifier DOMAIN CHECK OVERRIDE (or DCO) may be attached to the command or to an appropriate expression in the command.

See Feature RA-9 in Chapter 18 for the authorization required to use the DCO qualifier. See Feature RQ-9 in Chapter 10 for the DCO qualifier itself, and Feature RJ-6 in Chapter 11 for the DOMAIN CHECK ERROR indicator.

The normal mode of operation helps protect users from formulating RL commands incorrectly, but of course does not provide complete protection.

As just pointed out, those relational operators that involve comparing database values are normally constrained to compare pairs of values if and only if both values in a pair have the same extended data type (see Chapter 3). Occasionally, however, a function may be applied to one or more database values in certain columns to yield a value to be compared with database values in another column or columns. Alternatively, two functions, possibly distinct, may be applied to each pair of comparands before the comparison is carried out. Since ordinary programming languages can be used to implement the function(s) involved, and since these languages do not support the extended data types of the relational model, it is not easy to specify the extended data type of such function-generated values. The following feature should prove helpful.

RM-15 Operators Constrained by Basic Data Type

In using any operator that normally compares pairs of database values to compare a function-generated value with a database value

or a function-generated value with another function-generated value, the requirement of Feature RM-14 that the values to be compared must be of the same extended data type is relaxed: they are merely required to be of the same basic data type (e.g., both character strings or both integers).

RM-16 Prohibition of Essential Ordering

It is never the case that an R-table, whether base or derived, contains an ordering of rows or ordering of columns, in which the ordering itself carries database information not carried by values within the R-table.

If such an ordering were permitted, the information carried in or by that ordering would *not* be retrievable using relational operators. Assuming that such an ordering is *not* permitted, it is always possible to continue using RL in order to pursue a line of investigation by requesting additional queries or manipulations on these results. An example of this feature being ignored is the CONNECT command of the ORACLE product.

RM-17 Interface to Single-record-at-a-time Host Languages

The programming languages FORTRAN, COBOL, and PL/1 are obvious candidates (others may be candidates also) as host languages for any relational language RL. The DBMS must therefore be able to deliver the retrieved relation a block of rows at a time, where a block can be as small as one row, but is preferably many hundreds of rows.

A cursor that traverses the retrieved relation *may be supported* by the DBMS, although it is preferable that the traversal be executed using the HL. Normally this cursor scans from block to block, touching each block only once. Note that this type of cursor does not scan data within the database, but scans retrieved data only. Such a cursor is more easily managed by programmers in a bug-free way than those cursors that scan data within the database.

If the programmer has omitted the ORDER BY clause in his or her relational request, the program should not be based on the assumption that the sequence in which rows of the result are delivered by the DBMS will remain unchanged when a similar request is executed at a later time.

RM-18 The Comprehensive Data Sublanguage

The relational language RL is comprehensive with respect to database management in supporting *all* of the following items interactively at a terminal and by program (see the triple mode feature, Feature RM-9): (1) data definition, (2) view definition, (3) data manipulation, (4) integrity constraints, (5) authorization, and (6) transaction boundaries (BEGIN, COMMIT, and ROLLBACK). (This feature is Rule 5 in the 1985 set.)

In the relational approach, most of these services require the use of four-valued, first-order predicate logic. It seems counter-productive to require users to learn several different languages to make use of this power. Therefore, it does not make sense to separate the services just listed into distinct languages.

As an aside, in the mid-1970s ANSI/SPARC generated a document advocating 42 distinct interfaces and (potentially) 42 distinct languages for database management systems. Fortunately, that idea seems to have been abandoned.

12.6 ■ Library Check-out and Return

In some installations, a database may be used as an engineering tool. It will then contain details of the engineering design of various pieces of machinery. For each piece of machinery, there may exist several versions representing successive improvements in design. Since the creation or modification of a design can take hours or days to conceive and to express in detail, an engineer is likely to spend much more time on changing the database than that required for commercial transactions. Therefore, concurrency control for engineering-type activities must be quite different in nature from that appropriate for commercial transactions. It seems essential that the DBMS provide some support for distinctly engineered versions. The library check-out and return features that follow represent a minimum level of support for those DBMS products that are intended to support computer-aided engineering.

RM-19 Library Check-out

A duly authorized user can retrieve for several hours or days a copy of part of the database representing an engineering version of a piece of machinery (hardware or software) for the purpose of making design changes and creating a new version for that piece of machin-

ery. The DBMS marks the version from which the copy is retrieved as one that is being improved.

RM-20 Library Return

A duly authorized user can store a new version of the design of a piece of machinery in the database. The request to store this version must be accompanied by a new identifier for it. The request is rejected if this identifier already exists as a version identifier in the database.

Exercises

12.1 What is meant by navigating through the database one record at a time? Does the relational model support such navigation? State two reasons for your answer.

12.2 List 10 tasks to which the principal relational language can apply four-valued, first-order predicate logic.

12.3 What is a transaction? Is it safe for the database to lose integrity in some early portion of a transaction if it regains integrity by the end of that transaction?

12.4 What is the purpose of the CAT block? Explain how it works.

12.5 You wish to make the following changes in the database or in its description:

1. Rename one of the base relations.
2. Unload a second relation, drop it from the DBMS, reorganize the data, and reload a reorganized version of it, restoring the same name as before in the catalog.

In each case, is it necessary first to bring all of the database traffic to a halt? When the drop occurs in Case 2, is there a way to prevent all authorization data for this relation from being lost? Or to prevent all views defined on this relation from being lost? Explain.

12.6 List five kinds of activities that are supported dynamically by RM/V2 (i.e., without bringing the traffic on the database to a halt).

12.7 The triple mode feature indicates that the principal relational language can be used in three distinct ways. What are they?

12.8 List the truth values of (1) a OR f, (2) t AND i, (3) NOT i, and (4) NOT a, where t denotes TRUE, a denotes MISSING AND APPLICABLE, i denotes MISSING AND INAPPLICABLE, and f denotes FALSE.

12.9 Under what circumstances does the relational model merely require comparand columns to have the same basic data type, instead of requiring these columns to have the same extended data type? What are the reasons for this relaxation?

12.10 What six capabilities must the principal relational language have, if it is to be comprehensive?

■ CHAPTER 13 ■

Integrity Constraints

Preserving the accuracy of information in a commercial database is extremely important for the organization that is maintaining that database. Such an organization is likely to rely heavily upon that accuracy. Critical business decisions may be made assuming that information extracted from the database is correct. Thus, incorrect data can lead to incorrect business decisions.

In the relational model, the approach to maintaining the accuracy and integrity of the database is preventive in nature. General methods for preventing the database from being damaged by users of all kinds are far easier to conceive than general methods for repairing the damage once it is done.

One major step toward the goal of correctness of the data is the enforcement of integrity constraints by the DBMS. Many of these constraints represent rules pertaining to the business. When enforced, these constraints require the data to be continually consistent with those rules.

Continual and dynamic enforcement is the responsibility of the DBMS itself. Enforcement is totally misplaced if it is made the responsibility of a software package added on top of the DBMS as an afterthought, because such a package can easily be bypassed.

Without doubt, the relational approach is opening up databases to many more people than any previous approach. It is no longer the case that just a few members of an organization can access the data because of the highly specialized skills and knowledge needed. Therefore, far more responsibility must be placed on the DBMS to maintain the integrity of the data. Up to the time of writing this book, DBMS vendors have failed to provide adequate support for the integrity features of the relational model.

Occasionally in this chapter and the next, the term "user-defined integrity constraints" is used. This term means integrity constraints defined by suitably authorized users. Normally, such authorization is assigned to the DBA and his or her staff only, since these are the people ultimately responsible for the correctness of the data.

Many people have the incorrect notion that integrity constraints merely amount to validation of data upon its entry into the database. Integrity constraints, however, are much broader in scope. They may be applicable upon insertion, update, or deletion of data, and the timing of applicability is normally specified as part of the pertinent declarations.

13.1 ■ Linguistic Expression of Integrity Constraints

Early in the development of non-relational DBMS (and also in the development of artificial-intelligence prototypes), the objective was often adopted of casting as much as possible of the system's behavior into data structure. This approach was thought desirable because it might simplify the programming. Actually, the programming often became much more complicated, and the systems became much harder to understand.

In the relational approach, in sharp contrast to non-relational approaches, declaration of user-defined integrity constraints is made largely *independent of the data structure* (both physical and logical) to achieve integrity independence (see Feature RP-3 in Chapter 20). Such constraints must be specified *linguistically* using the principal relational language, RL.

Features RI-25–RI-27 and RI-31 in Chapter 14, together with Features RL-10 and RL-11 in Chapter 22, help users to specify those kinds of constraints that involve inter-set relationships, such as inclusion dependence (the inclusion of one set of database values within another).

13.2 ■ The Five Types of Integrity Constraints

Information about *inadequately identified* objects is never recorded in a relational database. To be more specific, the following two integrity constraints apply to the base relations in *every* relational database, and should be enforced by the DBMS:

1. Type E, *entity integrity*. No component of a primary key is allowed to have a missing value of any type. No component of a foreign key is allowed to have an I-marked value (missing-and-inapplicable).
2. Type R, *referential integrity*. For each distinct, unmarked foreign-key value in a relational database, there must exist in the database an equal value of a primary key from the same domain. If the foreign key is composite, those components that are themselves foreign keys and unmarked must exist in the database as components of at least one primary-key value drawn from the same domain.

Note that the domain concept plays a crucial role in this and in other kinds of integrity. For convenience, the following abbreviations are adopted: PK denotes primary key; FK denotes foreign key.

Cases in which the key is a combination of columns and some (perhaps all) of the component values of a foreign key-value are allowed to be marked as "missing, need special attention." *Those components of such a foreign-key value that are unmarked should adhere to the referential-integrity constraint.* This detail is often *not* supported in today's DBMS products, even when the vendors claim that their products support referential integrity.

Of these two types of integrity, some versions or releases of relational DBMS products support entity integrity. Only a few, however, provide even partial support for referential integrity. The most important reason for this partial support or lack of support is omission of support for the domain concept. Also, support is omitted in most products for primary and foreign keys. In addition, there is failure to support features that would enhance the performance of this kind of integrity constraint, such as domain-based indexes (see Feature RD-7 in Chapter 21).

To a large extent, Version 2 (Release 1) of IBM's DB2 supports referential integrity, a substantial improvement over Version 1. The support is incomplete, however, for the following reasons.

- It fails to include domains supported as extended data types (see Chapter 3).
- The primary key of each base relation is optional, when it should be mandatory.
- There is no TC timing (PK update problem).
- There is no TT timing (cyclic key state problem).
- A foreign key is allowed to cross-refer to only one primary key (when more than one base relation may each have a primary key based on the same domain). There is no partial check on composite FK, for which at least one component is missing and at least one is not missing.
- The only alternative to on-the-fly checking involves use of a utility program (see Sections 13.3 and 13.6, including Feature RI-22).

Before introducing the types of integrity constraints in the relational model, it is worth noting that the terms "integrity constraint" and "violation of an integrity constraint" convey the original motivation for the concept. These terms fail, however, to convey certain important future uses of the concept. Its use is likely to grow beyond maintenance of database integrity into application-oriented actions based on specified states of data arising in the database and on date and time occurrences.

For example, for certain kinds of parts held in inventory, whenever the quantity-on-hand sinks to pre-specified levels, the DBMS may take the action of ordering certain computed or pre-specified quantities of those parts. This kind of use will probably receive more attention in RM/V3.

RI-1–RI-5 Types of Integrity Constraints

Integrity constraints are of five types: (1) D-type or domain integrity (Feature RI-1), (2) C-type or column integrity (Feature RI-2), (3) E-type or entity integrity (Feature RI-3), (4) R-type or referential integrity (Feature RI-4), and (5) U-type or user-defined integrity (Feature RI-5).

In English, an easy way to remember the five types is that the corresponding letters are those of the words CURED (the pleasing one to remember) and CRUDE. All five types must be supported by the DBMS using declarations expressed in RL. In this task the full power of RL, including but not limited to four-valued, first-order predicate logic, must be applicable.

One reason that C-type integrity is part of the relational model is that it makes it possible to avoid the needless complexities and proliferations of domains that are subsets of other domains. For example, suppose that a database contains many currency columns, all of the same currency type (all U.S. dollars, say). Then, only one currency domain need be declared. The range of values that is included in the definition of this domain is wide enough for all company uses. Each column, however, that reflects a more narrowly defined range (maximum expenditure by certain departments, say) would have an additional range constraint applied to the column: in other words, a C-type integrity constraint that the DBMS links with the D-type constraint by logical AND.

Referential integrity is defined and discussed in Section 1.8. Its definition was briefly repeated earlier in this section. User-defined integrity is discussed in Chapter 14.

13.3 ■ Timing and Response Specification

In RM/V2 each integrity constraint is assigned a timing, and there are precisely two types of timing. The timing type TC specifies that integrity-constraint checking is to be executed by the DBMS no later than the end of execution of whatever relational request (normally originating from a user or application program) is now active. The timing type TT specifies that integrity-constraint checking is to be executed by the DBMS at the end of execution of whatever transaction the relational request participates in. Of course, a request may be free of any transaction context: that is, the request does not participate in any transaction. In this case, all those integrity constraints of type TT are inapplicable.

To explain these two timing types in more detail, consider the action taken by the DBMS whenever a relational request is being executed. It is advisable to remember that the normal source of each relational request is either an application program or a user who is interacting with the database using a terminal. On the other hand, the normal source of an integrity constraint is the catalog.

The catalog is the direct source for types D, C, R, and U, whose definitions are explicitly stored in the catalog. It is the indirect source for type E through use of the declarations of all primary keys, and, of course, these declarations are stored in the catalog. The action taken by the DBMS consists of Steps 1–5 for a request that participates in a transaction, and Steps 1–3 only for a request that does not participate in any transaction.

Step 1 Some time during the execution of the request (possibly at the very beginning) the DBMS determines which of the five types of integrity constraints and which of the possibly many instances of these types are applicable to the current request.

Step 2 The DBMS inspects the timing types of each applicable integrity constraint that is not of type E. Integrity constraints of type E are always of timing type TC. Most integrity constraints of types D and C are also of timing type TC.

Step 3 Before the end of execution of the relational request, the DBMS completes the checking of those constraints that are applicable to this request and of timing type TC.

Step 4 The DBMS appends those constraints that are applicable to this request and of timing type TT to a list pertaining to the current transaction.

Step 5 Immediately before committing all changes resulting from a transaction to the database, the DBMS checks all the constraints of type TT in the list of applicable type TT constraints accumulated during the execution of that transaction.

Note that Step 3 is applicable to every manipulative request regardless of whether that request participates or does not participate in a transaction.

Of course, designers of DBMS products may choose to implement early and tuple-by-tuple execution of type TC integrity checking for performance reasons, but then the onus is on them to prove that the total support for integrity checking in their DBMS product covers all the RM/V2 requirements (see Feature RI-22).

RI-6 Timing of Testing for Types R and U

Each constraint specification of Type R or U must include a symbol specifying a timing condition. Thus, whenever the DBMS determines that a particular constraint is pertinent to a command just executed, it must also examine the timing symbol to determine whether the constraint is to be tested either (1) immediately, upon completion of execution of the command being executed (type TC), or (2) as part of the execution of a COMMIT command in attempting to complete a transaction (type TT) and immediately before committing any changes to the database.

In Case (2), it is entirely possible that the DBA may have requested the abortion of any transaction that attempts to violate a particular integrity constraint. Note that the timing types TC and TT are independent of the types D, C, E, R, U cited earlier in Features RI-1–RI-5. Use of T-timing for types D, C, or E, however, will probably be quite rare.

The on-the-fly timing of IBM's DB2 is discussed in Section 13.6.

RI-7 Response to Attempted Violation for Types R and U

Accompanying each R-type and U-type integrity constraint, there must be a *violation response* V, which defines the action to be taken by the system in case of attempted violation of the constraint. The system permits this action to be expressed in RL, in the host language, or in both. Of course, every execution of V is also subject to whatever integrity constraints are applicable to V. These constraints may be of any of the five types D, C, E, R, or U, and either of the two timings TC or TT.

ROLLBACK is an example of a command that should be permitted. It is reasonable, however, for the DBMS to prohibit use of the COMMIT command as part of the violation response, because normally a transaction is in progress (originating from an application program or interactively from a terminal), and execution of that transaction may not be completed.

RI-8 Determining Applicability of Constraints

Before completion of the execution of any RL statement, the DBMS must examine the catalog to see whether any C-timed integrity constraints must be tested. Before completion of the execution of any transaction (indeed, before committing any of the changes to the database), the DBMS must examine the catalog to see whether any T-timed integrity constraints must be tested. Whenever the DBMS finds that a constraint must be tested, it proceeds to execute the specified test.

RI-9 Retention of Constraint Definitions for Types R and U

The DBMS stores the following in the catalog: (1) the definition of the violation response for each instance of a referential-integrity constraint, along with identification by column names of the perti-

nent PK-FK association; (2) the definition of each user-defined integrity constraint, including its violation response.

RI-10 Activation of Constraint Testing

As a consequence of Feature RI-8, integrity constraints are directly activated by the DBMS. They are *not* activated by an explicit call from any application program, and *not* by any user at a terminal. If integrity constraints were activated in either of these ways, it would be all too easy to bypass them altogether.

There are two important reasons for Features RI-9 and RI-10. First, integrity constraints are a concern of the community of users, not just of a single application programmer. Second, the act of keeping the database in compliance with any integrity constraint should not depend on *any voluntary action whatsoever* by any user or programmer (whether it be including the code in his or her program to do the checking or including a call to invoke a checking program).

Now, relational DBMS products have been put on the market without adequate support for integrity constraints. Thus, some users have been forced to place these constraints temporarily in their application programs until the vendor supports them in the catalog, which is where they belong. Consequently, when a vendor introduces support for new types of these constraints should in no way depend on whether each constraint does or does not appear in any application program.

Users of today's relational DBMS products are advised to provide themselves with standard procedures to develop application code so that whatever integrity constraints are incorporated in applications today can be easily identified and removed, when the products are improved in their handling of all five types of integrity constraints.

RI-11 Violations of Integrity Constraints of Types D, C, and E

Violations of integrity constraints of Types D, C, and E are never permitted. If the source of an attempted violation is an application program, the DBMS returns a code indicating that it has not executed the request. Then, the programmer can choose (if appropriate) to include commands in his or her program to bring the program to a complete halt if this code is encountered. If the source is a user at a terminal, the DBMS simply denies the user's request and sends a message explaining the denial.

If a user were allowed to record in the database the immediate properties of an object without recording that object's primary-key value, serious problems would arise in trying to maintain database integrity. For example, if two rows in the R-table for employees have equality in corresponding property values (whenever these values do not happen to be missing from the database), but one or both primary-key values is missing, how can one resolve the following question: Do these two rows represent two distinct employees or just one? In such a case, the count of the number of rows may not be equal to the number of employees in the company.

In the following example of a relation EMP that identifies and describes employees, notice that, in each of the two rows for employee(s) named Knight, the primary-key value is missing. Do these two rows represent two distinct employees or just one?

EMP (EMP#	ENAME	BIRTH_DATE	SALARY	H_CITY	BONUS)
E107	Rook	23-08-19	10,000	Wimborne	5,500
—A	Knight	38-11-05	12,000	Poole	—I
—A	Knight	—A	12,000	Poole	—I
E575	Pawn	31-04-22	11,000	Poole	3,100

"—A" denotes missing-and-applicable; "—I" denotes missing-and-inapplicable. This is a good example of a database that has lost its integrity. It is also one of the simplest of such examples.

13.4 ■ Safety Features

RI-12 User-defined Prohibition of Missing Database Values

For any column of any base R-table other than a column that is a component of the primary key of that table, the DBA can explicitly request that missing database values of specified types be prohibited. As a result, the DBMS will reject as unacceptable any execution of a single RL command that attempts to place an A-mark or an I-mark (whichever has been prohibited) in such a column.

See Feature RI-19 regarding the introduction of such a constraint. The DBMS must *not* require that, if C is a column in which missing values are prohibited, then C must be indexed, because indexes are supposed to be creatable and droppable at any time for *performance reasons only*. A prohibition of missing values of either type, of course, is quite redundant and

unnecessary if the column happens to be part of or the whole of the primary key of the pertinent base R-table. In this case, the entity-integrity constraint is automatically applied (see Feature RI-3). An explicit prohibition of I-marks in any foreign-key column is redundant for the same reason.

In no way, however, does Feature RI-12 make the entity integrity feature RI-3 unnecessary. The mere fact that a column or combination of columns of a base R-table is prohibited from accepting missing database values cannot be interpreted by the DBMS as a declaration that the pertinent combination is the primary key of that base R-table.

Duplicate values are automatically prohibited from each simple or composite column that happens to be the primary key of a base relation. Occasionally, there is a need to prohibit duplicate values from occurring in simple or composite columns *other than the primary key*. The following feature provides this capability.

RI-13 User-defined Prohibition of Duplicate Values

A suitably authorized user can declare of any simple or composite column in a base relation that duplicate values are prohibited from occurring in that column.

The DBMS may either ignore or reject any attempt by the user to apply Feature RI-13 to the primary key, since such an attempt is completely unnecessary. The DBMS prohibits duplicate values from occurring in the primary key without an explicit request to do so.

It is also unnecessary to apply Feature RI-13 to the combination of all columns in a base relation in an attempt to prevent the occurrence of duplicate rows, since the DBMS performs this task without an explicit request to do so.

RI-14 Illegal Tuple

A tuple consisting of nothing but marked values is prohibited from all R-tables, whether base or derived.

Such a tuple is already prohibited from base R-tables because each such table must have exactly one primary key, none of whose component values can be missing. In derived relations, such a tuple is clearly devoid of information.

RI-15 Audit Log

The DBA can request the DBMS to maintain an audit log of at least all the changes committed to the database (both description and contents). The information recorded in this log includes at least the date, time, and identifiers of the user, the terminal, and the application program (if any such program was involved).

The information in this log need not be directly recorded in a manner seen by users as a collection of relations, but it must be dynamically translatable to such a form by a program that is part of the DBMS. The audit log thus generated can be interrogated by any suitably authorized user who makes use of RL. The translating utility can be executed with a frequency specified by the DBA (once a day and once a week must be supported as options).

The term "dynamically" in this context means without bringing the database traffic to a halt.

Most database management systems maintain a recovery log, but this log is often inadequate for auditors to trace who was responsible for each change made to the database (in terms of both description and contents), and at what date and time the changes were made. In a few products, an audit log is supported that goes beyond the requirements of Feature RI-15 by recording all querying activity as well as all manipulative activity.

RI-16 Non-subversion

Languages other than RL may be supported by the DBMS for database manipulation (the relational model does not prohibit such languages). If any of these languages is non-relational (e.g., single-record-at-a-time), there must be a rigorous proof that it is impossible for the integrity constraints expressed in RL and stored in the catalog to be bypassed by using one of these non-relational languages. (Feature RI-16 is Rule 12 in the 1985 set.)

Note that an example of inability to bypass an integrity constraint does not constitute a proof of adequate generality. The following general assertion must be proved: For all possible database requests permitted by the DBMS product, all possible transactions permitted by that product, and all possible integrity constraints permitted by that product, it is impossible to bypass any applicable integrity constraint. Also note that Feature RI-16 is extremely difficult for a system to support if the system is "evolving" from a non-relational architecture to a relational architecture; such a system already supports an interface at a lower level of abstraction than the relational language in which the integrity constraints are specified.

13.5 ■ Creating, Executing, and Dropping Integrity Constraints

RI-17 Creating and Dropping an Integrity Constraint

RL includes a CREATE CONSTRAINT command and a DROP CONSTRAINT command. The CREATE command includes the following items: (1) a name for the constraint distinct from any constraint name currently in the catalog, (2) the type D, C, E, R, or U, (3) the constraint definition, and (4) the timing type TC or TT. Execution of this command causes this information to be stored in the catalog. The DROP command identifies the integrity constraint to be dropped by name. Its execution merely causes the named integrity constraint to be removed from the catalog.

RI-18 New Integrity Constraints Checked

When a new integrity constraint of any type (except the type covered in Feature RI-19) is introduced into the catalog, or an integrity constraint that already exists in the catalog is modified, the activity must be part of a CAT block (Feature RM-7). The DBMS immediately checks those parts of the database that are potentially affected by the new constraint in an attempt to find all violations of that constraint that already exist in the database. The user is notified of each such violation and the DBMS stops execution in the CAT block until the user responds by rectifying the violation. If no such remedial action is forthcoming within a reasonable time, the DBMS rejects the integrity constraint. After all violations of the new integrity constraint in the entire database have been detected and rectified, the DBMS accepts the integrity constraint and completes the CAT block. From then on, the DBMS enforces the constraint.

RI-19 Introducing a Column Integrity Constraint (Type C) for Disallowing Missing Database Values

When a Type C integrity constraint, which disallows the occurrence of missing values in a specified column, is introduced into the catalog, the database at that instant may be inconsistent with this constraint. That is, the pertinent column may happen to contain numerous occurrences of missing values. Enforcement of this integrity constraint in full is therefore delayed in the following sense.

Those marks that already occur in this column at the time of declaration of the constraint are allowed to continue to exist until each and every such mark is updated to an acceptable database value. On the other hand, the DBMS rejects any attempt to update a database value already in the pertinent column to a mark that indicates the value is now missing.

In other words, the introduction of a constraint that disallows missing database values in a specified column is enforced gradually. No new occurrences of marks are allowed, but those that exist in the specified column are allowed to continue to exist until they are updated to database values. This kind of gradual enforcement can be applied to certain types of integrity constraints other than the prohibition of missing values. The next version of the relational model (RM/V3) is likely to include such gradual enforcement as an additional option on these other types.

If the column happens to be a foreign-key column, referential integrity is applied as usual to the non-missing foreign-key values only. Note that a column of a base R-table that is either the whole primary key or a component of the primary key is not allowed to have any database values missing, beginning with the creation of that R-table.

13.6 ■ Performance-oriented Features

In the following feature, the database scope of a command or transaction is discussed. This is the part of the database that could have been adversely affected by execution of the command or transaction. Note that this scope can be broader than just the part of the database that was actually touched by the command or transaction.

For example, a simple update applied to a primary-key value touches only that primary-key value, but it may damage referential integrity in several parts of the database not touched. Those foreign-key values that previously matched this primary-key value (scattered widely in the database and not touched) may be, and probably will be, adversely affected by the change in primary-key value.

RI-20 Minimal Adequate Scope of Checking

When integrity constraints must be checked dynamically either at the end of a command or at the end of a transaction, the DBMS is designed to make this check over that part of the database that could have been adversely affected by the command or by the transaction, but no more than that.

The preceding feature is applicable during DBMS-initiated dynamic checking of integrity constraints. Occasionally, the DBA must be able to initiate a *complete check* of an integrity constraint over an entire relation or over several entire relations if it is a multi-relation constraint. For example, such a check is needed soon after loading a new relation from a non-relational source.

RI-21 Each Integrity Constraint Executable as a Command

One of the commands in RL is intended for DBA-initiated execution of any designated integrity constraint that is stored in the catalog. The name of the particular constraint, not its type, is specified as part of the command to designate the operand. The result of executing the command is a complete listing of all violations of the specified integrity constraint.

This command can be applied to integrity constraints of all five types. To support this feature, it is essential that each occurrence of a particular kind of integrity constraint (e.g., referential) must be given a distinct name (see Feature RN-13 in Chapter 6). The term "occurrence" in this sentence should not be interpreted in a row-by-row sense, but instead at the relational level.

To avoid a reduction in performance when the DBMS checks referential integrity, designers of IBM's DB2 invented the on-the-fly technique. This technique is also applicable to checking other kinds of integrity constraints. As its name implies, an integrity constraint is checked piece by piece as the execution of a command proceeds to affect pieces of the database.

This technique is excellent for attaining acceptable performance, but it fails to support cases in which the execution of a command or transaction initially violates an integrity constraint, but later recovers from this violation. In fact, the transaction concept of System R was invented for this reason.

The occasional inapplicability of the on-the-fly approach is the justification of the following feature. Note that it does not adversely affect performance in those cases in which the on-the-fly technique is applicable in a correct manner.

RI-22 On-the-fly, End of Command, and End of Transaction Techniques

If the DBMS uses the on-the-fly technique as its normal technique for checking integrity constraints, it must be able to resort to an

end-of-command or end-of-transaction technique in those cases when the on-the-fly technique does not work correctly.

IBM's DB2 Version 2 does not support this feature. Consequently, extreme care is necessary when a user wishes to update a primary-key value and have the corresponding foreign-key values similarly updated to conserve the matching of values that existed earlier. Using DB2 and assuming no specially favorable circumstances, neither of the following steps is valid as the first step:

1. update the primary-key value;
2. update one of the foreign-key values.

In both of these cases, the on-the-fly implementation of referential integrity fails.

Is there any way of handling this kind of update in DB2? Yes, but it is very complicated—and needlessly so, especially when a single command is adequate in the relational model (see Features RB-31 and RB-32 in Chapter 4). In the following explanation, assume that the foreign keys are those that refer to the given primary key, that x is the old primary-key value, and that y is the new primary-key value.

1. Copy a new row into the relation S with the same component values as the row whose primary key is to be updated, except for the primary-key value itself, which in the new row is set to y.
2. Update each corresponding foreign-key value from x to y.
3. Delete the row containing the old primary-key value x.

Although this algorithm may appear to be simple, its complexity is enormous. Step 1 involves using host-language commands (SQL alone is inadequate for this task). Step 2 requires the user either

- to know all of the columns that are foreign-key columns with respect to the given primary key column (a very risky assumption due to the highly dynamic nature of relational DBMS), **or**
- to develop a program to scan all the foreign-key declarations in the catalog and find all those that refer to the given primary key.

While this scan and all the remaining actions in Step 2 are taking place, the user of DB2 must ensure through explicit or implicit locking that no other user either introduces a new foreign-key referencing the given primary key, or deletes the row of S that has the new primary-key value.

At least one alternative algorithm handles this problem correctly in DB2, but it is also needlessly complicated. In any event, the algorithm involves multiple SQL statements instead of just one (See Feature RB-33 or RB-34), and therefore degrades performance.

With regard to the problem of updating primary keys, the complexity of DB2—from the user's standpoint and in terms of implementation—stems from the omission of a simple and cheap feature in the relational model (namely, support of domains as extended data types; see Chapter 3). When I introduced the domain concept into database management 20 years ago as part of the relational model [Codd 1971b, 1971a], it was regarded by almost all of my IBM colleagues as a purely academic exercise. It is now time for the implementors of DBMS products to recognize that the domain concept *must* be implemented as part of the DBMS if these products are to provide adequate support for database integrity. Without adequate support for database integrity, DBMS vendors are asking DBMS consumers to put their businesses at unnecessary risk.

Exercises

13.1 What are the five types of integrity in the relational model? Is recoverability from failures in hardware or software directly related to any of these five types? If so, how? Does it matter whether a vendor's DBMS supports any of these forms of integrity?

13.2 Is the relational approach to database integrity based on prevention or on cure? Why?

13.3 Are integrity constraints explicitly invoked from an application program? If not, why not, and how are these constraints invoked?

13.4 What does it mean to say that a cyclic key state exists in the description of a relational database? How does this concept relate to the transaction concept?

13.5 IBM introduced partial support for referential integrity in Version 2 (Release 1) of its DBMS product DB2. List six ways in which this release falls short of full support for referential integrity, and explain the consequences for users.

13.6 Sometimes people assert that it would be adequate if a DBMS always responded to an attempted violation of a referential integrity constraint by rejecting the user's request. Describe an example that demonstrates the need for at least one alternative reaction.

13.7 IBM's Version 2 of DB2 uses the on-the-fly technique during execution of a transaction for checking whether referential integrity is being maintained. Develop a set of necessary and sufficient conditions under which this technique is guaranteed to work correctly.

13.8 If Exercise 13.7 is solved, can the end-of-transaction timing be used by the DBMS as a fallback for checking referential integrity whenever the system discovers that the conditions for correctness of the on-the-fly technique are not in effect? What does this improved support provide in the case of row insertions when there happens to be a cyclic key state?

13.9 Can a user prohibit the occurrence of duplicate values in: (1) a simple column (2) a composite column? If so, how?

13.10 A row is generated in a derived relation, and it contains nothing but marked values.

1. What does RM/V2 do with the row?
2. Can such a row occur in a view?
3. Can such a row occur in a base relation? In each case, explain why RM/V2 behaves this way.

13.11 What are the major differences in content between a recovery log and an audit log?

13.12 Missing values were permitted in a column that has existed for some time. What problems can arise in introducing a new constraint on that column that prohibits the occurrence of missing values?

■ CHAPTER 14 ■

User-defined Integrity Constraints

Integrity constraints other than those of the domain, column, entity, and referential types are needed for relational databases. There are two main reasons. First, these user-defined integrity constraints permit the database administrator to define, in a way that can be enforced by the DBMS, many of the company regulations pertaining to the company operations that are reflected in the database. Second, these constraints permit the database administrator to define, also in a way that can be enforced by the DBMS, many of the government and other regulations that apply to these company operations. Once these constraints are defined and entered into the catalog, the DBMS enforces them. Consequently, there is no need to depend on voluntary compliance by application programmers or end users.

Although the term *user-defined* is applied to these integrity constraints, any user who is attempting to define such an integrity constraint must be authorized to do so (see Chapter 18, "Authorization"). Normally, where the number of users is large and the database is production-oriented, few users are so authorized. The DBA, of course, is one such user, since he or she bears primary responsibility for the safety and accuracy of the database and its compliance with company and governmental regulations. It is likely that any other users similarly authorized would be on the DBA's staff.

As mentioned in Chapter 13, it is important to keep in mind that eventually integrity constraints, especially those of the user-defined type, will be applied not only to keep the database in an accurate state by preventing violations of these constraints, but also to trigger specified pos-

itive actions (that cannot be interpreted as responses to violations) when specified conditions arise in the database. Actually, a small class of this type (clock-triggered actions) is supported by RM/V2.

14.1 ■ Information in a User-defined Integrity Constraint

What information must a user-defined integrity constraint contain? It is easy to see that the four components named in Feature RI-23 are necessary; normally, these four should also be sufficient.

RI-23 Information in a User-defined Integrity Constraint

A user-defined integrity constraint has four components: (1) Timing type TC or TT, (2) those actions by terminal users (TU), application programs (AP), or the date-time clock that trigger the testing of the condition, (3) a specification of the condition to be tested, and (4) the name of a procedure that specifies the action to be taken in case of attempted violation. Both the user-defined integrity constraint and its violation procedure are stored in the catalog.

Let us consider each of these items in turn. The timing type, described in Chapter 13, is set to TC if the specified condition (Item 3 in the preceding list) is to be tested at the end of execution of the triggering command. It is set to TT if the condition is to be tested at the end of execution of whatever transaction includes the triggering command.

Now for those actions by application programs or terminal users that trigger the testing of the specified condition.

RI-24 Triggering Based on AP and TU Actions

The DBMS detects as actions that trigger the testing phase of user-defined integrity constraints at least the following types of encounters: (1) a retrieval from a specified relation, (2) an insertion into a specified relation, (3) an update of a specified relation and column (either not involving an I-marked value or involving an I-marked value), and (4) a deletion from a specified relation. These actions are detected by the DBMS regardless of whether they stem from application programs (AP) or from terminal users (TU).

RI-25 Triggering Based on Date and Time

The DBMS is stimulated to invoke the testing phase of user-defined integrity constraints by the advance of date and/or time to pre-specified absolute values or by the lapse of pre-specified date and/or time intervals from some specified starting date and time.

The timing types TC and TT are inapplicable to integrity constraints for which the triggering is based on date and time.

Each integrity constraint of this type has exactly one absolute date and/or time. If it is to be periodically activated, this is merely the starting time, and a date and time interval need to be specified also. Such a timing is recorded in the catalog as the triggering action of an integrity constraint. If the condition-to-be-tested component of such a constraint is omitted, or if that component is specified and happens to have the value TRUE, action is invoked and it is specified in the integrity constraint as the action-to-be-triggered component.

It is worthwhile to digress for a moment into issues related to implementation. When the DBA enters a clock-triggered integrity constraint into the catalog, the request must indicate (perhaps indirectly) an absolute date and/or time and indicate whether the activation is to be periodic with some specified interval.

The DBMS maintains a queue of date-time combinations ordered so that the earliest is at the top of the queue. This earliest combination is transmitted to the clock as the next date and time to create an alarm. At that time the alarm takes the form of an interruption of the DBMS's activities (at some convenient time, but not significantly delayed). The DBMS then finds the pertinent integrity constraint, and executes its condition part. If the constraint is to be periodically activated, the DBMS places a freshly incremented date-time combination in the queue.

14.2 ■ Condition Part of a User-defined Integrity Constraint

The third component of a user-defined integrity constraint is the specification of a condition to be tested. Such a condition is normally a truth-valued expression of the relational language. This expression must have the value TRUE if the integrity constraint is to be satisfied; the qualifier MAYBE is not permitted in such an expression.

Conditions can be imposed either on states of the database or on changes in states of the database. Consider two conditions that stem from a company's policy. One is imposed on database states; the other, on changes of state:

1. each employee's salary cannot exceed a certain limit determined by the employee's position or job in the company;
2. each salary cannot be increased by more than a certain percentage determined by the employee's position or job in the company.

The following example illustrates the practical reasons for including user-defined integrity constraints in RM/V2 and the information contained in these constraints.

EMP	EMP#	Employee serial number
	ENAME	Employee name
	BIRTH_DATE	Date of birth of employee
	SALARY	Present salary
	JOBCODE	Position or job within company

EMP

(EMP#	ENAME	BIRTH_DATE	SALARY	JOBCODE)
e10	Rook	1923-08-19	17,000	j5
e91	Knight	1938-11-05	12,000	j7
e23	Knight	1938-11-05	14,000	j7
e57	Pawn	1931-04-22	10,000	j9
e01	King	1922-05-27	23,000	j1
e34	Bishop	1930-09-17	16,500	j7

A relation called CONTROL contains for each position held by employees the maximum salary for that position:

CONTROL	JOBCODE	Job within company
	MAXSAL	Limit on salary for the job
	PERCENT	Limit on percentage increase in salary

CONTROL

(JOBCODE	MAXSAL	PERCENT)
j1	30,000	20
j2	25,000	10
j5	20,000	10
j7	15,000	8
j9	15,000	8

Suppose that the salary of an employee is being raised to some new level. Clearly, this employee's row in the relation EMP must be modified. It is the new version of this row that must be checked by the DBMS before the row is committed to the database. It is quite inadequate for the DBMS to check intermediate results that are developed along the way to the new version of the pertinent row. The DBMS is responsible for ensuring that all the values being committed to the database conform to the integrity constraints.

Now, in the example under consideration, the salary increase may be entered from a terminal or may be computed. The DBMS is not responsible for monitoring how the new salary is created. After the new salary has been created and has become a component of the new row, and after this row is ready for commitment to the database and is no longer under the control of the application program or user, then the DBMS must go into action and check that the new salary complies with the pertinent integrity constraints. Regardless of how the salary increase is created, it is only a step on the way to the new salary. Checking this increase as an intermediate result is irrelevant.

That is why integrity constraint 2 in the preceding list is expressed in terms of the new and old salaries, not in terms of a salary increase that may have been generated already. Although these two versions of the increase in salary are very likely to be identical, the possibility of mathematical equality is not at issue. Instead, the question is which *occurrence* of the increase must be checked if database integrity is to be enforced.

Incidentally, the use of the prefixes "new" and "old" makes this kind of integrity constraint easier to write and more comprehensible to those who did not write it. Let the updated salary (when it is under the control of the DBMS) be denoted by new_SALARY.

Suppose that the pertinent employee has jobcode = j.

Condition 1: new_SALARY < CONTROL.MAXSAL
where JOBCODE = j

Condition 2: (new_SALARY − old_SALARY)
< old_SALARY × CONTROL.PERCENT
where JOBCODE = j

The kind of command that is to trigger the testing of these conditions is an update on the SALARY component of a row of the EMP relation. The timing type is TC.

On update of EMP.SALARY: If NOT condition 1, then REJECT
On update of EMP.SALARY: If NOT condition 2, then REJECT

For information on the REJECT command, see Section 14.9.

It is not difficult to conceive of similar examples that stem from government regulations instead of company policy. In one such example, the total year-to-date income tax withheld from each employee's salary must be within 10% of the total tax on the year-to-date salary, where that tax is defined by a formula that conforms to the pertinent law or regulation.

If the preceding condition is not satisfied, it is reasonable to say that an attempted violation of the pertinent integrity constraint has taken place. However, when specifying an appropriate user-defined integrity constraint, it is important to identify the condition when TRUE must trigger the exceptional action by the DBMS, because this is the condition that is required in that integrity constraint.

As an example of a clock-triggered constraint, consider the following command. Starting on October 8 at 3A.M. and every 7 days thereafter, archive the derived relation S consisting of all those rows of R for which the component DONE has the value 1. In a manufacturing company, R might be information about orders of parts, and DONE = 1 might mean that the ordered parts and invoice have been received, and the invoice paid. In an airline, R might be a passenger list for each flight on each day. Moreover, DONE = 1 might mean that the flight has been successfully completed. In this second case, deletion might be more appropriate as the triggered action instead of archiving.

14.3 ■ The Triggered Action

The fourth and final component of a user-defined integrity constraint is the action to be taken in the event the condition is TRUE (see Feature RI-23). This action takes the form of a simple procedure—known as the *triggered action*—encoded in some combination of the principal host language and the principal relational language. Such a procedure is stored in compiled form in the catalog and given a name.

The fourth component of a user-defined integrity constraint is the name of the triggered action procedure. For performance reasons, there might be a symbolic name table *under the covers* that would accelerate access to the code when needed.

14.4 ■ Excecution of User-defined Integrity Constraints

All user-defined integrity constraints (whether of timing type TC or TT) are examined by the DBMS at the end of executing each RL command to determine which ones are applicable. If an applicable constraint is of type TC, it is executed immediately. If an applicable constraint is of type TT, the DBMS notes that this constraint must be executed at the end of this transaction. In this way, the DBMS avoids, at the end of the transaction, a burdensome exploration of the commands within the transaction to determine which integrity constraints are applicable at that time.

Execution of the condition part of a constraint of type TT must be postponed to the end of the transaction. If executed earlier, the condition could evaluate to TRUE (triggering the exceptional action), even though, if executed at the end of the transaction, the condition would evaluate to FALSE (no exceptional action necessary).

Note that it is completely unnecessary for any application program or any terminal user to invoke any integrity constraint at any time. This statement applies to all such constraints, whether user-defined or not. Once an integrity constraint has been defined and entered into the catalog, it is the sole responsibility of the DBMS to invoke it whenever it is applicable. Thus, with a relational DBMS there is no reliance on voluntary action by users in order to maintain the integrity of the database.

Consider what happens whenever a relational command is executed. Let us assume that the user-defined integrity constraints are kept in the catalog in separate tables as shown next (although the relational model does not specify such an organization).

Table	Basis of Constraints
1	Pure retrieval (i.e., no update intended, this is the least important case)
2	Insertion
3	Update
4	Deletion
5	Date and time clock

The implementor may also decide to split each of the first four tables into two by timing type (TC and TT). With either of these organizations, if it is desired to base an integrity constraint upon two of these types of commands (e.g., insertion and update), the constraint must be recorded in two tables (a table for insertions and a table for updates).

Suppose that a relational command is being executed, and that it is one of the four types: pure retrieval, insertion, update, and deletion. Suppose also that this command involves just one relation, say R. Then, at the end of execution of this command, the DBMS scans the table that corresponds to the type of command (Table 1, 2, 3, or 4). From this table, the DBMS selects only those user-defined integrity constraints (if any) that specify the relation R.

For each constraint selected, the DBMS examines the timing type. Suppose that the timing type is TC. Then, the DBMS proceeds to execute the condition part of the integrity constraint. If the result of this execution is FALSE, the DBMS proceeds to the immediately following command. If the result of this execution is TRUE, the DBMS executes the designated procedure for attempted violation. Now, this procedure may simply reject the pertinent command. If so, the DBMS aborts the entire transaction, which means that none of the changes that this transaction would have made to the database are committed.

If the timing type is TT, the DBMS merely notes that the condition part of this integrity constraint must be checked at the end of the transaction. Just before committing the transaction, the DBMS checks whether any

integrity constraints have been postponed to this time. If there are several such constraints, the sequence in which they are executed must not affect the outcome. If the outcome is affected, this is most likely because of an inconsistency between integrity constraints (a DBA error)—this is a potential problem for which the DBA must maintain a careful watch. It will be some time before tools are available to simplify the discovery of inconsistencies between integrity constraints.

14.5 ■ Integrity Constraints Triggered by Date and Time

In many commercial installations certain activities need to be triggered on the basis of date and/or time only. A good example of an activity that needs to be done automatically on a routine basis is archiving of some information in the database. If such tasks are executed automatically, neither the DBA nor anyone else has to watch a calendar or clock. An assumption is that the DBMS has access to a clock within the computer system that registers both date and time, and that this clock can act as a rather sophisticated alarm.

Each integrity constraint triggered by date and time must contain a clause specifying either an absolute or a relative date and time. An example of a relative date is every seven days starting on October 8. An example of a relative time is every 24 hours starting at 3:00A.M. A combination of date and time might be every seven days at 3:00A.M. starting on January 12 at 3:00A.M. Normally, the DBMS acts as soon after the specified date and/or time as the necessary locks are released. Of course, at any time various locks may be held by commands and transactions that are already in the process of being executed.

Note that, if date d and time t are specified as a triggering event, the action to be taken when the combination d, t occurs is precisely that specified in the catalog as the triggered action procedure. As pointed out earlier, the phrases *triggering event* and *triggered action* are more appropriate in this context. Note also that in the case of actions triggered by date and time it is either the *truth* of the specified condition or the absence of such a condition that triggers the action.

14.6 ■ Integrity Constraints Relating to Missing Information

Marked values represent the fact that some information is missing from the database. How are these marked values created at data entry time and at later times? How is the choice made between an A-mark and an I-mark? These questions are answered by an insertion feature, RI-26, and an update feature, RI-27.

RI-26 Insertion Involving I-marked Values

In any tuple that is to be inserted into a database there may be component values missing. For each missing value, the DBMS must determine which of the following is appropriate: (1) a default value based on the source of the request (a terminal or work station or an application program), (2) an A-marked value, or (3) an I-marked value. If none of these is appropriate, the DBMS must reject the insertion of this tuple. Note that item (1) is a real value, and it must therefore comply with all the integrity constraints for this column. On the other hand, items (2) and (3) denote the fact that the value is actually missing.

At data entry time, one or more rows are inserted into a base relation, possibly through a view. If a component of a row is missing from the input, the DBMS examines the description in the catalog of the corresponding column and poses the following sequence of questions to the catalog:

1. Is there a default value based on the terminal or work station from which the input came (e.g., the branch identifier in the case of a bank with many branches)?
2. Which types of mark, if any, are permitted in that column?
3. Is there an integrity constraint that generates the correct type of mark?

If the answer to Step 1 is yes, the sequence terminates after Step 1: the DBMS inserts the default value and accepts the input. If the answer to Step 1 is no, the DBMS proceeds to Step 2. If the answer to Step 2 is that both types of marks are prohibited, the sequence terminates after Step 2, and the DBMS rejects the input.

If in Step 2 the DBMS finds that I-marks are prohibited (as in the case of a foreign key) but that A-marks are permitted, the DBMS terminates the sequence after Step 2, and prepares to insert an A-marked value as the pertinent datum. The actual insertion takes place only if it is in compliance with any existing integrity constraint pertaining to A-marked values in that column. In case of non-compliance, the DBMS terminates the sequence and rejects the input.

Similarly, if in Step 2 the DBMS finds that A-marks are prohibited but I-marks are permitted, the DBMS terminates the sequence after Step 2, and prepares to insert an I-marked value as the pertinent datum. The actual insertion takes place only if it is in compliance with any existing integrity constraint pertaining to I-marked values in this column. In case of non-compliance, the input is rejected.

Now, if both types of marked values are permitted, the DBMS first seeks an explicitly stated preference in the request. If a preference is stated there, the DBMS inserts the preferred type of marked value. If the system fails to find such a preference, it then searches the description of the pertinent column to see which one, A or I, is to be inserted. If a preference is stated in that part of the catalog, the DBMS honors the preference. Otherwise, the DBMS inserts an A-marked value (the default case).

An external symbol is needed for the marks in several cases. Whenever such a symbol is needed, the following are suggested:

Type of Mark	A-mark	I-mark
External symbol	??	!!

RI-27 Update Involving I-marked Values

In any tuple within the database that is to be updated, there may be an attempt to replace a database value or A-marked value by an I-marked value. The DBMS must then search the catalog to determine (1) if I-marked values are prohibited from belonging to that column or (2) if their entry is permitted, but they do not conform to some other integrity constraint. In either case a violation of some integrity constraint is being attempted, and the DBMS must invoke the appropriate violation response.

In the case of an update rather than an insertion, a concern that integrity is preserved arises if the update is an attempt to change a db-value into an I-marked value, or vice versa. The DBMS is designed to seek as a first step a DBA-defined integrity constraint in the catalog pertaining to I-marks and the column affected. If the system finds a pertinent constraint, it checks the constraint and either accepts the update (if the constraint is satisfied) or rejects it (if the constraint is not satisfied). If the DBMS fails to find a pertinent constraint, it accepts the update. In any event, of course, the user must be specially authorized to make such an update. The DBMS treats such authorization quite separately from the enforcement of integrity constraints.

14.7 ■ Examples of User-defined Integrity Constraints

In Section 14.2, the example was discussed of enforcing limits on salary increases by means of user-defined integrity constraints. Now follow some new examples.

14.7.1 Cutting Off Orders to Supplier s3

Suppose that an instruction has been issued within a company that no new orders are to be placed for parts from supplier s3, but that existing commitments to s3 will be completed. Then, user-defined integrity constraints such as the following two are needed:

On insertion into ORDER: If S# = s3, then REJECT
On update of S# in ORDER: If new_S# = s3, then REJECT

14.7.2 Re-ordering Parts Automatically

The first of the following two examples illustrates the need for responding to attempted violations of integrity constraints in more complicated ways than simply rejecting the command or the transaction. In this context, even the term "attempted violation" seems incongruous. A more appropriate term is "triggering event."

The first example involves two relations, one called PART, which identifies and describes the various kinds of parts, and one called REORDER, which provides "standard orders" for use when the quantity-on-hand of a particular kind of part falls so low that it becomes necessary to re-order that part. Suppose that the relations are as follows:

PART	P#	Part serial number	Primary key
	PNAME	Part name	
	SIZE	Part size	
	Q	Quantity of parts	
	OH_Q	Quantity of parts on hand	
	OO_Q	Quantity of parts on order	
	MIN_Q	Minimum quantity of parts of this kind to be stored	

There are only four domains.

	P#	PNAME	SIZE	Q		
PART	(P#	PNAME	SIZE	OH_Q	OO_Q	MIN_Q)
	p1	shaft	10	400	300	300
	p2	wheel	20	850	0	800
	p3	radiator	5	400	200	300
	p4	chassis	12	400	0	200
	p5	bumper	6	620	150	400
	p6	lever	15	420	200	350
	p7	fan	5	500	50	400

REORDER	P#	Part serial number	Primary key
	S#	Supplier serial number (preferred supplier)	Foreign key
	R_Q	Re-order quantity	
	O_D	Date of order	
	. . .	. . . . .	

REORDER (P#	S#	Q R_Q	. . .)
p1	s12	600	. . .
p2	s5	1600	. . .
p3	s5	600	. . .
p4	s17	400	. . .
p5	s6	800	. . .
p6	s2	700	. . .
p7	s8	800	. . .
. . .	. . .	. . .	

Whenever quantities of part p are withdrawn from inventory, and the remaining quantity is less than that specified in the minimum quantity component MIN_Q of the p row of the PART relation, a DBA-defined integrity constraint causes the DBMS to extract a copy of the p row from REORDER, to update the order date to the current date, and to transmit this order to the preferred supplier. The quantity of part p that should be ordered and the preferred supplier of part p are also components of the p row of REORDER. In this context, placing an order is merely constructing a one-line order.

14.7.3 Automatic Payment for Parts

Suppose that it is current company policy to make payment for parts as soon as they are delivered. Consider the action taken at the time of receipt of the parts—namely, the insertion of a new row into the parts-received relation PR. The new row specifies the supplier and the date of receipt. This insertion must trigger generating a check and recording in the PR relation that payment has been made.

If the company policy in this example is to pay at least 21 days later and no more than 31 days later, the problem takes on a new aspect, involving an action that is triggered on a delayed basis. The delay is begun by a delay-triggering event (the receipt of parts accompanied by an invoice). The reader may wish to consider the simple extensions to RM/V2 to enable it to cope with this practical requirement. Extensions of the relational model to cope with this practical need are postponed to RM/V3.

These exercises clearly indicate the need for the host language to be usable in programming the triggered action.

14.8 ■ Simplifying Features

In the late 1960s, I decided to take a close look at how databases were being designed. At that time it was clear to me that there did not exist any engineering discipline upon which database design could be established. One result was that designers found it extremely difficult—and frequently impossible—to explain why they had chosen a particular design. The only reason that appeared meaningful to me was the attainment of acceptable performance on the first application that was developed to run on the database. Of course, this often meant that the database design was inconsistent with attaining good performance on subsequently developed applications.

Two major problems, and hence challenges, presented themselves. First, there was a complete absence of concern for the database as an object that would continue to exist and evolve independently of any collection of application programs that might exist at some instant in time. Second, there was no rational basis for database design because there were no carefully conceived concepts at a sufficiently high level of abstraction. Database design cannot be successfully pursued if the only concepts available are bits and bytes. For these reasons, I developed the first three normal forms [Codd 1971b and 1971c] and created the field of normalization of relations for database management—a field that now requires a textbook of its own to explain adequately.

Normalization was originally conceived as a systematic way (with proper theoretical foundations, of course) of ensuring that a logical design of a relational database would be free from insertion, update, and deletion anomalies. And indeed, designs that are proposed today can be defended on a rational basis! This subject is pursued further in Section 17.5.1.

In developing the logical design for a database, it is now quite usual to consider the following types of dependency: functional, multi-valued, join, and inclusion. These dependencies, however, should not be treated as if they were valuable at database design time only. All of them should remain in effect until the database is redesigned in part or completely. This means that many of these dependencies should be cast in the form of DBA-defined integrity constraints. Certain elementary constraints can be managed by the DBMS without specific instruction from the DBA.

From time to time, but not frequently, it may be necessary to change one or two of the integrity constraints that define the dependencies. Consequently, to establish these dependencies in the first place, and to modify them later, there is a need for a data model that can easily accommodate such changes without impairing the correctness of already developed application programs.

The relational model was designed to accommodate these and other kinds of changes gracefully. Going beyond this adaptability to changes in database design, there is a need for several extensions to the principal

relational language: extensions that simplify the expression of these dependency constraints. Some sample features are listed in Section 14.8.1.

14.8.1 Integrity Constraints of the Database Design Type

These constraints are truth-valued expressions that are applicable whenever the pertinent data is present in the database. Each one is not applicable in those instances (tuples or rows) where any of the pertinent data happens to be missing from the database. In the following discussion, the term "column" should be interpreted as a column that may be simple or composite.

RI-28 Functional Dependency

Column B is functionally dependent on column A: R.A → R.B. For each base relation, the DBMS assumes that all columns that are not part of the primary key are functionally dependent on the primary key, unless otherwise declared.

Using this assumption, the DBA need not declare a very large number of obvious functional dependencies. A simple example is the case of a relation EMP that identifies employees by employee serial number and describes employees by their immediate properties that are of concern to the company, including the department DEPT# to which the employee is assigned. Suppose, however, that EMP also includes an immediate property of the department, namely the contract type CT for that department (incidentally, I am not advocating this step). In this case, the three pertinent functional dependencies within the EMP relation are as follows:

E# → DEPT#

E# → CT

DEPT# → CT.

Of these three functional dependencies, the DBA would need to declare only the third one.

RI-29 Multi-valued Dependency

Column B is multi-valued dependent on column A and column C is independent of B's dependency on A: R.A ↠ R.B / R.C.

The symbol ↠ is intended to distinguish this kind of dependency from the functional dependency of Feature RI-28, in which the symbol → was used.

RI-30 Join Dependency

Column A is join dependent on columns B and C: R.A = R.B * R.C.

RI-31 Inclusion Dependency

Column A is inclusion dependent on Column B. That is, the set of db-values in R.A is a subset of the db-values in R.B: R.A is-in R.B. The DBMS assumes that each declared foreign key is inclusion dependent on (1) its target primary key, if just one target is declared, or (2) the union of its target primary keys, if several happen to be drawn from the pertinent primary domain.

This assumption is justified by the fact that referential integrity must be maintained. The term "is-in" stands for "is included in." As usual, a different syntax may be adopted, but the truth-valued expressions should not be any more complicated than these. As noted earlier, the columns A, B, and C involved may be simple or composite.

This concludes the coverage of user-defined integrity constraints. This type of constraint represents an exciting opportunity for DBMS vendors to demonstrate their technical and inventive capabilities. It also presents new challenges for DBAs. In part this is due to the richness with which conditions can be expressed in the principal relational language, RL.

14.9 ■ Special Commands for Triggered Action

A few commands must be added to RL to enable the DBA to define certain kinds of responses to attempted violations of referential and user-defined integrity by application programs and terminal users. These commands make use of the awareness of the DBMS with respect to the following:

- whether the DBMS is making a C-timed check or a T-timed check;
- if it is a C-timed check, what kind of command is causing the attempted violation;
- whether a primary domain is directly involved;
- whether an application program or an interactive user is involved.

Regarding the last item, the desired response to attempted violation may be different in the case of an interactive user and an application program because the DBMS can communicate with the interactive user. For example, the system can tell the user that his or her request is denied, and supply the reason for this denial.

One of the commands in the immediately following Features RI-32, RI-33, and RI-34 is likely to prove to be what is needed in such a response. There is no requirement, however, that any one of these commands be used in any triggered action.

RI-32 The REJECT Command

If checking is C-timed, reject the command and, if that command is part of a transaction, reject the transaction also. If checking is T-timed, reject the transaction.

RI-33 The CASCADE Command

Case 1: If a primary key is being updated or deleted without using the cascade option in the command, and the CASCADE command is used in the violation response, the DBMS cascades the update or delete to all corresponding foreign keys. Case 2: Let D be a primary domain. If a foreign-key value from this domain D is being inserted in the database as a component of a row, and if there is no equal value in a primary key defined on D, then a new row is inserted into a relation whose primary key is defined on D, and the primary-key value in this row is equal to the foreign-key value just mentioned. This action takes place only if all the non-primary-key columns accept marks. If one or more of these columns does not accept marks, the DBMS executes a REJECT instead.

RI-34 The MARK Command

If the cause of attempted violation can be pinned on non-primary-key column(s) and missing values happen to be acceptable in those columns, mark the corresponding components as missing but applicable. If the marking fails, the DBMS executes a REJECT instead.

Of course, the marking can also fail because of declarations in the catalog that prohibit marks from occurring in certain columns.

Exercises

14.1 If you were a DBA, would you grant users who did not report to you permission to add new integrity constraints to the catalog? Supply reasons for your answer.

14.2 What are the three principal reasons for supporting user-defined integrity constraints? Supply a fourth that is more futuristic in nature.

14.3 What are the four main components of a user-defined integrity constraint? What does each component mean?

14.4 Are user-defined integrity constraints incorporated in application programs? If not, why not, where are they stored, and how are they invoked when they must be invoked?

14.5 What language features make it simpler to express the dependencies of database design as integrity constraints?

14.6 List three commands that must be part of the principal relational language, if that language is to support the types of violation responses that are frequently needed.

14.7 Why should the DBMS support integrity constraints based on date and time? Supply an application-oriented example that illustrates the need for this feature of RM/V2, and that does not involve archiving.

■ CHAPTER 15 ■

Catalog

An important property of the relational model is that both the database and its description are perceived by users as a collection of relations. Thus, with very few exceptions, the same relational language that is used to interrogate and modify the database can be used to interrogate and modify the database description. No new training is needed.

Of course, as we have seen in Chapter 7, there are a few extra commands in RL that deal primarily or even solely with the catalog. These commands cannot be applied to the regular data only. A user who wishes to access the database description or any of its parts must be authorized to do so. Otherwise, the authorization mechanism will prevent the user from gaining access.

The database description is stored in the catalog, which also contains its own description.

15.1 ■ Access to the Catalog

In the relational model, the *catalog* holds the database description. In some relational DBMS products this description is also called the catalog, while in others it is called the directory. Whatever it is called, it should be carefully distinguished from a *dictionary,* which normally includes all of the information found in the catalog, but also contains a large amount of information concerning the application programs that operate on a scheduled or non-scheduled basis upon various parts of the database.

It is important that the DBMS provide very fast access to the information in the catalog, to prevent a major bottleneck. On the other hand, normally the need for speed of access is significantly less in the part of the dictionary that does not include the catalog.

RC-1 Dynamic On-line Catalog

The DBMS supports a dynamic on-line catalog based on the relational model. The database description is represented (at the logical level) just like ordinary data, allowing authorized users to apply the same relational language to the interrogation of the database description as to the regular data. (This feature is Rule 4 in the 1985 set.)

This feature is a very important tool for database administrators. When asked whether a specific piece of information is in the database under his or her supervision, a DBA can rapidly use a simple terminal or workstation to interrogate the database description and obtain the answer, even if the DBMS is on a mainframe system. Pre-relational DBMS products often failed to provide the DBA with this tool.

One consequence of Feature RC-1 is that each user, whether an application programmer or end user, needs to learn only one data model, an advantage that most non-relational systems do not offer. For example, IMS (IBM, n.d.), together with its dictionary, required the user to learn two distinct ways of structuring data. Another consequence is that authorized users can easily extend the catalog so that it becomes a full-fledged, active, relational data dictionary, whenever the DBMS vendor fails to do so.

RC-2 Concurrency

The DBMS has a sufficiently sophisticated concurrency-control mechanism that it can support multiple retrieval and manipulative activities on the catalog, on the regular data, or on both concurrently.

It is important to remember, however, that the catalog can easily become a major bottleneck, since the DBMS must access the catalog when it processes many of the accesses to the regular data, whether by application programs or by terminal users. Therefore, during hours of heavy traffic on the regular data, it is unwise to grant many users the privilege of accessing the catalog. The term "regular data" means data not in the catalog.

15.2 ■ Description of Domains, Base Relations, and Views

Domains, relations, views, integrity constraints, and user-defined functions are each described separately because, to a large extent, they are objects whose existence is mutually independent.

- Many relations may make use of a single domain.
- Some views cite more than one base relation in their definitions.
- Integrity constraints often involve more than one base relation.
- User-defined functions are most often needed in constructing various types of queries.

Domains, base relations, and views are now discussed in that order. Integrity constraints were discussed in detail in the two preceding chapters. In Section 15.3, they are discussed from the standpoint of the catalog. User-defined functions are discussed from the standpoint of the catalog in Section 15.4 and in more detail in Chapter 19.

RC-3 Description of Domains

For each distinct domain (i.e., extended data type) upon which the database is built, the catalog contains its name, its basic data type, the range of values permitted, and whether the comparator LESS THAN (<) is meaningfully applicable to the values drawn from this domain.

Note that, if the comparator < is applicable, then all the other comparators are also applicable. For details of domain description, see Section 3.2.

RC-4 Description of Base R-tables

For each base R-table, the catalog contains at least the following items: (1) the R-table name, (2) synonyms for this name, if any (a DBA option), (3) the name of each column, (4) for each column, the name of an already-declared domain, from which the column draws its values, (5) for each column, which kinds of missing values are permitted (if any), (6) for each column, whether the values are required to be distinct within that column, (7) for each column, constraints beyond those declared for the domain, (8) for each column, the basic data type, if applicable, (9) whether the column is a component (possibly the only one) of the primary key (required

for a base R-table), and (10) for each foreign key, the sequence of columns (possibly only one column) of which it is composed, and the target primary keys (possibly only one) in the database.

Regarding Item 6, it must be possible to request distinctness of values in any column without that column having to be indexed! If the DBMS requires there to be an index in this case, the design is in error in coupling a semantic property (distinctness of values) with a performance-oriented feature (an index).

Regarding Item 9, according to Feature RS-8, each base R-table is required to have exactly one primary key. Regarding Item 10, according to Feature RS-10 (see Chapter 2), each base R-table may have any number of foreign keys, including the possibility of having none at all.

Occasionally a column is encountered in which the values are constant; that is, these values should not be updated, although any value can be removed if the entire row is deleted. Instead of introducing a declaration to this effect as one more property of a column, RM/V2 leaves it to the DBA to use the authorization mechanism to withhold updating privileges on such a column.

The declaration of any composite column is optional; this decision is normally made by the DBA. Each composite column that is declared is an ordered combination of two or more simple columns, all of which belong to a single base relation.

RC-5 Description of Composite Columns

For each composite column declared, the catalog contains its name, the name of each simple component column, and an order-defining integer for each of these simple columns. The order-defining integer is one for the first component, two for the second, and so on.

RC-6 Description of Views

For each view, the catalog contains at least the following items: (1) the view name, (2) synonyms for this name, if any, (3) the name of each simple column, (4) for each column, the name of an already declared domain (unless the column is not directly derived from a single base column), (5) whether the column is a component (possibly the only one) of the primary key (if applicable) of the view, (6) the RL expression that defines the view, (7) whether insertions of new rows in the view are permitted by the DBMS, (8) whether deletions of rows from the view are permitted by the DBMS, and (9) for each column of the view, whether updating of its values is

permitted by the DBMS. For more information on items 7, 8, and 9, see Feature RV-6 in Chapter 16 and the whole of Chapter 17.

The domains (extended data types) of computationally derived columns can be difficult to determine. Present-day host languages normally do not deal with this problem, although it seems necessary for both relational and host languages to deal with it. Hence, determining the domains of computationally derived columns is not a requirement at this time. The basic data type of each computationally derived column, however, should be recorded in the catalog.

15.3 ■ Integrity Constraints in the Catalog

As noted in Chapter 14, integrity constraints that are called user-defined are normally defined by the DBA or by staff reporting to the DBA. Each of these constraints represents company policy and rules, *or* government regulations, *or* database design factors that stem from the meaning of the data.

RC-7 User-defined Integrity Constraints

For each multi-variable integrity constraint of type U (user-defined), the catalog contains its complete definition. This includes its name, the triggering event, timing type, the logical condition to be tested, and the response to any attempted violation of this condition.

The DBMS fails to support this feature if it does not support user-defined integrity constraints. See Feature RI-5 in Chapter 13, and the whole of Chapter 14.

RC-8 Referential Integrity Constraints

For each integrity constraint of type R (referential), the catalog contains its complete definition. This includes its name, its triggering event, its timing type, the keys that are involved, and the response to attempted violation (relating this action to the keys involved).

The DBMS fails to support this feature if it does not support referential integrity constraints (see Feature RI-4 in chapter 13).

Features RC-7 and RC-8 are extensions of Feature RC-3.

15.4 ■ Functions in the Catalog

RC-9 User-defined Functions in the Catalog

For each user-defined function, the catalog contains its name, the source code, the compiled code, the names of relations in the database to which the function requires read-only access, whether the function has an inverse, the name of this inverse, the source code for the inverse, and the corresponding compiled code.

It is certainly permissible for the four types of code cited in this feature to reside in the regular database, especially if the host system has the performance-oriented feature that keeps any data that is frequently a bottleneck cached in fast memory.

15.5 ■ Features for Safety and Performance

RC-10 Authorization Data

The catalog contains all the data specifying which interactive users, which terminals, and which application programs are authorized to access what parts of the database for what kinds of operations and under what conditions (see Chapter 18).

In the relational model, all authorization is based on explicitly stated permission rather than explicitly stated denial. This means that users and application programs are unable to gain access to any part of the database other than those parts that they have been explicitly granted permission to access. The granting of permission must be by means of one or more GRANT commands from a user, such as the DBA, who has the pertinent authorization to grant.

RC-11 Database Statistics in the Catalog

The catalog contains all statistical information about the database that is used by the optimizer in precompiling and recompiling RL commands. This includes at least (1) the number of rows in each base R-table and (2) the number of distinct values in *every* column of *every* base R-table (not just those columns that happen to be

indexed at any specific time). (See also Features RD-8 and RD-9 in Chapter 21.)

Consider two extreme cases. If the catalog contains statistical information about the database, and if the optimizer fails to use any of this information in precompiling or recompiling each RL command, the DBMS fails to support this feature. Similarly, if the catalog does not contain any statistical information at all, the DBMS fails to support this feature, whatever use the optimizer makes of its privately held statistics.

Exercises

15.1 List the major items stored in the catalog. What extra information does a dictionary contain? Which of these components, catalog or dictionary, is used by the DBMS to compile or interpret relational requests?

15.2 How many primary keys can a base relation have? Can the number of primary keys change over time?

15.3 How many foreign keys can a base relation have? Can the number of foreign keys change over time?

15.4 Supply an example of a view that does not have a primary key. Which column(s) constitute the weak identifier?

15.5 List the four items that are required in the description of a *domain*. Supply a reason for each item.

15.6 List the five items that are required in the description of a *user-defined integrity constraint*. Supply a reason for each item.

15.7 List the five items that are required in the description of a *referential integrity constraint*. Supply a reason for each item.

15.8 List the eight items that are required in the description of a *user-defined function*, if that function has an inverse. Supply a reason for each item.

15.9 Concerning statistical information about the database, what is the minimum information required by RM/V2, and where must it be kept?

■ CHAPTER 16 ■

Views

Views are intended to insulate users, including application programmers, from the base relations, allowing (1) changes in definition to be made in the base relations, and (2) corresponding changes to be made in the view definitions, in such a way as to keep the views unchanged in content. Views also permit users to perceive the database in terms of just those derived relations that directly belong in their applications. These views can also be used to confine a user's interaction with the database by approving one or more views as the only way they are authorized to interact with the database.

16.1 ■ Definitions of Views

RV-1 View Definitions: What They Are

Views are virtual relations represented by their names and definitions only. Apart from these names and definitions, the DBMS does not retain any database information (other than DBMS-derived view-updatability information) explicitly for views. The DBMS stores view definitions in the catalog, and supports view definitions expressed in terms of the following three alternatives only: (1) base R-tables alone, (2) other views alone, or (3) mixtures of base R-tables and views.

Consider some examples of views. Suppose that the database includes relations as follows: S stands for suppliers, P stands for parts, and C stands for capabilities of suppliers in supplying parts. The relation S includes columns S# for supplier serial number, SNAME for name of the supplier, CITY for the city in which the supplier is located, and STATUS for a simple rating of the supplier. Suppose that the extension of S happens to be as follows:

S	(S#	SNAME	CITY	STATUS)
	S1	Smith	London	20
	S2	Jones	Poole	10
	S3	Blake	Poole	25
	S4	Clark	London	20
	S5	Adams	New York	15

The relation P includes columns P# for part serial number, PNAME for name of the part, SIZE for size of the part, OH_Q for quantity-on-hand, and OO_Q for quantity-on-order. Suppose that the extension of P happens to be as follows:

P	(P#	PNAME	SIZE	OH_Q	OO_Q)
	P1	nut	10	500	200
	P2	nut	20	235	150
	P3	bolt	5	39	240
	P4	screw	12	50	0
	P5	cam	6	50	8
	P6	cog	15	10	10

The capabilities relation C includes columns S# for supplier serial number, P# for part serial number, SPEED for speed of delivery expressed in business days, UNIT_Q for the quantity that represents a unit in which the part is sold, and PRICE for the cost of the unit quantity when obtained from the specified supplier. Suppose that the extension of C happens to be as follows:

C	(S#	P#	SPEED	UNIT_Q	PRICE)
	S1	P1	5	100	10
	S1	P2	5	100	20
	S1	P6	12	10	6000
	S2	P3	5	50	15
	S2	P4	5	100	15
	S3	P6	5	10	7000
	S4	P2	5	100	15
	S4	P5	15	5	3000
	S5	P6	10	5	3500

Then, an example of a view derived from a single base R-table (a so-called *single-table view*) is the relation that represents the suppliers located in London. Such a view would be represented in the catalog by a formula such as

S [CITY = London],

along with the name of the view and certain properties of the view that are discussed later in this chapter and the next.

The DBMS is designed to evaluate views as infrequently as possible and as partially as possible. If this view were fully evaluated from the base R-table S in the state just indicated, its extension would be as follows:

S	(S#	SNAME	CITY	STATUS)
	S1	Smith	London	20
	S4	Clark	London	20

An example of a more complicated view is the **equi-join** of S on S#, with C on S# represented by

S [S# = S#] C.

If this view were fully evaluated with the database in the state indicated above, the extension would be as follows:

VIEW	(S#	SNAME	CITY	S#	P#	SPEED	. . .	PRICE)
	S1	Smith	London	S1	P1	5		10
	S1	Smith	London	S1	P2	5	. . .	20
	S1	Smith	London	S1	P6	12		6000
	S2	Jones	Poole	S2	P3	5	. . .	15
	S2	Jones	Poole	S2	P4	5		15
	S3	Blake	Poole	S3	P6	5	. . .	7000
	S4	Clark	London	S4	P2	5		15
	S4	Clark	London	S4	P5	15	. . .	3000
	S5	Adams	New York	S5	P6	10		3500

The ellipses (". . .") indicate that the UNIT_Q column has been omitted to conserve space. Note that, just as with base relations, it is normally unnecessary at any time for the user to know what extension any view happens to have at that time.

RV-2 View Definitions: What They Are Not

No view definition is of a procedural nature (e.g., involving iterative loops). Also, no view definition entails knowledge of the storage

representation, access paths, or access methods currently in effect for any part of the database, whether these techniques directly support relations as operands or single records as operands.

This feature makes it simple for any user to define views, whether the user happens to be a programmer or not.

RV-3 View Definitions: Retention and Interrogation

View definitions are created using RL. These definitions are retained in the catalog. They may also be queried using the same language RL used for interrogating the regular data. In both activities—view definition and interrogation of such a definition—the full power of RL, including four-valued, first-order predicate logic, must be applicable.

Retention of view definitions in the catalog is important because views are normally of concern to the community of users, not just one user or programmer.

Figure 16.1 illustrates two views derived from a single base relation. One is a projection; the other, a row selection.

16.2 ■ Use of Views

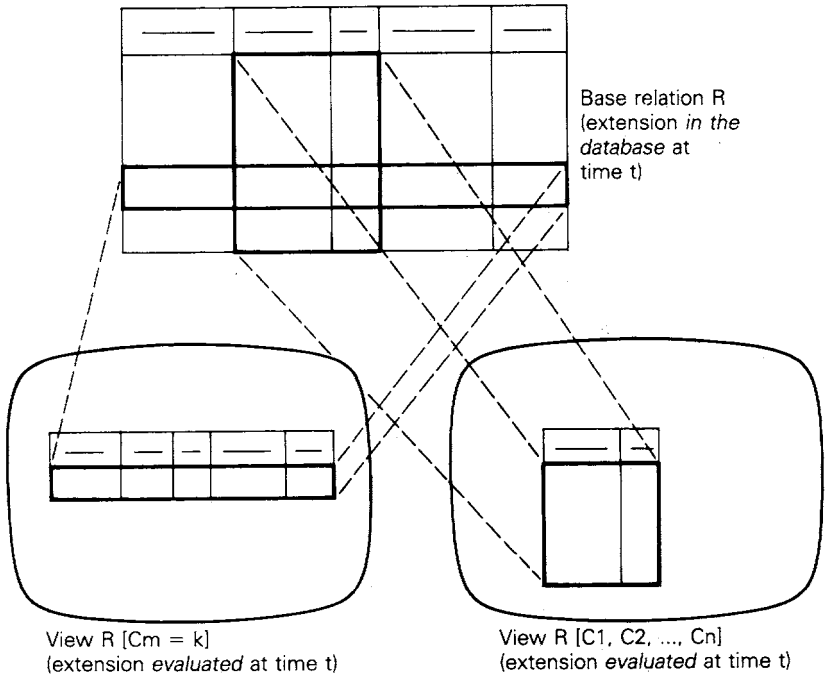
Features RV-4–RV-6 are motivated by a desire (1) to support a powerful authorization mechanism that depends heavily on views, and (2) to protect the user's investment in application programming and in training by requiring programs and users to interact directly with views, instead of the base R-tables.

RV-4 Retrieval Using Views

Neither the DBMS nor its principal relational language, RL, makes any user-visible distinctions between base R-tables and views with respect to retrieval operations. Moreover, any query can be used to define a view by simply prefixing the query with a phrase such as CREATE VIEW.

An example of an undesirable distinction is found in Version 1 of IBM's major database management product, DB2. The operator **union** can be used

Figure 16.1 **Two Views Derived from a Single Base Relation**



in a query on base R-tables, but cannot be used in creating a view. Such a restriction can make life difficult for companies that have similarly structured data at several different sites. Such companies frequently must create a view based on the **union** or **outer union** operator to allow headquarters staff to use the view as a source of planning data.

RV-5 Manipulation Using Views

Neither the DBMS nor its principal relational language, RL, makes any user-visible manipulative distinctions between base R-tables and views, except that (1) some views cannot accept row insertions, and/or row deletions, and/or updates acting on certain columns (Algorithm VU-1 or some stronger algorithm fails to support such action), and (2) some views do not have primary keys and therefore will not accept those manipulative operators that require primary keys to exist in their operands.

For more information on Algorithm VU-1, see Chapter 17, "View Updatability."

RV-6 View Updating

To evaluate the updatability of views at view-definition time, the DBMS includes an implementation of Algorithm VU-1 or some stronger algorithm. Neither the DBMS nor its principal relational language, RL, makes any user-visible manipulative distinctions between base relations and views, except that:

1. some views cannot accept row insertions, and/or row deletions, and/or updates acting on certain columns because Algorithm VU-1 or some stronger algorithm fails to support such action; and
2. some views do not have primary keys (they have weak identifiers only) and therefore will not accept those manipulative operators that require primary keys to exist in their operands.

(This feature is a slightly modified version of Rule 6 in the 1985 set.)

One result of adherence by a DBMS to Feature RV-6 is that all views that are theoretically updatable by Algorithm VU-1 are also correctly updatable by the system without the DBMS having to guess the user's intent. VU-1 tackles a large class of views, including those that are frequently encountered.

Note that a view is theoretically updatable if there exists a time-independent algorithm (based on data description and data content alone) for unambiguously determining a single series of changes to the base relations that will have as their effect precisely the requested changes in the view. Unfortunately, the general problem of determining whether or not a view is theoretically updatable cannot be decided logically [Buff 1986]. Thus, Features RV-5 and RV-6 are related to Algorithm VU-1 (see Chapter 17), which I consider just a beginning in tackling this problem.

In Feature RV-6, the phrase "theoretically updatable" is intended to include insertion and deletion, as well as modification of data that is already in the database. The views handled by VU-1 are those that retain primary keys, in the case of views defined on single relations; those that retain appropriate combinations of primary and foreign keys, in the case of **join**-type views; and those that involve traceability of source, in the case of **union**-type views.

An alternative way of expressing Feature RV-6 is that, in its language(s),

the DBMS must not make any manipulative distinctions between base R-tables and views, except for those views that, according to Algorithm VU-1 or a more powerful algorithm, cannot accept row insertions and/or row deletions and/or updates acting upon certain columns.

If a DBMS handles the view-updatability problem correctly for all those views supported by VU-1, and the vendor claims that its product can handle additional views, the view-updatability algorithm must be made publicly available for analysis, along with a proof that it is strictly more powerful than VU-1.

16.3 ■ Naming and Domain Features

RV-7 Names of Columns of Views

In creating a view, RL permits a user to name any column of this view differently from the way its source column (if such exists) is named. The DBMS however, retains in the catalog the name of the source column (if any), as well as the new name for the pertinent view column.

This feature is required to enable the DBMS to trace back to the base R-table and its appropriate column whenever an update is requested for this view column (see Chapter 17).

RV-8 Domains Applicable to Columns of Views

A view is created using a definition that does *not* indicate, for each column, the domain from which that column draws its values. Apart from the exception cited in the next paragraph, domain identification is deduced by the system at view-definition time, and is stored in the catalog along with the rest of the view definition. If, however, values for that column are computationally derived, then the basic data type (instead of the extended data type) is derived and stored at view-definition time.

Note that the command defining a view provides a one-to-one correspondence between the columns of the operands and the columns of the result, except in the case of computationally derived columns and certain kinds of **union**-type views.

Exercises

16.1 What is the main reason for supporting views? For now, disregard the use of functions to transform values from base relations into values that will appear in views. Does RM/V2 permit views to be defined using (1) the host language only, (2) the principal relational language only, or (3) a mixture of both? What is the answer to this question if functions are included in the view definition?

16.2 Can a view definition involve details concerning storage-representation and access methods in effect at view-definition time? Supply reasons for your answer.

16.3 Can a primary key be deduced for every RM/V2 view? If not, cite an example to support your assertion. What is the row-identifying component called that can be used as an alternative, when necessary? Why is this component bound to identify each row uniquely within any view?

16.4 A user asserts that RM/V2 requires that there should be no user-visible distinctions at all between base relations and views with respect to (1) retrieval, (2) insert, (3) update, and (4) delete. Which of these can be achieved for all views? Explain.

16.5 When a view is created, the domain of each column does not have to be declared. Under what circumstances is the DBMS unable to determine the domain for a column? What does the DBMS do in such a case? Supply one reason why the DBMS must know the domain of each column.

■ CHAPTER 17 ■

View Updatability

In the relational model, a view is a virtual relation represented by its defining declaration, inserted by means of a command such as CREATE VIEW. It is not represented directly by stored data. Insertions, updates, and deletions can be requested as operators upon views in a relational database management system.

Some views, however, cannot accept *some* of these operators unless the system guesses the user's intent. Such guessing is extremely dangerous unless the system checks with the user regarding his or her intent—which is not always possible and, when possible, not always convenient.

The problem discussed in this chapter is the *view-updatability problem:* how to design the DBMS so that it is able to determine whether a request for an insertion, update, or deletion can be honored without guessing the user's intent. I introduce two algorithms, VU-1 and VU-2, as a first step in solving this problem for the whole range of basic operators in the relational model.

Before proceeding, it is useful to consider two simple examples of nonupdatable views to make sure that all readers understand the problem. First, suppose that the database contains a relation EMP that uniquely identifies employees by means of the primary key EMP# and provides their immediate properties:

Base: EMP (EMP# NAME BIRTH_DATE GENDER
SALARY . . .)

Now suppose a single-table view E is created that is the projection of EMP onto two columns, neither of which is the primary key, say GENDER and SALARY. When the view is evaluated, true projection eliminates any duplicate rows. Corrupted projection (supported in some DBMS products, but not part of the relational model) does not. Regardless of whether duplicate rows are eliminated or not, suppose that a user is authorized to delete one or more rows from the view.

Such a request must be reflected in some change applied to the base relations because they are the only relations that reflect the true state of the database. Corresponding to a single row in the view E, there may be many rows in EMP. Thus, the question arises: How can the DBMS decide which row or rows in the base relations must be deleted? Should it delete all the rows in EMP that have the particular combination of gender and salary specified in the request, or should it merely delete an arbitrarily selected row in EMP that has this combination? Whatever it does, the DBMS would be guessing the user's or program's intent; such behavior is unacceptable in managing a shared database.

Now for a second example, this one involving **union** and a view based on two relations, not just one as in the first example. Suppose that two of the base relations in the database are SE and SW, where SE provides the identification and immediate properties of suppliers east of the Mississippi River, while SW provides similar information about suppliers west of the Mississippi. Suppose also that SE and SW are union-compatible and that neither SE nor SW contains a column that indicates directly by its values whether the supplier is east or west of the Mississippi.

Base: SE (S# SNAME CITY STATE . . .)
Base: SW (S# SNAME CITY STATE . . .)

Now, suppose that a view S is created as the **union** of SE and SW. Suppose also that a user is authorized to enter a new row into the view S. Such a request must be reflected in some change applied to the base relations, which are the only relations that reflect the true state of the database. How does the DBMS decide which of the two base relations SE and SW is to be the recipient of this row? Even if two of the immediate properties of suppliers recorded in SE and SW are the city and state in which each supplier is located, it is not appropriate to assume that the DBMS or the database has any knowledge about geography, and in particular about which cities and states are on which side of the river.

It is worth noting that, in this second example, the view S is actually the disjoint **union** of SE and SW, a reasonably simple case; still, however, entry of new rows into the view is not admissible. Nevertheless, whatever it does, the DBMS would be guessing the user's or program's intent, and such behavior is unacceptable in managing a shared database.

Returning to the more general aspects of view updatability, in an article published in two parts in *Computerworld* [Codd 1985], I specified 12 rules intended to help users evaluate DBMS products that are claimed to be relational. Rule R6, pertaining to the question of view updatability, asserted in its original form

> All views that are theoretically updatable are also updatable by the system.

This rule was a reaction to the ad hoc nature of the design of many relational DBMS products, specifically in regard to requests for row insertion, updating, and row deletion applied to views. Part of the problem with these systems, as we shall see, was and is their incredible lack of support for primary keys, foreign keys, and domains—incredible because I made it clear to the designers well in advance that it was important not to omit these particular features.

A few months after publication of the 1985 *Computerworld* article, I received a letter from H. W. Buff [1986] of a Swiss re-insurance firm in Zurich proving that the general question of whether a view is updatable cannot be decided in the logical sense. This means that there does not exist any general algorithm to determine whether an arbitrary view is updatable or not. What, then, can be designed into the system, if it is to be reasonably systematic in its support of views, and yet avoid unreasonable overhead?

First, consider the question of whether a user is authorized to access data through a specific view, and whether he or she can cause the DBMS to take actions such as insertion, update, and deletion in accordance with this view. It is important to observe that this question can be separated completely from the topic of view updatability discussed in this chapter. In fact, the relational model *requires* these two topics to be treated separately from one another. Authorization is discussed in Chapter 18.

In the approach adopted here, as a *first step* I define an algorithm that determines for any given view whether it belongs to an elementary class of views, each of which is clearly updatable in a non-ambiguous manner. If the view is found not to belong to this class, the system merely reports it cannot handle the request, avoiding any assertion that the view is not updatable at all. In Section 17.6, I cite a reasonable change to the 1985 form of rule R6, intended to reduce the possibility that it might be misleading.

One of the reviewers for this book stated that [Dayal and Bernstein 1982] and [Keller 1986] reported independent work on view updatability that is somewhat similar to the approach I describe in this chapter. I regret that, at the time of writing this book, I was unaware of this work and still have not seen the papers.

One approach to view updatability that does not represent a solution places the burden completely on the DBA staff, in the following sense. For each view, the DBA is required to supply a program that translates each

kind of action on the view into corresponding actions on one or more base relations. There may have to be an escape mechanism of this kind, but it should not be the routine mechanism for handling views.

17.1 ■ Problem-oriented Definitions

The term "tuple" is frequently used in this chapter. The reader is reminded that a tuple of a relation is a row of an R-table. In the title and in this chapter so far, the term "updatability" is used in making a general reference to the collection of operators: tuple **insertion,** tuple **deletion,** and **update** of specific components of a tuple that already exists in the database.

Now it is necessary to be more specific, clearly distinguishing among these three kinds of operators.

A view is considered *tuple-insertible by a DBMS,* if the DBMS accepts any collection of tuples (all of the same type and all compatible with the relation type) as an insertion to the view and correctly executes this insertion, provided only that the set of tuples and every one of its components meet the integrity constraints in the context of the transaction in process, and no help is needed from the user to resolve any ambiguities.

Similarly, a view is considered *tuple-deletable by a DBMS,* if the DBMS accepts and correctly executes a request to delete any subset of its tuples, provided only that such a **deletion** meets the integrity constraints in the context of the transaction in process, and, once again, that no help is needed from the user to resolve any ambiguities.

In dealing with the **update** operator, it is necessary to consider the action for each component of each of the tuples involved, and not deal with it in terms of complete tuples of a view as a whole. A column of a view is *component-updatable by a DBMS,* if the DBMS accepts and correctly executes a request to update that column, provided only that such an **update** meets the integrity constraints in the context of the transaction in process, and no help is needed from the user to resolve any ambiguities.

In each of these three cases, "correct execution" means that, for the requested action upon the view V, the DBMS determines that there is either a unique or a uniquely sensible collection of corresponding changes to be made to the base relations—changes that have as their effect upon the extension of V (a view that is not necessarily materialized) precisely those changes requested on the view. Another way of expressing this is that the changes applied to the view would hold if view V were conceptually changed into a base table.

Again, in each of these three cases, along with the phrase "meets the integrity constraints" goes the phrase "in the context of the transaction in process." This extra phrase is necessary because a single command within a multi-command transaction can validly and temporarily create a violation of any integrity constraint that has T-type timing for testing to see whether there has been an attempted violation. As explained in Feature RI-6 (see

Chapter 13), T-type timing means just before committing the changes resulting from the transaction to the database. Normally, of course, temporary violations in the middle of a transaction are removed by the end of the transaction.

17.2 ■ Assumptions

By way of introduction, a view V for a relational database is defined solely in terms of base relations, other views, or both, using a relational language. If the definition of V happens to involve other views, the occurrences of names of these views can be replaced by their definitions, and so on until the definition of V has been expanded to involve base relations only.

For brevity, the original view definition is called the *unexpanded* version, and the fully expanded definition is called the corresponding *fully expanded* version. Of course, if the definition of a view is given in terms of base relations only, then these two versions are identical.

The two algorithms are collectively called the *view updatability algorithms*. Whenever it is necessary to describe a property that is applicable to both of the algorithms VU-1 and VU-2, the term VU will be used.

It is now appropriate to consider Assumptions A1–A4 underlying both of the proposed view updatability algorithms, VU-1 and VU-2.

17.2.1 Assumption A1

The definition of a view and its consequences with respect to **insertion, deletion,** and **update** of its tuples must be understood by users. Users, however, need not know or exploit any details of the DBMS implementation or storage-representation. This includes as a special case that users need not know or exploit the ordering of tuples in base tables or any internal identifiers for specific tuples (so-called *tuple ids*).

Assumption A1 should not be interpreted as *requiring* all users to understand the view-updatability algorithms; it is absolutely necessary that only the DBA and his or her staff should understand these algorithms. Many users may not wish to concern themselves with this issue. They may prefer to think of a view as if it were a base relation, although I am not advocating this over-simplification.

17.2.2 Assumption A2

The decision regarding whether a view is tuple-insertible, tuple-deletable, or component-updatable can be made on the basis of the following:

- the fully expanded definition of the view (not its extension);
- the declarations of the base tables stored in the catalog;
- integrity constraints in the catalog;

- simple information about any statistical or aggregate functions explicitly involved in the view definition.

The simple information in the last item amounts to whether the function has an inverse, and, if so, the name and program for this inverse.

17.2.3 Assumption A3

The decision regarding whether a view V is tuple-insertible, tuple-deletable, or component-updatable is, on the one hand, dependent on parts of the database description at *view-definition time*. On the other hand, this decision is required to be independent of (1) whether any view other than V is affected by updating on V, and (2) the extension of the database at view-definition time.

This decision can therefore be made without considering as a whole the potential or actual value of that view (the so-called *extension* of that relation) or the actual value of the base relations from which that view is derived.

Thus, it is *not* necessary for algorithm VU to evaluate view V in order to make the decision regarding the updatability of view V. Assuming that the DBMS has decided that the view is updatable in one or more of the three senses cited, the system may have to examine part of the extension of the view whenever it encounters a manipulative request (a particular **insertion, deletion,** or row-component **update**) in order to handle this request correctly.

The decision making by the DBMS in Assumption A3 is concerned with determining whether or not a view is updatable and, if so, in what ways it is updatable. The process of actually applying a request for an **insertion, update,** or **deletion** to a view is entirely different. Nothing in Assumption A3 prohibits this latter request-time process from including inspection of the extension of the pertinent view or of its operands.

17.2.4 Assumption A4

The translation activity invoked at *request time* is not permitted to convert an operator of one type into an operator of a quite different type. More specifically, an **insertion** must be converted into one or more insertions, an **update** into one or more updates, and a **deletion** into one or more deletions.

For users, this constraint makes the updating of views much more comprehensible. Now let us consider the purposes served by each one of the first three assumptions.

17.2.5 Purposes of Assumptions

Assumption A1 is valuable whenever the system responds with a message to the effect that a requested **insertion, deletion,** or **update** is refused. Users

can examine the problem themselves, if they so desire, because an examination does not entail use of the unavailable information.

Assumption A2 permits the decision algorithm to be independent of the implementation of any relational DBMS.

Assumption A3 enables the DBMS to determine the tuple-insertible, tuple-deletable, and component-updatable characteristics of a view at the time of entry of the view definition—when it should be done—instead of every time the view is used. Suppose that a suitably authorized user, perhaps the DBA, requests a change in the database description that might affect the updatability of one or more existing views. Now, because the view-updatability decision is made at view-definition time and is dependent on parts of the database description, the DBMS must examine what effect, if any, the requested change in database description might have upon this decision. If the DBMS finds that the decision might be altered, it must re-execute Algorithm VU exactly as if the pertinent view definition had just been entered into the catalog.

The assumptions underlying Algorithm VU-2 consist of Assumptions A1–A4, together with one additional assumption, A5.

17.2.6 Assumption A5

For any view that the DBA or any other authorized user can introduce into the catalog, *interpretation algorithms* determine the action to be taken when a request is made for an **insertion, update,** or **deletion** to be applied to this view.

17.3 ■ View-updatability Algorithms VU-1 and VU-2

These algorithms are alternatives; only one is needed to make a decision regarding view updatability. Thus, only one should be implemented in a relational DBMS. Algorithm VU-2 is intended to be strictly more capable than Algorithm VU-1. As will become apparent, however, Algorithm VU-2 depends heavily on the interpretation algorithms of Assumption A5. Also, more research is needed to ensure compliance of both algorithms with Assumption A4.

Each algorithm VU establishes the following for any given view whose definition involves only the basic relational operators:

- whether that view is tuple-insertible,
- whether that view is tuple-deletable, and
- which of its columns, if any, are component-updatable.

These algorithms are intended to be invoked by the relational DBMS whenever it receives the definition of a proposed view. The results generated

by either algorithm are stored in the catalog, and are therefore available to anyone who has authorization to access that part of the catalog.

Incidentally, it may happen that users need a view for retrieval purposes only. Thus, when Algorithm VU encounters the definition of a view and finds that this view is not tuple-insertible, not tuple-deletable, and not component-updatable, it is *not* appropriate for VU to reject the view request altogether. Instead, VU returns its three-fold decision both by turning on appropriate indicators (see Features RJ-12–RJ-14 in Chapter 11) and by recording these results in the catalog along with the view definition.

It is easier to understand Algorithm VU by keeping in mind that the algorithm makes its decision at view-definition time—not later, at request time, when an actual request for a **deletion, insertion,** or **update** on a view is received. It is the responsibility of the DBMS to respond to a request made at request time in a manner that is consistent with the decision made by Algorithm VU at view-definition time. Nevertheless, to explain how VU works at view-definition time, it is necessary to look at examples of the subsequent response by the DBMS at request time.

The main steps in algorithm VU are as follows:

1. Convert the fully expanded view definition from the source language (e.g., QUEL [Relational Technology 1988] or SQL) into a sequence of operations of the relational algebra, a sequence that contains no superfluous operations and no **Cartesian product.**
2. Examine each of the relational algebra operations to determine whether it generates a view that is tuple-insertible, tuple-deletable, or component-updatable, or some combination of these properties.
3. Let property P denote any one of the three properties tuple-insertible, tuple-deletable, component-updatable. Consider the collection of algebraic operations resulting from Step 1. If any one of these operations by itself yields a view that does not have property P, write in the catalog that the given view definition yields a view that does not have property P.

Step 1 was treated in [Codd 1971d] and [Klug 1982].

The remainder of this section deals with the treatment of each basic operator of the relational algebra by algorithm VU. No claim is made that either VU-1 or VU-2 is able to discover *all* the possibilities of tuple **insertion, deletion,** and **updating.** An implementor of VU within a DBMS may find it advantageous to save some of the intermediate results to help later in the actual execution stage of those **insertions, deletions,** and **updates** permitted on the pertinent view.

17.3.1 Prohibition of Duplicate Rows within a Relation

An operator of the relational algebra may have either one or two operands. No operand is permitted to have duplicate rows, and the result does not

contain any duplicate rows. One of the many reasons for adhering to the prohibition of duplicate rows in any relation is that view updatability is impaired if duplicate rows are permitted at any stage (for details, see Chapter 23).

It is important to note that some of the relational languages supported by today's DBMS products are defective in permitting duplicate rows within a relation. SQL is one of these defective languages. Thus, much of what follows is applicable to an SQL environment *only if the following discipline is pursued:*

- specify one primary key for each and every base relation;
- use DISTINCT in every SQL command to which it can be applied;
- avoid use of the qualifier ALL on each and every UNION command.

The potential occurrence of duplicate tuples at any stage means that the DBMS will be unable to trace the origins of each tuple occurrence back to a particular row of the corresponding operand by means of an algorithm that is independent of the database state. Moreover, this inability to trace origins applies quite often even if duplicate tuples are eliminated as a last step in the derivation.

Therefore, let us assume that, for each of the algebraic operators considered next, each operand and each result is assumed to be devoid of duplicate tuples. Let us also assume (but only for the time being) that, for each of the algebraic operators considered next, the operands are base relations and the result is a view. Each operator is examined first with regard to tuple-insertions, then tuple-deletions, and finally updates of tuple-components.

17.3.2 Solution-oriented Definitions

When a component value of a row in a view is to be updated, or when a new row is to be inserted, it is necessary to consider which of the following cases is applicable:

- *The untransformed case:* the pertinent component is a value stored in the database;
- *The transformed case:* the pertinent component is the result of applying some function either to a single value or to several values stored in the database.

The transformed case involves finding the name of the pertinent function in the catalog, searching its description for the name of its inverse (if any exists in the database), and finding where in the database the code for the inverse is stored. This phase of the inspection is called Part 1. In this case, if no inverse exists in the database, Algorithm VU declares that the pertinent

column of the view is not component-updatable, and Part 2 (specified next) is ignored.

A column of a view is *back-traceable* if one of the following conditions is applicable:

- The transformed case: an inverse function exists, and the code for this inverse is retrievable from the database;
- The untransformed case.

Both cases involve tracing the row and its components back to one or more specific rows in the operand relation(s); this task is called Part 2 of the inspection. Whenever Part 2 is successful, the view is said to be *back-traceable with respect to its rows*. Part 2 is described for each operator when dealing with the insertion of rows.

A view is *completely back-traceable* if every row and every column of that view is back-traceable. Part 1 of the inspection is successful if every column of the pertinent view is back-traceable. When Algorithm VU has determined that every column of the view is back-traceable, it proceeds to Part 2 and determines whether every row is back-traceable also. This last part of the inspection, Part 2, is described for each of the basic operators.

17.3.3 General Remarks about the Decision Problem

The view-updatability decision made by Algorithm VU is based on its finding in regard to the back-traceability of the view, as follows:

- If the view is back-traceable with respect to rows, it is tuple-deletable;
- If it is completely back-traceable (rows and all columns) it is tuple-insertible;
- If it is back-traceable with respect to rows and with respect to a specific column, then that column is component-updatable.

Some views lack normalization, even those that the DBMS decides are updatable in one or more of the three respects just cited. For example, a **join** that matches primary-key values in one relation to corresponding foreign-key values in another is not normalized. In these cases, the user or program may encounter update anomalies of the type described in [Codd 1971b].

It is the responsibility of the DBA to declare for each base relation and view whether or not it is fully normalized. The information in this declaration is saved in the catalog.

17.3.4 The Select Operator

Suppose that the view

T = SELECT R (A # x),

where R is a relation, # is one of the comparators applicable to the **select** operator, and x is a constant, a host-language variable, or the name of a second column (say, B) of relation R.

The **select** operator of the relational algebra selects complete tuples from the operand relation. Consequently, if there are no duplicate tuples in the operand, there are none in the result. Thus, the system encounters no problem relating which tuple of the result (the view) corresponds to which tuple of the operand. This view is therefore back-traceable with respect to its rows.

Accordingly, Algorithm VU declares the following for every view based on the operator **select:**

- The view is tuple-deletable;
- If it has nothing but back-traceable columns, then it is tuple-insertible;
- Each column that is back-traceable is component-updatable.

Later, at request time, when a tuple is presented for insertion into the view T, the DBMS checks to see that this tuple satisfies the condition part of the view definition (A # x). If it does not, that particular insertion is rejected, and an error indicator is turned on.

One final remark on updating views: an update to a component value of a row in a view can be non-compliant with the definition of the view. It can cause the entire row to be removed from the view. For example, if a row in the view

$$T = R [A < 100]$$

happens to contain 90 as its value of A, and if a user requests that value be incremented by 25, then the DBMS updates the corresponding value of A in R to 115. The effect of this update is that the pertinent row is removed from the view T.

If the DBA would prefer the request to be rejected, he or she must *either* place an additional authorization constraint upon the user—namely, that no **update** is allowed to take a row out of the view (this is one more reason why authorization should not be based on views only) *or* make use of the *N*-person turn-key feature, Feature RA-5 (see the remarks following Feature RA-6 in Chapter 18).

17.3.5 The Project Operator

Suppose that

$$T = R [A, B, C, \ldots],$$

where A, B, C denote columns of the relation R.

When executed, the **project** operator selects only those columns whose names are cited in the view-defining command. If the primary key is included

in the list of columns to be selected, there is a clear one-to-one correspondence between the rows in the view and the rows in the operand. Such a view is declared by Algorithm VU to be tuple-insertible and tuple-deletable. Although Algorithm VU could make the same decision if a candidate key (and not the primary key) were included in the list of columns to be selected, it does not do this, partly because the class of updatable views would not be significantly enlarged in this way, and partly because RM/V1 and RM/V2 do not require all of the candidate keys for every base relation to be recorded in the catalog.

If, on the other hand, neither the primary key nor any candidate key is included in the list of columns to be selected, there is no guarantee that any row in the end result (a relation) corresponds to precisely one row in the operand. One way of describing this situation is that one or more rows in the result can have *ambiguity of origin*. Hence, for reasons of safety, a view based on **projection** in which the primary key is not preserved must be treated as not tuple-insertible, not tuple-deletable, and not component-updatable. Accordingly, for every view V that is based on the operator **project** and includes the primary key of the operand, Algorithm VU declares that,

- view V is tuple-deletable;
- if V has nothing but back-traceable columns, V is then tuple-insertible;
- each column that is back-traceable is component-updatable.

A possible improvement over both VU-1 and VU-2 in regard to **projection** is for the DBMS to treat as tuple-deletable a projection that includes a column, all of whose values are declared in the catalog to be distinct within the column and in which it is also declared that missing information is not allowed. This is so slight an improvement, however, that it is not included in either algorithm.

It is now appropriate to consider views, each of which is based on two relations.

17.3.6 The Equi-join Operator

A simple example may help the reader to understand the problem. This example includes detailed commentary on the extension of a view, even though the aim is to have the DBMS decide whether the view is tuple-insertible, component updatable, and/or tuple-deletable at view-definition time. Think of this commentary as nothing more than an attempt to explain the problem.

It should be remembered that extensions of base relations, and therefore of views when they are evaluated, are continually changing because of the many interactions by users with the database. One reason that the updatability decision should be made by the DBMS at view-definition time is that it is quite unstable behavior for the DBMS to decide when one request is

made that the view is updatable, then at an immediately following request to decide that it is not updatable, and at still another time to decide that, once again, it is now updatable.

Suppose that R and S are two base relations, each having the present extension shown next. Suppose that column B of R and column C of S draw their values from a common domain. These two columns are accordingly chosen to act as the *comparand columns* in the **equi-join** of R with S. Suppose that V is a view defined by

CREATE VIEW V ← R [B = C] S.

The extension of V is shown next, along with the operand relations R and S:

R (A	B)
a1	1
a2	2
a3	2
a4	3
a5	4
a6	4

S (C	D)
1	b1
2	b2
3	b3
3	b4
4	b5
4	b6
4	b7

V (A	B	C	D)	
a1	1	1	b1	1
a2	2	2	b2	2
a3	2	2	b2	3
a4	3	3	b3	4
a4	3	3	b4	5
a5	4	4	b5	6
a5	4	4	b6	7
a5	4	4	b7	8
a6	4	4	b5	9
a6	4	4	b6	10
a6	4	4	b7	11

The row numbers on V are for explanatory purposes only.

Table 17.1 indicates what should take place when deletions are applied to the view V, *if* the DBMS were to decide on the acceptability of a deletion at the time a request is made. The "if" clause is for explanatory purposes only.

Deletion of any one of Rows 1–5 applied to V can be put into effect by deleting just one row in R only, in S only, or in both base relations. The effect is to delete just one row of V, exactly the one the user requested. It is worth noting that Row 1 involves matching one row of R with one row of S (the comparand value is one), while each of Rows 2 and 3 involves matching *many* rows of R to just *one* row of S (the comparand value is two), and each of Rows 4 and 5 involves matching *one* row of R to *many* rows of S (the comparand value is three).

The reason why the DBMS rejects deletion of any one of the Rows 6–11 of V is that it would be necessary for the DBMS to delete more than one row in V to maintain the view V in conformity with its definition as a **join.**

The effect of this is to delete more information than the user would anticipate, and to make the view behave differently from a base relation.

Table 17.1 **Effect of Deletions on the View V**

Row in V Deleted	Action on R	Action on S
1	Delete row 1	Delete row 1
2	Delete row 2	Nil
3	Delete row 3	Nil
4	Nil	Delete row 3
5	Nil	Delete row 4
6	Reject	Reject
7	Reject	Reject
8	Reject	Reject
9	Reject	Reject
10	Reject	Reject
11	Reject	Reject

For example, deletion of Row 7 in view V < a5, 4, 4, b6 > appears to require deletion of the row < a5, 4 > in R, and the row < 4, b6 > in S. If just these deletions are executed, the rows that disappear from the view V are Rows 6, 7, 8, and 10—three more than the user requested. It is worth noting that, in each of these cases, the relationship between rows of R and rows of T that have a common value in R.B and S.C is *many-to-many*.

It is clearly unnecessary work for the DBMS to decide view updatability each time a request is received for **insertion, update,** or **deletion.** As pointed out earlier, repeated decision making at request time makes the DBMS behave in an unstable fashion.

Furthermore, an early decision regarding view updatability at view-definition time ensures that the decision is not based on the somewhat ephemeral extension of the view. Therefore, the DBMS must know when it can depend on the continued existence of a one-to-one, many-to-one, or one-to-many relationship between those rows of R and those rows of S that have equal comparand values. In the relational model, such relationships are guaranteed, regardless of time and regardless of changes in extension, when one comparand column is a primary key and the other is either a primary key or a foreign key whose values are drawn from the same domain. Both of the algorithms VU take advantage of this fact.

Any additional time-independent relationships that are guaranteed not to be many-to-many must be peculiar to a particular database. These relationships are often represented by a declaration for column R.B that all of the values in R.B are distinct from one another and that there are no missing values in R.B. In other words, R.B is a candidate key for relation R. Then, for any column S.C that draws its values from the same domain as R.B, a one-to-many relationship exists between column R.B and S.C. Only Algo-

rithm VU-2 takes advantage of these relationships in making its decision on view updatability.

Consider the view

T = R [B = C] S,

where B denotes a column of relation R, and C denotes a column of relation S. Suppose also that the simple or composite columns B and C whose values are being compared are as cited in Cases 1, 2, or 3.

1. The primary key of R is being compared with the primary key of S;
2. The primary key of R is being compared with a corresponding foreign key of S;
3. The primary key of S is being compared with a corresponding foreign key of R.

In each of these three cases, one should assume that the keys being compared are drawn from the same domain, since this is what is meant by the term "corresponding" in the three cases.

Now for some terminology. In any relational operation that involves comparing values between a pair of columns (simple or composite), this pair of columns must normally draw its values from a common domain. As introduced earlier, columns being compared are called the *comparand columns*. Between the values in a pair of comparand columns (say R.B, S.C), there may exist a relationship indicated in Row 1 of the table below. The **join** has a corresponding description in row 2:

Row 1	Relationship	One-to-one	One-to-many
Row 2	Join	One-to-one **join**	One-to-many **join**
Row 1	Relationship	Many-to-one	Many-to-many
Row 2	Join	Many-to-one **join**	Many-to-many **join**

Of interest are relationships such as these that are independent of time, not those that happen to exist for a short time because of the data that happens to be active in that time interval. Therefore, one can expect to encounter phrases such as the *time-independent PK-to-FK relationship,* where PK is an abbreviation for primary key and FK is an abbreviation for foreign key.

In Case 1, it makes little sense to permit R = S, since a relation is allowed to have only one primary key. On the other hand, in Cases 2 and 3 it may happen that R and S are either the same or distinct relations. It is assumed, however, that in all three cases the pairs of columns being compared are defined on the same domain; of course, the DBMS checks whether this is the case.

Case 1 is the simplest. No ambiguity of origin can arise in the result. Hence, any view defined as in Case 1 is tuple-deletable; if the view is back-traceable with respect to its columns, it is tuple-insertible also.

It is easy to see that what applies to Case 2 must also apply to Case 3, with the appropriate interchange of relations R and S in the reasoning. In Case 2, if the view is back-traceable with respect to its columns, insertion of a new tuple into the view can easily be put into effect by splitting this tuple into two parts:

1. one part (say p1) corresponding in type to relation R;
2. the other part (say p2) corresponding in type to relation S.

Let the result of appropriately back-transforming p1 and p2 be p1" and p2", respectively. If p1" does not already occur in R, it is inserted into R. If p2" does not already occur in S, it is inserted into S. If both p1" and p2" already occur in R and S, respectively, the DBMS rejects the insertion as an attempt to put a duplicate row into the view, and an error indicator is turned on.

In Case 2, deletion of a tuple that does not exist in the view is rejected, and an error indicator is turned on. Deletion of a tuple that already exists in the view can be handled in a fashion rather similar to that for insertion.

First, split the tuple to be deleted into the two parts p1 and p2 as before. Then, examine the view to see whether any tuple other than the one being deleted has p2" as its type S part. If not, delete p2" from S. Regenerate the view and check it to see whether any tuple other than the one being deleted has p1" as its type R part. If not, delete p1" from R. If no deletion is indicated in either operand, the original deletion command should be rejected as inapplicable, and an error indicator is turned on.

The term *quad* means a contribution of several rows to a **join** arising from a specific value that occurs at least twice in each comparand column (say m times in the first-cited comparand column and n times in the other comparand column). Such a contribution to the **equi-join** must consist of a number of rows that is the product of the two integers m and n. Since each integer is at least two, this product cannot be less than four: hence, the name "quad." Clearly, a quad contribution cannot consist of 3, 5, 7, 11, or any prime number of rows. The integers m, n are the parameters of any selected quad.

Quads are a phenomenon pertinent to the **join** operators when they are applied to operands that have a many-to-many relationship between the comparand columns. This phenomenon should not be confused with the phenomenon of ambiguity in origins, which is applicable to many relational operators.

Suppose that a view is the **equi-join** of two given relations. Suppose also that the comparand-column relationship is many-to-many, and hence one or

more quads can exist in that **equi-join.** Then, the deletion of exactly one row that happens to belong to one of these quads generates a relation that can no longer be the **equi-join** of the given relations. The product *mn* reduced by one cannot be either *m* (*n*-1) or (*m*-1) *n,* because each of *m* and *n* is greater than or equal to two. Similarly, the insertion of exactly one row that happens to expand one of the quads by one row can no longer be the **equi-join** of the two given relations. Algorithm VU-1 does not conduct any searching for quads, since that could be effective only at request time. Instead, at view-definition time, the algorithm rejects any attempt to delete rows from or insert rows into any view that is a many-to-many **equi-join** (that is, an **equi-join** for which there exists a many-to-many relationship between the comparand columns).

Similar remarks apply to any deletion of several rows from (or insertion of several rows into) a many-to-many **equi-join** that leaves any quad in that **join** with a prime number of rows.

The methods of handling insertion and deletion in Cases 2 and 3 work because a PK-to-FK relationship is a time-independent, one-to-many relationship, provided the keys are drawn from a common domain. Therefore, quads cannot occur in the corresponding **join.** Hence, quads and their associated problems are not encountered in a view that is a **join** involving PK and FK columns based on a common domain.

Equi-joins according to columns other than those cited in Cases 1–3 could have quads since the time-independent relationship between the comparand columns *can be* many-to-many. In Algorithm VU, it is not assumed that either the DBA or the DBMS is aware of those cases (if any exist in the given database) in which it happens that the time-independent, non-key, comparand-column relationship is not many-to-many. Therefore, VU rejects as non-updatable all views based on **equi-join** other than those cited in Cases 1–3.

In summary, for every view T based on **equi-join,** Algorithm VU inspects the catalog to see whether Case 1, 2, or 3 applies. If one of these cases is applicable, VU declares the view T to be tuple-deletable. VU also examines every such view T to see whether it has nothing but back-traceable columns. If this additional condition is applicable, VU declares the view to be tuple-insertible. Finally, for each component that is back-traceable, VU declares that component to be updatable. If none of the Cases 1–3 applies to the view T, Algorithm VU declares T to be not tuple-deletable, not tuple-insertible, and not component-updatable.

17.3.7 Inner Joins Other than Equi-joins

Consider a view that is a **join** based on the comparator LESS THAN (<). Suppose that the relation R is joined with the relation S using column A of R and column B of S as comparand columns (A and B may both be simple

or both composite). In such a **join,** each A-value occurrence is likely to be associated with many B-values.

Consider the following example:

T = R [A < B] S.

R (A	C)
2	c1
4	c2
5	c3
9	c5
13	c6

S (B . .)
1
3
4
7
11

T (C	A	B . .)
c1	2	3
c1	2	4
c1	2	7
c1	2	11
c2	4	7
c2	4	11
c3	5	7
c3	5	11
c5	9	11

Note that, in this example, the value 13 in column R.A and the value 1 in column S.B do not participate in the **join** based on LESS THAN. Moreover, the value 9 in column R.A and the values 3 and 4 in column S.B are the only values that occur once in the corresponding columns of the **join** T. Thus, these three values participate in their respective columns and in exactly one row each.

This once-only occurrence permits the DBMS to select a specific row of R (in the case of 9 in R.A) and of S (in the case of 3 and 4 in S.B) to be deleted when the corresponding row in the **join** T is deleted. However, the DBMS cannot make any sensible deletion in R or in S for any other row in the **join.** Since only a relatively few values at the low end in S.B and at the high end in R.A enjoy the once-only occurrence in the **join** T, it seems simplest to reject deletions of rows altogether in any view based on a LESS THAN **join.** This is precisely the action taken by Algorithm VU.

Similar remarks apply to the **inner joins,** for which the comparators are LESS THAN OR EQUAL TO, GREATER THAN, GREATER THAN OR EQUAL TO, and NOT EQUAL TO. However, the four **inner joins,** for which the comparators are limit-imposed (GREATEST LESS THAN, GREATEST LESS THAN OR EQUAL TO, LEAST GREATER THAN, and LEAST GREATER THAN OR EQUAL TO) deserve special attention.

Inner joins can be used with or without the ONCE qualifier. With the ONCE qualifier, each tuple of each operand can be used only in at most one tuple of the result. Only this case is examined here. Treatment of these **joins** is illustrated using the GREATEST LESS THAN comparator and the ONCE qualifier, first assuming that the values in R.A are distinct and that the values in S.B are also distinct.

T = R [A G< B] S ONCE

R (A	C)
2	c1
4	c2
5	c3
9	c5
13	c6

S (B	. .)
1	
3	
4	
7	
11	

T (C	A	B	. .)
c1	2	3	
c3	5	7	
c5	9	11	

A request to delete the row < c3,5,7 > from the view T can be interpreted in one of three ways:

1. delete the row containing 5 from R;
2. delete the row containing 7 from S;
3. delete both rows (5 from R and 7 from S).

Of these three versions, the third is selected by VU because it is both simple and the most symmetric (i.e., lacking in bias).

Now consider an example that is similar in all respects, except that: (1) the values in R.A are not all distinct, and (2) column C in R is explicitly illustrated to distinguish between the two rows of R that contain 5 in R.A.

Note that the result is different from the previous result with respect to the comparand columns:

R (A	C)
2	c1
4	c2
5	c3
5	c4
9	c5
13	c6

S (B	. .)
1	
3	
4	
7	
11	

T (C	A	B	. .)
c1	2	3	
c3	5	7	
c4	5	11	

The request to delete the row < c3, 5, 7 > from the view T is interpreted by VU as a deletion of row < 5, c3 > from R and the row containing 7 from S.

Accordingly, algorithm VU declares that, for every view based on one of the **inner joins** other than **equi-join,** and based on the four limit-imposed comparators with the qualifier ONCE attached:

- that view is tuple-deletable;
- if that view has nothing but back-traceable columns, it is tuple-insertible;
- each column that is back-traceable is component-updatable.

All other views based on **inner joins** (except **natural join** and **equi-join**) are not tuple-deletable, not tuple-insertible, and not component-updatable.

17.3.8 The Natural Join Operator

Both Algorithm VU-1 and Algorithm VU-2 treat views defined as natural **joins** in a way that is very similar to their treatment of views defined as **inner equi-joins.** Removal of the redundant column from the **equi-join** does not affect the action taken by these algorithms significantly.

17.3.9 The Outer Equi-join Operator

Algorithm VU handles **outer equi-joins** in the same way as the **inner equi-joins,** except for the two items that follow. Assume that, in the definition of the view T, the first-cited operand is R and the second is S. There are some differences in the dynamic handling by the DBMS at request time of insertions and deletions because of two facts:

1. The R-part of a tuple to be inserted into T may happen to have all of its component values missing, if the operator is either **right** or **symmetric outer join;**
2. The S-part of a tuple to be inserted into T may happen to have all of its component values missing, if the operator is either **left** or **symmetric outer join.**

17.3.10 The Relational Division Operator

Algorithm VU rejects tuple insertions, tuple deletions, and component updates applied to any view defined using relational division. Such changes have a major effect on the operand relations in terms of which the view is defined. In addition, appropriate interpretations of such operations are not at all clear.

17.3.11 The Union Operator

Consider a view

$$T = R \cup S,$$

where R and S are relations that are union-compatible. Normally it is impossible to deduce from the relation T alone which of its rows came from R, which from S, and which from both R and S.

Deletions of rows from a union-based view are a simple matter and are always accepted by VU. At request time, the deletion of a row from T causes the DBMS to check whether that row occurs in R, in S, or in both. If the row occurs in R, it is deleted from R. If it occurs in S, it is deleted from S.

Algorithm VU-1 on Union Suppose that an insertion of just one row is to be made into the view T. The question arises as to whether this row should

be inserted into R alone, into S alone, or into both. Normally, there is no basis for the DBMS to decide which action to take. If T happens to be the *disjoint union* of R and S at all times of integrity, however, the row inserted into T should be inserted into either R or S, but not both. Under these circumstances, how can the DBMS determine which one? One reasonable source for this information is the user-defined integrity constraints in the catalog. One of these constraints should clearly state the following:

- that T is the disjoint union of R and S (not just any union); **and**
- that the values in a particular simple or composite column of R identify the corresponding rows as originating from R (not S), while the values in the corresponding column of S identify the corresponding rows as originating from S (not R).

When these two catalog-based conditions are satisfied, and every column is back-traceable, Algorithm VU-1 accepts insertions into the view T. For all other views based on the **union** operator, it rejects insertions.

When these two catalog-based conditions are satisfied, each column that is back-traceable is component-updatable. Otherwise, Algorithm VU-1 rejects updating components in such a view.

Algorithm VU-2 on Union When inserting a row into a view that is a **union** of two relations R, S (not necessarily disjoint), it is possible to use a function (normally defined by the DBA) to determine whether the row should actually be inserted into R only, into S only, or into both relations. Such a function is called a *view-interpretation* function.

The DBMS treats this function just like an integrity constraint except that it is neither C-timed nor T-timed (see Feature RI-6 in Chapter 13). Instead, the DBMS examines it at *command-interpretation time* (early in the execution of an RL command); it is said to be *I-timed.*

At view-definition time, Algorithm VU-2 looks in the catalog to see whether a view-interpretation function for this view has been stored there. If so, the DBMS records in the catalog that the requested **union** view is tuple-insertible. If no interpretation function is found for this view, the DBMS resorts to making this decision according to VU-1.

Consider again the example of a **union** view cited at the beginning of this chapter. This view (R UNION S) concerned suppliers west of the Mississippi River (relation R) and those located east of the Mississippi (relation S). Using Algorithm VU-1, this view was found to be *not* tuple-insertible.

Using VU-2 and a suitable view-interpreting function, the view R **union** S can now be treated as tuple-insertible. All that this function need do is to use the city and state components of each row offered to the DBMS for insertion into the view R **union** S. By consulting an extra table stored in the database and indicating which states are west of the Mississippi and which ones are east (Minnesota and Louisiana excluded), the view-defining algo-

rithm, and hence the DBMS, can determine which side of the river the supplier was on, and hence whether to enter the row into R or into S. (In the case of Minnesota and Louisiana, the function must examine a separate table indicating which cities in these states are on which side of the Mississippi (New Orleans straddles the river)).

Pending the development of a clearly superior algorithm, Algorithm VU-2 is being held as a candidate for insertion into RM/V3. Note that VU-2 does not require the operand relations to be disjoint.

17.3.12 The Outer Union Operator

Insertions of rows into a view based on **outer union** are treated by Algorithms VU-1 and VU-2 just as they treat such insertions in the case of **union.** Note, however, that if the row being inserted belongs in one of the operand relations, say R, and if it contains a component value that for R should be missing (since R does not include the corresponding column), that value will be dropped as the insertion into R is made. A similar constraint also applies to S.

For both operands, the DBMS turns on a warning indicator whenever a component value of an inserted row is dropped altogether upon entry into the database. Deletions of rows from a view based on **outer union** are also treated just as in **union.** To execute the **outer union** correctly, however, the DBMS must take note of the type differences between the view T and its operands R and S.

17.3.13 The Intersection Operator

Consider a view

$$T = R \cap S,$$

where R and S are relations that are union-compatible. Then, every row of T occurs in both R and S.

If the user requests deletion of a row from the view T, the DBMS must delete it from both R and S. Algorithm VU-1 declares such a view to be tuple-deletable.

An insertion of a new row into T requires the DBMS to insert the back-transformed version of that row as follows:

- into R if it is already present in S;
- into S if it is already present in R;
- into both R and S if it is present in neither.

Algorithm VU-1 therefore declares such a view to be tuple-insertible, provided every column is back-traceable. Note that, if at request time the back-transformed version of the row to be inserted is already present in

both R and S, the DBMS treats the request as an attempt to create duplicate rows in T, rejects that particular request, and turns on an error indicator.

For a view based on **intersection,** each column that is back-traceable is component-updatable.

17.3.14 The Outer Intersection Operator

Deletions of rows from a view based on **outer intersection** are treated just as in the case of **intersection.** In some circumstances, however, the DBMS must take into account the type differences between the view T and its operands R and S.

Insertions of rows into a view based on **outer intersection** are also treated just as in **intersection.** Note, however, that when making corresponding insertions into the operands R and S, the DBMS must take into account the type differences between the view T and its operands R and S.

Once again, for every view based on **outer intersection,** each column that is back-traceable is component-updatable.

17.3.15 The Relational Difference Operator

Consider a view

$$T = R - S,$$

where R and S are relations that are union-compatible. Then, every row in T is required to occur in R, but must not occur in S.

The DBMS can handle requests for deletions of rows from T by simply making those deletions effective on R. The alternative—introducing the given row into S as a new row—would have the same effect on T as deleting that row from T because of the definition of the view T. This action, however, is deemed inconsistent with user expectations regarding any deletion. Users normally expect every request for a deletion to cause information to be removed from the database.

A user's request to insert a new row in T can be correctly handled by the DBMS if (1) every column in T is back-traceable and (2) the DBMS simply inserts the back-transformed version of that row into R only. As a precaution, the DBMS should check first that the back-transformed version of the row being inserted into R does not already occur in S. If the system finds that the row in question does already exist in S, it should reject the insertion and turn on an error indicator.

Thus, views based on **relational difference** are treated by VU-1 as follows:

- as tuple-deletable;
- as tuple-insertible, provided every column is back-traceable;
- as component-updatable for each back-traceable column.

17.3.16 The Outer Difference Operator

For all views based on **outer difference,** the manipulative activities (deletions of rows, insertion of new rows, and updating of components) are treated just as in the case of **relational difference.** In some circumstances, however, the DBMS must take into account the type differences between the view T and the operands R and S.

17.4 ■ More Comprehensive Relational Requests

Usually a single command in a relational language based on predicate logic will require that a sequence of algebraic operators be executed. The command is decomposed into such a sequence. Then, this sequence is examined by Algorithm VU-1, one operator at a time.

Let us refer to the properties tuple-insertible, tuple-deletable, and component-updatable by the generic name *property P.* Only if three conditions are satisfied does Algorithm VU-1 declare the whole view (defined by the pertinent command) to have property P:

1. every operator in the sequence is found to have property P;
2. at most one outer operator occurs in the view definition, and it generates the final result only;
3. there is no occurrence of any one of the MAYBE qualifiers.

One consequence of this approach is that columns such as primary-key columns that are crucial to property P in one part of a relational command cannot be discarded by projection in another part of that command. For example, the comparand columns in an updatable **join** must be retained in the ultimate result.

In what way does Algorithm VU-2 open up Pandora's box? Unless great care is taken in designing the DBMS as a host to view defining functions, database administrators will be able to use this facility to re-interpret actions on views in extremely irregular ways. For example, an insertion into a view could be re-interpreted as a deletion from that view.

On any column C that the DBA chooses, it is possible for him or her to impose two semantic constraints:

1. that all the values in column C are distinct;
2. that no values are missing from column C.

Since these two constraints are applied by the DBMS to all primary keys unconditionally, it is senseless for the DBA to attempt to impose or drop these constraints on primary keys. Suppose, therefore, that column C is not the primary key of the pertinent relation. In supporting views to which manipulative actions can be correctly applied (without any guessing by the

DBMS), these constraints appear to make column C as good as a primary key.

However, neither Algorithm VU-1 nor Algorithm VU-2 exploits constraints of this type defined by the DBA. The main reason for this "weakness" in these algorithms is that the DBA is free at any time to drop either or both of these constraints—as free as he or she is to introduce them. Such a drop could cause certain views to change drastically with regard to their updatability. Some or all of the ability to delete tuples, update tuples, and insert tuples is likely to be lost when these constraints are dropped.

17.5 ■ Fully and Partially Normalized Views

So far, the question of view updatability has been discussed with little regard for the meaning of the insertions, updates, and deletions. The main concern has been to identify and avoid situations in which the DBMS would have to guess the user's intent because of difficulties in back-tracing from views to base relations, or because of functions that do not have inverses. Now, it is appropriate to bring into focus the fact that an updatable view may not be a fully normalized relation.

I introduced and discussed normalization of relations in 1971 [Codd 1971b and 1971c]. My main goal was to develop some theory that would be applicable to logical database design, and especially to the creation of a sound collection of base relations.

Our main concern here, however, is in the creation of views, not base relations. Now, a view, just like a base relation, may be fully normalized or not. This property holds even if the view is tuple-insertible, component-updatable, and tuple-deletable. If the view is likely to be subjected to many insertions, updates, and deletions, the DBA must examine whether it is normalized or not.

17.5.1 Normalization

Because some readers may not be familiar with the concepts involved in normalizing relations, these concepts are briefly discussed here.

One of the aims of normalizing a collection of relations is to make the insertions, updates, and deletions clear in meaning and therefore easily understandable. Normalization has little to do with pure retrieval. In fact, normalization usually involves breaking relations into relations of smaller degree (those with fewer columns); this tends to reduce performance on pure retrieval because many more **joins** must often be executed.

Every database is intended to model some micro-world. Thus, the objects to which reference is made in the following list are those found in this micro-world. The basic ideas in normalization are to organize the information in a database as follows:

- Each distinct type of object has a distinct type identifier, which becomes the name of a base relation.
- Every distinct object of a given type must have an instance identifier that is unique within the object type; this is called its primary-key value.
- Every fact in the database is a fact about the object identified by the primary key.
- Each such fact contains nothing other than the single-valued immediate properties of the object.
- Such facts are collected together in a single relation, if they are about objects of the same type. The result is a collection of facts, all of the same type.

Note that this methodology makes no distinction between abstract objects and concrete objects. Furthermore, no distinction is made between entities and relationships.

It is the coupling together of facts of different type that gives rise to problems. Such facts are likely to be independent of one another with regard to their truth in the micro-world and their existence in the database. Inserting a fact of one type does not usually require inserting a fact of another type at the same time. Deleting a fact of one type does not normally require deleting a fact of another type at the same time. As I discussed in [Codd 1971b, 1971c], the problem with relations that are not fully normalized is that insertions, updates, and deletions can create unpleasant surprises for users because of anomalies in their behavior and meaning.

Consider an example involving suppliers and simple shipments of parts. A typical fact about a supplier includes an identifier (the supplier serial number), the company name and address, a suitable contact within the company, and his or her telephone number. A typical fact about a simple shipment includes the supplier serial number, the part serial number, the quantity of parts shipped, the date of receipt at the receiving end, the amount to be paid, whether this amount has been paid, and the date of payment.

Suppose that the fact f about each supplier is coupled with the facts g1, g2, . . . gn about shipments from that supplier. Because it is then necessary to repeat f with every g, the first problem noticed is the serious level of redundancy in the relational representation. The adoption of a hierarchic structure to remove this redundancy is a backward step, one that introduces a whole new set of complexities and problems. These have been thoroughly discussed elsewhere (see [Codd 1970]).

It is now appropriate to comment on insertions, updates, and deletions applied to the unduly coupled relation.

Insertion Anomalies Usually the suppliers from which a company acquires its parts constitute a relatively stable collection. On the other hand, fresh

orders are continually being placed with each supplier, and for almost every order there will be a new shipment. Thus, there is a continual need for insertions of new facts concerning new shipments. Every time a new shipment is entered into the database using a normal insert command (see Feature RB-31 in Chapter 4), it must be accompanied by the fact pertaining to the cited supplier, even if that fact already occurs many times in the database. This is clearly an unnecessary burden to place on the users.

When a new supplier is being entered into the database, it is unfortunately necessary to enter information concerning a shipment from this supplier. It would help users if there were a concise way of asserting that the shipment information is missing from the unduly coupled tuple.

Update Anomalies Suppose that one of the suppliers moves from one location to another. A change must be made in the supplier's address, and perhaps in other properties also. This address, however, may occur in many rows of the unduly coupled relation. Unless the user employs a command more sophisticated than the update commands described in Chapter 4 (Features RB-30–RB-32), translation of this request into correct commands is a tedious task, one in which the user must be aware of the unfortunate redundancy cited earlier.

Archiving and Deletion Anomalies Suppose that a particular supplier has five distinct shipments recorded in the database. As just described, each recording of a distinct shipment is accompanied by a repetition of the basic fact about the supplier that is shipping the parts involved.

Suppose that when a shipment is paid for, the database fact pertaining to this shipment is either archived or deleted. With each archiving or deletion, the redundancy level of the fact pertaining to the supplier is reduced by one. It may happen that this supplier receives no fresh orders for such a long time that the level of redundancy is reduced step by step from five down to one, and finally to zero. In the range from five down to one no problem arises because the basic fact concerning the pertinent supplier is retained in the database. In the final step, however, when payment is made for the last of the five shipments, this fact is removed from the database altogether. In this example, as in others, the archiving or deletion proceeds in a regular manner until the final step. Then, and only then, is there a substantial, and probably unexpected, side effect: the total removal from the database of information about the supplier involved.

17.5.2 Relating View Updatability to Normalization

If the definition of a view includes a **join** of some kind, it will not be unusual for the DBMS to make a check to see whether referential integrity has been maintained. Such a check can cause any insert, update, or delete request to be rejected.

The DBMS (and preferably all users) must know which relations are the contributors to every view involving a **join,** allowing insertions, updates, and deletions to be intelligently requested. Thus, if a base relation T is the **outer equi-join** of two relations R and S that are more fundamental than T, but are not base relations themselves, R and S should nevertheless be described in the catalog, and T should be defined in terms of R and S. Such relations are then called *conceptual relations.*

17.5.3 New Operators for Partially Normalized Views and Base Relations

As an aid in explaining these operators, consider a quite useful example based on the **outer equi-join.** Let T denote the **right outer equi-join** of relation S on B with relation K on C:

$$T \leftarrow S\ [\ B\ =\ C\backslash\ ]\ K.$$

It is adequate to consider either **left** or **right outer equi-join** only since the main concern is with T as an updatable view (not a base relation), and it has already been established that a view involving many-to-many matching of values in the comparand columns is not tuple-insertible, not component updatable, and not tuple-deletable.

In the examples of the use of the four new operators presented in Features RZ-41–RZ-44, assume that S denotes suppliers and K denotes capabilities of suppliers in supplying parts. If T happens to be a base relation, assume that S and K are declared as conceptual relations (not base, not view, and not query).

RZ-41 The Semi-insert Operator

An insertion into T of a fact f represented by a semi-tuple is requested. The DBMS examines the pertinent half of T to see whether the fact f already occurs there. If f is already in T, the DBMS rejects the request. If not, the DBMS associates the fact f with either an existing pairing fact that happens to have its other half missing or, if no such attaching point is available, creates such an attaching point by making a copy of a fact that can successfully pair with it.

Consider as an example the insertion of a capability for supplier s3 and part p15. If supplier s3 occurs at all in T with a missing capability, it must occur just once, and the DBMS updates this tuple to include the new capability. If s3 does not occur at all in T, the DBMS rejects the request. If s3 occurs in one or more supplier semi-tuples of T, but always paired

with a capability semi-tuple, the DBMS copies one of these supplier semi-tuples and pairs it with the new capability.

RZ-42 The Semi-update Operator

An update is requested that is to be applied to a fact that is represented by a semi-tuple of T. If the DBMS is able to find at least one semi-tuple to which the update pertains, it proceeds to update every copy of the pertinent fact that exists in T. If the DBMS is unable to find such a semi-tuple, it rejects the request.

Consider as an example of an update to change the address of a specific supplier. Every copy of the supplier semi-tuple that exists in rows of T is similarly updated. If no semi-tuple is found for the specified supplier, the request is rejected.

RZ-43, RZ-44 The Semi-archive and Semi-delete Operators

The DBMS checks to see whether the fact to be archived or deleted occurs in more than one semi-tuple of T. If so, as Step 1, it archives or deletes all rows of T (except one row) in which the fact occurs. As Step 2, the DBMS marks as missing all components of the one remaining semi-tuple of T. If at the start the fact to be archived or deleted occurs only once, Step 1 is omitted and Step 2 is executed. If the fact to be archived or deleted does not occur at all in T, the DBMS rejects the request.

As an example of an archive or deletion, consider the deletion of the capability of supplier s3 to supply part p15. The DBMS checks to see whether supplier s3 occurs in just a single row of T. If so, it marks as missing all components of the capability semi-tuple. If s3 occurs in more than one row, the DBMS makes a simple deletion of the particular row in which the specified capability occurs. If s3 does not occur at all, the DBMS rejects the request.

17.5.4 Outer Equi-join versus Inner Equi-join as Views

Suppose that a decision has been made that facts of two different types must be combined in a single view by making use of either an **outer** or an **inner equi-join** with the primary key of one relation (containing facts of Type 1,

say) matching the foreign key of the second relation (containing facts of Type 2, say). The question arises, Which is the better operator to choose?

Given a fact of Type 1, it is useful to say of a fact of Type 2 that it *matches* the Type 1 fact if the primary-key value in the Type 1 fact equals the foreign-key value (drawn from the same domain) in the Type 2 fact. The following question is crucial: Is it necessary for one or more Type 1 facts to exist in the database when there are no matching Type 2 facts, and for this to be obvious from the content of the view?

If this question is answered affirmatively, the choice is clearly **outer equi-join. Outer equi-join** permits the continued existence of Type 1 facts even when matching Type 2 facts have not been entered or have become obsolete and been archived or deleted.

17.6 ■ Conclusion

View updatability is extremely important because application programs and end users at terminals should always use views as the means of interacting with a relational database—the only way now known for application programs and end users to be able to cope with many kinds of changes in the logical database design without the need for reprogramming and retraining. This is known as *logical data independence.* Algorithms VU-1 and VU-2 are the tools by which relational DBMS products can adequately support determination by the DBMS of view updatability at view-definition time.

The original version of Rule R6 in the 1985 set was stronger than theoretically achievable. In Version 2 of the relational model, Rule R6 has become Feature RV-6 (repeated here).

RV-6 View Updating

To evaluate the updatability of views at view definition time, the DBMS includes an implementation of Algorithm VU-1 or some stronger algorithm. Neither the DBMS nor its principal relational language, RL, makes any user-visible manipulative distinctions between base relations and views, except that

1. some views cannot accept row insertions, and/or row deletions, and/or updates acting on certain columns because Algorithm VU-1 or some stronger algorithm fails to support such action;
2. some views do not have primary keys (they have weak identifiers only) and therefore will not accept those manipulative operators that require primary keys to exist in their operands.

Why do present versions of relational DBMS products handle the updating of views in such an ad hoc, severely limited, and ill-conceived manner?

Performance is not the reason. Perhaps one reason is that, for some time, very few people have been aware that view updatability is a major factor in attaining logical data independence.

Furthermore, if the columns that constitute the primary key are not explicitly declared to be the primary key for *at least* each base relation, then it will be extremely difficult, if not impossible, for the DBMS to determine at view-definition time whether or not that view is tuple-insertible or tuple-deletable, and which of its tuple-components are updatable (if any).

The development of algorithms that, when compared with Algorithms VU-1 and VU-2, are more efficient or more thorough (or both) can be expected. The situation is very similar to that which came about after I completed development of the first three normal forms for database organization [Codd 1971b]. I named them normal forms 1, 2, and 3 to encourage researchers to create additional normal forms—which they did.

Exercises

17.1 What are the four assumptions upon which both of the view-updatability algorithms are founded in RM/V2? What is the fifth assumption on which VU-2 only is based?

17.2 Suppose that a view is defined as a projection on a base relation and does not include the primary key of that relation. Discuss whether view-updatability Algorithm VU-1 would accept or reject the insertion of rows into the view.

17.3 Are all of the views that are definable by RM/V2

- Tuple-insertible by RM/V2?
- Tuple-deletable by RM/V2?
- Component updatable by RM/V2?

If not, why not? Supply one example for each case.

17.4 Supply a brief description of Algorithm VU-1. What are the main improvements in VU-2?

17.5 Suppose that a view is defined as the **equi-join** of two base relations, and this **join** does not involve the primary key of either relation as one of the comparand columns. Discuss whether view-updatability Algorithm VU-1 would accept or reject the insertion of rows into the view.

17.6 As the security chief for a database, you have been asked to make available to a user certain columns of a relation, but these do not include the primary key of that relation. You have also been asked to grant that user the privileges of inserting and deleting rows in the pertinent projection. Should you grant all of these privileges? If not, what are the problems?

17.7 As the DBA of a database, a user has requested you to define a **union** of two base relations as an updatable view. Upon examining the pertinent base relations, you find that they are union-compatible, but that their intersection is either non-empty or not guaranteed to remain empty. Can you grant the user's request, assuming your DBMS supports view-updating Algorithm VU-1? Explain your answer. Can VU-1 be improved to enable the request to be granted? If so, how?

■ CHAPTER 18 ■

Authorization

To quote from Chapter 13,

> Preserving the accuracy of information in a commercial database is extremely important for the organization that is maintaining that database.

A major step cited in Chapter 13 concerned the preservation of integrity. In this chapter a second major step is discussed, namely, controlling who has access to what parts of the database and for what purposes.

Because of its ease of use, the relational approach to database management is without doubt opening up databases to many more people than did any previous approach. No longer can just a few members of an organization with highly specialized skills and knowledge access data. Therefore, far more responsibility must be placed on the DBA, and on the DBMS, to protect the data from damage by people who lack adequate knowledge of the pertinent company operations, procedures, and policies.

Many of those who are authorized to access the data should be permitted by the DBA and DBMS to read the data, but not to modify it. Even when restricted to no more than reading data, such users may be authorized to read only specified parts of the database.

Consider the example of a production database (one that is supposed to reflect the reality of company operations) and a user who is a member of the planning staff. Suppose that this planner must investigate various "what-if" types of questions. He or she may want to make some changes

in the database that reflect possible future changes, either within the company or in its environment, such as changes in the marketplace. Generally, such changes cannot be permitted on data in a production database because that database would no longer reflect the reality of present company operations. A useful approach to this problem is to request the DBMS to deliver to planning staff workstations, with some specified regularity (e.g., once a week), snapshots or summaries of parts of the production database.

The responsibility of the DBA with regard to authorization is to enter into the catalog a collection of statements that specify who is to access what information, for what operational purposes, and under what time constraints. Continual and dynamic enforcement is the responsibility of the DBMS itself. Because there can be millions of accesses every day, it would not be practical to require the DBA to adjudicate every access. Enforcement by software, however, is totally misplaced if it is made the responsibility of a software package added as an afterthought on top of the DBMS. Such a package can easily be bypassed.

It is quite normal in companies and government institutions for a significant variety of kinds of information to be present in a database. Some of this data, perhaps much of it, is not intended to be spread around within the organization (e.g., employees' salaries). Generally, the information should be available to individuals to the extent required by their job and responsibilities. This basis for the availability of information is sometimes called the *need to know*.

Institutions of different kinds often establish quite different procedures intended to safeguard the security of their information. The approach to security and authorization that is incorporated in the relational model is sufficiently flexibile that these institutions can maintain the procedures they are accustomed to using, either without any changes, or, at worst, with only minor changes.

Availability of the information must be distinguished from authorization to modify the information, whether by (1) insertion of new data, (2) update of existing data, (3) archiving of old data, or (4) deletion of obsolete data. For any given part of the database, these four distinct kinds of activities should be separately authorizable. Normally, even fewer people are authorized to engage in these activities than those who are merely authorized to access the information on a read-only basis.

In present relational DBMS products, there is a strong coupling between views and authorization, an idea that probably had its origins in IBM's System R prototype [Chamberlin et al. 1981]. One benefit of this approach to authorization is that it avoids needless complexity in the implementation. The consequences of this coupling, however, must be examined. One major consequence is examined here. Some minor ones are considered later in this chapter.

If one or more programs or users are authorized to manipulate certain rows or columns of a relation, then the scope of this authorization must be expressed in terms of a view containing just those rows and columns. Such

a view must therefore be defined to put such authorization into effect. One consequence is that numerous views are defined for authorization reasons. This means that the capability of the DBMS in terms of view updatability must be strong. Unfortunately, as noted in the preceding chapter, the DBMS products available today are quite weak in this respect.

18.1 ■ Some Basic Features

RA-1 Affirmative Basis

All authorization is granted on an affirmative basis: this means that users are explicitly *granted permission* to access parts of the database and parts of its description instead of explicitly being *denied access.*

In non-relational DBMS, the approach to authorization was (and is) often negative—that is, based on explicit denial of access. In a relational DBMS, if user A grants authorization to user B, user A specifies what B *can* do, not what B *cannot* do. As a consequence, the introduction of new kinds of data into the database does not require urgent examination of any access denials to see how these denials should be extended. Instead, access approvals can be introduced quite safely and gradually as they are conceived and found to be in line with company policies or government regulations.

RA-2 Granting Authorization: Space-time Scope

In granting authorization, the full power of RL (including four-valued, first-order predicate logic) must be applicable in defining (1) the parts of the database and its description accessible for specified purposes (retrieving, inserting, or updating database values, archiving or deleting, or any combination of these activities), and (2) at what time access is permitted (using the date and time functions of RL).

If the applicability of the full power of RL in supporting authorization is achieved through views (the usual method in relational DBMS products today), then Features RV-4 and RV-5, relating to retrieval and manipulating power on views (see Chapters 16 and 17), must be fully supported in the DBMS in a systematic (not ad hoc) fashion.

When a DBMS is designed to support Feature RA-2, the usual approach taken with regard to allocating parts of the database to each user or program is flawed. Access control of this kind is achieved by exploiting views as the

sole tool. To exploit views as a tool is fine because it leads to a simple design for the authorization mechanism. To exploit views as the *sole tool* for this aspect of authorization, however, leads to serious difficulties. The following example illustrates the problem.

Suppose that one of the base relations is the usual employee relation EMP, with employee serial number as the primary key. Suppose also that two of the immediate properties of employees included as columns in EMP are the present job title and the present salary. One possible requirement is that a particular user be allowed access to the entire job title and salary columns for the purpose of analyzing the correspondence between these two factors.

Further, in order to keep the salaries of individuals from becoming public knowledge, suppose that the DBA denies this user access to the primary key by omitting that column from the pertinent view that this user is authorized to access. This is the DBA action that is required by the DBMS that exploits views as the *sole tool* for authorizing what parts of the database can be accessed by each user.

In this particular example, the view to be defined must include the job title and salary columns, and exclude the primary-key column; in other words, the view must be a projection of EMP onto these two columns only. Assuming that the DBMS supports true **projection** and not a corrupted version of this operator, duplicate rows do not appear in the result of projecting the base relation EMP onto job title and salary only, even though there may be duplicates of these pairs of values in the EMP relation. Thus, statistical functions applied to the projection are likely to yield answers that are different from those that would be obtained from the job title and salary columns of the EMP relation itself. Given the intent of the user, the answers obtained from the non-key **projection** are simply wrong.

Does this mean that the definition of **projection** should be altered to permit duplicate rows to be retained in the result? The answer is definitely no, given the seriously adverse consequences of permitting duplicate rows (see Chapter 23).

There is a better solution to the problem. It involves imposing the database space-time constraints on the pertinent user *partly* through a view (in this case, a **projection** that includes the primary key) and *partly* through an additional mechanism that blocks this user from seeing any values in the primary key column.

It is clear that this design of the authorization mechanism is not as simple as the one that exploits views as the *sole tool* but at least it is not obviously wrong in its actions. However, if adopting the use of views as the *sole tool* requires permitting duplicate rows, then the overall simplicity of the DBMS design is significantly reduced by adding this small complexity to the authorization mechanism. Even more important, simplicity for users is achieved (see Chapter 23).

A reasonable question to ask concerning the example just described is, "How does the user distinguish between (1) a view that is a key-based **projection** with the key hidden, and (2) a view that is non-key-based and is a corrupted **projection** (one in which duplicate rows are permitted)?"

The answer is that there is no difference from the standpoint of interrogation and insertion. There could be a difference, however, from the standpoint of update and deletion. I am not advocating that the view-updatability Algorithm VU-1 (see Chapter 17) be extended to handle the case of hidden keys. The main advantage of using an approach based on view 1 rather than view 2 is that no user, not even the DBA, can create a relation that contains duplicate rows, along with all of the headaches that result therefrom.

The next obvious question is, "In requesting authorization from the DBA, how does the user distinguish between the need for (1) a view that is a key-based **projection** with the key hidden, and (2) a view that is a true **projection** on the non-key columns (one in which duplicate rows are not permitted)?"

Under this scheme, both of these requests are legitimate, and they are quite distinct from one another in meaning. A user who is carrying out statistical analysis and who is not allowed to see primary-key values is likely to want view 1. The main distinction is whether the primary key participates at all in the view. The DBA can already select participation by the primary key in a view. What is new here is that the DBA has the additional option of hiding or not hiding all of the primary-key values.

RA-3 Hiding Selected Columns in Views

A suitably authorized user such as the DBA can not only define what parts of the database a user is authorized to access by means of views, but he or she can also select columns of each view that are to be blocked from that user's access.

Suppose that a user is authorized to access a view V and apply the update operator to column A. An update to the A-component of a row w in a view can make w non-compliant with the definition of the view, causing row w to be removed from the view.

For example, if a row in the view

$$V = R [A < 100]$$

happens to contain 90 as its value of A, and if a user requests that value be incremented by 25, then the DBMS can update the corresponding value of A in R to 115. The effect of this update is that the pertinent row is removed

from the view V. If the DBA prefers that the request be rejected, he or she must place an additional authorization constraint upon the user, namely, that no update is allowed to take a row out of the view. This is one more reason why authorization should not be based on views only.

RA-4 Blocking Updates That Remove Rows From a View

Suppose that a user is authorized to access a view V and to update a column A in V. The DBA has the choice of providing or denying this user the additional authorization to apply those updates to values in column A that take the corresponding row out of the view.

A company can become critically dependent on some database. If a disgruntled or careless employee is authorized to use the DROP RELATION command, he or she could issue numerous commands of this type and cause a complete or near-complete loss of the database. The following two features are aimed at protecting companies and institutions from this serious problem.

RA-5 *N*-person Turn-key

In those DBMS installations at which the continued existence and integrity of the database are critical to the company or institution, the DBMS must support an *N*-person turn-key in order for certain selected activities to be requested by a user successfully ($N > 1$).

A simple use of this feature is to require that both the DBA and his or her manager approve the following:

- any execution of the DROP RELATION command;
- any execution of the DELETE command;
- any change in the delay period cited in Feature RA-6, following.

Feature RA-6 delays the execution of drops and large-scale deletions (possibly all deletions) by archiving the data for a specified number of days or weeks. The DBMS executes these commands in two steps:

1. archive the data immediately;
2. delete the data later.

This delay gives the installation time to react and fully recover from the damage, whether intended or not.

RA-6 Delayed Deletions of Data and Drops By Archiving

Execution of the command DROP RELATION results in the specified relation being archived for a period of at least seven days. Execution of large-scale deletions (possibly all deletions) is delayed by archiving in a similar fashion. Seven days is the default value if no longer period is specified.

Together, Features RA-5 and RA-6 provide the fundamental security that a company needs if it depends heavily on the continued existence and accuracy of its databases. A possible additional use for the *N*-person turnkey is applicable to the type of updates called *exporting updates* in distributed databases. When executed, these updates cause the DBMS to move one or more rows from a relation at one site into a relation at another site. The user or application program at the FROM site would have to be authorized to update beyond the range permitted at that site. The DBA in control of the receiving site might have to authorize reception at that site of the updated information as an insertion (see Section 24.6.2).

A quite different concern in some installations is that the private use of storage for preserving the results of queries is escalating at an alarming rate. Two approaches to limiting this questionable consumption of resources appear useful.

In the first approach, a feature is introduced into RM/V2 that blocks execution of any query for which the result exceeds the quota of storage assigned to a user or group of users, either on a per-query basis or with respect to a specified total.

The second approach requires each result of a query that the user requests the DBMS to store to be transmitted to his or her personal computer or to some storage unit that is specifically assigned to that user. For the time being, no feature of RM/V2 is proposed to handle this requirement.

18.2 ■ Authorizable Actions

RA-7 Authorizable Database-control Activities

There are at least 13 database-control activities that must be separately authorizable and authorizable in combination.

The 13 such database-control activities are as follows:

1. creating and dropping a domain;
2. creating and dropping a base R-table;

3. creating and dropping a column of an existing base R-table;
4. creating and dropping a view;
5. creating and dropping an integrity constraint by type;
6. creating and dropping a user-defined function;
7. creating and dropping a performance-oriented access path (such as an index);
8. creating a foreign key in one R-table referencing a primary key in another R-table (possibly the same R-table);
9. requesting that a specific authorization be granted or discontinued;
10. requesting a snapshot;
11. requesting that an audit log be maintained or discontinued (see Feature RI-15 in Chapter 13);
12. establishing a condition for archiving with a new label;
13. establishing the UP or DOWN mode for "rounding" pseudo-dates (e.g., February 30 or March 32).

RA-8 Authorizable Query and Manipulative Activities

At least seven database query and manipulation activities must be separately authorizable and authorizable in combination. The seven such activities are as follows:

1. retrieving on specific R-tables (base or view);
2. inserting into specific R-tables (base or view);
3. updating specific components of rows in specific R-tables (base or view);
4. updating the primary key of a specific R-table;
5. archiving rows from specific R-tables (base or view);
6. deleting rows from specific R-tables (base or view);
7. updating an I-marked value to either an A-marked value or a database value, and vice versa (see Chapter 8).

The granting of any of these operations, except the second one, may be not only confined to specific R-tables, but may also be value-dependent.

RA-9 Authorizable Qualifiers

Use of all qualifiers must be separately authorizable and authorizable in combination (see Chapter 10). The thirteen qualifiers are as follows:

Feature	Qualifier
RQ-1	A-MAYBE
RQ-2	I-MAYBE
RQ-3	MAYBE
RQ-4	AR(x)
RQ-5	IR(x)
RQ-6	ESR(x)
RQ-7	ORDER BY
RQ-8	ONCE ONLY
RQ-9	DOMAIN CHECK OVERRIDE
RQ-10	EXCLUDE SIBLINGS
RQ-11	DEGREE OF DUPLICATION
RQ-12	SAVE
RQ-13	VALUE

Other activities that should be subject to special authorization are the use of various functions recorded in the catalog, as well as the use of the date-conversion functions (see Item 14, following Feature RT-4 in Chapter 3). Although support within a DBMS product for this special kind of authorization is optional at this time, such a product should be at least designed to accept this extension later.

RA-10 Granting and Revoking Authorization

Authorization to access or modify parts of the database may be assigned to a user or to an already-declared user group and, at a later time, withdrawn from the user or from the group by using statements in the relational language RL. Cycles in which user A makes a grant directly or indirectly to user B, and user B makes a grant directly or indirectly to A, are prohibited.

When two or more users independently grant another user two or more authorizations to access parts of the database (the DBMS certainly supports this), these authorizations may overlap each other in space–time scope either

partially or completely. Later withdrawal of any one of these leaves the others in effect.

Relational DBMS products on the market in the early 1980s often failed to support user groups in their authorization mechanisms. This failure meant that at least one authorization declaration was needed for each individual user, a severe burden on those responsible for this task.

RA-11 Passing on Authority to Grant

Suppose that a user authorizes another user or user group to access part of the database and to execute specified database operations. Suppose also that the grantor is authorized to pass on to other users the granting option. Then, the grantor has the option of granting to or witholding from the recipient permission to make further grants of part or all of this authorization.

This feature is compatible with government-type security, in which a few distinct classes of clearance are set up (e.g., top secret, secret, confidential). Few institutions, however, want to adopt this government-type security, which is relatively rigid and forces the DBA or security officer to establish a class structure on all users.

Therefore, the authorization class of features in RM/V2 has been designed to permit the adoption of many different approaches to database security, ranging from strongly centralized to strongly decentralized.

RA-12 Cascading Revocation

Consider three distinct users A, B, C. If user A grants specific authorization to user B, and if user B passes on part or all of this authorization to user C, revocation of the grant from A to B causes the DBMS to revoke the corresponding grant from B to C. If user U receives identical authorization from two or more sources, then U retains the pertinent authorization until every one of the sources has revoked the authorization.

18.3 ■ Authorization Subject to Date, Time, Resource Consumption, and Terminal

RA-13 Date and Time Conditions

Authorization can be conditioned by day of the month, by day of the week, by a time interval during the day, or by a combination of these.

RA-14 Resource Consumption (Anticipated or Actual)

Authorization can be conditioned by either (1) the system's estimated resource consumption to complete the execution of any request submitted by the user (the system must make this estimate in any case as part of the optimization), or (2) limits on the resource consumption permitted for any request from the user. In the latter case, the request is started unconditionally but is aborted if the specified limits are exceeded.

Of these two options, the first one is preferred, provided the system's estimate of resource consumption is reasonably accurate. Even then, Option 2 is a good additional safety precaution.

RA-15 Choice of Terminal

Authorization can be conditioned by the particular terminal or workstation from which a user is operating.

RA-16 Assigning Authorization

For each user who interacts with a relational database, there must be at least one declaration in the catalog that he or she is authorized to engage in activities (A) within a specified space-time scope (S). Normally very few users would be authorized to pass on to another user part or all of the authorization they possess. This process of passing on authorization is called *granting*.

Occasionally it is necessary nevertheless for someone who does not have authorization with space-time scope (S) and activities (A) to be able to assign that authorization to another user. This action is called *assigning authorization*. Very few users would be authorized to assign authorization.

Thus, when a user assigns authorization to some other user or users, the assignor is granting an authorization whose scope and permitted actions are not within the range of what is owned by the assignor. It is usually the DBA and some of the DBA's staff that need to be able to assign authorization.

It is the DBA staff that is normally responsible for authorizing users to access and, in some cases, modify specified parts of the database. How would the DBA or the DBA staff then cope with a company policy that requires these people NOT to be able to access data in a relation they

created? The answer is that after the DBA or staff create ANY relation, the DBMS does not automatically give the creators the right to be able to access and modify whatever data is entered into that relation. Instead, the DBA can assign user privileges to other users without having those privileges himself or herself.

Exercises

18.1 Is the scope of what a user is allowed to interrogate limited to a DBA-specified list of columns in a single relation? If not, what can the scope be, and how is it specified?

18.2 Should the authorization mechanism in a DBMS be designed so that the power of the relational language in limiting the space-time scope that is authorized for each user is totally and exclusively dependent on applying that power to defining views? Explain your position.

18.3 What are the N-person turn-key and seven-day archiving features? Why should both of these features be supported within a relational DBMS?

18.4 List the seven "create and drop" capabilities that should be separately authorizable and authorizable in combination.

18.5 Why should user groups as well as individual users be supported in regard to authorization?

18.6 Cycles are prohibited in the granting of authorization. Describe the kind of cycles that are prohibited. Explain how cascaded revocation is related to this prohibition.

18.7 What authorization features of RM/V2 enable a DBA to permit a certain amount of ad hoc query to accompany a significant load of production-oriented transactions?

■ CHAPTER 19 ■

Functions

A query expressed as a relational command indicates what types of items are to be retrieved by listing a sequence of pairs of names. Each pair consists of a relation name followed by a column name; this is called the *target list.* A relational command of the query type also indicates the condition to be satisfied by the particular items that are to be extracted; this is called the *condition.* The condition is expressed in four-valued, first-order predicate logic.

Suppose that a database contains information about shipments within a relation named SHIP. Suppose further that for each shipment this information includes the supplier serial number s#, the part serial number p#, and the date of the shipment. An example of a query is as follows: Obtain all of the part serial numbers and dates of shipment for parts shipped after January 31, 1988. A simple relational command for this request is as follows:

get SHIP.p#, SHIP.ship_date where ship_date > 88-1-31.

The target list in this command is SHIP.p#, SHIP.shipdate. The condition is that part of the query that follows the word "where," namely, "shipdate > 88-1-31."

Functions are needed in database management for two purposes. The first purpose is to transform target database values within a query or view definition. The function is then part of the expression for the target list.

Consider as an example the database just cited. Suppose that the primary key of SHIP is the combination of supplier serial number s# and part serial number p#. Suppose that both shipdate and quantity of parts shipped are

immediate properties included as columns in the relation SHIP. Someone must know the number of shipments of part p2 after January 31, 1988. Suppose also that the DBMS supports the COUNT function. Then, an appropriate query is as follows:

get COUNT(SHIP.s#, SHIP.p#) where ship_date > 88-1-31.

Note that the function COUNT occurs in the target list.

The second purpose of including functions in database management is to determine the condition to be satisfied by the target database values in retrieval and in data manipulation. The function is then part of the expression for the condition.

Consider as an example the same database, but a different query: Find the serial numbers of only those parts for which the number of shipments recorded in the database exceeds 10. An appropriate query is as follows:

get SHIP.p# where COUNT (SHIP.s#, SHIP.p#) > 10.

Note that the function COUNT occurs in the condition. Note also that this query could not be expressed as simply in SQL.

19.1 ■ Scalar and Aggregate Functions

The two types of functions discussed in this chapter are scalar functions and aggregate functions. Each type of function can be used in both of the ways just described.

A *scalar function* transforms a scalar into a scalar. An example of such a function is a currency-exchange function, which transforms an amount of money expressed in one currency into a corresponding amount expressed in some other currency.

An *aggregate function* transforms a set of scalars or a set of tuples into a scalar. An example of such a function is the COUNT function just mentioned. Another example is the SUM function, which scans a collection of numbers in a column of some relation (such as amounts that are all expressed in some common currency) and computes their sum.

RF-1 Built-in Aggregate Functions

The DBMS provides at least the five aggregate functions—COUNT, SUM, AVERAGE, MAXIMUM, MINIMUM—as built-in functions for use either in transforming target database values within a query or view-defining command, or in determining the condition to be satisfied by the target database values in retrieval and in data manipulation.

Note that an aggregate function in this context normally transforms many scalar values into a single scalar value. The usual source of the many scalar values is a simple or composite column. When an aggregate function is applied to a column that happens to contain duplicate values, all occurrences of those values participate in the action.

For example, if the SUM function is applied to a currency column C in relation R, the result obtained is the sum of *every value occurrence* in C, which is normally not the same as the sum of every distinct value in C.

Consider the example of a relation EMP that identifies employees and records their immediate properties. The SALARY and BONUS components of each row have as their values the year-to-date salary and commission earned by the employee described in that row:

EMP	(EMP#	ENAME	DEPT#	SALARY	BONUS)
	E10	Rook	D12	12,000	15,800
	E91	Knight	D12	10,000	6,700
	E23	Knight	D05	13,000	13,000
	E57	Pawn	D02	7,000	3,100

The following queries illustrate the built-in functions:

How many employees are there?	get COUNT (EMP)
What is the total bonus earned?	get SUM (EMP.BONUS)
What is the average bonus earned?	get AVERAGE (EMP.BONUS)
What is the maximum salary earned?	get MAXIMUM (EMP.SALARY)
What is the minimum bonus earned?	get MINIMUM (EMP.BONUS)

There is a simple way to obtain the sum or the count of the distinct values in a simple or composite column C, if that is what is needed: take the **projection** (uncorrupted, of course) of the relation R onto column C, and then apply the SUM or COUNT function to the result. This method takes advantage of the fact that true **projection** eliminates duplicate rows from the result.

The following query is based on the same EMP relation: What is the number of distinct salaries? get COUNT (EMP [SALARY])

A good measure of the degree of duplication of values in column C is obtained by taking the count of rows in the relation containing C and dividing that count by the count of distinct values in C. In a relation that provides the immediate properties of employees, it is likely that the degree of dupli-

cation in the gender column (which has only two values, male and female) would be very high, while the degree of duplication in the last-name column would be very low.

The abbreviation DOD stands for DEGREE OF DUPLICATION. Suppose that the values in a simple or composite column C are the intended arguments of a statistical function f. Then, for each row containing the DOD component n, the contribution of the C-component of that row is n times the value of that C-component.

RF-2 The DOD Versions of Built-in Statistical Functions

For each statistical function built into the DBMS (including SUM and AVERAGE as required by Feature RF-1), there is also a DOD version built into the DBMS. (See Feature RQ-11 in Chapter 10 for details concerning the DOD qualifier.)

RF-3 Built-in Scalar Functions

The DBMS supports at least the following arithmetic scalar functions as built-in functions and expressions in RL: addition, subtraction, multiplication, division, and exponentiation. The DBMS also supports at least the following string scalar functions as built-in functions and expressions in RL: concatenation, substring directly specified, and substring by pattern-directed search.

These functions are for use either in transforming target database values within a query or view-defining command, or in determining the condition to be satisfied by the target database values in retrieval and in data manipulation.

Note that a scalar function in this context has a fixed number of arguments (usually one or two, seldom more), that each argument is a scalar, and that the function transforms each of its arguments into a scalar result.

19.2 ■ User-defined Functions

Clearly, the number of functions supplied by the DBMS vendor as part of the DBMS is small, and therefore is not likely to satisfy all DBMS users. There must be a means by which users can add functions suited to their own businesses or institutional activities. Feature RF-4 provides the means. Subsequent features provide additional support or constraints.

RF-4 User-defined Functions: Their Use

Users can define their own functions. The DBMS can then record such functions (scalar, aggregate, or other types) together with their names and types. The system then supports use of such functions in access targeting, in access conditioning, or in both. Use is more generally determined by the orthogonality feature, Feature RL-7 (see Chapter 22). If the function that is required happens to be statistical in nature, users can define a non-DOD version, a DOD version, or both.

Note that the non-DOD version ignores the degree-of-duplication column, if one occurs in the operand relation. On the other hand, the DOD version causes each row that has n as its DOD component to provide a contribution to f that is equal to that from n occurrences of the value in the contributing column.

RF-5 Inverse Function Required, If It Exists

Together with each user-defined function recorded in the catalog, a symbol is entered that indicates whether this function has an inverse. If so, the name of this inverse, together with the code for the inverse, is also recorded in the catalog.

Rarely is it the case that an aggregate function has an inverse.

Some functions are expected to be used in defining some columns of views. If such a function happens to have an inverse, there is a chance that the pertinent column of the view will be updatable, since in such a case the DBMS can compute from any new value in that column of the view the corresponding value (simple or composite) that it should enter into appropriate column(s) of the base R-tables.

RF-6 User-defined Functions: Compiled Form Required

The DBMS requires that each user-defined function and its inverse (if any) be written in one of the host languages, and compiled before the function is stored in the catalog.

Note that, when a user programs a user-defined function and incorporates it into a DBMS, he or she need not know anything about the internal coding or internal structure of the DBMS. The term "host language" is used to identify any one of the so-called general-purpose programming languages such as FORTRAN, COBOL, and PL/1. The relational model supports at least these three as languages with which the principal relational language can communicate.

RF-7 User-defined Functions Can Access the Database

The DBMS is capable of handling user-defined functions that make use of data extracted from the database at the time of execution of these functions.

An example of the need for this feature is found in transactions that involve currency exchange between various currencies. It is then quite likely that the database will contain a relation (say EX) that reflects the exchange rates currently in effect. Four of the columns of EX would be as follows: (1) identification of the FROM currency (primary key); (2) identification of the TO currency (primary key); (3) an amount of the FROM currency expressed in that currency; and (4) the corresponding amount expressed in the TO currency. An exchange function would have to access this relation EX with three arguments:

1. the type of currency from which the exchange is to be made;
2. the type of currency to which the exchange is to be made;
3. the number of units of the FROM currency to be exchanged.

19.3 ■ Safety and Interface Features

RF-8 Non-generation of Marked Values by Functions

The application of a scalar function to unmarked arguments and of an aggregate function to a set of unmarked database values (even if the set is empty) never yields a marked result.

In IBM's DB2, the application of the AVERAGE function to an empty set yields the NULL of SQL—a mistake because this is a case of the result being undefined, and not a case of a value missing from the database. The mistake probably resulted from a confusion about two kinds of facts:

1. The fact that a database value might be missing (represented by the NULL of SQL).
2. The fact that, for some arguments, a function may not have a defined result.

In this example, the average of zero elements is normally taken to be undefined. Whenever an empty set is encountered as an argument to the AVERAGE function, the relational model supports the options of (1) signaling this undefined state, and (2) executing user-defined action if needed (see Feature RQ-6 in Chapter 10).

Note that the outer operators (see Chapter 5) almost always generate marked values, even if the arguments contain no marked values. The addition of a new column to a base R-table also generates marks within that column. Thus, it is not intended that these operators be perceived as functions when interpreting Feature RF-8.

The power and usefulness of a relational database is increased if the following three steps are taken:

1. Columns are permitted to contain names of invokable functions, as well as the usual kinds of value-oriented data.
2. Columns are permitted to contain names of arguments, as well as the usual kinds of value-oriented data.
3. Both the relational language and the host language have the capability of invoking a function for which the name and the values of its arguments can all be obtained, either directly from the database, by use of argument names in the database, or partly from the database and partly from the host-language program and its data.

The following two features make it less likely that the user will feel that he or she is becoming more and more isolated from the outside world as interrogation of the database proceeds.

RF-9 Domains and Columns Containing Names of Functions

One of the domains (extended data types) that is built into the DBMS is that of function names. Such names can be stored in a column (possibly in several columns) of a relation by declaring that the column(s) draw their values from the domain of function names. Both RL and the host programming language support the assemblage of the arguments together with the function name, followed by the invocation of that function to transform the assembled arguments.

RF-10 Domains and Columns Containing Names of Arguments

One of the domains (extended data types) that is built into the DBMS is that of argument names. Such names can be stored in a column (possibly in several columns) of a relation by declaring that the column(s) draw their values from the domain of argument names. These arguments have values that can be retrieved either from the database or from storage associated with a program expressed in the HL.

Features RF-9 and RF-10 are very likely to be useful in making extensions to the relational model that involve user-defined functions. Therefore, more is said about them in Chapter 28.

Exercises

19.1 In the relational model, each value in a column of a relation is required to be atomic with respect to the DBMS. Under what circumstances, if any, are such values not atomic? Give examples that illustrate the use of database values in a non-atomic role.

19.2 What are scalar and aggregate functions? List the five aggregate functions that should be built into the DBMS.

19.3 For each user-defined function, which of the following is required to be stored in the catalog?

- The source code.
- The compiled code.
- Both the source and compiled code.

19.4 Identify those items that are also stored in the catalog for each user-defined function, and that facilitate the updating of certain kinds of views. Explain the items you selected and why you chose them.

19.5 In certain kinds of manufacturing, the production of most products involves the use of several machines, each of which has an associated minimum interval of use and a distinct cost-of-use function. Describe a feature of RM/V2 that is essential if the computation of total cost is to be incorporated in the retrieval of data from the database.

■ *CHAPTER 20* ■

Protection of Investment

When a company or institution acquires a database management system, the immediate investment consists of the cost of the hardware and software. However, there is a longer-term investment, that may be even larger than the initial one: the investment in the development of application programs and in the training of users, both programmers and end users. The total investment is quite heavy, and the purchaser of a DBMS needs some assurance that the risk is slight.

This chapter deals with features of the relational model designed to protect the user's total investment. In particular, these features enable application programs to continue to run correctly when a variety of changes are made in the database, including changes in the physical representation of data, the logical representation, the integrity constraints, and the distribution of data between a given collection of sites.

20.1 ■ Physical Protection

RP-1 Physical Data Independence

The DBMS permits a suitably authorized user to make changes in storage representation, in access method, or in both—for example, for performance reasons. Application programs and terminal activities remain logically unimpaired whenever any such changes are made. (This feature is Rule 8 in the 1985 set.)

To handle physical data independence, the DBMS must support a clear, sharp boundary between the logical and semantic aspects, on the one hand, and the physical and performance aspects of the base R-tables, on the other; application programs and terminal users must deal with the logical and semantic aspects only. Relational DBMS can and should support this feature totally. Non-relational DBMS rarely provide complete support for this feature (in fact, I know of none that do).

20.2 ■ Logical Protection

RP-2 Logical Data Independence

Application programs and terminal activities remain logically unimpaired when information-preserving changes are made to the base R-tables, provided these changes are of the kind that permit unimpairment, either according to Algorithm VU-1 or according to a strictly stronger algorithm. (This feature is Rule 9 in the 1985 set.)

Three examples of information-preserving changes are listed below.

1. Partitioning an R-table into two or more tables by rows using row content.
2. Splitting an R-table into two tables by columns using column names, provided the original primary key is preserved in each result.
3. Combining two R-tables into one by means of a non-loss **join.** (Note that many authors now call non-loss **joins** "lossless.")

To provide this service whenever possible, the DBMS must be capable of handling insertions, updates, and deletions on all views that are updatable in accordance with Algorithm VU-1 (see Feature RV-5 in Chapter 16, as well as Chapter 17). Features RP-1 and RP-2 permit logical database design to be tackled with a high degree of independence from physical database design. Feature RP-2 also permits the logical design to be dynamically changed if necessary for any reason, without damaging the user's investment in application programs.

The physical and logical data-independence features permit database designers for relational DBMS to make mistakes in their designs without the heavy penalties levied by non-relational DBMS. In turn, this means that it is much easier to get started with a relational DBMS, because not nearly as much performance-oriented planning is needed before putting the database into operation.

20.3 ■ Integrity Protection

RP-3 Integrity Independence

Integrity constraints specific to a particular relational database must be definable in the relational data sublanguage RL, and storable in the catalog. Application programs and terminal activities remain logically unimpaired when changes are made in these integrity constraints, provided such changes theoretically permit unimpairment (where "theoretically" means at a level of abstraction for which all DBMS implementation details are set aside). (This feature is Rule 10 in the 1985 set.)

Safe control of database integrity cannot be guaranteed if these constraints are included in the application programs only.

In addition to the two general integrity constraints—entity integrity and referential integrity—which apply to each and every relational database, there is a clear need to be able to specify additional integrity constraints of the domain type, column type, and the user-defined type. Such constraints usually reflect business policies, and/or government regulations, and/or the principal types of well-understood semantic dependencies (functional, multivalued, join, and inclusion).

If, as sometimes happens, either business policies or government regulations change, it is probably necessary to change the user-defined integrity constraints. Normally, this can be accomplished in a fully relational DBMS by changing one or more integrity statements stored in the catalog. The DBMS is designed not to require any changes in the application programs or in the terminal activities, unless such changes are unavoidable. Nonrelational DBMS rarely support this feature as part of the DBMS (where it belongs). Instead, they depend on a dictionary package or application generator (which may or may not be present, and can readily be bypassed).

20.4 ■ Re-distribution Protection

RP-4 Distribution Independence

A relational DBMS has a data sublanguage RL, which enables application programs and terminal activities to remain logically unimpaired under two circumstances:

- When data distribution is first introduced (this may occur because the DBMS originally installed manages non-distributed data only);
- When data is redistributed (if the DBMS manages distributed data)—the redistribution may involve an entirely different decomposition of the totality of data.

(This feature is Rule 11 in the 1985 set.)

The DBMS property defined in Feature RP-4 is called *distribution independence.* This definition is carefully worded so that both a distributed and a non-distributed DBMS *can* fully support Feature RP-4. Whether a DBMS product provides such support is primarily resolved by examining the data sublanguage(s) that the product supports. Is at least one of these languages at a sufficiently high level to support both of the situations just stated, and has this been demonstrated (e.g., by a prototype)? I believe that the relational level is essential. Certainly, no non-relational approach has been published that has proved its success in supporting Feature RP-4.

Some examples of DBMS products in the marketplace that fully support Feature RP-4 are SQL/DS and DB2 of IBM, INGRES of Relational Technology, and NonStop SQL of Tandem (in their current releases). Other vendors are rapidly entering the distributed database management market. Except for distribution independence and decomposition and recomposition (see Feature RP-5), the features that appear necessary for a DBMS to excel in distributed-database management are discussed separately (see Chapters 24 and 25).

Support of Feature RP-4 by the IBM systems has been demonstrated as follows: SQL programs written to operate on non-distributed data, using System R, run correctly on distributed versions of that data using System R*, the IBM San Jose Research prototype of a distributed database management system [Williams et al. 1981]. The distributed INGRES project has shown a similar capability for the QUEL language of INGRES [Stonebraker 1986].

Distribution independence is a more serious requirement than mere location independence. The former concept permits not only all the data at any one site to be moved to another, but also a completely different decomposition of the totality of data at all sites into fragments to be deployed at the various sites.

It is important to distinguish among (1) distributed processing, (2) networking, and (3) distributed data. In the first case, application programs are transmitted to the data. In the second case, messages can be sent from a processing unit at any site to a processing unit at another. In the third case, data is derived from possibly multiple sites (the derivation being executed at whatever sites the optimizer selects for efficiency) and directed

to the requesting program or terminal. Many non-relational DBMS support distributed processing, but not the management of distributed databases. Support for distributed database management—in which all the data, whether stored locally or remotely, appears to be local—has been demonstrated by relational DBMS prototypes. Whether it can be supported by any other kind of DBMS remains to be seen.

In the case of distributed relational DBMS, a single transaction may straddle several remote sites. Such straddling is managed entirely under the covers—the system may have to execute recovery at multiple sites. Each program or terminal activity treats the totality of data as if it were all local to the site where the application program or terminal activity is being executed.

A fully relational DBMS that does *not* support distributed databases has the capability of being extended to provide that support, while leaving application programs and terminal activities logically unimpaired, both at the time of initial distribution and whenever later re-distribution is made. There are four important reasons why relational DBMS enjoy this advantage.

1. *Decomposition flexibility* in deciding how to deploy the data.
2. *Recomposition power* of the relational operators when combining the results of sub-transactions executed at different sites.
3. *Economy of transmission* resulting from the fact that the DBMS has multiple-record-at-a-time capability. Thus, there need not be a request message sent for each record to be retrieved from any remote site, or a reply message for each result record to be transmitted back.
4. *Analyzability of intent* (due to the very high level of relational languages) for vastly improved automatic optimization of execution—and, when necessary, automatic re-optimization.

RP-5 Distributed Database Management: Decomposition and Recomposition

If the DBMS supports distributed database management, it uses the full power of RL (including four-valued, first-order predicate logic) to decompose each RL statement into simpler RL statements, each of which is capable of being executed at a single site. Such a DBMS also uses this full power to recombine the results from the subrequests to yield a coherent and correct response to the whole request.

Note that Feature RP-5 is not applicable if the DBMS is not claimed to support distributed database management.

20.5 ■ Summary of the Practical Reasons for Features RP-1–RP-5

Features RP-1–RP-4 represent *four different types of independence* aimed at protecting users' investment in application programs, terminal activities, and training. Features RP-1 and RP-2—physical and logical data independence—have been widely discussed for many years. Nevertheless, today there still exist DBMS products that fail to support these features. Features RP-2, RP-3, and RP-4—logical data independence, integrity independence, and distribution independence—have received inadequate attention to date, but are likely to become as important as Feature RP-1. There is no claim that these four types of independence are easily implemented in a DBMS. Feature RP-5 may help the reader understand (1) the complexity of the problem, and (2) the fact that the relational model has the capability of solving it.

Exercises

20.1 What are the four main types of investment protection that are obtainable from a fully relational DBMS?

20.2 If at its inception a database is properly designed logically and physically, why should it ever be necessary to change that design (1) logically and/or (2) physically?

20.3 Why should it ever be necessary to change integrity constraints?

20.4 If integrity constraints are changed, why be concerned about having to change a few application programs, especially if the programs are written in a "fourth generation language"?

20.5 Someone asserts that determining how data should be distributed to various sites is a design problem that occurs just once (at installation time), and concludes that distribution independence is a feature of no value. Is there any merit in this argument?

■ *CHAPTER 21* ■

Principles of DBMS Design

The relational model is based on the fundamental laws discussed in Chapter 29. It is intended that implementations of the model are to be based on the design principles described in this chapter. The main motivation for formulating these principles in explicit terms was the numerous blunders that have been made in various DBMS implementations.

In the next chapter, design principles for relational languages are discussed. I recommend to DBMS vendors that they use the principles in both chapters to get the designers of their DBMS products back on track.

RD-1 Non-violation of any Fundamental Law of Mathematics

> The DBMS and its relational language(s) do not violate any of the fundamental laws of mathematics.

At first glance, this appears to be a completely unnecessary feature. In examining a database management system in 1969, however, I discovered that, under certain conditions, it failed to support the commutativity of logical AND. More specifically, suppose that X and Y are correctly formulated truth-valued expressions, and that Q1, Q2 are two identical queries except that Q1 has the condition X AND Y, while Q2 has the condition Y AND X. Under the special conditions, Q1 and Q2 failed to yield the same

result on identical databases. This fact was not disclosed in the manuals supplied to users. Fortunately, this DBMS product failed in the marketplace.

The reader may imagine that such a blunder could have happened only in the 1960s. Unfortunately, a recent release of a well-known relational DBMS product, marketed by a vendor with an excellent reputation, fails under certain conditions to yield x from the expression x + y − y when x happens to be a date and y happens to be a date interval, even though x and y are correctly formulated. The error in the result can be as much as three days.

In my opinion, both examples illustrate the appalling lack of real concern for the quality of their products by the workers and managers in software, and an astonishing lack of concern for the very large number of institutions that are likely to be adversely affected by such sloppy work. The examples also illustrate an apparent lack of understanding of basic user requirements and basic mathematics.

RD-2 Under-the-covers Representation and Access

The DBMS may employ any storage representations and any access methods for data, provided these are implemented "under the covers"—that is, they must not be exposed to users (with the possible exception of giving the DBA a few types of commands to create and drop performance-oriented structures and access methods). Once these structures are created, the responsibility for maintaining them during insertion, update, and deletion activities belongs to the DBMS, not to users and not to the DBA.

RD-3 Sharp Boundary

The DBMS makes a sharp separation between two aspects: (1) performance-oriented features (such as indexes), and (2) semantic and logical features (such as the uniqueness of values in a column or combination of columns, the exclusion of missing values, the primary-key property, and the foreign-key property).

In general, most users should be protected from having to deal with the first item altogether. In particular, if the DBA drops an index (for example), there should be no loss of semantic features, such as those cited in the second item. Some existing DBMS products fail in this respect because they require an index to exist on any column whose values must all be distinct from one another. Distinctness of values is a semantic feature, while an index is a performance-oriented feature.

Should application programmers or end users be burdened by concern for the logical aspects of concurrency control? This question deserves special consideration.

Two types of concurrency must be supported by a relational DBMS:

1. *intra-command concurrency* consists of treating various portions of a single relational command as independent tasks, and executing these tasks concurrently;
2. *inter-command concurrency* consists of executing two or more relational commands concurrently.

RD-4 Concurrency Independence

Application programs and activities by end users at terminals must be logically independent of whether the DBMS supports intra-command concurrency, inter-command concurrency, neither, or both. Application programs and activities must also be independent of the controls (usually locking) that protect any one action A from interfering with or damaging any other action that happens to be concurrent with A.

A relational DBMS never requires the user or application program to make an explicit request for some kind of lock. Such an action would be oriented too heavily to a particular implementation. An interrogative or manipulative request, however, may represent an implicit request for some kind of lock.

Some requests cause the DBMS to impose long-term locking. An example is the updating of several primary-key values, each of which triggers corresponding updating of foreign-key values scattered in various parts of the database.

Occasionally, a terminal user makes a request that requires the DBMS to lock a large quantity of data. Then the user may leave the terminal, absent-mindedly signing off before completing whatever action would release the locks. The DBMS must protect other users and programs from unauthorized long-term locking.

RD-5 Protection Against Unauthorized Long-term Locking

The DBA can specify a time block T permitted on every locking action caused by any request from a terminal user or application program. For each user and program, the DBA can also specify a locking quota expressed in multiples of T. Whenever one block is

consumed, the DBMS checks the authorization data to see whether the quota, for which the user or application program is authorized, has been consumed. If the quota has been consumed, the DBMS aborts the request. If not, the DBMS reduces the quota by one, and proceeds with the transaction.

Whenever a terminal user is involved, each time a time block is consumed, the DBMS should check that the user is still at his or her terminal. This can easily be done by requesting the user to press a certain key if he or she wants the action to continue and wants the locks held for one more time block.

RD-6 Orthogonality in DBMS Design

Any coupling of one feature with another in the design of a DBMS must be justified by some clearly stated, unemotional, logically defensible reason.

Feature RD-6 should not be confused with Feature RL-7 (see Chapter 22). Two examples of unjustified coupling of features in a very prominent relational DBMS may clarify the intent behind Feature RD-6. Both examples involve indexes, and it should be understood that, from the standpoint of the relational model, an index is purely a performance-oriented concept, one that should be kept hidden from all users except possibly the few who are authorized to create and drop indexes.

In the first example, the relational DBMS requires that any column in which the values are constrained to be distinct from one another (a frequently encountered semantic constraint) must be indexed. Unfortunately, this means that such an index cannot be dropped purely for performance reasons.

In the second example, the relational DBMS maintains statistics for use by the optimizer to deliver improved performance. This is a good performance-oriented feature that should be independent of whatever columns happen to be indexed at any time. The statistics consist chiefly of the number of distinct values in each column. The relational DBMS, however, maintains these statistics only for those columns that are indexed—the simplest task for the implementors. As a result, it is easy to construct examples of SQL commands for which the relational DBMS will give unnecessarily poor performance. Perhaps the designers failed to realize that statistics normally do not change rapidly, and therefore need not be (and should not be) updated every time any part of the database content is changed.

In each of these examples, there is an unjustified coupling of two independent features—in the first case, a semantic feature coupled with a performance-oriented feature; in the second case, one performance-oriented feature with another quite distinct and independent one.

RD-7 Domain-based Index

For those DBMS that are based primarily on software, the DBMS supports the creation and dropping of domain-based indexes by suitably authorized users. This may also provide advantages in performance for hardware-based DBMS.

A domain-based index is a single index on the combination of columns defined on the specified domain (normally all of those columns are involved, but a suitably authorized user may select only those columns that will prove advantageous). Such an index will usually be a multi-table index. It facilitates the efficient execution of referential integrity (if the domain is a primary domain) and other occurrences of inclusion dependence. So far, the only relational DBMS I have encountered with a domain-based index is one developed at the University of Nice.

RD-8 Database Statistics

In the catalog, the DBMS stores statistics of the data (see Feature RC-11 in Chapter 15). This information is used by the optimizer to select the most efficient method of handling each retrieval and manipulative command. The DBMS updates these statistics only occasionally—certainly not upon every insert, delete, or update. The updating of statistics is executed by the DBMS with a frequency and at times requested by the DBA or any suitably authorized user.

The DBMS fails to provide full support for this feature if it either (1) supports no database statistics at all or (2) supports statistics for only those columns that happen to be indexed or happen to have some other performance-oriented property.

RD-9 Interrogation of Statistics

The DBMS statistics cited in RD-3 may be interrogated by use of RL by the DBA or by any suitably authorized user.

RD-10 Changing Storage Representation and Access Options

Commands must be available to the DBA for dynamically changing the storage representation and access method in use for any base

relation *without causing logical impairment of any transaction in source code form* (whether already compiled or not), or any noticeable delays in the execution of the transactions in progress or of transactions waiting to be processed.

If transactions are normally compiled before their first execution, and if the change in storage representation or access method necessitates re-compilation, then this feature requires that the re-compilation must be automatically called into action by the DBMS without manual intervention by the DBA or by any user.

RD-11 Automatic Protection in Case of Malfunction

In case of a malfunction that causes one or more transactions to fail to complete, the DBMS must protect the database from the effects of the failed transactions.

RD-12 Automatic Recovery in Case of Malfunction

In case of a malfunction that causes one or more transactions to fail to complete, the DBMS must be able—without user intervention—to recover immediately after the cause of the malfunction has been repaired. Recovery can be deemed effective when the aborted and delayed transactions have been successfully re-executed using the state of the database effective at re-execution time.

The DBMS maintains a recovery log for this purpose. This feature is included because it is considered an essential requirement of any DBMS, whether relational or not.

RD-13 Atomic Execution of Relational Commands

Each relational command is executed in its entirety without breaks or stoppages because of restrictions on the size of operands or the size of results imposed by the DBMS implementation, or any other reason except malfunction.

A few DBMS products discontinue the processing of an RL command after processing a fixed number of rows (50,000 in one case) from any one R-table; the number is a DBMS implementation constraint. This could be an unpleasant surprise for the many users who must process much larger tables, involving in some cases millions of rows.

Sometimes programmers using a relational DBMS must apply an update across a relation that has a million rows. It is good practice to avoid tackling such an update on a global, single-commit basis, because of the possibility of a massive rollback if anything goes wrong in the later stages. The other extreme consists of processing the updates on a row-by-row basis (single-record-at-a-time); this is likely to detract from performance. An efficient solution involves tackling the update in batches of several thousand rows each at a time. Every time such a batch is successfully completed, the corresponding updates are committed to the database. Support for this progressive, batch-by-batch activity is needed in relational languages, and can be expected in RM/V3.

RD-14 Automatic Archiving

The DBMS supports the automatic archiving of data when it reaches an age specified by the DBA. The frequency of archiving is also specified by the DBA (e.g., once every quarter of a year). It must be possible to re-activate any relational snapshot that has been archived.

RD-15 Avoiding Cartesian Product

During the execution of any single RL command, the DBMS avoids generating the **Cartesian product** of two R-tables as an intermediate result, and never generates the **Cartesian product** as the final result of an RL command, except possibly in the case of a **join** being requested without any **join** condition. In this case, the DBMS issues a warning message.

The **Cartesian product** is wasteful in terms of memory space, channel time, and processing-unit cycles. Thus, if it is requested as a final result, the user should be aware that it is expensive and contains no more information than its factors.

RD-16 Responsibility for Encryption and Decryption

It is the sole responsibility of the DBMS to invoke programs for encrypting data immediately before storing it in the database, and for decrypting data upon retrieval.

The DBA must enforce this feature until an acceptable way is found for the DBMS to enforce it. Clearly, if programmers are allowed to include encrypting and decrypting functions in their own programs, the DBMS and the DBA have abandoned the maintenance of integrity in at least the encrypted data, and possibly in closely related data also.

The design principles introduced in this chapter are aimed at cleanliness of design and ease of use by the whole community of users. They are *not* aimed at reducing either the creative originality of individual implementors or the degree of competition among their respective companies.

- RD-1 Non-violation of any fundamental law of mathematics
- RD-2 Under-the-covers representation and access
- RD-3 Sharp boundary between performance-oriented features
- RD-4 Concurrency independence
- RD-5 Protection against unauthorized locking
- RD-6 Orthogonality in DBMS design
- RD-7 Domain-based index
- RD-8 Database statistics
- RD-9 Interrogation of statistics
- RD-10 Changing storage representation and access options
- RD-11 Automatic protection in case of malfunction
- RD-12 Automatic recovery in case of malfunction
- RD-13 Atomic execution of relational commands
- RD-14 Automatic archiving
- RD-15 Avoiding cartesian product
- RD-16 DBMS responsible for all encryption and decryption of data

Exercises

21.1 A designer who helped design a DBMS product says, "Who cares about any violations of mathematics? Mathematics is a subject strictly for mathematicians, whereas our DBMS is concerned with the real world." Decide whether this is a defensible position or an untenable position. Explain.

21.2 Is it required that the relations of the relational model be represented as tables in storage? If not, what are the only constraints on this representation?

21.3 A DBMS designer says, "In the old days, we never bothered about separating storage representation and access methods, on the one hand, from the logical representation, on the other hand. I fail to see why such sharp separation is necessary." Try to provide this designer with some insight regarding the need for this separation.

21.4 Supply the names and definitions of the two types of concurrency in database management. Which of these would be more advantageous if your workload involved:
1. many complex requests?
2. many simple requests?

21.5 What is a domain-based index, and what is it good for?

21.6 Can skilled application programmers develop code that runs more efficiently on a pre-relational single-record-at-a-time DBMS than on a relational DBMS that has a well-designed optimizer? Give reasons for your position. (See Chapter 26.)

21.7 Will the code described in Exercise 21.6 continue to run more efficiently (if it does) in spite of changes in the business and hence in the traffic on the database? How readily can this code be adapted to the new traffic? Explain.

21.8 A fully relational DBMS provides the following:
- protection from hardware malfunction;
- continuation of those processes that have not been damaged by the hardware malfunction;
- recovery without loss of information or commands after the hardware malfunction has been repaired.

State which of these services are supported by RM/V2.

21.9 Why should **Cartesian product** be avoided in:
1. designing a relational DBMS product?
2. in using a relational DBMS that happens to support this operator?