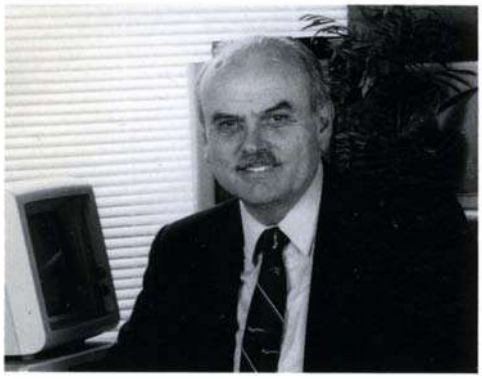
E. F. CODD

# The RELATIONAL MODEL for DATABASE MANAGEMENT

VERSION 2

# The RELATIONAL MODEL for DATABASE MANAGEMENT: VERSION 2

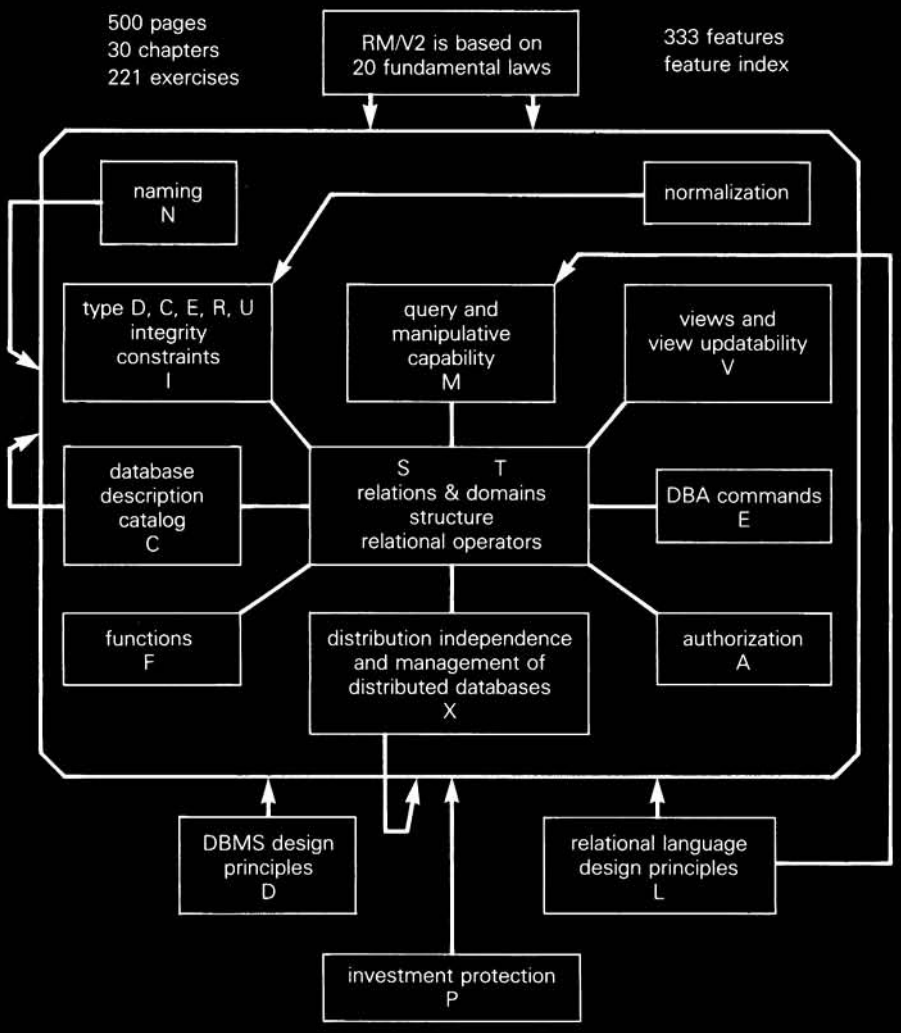


**About the Author**

**Dr. Edgar F. Codd** joined IBM in 1949 as a programming mathematician for the Selective Sequence Electronic Calculator. During the 1950s he participated in developing several important IBM products. Beginning in 1968, Dr. Codd turned his attention to the management of large commercial databases and developed the relational model as a foundation. Since the mid-1970s, Dr. Codd has been working persistently to encourage vendors to develop relational DBMSs and to educate users, DBMS vendors, and standards committees regarding the services such a DBMS should supply and why users need all these services.

In 1985, Dr. Codd established two lecturing and consulting companies in San Jose. These companies specialize in all aspects of relational database management, relational database design, and evaluation of products that are claimed to be relational.

Continued on back flap




333 features / 18 classes // average # of features per class = 18

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| classes | A | B | C | D | E | F | 6 | 18 classes of features |
| | I | J | | L | M | N | 5 | |
| | P | Q | | | S | T | 4 | |
| | V | | X | | | Z | 3 | |

*E. F. CODD*

# The Relational Model

# for Database Management

▪ *Version 2* ▪

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

**Library of Congress Cataloging-in-Publication Data**

Codd, E. F.
The relational model for database management : version 2 / E.F. Codd.
p. cm.
Includes index.
ISBN 0-201-14192-2
1. Data base management. 2. Relational data bases. I. Title.
QA76.9.D3C626 1990
005.75'6—dc20 89-6793
CIP



ABCDEFGHIJ-HA-943210

To fellow pilots and aircrew
in the Royal Air Force
during World War II

and the dons at Oxford.

These people were the source of my determination to fight for what I believed was right during the ten or more years in which government, industry, and commerce were strongly opposed to the relational approach to database management.

# PREFACE

Today, if you have a well-designed database management system, you have the keys to the kingdom of data processing and decision support. That is why there now exists a prototype machine whose complete design is based on the relational model. Its arithmetic hardware is a quite minor part of the architecture. In fact, the old term "computer system" now seems like a misnomer.

My first paper dealing with the application of relations (in the mathematical sense) to database management was a non-confidential IBM research report made available to the general public that was entitled *Derivability, Redundancy, and Consistency of Relations stored in Large Data Banks* [Codd 1969]. I placed a great deal of emphasis then on the preservation of integrity in a commercial database, and I do so now. In this book, I devote Chapters 13 and 14 exclusively to that subject.

Another concern of mine has been, and continues to be, precision. A database management system (DBMS) is a reasonably complex system, even if unnecessary complexity is completely avoided. The relational model intentionally does not specify how a DBMS should be built, but it does specify what should be built, and for that it provides a precise specification.

An important adjunct to precision is a sound theoretical foundation. The relational model is solidly based on two parts of mathematics: first-order predicate logic and the theory of relations. This book, however, does not dwell on the theoretical foundations, but rather on all the features of the relational model that I now perceive as important for database users, and therefore for DBMS vendors. My perceptions result from 20 years of practical experience in computing and data processing (chiefly, but not exclusively, with large-scale customers of IBM), followed by another 20 years of research.

I believe that this is the first book to deal exclusively with the relational approach. It does, however, include design principles in Chapters 21 and 22. It is also the first book on the relational model by the originator of that model. All the ideas in the relational model described in this book are mine, except in cases where I explicitly credit someone else.

In developing the relational model, I have tried to follow Einstein's advice, "Make it as simple as possible, but no simpler." I believe that in the last clause he was discouraging the pursuit of simplicity to the extent of distorting reality. So why does the book contain 30 chapters and two appendixes? To answer this question, it is necessary to look at the history of research and development of the relational model.

From 1968 through 1988, I published more than 30 technical papers on the relational model [Codd 1968–Codd 1988d]. I refer to the total content of the pre-1979 papers as Version 1 of the relational model (RM/V1 for brevity).

Early in 1979, I presented a paper to the Australian Computer Society at Hobart, Tasmania, entitled "Extending the Database Relational Model to Capture More Meaning," naming the extended version RM/T (T for Tasmania). My paper on RM/T later appeared in *ACM Transactions on Database Systems* [Codd 1979]. My aim was for the extensions to be tried out first in the logical design of databases and subsequently to be incorporated in the design of DBMS products, but only if they proved effective in database design.

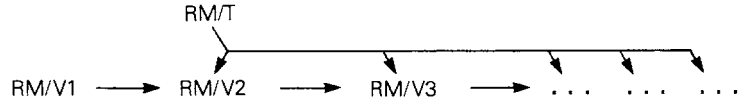
Progress in this direction has been much slower than I expected. Vendors of DBMS products have in many cases failed to understand the first version RM/V1, let alone RM/T. One of the reasons they offer is that they cannot collect all the technical papers because they are dispersed in so many different journals and other publications.

This book collects in one document much of what has appeared in my technical papers, but with numerous new features, plus more detailed explanation (and some emphasis) on those features of RM/V1 and RM/V2 that capture some aspects of the meaning of the data. This emphasis is intended to counter the numerous allegations that the relational model is devoid of semantics. I also hope that this document will challenge vendors to get the job done.

Figure P.1 is intended to show how features of RM/T are expected to be gradually dropped into the sequence of versions RM/V2, RM/V3, . . . . The dropping will be gradual to allow DBMS vendors and consumers time to understand them.

RM/V2 consists of 333 features. A few of these features are of a proscriptive nature, which may sound surprising or inappropriate. However,

***Figure P.1*** **Relationship between the Various Versions of the Relational Model**



they are intended to improve the understanding of the relational model and help DBMS vendors avoid extensions that at first glance seem quite harmless, but later turn out to block extensions needed to advance the DBMS from a primitive to a basic status. The most famous example of a proscriptive feature in the computing field was Dijkstra's assertion that new programming languages should exclude the GO TO command.

The features of RM/V2 include all of the features of RM/V1, roughly 50 of them. Thus, this book covers both versions of the relational model. However, except for some of the advanced operators in Chapter 5, there is no sharp boundary between RM/V1 and RM/V2. This is partly due to changes in some of the definitions to make them more general—for example, entity integrity and referential integrity. Incidentally, the new definitions [Codd 1988a] were available to DBMS vendors well before their first attempts to implement referential integrity.

Domains, primary keys, and foreign keys are based on the meaning of the data. These features are quite inexpensive to implement properly, do not adversely affect performance, and are extremely important for users. However, most DBMS vendors have failed to support them, and many lecturers and consultants in relational database management have failed to see their importance.

Most of the new ideas in RM/V2 have been published in scattered technical journals during the 1980s. What is different about this version of the relational model? Is all of RM/V1 retained?

Versions 1 and 2 are at the same high level of abstraction, a level that yields several advantages:

- independence of hardware support;
- independence of software support;
- occasionally, vendors can improve their implementations "under the covers" without damaging their customers' investment in application programs, training of programmers, and training of end users.

A strong attempt has been made to incorporate all of RM/V1 into RM/V2, allowing programs developed to run on RM/V1 to continue to operate correctly on RM/V2. The most important additional features in RM/V2 are as follows:

- a new treatment of items of data missing because they represent properties that happen to be inapplicable to certain object instances—for example, the name of the spouse of an employee when that employee happens to be unmarried (Chapters 8 and 9);
- new features supporting all kinds of integrity constraints, especially the user-defined type (Chapter 14);
- a more detailed account of view updatability, which is very important for users but has been sadly neglected by DBMS vendors (Chapter 17);

- some relatively new principles of design applied to DBMS products and relational languages (Chapters 21 and 22);
- a more detailed account of what should be in the catalog (Chapter 15);
- new features pertaining to the management of distributed databases (Chapters 24 and 25);
- some of the fundamental laws on which the relational model is based (Chapter 29).

A few of the ideas in RM/T have been incorporated into RM/V2. Many, however, are being postponed to RM/V3 or later versions, because the industry has not been able to maintain an adequate pace of product development and improvement. Additionally, errors made in the design of DBMS products along the way are also hindering progress—often it is necessary to continue to support those errors in order to protect a customer's heavy investment in application programs.

In this book, I attempt to emphasize the numerous semantic features in the relational model. Many of these features were conceived when the model was first created. The semantic features include the following:

- domains, primary keys, and foreign keys;
- duplicate values are permitted within columns of a relation, *but duplicate rows are prohibited*;
- systematic handling of missing information independent of the type of datum that is missing.

These features and others go far beyond the capabilities of pre-relational DBMS products.

Except in Chapter 30, very little is said about models for database management other than the relational model. The relational model, invented in 1969, was the first model for database management. Since then, it has become popular to talk of many other kinds of data models, including a network data model, a hierarchical data model, a tabular data model, an entity-relationship model, a binary relationship model, and various semantic data models.

Historically, it has often been assumed that the hierarchic and network data models pre-dated not only the relational model, but also the availability of hierarchical and network DBMS products. Actually, judging by what has been published, no such models existed before the relational data model was invented or before non-relational DBMS products became available. With the sole exception of relational systems, database management system products existed before any data model was created for them.

The motivations for Version 2 of the relational model included the following five:

1. all of the motivations for Version 1;
2. the errors in implementing RM/V1, such as the following:
   a. duplicate rows permitted by the language SQL;
   b. primary keys have either been omitted altogether, or they have been made optional on base relations;
   c. major omissions, especially of all features supporting the meaning of the data (including domains);
   d. indexes misused to support semantic aspects;
   e. omission of almost all the features concerned with preserving the integrity of the database.
3. the need to assemble all of the relational model in one document for DBMS vendors, users, and inventors of new data models who seem to be unaware of the scope of the relational model and the scope of database management;
4. the need for extensions to Version 1, such as the new kinds of joins, user-defined integrity, view updatability, and features that support the management of distributed databases;
5. the need for users to realize what they are missing in present relational DBMS products because only partial support of the relational model is built into these products.

In Appendix A, the features index, there is a specialized and comprehensive index to all of the RM/V2 features. This index should facilitate the cross-referencing that occurs in the description of several features. In addition to the exercises at the end of each chapter, simple exercises in predicate logic and the theory of relations appear in Appendix B. The reference section, in addition to full citations to the many papers and books cited in the text, includes a short bibliographical essay.

I have tried to keep the examples small in scale to facilitate understanding. However, small-scale examples often do not show many of the effects of the large scale of databases normally encountered.

Finally, I would like to acknowledge the encouragement and strong support provided by friends and colleagues, especially Sharon Weinberg, to whom I am deeply indebted. I also wish to thank the reviewers of my manuscript for their many helpful comments: Nagraj Alur, Nathan Goodman, Michel Melkanoff, Roberta Rousseau, Sharon Weinberg, and Gabrielle Wiorkowski.

I hope that all readers of this book—whether they are students, vendors, consultants, or users—find something of value herein.

E. F. Codd

*Menlo Park, California*

# ■ CONTENTS ■

*CHAPTER 23*

## Serious Flaws in SQL 371



*CHAPTER 24*

## Distributed Database Management 391

*CHAPTER 30*

## Claimed Alternatives to the Relational Model 467



*APPENDIX A*

## RM/V2 Feature Index 483

■ *CHAPTER 1* ■

# Introduction to Version 2 of the Relational Model

## 1.1 ■ What Is a Relation?

The word "relation" is used in English and other natural languages without concern for precise communication. Even in dictionaries that attempt to be precise, the definitions are quite loose, uneconomical, and ambiguous. *The Oxford English Dictionary* devotes a whole page of small print to the word "relation." A small part of the description is as follows:

> That feature or attribute of things which is involved in considering them in comparison or contrast with each other; the particular way in which one thing is thought of in connexion with another; any connexion, correspondence, or association, which can be conceived as naturally existing between things.

On the other hand, mathematicians are concerned with precise communication, a very high level of abstraction, and the economy of effort that stems from making definitions and theorems as general as possible. A special concern is that of avoiding the need for special treatment of special cases except when absolutely necessary. The generally accepted definition of a relation in mathematics is as follows:

> Given sets S1, S2, . . ., S*n* (not necessarily distinct), R is a relation on these *n* sets if it is a set of *n*-tuples, the first component of which is drawn from S1, the second component from S2, and so on.

More concisely, R is a subset of the Cartesian product S1 × S2 × . . . × S*n*. (For more information, see Chapter 4.) Relation R is said to be of *degree n*. Each of the sets S1, S2, . . ., S*n* on which one or more relations are defined is called a *domain*.

It is important to note that a mathematical relation is a set with special properties. First, all of its elements are tuples, all of the same type. Second, it is an unordered set. This is just what is needed for commercial databases, since many of the relations in such databases are each likely to have thousands of tuples, sometimes millions. In several recently developed databases, there are two thousand millions of tuples. In such circumstances, users should not be burdened with either the numbering or the ordering of tuples.

The relational model deals with tuples by their information content, not by means extraneous to the tuples such as tuple numbers, tuple identifiers, or storage addresses. The model also avoids burdening users with having to remember which tuples are next to which, in any sense of "nextness."

As one consequence of adopting relations as the user's perception of the way the data is organized, application programs become independent of any ordering of tuples in storage that might be in effect at some time. This enables the stored ordering of tuples to be changed whenever necessary without adversely affecting the correctness of application programs.

Changes in the stored ordering may have to be made for a variety of reasons. For example, the pattern of traffic on the database may change, and consequently the ordering previously adopted may no longer be the most suitable for obtaining good performance.

A mathematical relation has one property that some people consider counter-intuitive, and that does not appear to be consistent with the definition in *The Oxford English Dictionary*. This property is that a unary relation (degree one) can conform to this definition. Thus, a mathematical relation of degree greater than one does inter-relate two or more objects, while a mathematical relation of degree one does not. In some cases, intuition can be a poor guide. In any event, whether the concept of a unary relation is counter-intuitive or not, mathematicians and computer-oriented people do not like to treat it any differently from relations of higher degree.

In applying computers effectively (whether in science, engineering, education, or commerce) there is, or should be, a similar concern for precise communication, a high level of abstraction, and generality. If one is not careful, however, the degree of generality can sometimes be pursued beyond what is needed in practice, and this can have costly consequences.

A relation R in the relational model is very similar to its counterpart in mathematics. When conceived as a table, R has the following properties:

- each row represents a tuple of R;
- the ordering of rows is immaterial;
- all rows are distinct from one another in content.

From time to time, objects are discussed that violate the last item listed, but that are mistakenly called relations by vendors of database management systems (abbreviated DBMS). In this book, such objects are called *improper relations* or *corrupted relations*. Reasons why improper relations should *not* be supported in any database management system are discussed in Chapter 23.

The fact that relations can be perceived as tables, and that tables are similar to flat files, breeds the false assumption that the freedom of action permitted [with] tables or flat files must also be permitted when manipulating relations. The manipulation differences are quite strong. For example, rows that entirely duplicate one another are not permitted in relations. The more disciplined approach of the relational model is largely justified because the database is shared by many people; in spite of the heavy traffic, all of the information in that database must be maintained in an accurate state.

The concept of a relation in the relational model is slightly more abstract than its counterpart in mathematics. Not only does the relation have a name, but each column has a distinct name, which often is not the same as the name of the pertinent set (the domain) from which that column draws its values. There are three main reasons for using a distinct column name:

1. such a name is intended to convey to users some aspect of the intended meaning of the column;
2. it enables users to avoid remembering positions of columns, as well as which component of a tuple is next to which in any sense of "nextness;"
3. it provides a simple means of distinguishing each column from its underlying domain. A column is, in fact, a particular use of a domain.

One reason for abandoning positional concepts altogether in the relations of the relational model is that it is not at all unusual to find database relations, each of which has as many as 50, 100, or even 150 columns. Users therefore are given an unnecessary burden if they must remember the ordering of columns and which column is next to which. Users are far more concerned with identifying columns by their names than by their positions, whether the positions be those in storage or those in some declaration. It makes much more sense for a user to request an employee's date of birth by name than by what its position happens to be (for example, column # 37).

One reason for discussing relations in such detail is that there appears to be a serious misunderstanding in the computer field concerning relations. There is a widely held misconception that, for one collection S of data to be related to another collection T, there must exist a pointer or some kind of link from S to T that is exposed to users. A pointer to T, incidentally, has as its value the storage address of some key component of T. A recent article [Sweet 1988] shows that this false notion still exists.

***Table 1.1*** **Relations in Mathematics Versus Relations in the Relational Model**

| Mathematics | Relational Model |
|---|---|
| Unconstrained values | Atomic values |
| Columns not named | Each column named |
| Columns distinguished from each other by position | Columns distinguished from each other and from domains by name |
| Normally constant | Normally varies with time |

For many reasons, pointers are extremely weak in supporting relations. In fact, an individual pointer is capable of supporting no more than a relation of degree 2, and even then supports it in only one direction. Moreover, pointers tend to foster needlessly complex structures that frustrate interaction with the database by casual users, especially if they are not programmers. The slow acceptance of artificial intelligence (AI) programs has been largely due to the use of incredibly complex data structures in that field. This supports the contention that AI researchers write their programs solely for other AI researchers to comprehend.

It is therefore a basic rule in relational databases that there should be no pointers at all in the user's or programmer's perception. For implementation purposes, however, pointers can be used in a relational database management system "under the covers," which may in some cases allow the DBMS vendor to offer improved performance.[1]

The term "relation" in mathematics means a fixed relation or constant, unless it is explicitly stated to be a variable. In the relational model exactly the reverse is true: every relation in the relational model is taken to be a variable unless otherwise stated. Normally it is the extension of relations (i.e., the tuples or rows) that is subject to change. Occasionally, however, new columns may be added and old columns dropped without changing the name of the relation.

The distinctions between the relation of mathematics and that of the relational model are summarized in Table 1.1.

A final note about relations: every tuple or row coupled with the name of the relation represents an assertion. For example, every row in the EMPLOYEE relation is an assertion that a specific person is an employee of the company and has the immediate single-valued properties cited in the row. Every row in the CAN_SUPPLY relation is an assertion that the cited supplier can supply the cited kind of part with the cited speed in the cited minimum package and at the cited cost. It is this general fact that makes relational databases highly compatible with knowledge bases.

1. Occasionally it is necessary in this book to discuss some features of database management that are at too low a level of abstraction to be included in the relational model. When this occurs, such features are said to be *under the covers* or *hidden from the user's view.*

In a reasonably complete approach to database management, it is not enough to describe the types of structure applied to data. In the recent past, numerous inventors have stopped at that point, omitting the operators that can be used in query and manipulative activities, data-description techniques, authorization techniques, and prevention of loss of integrity (see [Chen 1976] for an example). All of these capabilities should be designed into the DBMS from the start, not added afterwards. If a similar approach were adopted in medical science, all that would be taught is anatomy. Other important subjects such as physiology, neurology, and cardiology would be omitted. Database management has many facets in addition to the types of structure applied to data. Of key importance is the collection of operators that can be applied to the proposed types of data structure.

The relational model provides numerous operators that convert one or more relations into other relations; these are discussed in Chapters 4 and 5. Very few of these operators were conceived by mathematicians before the relational model was invented. One probable reason for this was the widely held belief that any problem expressed in terms of relations of arbitrary degree can be reduced to an equivalent problem expressed in terms of relations of degree one and two. My work on normalizing relations of assorted degrees shows this belief to be false.

## 1.2 ■ The Relational Model

A database can be of two major types: production-oriented or exploratory. In commerce, industry, government, or educational institutions, a production-oriented database is intended to convey at all times the state of part or all of the activity in the enterprise.

An exploratory database, on the other hand, is intended to explore possibilities (usually in the future) and to plan possible future activities. Thus, a production-oriented database is intended to reflect reality, while an exploratory database is intended to represent what might be or what might happen. In both cases the accuracy, consistency, and integrity of the data are extremely important.

Database management involves the sharing of large quantities of data by many users—who, for the most part, conceive their actions on the data independently from one another. The opportunities for users to damage data shared in this way are enormous, unless all users, whether knowledgeable about programming or not, abide by a discipline.

The very idea of a discipline, however, is abhorrent to many people, and I understand why. For example, I have encountered those who oppose a special feature of the relational model, namely, the prohibition of duplicate rows within all relations. They declare, "Why shouldn't I have duplicate rows if I want them? I am simply not prepared to give up my freedom in this regard." My response is as follows. If the data were a purely private concern (to just this single user), it would not matter. If, on the other hand, the data is shared or is likely to be shared sometime in the future, then *all*

*of the users of this data* would have to agree on what it means for a row to be duplicated (perhaps many times over). In other words, the sharing of data requires the sharing of its meaning. In turn, the sharing of meaning requires that there exist a single, simple, and explicit description of the meaning of every row in every relation. This is necessary even though one user may attach more importance to some facet of the meaning than some other user does.

Returning to the questionable support of duplicate rows, if the DBMS supports duplicate rows or records, it must be designed to handle these duplicates in a uniform way. Thus, there must be a general consensus among *all of the users of a DBMS product* regarding the meaning of duplicate rows, and this meaning should *not* be context-sensitive (i.e., it should not vary from relation to relation). My observation is that no such consensus exists, and is not likely to exist in the future.

For this reason and others, the discipline needed for the successful sharing of important data should be embodied within the database management system. The relational model can be construed as a highly disciplined approach to database management. Adherence to this discipline by users is enforced by the DBMS provided that this system is based whole-heartedly on the relational model.

As a normal mode of operation, if a user wishes to interpret the data in a database differently from the shared meaning, the DBMS should permit that user to extract a copy of the data from the database for this purpose (provided that the user is suitably authorized), and should disallow re-entry of that data into the database.

The management of shared data presents significantly tougher problems than the management of private data. Also, the role of shared data in efficiently carrying out business and government work is rapidly becoming a central concern. These two facts strongly suggest that no compromises be made on the quality of systems that manage the sharing of data simply to support a small minority of users of private data.

### 1.2.1 Relation as the Only Compound Data Type

From a database perspective, data can be classified into two types: atomic and compound. Atomic data *cannot* be decomposed into smaller pieces by the DBMS (excluding certain special functions). Compound data, consisting of structured combinations of atomic data, *can* be decomposed by the DBMS.

In the relational model there is only one type of compound data: the relation. The values in the domains on which each relation is defined are required to be atomic with respect to the DBMS. A relational database is a collection of relations of assorted degrees. All of the query and manipulative operators are upon relations, and all of them generate relations as results. Why focus on just one type of compound data? The main reason is that any additional types of compound data add complexity without adding power.

This is particularly true of the query and manipulative language. In such a language it is essential to have at least four commands: **retrieve, insert, update,** and **delete.** If there are $N$ distinct types of compound data, then for these four operations $4N$ commands will be necessary. By choosing a single compound data type that, by itself, is adequate for database management, the smallest value (one) is being selected for $N$.

In non-relational approaches to database management, there was a growing tendency to expose more and more distinct types of compound data. Consequently, the query and manipulative languages were becoming more and more complicated, and at the same time significantly less comprehensible to users, even those who were knowledgeable about programming.

Relational databases that have many relations, each with few rows (tuples), are often called *rich,* while those that have few relations, each with many rows, are called *large.* Commercial databases tend to be large, but not particularly rich. Knowledge databases tend to be rich, but not particularly large.

About six years after my first two papers on the relational model [Codd 1969 and 1970], Chen [1976] published a technical paper describing the entity-relationship approach to database management. This approach is discussed in more detail in Chapter 30, which deals with proposed alternatives to the relational model. Although some favor the entity-relationship approach, it suffers from three fundamental problems:

1. Only the structural aspects were described; neither the operators upon these structures nor the integrity constraints were discussed. Therefore, it was not a data model.
2. The distinction between entities and relationships was not, and is still not, precisely defined. Consequently, one person's entity is another person's relationship.
3. Even if this distinction had been precisely defined, it would have added complexity without adding power.

Whatever is conceived as entities, and whatever is conceived as relationships, are perceived and operated upon in the relational model in just one common way: as relations. An entity may be regarded as inter-relating an object or identifier of an object with its immediate properties. A relationship may be regarded as a relation between objects together with the immediate properties of that relationship.

### 1.2.2 Inter-relating the Information Contained in Distinct Relations

Some people who are used to past approaches find it extremely difficult to understand how information in distinct relations can possibly be inter-related by the relational model without the explicit appearance in the user's perception of pointers or links.

The fundamental principle in the relational model is that all inter-relating is achieved by means of comparisons of values, whether these values identify objects in the real world or whether they indicate properties of those objects. A pair of values may be *meaningfully compared,* however, if and only if these values are drawn from a common domain.

Some readers may consider the "common domain" constraint to be an unnecessary restriction. The opportunities for comparing values even with this constraint, however, are vastly superior in numbers and quality over the old approach of requiring pointers, links, or storage contiguity. Regarding the numbers, it should be remembered that not only object-identifiers can be compared with each other, but also simple properties of objects. Regarding the quality, those relational operators that involve the comparing of values require the values that are compared to be drawn from a common domain. In this way, these operators protect users from making very costly kinds of errors.

An example may help. Suppose the database contains serial numbers of suppliers and serial numbers of parts. Then, the immediate properties of a supplier contained in the SUPPLIER relation can be inter-related to the immediate properties of a supplier's capability contained in the CAPABILITIES relation by means of a single relational operator. This operator is the **equi-join,** and its application in this case involves comparing for equality the serial numbers of suppliers in the SUPPLIERS relation with those serial numbers in the CAPABILITIES relation.

Suppose that values for the serial numbers of suppliers and parts happen to have the same basic data type (i.e., character strings of the same length). Naturally, it is not meaningful to compare the supplier serial number in the SUPPLIER relation with the part serial number in the CAPABILITIES relation, even though they happen to have the same basic data type. Thus, the domain concept plays a crucial role in the inter-relating game. In fact, in Chapter 3 I discuss the general problem of determining of a given collection of relations whether they are all inter-relatable; domains are an essential and central concept in that discussion.

### 1.2.3 Examples of Relations

Two examples of relations in the relational model, described next, are intended to convey the structural uniformity of the approach to representing the information in relational databases for users (including application programmers). The first of these examples is the parts relation P, which identifies and describes each kind of part carried in inventory by a manufacturer.

| | |
|---|---|
| P# | Part serial number |
| PNAME | Part name |
| SIZE | Part size |
| QP | Quantity of parts |

| | |
|---|---|
| OH_QP | Quantity of parts on hand |
| OO_QP | Quantity of parts on order |
| MOH_QP | Minimum quantity of parts to be in inventory |

There are only four domains: P#, PNAME, SIZE, and QP.

| | | | | QP | | |
|---|---|---|---|---|---|---|
| **P** | **P#** | **PNAME** | **SIZE** | **OH_QP** | **OO_QP** | **MOH_QP** |
| | p1 | nut | 10 | 500 | 300 | 400 |
| | p2 | nut | 20 | 800 | 0 | 300 |
| | p3 | bolt | 5 | 400 | 200 | 300 |
| | p4 | screw | 12 | 1200 | 0 | 800 |
| | p5 | cam | 6 | 150 | 150 | 100 |
| | p6 | cog | 15 | 120 | 200 | 100 |
| | p7 | cog | 25 | 200 | 50 | 100 |

This relation has six columns and therefore is of degree six. All of the rows (seven in this example) constitute the *extension* of the parts relation P. Sometimes the extension of a relation is called its *snapshot.* The remaining descriptive information constitutes the *intension* of the parts relation P.

The second example, the capabilities relation C, is intended to provide information concerning which suppliers can supply which kinds of parts.

In many approaches to database management, such a concept is treated entirely differently from the information concerning parts, differently from both the structural and the manipulative points of view. (Most of these approaches are pre-relational.) Parts are called entities, while each capability is called a relationship between suppliers and parts. The problem is that capabilities have immediate properties just as parts do. Examples of properties that are applicable to capabilities are the 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.

This example may help the reader understand why, in the relational model, precisely the same structure is adopted for capabilities as for parts—and, more generally, precisely the same structure is adopted for entities as for relationships between entities.

| | |
|---|---|
| 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 | 5 | 100 | 20 |
| s1 | p6 | 12 | 10 | 600 |
| s2 | p3 | 5 | 50 | 37 |
| s2 | p4 | 5 | 100 | 15 |
| s3 | p6 | 5 | 10 | 700 |
| s4 | p2 | 5 | 100 | 15 |
| s4 | p5 | 15 | 5 | 300 |
| s5 | p6 | 10 | 5 | 350 |

This relation has five columns and is therefore of degree five. All of the rows (nine in this example) constitute the *extension* of the capabilities relation C. The remaining descriptive information constitutes the *intension* of relation C.

### 1.2.4 Omission of Features

When implementing a relational database management system, many questions arise regarding the relational model. Occasionally, support for some basic feature has been omitted due to it being assessed as useless. Unfortunately, the relational model has always had features that are inextricably intertwined. This means that omission of one feature of the model in a DBMS can inhibit implementation of numerous others. For example, omission of support for primary keys and foreign keys (defined in Section 1.8) jeopardizes the implementation of

- view updatability (see Chapter 17),
- the principal integrity constraints (see Chapter 13), and
- logical data independence (see Chapter 20).

### 1.2.5 The Goals of RM/V2

Version 2 of the relational model (abbreviated RM/V2) now has 333 features, which are even more inextricably intertwined than the approximately 50 features of Version 1 (abbreviated RM/V1). Most of the original definitions and features of RM/V1 have been preserved unchanged in Version 2. A very few of the original definitions and features have been extended to become broader in scope!

In late 1978 I developed an extended version of the relational model called RM/T [Codd 1979]. A principal aim was to capture more of the meaning of the data. Acceptance of the ideas in this version has been exceptionally slow. Consequently, it seems prudent to develop a sequence

of versions V1, V2, V3, . . . that are more gradual in growth. As the development of this sequence proceeds, certain features of RM/T will be selected and incorporated in appropriate versions.

My goals in developing Version 2 of the relational model included all those for the original relational model, RM/V1. Three of the most important of these goals were, and remain,

1. simplifying interaction with the data by users
   a. who have large databases,
   b. who need not be familiar with programming, and
   c. who normally conceive their interactions independently from all other users;
2. substantially increasing the productivity of those users who are professional programmers;
3. supporting a much more powerful tool for the database administrator to use in controlling who has access to what information and for what purpose, as well as in controlling the integrity of the database.

If these goals were attained, and I believe they have been, the market for DBMS products would be expanded enormously. This suggests one more goal, namely, that a very strong emphasis be placed on the preservation by the DBMS of database integrity. Chapters 13 and 14 are devoted to the treatment of integrity by the relational model. The DBMS products available so far have supported very few of the integrity features of the model.

It is the database administrator (abbreviated DBA) who is responsible for imposing controls on the database: controls that are adequate for the DBMS to maintain the database in a state of full integrity, as well as controls that permit access for specified purposes to only those users with authorized access for those purposes. The DBMS, however, must provide the DBA with the tools to carry out his or her job. Pre-relational DBMS products failed to provide adequate tools for this purpose.

Implementation of a high-performance DBMS that supports every feature of RM/V2 is not claimed to be easy. In fact, it is quite a challenging task. There are already clear indications that the DBMS products leading in performance and in fault tolerance will be those based on new hardware and software architectures, both of which exploit the many opportunities for concurrency provided by the relational model.

### 1.2.6 The Relational Model as an Abstract Machine

The term "abstract" scares many people who work in computing or data processing, even though they deal with abstractions every day. For example, speed and distance are abstractions. An airline reservation is an abstraction. Bits and bytes are abstractions. So are computer commands.

In my book *Cellular Automata* [Codd 1968], I make use of at least four levels of abstraction to explain concisely how a self-reproducing computer that is capable of computing all computable functions might be designed from a large number (in fact, millions) of simple identical cells, each of which interacts with only its immediate neighbors.

It is useful to think of RM/V2 as an abstract machine. Its level of abstraction is sufficiently high that it can be implemented in many distinctly different ways in hardware, in software, or in both. This machine can be advantageously treated by all DBMS vendors, standards committees, and DBMS users as an abstract machine standard.

For example, consider the structural features introduced in Chapter 2. Their level of abstraction is necessary for enabling different types of hardware and software (possibly from different vendors) to communicate with one another about their databases. The abstract machine must be complemented with standards that deal with the following:

- the physical representation of data for inter-computer communication;
- transaction-control signals to facilitate adequate control of each transaction that straddles two or more computer systems (e.g., the signal from one system to the other "Are you ready to commit your data?");
- specific relational languages that have specific syntax.

These topics are discussed in detail in Chapters 24 and 25.

### 1.2.7 The Structured Query Language (SQL)

Many people may contend that a specific relational language, namely SQL, already exists as a standard. SQL, standing for structured query language, is a data sublanguage invented by a group in IBM Research, Yorktown Heights, N.Y. [IBM 1972].

SQL was invented in late 1972. Although it was claimed that the language was based on several of my early papers on the relational model, it is quite weak in its fidelity to the model. Past and current versions of this relational language are in many ways inconsistent with both of the abstract machines RM/V1 and RM/V2. Numerous features are not supported at all, and others are supported incorrectly. Each of these inconsistencies can be shown to reduce the usefulness and practicality of SQL.

The most noteworthy error in several current implementations is that SQL permits the use and generation of improper or corrupted relations, that is, "relations" that contain duplicate tuples or rows. In Chapter 23 this problem is examined in sufficient detail to demonstrate the seriously adverse consequences of this very simple infidelity to the model. As each feature of RM/V2 is introduced in this book, the attempt is made to comment on SQL's support, non-support, or violation of the feature.

Several relational languages other than SQL have been developed. An

example that I consider superior to SQL is Query Language (abbreviated QUEL). This language was invented by Held and Stonebraker at the University of California, Berkeley, and was based on the language ALPHA [Codd 1972]. More attention is devoted to SQL, however, because it has been adopted as an ANSI standard, and is supported in numerous DBMS products.

### 1.2.8 An Abstract Machine Standard

The computing field clearly needs an abstract machine standard for database management for at least the following reasons:

- The intrinsic importance of computer-based support for the sharing of business information interactively and by program.
- The users involved in this sharing normally conceive their modifications of the information independently of one another.
- Clearly the field of database management is moving toward the management of distributed databases, and at each of the sites involved there may be hardware and software from a variety of vendors. Intercommunication among these systems will be a vital requirement.
- The boundary between hardware and software is moving out from the von Neumann position. Hardware is taking on more of the tasks previously handled by basic software, and already there are products in which numerous components of operating systems and DBMS are supported by hardware. An abstract machine standard for database management should enable this boundary to move without the necessity of continual reformulation of a new standard.

The relational model deals with database management from an abstract, logical point of view only, never at the detailed level of bits and bytes. Does this make the relational model incomplete? If incomplete in this sense, the model is intended to be this way. It is important to stop short of prescribing how data should be represented and positioned in storage, and also how it should be accessed. This not only makes users and programmers more productive, but also permits both hardware and software vendors to compete in lower-level techniques for obtaining good performance.

This is an area of considerable technical significance in which DBMS vendors can productively compete with one another. In the case of DBMS products that are carefully based on the relational model, such competition need not adversely affect the users' investment in training and application development, precisely because their perception is at a high level of abstraction.

One final reason for a high level of abstraction is that the choice of representation for data in storage and the choice of access methods depend heavily on the nature and volume of the traffic on the database.

Publication of the relational model in the June 1970 issue of *Communications of the Association for Computing Machinery* [Codd 1970] preceded the completion of development of relational DBMS products by at least a decade. The model is more abstract than these systems and has an existence that is completely independent of them.

This is an advantage in many ways. First, it provides a useful goal, target, and tool for the designers and developers of new DBMS products. Second, it provides a special kind of standard against which dissimilar DBMS products can be measured and compared. No DBMS product or data sublanguage marketed in the western world today fully supports each and every feature of the relational model, even Version 1 [Codd 1969, 1970, 1971a–d, 1974a]. Third, it provides a foundation upon which theoretical work in database management has been and will continue to be based.

## 1.3 ■ The Transaction Concept

Brief reference was made to the concept of a transaction in the preceding discussion of the additional kinds of standards that are now needed. In the relational model, this concept has a precise definition.

A *transaction* is a collection of activities involving changes to the database, all of which must be executed successfully if the changes are to be committed to the database, and none of which may be committed if any one or more of the activities fail. Normally, such a collection of activities is represented by a sequence of relational commands. The beginning of the sequence is signaled by a command such as BEGIN or BEGIN TRANSACTION. Its termination is signaled by a command such as END or COMMIT—or, if it is necessary to abort the transaction, ABORT.

A simple example of a transaction is that in which a bank customer requests the bank to transfer $1,000 from his checking account into his savings account. In the bank's computer program the first action is to check that there is at least $1,000 in the customer's checking account. If so, this amount is deducted from the balance in that account. The next action is to credit the customer's savings account with the $1,000.

If the first action were successful and the second action failed (due to hardware malfunction, for example), the customer would lose the $1,000; this would be unacceptable to most customers. Therefore, this is a case in which both actions must succeed or neither must cause any change in the database.

In DBMS products two methods of handling a transaction are as follows:

1. to delay storing in the database any data generated during execution of a transaction until the DBMS encounters a COMMIT or END TRANSACTION command, and then store all of this data;
2. to write details of each change in the recovery log as each change is generated, and immediately record the change in the database. This log

is then used for recovery purposes if an ABORT TRANSACTION command is to be executed.

## 1.4 ■ Classification of RM/V2 Features

Each feature of the relational model RM/V2 is assigned to one of the 18 classes listed in Table 1.2. The table includes the number of the chapter in which each class of features is described. Each letter identifies the class. Each feature has a unique label. Thus, in the feature RS-9, R stands for relational, S for the structure class, and 9 for the ninth feature in that class. The numbering of features within a class should be interpreted as a distinctive label only, not as an ordering of importance.

There is no claim that the features of RM/V2 are all independent of one another. In fact, as discussed earlier, there are numerous inter-dependencies among the features. A minimal, totally non-redundant set would be more difficult to understand, would probably reduce user productivity significantly, and would probably lead to even more errors by vendors in designing their relational DBMS products. Of course, I am not advocating

*Table 1.2* **The 18 Classes of Features and the Pertinent Chapters**

| Chapter | Label | Class |
|---|---|---|
| 2 | S | Structural |
| 3 | T | Extended data types |
| 4 | B | Basic operators |
| 5 | Z | Advanced operators |
| 6 | N | Naming |
| 7 | E | Elementary commands |
| 10 | Q | Qualifiers |
| 11 | J | Indicators |
| 12 | M | Manipulative |
| 13,14 | I | Integrity |
| 15 | C | Catalog |
| 16,17 | V | View |
| 18 | A | Authorization |
| 19 | F | Scalar and aggregate functions |
| 20 | P | Protection of user investment in the DBMS, the database, application programming, and user training |
| 21 | D | DBMS design principles |
| 22 | L | Language-design principles |
| 24,25 | X | Distributed database management |

the other extreme, namely complexity, since this runs counter to comprehensibility.

Two very important concepts of relational DBMS products are the *catalog* and *views*. Some think that the basic relational model does not mention the catalog or views, but these concepts were discussed in my early papers on the relational model, although not by these names. I referred to the catalog as the *relational schema* and to views as *derived relations* whose definitions were stored in the schema. In this book I have adopted the System R terms "catalog" and "view" [Chamberlin et al. 1981] because they are concise and very usable, and are now quite widely used. System R was one of three DBMS prototypes developed in distinct divisions of IBM and based on the relational model.

When DBMS products are evaluated today, the evaluation should include fidelity of the product to the relational model, and specifically RM/V2. In part, this is required because almost all vendors claim that their DBMS products are relational. Therefore, one important concern for potential users of these products is that they reap all the benefits of fidelity to the relational model.

As with RM/V1, the features that are included in RM/V2 are intended to be helpful for all users of relational DBMS, both application programmers and end users. Also, as with RM/V1, they are intended to help the designers, implementors, and evaluators of relational DBMS products. RM/V2 features include all the features of RM/V1, together with the following:

- new features that cover important aspects of relational DBMS not previously included in RM/V1, either because they were overlooked or because I considered them too obvious to mention, until I discovered that many people had not realized their obvious importance;
- new features that are in line with the original RM/V1, but at a slightly lower level of abstraction.

A DBMS product is *fully relational* in the 1990s if it fully supports each and every one of the features of RM/V2 defined in this book. A DBMS product that is not fully relational can nevertheless qualify to be called *relational* in the early 1990s by fully supporting each one of the roughly 40 features listed in Appendix A.

Like the basic relational model RM/V1, all the features of RM/V2 are based on the practical requirements of users, database administrators, application programmers, security staff, and their managers. Along with the description of each feature, I attempt to explain the practical reasons for that feature.

The relational model RM/V2 is based on the original model RM/V1 and on a single fundamental rule, which I call Rule Zero:

> For any system that is advertised as, or claimed to be, a relational database management system, *that system must be able to manage*

*databases entirely through its relational capabilities,* no matter what additional capabilities the system may support.

This must hold *whether or not* the system supports any non-relational capabilities of managing data. Any DBMS that does *not* satisfy this Rule Zero is *not* a *relational* DBMS.

The danger to buyers and users of a system that is claimed to be a *relational* DBMS and fails on Rule Zero is that these buyers and users will expect all the advantages of a truly relational DBMS, but will fail to receive them.

One consequence of Rule Zero is that any system claimed to be a relational DBMS must support database insert, update, and delete at the *relational* level (multiple-record-at-a-time). (See Feature RM-6 in Chapter 12.) Another consequence is the necessity of supporting the information feature and the guaranteed-access feature. (See Feature RS-1 in Chapter 2 and Features RM-1 and RM-2 in Chapter 12.)

Incidentally, "multiple-record-at-a-time" includes the ability to handle those situations in which zero or one record happens to be retrieved, inserted, updated, or deleted. In other words, a relation (often carelessly called a table) may have either zero tuples (rows) or one tuple (row) and still be a valid relation. Note that although it may be unusual for a relation to have either zero rows or one row, it does not receive special treatment in the relational model, and therefore users do not have to treat such a relation in any special way either.

## 1.5 ■ Tables versus Relations

Actually, the terms "relation" and "table" are not synonymous. As discussed earlier, the concept of a relation found in mathematics and in the relational model is that of a *special kind of set.* The relations of the relational model, although they may be *conceived* as tables, are then special kinds of tables. In this book they are called *R-tables,* although the term "relation" is still used from time to time to emphasize the underlying concept of mathematical sets, to refer to the model, or to refer to languages developed as part of implementations of the model.

*R-tables have no positional concepts.* One may shuffle the rows without affecting information content. Thus, there is no "nextness" of rows. Similarly, one may shuffle the columns without affecting information content, providing the column heading is taken with each column. Thus, there is no "nextness" of columns.

Normally, neither of these shuffling activities can be applied with such immunity to arrays. That is why I consider it extremely misleading to use the term "array" to describe the structuring of data in the relational model.

Those relations, or R-tables, that are internally represented by stored data in some implementation-defined way are called the *base relations* or

*base R-tables.* All R-tables other than base R-tables are called *derived relations* or, synonymously, *derived R-tables.* An example of a derived relation is a *view.* A view is a virtual R-table defined in terms of other R-tables, and is represented by its defining expression only.

In both RM/V1 and RM/V2, *duplicate rows are not permitted in any relations,* whether base relations, views, or any other type of relations. For details, see Features RS-3 and RI-3 in Chapters 2 and 13, respectively. This rule has been applied in all of my technical papers on the relational model, even the first one [Codd 1969].

In RM/V2, duplicate rows are still excluded from all relations. They are excluded from *base R-tables,* primarily as a step to retain integrity of the database: each row in such a table represents an object whose distinctiveness is lost if duplicate rows are allowed in these R-tables. A very fundamental property of the relational model is that each object about which information is stored in the database must be uniquely and explicitly identified, and thereby distinguished from every other object. As we shall see, the unique identifier is the name of the R-table, together with the primary key value. This fundamental property, an integrity-preservation feature, is not enforced by any other approach to database management.

Duplicate rows are still excluded from all *derived R-tables* for semantic reasons (see Fundamental Law 20 in Chapter 29). They are also excluded because such duplicates severely reduce the interchangeability of sequencing of operators within a relational command or in a sequence of commands. This reduction in interchangeability has two serious consequences (see Chapter 23 for more detail):

1. it reduces the optimizability of relational commands;
2. it imposes severe conceptual problems and severe constraints on users.

Two of the early prototypes of relational DBMS products were developed in the mid-1970s by the System R team at IBM Research in San Jose, Calif. [Chamberlin et al. 1981] and the INGRES team at the University of California Berkeley [Stonebraker 1986]. Curiously, both of these teams made the same two criticisms of the relational model:

1. the expected loss of performance if duplicate rows had to be eliminated in several types of relational operations without the user explicitly requesting that elimination;
2. the alleged impossibility of applying statistical functions correctly to columns that happen to have duplicate values legally.

Based on the first point, I conditionally agreed to the idea that duplicate rows should be permitted in derived R-tables *only,* not in base R-tables. The condition for this concession was that all of the effects upon the relational operators be carefully examined for possibly damaging consequences.

On the second point, I found myself in strong disagreement, because the mistake made in these two prototypes was to apply as a first step the projection operator (see Chapter 4) on the column or columns for which statistics were needed. Instead, I advised the researchers to apply the statistical function first in the context of whatever relation was given, and then apply the projection operator, only if such action were necessary for other reasons.

It now appears that neither project adequately examined the severely damaging effects of duplicate rows (1) on the operators and (2) on common interpretability by users (see Chapter 23).

This latter concern is related to the fact that, when hundreds, possibly thousands, of users share a common database, *it is essential that they also share a common meaning for all of the data therein that they are authorized to access.* There does not exist a precise, accepted, context-independent interpretation of duplicate rows in a relation. These adverse consequences are the reason that I still find that duplicate rows in any relation are unacceptable.

Let us turn our attention to a table that is extracted from a non-relational source for storage in a relational database. If it happens to contain duplicate rows, these duplicate rows can easily be removed by means of a special operator (see Feature RE-17 in Chapter 7). This operator removes all except one occurrence of each row that has multiple occurrences. It leaves the table unchanged if it happens to contain no duplicate rows.

From an evaluation standpoint, the RM/V2 features defined in this book have been created with primary concern for those DBMS products that are designed to support multiple users concurrently accessing shared data and engaged in tasks that can be (and often are) conceived independently of one another. Therefore, some of the features are not applicable to a DBMS intended for very small computer systems, particularly single-user systems such as personal computers.

An example of such a feature is concurrency control. Although locking as a form of concurrency control is mentioned in very few features of RM/V2, it is accompanied by the phrase "or some alternative technique for concurrency control that is at least as powerful as locking (and provably so)." I plan to say more about the subject of locking in a forthcoming book on computer-aided development (CAD) and engineering extensions to the relational model.

The logic that is usually encountered in data processing is propositional logic, often called Boolean logic, which deals with only two truth-values: TRUE and FALSE. In the field of database management, one reason that propositional logic was considered adequate in the past is that, before the relational model, logic was considered relevant to query products only, and such products normally supported the use of logic in the querying of single files only. Two truth values were considered adequate because no attempt was made to handle missing values in a uniform and systematic manner across the entire database.

Mathematical logic plays a central role in the relational model. In RM/V1 the logic is *three-valued,* first-order predicate logic, where the three truth-values are TRUE, FALSE, and MAYBE. This logic is substantially more powerful than propositional logic. The MAYBE truth-value means that the DBMS cannot decide whether a truth-valued expression is TRUE or FALSE due to values missing from the database.

In RM/V2 this logic is extended to *four-valued,* first-order predicate logic, where the four truth-values are TRUE, FALSE, MAYBE BUT APPLICABLE, and MAYBE BUT INAPPLICABLE. This is especially relevant when it becomes necessary to handle information that may contain some database values that are applicable but missing because they have not been entered yet, and some values that are missing because the property in question is inapplicable to the pertinent object (see Chapter 8 on missing information).

## 1.6 ■ Terminology

In this account of RM/V2, several terms that are now popular in database management are used, instead of the longer established and more carefully defined mathematical terms. Any ambiguity that is perceived in the use of the database-oriented terms can be resolved by referring to Table 1.3.

The degree $n$ of a relation is the number of columns, which can be any positive integer, including the special case of a unary relation for which $n = 1$. A relational database is perceived by all users, whether application programmers or end users, as a collection of relations of assorted degrees. Each relation can be thought of as inter-relating the identifying properties of a type of object with the other immediate properties of that type of object. Every value appearing in a relation is treated by the DBMS as atomic, except for certain special functions that are able to decompose certain kinds of values (see Chapter 19).

The phrases "delete duplicates" and "delete redundant duplicates" mean *delete all occurrences except one* of an object (the object is determined by the context in which this phrase is used, and it is usually a complete row of an R-table).

***Table 1.3*** **Mathematical and Database Terms**

| Mathematical Term | Database Term |
|---|---|
| Relation of degree *n* | R-table with n columns |
| Attribute | Column of R-table |
| Domain | Extended data type |
| Tuple | Row of R-table |
| Cardinality of relation | Number of rows in R-table |

In this book the terms "interrogation," "query," and "retrieval" are used synonymously. Each of these terms denotes a read-only operation. No data modification is involved. Notwithstanding their names, identifying the database languages SQL and QUEL as just query languages is quite incorrect, since both support much more than interrogation.

The terms "modification" and "manipulation" are used whenever data modification is involved, whether it be data entry, deletion, or updating. Except where otherwise indicated, the term "updating" denotes a particular kind of modification, namely, modification applied to values already within the database. Therefore, updating is normally an operation that is distinct from both data entry and deletion.

When applied to any database activity, the term "dynamically" means *without* bringing any database traffic to a halt.

## 1.7 ■ Role of Language in the Relational Model

Early in the development of the relational model (1969-1972), I invented two languages for dealing with relations: one algebraic in nature, and one based on first-order predicate logic [Codd 1971a]. I then proved that the two languages had equal expressive power [Codd 1971d], but indicated that the logic-based language would be more optimizable (assuming that flow tracing was not attempted) and easier to use as an interface to inferential software on top of the DBMS.

During subsequent development of the relational model, I have avoided the development of a specific language with specific syntax. Instead, it seemed appropriate that my work remain at a very high level of abstraction, leaving it to others to deal with the specific details of usable languages. Thus, the relational model specifies the semantics of these languages, and does not specify the syntax at all.

The abbreviation RL denotes the principal relational language supported by the DBMS—a language intended specifically for database management, and one that is not guaranteed to be usable for the computation of all computable functions. RM/V2 specifies the features that RL should have, and the specification is (as we just saw) semantic, not syntactic. Examples of existing relational languages are SQL and QUEL, although neither of these supports more than half the relational model.

The power of RL includes that of four-valued, first-order predicate logic [Church 1958, Suppes 1967, Stoll 1961, Pospesel 1976]. The complete power of RL should be fully exploitable in at least the following contexts:

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

- integrity constraints;
- authorization constraints;
- if the DBMS is claimed to be able to handle distributed data, distributed database management with distribution independence, including automatic decomposition of commands by the DBMS and automatic recomposition of results by the DBMS (see Feature RP-4 in Chapter 20).

One of the main reasons that "object-oriented" DBMS prototypes and products are not going to replace the relational model and associated DBMS products is these systems appear to omit support for predicate logic. It will take brilliant logicians to invent a tool as powerful as predicate logic. Even then, such an invention is not an overnight task—once invented, it might well take more than a decade to become accepted by logicians. Thus, features that capture more of the meaning of the data should be added to the relational model [Codd 1979], instead of being proposed as replacements.

In the development of application programs, a relational language normally needs as a partner a host language such as COBOL, PL/1, FORTRAN, or some more recently developed programming language. Some relational DBMS support several such host languages to be used as partners, although the user is normally required to select just one for developing an application program. In this book I occasionally use the term "HL" (for "host language") to denote such a language.

Languages are being developed that are significantly higher in level than COBOL, PL/1, and FORTRAN; such languages frequently include statements that must be translated into RL. Thus, an important requirement for RL is that it be both convenient and powerful in two roles: as a source language and as a target language.

Sometimes I am asked why I do not extend relational languages to include the features of PROLOG or of someone's favorite "fourth-generation" language. My usual reply is that I do not wish to tie the destiny of the relational model to any tool that has not been overwhelmingly accepted or does not appear to be defined at the same level of abstraction as the relational model. Moreover, I believe that the days of monstrous programming languages are numbered, and that the future lies with specialized sublanguages that can inter-communicate with one another.

## 1.8 ■ Keys and Referential Integrity

The term "key" has been used in the computing field for a long time, and with a great variety of meanings. In the relational model the term is normally qualified by the adjectives "candidate," "primary," and "foreign," and each of these phrases has a precisely defined meaning.

Each base R-table has exactly one primary key. This key is a combination of columns (possibly just one column) such that

- the value of the primary key in each row of the pertinent R-table identifies that row uniquely (i.e., it distinguishes that row from every other row in that R-table);
- if the primary key is composite and if one of the columns is dropped from the primary key, the first property is no longer guaranteed.

Sometimes these two properties are called the *uniqueness property* and the *minimality property,* respectively. Note, however, that "minimality" in this context does not mean the shortest in terms of bits or bytes or the one having the fewest components.

It is equally valid to interpret the uniqueness property in terms of object identification: the value of the primary key in each row of the pertinent R-table identifies the particular object represented by that row uniquely within the type of objects that are represented by that relation. Everywhere else in the database that there is a need to refer to that particular object, the *same* identifying value drawn from the *same* domain is used. Any column containing those values is called a *foreign key,* and each value in that column is called a *foreign key value.*

*Referential integrity* is defined as follows:

> Let $D$ be a domain from which one or more primary keys draw their values. Let $K$ be a foreign key, which draws its values from domain $D$. Every unmarked value which occurs in K must also exist in the database as the value of the primary key on domain D of some base relation.

Incidentally, a value in the database is *marked* if and only if it is missing. The subject of missing information is discussed in some detail in Chapters 8 and 9.

The case in which K 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, 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 current DBMS products, even when the vendors claim that their products support referential integrity.

To make use of this definition, it is necessary to understand primary keys (PK) and foreign keys (FK). The example in the following subsection is intended to give the reader some understanding of the semantic nature of these keys.

### 1.8.1 Semantic Aspects of Primary and Foreign Keys

Notice that referential integrity applies to pairs of keys only, one a primary key PK and the other a foreign key FK. The keys may be simple (single-column) or composite (two or more columns). The DBMS should not require

that each and every combination of simple keys within a single relation be treated as a foreign key, even if that combination appears as a composite primary key in the database. This is clearly an issue that is related to the meaning of the data.

Suppose, for example, that a database contains the relations listed in Table 1.4.

***Table 1.4*** **Example Relations in a Database**

| Relation | Meaning | Primary Key |
|---|---|---|
| R1 | Suppliers | S# |
| R2 | Parts | P# |
| R3 | The *capabilities* of suppliers to supply parts, including price and speed of delivery | (S#,P#) |
| R4 | *Orders* for parts placed with specified suppliers, including date the order was placed | (S#,P#, DATE) |

To avoid an extra relation and keep the example simple, assume that every order is a one-line order (that is, only one kind of part appears on any order) and that it is impossible for two orders with the same order date to refer to identical kinds of parts.

Suppose that each of two companies has a database of this kind. In company A, however, the relation R3 is used as advisory information, and there is no requirement that every combination of (S#,P#) that appears in R4 must appear in R3. In company B, on the other hand, R3 is used as controlling information: that is, if an order is placed for part p from supplier s, it is company policy that there must be at least one row in relation R3 stating that p is obtainable from s, and incidentally indicating the price and the speed of delivery. Of course, there may be other rows in R3 stating that p is obtainable from other suppliers. Thus, if referential integrity were applied to the combination (S#,P#) as primary key in R3 and foreign key in R4, it would be correct in company B, but incorrect in company A.

There are two ways in which this example (and similar ones) could be handled:

1. Make the referential integrity constraint applicable to all PK-FK pairs of keys (whether simple or composite) in which one key PK is declared to be primary, and the other key FK is declared to be foreign. In company B, declare the (S#,P#) combination in R4 as a foreign key that has as its target the (S#,P#) primary key of R3. In company A, avoid altogether the declaration that (S#,P#) in R4 is a foreign key.
2. Make the referential integrity constraint applicable to simple PK-FK pairs of keys only, and require the DBA to impose a referential con-

straint on just those compound PK-FK pairs of keys for which the constraint happens to be applicable in his or her company—by specifying a user-defined integrity constraint, expressed in the relational language.

Method 1 is the approach now adopted in the relational model. It makes the foreign-key concept a more semantic feature than does Method 2. After all, the concepts of keys in the relational model were always intended to identify objects in the micro-world that the database is supposed to represent. In other words, keys in the relational model act as surrogates for the objects being modeled. Once again, Method 1 is adopted.

### 1.8.2 Primary Keys on a Common Domain

Let us consider an example of the fact that primary keys on a given domain can occur in more than one base relation. This is the database in which there are two base R-tables that provide the immediate properties of suppliers: one for the domestic suppliers, one for the foreign suppliers. There would normally be some properties in the foreign suppliers table that do not occur in the domestic suppliers table. Each R-table has as its primary key the supplier serial number. Nevertheless, the database may contain several R-tables that include the supplier serial number as a foreign key without making any distinction regarding the R-tables in which the corresponding primary key value resides. In general, that value may reside as a primary key value in one, two, or even more R-tables.

No assumption is made in either RM/V1 or RM/V2 concerning the adoption of the tighter discipline of the extended relational model RM/T [Codd 1979]. For example, there is no requirement that type hierarchies be incorporated in the database design, wherever they are appropriate. Moreover, there is no requirement that, for each primary key, a unary relation (called the *E-relation* in RM/T) exists to list all of the distinct values in use for that primary key.

A second example in the non-distributed case is that of a base relation R that happens to have many columns, but a large amount of traffic on only 20% of these columns (call this A) and a very modest amount on the remaining 80% (call this B). In such a case the DBA may decide to improve performance by storing the data in the form of two base relations instead of one:

1. a projection of R onto its primary key together with A;
2. a projection of R onto its primary key together with B.

A specific feature of the relational model that requires support in the DBMS for multiple primary keys from a common domain is RS-10, described in the next chapter.

In the case of distributed database management, it is not at all uncommon

to have the information distributed in such a way that several relations at several sites all have a primary key based on a common domain. See Section 24.4 for a detailed discussion of the relational approach to distributing data.

### 1.8.3 Comments on Implementation

Referential integrity is discussed further in Chapter 13. It should be implemented as far as possible as a special case of user-defined integrity (see Chapter 14) because of their similarities. One such common need, for example, is to give the DBA or other authorized user the freedom to specify linguistically how the system is to react to any attempt to violate these integrity constraints, whether the constraints are referential or user-defined.

Further, it should be remembered that referential integrity is a particular application of an *inclusion constraint* (sometimes called an inclusion dependency). Such a constraint requires that the set of distinct values occurring in some specified column, simple or composite, must be a subset of the values occurring in some other specified column (simple or composite, respectively). In the case of referential integrity, the set of distinct simple FK values should be a subset of the set of distinct simple PK values drawn from the same domain.

Inclusion constraints, however, may apply between other pairs of attributes also (e.g., non-keys). When declared and enforced, such additional constraints reflect either business policies or government regulations. One would then like the DBMS to be designed in such a way as to provide reasonably uniform support for referential integrity and these additional (user-defined) inclusion constraints.

## 1.9 ■ More on Terminology

The following terms are used in connection with relational languages and user-defined functions.

- *retrieval targeting:* specifying the kinds of database values to be extracted from the database, and then possibly specifying transformations to be applied to occurrences of these values;
- *retrieval conditioning:* specifying a logical condition in a retrieval or manipulative statement of a particular relational language for the purpose of conditioning access;
- *PK-targeting:* finding the primary key(s) corresponding to any given foreign key;
- *FK-targeting:* finding the foreign key(s) corresponding to any given primary key;

- *PK-based projection:* a projection that includes the *primary key* of the operand R-table;
- *non-PK projection:* a projection that does not include the *primary key* of the operand R-table.

## 1.10 ■ Points to Remember

Four important points concerning relations follow:

1. every relation is a set;
2. *not* every set is a relation;
3. every relation can be perceived as a table;
4. *not* every table is a correct perception of a relation.

Designers of the relational DBMS products of many vendors appear to be ignorant of these facts or to have ignored them.

## Exercises

Note that, for some exercises, additional chapters are identified as sources of more information.

1.1 Identify the 18 classes of features in RM/V2. Supply a brief description of each class.

1.2 When a relation is perceived as a table, what are the special properties of that table? Is the ordering of columns crucial? Is the ordering of rows crucial? Can the table contain duplicate rows?

1.3 The terms "table" and "relation" are not synonymous. Supply a simple example of a table that is neither a relation of the relational model nor a relation of mathematics.

1.4 What is your position on the entity-relationship approach? (See also Chapter 30.) Will it replace the relational model? Give five technical reasons for your answer.

1.5 What is a transaction in the relational model? Describe an application that illustrates that there is a practical need for this concept.

1.6 Are either of the following statements true about the structures of the relational model?

They are merely flat files.
They are merely tables.

In each case, if your answer is no, give an example of a flat file that is not a relation or an example of a table that is not a relation.

1.7 What is your position on the object-oriented approach? (See also Chapter 30.) Will it replace the relational model? Give five reasons for your answer. You may wish to postpone this exercise, as well as Exercise 1.8, until you have absorbed Chapter 28.

1.8 Can any object-oriented concepts be added to the relational model without violating any of the principles on which the model is based? Which concepts? (See also Chapter 30.)

1.9 When designing a database, is it possible to anticipate all of the uses to which the data will be put? Is it possible to anticipate the batch load, on-line teleprocessing load, and interactive query load for the next three, five, or seven years? Conclude from your answer what properties the DBMS should have if it is to protect the user's investment in application programs. (See also Chapter 26.)

1.10 Are duplicate rows needed in a relation? If so, what for? Supply an example. Should duplicate rows be allowed in any relation? State reasons why or why not, whichever is applicable, and supply examples. (See also Chapters 2 and 23.)

1.11 In RM/V2 does the prohibition of duplicate rows within every relation imply that no duplicate values (e.g., currency values) can occur in any column? Explain.

■ *CHAPTER 2* ■

# Structure-Oriented and Data-Oriented Features

Chapters 2 through 25 describe and explain the 333 features of RM/V2. In this chapter attention is focused on the way a relational database is structured and how the information in various parts of the database is inter-related. Each feature has a brief title and a unique label of the type R*Y*-*n*, where *Y* is a character that denotes the pertinent class of features and *n* is the feature number within this class.

Reference is made occasionally to "the 1985 set" [Codd 1985]. This set of 12 rules, a quick means of distinguishing the DBMS products that are relational from those that are not relational, can still be used for coarse distinctions. The features of RM/V2, however, are needed for distinctions of a finer grain.

## 2.1 ■ Databases and Knowledge Bases

As explained in Chapter 1, both a commercial database in the relational sense and a knowledge base consist largely of assertions. In commercial databases most of the assertions contain no variables. There are few distinct kinds of assertions, and very many assertions of each type (perhaps hundreds of millions). In knowledge bases, on the other hand, most of the assertions contain variables that are bound in the logician's sense. Moreover, there are many distinct kinds of assertions, and very few of each type (often just one).

The relational model is intended to be applied primarily to commercial

and industrial databases. Thus, it takes advantage of the large numbers of assertions all of the same type. The predicate in the logician's sense that is common to all the assertions of one type is factored out and becomes the relation name.

There is a component of every relational database, however, that is very similar to a knowledge base, and that is the database description. Further discussion of this subject is postponed until Chapter 15. In any event, due to the focus of the relational model on cleanly expressed assertions unencumbered with irrelevant structural details (for example, the clutter of pointers), this model has an outstandingly clean interface to knowledge bases.

## 2.2 ■ General Features

### *RS-1* The Information Feature

> See Rule 1 in the 1985 set. The DBMS requires that all database information seen by application programmers (AP) and interactive users at terminals (TU) is cast explicitly in terms of values in relations, and *in no other way* in the *base relations.* Exactly one additional way is permitted in *derived relations,* namely, ordering by values within the relation (sometimes referred to as inessential ordering).

This means, for example, that, in the database, users see no repeating groups, no pointers, no record identifiers (other than the declared primary keys), no essential (i.e., non-redundant) ordering, and no access paths. Obviously this is not a complete list. *Such objects may, however, be supported for performance reasons under the covers* because they are then not visible to users, and hence impose no productivity-reducing burden on them.

### 2.2.1 Repeating Groups

Many people experienced with pre-relational systems express shock or dismay at the exclusion of repeating groups from the relational model. Repeating groups had their origin in the trailer cards of the punched-card era. Thus, it seems worthwhile to consider an example that illustrates how repeating groups can be avoided.

Suppose that, at database-design time, it is decided that the database is to contain information concerning parts in the inventory and orders placed for more parts to be added to the inventory. An order usually consists of heading information such as the name and address of the supplier with whom

the order is to be placed, an appropriate employee in the supplier company, the telephone number of this employee, the date when the supplier promises to ship the parts, the expected cost, and possibly other items of information. The line items then follow, and it is here that some think that a repeating group is needed.

In the relational approach the heading information for all orders is incorporated into a single relation, the *heading relation.* Similarly, the line items for all orders are incorporated into a single relation, the *trailing relation.* Each distinct order in the heading relation includes an order serial number as a unique identifier of the order. This identifier is, of course, the primary key of the heading relation. Every line item in the trailing relation includes the pertinent order serial number as a foreign key. This identifier, together with the line number for the pertinent line item, constitute the primary key of the trailing relation.

It will be seen that in eliminating repeating groups from the database design no information has been lost. Now the question arises, "Why eliminate repeating groups? Surely, they are both natural and harmless!"

In fact, they are not harmless. One of the principal penalties is that each repeating group must be positioned next to its heading information, and all members of a group must be positioned next to each other. The relational model avoids all decisions regarding positioning of information, allowing positioning to be used purely for performance purposes.

A second penalty is that repeating groups represent an additional way of representing information. Hence, one more retrieval command, one more insertion command, one more update command, and one more deletion command are needed in the data-manipulation vocabulary.

A third penalty is that logical database design now involves more decision making without a theoretical foundation upon which to base those decisions. Thus, if repeating groups were adopted, it would be necessary to conceive clear, concise, and rigorously established criteria for the database designers to choose whether or not to exploit repeating groups.

### 2.2.2 More on the Information Feature

Returning to the information feature RS-1, even R-table names, column names, and domain names are represented as character strings in some R-tables. R-tables containing such names are normally part of the built-in database catalog (see Chapter 15). The catalog is accordingly a relational database itself—one that is dynamic and active and that represents the *meta-data.* Meta-data consists of data that describes all of the data in the database, as well as the contents of the catalog itself.

The information feature is supported for several reasons. First, as explained in Chapter 1, this feature makes a simpler data sublanguage possible, and therefore supports user productivity. Second, it greatly simplifies the interface between the DBMS and software packages "on top of" the DBMS.

Examples of such software packages are application-development aids, expert systems, and dictionaries. These packages must not only interface with relational DBMS but, by definition, must be well integrated with the DBMS. This integration is necessary because these packages retrieve information already existing in the database (including the catalog) and, as needed, put new information in the database (and possibly in the catalog also).

An additional reason is to simplify the database administrator's task of maintaining the database in a state of overall integrity and to make this task more effective. There is nothing more embarrassing to a DBA than being asked whether the database contains certain specific information, and, after a week's examination of the database or of documents that allegedly describe the database, of having to reply that he or she does not know.

## *RS-2* Freedom from Positional Concepts

The DBMS protects the application programmers and terminal users from having to know any positional concepts in the database.

Examples of positional concepts follow:

- Where is a particular relation stored?
- Which row is next to a given row?
- Which column is next to a given column?

In dealing with databases that contain R-tables with thousands (sometimes millions) of rows, as well as hundreds of columns, it would place a heavy and unnecessary burden on users if the DBMS required them to remember which row or which column is next. User productivity would suffer seriously.

## *RS-3* Duplicate Rows Prohibited in Every Relation

The DBMS prohibits the occurrence of duplicate rows in any relation (whether base, view, or derived), and in this way protects the user from the subtle complexities and reduced optimizability that stem from permitting duplicate rows.

As mentioned in Chapter 1, for duplicate rows there is no precise, accepted definition that is context-independent. Consequently, there is no common interpretation that all users can share. Many of the present versions

of SQL fail to support Feature RS-3. The adverse consequences are discussed in some detail in Chapter 23. Note that Feature RS-3 refers to entire rows being duplicated. It does *not* prohibit the occurrence of duplicate values in any column.

### *RS-4* Information Portability

If a row of a base R-table is moved in any kind of storage by the DBMS, its information content as perceived by users remains unchanged, and therefore need not be changed. The entire information content of the DBMS as seen by users must not be dependent upon the site or equipment in which any of the data is located.

An example of such a move is the archiving of a portion of an R-table. Another example is the re-distribution to different sites of parts of a distributed database. Thus, any hashing of data done by the system must not be perceptible to users. Similarly, if the primary key value is ever system-generated, that value cannot be a pointer or an address, and cannot be location-dependent in any way.

For the time being, this feature is not intended to include the case of vendor-independent physical representation of data stored in or retrieved from databases, especially on communication lines. This topic should be treated by standards organizations as a matter needing urgent attention. This feature is also not intended to apply to performance-oriented objects such as indexes.

## 2.2.3 Three-Level Architecture

About 1976 the SPARC committee of the American National Standards Institute (ANSI) announced with great fanfare something called the "three-schema architecture." The definitions of the three levels supplied by the committee in a report were extremely imprecise, and therefore could be interpreted in numerous ways. Upon reflection, however, I believe the idea had been already conceived and published as part of the relational model and as part of the System R architecture [Chamberlin et al. 1981]. Of course, the definitions in the relational model and in System R were much more precise, but the terminology was different.

*Base relations* are those relations represented directly by stored data (not by formulas or relational commands). *Views* are virtual relations defined in terms of the base relations and possibly other views using the relational operators. It is these definitions (either by formulas or by relational commands) that are stored in the catalog. *Storage representation* is the representation in storage for the information contained in base relations.

With an appropriate interpretation of the ANSI definitions, the correspondence in terms is as follows:

| ANSI term | R-term |
|---|---|
| External schema | Views |
| Conceptual schema | Base relations |
| Internal schema | Storage representations |

### *RS-5* Three-Level Architecture

A relational DBMS has a three-level architecture consisting of views, base relations, and storage representations.

This feature is concerned with the structural aspect only. Full support of the three-level architecture includes full support of view-manipulative Features RV-4, RV-5, and RV-6. It also includes a systematic approach to view updatability, which is the subject of Chapter 17.

## 2.3 ■ Domains, Columns, and Keys

Let us turn our attention to the concepts upon which the relations of the relational model are built.

### *RS-6* Declaration of Domains as Extended Data Types

Each semantically distinct domain is distinctly named and declared separately from the R-table declarations (since it may be used in several R-tables). Each domain is declared as an extended data type, not as a mere basic data type.

For an explanation of the differences between basic data types and extended data types, see the introductory text in Chapter 3 (preceding Feature RT-1). Feature RC-3 states in detail the kind of domain declaration that should be stored in the catalog.

The concept of a domain is quite fundamental. It is essential in determining whether a given relational database can be split into two or more independent databases without loss of meaningful derivable information—or, to express it another way, without loss of inter-relatedness (see Chapter 3).

Many features of RM/V2, as well as the original RM/V1, depend on the

domain concept. Some of the advantages of supporting domains as extended data types are as follows:

- a large component of the description of every column that draws its values from a given domain need be declared only once in the domain declaration (this is called the *factoring advantage*);
- for every operator that involves comparing pairs of database values, the DBMS can ensure that each of the two components are semantically comparable by checking either at the start of the operation or (when possible) at compile time that both columns involved draw their values from a common domain (see Feature RM-14 in Chapter 12);
- integrity checks are facilitated (see Chapters 13 and 14, and Feature RD-7 in Chapter 21).

### *RS-7* Column Descriptions

For each column of each base R-table, there should be the capability of declaring (1) from which domain that column draws its values (thus identifying the extended data type) and (2) what additional constraints, if any, apply to values in that column.

When the DBMS fully supports the domain concept, it can detect errors resulting from users forgetting which columns have values of a given extended data type. Therefore, users can depend on the system to check whether the values in two given columns are semantically comparable (i.e., of the same extended data type).

### *RS-8* Primary Key for Each Base R-table

For each and every base R-table, the DBMS must require that one and only one primary key be declared. All of the values in this simple or composite column must be distinct from one another at all times. No value in any component column is allowed to be missing at any time.

Whether the primary key is indexed or not is a purely performance concern, and therefore a decision to be made by some authorized user quite separately (see Feature RD-3 in Chapter 21). The primary key is constrained by the DBMS to contain distinct primary key values in distinct rows, and (because of Feature RI-3 in Chapter 13) is not permitted to have missing values.

The constraints that values must be distinct and that missing values must be excluded may also, at the discretion of the DBA, be enforced on columns other than the primary key. Therefore, the mere existence of these constraints on some column or combination of columns does not identify which column(s) constitute the primary key.

A primary key may consist of a simple column or a combination of columns. When it consists of a combination of columns, the key is said to be *composite.* Each column participating in a composite primary key may be, but need not be, a foreign key.

## *RS-9* Primary Key for Certain Views

For each view the DBMS must support the declaration of a single *primary key* whenever the DBA observes that the definition of that view permits the existence of such a key, including adherence to the entity-integrity feature (see Feature RI-7 in Chapter 13). Where possible, the DBMS must check that a primary key declaration for a view is consistent with primary key declarations for the base R-tables.

Note that, in some cases, views can have instances of missing information *in every column* (although not all in one row, of course). In these cases, it is impossible to declare a primary key for the view and have it adhere to the entity-integrity feature. A simple example is a view that is a projection of a base R-table that does not include any column of the table from which missing values are prohibited (and thus does not include the primary key of that R-table). In these circumstances and in the view thus defined, use is made of the concept of a *weak identifier,* defined at the end of Section 5.3.

Generally, it is clearer to limit the specification of all kinds of integrity constraints to *base R-tables.* It is helpful to users, however, to have a primary key declared for each view whenever possible, because such a key plays a significant role as a unique identifier of objects during user interaction with a relational database. Further, it is reasonable to expect users to interact with views rather than with base R-tables, because in this way they enjoy more logical data independence. (See Feature RP-2 in Chapter 20.)

## *RS-10* Foreign Key

The DBMS permits the declaration of any column or combination of columns of a base R-table as a *foreign key* (where this is semantically applicable). Included in this declaration are the target primary keys (usually just one) for this foreign key. However, the DBMS must *not,* through its design, constrain the target to be just

one primary key for a given foreign key, even though the most frequently occurring case may be just one.

---

In supporting more than one primary key as the target for a given foreign key, the DBMS must *not* assume that the corresponding primary key values are partitioned into disjoint sets in distinct R-tables. Moreover, the DBMS must *not* deduce foreign keys and their targets from the declared primary keys and their domains. In the case of composite foreign keys, the DBMS could be in error because of the semantic nature of key targeting (i.e., the association of foreign keys with primary keys). An example of this semantic nature is described in Section 1.8.1.

## *RS-11* Composite Domains

A user-chosen combination of simple domains can be declared to have a name, providing that name is distinct from that of any other domain (simple or composite). The sequence in which the component domains are cited in this declaration is part of the meaning of the combination.

---

An example of a composite domain is the combination of two simple domains: the supplier serial number domain and the part serial number domain. Several composite columns in a database may draw their values from this composite domain.

This declaration of combinations of domains enables them to be treated as unit pieces of information without specifying their components (in other words, as if they were simple domains). For example, if an **equi-join** (see Chapter 4) is required that involves comparing one combination of columns with another, and if both combinations happened to be based on the same composite domain, the user would be able to gain confidence more rapidly concerning the correctness of the **join,** and the DBMS would be able to check this correctness more rapidly.

For each composite domain, of course, the sequence in which the domains are specified is a vital part of the definition. For any composite column based on a composite domain, the sequence in which the combination of columns is specified must match the sequence in which the simple domains are specified that participate in the composite domain.

## *RS-12* Composite Columns

A user-chosen combination of simple columns within a base R-table or view can be declared to have a name, providing that name is

distinct from that of any other column (simple or composite) within that R-table, and providing a composite domain has already been declared from which this composite column is to draw its values. The sequence in which the component columns are cited in this declaration is part of the meaning of the combination, and it must be identical to the sequence cited in the declaration of the corresponding composite domain.

---

An example of a frequently needed composite column is a postal address: the combination of an apartment or suite number in a building, the building number on some street, the street name, the city name within a state or country, and finally the state or country.

Note that, if a composite column is declared to contain every column of some R-table (admittedly a rare event), that combined column is not itself an R-table and should not be treated as if it were. In this case, the name of the R-table and the name of the composite column should be distinct.

The naming and declaration of combinations of columns enables them to be treated as unit pieces of information without specifying their components (in other words, as if they were simple columns). For example, if an **equi-join** (see Chapter 4) is required that involves comparing one combination of columns with another, and if these combinations happened to be declared and each named as such, it would be easier to express the comparison in terms of the combination names. For each composite column, of course, the sequence of simple domains corresponding to the sequence of components in each combination would have to be identical in order for the values to be comparable with one another.

Note that a composite column is restricted to combining simple columns within a single base R-table. This restriction should be interpreted in the sense that at present I am not taking a position either for or against the kind of composite columns that combine columns that may be simple or composite. Such columns are sometimes loosely referred to as "composites of composites." So far, I fail to see the practical need for them.

The name of the composite column can be the same as the name of the composite domain from which the composite column draws its values (barring ambiguity within the pertinent R-table). The name, however, should be distinct from that of any R-table (base or view), and it *must* be distinct from

- any simple column within the pertinent base R-table, and
- any other composite column already declared and still in effect for that R-table, and
- any word assigned special meaning within the relational language (i.e., any keyword of RL).

To explain the meaning of a comparator such as LESS THAN (<) applied to a pair of composite columns, suppose that the composite column C, consisting of C1, then C2, then C3, is to be compared with a composite column D, consisting of D1, then D2, then D3. The condition C < D is equivalent to making a sequence of tests:

C1 < D1, then C2 < D2, then C3 < D3.

The first test that fails causes the whole test to fail for the truth-valued expression C < D and that application of it. If duplicate values occur in some or all of these columns, there is no guarantee that a request for an ordering based on them will deliver exactly the same sequence of tuples in repeated execution of the request. To obtain precise repeatability, requests for ordering should be based on columns, which contain values that are distinct from one another.

The treatment is tantamount to C1 and D1 being treated as high-order, C2 and D2 as middle-order, and C3 and D3 as low-order in the usual arithmetic sense (whether the operands are numeric or character strings). The LESS THAN comparator applied to a pair of character strings makes use of a collating sequence established as a standard for database management and for data processing.

## 2.4 ■ Miscellaneous Features

### *RS-13* Missing Information: Representation

Throughout the database, the fact that a database value is missing is represented in a uniform and systematic way, independent of the data type of the missing database value. Marks are used for this purpose. (Note that RS-13 is the structural part of Rule 3 in the 1985 set; see Feature RM-11 in Chapter 12 for the manipulative part.)

The semantics of the fact that a database value is missing are quite different from the semantics of the value that is missing. See Chapters 8 and 9 or [Codd 1986a and 1987c] for more details. Marks were previously called nulls, and occasionally null values—which is even more misleading because it suggests that nulls behave just like other database values. To be independent of data type, they must be distinguishable from all database values of all types. Thus, any value whose bit representation lies within the bit boundaries of a database value is unacceptable in the role of representing the fact that a database value is missing. For example, if the following are stored in the same R-column as the corresponding non-missing database

values, they are unacceptable as representations of missing values (whether built-in, default, DBA-declared, or user-declared):

- the empty character string,
- a string of blank characters or any other character string,
- zero or any other number,
- any string of bits.

An example of a conforming representation for missing information is that adopted in IBM's DB2. In this system, any column in which database values are permitted to be missing is assigned an extra byte outside the bit boundaries of the database values. This byte is reserved to indicate to the DBMS whether the corresponding value, which is represented by the bits within the bit boundaries of the database values in this column, is to be taken seriously as an actual value or as a fictitious value left over from some previous use.

To support database integrity, the DBMS should deduce from the primary key declaration that marks are not allowed for that column or combination of columns. It must be possible, however, for the DBA or some other suitably authorized user to specify "marks not allowed" for any other columns for which this happens to be an appropriate integrity constraint. An example would be certain foreign key columns.

Note that the "marks not allowed" declaration is *not* an alternative to an explicit declaration of the primary key. By itself, such a declaration is inadequate to distinguish primary key columns from other columns.

Techniques in database management before the relational approach required users to reserve a special value peculiar to each column or field to represent missing information. This would be most unsystematic in a relational database because users would have to employ different techniques for each column or for each domain. This is a difficult task because of the high level of language in use, and one that would lower the productivity of users significantly.

## *RS-14* Avoiding the Universal Relation

Neither the collection of all base relations nor the collection of all views should be cast by the DBMS in the form of a single "universal relation" (in the Stanford University sense) [Vardi 1988]. The DBA, however, should have the option of creating such a relation as one of the many views.

Generally, the database should be perceived as a collection of base relations and a collection of virtual relations (views), all of assorted degrees.

If all of the base relations or all of the views are each cast as a single universal relation (in the Stanford University sense), there are at least three adverse consequences:

1. waste of space (disk and memory) due to the large number of values that must be marked "property inapplicable," and waste of channel time for the same reason;
2. loss of adaptability to changes in the kinds of information stored in the database:
   a. more human effort is required when the counterpart of a new base R-table or view is defined,
   b. application programs will be adversely affected because their logic is not immune to the restructuring of the "universal relation," and
   c. more of a reorganizing load at the storage level is likely to be involved;
3. the need for joins on non-key values is not decreased, and their counterpart is more difficult for the user to request, because the key-based **join** built into the "universal relation" must be decomposed before the **join** based on non-keys is constructed.

In the relational model a database is treated as a collection of relations of assorted degrees. The Stanford University research on the universal relation concept was, I believe, motivated by the desire to eliminate the need for joins. In the universal relation approach, however, the combining of several relations (perhaps many) into a single relation is necessarily based on only one method of combination, and the method normally selected is by key-based equi-joins.

One of the great advantages of the relational approach is that it supports joins of all kinds, whether based on keys or not. It is quite likely that a database that adheres to the relational model has more distinct kinds of non-key-based joins than key-based ones. The approach adopted in the relational model is far more flexible and more adaptable to change.

When new kinds of information are introduced into a database, one can merely define new domains, new columns, and new R-tables as necessary. On the other hand, using the Stanford approach, this new information would have to be carefully fitted into the existing universal relation, which is a much more complicated problem. For more detail, see Chapter 30.

## Exercises

2.1 Does RM/V2 permit any information to be carried solely in the ordering of rows in base relations? State reasons why or why not, whichever is applicable.

2.2 What is the precise definition of a domain in the relational model? Which of the following statements is always true, which always false, and which can be true in some cases and false in others?

1. A domain determines the type and range of values that may occur in one or more columns.
2. A domain determines the type and range of values that may occur in exactly one column.
3. Exactly one column determines the type and range of values that may occur in a domain.
4. One or more columns determine the type and range of values that may occur in a domain.

2.3 How should domains be supported? What is wrong with storing for each and every domain all of the values that belong to it? **(10, 17)**

2.4 IBM's DB2 supports the language SQL for database interrogation, manipulation, and control. Should the declaration of a primary key for each base relation be optional, as in the SQL of Version 2 of IBM's DB2? Give reasons for your answer.

2.5 Define the candidate-key concept. Define the primary-key concept.

2.6 Why does the primary-key concept preclude there being two or more primary keys in a base table? Explain the problems that result from having two or more primary keys per relation.

2.7 Why is a single primary key mandatory for each base relation? Does this requirement reduce the range of applicability of the relational model?

2.8 Does the fact that a column has been declared to be the primary key of some base relation mean that the column must be indexed? How about the reverse implication?

2.9 Does RM/V2 *require* a relation to be stored as (1) a table, (2) an array, (3) a flat file, or (4) a collection of records connected by pointers? (See also Chapter 20.)

2.10 Does RM/V2 *require* a relation to be stored row by row? Does RM/V2 *require* the components to be ordered within each row in storage in the same sequence as the columns are declared in the catalog? (See also Chapter 20.)

2.11 What does it mean for a value to be atomic with respect to the DBMS? Is not an atomic value always atomic?

2.12 Supply two reasons why the DBA should always control the introduction of new types of values into the database to ensure that these values are atomic in meaning as well as atomic with respect to the DBMS.

■ *CHAPTER 3* ■

# Domains as Extended Data Types

## 3.1 ■ Basic and Extended Data Types

The concept of domains has played a very important role in the relational model since the model was conceived. It is not overstating the case to say that the domain concept is fundamental. It participates crucially in the definition of numerous features of RM/V1 and RM/V2. *Consequently, many of these features cannot be fully supported by a DBMS unless that DBMS supports domains.* Omission of support for domains is the most serious deficiency in today's relational DBMS products.

In my first two papers on the relational model [Codd 1969 and 1970], domains and columns were inadequately distinguished. In subsequent papers (e.g., Codd 1971a, 1971b, and 1974a), I realized the need to make this distinction, and introduced *domains* as declared data types, and *attributes* (now often called *columns*) as declared specific uses of domains. It has become clear that domains as data types go beyond what is normally understood by data types in today's programming languages. Consequently, as noted in Chapter 2, when domains are viewed as data types, I now refer to them as *extended data types.* With regard to the data types found in programming languages (excluding PASCAL and ADA), I refer to them as *basic data types.*

An extended data type is intended to capture some of the meaning of the data. It is conceptually similar to a basic data type found in many programming languages. If, however, two semantically distinguishable types of real-world objects or properties happen to be represented by values of

the same basic data type, the user nevertheless assigns distinct names to these types and *the system keeps track of their type distinction.*

The description of an extended data type includes its basic data type together with information concerning the range of values permitted, and whether the less than comparator (<) is *meaningfully applicable* to its values. The distinction between extended data type and basic data type is *not* that the first is user-defined and the second is built into the system, even though many of the extended data types will, in practice, be user-defined.

Note that, unless it is built into the system, a domain, and hence an extended data type, must be declared as an object itself before any use can be made of it. Whenever an extended data type is built into the system, its name as an object must be available to users.

In contrast, a basic data type is normally a property associated with an object at the time of the declaration of that object. As an aside, given the present state of the data sublanguages SQL and QUEL, a CREATE DOMAIN command must be added to each language. Table 3.1 lists some of the distinctions between basic and extended data types.

By now it should be clear that it is quite incorrect to equate either (1) basic data types with built-in data types, or (2) extended data types with user-defined data types.

The distinction between built-in data types and user-defined data types is a purely temporary consideration based largely on the kind of hardware that is economically available; this boundary moves at least every decade, possibly more frequently. On the other hand, the distinction between basic and extended data types is both non-temporary and conceptual in nature: it is closely related to the question of levels of abstraction.

In apparent opposition to the first row of Table 3.1, it has been contended that both basic data types and extended data types have names. In a sense this is true. The "names" of basic data types, however, can be used only in designating certain properties of data (e.g., in a program). On the other hand, the names of extended data types can be used not only in designating properties of data, but also as objects themselves, when the user

***Table 3.1*** **Basic versus Extended Data Types**

| Basic Data Type | Extended Data Type |
|---|---|
| No object-oriented name | Object-oriented name |
| A property of an object | An object |
| Not independently declarable | Independently declarable |
| Range of values is *not* specifiable | Range of values is specifiable |
| Applicability of <, > is *not* specifiable | Applicability of <, > is specifiable |

Two database values with the same basic data type need not have the same extended data type.

wishes to interrogate them or modify them either interactively from a terminal or by using a program. This holds true of extended data types because their names and descriptions are stored as data in the catalog.

## 3.2 ■ Nine Practical Reasons for Supporting Domains

Full support for many of the features of the relational model depends on full support of the domain concept. Some of the advantages of supporting domains fully follow.

*First,* full support of the domain concept is the single most important concept in determining whether a given relational database is integrated. Consider the consequences of alleging that a relational database viewed as a collection CD of domains and a collection CR of relations could be split into two databases, without any loss of information or of retrieval capability. How would one check whether this assertion were true?

One way of solving this problem is to look for a subset cd of the domains CD and a subset cr of the relations CR with the following two properties:

1. the relations in cr make use of domains in cd only (no other domains);
2. the relations in ( CR - cr ) make use of domains in ( CD - cd ) only (no other domains).

Note that in Property 1 a relation makes use of a domain if at least one of its columns draws its values from that domain. In Property 2, the symbol "-" in ( CR - cr ) and ( CD - cd ) denotes set difference.

When both these properties hold, the relational operators will not permit the derivation of any relations that include information from cr as well as information from ( CR - cr ), whether these two collections are regarded by the DBA or by anyone else as a single database or as two databases. There is a sound reason for this related to (1) value-comparisons that make sense and (2) value-comparisons that do not. (See the third practical reason for supporting domains later in this section.)

This first reason for supporting domains can be concisely stated as follows: *domains are the glue that holds a relational database together.* Notice that I said *domains,* not primary keys and foreign keys. The concept of keys in the relational model provides an important additional and specialized kind of glue.

*Second,* support of domains is necessary if the factoring advantage is to be realized in declaring the types of data permitted in columns.

A large component of the description of *every column* that is defined on a given domain need be declared only once for that domain. As an example, consider a financial database that has 50 columns containing currency of some type (e.g., all U.S. dollars).

Using pre-relational DBMS, it was usually necessary to store 50 declarations, one for each column; this task was often left to numerous application

programmers, who inserted these declarations into their programs. With this approach, the usual result was that no two currency declarations were in precise agreement, placing an immense and unnecessary burden on the DBA and on the community of users.

It was this phenomenon that spurred the development of add-on packages called *dictionaries*. Control over names and declarations, however, should be handled by the DBMS itself, making it much more difficult for any user to circumvent such control.

Using the relational approach, just one declaration of semantic data type for the currency domain will suffice for all 50 currency columns. Then, for any currency column that needs tighter control (an interval of permitted values more narrow than that declared for the domain), an extra range constraint can be declared for that column (type C or column integrity). All domain declarations and all additional constraints applicable to specified columns are stored by the DBA in the catalog, which is where they belong if users are to receive the important benefit of not having to change application programs whenever integrity constraints are changed.

*Third,* support of domains is necessary if domain integrity is to be supported. Domain integrity consists of those integrity constraints that are shared by all the columns that draw their values from that domain. Three kinds of domain integrity constraints that are frequently encountered are (1) regular data type, (2) ranges of values permitted, and (3) whether or not the ordering comparators greater than (>) and less than (<) are applicable to those values.

*Fourth,* full support of domains includes domain-constrained operators and domain-constrained features of other kinds to protect users from costly blunders. For every operator that involves comparing pairs of database values, the DBMS ensures that the two components to be compared are semantically comparable. It does this by checking either at the start of the operation or (when possible) at compile time that both columns involved are defined on a common domain. This constraint is supported by a relational DBMS to help protect users from incorrectly formulating commands such as **joins** (for example, a **join** in which quantities of parts are being compared with quantities of people). If a special need arises "to compare apples with oranges," with special authorization—and a DBA would grant such authorization rarely and for only short intervals—a user may employ the DOMAIN CHECK OVERRIDE qualifier in his or her command.

## *RT-1* Safety Feature when Comparing Database Values

When comparing a database value in one column with a database value in another, the DBMS checks that the two columns draw their values from a common domain, unless the domain check is overrid-

den (see Feature RQ-9 in Chapter 10). When comparing (1) a computed value with a database value or (2) one computed value with another computed value, however, the DBMS checks that the basic data types are the same.

---

The main reason for checking the weaker basic data types when computed values are involved is that the computing is likely to be expressed in some host-programming language. Most such languages do not yet support the extended data types of the relational model.

The following operators and features of the relational model require implementation of domains to ensure that data is handled properly by the DBMS and by the user:

- Operators for retrieval and modifying:
  - **theta-select** (whenever two components of a tuple are being compared),
  - **theta-join,**
  - **t-join,**
  - **relational division,**
  - **relational union, intersection,** and **difference,**
  - all of the outer operators **(outer join, outer union, outer intersection, outer difference),**
  - **primary key update.**
- Integrity features:
  - referential integrity (type R),
  - other inclusion dependencies,
  - user-defined integrity constraints (type U) involving any of the operators just listed,
  - every integrity constraint involving cascading the action to all equal values in all columns defined on the same domain (e.g., **cascade delete, cascade update, cascade insert).**

*Fifth,* in the highly dynamic environment supported by a fully relational DBMS, it is necessary to support domains in order to support transactions that single out all occurrences of some value as a value of a specified extended data type. Consider an example: business activity with supplier s3 has been terminated in a completed state (no shipments are still due from this supplier and all bills have been paid). A company executive requests that all rows of a certain kind (wherever they occur in the database) be archived—specifically, all rows of all relations that contain s3 as a supplier serial number.

In executing this action, it is important to avoid archiving rows that happen to contain s3 as something other than a supplier serial number (say, a part serial number), if these rows do not also contain s3 as a supplier serial number. It is inadvisable to expect any application programmer to remember the information as to which columns draw their values from the supplier serial number domain: this information may change even while the programmer is thinking about the transaction because of activities by other users and programmers. A relational DBMS supports a great variety of users engaged concurrently in actions conceived independently of one another. Most of these actions may be simple changes to the data, but some may be changes in the database description. Thus, while the application programmer is thinking about the transaction needed to archive all rows that contain s3 as a supplier serial number, some other user may be introducing a new relation or adding a new column containing supplier serial numbers. That user or another one may then insert s3 in such a column.

Consequently, it is essential that the DBMS retain in an extremely up-to-date state the information as to which columns draw their values from the supplier serial number domain. It is also essential that the relational language contain a command that is capable of referring to all columns currently drawing their values from a specified domain *without the user having to list these columns within the command or elsewhere.* At present, SQL and its dialects lack such a command.

*Sixth,* support of domains facilitates certain user-defined integrity checks by the DBMS. An example is an inclusion constraint, in which the values appearing in one column C1 (simple or composite) are required to be a subset of the values appearing in another column C2. In this case, the relational model requires that C1 and C2 draw their values from a common domain.

*Seventh,* the domain concept participates in many definitions in the relational model, including the definitions of primary domain, foreign key, all value-comparing operators (as noted earlier in the third reason), union compatibility, referential integrity, and inclusion constraints.

*Eighth,* domains can be used by the DBMS to establish the extent of naming correspondence needed from the user when a **union** or similar relational operator is requested. When forming R UNION S in the relational model, it is required not only that the degree of R is equal to the degree of S, but also that there exist at least one mapping (one-to-one) of the columns of R onto the columns of S, such that the two columns of each pair belonging to the mapping have a common domain.

Clearly, if all the domains of columns of R are distinct, then all the domains of S must also be distinct, and the domains can be used by the DBMS to determine the mapping of columns on columns completely. Thus, in this case, the user need not become involved at all in the pair of column names.

If two or more domains of columns of R are identical, however, the same must be true of S, and the user is faced with alternatives. He or she must therefore be involved in column correspondence between R and S to the extent of having to specify a mapping for just those columns of R that share a common domain and just those columns of S that share a common domain. Obviously, determining this correspondence can also affect the naming of columns of the result.

*Ninth,* and last, it is necessary to support domains in order to support an important performance-oriented tool, namely domain-based indexes. While this tool is not itself part of the relational model, it is important because it can cause certain types of relational DBMS (specifically those that make use of indexes under the covers) to perform competitively with non-relational DBMS.

A domain-based index is a single index on the combination of all the columns that draw their values from the specified domain. Such an index is usually a multi-relation index. However, not every multi-relation index is a domain-based index. It provides immediate direction for the DBMS to find all occurrences of each currently active value from that domain.

Once such an index has been declared by the DBA, the introduction into the database of a new column drawing its values from the pertinent domain causes automatic expansion by the DBMS of the domain-based index. Similarly, whenever the DBA requests the dropping of a column that is referenced by a domain-based index, the DBMS automatically removes references to this column from the pertinent index.

It is worth noting that domain-based indexes can do more than improve the performance of certain kinds of DBMS in the execution of **joins** and other value-comparing operations. Also, when applied to *primary domains* (domains from which primary keys draw their values), they can improve the performance of tests of referential integrity.

One extreme case may be interesting, although I am not advocating it as a preferred approach: the case in which the entire collection of values in the database is stored in domain-based indexes only. Of course, in this case such indexes cannot be dropped without losing information from the database.

## 3.3 ■ RM/V2 Features in the Extended Data Type Class

### 3.3.1 General Features

#### *RT-2* Extended Data Types Built into the System

The DBMS supports calendar dates, clock times, and decimal currency as extended data types, including the various kinds of dates

and time units in common use, the computation of date intervals and time intervals, and the use of partial as well as full dates (see Features RT-3–RT-9, following). The DBMS must have access to the date of the current day and the time of day at all times.

---

A justification for this feature is that very few institutions, whether commercial, industrial, or governmental, can manage themselves without these types of data. Incidentally, each built-in extended data type must have a name that is usable in the catalog for appending integrity constraints of type D to the data type.

### *RT-3* User-defined Extended Data Types

The DBMS permits suitably authorized users to define extended data types other than those for which it provides built-in support in accordance with Feature RT-2. These data types can be used to enrich the retrieval-targeting and retrieval-conditioning capabilities of the principal relational language (e.g., with respect to text manipulation and computer-aided development/computer-assisted manufacturing).

---

## 3.3.2 Calendar Dates and Clock Times

As described in Features RT-4–7, the treatment of dates and clock times (separately and in combination) in the relational model is intended to be flexible enough to be suitable for managing databases of any of the following types, among others:

- genealogical types at the headquarters of the Mormon church in Salt Lake City, Utah;
- air-traffic control at major air-traffic-control centers of the Federal Aviation Administration;
- commercial databases used by institutions operating in a variety of different time zones (possibly many time zones).

### *RT-4* Calendar Dates

From the user's standpoint, dates appear to be treated by the DBMS as if they were atomic values. However, the DBMS supports functions that are capable of treating as separate components the year, month, and day of the month.

---

The services provided by Feature RT-4 include the 14 that follow:

4.1. Independence of date and time from particular time zones in which users are located, by use of Greenwich dates and Greenwich mean time.

4.2. The function called NOW yields for any site the current date and time that are in effect in the time zone of the site.

4.3. Extraction of any one or any pair of the three components, a form of *truncation.*

4.4. Extraction with *rounding* of either year alone or year followed by month.

4.5. Conversion of the combination year, month, day of the month to the year followed by day of the year, as well as conversion in the opposite direction.

4.6. Computation of the difference between two dates of similar or distinct external types, where each argument is expressed as
   a. years only, or
   b. years and months, or
   c. years, months, and days of the month, or
   d. years, and days of the year.

   These four options must be available to users, and *the result must be of the same external type as the argument that is coarser.*

4.7. Conversion of date intervals into years only or months only or days only, using truncation or rounding as specified, if the conversion is from fine units to coarser units.

4.8. Arithmetic on dates, including computation of a date from a given date plus or minus a date interval, without the adoption of dates and date intervals as distinct data types.

4.9. Pairwise comparison of dates, including testing of pairs of dates to see which is the more recent and which is the less recent.

4.10. Finding the most recent date of a collection.

4.11. Finding the least recent date of a collection.

4.12. All varieties of **joins** based on comparing dates.

4.13. The ability to report dates in at least one of the following formats:
   a. European format: D,M,Y;
   b. North American format: M,D,Y;
   c. computer format: Y,M,D;
   d. in the Indian calendar with lunar months.

4.14. Two types of date-conversion functions:
   a. DATE_IN for transforming dates from external representation of dates to the internal representation;

b. DATE_OUT for transforming dates in the reverse direction, with the DBA having the option of putting into effect functions defined and specified by the DBA either for all users of a given DBMS or for specified classes of users (instead of or in addition to those supplied by the DBMS vendor).

This option is needed because users with different responsibilities and those located in different countries (even within a single country) may employ different kinds of dates externally with respect to the DBMS. Table 3.2 shows an example of two distinct types of conversions that may be needed in countries that use the Gregorian calendar.

Of course, the DBMS must know how many days there are in each of the calendar months, and which years are leap years. The DBMS should also have a standard internal representation of dates oriented toward arithmetic on dates. The preferred representation is a date origin established by the DBA (such as the first day of the year 1900), coupled with the number of days following that day. This representation is compatible with the handling of arithmetic operations upon dates (while abiding by all the usual laws of arithmetic) and of comparisons between pairs of dates (such as LESS THAN). Such a standard would ease the problem of communication between heterogeneous DBMS products. Finally, the DBMS must know the date of the current day.

Note that the DBA may impose bounds, both lower and upper, on acceptable values of dates using type D and type C integrity constraints. Dates occur in the numerous examples included in this book. The standard representation adopted is the computer format Y,M,D (year, month, day) cited earlier in the discussion of Feature RT-4.13.

## *RT-5* Clock Times

From the user's standpoint, clock times appear to be treated by the DBMS as if they were atomic values. The DBMS however, supports

***Table 3.2*** **Examples of Date Conversions**

| | | Two examples | |
|---|---|---|---|
| | **Type of Date** | **Any Year** | **Non-Leap Year** |
| | Internal date | September 31 | February 29 |
| DOWN → | Closest earlier date | September 30 | February 28 |
| UP → | Closest later date | October 1 | March 1 |

functions that are capable of treating as separate components the hours, minutes of the hour, and seconds of the minute. The services provided include counterparts to the first 12 of the 14 services listed in the discussion of RT-4.

---

Whenever it is necessary to store fractions of a second (e.g., milliseconds or microseconds) in one or more columns of the database, a user-defined extended data type should be established for each distinct and pertinent unit of time.

### *RT-6* Coupling of Dates with Times

The DBMS supports a composite data type consisting of the data type DATE coupled with the data type TIME, allowing the functions applicable to dates alone or times alone to be applied to combinations in which DATE plays the role of the high-order part and TIME the low-order part.

---

### *RT-7* Time-zone Conversion

The DBMS supports (1) the conversion of every date-time pair from any specified time zone to Greenwich date and Greenwich mean time, and (2) the inverse conversion of Greenwich date-time pairs back into a specified time zone.

---

## 3.3.3 Extended Data Types for Currency

### *RT-8* Non-negative Decimal Currency

The DBMS supports non-negative decimal currency as a built-in extended data type, but does not necessarily support automatic conversion between currencies of different countries. The basic data type is non-negative integers.

---

This extended data type permits the DBMS to distinguish those non-negative integers representing currency from those that represent anything else (e.g., numbers of people). The basic data type represents the monetary amount expressed in terms of the smallest monetary unit in the currency (cents in U.S. currency, new pence in British sterling).

Monetary amounts can be expressed in larger units in selected columns. For example, there may be a need to express such amounts in units of one thousand dollars or one million dollars. If the DBMS does not support conversion functions to convert a monetary amount expressed in small units into much larger units (including either rounding up or truncation according to the user's request), a user-defined extended data type will be necessary for each column that employs units different from the smallest.

If an attempt is made to introduce a negative number into any column drawing its values from the non-negative currency domain, the DBMS rejects the request with an explanatory error message.

## *RT-9* Free Decimal Currency

The DBMS supports decimal currency (in which values may be negative, zero, or positive) as an extended data type. The basic data type is integer.

The DBMS does not necessarily support automatic conversion between currencies of different countries. This requires user-defined functions, together with a relation of degree of at least three, containing the current conversion rates.

With some databases, the DBA may choose to use just one of the two built-in extended data types for currency, namely the free version, because of the need to compare currency values involving pairs of currency columns, for which one column is not permitted to have negative values and the other can accept both negative and positive values. If such comparisons are routine, the DBA may with good reason decide that the pairs of columns involved should be declared to have a common domain. Whenever the DBA makes this choice, the prohibition of negative values in certain currency columns be expressed as an extra constraint of type C for those columns only (not for the domain).

Two additional extended data types (Features RF-9 and RF-10) built into the DBMS are described in this chapter and in Chapter 19. These types pertain to the names of functions and the names of arguments of functions. They make the relational model easier to interface with activities that really do not belong in the model. Their definitions are given here for ease of reference, but are expressed differently.

## *RF-9* Domains and Columns Containing Names of Functions

The DBMS supports names of functions as an extended data type. The DBMS can use any one of these names to (1) retrieve the

corresponding code for the function, (2) formulate a call for this function as a character string, and (3) execute the string as an invocation of the function.

---

Note that, prior to execution, the names of appropriate arguments must be plugged into the character string that represents the invocation.

### *RF-10* Domains and Columns Containing Names of Arguments

The DBMS supports names of arguments of functions as an extended data type. These arguments can be variables used in a host-language program. The names are character strings that can be incorporated with the name of a function to formulate a call for this function to be applied to the pertinent arguments. The names can also be incorporated in a source program expressed in RL, in the HL, or in both.

---

## 3.4 ■ The FIND Commands

In the next two chapters the basic operators and the advanced operators of RM/V2 are described. Each of these operators transforms either one or two relations into a relation. None of them deals with the entire database, which may contain any number of relations.

The FIND commands presented here are different in type, because:

1. each has an operand that consists of all of the columns in the database that draw their values from a specified domain; and
2. although the result is a relation, at least two of the columns are concerned with database descriptive values.

The FIND commands are also intended for the DBA and his or her staff. That is why as features of RM/V2 they are labelled RE-1 and RE-2 (see Chapter 7).

In the relational model what does it mean to find something? Disk addresses and memory addresses are concepts that do not belong to the model. However, each occurrence of every atomic value in a relational database has a uniquely identified "location." The identification of this location is the combination of a relation name, a primary key value, and a column name. The first two of these identify a row uniquely with respect to the entire database, while the third identifies a component of that row. Now it is possible to introduce the FIND commands one by one.

Normally the FIND commands are limited to finding character strings, because finding numeric values or the truth values of logic is not usually an interesting thing to do.

### 3.4.1 The FAO__AV Command

---

**_RE-1_ The FAO__AV Command**

This command is intended to find all occurrences of all active values drawn from a specified domain. Note that for any domain (with the possible exception of those domains that have very few legal values) it is highly unlikely that all of its values are active (i.e., exist in the database) at any one time.

---

Suppose the DBA believes that certain city names in the database are incorrectly spelled, and he wants to check all of them either by reading a list of all of them or by means of a spelling checker program along with a dictionary of city names. Then, the following query would be appropriate:

Example: find all occurrences of all city names that exist *anywhere* in the database.

If the domain of city names happens to be called CITY, and if the resulting relation is to be called CITY__DOM, this query could be expressed as: CITY__DOM← FAO__AV domain CITY.

The result is a relation CITY__DOM (RELNAME COLNAME PK VALUE) where the RELNAME and COLNAME columns are always simple columns, (degree one) the VALUE column is either simple or composite depending on the example, and the PK column is usually a composite column of degree n (consisting of n simple columns), where n is adequate to hold values for the primary key of highest degree in the entire database. Note that, whenever the value of a primary key of degree p is inserted in the result and $p < n$, the excess components in that row (there are n-p of them) are marked as inapplicable (see chapter 8).

To find all these city names the DBMS takes the first step of searching the catalog to find a column in some base relation that draws its values from the domain CITY. Suppose it finds that column C1 of relation R1 is such a column, and that R1 has N1 rows. It then copies the value part of the primary key column of R1 (possibly composite) along with the value part of column C1 into the relation CITY__DOM being developed. To this partial relation it appends a pair of columns containing the name of the relation R1 and the name of the column C1 repeated N1 times. It names these two columns RELNAME AND COLNAME.

The DBMS then takes the second step searching the catalog again and finds another column C2 of a relation R2 that draws its values from the

domain CITY. If R1 happens to have two such columns, then R1 and R2 are the same relation. However, the pair R1 and C1 cannot be identical to the pair R2 and C2. It copies the value part of the primary key column of R2 (possibly composite) along with the value part of column C2.

This process continues step by step until the last column in the database based on the CITY domain has been treated. Now a spelling checker program along with a dictionary of city names can be used as a utility to check the spelling of each and every occurrence of the city names as listed in a column of the relation CITY_DOM.

This FIND command (FAO_AV) can be applied not only to a simple domain, but also to a composite domain: for example, the domain CITY-STATE. Then the VALUE column is also composite (in this case, degree 2), since it has to hold pairs of simple values, where each pair consists of a city name together with a state name.

### 3.4.2 The FAO_LIST Command

---

**_RE-2_ The FAO_LIST Command**

This command is intended to find all occurrences in the database of each of the currently active values in a given list of distinct values, all of which are drawn from one domain, and that domain is specified in the command.

---

Suppose the DBA wishes to inquire whether any of ten city names occur in the database, and if so where they occur.

Example: find all occurrences in the database of each of the eight city names in a given list L drawn from the domain CITY (where L is a unary relation).

Suppose that the resulting relation is to be called OCC. Then, this command could be expressed as:

OCC ← FAO_LIST L domain CITY.

The result is a relation OCC (RELNAME COLNAME PK VALUE), where PK is usually a composite column consisting of n simple columns and n is adequate for the primary key of highest degree in the entire database. Note that, whenever the value of a primary key of degree p is inserted in the result and $p < n$, the excess components in that row (there are n-p of them) are marked as inapplicable (see chapter 8).

Incidentally, when the FAO_LIST command is executed, it is possible that the resulting relation OCC is entirely empty. This means that each and every one of the eight city names does not occur at all in the database. Of

course, the list L could have contained any strictly positive number of city names.

Just like the FIND command introduced in section 3.4.1, this FIND command (FAO_LIST) can be applied not only to a simple domain, but also to a composite domain: for example, the CITY-STATE domain. Then the VALUE column is also composite (in this case, degree 2), since it has to hold pairs of simple values, where each pair consists of a city name together with a state name.

Another feature of this FIND command is that a qualifier called SUBSTRING may be attached to it, and this indicates that the list of values is actually a list of substrings that the DBMS must search for in all columns of the database that draw their values from the specified domain. Whenever a value is found in such a column that has as a substring one of the strings in the given list, then that value and its location in the database are included as a row in the resulting relation.

In both types of FIND command, if a domain is specified that the DBMS discovers does not exist in the catalog, it turns on the DOMAIN NOT DECLARED indicator (see chapter 11), and the relation that is generated is empty.

Frequently, people get confused about the meaning of the following two questions, although they are quite different. The first question is Q1: "Is v a legitimate value for currency in this database?" The second question is Q2: "Is v in the database at this time as a currency value?"

To check Q1 the user simply examines the declaration pertaining to currency for this database, and of course this declaration is stored in the catalog. Then he or she examines the value v to determine if its data type is within the scope of the currency data type. If and only if it is, v is a legitimate value for currency in the database.

Before checking Q2 let us assume that Q1 has been answered affirmatively. Then one can (1) establish v as a single member of a unary relation L; and (2) issue the request FAO_LIST L CURRENCY. This will find all occurrences of v in the entire database as a value drawn from the currency domain. If the result is an empty relation, the proper conclusion is that v is admissible as a currency value, but v does not occur in this role in this database at this time.

Of course, if Q1 is answered negatively, then the proper answer for Q2 is that v does not occur in the database as a value of currency, but it is also inadmissible in this role.

## Exercises

3.1 Give at least eight reasons why domains should be fully supported in a DBMS. List at least 10 features of RM/V2, other than Feature RS-5, that a DBMS will be unable to support if it does not support domains as extended data types.

3.2 Your DBA makes the whole of the catalog available to you for reading. How would you determine where there exist parts of the database that are not inter-relatable?

3.3 A critic has stated that basic data types and extended data types are really built-in and user-defined, respectively. Supply the pertinent definitions and defend whatever position you take on this subject.

3.4 According to Feature RT-1, which extended data types are required to be built into the DBMS?

3.5 List 10 of the 14 requirements for full support of calendar dates.

3.6 Assume that you have currency values expressed as follows:

- in U.S. dollars in column C of R-table S,
- in British sterling in column D of R-table T.

By introducing a currency-conversion relation, develop a method of supporting automatic conversion to pounds sterling whenever a **join** between S and T is executed using columns C and D as comparand columns.

## ■ CHAPTER 4 ■

# The Basic Operators

The basic operators are intended to enable any user to retrieve information from all parts of the database in a very flexible and powerful way, but without requiring him or her to be familiar with programming details. First-order predicate logic is a standard against which such power can be measured. (See the listing under "Texts Dealing with Predicate Logic" in the reference section.) It has been proved [Codd 1971d, Klug 1982] that, collectively, these operators have the same expressive power as first-order predicate logic.

This logic is the standard adopted by the relational model. The operators are not intended to be directly incorporated into a relational language. Instead, a language based more directly on first-order predicate logic, such as ALPHA [Codd 1971a], is more capable of supporting better performance, because in use its statements are more likely to be optimizable to a greater degree.

The operators of the relational model transform either a single relation or a pair of relations into a result that is a relation. The operators are designed to be able to express a class of queries that, if expressed in terms of a logic, would require the power of at least four-valued, first-order predicate logic. Such a logic is more powerful than any supported by pre-relational database management systems.

The main reason for insisting on operators that yield relations from relations is that this form of operational closure permits an interrogator to conceive his or her ongoing sequence of queries based on the information gleaned to date. In this "detective" mode, it is essential that any result produced so far in the activities be capable of being used as an operand in

later activities. This operational closure is similar to the operational closure in arithmetic: every arithmetic operator acting upon numbers yields numbers (except for the case of dividing by zero). It would be next to impossible to handle accounting were it not for the operational closure in integer arithmetic. In a few decades, I predict, we shall comment similarly on the virtual impossibility of managing databases if the operational closure property in relational DBMS were abandoned.

In this chapter we adopt the algebraic approach to explaining how a relational language works, for two reasons:

1. upon first encounter, that approach appears easier to understand;
2. it is much easier to explain integrity constraints, the authorization mechanism, and the view-updatability algorithms described in Chapter 17 using the relational algebra than in terms of predicate logic.

It should not be assumed, however, that an algebraic approach is to be preferred over a logic-based approach when it comes to designing a relational language, even though both approaches are at the same level of abstraction. Quite the contrary—a logic-based approach encourages users who have complicated queries to express each query in a single command, whereas an algebraic approach seems to encourage users to split their queries into several commands per query. The optimizers of existing relational DBMS products are unable to optimize more than one command at a time. Therefore, they accomplish more if more activity is packed into each single command. The net result is that improved performance can be obtained with a logic-based approach over that achievable with an algebraic approach.

In this connection, it is important to remember that very few optimizers in the compilers for programming languages, and even fewer optimizers in relational DBMS, attempt to optimize across more than a single command. Otherwise, the optimizers would have to engage in flow tracing—a difficult problem at best, and often impossible because the flow may not be traceable at all when it is expressed in certain programming languages.

Some consider it incongruous that the algebra is used as an explicative tool, when logic is preferred as the basis for a relational language. The example at the end of Section 4.2 illustrates the algebraic and logic-based alternatives. It may also show the reader why it is easier to introduce the query and manipulative capabilities step by step using the algebra.

Every relational operator in the relational model is designed to work with operands that are relations free of duplicate rows, and to generate as a result a relation that is also free of duplicate rows.

The feature of permitting duplicate rows in any relation, base or derived, was added to SQL with the idea of making that language more powerful in some sense. In fact, this feature made SQL less powerful, because duplicate rows made use of the language more error-prone and damaged the interchangeability of ordering of the relational operators (see Chapter 23 for more detail). I call this mistaken belief—that one more feature cannot

possibly detract from a system's usability and power—the "one-more-feature trap."

Section 4.1 deals with the techniques used in explaining the operators. Section 4.2 describes the *basic operators,* which may help the reader to understand the power of relational languages. Section 4.3 discusses the *manipulative operators* such as **relational assignment, insert, update,** and **delete.** Chapter 5 deals with certain *advanced operators,* including outer equi-joins, outer unions, T-joins, user defined joins, and recursive joins.

The explanation of each operator includes a rough idea of what it does and how it is intended to be used. Also included are a precise definition, a formal algebraic notation, and some practical examples.

The notations used in this chapter are not intended to be taken as a serious proposal for a data sublanguage, but as illustrative and primarily for conceptual purposes. The operators are, however, intended to be interpreted as one definition of the power a relational language should possess. In Chapter 8, the way these operators deal with operands, from which certain kinds of information are missing, is discussed.

## 4.1 ■ Techniques for Explaining the Operators

Consider a relation S of degree *m* that has the following attributes:

Attribute A1 drawing its values from domain D1.
Attribute A2 drawing its values from domain D2.
. . . . . .
Attribute A*m* drawing its values from domain D*m*.

The relation S is abbreviated

S ( A1:D1 A2:D2 . . . A*m*:D*m*),

and often

S ( A1 A2 . . . A*m*)

when the domains are listed separately.

It must be remembered that the ordering

A1, A2, . . . A*m*

is insignificant in practice even to the user (except in the few cases where the command being explained requires the column ordering to be made explicit). Normally, we shall make use of column ordering for explanatory purposes only, just as is done in mathematics. There is no implication that components of tuples (i.e, rows) must be *stored* in any of the sequences used in explaining how the operators transform the operands into results.

In explaining the basic relational operators, we shall make use of the relation S of degree $m$ just cited, and also a relation T of degree $n$, denoted

T ( B1:E1 B2:E2 . . . B$n$:E$n$).

We shall consider the example of **Cartesian product** to illustrate the explanatory techniques. Incidentally, the major use of this operator is itself explanatory. Definitions of some of the other operators (such as **relational division**) make conceptual use of it. The designer of a DBMS product is advised to implement **Cartesian product** as a special case of the **theta-join** discussed later because it is rarely needed in practice.

To introduce the relational version of **Cartesian product** gradually, let us consider a simple example first. Suppose that a database contains information about suppliers and shipments received from these suppliers. The supplier relation S contains the serial number S# of all the suppliers in the database, their names SNAME, and other immediate properties (marked ". . ."). The shipment relation SP contains the serial number S# of the supplier making each shipment, the serial number P# of the part shipped, the date SHIP_DATE the shipment was received, and other immediate properties of the shipment (marked ". . .").

To keep the example simple, suppose that relation SP″ is SP restricted to those shipments with SHIP_DATE between 89-01-01 and 89-03-30 inclusive. Relation S has three rows; relation SP″, four rows:

| S | (S# | SNAME | . . .) |
|---|---|---|---|
| | s1 | Jones | . . . |
| | s2 | Smith | . . . |
| | s3 | Clark | . . . |

| SP″ | (S# | P# | SHIP_DATE | . . . ) |
|---|---|---|---|---|
| | s1 | p1 | 89-03-31 | . . . |
| | s1 | p2 | 89-03-20 | . . . |
| | s2 | p7 | 89-02-19 | . . . |
| | s3 | p2 | 89-01-15 | . . . |

In the relational model, the **Cartesian product** C of relation S with relation SP″ is as shown in the following 12-row relational table:

| C | (. . . | SNAME | S# | S# | P# | SHIP_DATE | . . .) |
|---|---|---|---|---|---|---|---|
| | . . . | Jones | s1 | s1 | p1 | 89-03-31 | . . . |
| | . . . | Jones | s1 | s1 | p2 | 89-03-20 | . . . |
| | . . . | Jones | s1 | s2 | p7 | 89-02-19 | . . . |
| | . . . | Jones | s1 | s3 | p2 | 89-01-15 | . . . |
| | . . . | Smith | s2 | s1 | p1 | 89-03-31 | . . . |
| | . . . | Smith | s2 | s1 | p2 | 89-03-20 | . . . |
| | . . . | Smith | s2 | s2 | p7 | 89-02-19 | . . . |
| | . . . | Smith | s2 | s3 | p2 | 89-01-15 | . . . |
| | . . . | Clark | s3 | s1 | p1 | 89-03-31 | . . . |
| | . . . | Clark | s3 | s1 | p2 | 89-03-20 | . . . |
| | . . . | Clark | s3 | s2 | p7 | 89-02-19 | . . . |
| | . . . | Clark | s3 | s3 | p2 | 89-01-15 | . . . |

The ordering of columns shown is there merely because the result must be displayed on paper. This ordering should be ignored. Note how each row of S is combined with each and every row of SP″. The same result is generated if the relational version of the **Cartesian product** of SP″ with S is requested.

Clearly, the **Cartesian product** has two operands, each one a relation. Consider the **Cartesian product** of S and T, denoted S × T. To form the **Cartesian product** U = S × T, concatenate each and every tuple of S with each and every tuple of T. In textbooks on mathematics the usual explanation is that U is the set of all tuples of the form

$$< < a1,a2,. \ . \ .,am >,< b1,b2,. \ . \ .,bn > >,$$

where $< a1,a2,. \ . \ .,am >$ is a tuple of S and $< b1,b2,. \ . \ .,bn >$ is a tuple of T. In this case, U would be a binary relation whose pairs have an $m$-tuple as the first component, and an $n$-tuple as the second component. For purposes of database management, it is more useful to adopt a slightly different definition, and to say that U is the set of all tuples of the form

$$< a1,a2,. \ . \ .,am,b1,b2,. \ . \ .,bn >.$$

In this case, U is a relation of degree $m + n$. Actually, because positional concepts are de-emphasized in the relational model, it is preferable to define U as a relation of degree $m + n$ that has the form

U ( A1:D1 A2:D2 . . . A$m$:D$m$ B1:E1 B2:E2 . . . B$n$:E$n$ ),

where column names and domain names are used rather than the positioning of typical elements of these columns in tuples. Note that, with this de-emphasis on positioning concepts, a surprising consequence is that

$$S \times T = T \times S,$$

which holds true for all relations S and T.

It is necessary, however, to face up to the problem that, for some $i$ and some $j$, it may happen that the pair of names in the expression A$i$:D$i$ may be the same as the pair of names in the expression B$j$:E$j$—the A and D names come from S, whereas the B and E names come from T. According to Feature RN-3 of the relational model (see Chapter 6), it is required that, given any relation, every one of its columns must have a distinct name.

Assume that S and T do not refer to one and the same relation. Then, all we need do to make sure the columns of the result are distinctly named is to attach the prefix S to A$i$ (where S is the name of the relation from which A$i$ came), and attach the prefix T to B$j$ (where T is the name of the relation from which B$j$ came). Prefixing in these cases means adopting a syntax such as S.A$i$ and T.B$j$.

This technique will not work if S = T. In this case, if $m$ is the degree of S, there will be $m$ pairs of columns having the property that each member

of the pair has the same name, as well as precisely the same database values. To make the column names distinct for each of the members of these pairs of columns, each column name is qualified by the name of the source relation (namely S in each case), followed by a punctuation mark (such as a period), followed by a citation number, followed by a second punctuation mark (this can be a period also). The citation number is either 1 or 2, depending on whether the first-cited or the second-cited occurrence of relation S is the source of the pertinent column in the result. In Chapter 6 we discuss column naming and column ordering in more detail.

### *RB-1* De-emphasis of Cartesian Product as an Operator

A relational DBMS must not support **Cartesian product** as an explicitly separate operator. A relational command, however, may have an extreme case that is interpreted by the DBMS as a request for a **Cartesian product.**

Note that the **Cartesian product** U contains no more information than its components S and T contain together. However, U consumes much more memory or disk space and channel time than the two relations S and T consume together. These are two good reasons why, in any implementation of the relational model, **Cartesian product** should be de-emphasized, and used primarily as a tool for explanatory or conceptual purposes. The main purpose in discussing this operator in some detail at this point is to show how the relational operators will normally be treated.

One may ask, "Why is it necessary to discuss column naming and column ordering at all?" The simple answer is that every result of executing a relational operation can be either an intermediate or a final result of a relational command. In either case, a subsequent relational operation may employ this result as an operand, and, as we shall see, the specification of some kinds of relational operation involves employing the names of columns of one or other of the operands and sometimes the ordering of these columns also.

Of course, we are not referring to "stored ordering" when we speak of column ordering. We are referring either to the *citation ordering* (the ordering in which columns are cited in some relational command) or to an ordering derived from such citation.

## 4.2 ■ The Basic Operators

The basic operators of RM/V2 include **projection, theta-selection, theta-join, natural join, relational division, relational union, relational difference,** and

**relational intersection. Projection** and **theta-selection** act upon one operand only, while each of the remaining operators acts upon two operands. In every case, operands and results are true relations with no duplicate rows.

Figure 4.1 is a simple guide to the Basic operators of the relational model. All of the operators except **Cartesian product** are intended to be implemented in a relational DBMS. **Cartesian product,** although a good conceptual tool, wastes storage space and channel time if implemented in the DBMS and used by application programmers or invoked interactively from terminals.

## *RB-2* The Project Operator

The **project** operator employs a single R-table as its operand. The operator generates an intermediate result in which the columns listed by name in the command are saved, and the columns omitted from the command are ignored. From this R-table it then generates the final result by removing all occurrences except one of each row that occurs more than once.

Suppose that the **project** operator is applied to the following R-table EMP of degree five. Suppose that the columns are named as follows:

EMP# (primary key), ENAME, BIRTH_DATE, SALARY, and H_CITY.

| EMP | (EMP# | ENAME | BIRTH_DATE | SALARY | H_CITY ) |
|---|---|---|---|---|---|
| | E107 | Rook | 23-08-19 | 10,000 | Wimborne |
| | E912 | Knight | 38-11-05 | 12,000 | Poole |
| | E239 | Knight | 38-11-05 | 12,000 | Poole |
| | E575 | Pawn | 31-04-22 | 11,000 | Poole |

Note that there are two employees named "Knight," and that these two Knights have the same birthdate, earn the same salary, and live in the same city. The only piece of information that tells us that these are distinct employees is the primary key EMP# (E912 for one person, E239 for the other). If the primary-key values were not in the database, there would be clear ambiguity in the data: specifically, do the two rows represent two distinct persons, or are the duplicate rows there by accident? This is one of the many reasons why the primary-key concept should be explicitly supported in every relational database management system.

The intermediate result that is generated when applying **project** onto ENAME, BIRTH_DATE, SALARY, and H-CITY is a table (but not an R-table) obtained by selecting only those columns cited in the **project** command:

| X | (ENAME | BIRTH_DATE | SALARY | H_CITY ) |
|---|---|---|---|---|
| | Rook | 23-08-19 | 10,000 | Wimborne |
| | Knight | 38-11-05 | 12,000 | Poole |
| | Knight | 38-11-05 | 12,000 | Poole |
| | Pawn | 31-04-22 | 11,000 | Poole |

The final result, in which duplicate rows are removed, is the following R-table:

| Y | (ENAME | BIRTH_DATE | SALARY | H_CITY ) |
|---|---|---|---|---|
| | Rook | 23-08-19 | 10,000 | Wimborne |
| | Knight | 38-11-05 | 12,000 | Poole |
| | Pawn | 31-04-22 | 11,000 | Poole |

Notice how, in generating the R-table Y from the table X, the duplicate row for "Knight" was removed. It has been asserted that duplicate rows should not be removed, and in fact the language SQL fails to remove duplicate rows unless the user adds to the command the qualifier DISTINCT. When the user fails to add this SQL qualifier, the result is then no longer a true relation in the mathematical sense. If such tables are permitted, the consequences are devastating. The power of any relational language is severely reduced because of the reduced interchangeability in the ordering of relational operators (see Chapter 23). The form of projection in which duplicate rows are not removed from the result is called a corrupted version of the **project** operator.

If the user must retain the two occurrences of rows containing the name "Knight," he or she should retain the primary key EMP# as well by projecting onto EMP#, ENAME, together with zero or more of the remaining columns of EMP. In Chapter 17 on view updatability, much emphasis is placed on retaining the primary key in defining a virtual relation (usually called a view). When this is done, the user normally gains the advantage of being able to perform **inserts, updates,** and **deletes** on this view.

A second example shows that removing duplicates can be precisely what the user needs. Suppose we wish to find the cities in which the employees live. By taking the projection of EMP onto H_CITY, we obtain

| Z | (H_CITY ) |
|---|---|
| | Wimborne |
| | Poole |

which is exactly what was requested. If 100 employees happen to live in Poole, why obtain 100 copies of the name Poole? If the number living in each city is needed, the DBMS should be capable of counting these numbers (as is RM/V2) and capable of providing the information as output in the form of a second column (this one computed).

Using the notation I introduced in early papers on the relational model

[Codd 1970, 1971a–d], the three applications of the **project** operator just introduced are represented as follows:

Example 1: Z ← EMP [ ENAME, BIRTH_DATE, SALARY, H_CITY ]

Example 2: Z ← EMP [ EMP#, SALARY, H_CITY ]

Example 3: Z ← EMP [ H_CITY ]

The naming and virtual ordering of columns pertinent to the result of a **project** operation are discussed in Chapter 6.

The version of the **project** operator just described is useful whenever few columns are to be *saved* (and possibly many dropped). A second version is useful whenever few columns are to be *dropped* (and possibly many saved). In this version the columns to be dropped are listed, instead of the columns to be saved. However, this version is less adaptable to changes in the source relation. For example, if a new column is added to the source relation, its name may have to be added to the list of columns to be dropped in the projection.

### *RB-3–RB-12* The Theta-Select Operator

The **theta-select** operator, originally called **theta-restrict,** employs a single R-table as its operand. In normal use the term **theta-select** is abbreviated to **select,** and this means that the equality comparator '=' should be assumed unless there is an explicit alternative comparator specified. It generates as a result an R-table that contains some of the same complete rows that the operand contains—those rows, in fact, that satisfy the condition expressed in the command. To distinguish **theta-select** from the **select** command of SQL, we shall sometimes refer explicitly to **theta-select** as the algebraic **select,** and refer explicitly to SQL's **select** as the **select** of SQL. It is important to remember that the operand contains no duplicate rows, and that therefore neither does the result.

The complete class of **select** operators is called **theta-select,** where **theta** stands for any of the 10 comparators listed on page 70. Features RB-3–RB-12 are the 10 types of **theta-select** operators corresponding respectively to the 10 comparators. An extension of the **select** operator (Feature RB-13) and an extension of the **join** operator (Feature RB-25) are also discussed. These extensions permit Boolean combinations of comparisons.

Suppose that the **select** operator is applied to the same R-table as used in the explanation of **project,** and that the rows specified to be saved are those for which the SALARY has a specified value. In the three examples of this kind of query cited next, I again use the notation introduced in my

early papers on the relational model [Codd 1970, 1971a–d]. The result for each query is listed immediately after the query.

Example 4: Z ← EMP [ SALARY = 12,000 ]

| Z | (EMP# | NAME | BIRTH_DATE | SALARY | H_CITY ) |
|---|---|---|---|---|---|
| | E912 | Knight | 38-11-05 | 12,000 | Poole |
| | E239 | Knight | 38-11-05 | 12,000 | Poole |

Example 5: Z ← EMP [ SALARY = 11,000 ]

| Z | (EMP# | NAME | BIRTH_DATE | SALARY | H_CITY ) |
|---|---|---|---|---|---|
| | E575 | Pawn | 31-04-22 | 11,000 | Poole |

Example 6: Z ← EMP [ SALARY = 20,000 ]

| Z | (EMP# | NAME | BIRTH_DATE | SALARY | H_CITY ) |
|---|---|---|---|---|---|

Example 6 yields an R-table that happens to have no rows at all (since no employee earns this amount); the five columns have the same headings as the R-table EMP. An R-table of this kind is perfectly legitimate. It represents an empty relation of degree five with the same column headings as EMP. In each of the three examples (4, 5, and 6), the result is union-compatible with EMP (see RB-26 for an explanation of union-compatibility).

All of the **select** operators discussed so far involve the comparison of database values, on the one hand, with a constant, on the other hand; the comparator in each case has been equality. The **theta-select** operator can be used to compare a database value within a row of an R-table with another database value within the same row, and to execute this comparison for all rows of that R-table (see Example 8).

Further, any of the following 10 comparators may be used:

1. EQUALITY
2. INEQUALITY
3. LESS THAN
4. LESS THAN OR EQUAL TO
5. GREATER THAN
6. GREATER THAN OR EQUAL TO
7. GREATEST LESS THAN
8. GREATEST LESS THAN OR EQUAL TO
9. LEAST GREATER THAN
10. LEAST GREATER THAN OR EQUAL TO

In the following example of **theta-select, theta** is taken to be the LESS

THAN comparator (<):

Example 7: Z ← EMP [ SALARY < 12,000 ].

| Z | (EMP# | ENAME | BIRTH_DATE | SALARY | H_CITY ) |
|---|---|---|---|---|---|
| | E912 | Rook | 23-08-19 | 10,000 | Wimborne |
| | E575 | Pawn | 31-04-22 | 11,000 | Poole |

The relational model provides a very important safety feature, introduced at the beginning of Chapter 3 and repeated here for convenience:

### *RT-1* Safety Feature when Comparing Database Values

When comparing a database value in one column with a database value in another, the DBMS checks that the two columns draw their values from a common domain, unless the domain check is overridden (see Feature RQ-9 in Chapter 10). When comparing (1) a computed value with a database value or (2) one computed value with another computed value, however, the DBMS merely checks that the basic data types are the same.

Table 4.1 shows the columns of EMP and the domains from which the columns draw their values. Note that no two columns of EMP draw their values from a common domain. Therefore, let us expand the EMP relation by appending another column with the heading BONUS, and assume certain values for the bonus component. All of the BONUS values are drawn from the currency domain. The modified version of EMP follows:

| EMP″ | (EMP# | ENAME | BIRTH_DATE | SALARY | H_CITY | BONUS) |
|---|---|---|---|---|---|---|
| | E107 | Rook | 23-08-19 | 10,000 | Wimborne | 15,800 |
| | E912 | Knight | 38-11-05 | 12,000 | Poole | 6,700 |
| | E239 | Knight | 38-11-05 | 12,000 | Poole | 13,000 |
| | E575 | Pawn | 31-04-22 | 11,000 | Poole | 3,100 |

***Table 4.1*** **Columns of EMP and Domains from which Values are Drawn**

| Column | Domain |
|---|---|
| EMP# | Employee serial numbers |
| ENAME | Employee names |
| BIRTH_DATE | Dates |
| SALARY | Currency |
| H_CITY | City names |

Now consider Example 8:

$Z \leftarrow EMP''$ [BONUS > SALARY].

This command yields the following R-table:

| Z | (EMP# | ENAME | BIRTH_DATE | SALARY | H_CITY | BONUS) |
|---|---|---|---|---|---|---|
| | E107 | Rook | 23-08-19 | 10,000 | Wimborne | 15,800 |
| | E239 | Knight | 38-11-05 | 12,000 | Poole | 13,000 |

The naming and virtual ordering of the columns in the result of a **select** operator are discussed in Chapter 6.

## *RB-13* The Boolean Extension of Theta-Select

Let R denote any relation whose simple or composite columns include A and B. Let @ denote one of the 10 comparators used in **theta-select,** and let *x* denote a host-language variable or constant. Suppose that R [ A @ *x* ] and R [ A @ B ] denote **theta-select** operations. Then A @ *x* and A @ B are called *comparing terms,* and each comparing term is truth-valued.

The usual comparing terms in all 10 types of **theta-select** can be used in any Boolean combination within a single operator. Such an operator is called **extended theta-select.** A Boolean combination of the comparing terms, each of which is truth-valued, is any combination of these terms using the elementary logical connectives NOT, OR, AND, and IMPLIES.

Clearly, this operator is redundant from the user's viewpoint in the sense that a query involving **extended theta-select** can be re-expressed in terms of simple **theta-selects,** together with some combination of relational unions, differences, and intersections. However, it permits such queries to be expressed more clearly and concisely.

Consider a simple example of the practical use of this extended **join.** Suppose that a database includes a relation E describing employees, and that two of the immediate properties recorded are the gender (male M or female F) and present salary. The Boolean extension of **theta-select** can be used to place a request for the names of the employees who are female and earn more than $20,000:

E [(GENDER = F) $\wedge$ (SALARY > 20,000)].

One DBMS product on the market in the mid-1980s supported this single feature of the relational model and no others. The vendor falsely advertised that the product was a relational DBMS.

## *RB-14–RB-23* The Theta-Join Operator

The **theta-join** operator employs two R-tables as its operands. It generates as a result an R-table that contains rows of one operand (say S) concatenated with rows of the second operand (say T), but only where the specified condition is found to hold true. For brevity, this operator is often referred to as **join.**

The condition expressed in the **join** operator involves comparing each value from a column of S with each value from a column of T. The columns to be compared are indicated explicitly in the **join** command; these columns are called the *comparand columns.* This condition can involve any of the 10 comparators cited in the list presented on page 74 (after a description of the operator applied with equality as the comparator).

Suppose that the **join** operator is applied to the following two relational tables:

| S | (EMP# | ENAME | H_CITY | ) |
|---|---|---|---|---|
| | E107 | Rook | Wimborne | |
| | E912 | Knight | Poole | |
| | E239 | Pawn | Poole | |

| T | (WHSE# | W_CITY | ) |
|---|---|---|---|
| | W17 | Wareham | |
| | W34 | Poole | |
| | W92 | Poole | |

For this example, we shall assume that the only pair of columns having a common domain are S.H_CITY and T.W_CITY; each of these two columns draws its values from the CITY domain. Therefore, in this example, these are the only two columns that can be used as comparand columns in a **join** of these two relations. Normally, there are several pairs of potential comparand columns, not just one pair.

The **join** operator based on equality that is now being discussed is called **equi-join**. An **equi-join** of S on H_CITY with T on W_CITY finds all of the employees and warehouses that are located in the same city. The formula for this **join** is

$$U = S [ H\_CITY = W\_CITY ] T.$$

The result obtained assuming the values cited earlier is as follows:

| U | (EMP# | NAME | H_CITY | WHSE# | W_CITY) |
|---|---|---|---|---|---|
| | E912 | Knight | Poole | W34 | Poole |
| | E912 | Knight | Poole | W92 | Poole |
| | E239 | Pawn | Poole | W34 | Poole |
| | E239 | Pawn | Poole | W92 | Poole |

Some people find it easier to think of the **equi-join** of S on H_CITY with T on W_CITY as the **Cartesian product** of S with T, followed by the selection of just those rows for which

H_CITY = W_CITY.

Naturally, if the goal is good performance, the designer of a DBMS should not implement **equi-join** in this way.

Any of the following 10 comparators may be used in a **join:**

1. EQUALITY
2. INEQUALITY
3. LESS THAN
4. LESS THAN OR EQUAL TO
5. GREATER THAN
6. GREATER THAN OR EQUAL TO
7. GREATEST LESS THAN
8. GREATEST LESS THAN OR EQUAL TO
9. LEAST GREATER THAN
10. LEAST GREATER THAN OR EQUAL TO

The complete class of **joins** is called **theta-join,** where **theta** stands for any of the 10 comparators just listed. As an example, consider

U = PART [ OH_Q < SHIP_Q ] SHIP.

The result U is generated from the following operands:

**PART**

| P# | PNAME | OH_Q |
|---|---|---|
| N12 | nut | 1000 |
| B39 | bolt | 1500 |

**SHIP**

| WHSE# | SHIP_Q |
|---|---|
| W34 | 1200 |
| W92 | 2000 |

**U**

| P# | PNAME | OH_Q | WHS# | SHIP_Q |
|---|---|---|---|---|
| N12 | nut | 1000 | W34 | 1200 |
| N12 | nut | 1000 | W92 | 2000 |
| B39 | bolt | 1500 | W92 | 2000 |

Just as mentioned in the discussion of the **select** operator, when comparing a database value in one column with a database value in another, the DBMS checks that the two columns draw their values from a common domain, unless the domain check is overridden (see Feature RQ-9 in Chapter 10 and Feature RJ-6 in Chapter 11). The practical reasons for this check and its override were discussed in Chapter 3 (see the fourth reason to support domains in a DBMS together with Feature RT-1).

It is possible to conceive of each of the 10 **theta-joins** of S with T as a subset of the **Cartesian product** of S with T. However, as stated for **equi-**

**join, Cartesian product** should not be used in the implementation of any one of the 10. The result of each of the 10 **theta-joins** has a degree equal to the sum of the degrees of the operands.

The result of a **join** based on Comparators 3, 4, 5, or 6 in the preceding list is often quite large in terms of the number of rows. Not surprisingly, this number is often nearly as large as the number of rows in the **Cartesian product.** If, on the other hand, any one of Comparators 7, 8, 9, or 10 in the list is selected, the resulting relation is quite modest in size. That is one of the more important reasons why a user may choose one of these comparators instead of Comparators 3, 4, 5, or 6.

It is certainly possible to join a relation with itself, provided that it has two or more columns on a common domain. Let us modify the relation EMP by adding a column that identifies the immediate manager for each and every employee.

| EMP | (EMP# | ENAME | BIRTH_DATE | SALARY | CITY | MGR#) |
|---|---|---|---|---|---|---|
| | E107 | Rook | 23-08-19 | 10,000 | Poole | E321 |
| | E912 | Knight | 38-11-05 | 12,000 | Wareham | E321 |
| | E239 | Knight | 38-11-05 | 12,000 | Wareham | E107 |
| | E575 | Pawn | 31-04-22 | 11,000 | Poole | E239 |
| | E321 | Queen | 27-02-28 | 20,000 | Wimborne | — |

In this table, MGR# is the employee serial number of the immediate manager of the person designated by the left-most component (EMP#). The employee designated E321 appears to be "top dog," since the serial number of his or her immediate manager is unknown.

Because the columns EMP# and MGR# both draw their values from the common domain of employee serial numbers, it is clear that we may join EMP with itself, using the EMP# and MGR# columns as comparands. The assignment could be

$$Z \leftarrow \text{EMP [EMP\# = MGR\#] EMP},$$

but fewer columns are needed in the final result. Suppose that projection is applied to the result from the **join,**

$$Z \leftarrow (\text{EMP [EMP\# = MGR\#] EMP})[\text{EMP\#, ENAME, CITY, MGR\#, MGR\_NAME}],$$

where MGR_NAME denotes the employee name ENAME of the manager. The result of this evaluation is as follows:

| Z | (EMP# | NAME | CITY | MGR# | MGR_NAME) |
|---|---|---|---|---|---|
| | E107 | Rook | Poole | E321 | Queen |
| | E912 | Knight | Wareham | E321 | Queen |
| | E239 | Knight | Wareham | E107 | Rook |
| | E575 | Pawn | Poole | E239 | Knight |

The tuple < E321, Queen, Wimborne, —, — > is omitted because it does not satisfy the equality condition. The query "Who earns a higher salary than his immediate manager?" can be answered by a **join** of EMP with itself, once again using EMP# and MGR# as comparand columns.

Under certain quite broad conditions, the **project, select,** and **join** operators can be executed in any sequence and yield a result that is independent of the sequence chosen. This fact is important when the DBMS is attempting to generate the most efficient target code from a relational language acting as source code. This process is called *optimization;* the DBMS component involved is called the *optimizer. This inter-changeability of ordering of the operators is damaged when duplicate rows are allowed.* More is said about this point in Chapter 23.

The following observations are intended to complete the discussion of **theta-joins.** Users who are wedded to the approaches of the past often think that the only kind of **join** in the relational model is either the **equi-join** or the **natural join** (discussed later in this section), and that the only comparand columns allowed are primary keys or foreign keys. In other words, these users see **joins** as key-based **equi-joins** only. This tunnel vision may be due to the fact that DBMS products of the pre-relational variety tended to support pointers or links only where the relational model supports primary and foreign keys.

**Theta-joins** are the algebraic counterparts of queries that use the existential quantifier of predicate logic. A brief explanation of these quantifiers appears later in the explanation of **relational division** (Feature RB-29). Incidentally, the result of an **equi-join** can be empty, even if neither of its operands is empty.

Note the following identities:

R [ A = B ] S = S [ B = A ] R
R [ A < B ] S = S [ B > A ] R.

**Equi-join** is commutative, whereas **join** on LESS THAN (<) is related simply to **join** on GREATER THAN (>), and is not commutative.

## *RB-24* The Boolean Extension of Theta-Join

Let R, S denote any relations whose simple or composite columns include R.B and S.C. Suppose that R.B and S.C draw their values from a common domain. Let @ denote one of the 10 comparators used in **theta-join.** Suppose that R [ B @ C ] S denotes a **theta-join** operation. Then B @ C is called a *comparing term,* and each comparing term is truth-valued.

The usual comparing terms in all 10 types of **theta-join** can be used in any Boolean combination within a single operator of the extended type. It is important to remember that each pair of comparand columns cited in the comparing terms must draw their values from a domain common to the pair. Such an operator is called **extended theta-join.** A Boolean combination of the comparing terms, each of which is truth-valued, is any combination of these terms using the elementary logical connectives NOT, OR, AND, and IMPLIES.

Consider a simple example of the practical use of this **extended join.** Suppose that a database includes two relations T1 and T2, respectively describing the members of two teams of employees (each team is assigned to a different project). Suppose that the need arises to pair off individual members of Team 1 with individual members of Team 2 based on both of the following:

- equality in jobcode (abbreviated J1, J2);
- birthdate B1 of the Team 1 member being earlier than the birthdate B2 of the Team 2 member.

The Boolean extension of **theta-join** can be used to place a request for all the eligible pairs of team members who satisfy both of these conditions:

$$T1\ [(J1 = J2) \wedge (B1 < B2)]\ T2.$$

Note that, if jobcode happened to have the same column name in T1 as in T2, the name would have to be prefixed by the pertinent relation name (e.g., T1.J). The same applies to the birthdate columns.

## *RB-25* The Natural Join Operator

As described in the last section, an **equi-join** generates a result in which two of the columns are identical in values, although different in column names. These two columns are derived from the comparand columns of the operands; of course, the columns may be either simple or composite. Of the 10 types of **theta-join, equi-join** is the only one that yields a result in which the comparand columns are completely redundant, one with the other. The **natural join** behaves just like the **equi-join** except that one of the redundant columns, simple or composite, is omitted from the result. To make the column naming clear and avoid impairing the commutativity, the retained comparand column is assigned whichever of the two comparand-column names occurs first alphabetically.

The degree of the result generated by **natural join** is less than that generated by **equi-join** on the same operands. The degree of the former result is the sum of the degrees of the operands reduced by the number of simple columns in the comparand column of either operand.

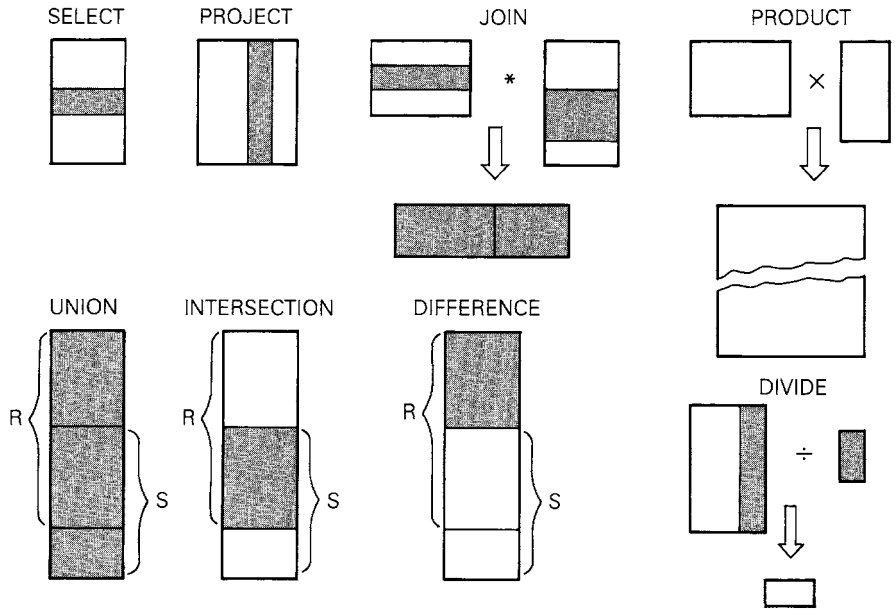
**Natural join** is probably most useful in the theory of database design, especially in normalizing a collection of relations. It is included here primarily for that reason.

As an aid to understanding the following three operators—**relational union, relational intersection,** and **relational difference**—the reader may wish to refer to Figure 4.1.

## *RB-26* The Union Operator

The **relational union** operator is intentionally not as general as the union operator in mathematics. The latter permits formation of the union of a set of buildings with a set of parts and also with a set of employees. On the other hand, **relational union** permits, for example, (1) a set of buildings to be united with another set of buildings, (2) a set of employees to be united with another set of employees, or (3) a set of parts to be united with another set of parts.

***Figure 4.1*** **The Basic Operators of the Relational Model**



**Relational union** is intended to bring together in one relation all of the facts that happen to exist in whatever two relations are chosen to be its operands, provided these two relations contain the same kind of facts. It does this by copying rows from both of its operands into the result, but *without generating duplicate rows in the result.* The relations that are combined by the **relational union** operator must be compatible with one another in having rows of similar type, thus ensuring that the result is a relation. It must be remembered that all relations are sets, but that not all sets are relations. Thus, **relational union** is intentionally *not* as general as set union. Union compatibility is now discussed in more detail.

Suppose that S and T are two relations. Then, S and T are *union-compatible* if they are of the same degree and it is possible to establish at least one mapping between the columns of S and those of T that is one-to-one, and with the property that, for every column A of S and every column B of T, if column A is mapped onto column B, then A and B draw their values from a common domain. Of course, the number of such mappings between S and T may be zero, one, two, or more. If it happens that no such mapping exists, then S and T are not union-compatible, and any request from the user to form S UNION T causes an error code to be returned.

The **union** operator requires that its two operands (which are relations, of course) be union-compatible. In practice, it is rare that two base relations are union-compatible, but not at all rare that two derived relations are. The **union** operator also requires that the column alignment for its two operands, whether explicit or implied, be in conformity with one of the mappings that guarantees union compatibility (this is discussed further later in this section).

The result of applying **union** to relations S and T is a relation containing all of the rows of S together with all of the rows of T, but with duplicate rows removed. The removal of duplicate rows becomes necessary when it happens that relation S has some rows in common with relation T. Those DBMS implementations that either require or permit the retention of duplicate rows in the final result of a **union** will give rise to the same severe problems cited earlier in the description of the **project** operator.

The principal reason why **relational union, intersection,** and **difference** are not as general as their mathematical counterparts is that, in a relational system, it must be easy to find any desired information in the result. Hence, the result is required to be a relation. The result of these operators is also constrained to be a relation because this is necessary for operational closure (see Feature RM-5 in Chapter 12).

Consider an example involving domestic suppliers S and overseas suppliers T. These relations are likely to be separated from one another because certain properties are applicable to one but not the other. For the example, suppose that we take the projection S″ of S and the projection T″ of T on certain columns common to both S and T, namely, supplier serial number and supplier name.

Suppose that the corresponding snapshots are as follows:

| S″ | S# | NAME |
|---|---|---|
| | S11 | Peter |
| | S12 | Smith |
| | S17 | Clark |
| | S23 | Rock |
| | S25 | Roth |

| T″ | S# | NAME |
|---|---|---|
| | S2 | Jones |
| | S17 | Clark |
| | S3 | Blake |
| | S7 | Tack |

Applying the operator **union,** we obtain the following:

| Z | S# | NAME |
|---|---|---|
| | S11 | Peter |
| | S12 | Smith |
| | S17 | Clark |
| | S23 | Rock |
| | S25 | Roth |
| | S2 | Jones |
| | S3 | Blake |
| | S7 | Tack |

**Z ← S″ union T″**

We observe that Clark, with the serial number S17, is both a domestic and an overseas supplier. Note that the row < S17, Clark > is not repeated in the result.

The designer of a relational language must face the difficulty that, when applying the **union** operator in some circumstances, the user must specify in some detail which columns of one relation are to be aligned with which columns of the second relation. This alignment is particularly relevant when two or more columns of one operand have the same domain. When this is true of one operand, it must be true of the other, if they are to satisfy the requirement of union compatibility.

The simplest approach appears to be as follows:

- if all the domains of one relation are distinct, then the DBMS aligns the columns by ensuring that aligned pairs have the same domain;
- if not all the domains of one relation are distinct, then
  1. for those columns of one operand that have distinct domains within that operand, the DBMS aligns them with the columns of the other operand by ensuring that aligned pairs have the same domain; and
  2. it aligns the remaining columns by accepting the pairing of these columns as specified by the user in his or her request or, if no such pairing is specified, it pairs columns by name alphabetically (lowest alphabetically from one operand with lowest alphabetically from the other, and so on).

The relational model requires this approach to be adopted within the DBMS for the operators **relational union, intersection,** and **difference.**

In countries that do not use the Roman alphabet, it may be necessary to replace the alphabetic default by some other kind of default.

The DBMS sends an error message if either the implicit alphabetic ordering or the explicit alignment declared by the user fails to satisfy the constraint that pairs of columns that are aligned for the **union** operator must draw their values from a common domain. This approach to column alignment is required by the relational model until such time as a simpler technique is devised to deal with this column-alignment problem.

The following special case is noteworthy. Whenever the two operands of a **union** have primary keys PK1 and PK2, which draw their values from a common domain, and whenever PK1 and PK2 happen to be aligned for a requested **union,** then the DBMS deduces that the primary key of the result is a column PK that is formed by uniting PK1 with PK2. One consequence of this is that the DBMS rejects duplicate values in column PK of the result.

## *RB-27* The Intersection Operator

Suppose that S and T are two relations that are union-compatible. Then, they are sufficiently compatible with one another for the **intersection** operator to be applicable. Columns have to be aligned in the same way as for the **union** operator. The result of applying **intersection** to relations S and T is a relation containing only those rows of S that also appear as rows of T. Of course, the resulting relation contains no duplicate rows, since neither of the operands contain any.

Consider the same example as before involving domestic suppliers S and overseas suppliers T. Assume that the same projections as before have been made to generate relations S″ and T″.

| S″ | S# | NAME |
|---|---|---|
| | S11 | Peter |
| | S12 | Smith |
| | S17 | Clark |
| | S23 | Rock |
| | S25 | Roth |

| T″ | S# | NAME |
|---|---|---|
| | S2 | Jones |
| | S17 | Clark |
| | S3 | Blake |
| | S7 | Tack |

Applying the operator **intersection,** we obtain the following R-table:

| Z | S# | NAME |
|---|---|---|
| | S17 | Clark |

Z ← S″ intersection T″

The supplier Clark with the serial number S17 is both a domestic and an overseas supplier. Note also how the row < S17, Clark > was not repeated in the result.

The approach to column alignment is the same as with **union:**

- if all the domains of one relation are distinct, then the DBMS aligns the columns by ensuring that aligned pairs have the same domain;
- if not all the domains of one relation are distinct, then
  1. for those columns of one operand that have distinct domains within that operand, the DBMS aligns them with the columns of the other operand by ensuring that aligned pairs have the same domain; and
  2. it aligns the remaining columns by accepting the pairing of these columns as specified by the user in his or her request or, if no such pairing is specified, it pairs columns by name alphabetically (lowest alphabetically from one operand with lowest alphabetically from the other, and so on).

In countries that do not use the Roman alphabet, it may be necessary to replace the alphabetic default by some other kind of default.

The DBMS sends an error message if either the implicit alphabetic ordering or the explicit alignment declared by the user fails to satisfy the constraint that pairs of columns that are aligned for the **intersection** operator must draw their values from a common domain.

## *RB-28* The Difference Operator

Suppose that S and T are two relations that are union-compatible. Then, they are sufficiently compatible with one another for the **relational difference** operator to be applicable. Columns must be aligned in the same way as for the **union** operator. The result of applying **relational difference** to relations S and T is a relation containing only those rows of S that do not appear as rows of T. Of course, the resulting relation contains no duplicate rows.

Consider the same example involving domestic suppliers S and overseas suppliers T. Assume that the same projections as before have been made to generate relations S″ and T″:

| S″ | S# | NAME |
|---|---|---|
| | S11 | Peter |
| | S12 | Smith |
| | S17 | Clark |
| | S23 | Rock |
| | S25 | Roth |

| T″ | S# | NAME |
|---|---|---|
| | S2 | Jones |
| | S17 | Clark |
| | S3 | Blake |
| | S7 | Tack |

Applying the operator **relational difference,** we obtain the following:

| Z ( S# | NAME ) |
|---|---|
| S11 | Peter |
| S12 | Smith |
| S23 | Rock |
| S25 | Roth |

$Z \leftarrow S'' - T''$

The supplier Clark with the serial number S17 is both a domestic and an overseas supplier, which explains why the row < S17, Clark > does not appear at all in the result.

The approach to column alignment is the same as with **union:**

- if all the domains of one relation are distinct, then the DBMS aligns the columns by ensuring that aligned pairs have the same domain;
- if not all the domains of one relation are distinct, then
  1. for those columns of one operand that have distinct domains within that operand, the DBMS aligns them with the columns of the other operand by ensuring that aligned pairs have the same domain; and
  2. it aligns the remaining columns by accepting the pairing of these columns as specified by the user in his or her request or, if no such pairing is specified, it pairs columns by name alphabetically (lowest alphabetically from one operand with lowest alphabetically from the other, and so on).

In countries that do not use the Roman alphabet, it may be necessary to replace the alphabetic default by some other kind of default.

The DBMS sends an error message if either the implicit alphabetic ordering or the explicit alignment declared by the user fails to satisfy the constraint that pairs of columns that are aligned for the **relational difference** operator must draw their values from a common domain.

## *RB-29* The Relational Division Operator

**Relational division** is similar in some respects to division in integer arithmetic. In **relational division,** just as in integer arithmetic division, there is a dividend, a divisor, the quotient, and even a remainder. Thus, relational division has similarly named operands and results. Instead of being integers, however, these operands and results are all relations. None of them need contain any numeric information at all, and even if the operands do contain such information, it need not be the numeric components that play a crucial role in relational division.

Consider an example of division in integer arithmetic. Suppose that we are dividing 29 by 7. One must find the largest multiplier for 7 that yields a product that is equal to or less than 29. That multiplier is 4, since $4 \times 7 = 28$, and 28 is less than 29; while $5 \times 7 = 35$, and 35 is greater than 29.

In **relational division** the relational operator corresponding to multiplication is **Cartesian product.** The relational comparator corresponding to LESS THAN OR EQUAL TO ($\leq$) is SET INCLUSION.

When dividing one relation by another, at least one pair of columns (one column of the pair from the dividend, the other column from the divisor) must draw their values from the same domain. Such a pair of columns can be used as comparand columns (just as if we were attempting an **equi-join**).

Suppose that (1) relation S is the dividend, (2) relation T is the divisor, (3) the comparand columns are B from S and C from T, and (4) the column A from S is to be the source of values for the quotient. Then suppose that Q is the quotient obtained by dividing S on B by T on C. The assignment to Q is represented by

$$Q \leftarrow S\ [\ A, B\ /\ C\ ]\ T,$$

and we obtain the largest relation Q, such that Q[A] $\times$ T[C] is contained in S[A,B]. The term "largest relation" in this context means the relation that has the most tuples (rows), while still satisfying the specified condition.

As an example, suppose that we have a list of parts required for a certain job, and that the list is presented as a unary relation named LIST containing part serial numbers. LIST is an R-table with one column named P#. Suppose also that CAP has the same meaning as the CAPABILITY relation used in Section 1.2.3. Suppose that CAP and LIST have the following extensions:

**CAP**

| (S# | P# | SPEED | UNIT_Q | PRICE) |
|---|---|---|---|---|
| S1 | P1 | 5 | 100 | 10 |
| S1 | P2 | 5 | 100 | 20 |
| S1 | P6 | 12 | 10 | 600 |
| S2 | P3 | 5 | 50 | 15 |
| S2 | P4 | 5 | 100 | 15 |
| S3 | P6 | 5 | 10 | 700 |
| S4 | P2 | 5 | 100 | 15 |
| S4 | P3 | 5 | 50 | 17 |
| S4 | P5 | 15 | 5 | 300 |
| S4 | P6 | 10 | 5 | 350 |

**LIST**

| (P#) |
|---|
| P2 |
| P5 |
| P6 |

Note that SPEED denotes speed of delivery in number of working days.

Consider the query "Find the suppliers each of whom can supply every one of the parts listed in the given R-table LIST." This query is equivalent to

QUOT ← CAP [ S#, P# / P# ] LIST.

This query calls for CAP on S#, P# to be divided by LIST on P#. The result obtained is as follows:

| QUOT | (S# | SPEED | UNIT_Q | PRICE) |
|---|---|---|---|---|
| | S4 | 5 | 100 | 15 |
| | S4 | 15 | 5 | 300 |
| | S4 | 10 | 5 | 350 |

Note that the relation QUOT [S#] contains only one row, and this row contains only one component S4. Thus, the **Cartesian product**

CP ← QUOT [ S# ] × LIST [ P# ] is as follows:

| CP | (S# | P#) |
|---|---|---|
| | S4 | P2 |
| | S4 | P5 |
| | S4 | P6 |

This is contained in CAP [ S#, P# ]. The quotient is accordingly the relation QUOT of degree four just shown. The remainder is simply the dividend with some of its rows removed—namely, those appearing in the quotient QUOT with the P# column removed:

| RMDR | (S# | P# | SPEED | UNIT_Q | PRICE) |
|---|---|---|---|---|---|
| | S1 | P1 | 5 | 100 | 10 |
| | S1 | P2 | 5 | 100 | 20 |
| | S1 | P6 | 12 | 10 | 600 |
| | S2 | P3 | 5 | 50 | 15 |
| | S2 | P4 | 5 | 100 | 15 |
| | S3 | P6 | 5 | 10 | 700 |
| | S4 | P3 | 5 | 50 | 17 |

Note the degrees of the relations:

| | |
|---|---|
| Dividend CAP | Degree 5 |
| Divisor LIST | Degree 1 |
| Quotient QUOT | Degree 4 |
| Remainder RMDR | Degree 5 |

More generally, the degree of the quotient is equal to the degree of the dividend minus the degree of the divisor. The degree of the remainder is equal to the degree of the dividend.

**Relational division** is the principal algebraic counterpart of queries that involve the universal quantifier of predicate logic. Now follows the promised and brief explanation of the quantifiers of predicate logic. The example just used to explain **relational division** is now used with more concise notation to explain the two quantifiers: the *existential* and the *universal.*

Suppose that the relations in a database include S, P, and C, where S describes suppliers, P describes parts, and C describes capabilities of suppliers in supplying parts. Let the primary key of suppliers be supplier serial numbers S#; for parts, the primary key is part serial numbers P#; for capabilities, the primary key is the combination of S# and P#. Let the description of suppliers include for each supplier its name; the corresponding column is called SNAME. Suppose also that a list of parts needed for some project is given as a unary relation L whose only column P# draws its values from the part serial number domain P#.

Two quite different kinds of queries are now discussed from the standpoint of a relational language ALPHA [Codd 1971a], which is based on predicate logic, rather than algebra, and uses tuple variables, rather than domain variables. Q1 requires the existential quantifier, and Q2 requires the universal quantifier.

Q1: Retrieve the names of all suppliers, each of whom can supply at least one of the parts listed in L.

```
range of s is S
range of p is L
range of c is C
s, p, and c are tuple variables.
get s.SNAME where
EXISTS p EXISTS c (c.s# = s.s# AND c.p# = p.p#).
```

The term "EXISTS" denotes the existential quantifier of predicate logic. It corresponds to the **theta-join** operators. It does *not* denote the same use of the term "EXISTS" as in the language SQL.

Q2: Retrieve the names of all suppliers, each of whom can supply all of the parts listed in L.

Assuming the same three range statements as listed under Q1,

```
get s.SNAME where
FOR ALL p EXISTS c (c.s# = s.s# AND c.p# =p.p#)
```

The phrase "FOR ALL," which denotes the universal quantifier of predicate logic, corresponds to **relational division.** Present versions of the

language SQL cannot express **relational division** in any direct manner. Consequently, SQL users must translate Q2 into the following:

Q2″: Retrieve the name of every supplier, for whom it is not true that there exists a part in the list L that it cannot supply.

This kind of translation represents a significant burden on users that is completely unnecessary. (Refer back to Figure 4.1 on page 78 for a summation of all the basic operators.)

## 4.3 ■ The Manipulative Operators

The manipulative operators are those concerned with making changes to the contents of the database. Eight such operators are described:

RB-30 **Relational assignment**
RB-31 **Insert**
RB-32 **Update**
RB-33 **Primary key update with cascaded update**
RB-34 **Primary key update with cascaded marking**
RB-35 **Delete**
RB-36 **Delete with cascaded deletion**
RB-37 **Delete with cascaded marking.**

In contrast to pre-relational DBMS, each one of these operators is capable of handling multiple-records-at-a-time, where "multiple records" means zero, one, two, or more rows of a relation. No special treatment is given to any data on account of the number of records.

### *RB-30* Relational Assignment

When querying a database, the user may wish to have the result of the query (a relation, of course) retained in memory under a name of his or her choosing. The user may also wish to be able to require this retained relation to participate in some later relational query or manipulative activity. Both of these desires are satisfied to a certain extent by **relational assignment.** This operator is denoted by ← in the expression T ← rve, where (1) rve denotes a *relation-valued expression* (an expression whose evaluation yields a relation), and (2) T denotes a user-selected name for the relation that is specified by rve and that is to be retained in memory.

Note that the relation obtained by executing rve may, as usual, contain zero, one, two, or more rows.

Since a relation may consist of a very large number of rows, and since each row is likely to consist of a combination of character strings, numbers, and logical truth-values, a **relational assignment** is beyond the capability of most programming languages. However, a fully relational language must be able to express **relational assignment,** while a fully relational DBMS must be able to execute such an assignment.

If the qualifier SAVE is attached to the command, the DBMS establishes the data description of T in the catalog, unless an appropriate description of T is already there. The domain of any column of T in which the values are derived by means of a function is identified as *function-derived,* because the DBMS usually cannot be more specific than that.

When the qualifier SAVE is attached, the user should be required to declare which simple column or combination of simple columns constitutes the primary key.

If the user needs this relation only temporarily (within a particular interactive session or within a single execution of an application program), the qualifier SAVE may be omitted. Then,

1. the DBMS does not record the description of T in the catalog;
2. if T still exists, T is dropped by the DBMS at the end of the interactive session or at the end of execution of the program.

## *RB-31* The Insert Operator

The **insert** operator permits a collection of one or more rows to be inserted into a relation. The user has no control, however, over where these rows go. They may even be appended by the DBMS "at one end or the other" of the target relation. I place this phrase in quotation marks because there is no concept of the end of a relation in the relational model. It is the responsibility of the DBMS alone to determine exactly where the rows should be stored, although this positioning may be affected by the access paths already declared by the DBA for that relation. It is assumed that, for insertion of new rows into a relation T, the catalog already contains a detailed description of T.

If the collection of rows to be inserted includes two or more rows that are duplicates of one another, only one of these rows is inserted. If the collection of rows to be inserted includes a row that duplicates any one of the rows in the receiving relation, that row will not be inserted. Thus, at the end of the insertion the resulting relation contains no duplicate rows; to achieve this, several rows in the collection to be inserted may have been withheld. Whenever rows are withheld by the DBMS from insertion (to

avoid duplicate rows in the result), the *duplicate row indicator* is turned on (see Feature RJ-8 in Chapter 11, "Indicators").

One or more rows in the collection to be inserted may be withheld by the DBMS for another reason: the resulting relation is not allowed to have duplicate values in its primary key. In the event that such a withholding occurs for this reason, the *duplicate primary-key indicator* is turned on (see Feature RJ-9 in Chapter 11). This constraint is more restrictive than the no-duplicate-row constraint, since it is entirely possible that the non-primary-key components may be different from one another, even though the primary-key values are identical.

If one or more indexes exist for the target relation, the DBMS will automatically update these indexes to support the inserted rows.

If the new rows for relation T are derived from one or more other relations in the same relational database in accordance with a relation-valued expression rve, then an alternative way of obtaining the result of inserting these rows into T is by using the **union** operator and **relational assignment:**

T ← T UNION rve.

However, to be able to use this method, T must be either a base relation or a special kind of view—the kind that the DBMS at view-definition time has determined can accept insertions. It is worth noting that not all views can accept insertions, a point discussed in detail in Chapter 17, "View Updatability." It is worth noting that the **insert** operator eliminates duplicate rows and duplicate primary-key values just as the **union** operator does.

## *RB-32* The Update Operator

In managing a database, it may be necessary occasionally to change the values of one or more components of one or more rows that already exist within a relation. This is usually distinguished from inserting entirely new rows because the components to be changed in value may represent a very small percentage of the number of components in each row.

This observation is the justification for the **update** operator. The information that must be supplied with this operator consists of the name of the relation to be updated, the specification of the rows in that relation to be updated, and the column names that identify the row components of these rows to be identified, and the new values for these components. The DBMS should provide two options for identifying the rows to be updated; the user should supply either a list of primary key values or an expression that (1) is a valid condition for a **select** operation and that (2) involves the DBMS in conducting a search for those rows that satisfy this condition.

Existing indexes for the target relation are automatically updated by the DBMS to reflect the requested update activity.

Referential integrity may be damaged if the column to which the update is applied happens to be the primary key of the pertinent relation or a foreign key. Normally, Feature RB-33 should be used to update a primary-key value. When updating a foreign-key value only, the user should make sure that the new value for this key exists as the value of a primary key defined on the same domain. Otherwise, the DBMS will reject the update.

## *RB-33* Primary-Key Update with Cascaded Update of Foreign Keys and Optional Update of Sibling Primary Keys

It is seldom necessary to update the value of a primary key, but, when this is necessary, it is very important that it be done correctly. Otherwise, integrity in the database will be lost, and it will be quite difficult to recover from the damage.

An important check made by the DBMS is that each allegedly new value for a primary key is not only of the data type specified for that key, but is also new with respect to that simple or composite column: that is, at this time the new value does not occur elsewhere in that primary-key column.

When a primary-key value is changed, it is usually necessary to make the same change in value of all of the matching foreign-key values drawn from the same domain. Why cannot this be programmed as a transaction that includes an **update** command for the row that contains the primary key value followed by an **update** command for each of the rows in the database that contain that same value as a foreign key whose domain is the same as that of the primary key? To prepare such a transaction correctly, the user must have extremely recent knowledge of which columns in the entire database draw their values from the domain D of the given primary key; "recent" means down to the millisecond level or some shorter time interval.

It is important to remember that the relational approach is highly dynamic, and that users who are appropriately authorized can at any time request new columns be added to one or more relations. Thus, it would be very risky to assume that any user (even the DBA) knows at any time precisely which columns draw their values from any given domain. It is precisely for this reason that

- the kind of transaction cited in the preceding paragraph is unacceptable;
- the relational model includes the *cascading option* in some of its manipulative operators and in the reaction of the DBMS to attempted violation of certain integrity constraints.

If a DBMS is fully relational, it maintains in the catalog the knowledge concerning which columns draw their values from any given domain in a state that is consistent with the most recently executed relational command.

This means that the DBMS is in a better position than any user to handle correctly the updating of *all* the matching foreign-key values drawn from the pertinent primary domain.

The **primary-key update** command not only updates a primary-key value, but also updates in precisely the same way all of the matching foreign-key values drawn from the same domain as the primary-key value. For the DBMS to support this command, it is essential that the DBMS support domains.

This command is not included in the present version of the language SQL. In fact, it is impossible to express a precisely equivalent action in SQL. Moreover, it is extremely cumbersome to express any action in SQL that is even superficially similar, but is based on the false assumption that some user knows precisely which columns draw their values from a given domain. Such an expression takes about three pages of commands, some expressed in SQL and some in a host language. This is just one of the severe penalties stemming from the failure of SQL to support domains as extended data types (see Chapters 3 and 23).

A more detailed account follows. The **primary-key update** operator is intended to simplify the updating of primary-key values. The DBMS finds from the catalog which domain (say D) is the domain of the specified primary key. It then finds all of the columns in the entire database that draw their values from domain D. From this set of columns, it selects two subsets:

S1:D Those columns that are primary-key columns for other relations, but defined on D

S2:D Those columns that are declared to be foreign-key columns with respect to the given primary key

The set S1:D is called the set of *sibling primary keys*. The set S2:D is called the set of *dependent foreign keys*, where "dependent" refers to the fact that foreign-key values are existence-dependent on their primary key counterparts.

Unless the qualifier EXCLUDE SIBLINGS is attached to the command, the DBMS takes primary-key action as follows. It hunts in each column cited in S1:D for the value of the given primary key. It then updates this value in precisely the same way as the original primary key was updated. Unconditionally, the DBMS takes foreign-key action as follows. It hunts in each column cited in S2:D for the value of the given primary key, now occurring in a foreign-key role. Whenever such a value is found, it is updated in precisely the same way as the primary key was updated. In this way, referential integrity is maintained.

Thus, the DBMS executes all of these primary- and foreign-key updates in addition to the update of the specified primary-key value, except that the action on other primary keys is omitted if the qualifier EXCLUDE SIBLINGS is attached to the pertinent command. Whenever an index involves any of the keys (primary or foreign) being updated, that index is also automatically updated by the DBMS to reflect the updating of the actual key.

The entire sequence of activities is treated as if it had been requested as a transaction. Thus, either the whole series of updates is successful, or none of it is successful. This is what one should expect in any case, since only a single relational command is involved.

Existing indexes for all of the columns of all of the relations involved are automatically updated by the DBMS to reflect the requested update activity. These changes are committed to the database if and only if the aforementioned changes are committed.

## *RB-34* Primary-key Update with Cascaded Marking of Foreign Keys

This operator behaves in the same way as that of Feature RB-33, except in regard to all of the foreign keys based on the same domain as the primary key. Instead of updating the matching foreign-key values, the DBMS marks each foreign-key value as missing-but-applicable. Of course, if one or more of these foreign keys happens to have a DBA-declared constraint that there be no missing values, then the whole command is rejected by the DBMS.

## *RB-35* The Delete Operator

The **delete** operator permits a user to delete multiple rows from a relation: "multiple" includes the special cases of zero and one, and these cases do not receive special treatment. Why include zero as a possibility? One reason is that a condition that the user has incorporated in the **delete** command might not be satisfied by any row. Of course, it is necessary for the user to specify the pertinent relation and identify the rows to be deleted in either of the two ways permitted by the simple **update** operator.

Existing indexes for the target relation are automatically updated by the DBMS to reflect the requested deletion activity.

Users of the **delete** command are advised to be very cautious, since every row of a base relation that is deleted results in the deletion of some primary-key value. Then, if in the database there happen to be matching values of foreign keys, referential integrity can be damaged: hence, the next two commands.

## *RB-36* The Delete Operator with Cascaded Deletion

This **delete** operator is similar to that of Feature RB-35, except that it takes into account the fact that a simple or composite component of each of the rows being deleted happens to be the value of the primary key of a base relation. This is true even if the deletion is executed through a view (a virtual relation). Thus, execution of RB-35 will often violate referential integrity. Since usually referential integrity is not fully checked until the end of a transaction, this violation may be just a transient state that is permitted to exist within the pertinent transaction only. (See Chapter 13 for more details.)

When a primary-key value of a base relation participates in a deletion, referential integrity is normally violated if any foreign keys exist elsewhere in the database, in that relation or in others, that are drawn from the same domain as that primary key and are equal in value to it. There is no violation if the primary-key value still exists as a *sibling primary-key value.* This is the primary key of some other relation, provided that key draws its values from the same domain. Use of the **delete with cascaded deletion** operator causes the DBMS to propagate deletions to those rows in the database that happen to contain dependent foreign-key values as components.

If the qualifier EXCLUDE SIBLINGS is attached to the command, no action is taken with respect to the other occurrences of this value as a sibling primary key from this domain. If this qualifier is not attached, the rows that contain the same value in the role of a sibling primary key are deleted also.

Existing indexes for all of the relations involved are automatically updated by the DBMS to reflect the requested deletion activity.

This operator should be used with great care. In fact, few people in any installation should be authorized to use it; they should probably be on the staff of the DBA. The reason is simple. The deletions can occur in wave after wave, all automatically. (See Chapter 18, "Authorization," for more details.)

The deletion of each row that contains a pertinent foreign-key value must also result in the deletion of a value of some primary key; this primary key will often be different from the primary key that initiated the cascading action. Quite often, the initial deletion of one row results in the deletion of many other rows elsewhere in the database. Then, each of these deletions results in the deletion of many other rows in the database, and so it proceeds.

## *RB-37* The Delete Operator with Cascaded A-marking and Optional Sibling Deletion

This operator is similar to that of Feature RB-36, but is far less dangerous, because the initial cycle of cascading does not trigger any subsequent cycles of deletion. This reduced danger is a strong reason why foreign keys should be allowed to have missing values, unless the DBA has an overriding reason why not.

The DBMS finds all of the columns that draw their values from the domain of the primary key involved: primary domain D. From these columns it selects the two subsets S1:D (the sibling columns) and S2:D (the dependent foreign-key columns) as defined earlier. The DBMS then examines the catalog to see whether any column cited in S2:D has the declaration that missing values are prohibited.

Suppose that one or more of the columns cited in S2:D is of the missing-values-prohibited type. If the value $v$ (say) found in the primary key of the row or rows to be deleted does not occur at all in any of the missing-values-prohibited columns, then execution of the command may proceed. If, however, there is at least one occurrence of the value $v$ in these missing-values-prohibited columns, then the DBMS aborts the deletion and marking altogether. It also turns on an indicator asserting that the deletion has been aborted. If the command participates in a transaction, then the transaction is aborted also.

Assume that the tests just described are satisfactorily passed, and that no abortion occurs. Then, the DBMS searches all of the foreign-key columns cited in S2:D to find all occurrences of the pertinent value and marks each one as missing-but-applicable.

If the qualifier EXCLUDE SIBLINGS is attached to the pertinent command, no action is taken on the columns cited in S1:D. If this qualifier is not attached, however, all of the rows containing the pertinent primary key value in each of the columns cited in S1:D are deleted. Then, regardless of any attached qualifier, all of the columns cited in S2:D that permit missing values are searched for the value $v$, and each such value is A-marked.

Existing indexes for each of the relations involved are automatically updated by the DBMS to reflect the requested deletion activity. Of course, these changes are committed if and only if the aforementioned changes are committed.

In subsequent chapters frequent use is made of the basic operators described in this chapter. A good understanding of these operators is essential to any real understanding of the relational model.

## Exercises

4.1 The connectives LESS THAN, EQUAL TO, GREATER THAN participate in almost every programming language. *In that context* they are often called the relational operators. Are the relational operators of the relational model simply these connectives revisited? If not, explain.

4.2 In the relational model why is row selection called **select,** while column selection is called **project?** Hint: do not confuse the **select** of the relational model with the **select** of SQL.

4.3 Execution of a **join** usually involves comparing values drawn from pairs of columns (each simple or composite) in the database. These are called the comparand columns for this operator.

- What is wrong with requiring for every **join** that the comparand columns in a **join** be identically named?
- What constraints, if any, are placed by the relational model on pairs of comparand columns? Why?

4.4 Is it adequate for the DBMS to check that the comparand columns involved in **joins** contain values of the same basic data type? State reasons for your answer.

4.5 Consider the following query: find the suppliers, each of whom can supply every part in some given list of parts. What relational operator provides the most direct support for this query? Is this a brand-new operator? What is the shortest SQL representation of this request?

4.6 In what sense are the **union, intersection,** and **difference** operators of the relational model different from their counterparts in set theory?

4.7 What is union compatibility? To which of the operators does the relational model apply this as a constraint? Why does the model make union compatibility a requirement in these cases? Can this constraint be overridden?

4.8 In what sense are the operators of the relational model closed? Does this mean that no new operators may be invented? How is this closure useful in the real world?

4.9 What are the sibling primary keys of a given primary key? Does every primary key have a sibling primary key?

4.10 Is updating a primary key any more complicated than updating any other piece of data? If so, state why, and then describe how RM/V2 handles the problem.

4.11 Do present versions of the SQL language handle the problem stated in Exercise 4.10 correctly without requiring the assistance of the host language? Do present versions of SQL handle this problem correctly with assistance from the HL? Explain.

## ■ CHAPTER 5 ■

# The Advanced Operators

The operators discussed in this chapter are intended to meet some practical needs and, in so doing, increase the flexibility and power of the relational model without introducing programming concepts. Any reader who finds this chapter difficult to understand can, and should, skip it on first reading; most of the following chapters are simpler.

The advanced operators include **framing a relation,** the **extend** operator, **semi-join, outer join, outer union, outer difference, outer intersection,** the **T-joins, user-defined selects, user-defined joins,** and **recursive join.** The set of advanced operators is intentionally open-ended. When conceiving extensions, however, it is very important to adhere to the operational closure of all relational operators. See Feature RM-5 in Chapter 12, as well as Chapter 28.

In this chapter, unlike its predecessors, the sample relations that explain each operator involve the use of abstract symbols to denote values. For each column, however, the values must be assumed to be all of one declared data type. For example, all the values may be character strings, integers, floating-point numbers, or the truth-values of some logic. On the other hand, the collection of columns belonging to a relation can have any mixture of these data types. A reader who is unfamiliar with the use of symbols to denote values of these types may wish to take the time to substitute actual values of his or her choosing in place of the symbols, taking care to abide by any domain constraints explicitly mentioned.

Thus, in the example in Section 5.1.2—namely R1 ( K A C D E )—K is intended to be the primary key, so all of its values must be distinct from

one another, and C is specified as a numeric column, so all of its values must be numeric. Thus, a specific case of the relation R1 would be the PARTS relation with K as the part serial number (character-and-digit strings of length 8); A as the part name (strings of characters only, of fixed length 12); C as the quantity-on-hand (modest-sized non-negative integers); D as the quantity-on-order (of the same extended data type as quantity-on-hand); and E as the minimum quantity that should be maintained in inventory (of the same extended data type as quantity-on-hand).

## 5.1 ■ Framing a Relation

### 5.1.1 Introduction to Framing

Occasionally users must partition relations into a collection of subrelations, whose comprehensive union restores the original relation. Each of the subrelations is a member of the partition. A well-known property of a partition is that every pair of these subrelations has an empty intersection.

Consider a relation that contains information about employees in a company. Suppose that it includes a column containing the employee's present annual salary, and another column indicating the department to which that employee is assigned. With this database, consider a request that involves a partition: find the total salaries earned in each department along with the department identification. A user may wish to find for each department the sum of all the present salaries earned by employees in that department, and may want the corresponding totals to appear in one of the columns of the result, along with other columns such as the department number.

Many subrelations may be involved in partitioning a given relation. The approach described next avoids generating these subrelations as a collection of separate relations, for two reasons:

1. each of the subrelations would have to be assigned a distinct name;
2. a new type of operand and result would have to be introduced—namely, a collection of relations. This type appears in RM/T, an earlier extended version of the relational model [Codd 1979], but for other reasons.

Instead of the approach just described, a *frame* is placed on the relation to be partitioned.

### *RZ-1* Framing a Relation

A frame separates the set of rows in any one member of the partition from the set of rows in any other member. This separation is achieved by appending a new column to the relation and, within

this column, assigning a distinct value for each distinct member of the partition. The standard name for this column is FID, or *frame identifier.*

---

In the relational model, in line with the emphasis on basing all operators on explicit values in the database, the act of partitioning is always based on values. In generating partitions, RM/V2 offers the options of using the individual values that occur in a column, simple or composite, or else specified ranges of values that occur therein. Since individual values are simpler to understand, let us consider them first.

### 5.1.2 Partitioning a Relation by Individual Values

The following steps are taken to generate a simple partition of a relation R by changes in values within a simple or composite column C:

1. The DBMS reorders the rows of relation R into ascending order by the values encountered in column C, whether these values are numeric, alphabetic, alphanumeric, or even the truth-values of some logic. (In the last case, the ascending sequence is FALSE, TRUE, MAYBE-AND-APPLICABLE, and MAYBE-BUT-INAPPLICABLE pending the establishment of a standard.);
2. The DBMS appends the new frame-identifier column FID to R. The initial value in the frame identification column (FID) is 1; this value is increased by one each time that a distinct value in C is encountered in the ascending order cited in Step 1.

In the case of alphabetic and alphanumeric columns, the DBMS uses some standard collating sequence for ordering purposes.

The result is a single relation with a frame that represents partitioning of R according to the distinct values in C. The frame is identified by the integers in column FID. Let R /// C denote relation R framed according to column C.

For example, relation R2 is R1 framed according to column C in the simple sense just described. The dotted lines portray the frame. In this example the pivotal column C happens to contain numeric values. Those readers who like "real" examples can assume the following denotations:

R1 PARTS relation
K Part serial number
A Part name
C Quantity on hand
D Quantity on order
E Quantity for triggering reorder

R2 = R1 /// C

| R1 (K | A | C | D | E ) |
|---|---|---|---|---|
| k1 | a1 | 13 | d1 | e3 |
| k2 | a1 | 9 | d2 | e7 |
| k3 | a2 | 37 | d1 | e2 |
| k4 | a3 | 24 | d2 | e6 |
| k5 | a3 | 13 | d3 | e1 |

| R2 (K | A | C | D | E | FID) |
|---|---|---|---|---|---|
| k2 | a1 | 9 | d2 | e7 | 1 |
| k5 | a3 | 13 | d3 | e1 | 2 |
| k1 | a1 | 13 | d1 | e3 | 2 |
| k4 | a3 | 24 | d2 | e6 | 3 |
| k3 | a2 | 37 | d1 | e2 | 4 |

Column FID identifies the interval and makes it unnecessary for the DBMS to keep the row ordering illustrated. Note that relation R1 has five tuples, but that column C has just four distinct values. Consequently, R1 framed according to C by individual values has precisely four members. Of course, each of these four members is a subrelation, which is a set. Three of the members of the partition are sets consisting of just one tuple, while the fourth member is a set containing two tuples.

### 5.1.3 Partitioning a Relation by Ranges of Values

A more complicated partitioning involves a sequence of ranges of values in the pivotal column C. Suppose the desired ranges for this new partitioning are as follows:

1–10, 11–20, 21–30, 31–40, and so on.

This sequence could be expressed more concisely as follows:

Begin at 1; the range interval is 10.

When the range interval is not constant, a ternary relation may be used as a listing of all the range intervals. For example,

| RANGE (FROM | TO | FID) |
|---|---|---|
| 1 | 11 | 1 |
| 12 | 25 | 2 |
| 26 | 32 | 3 |
| 33 | 48 | 4 |

Note that in such a table it is required that the intervals do not overlap one another.

Thus, a user may wish to request the DBMS to use the RANGE relation (any name that satisfies the naming features will do) for determining the

starting value and intervals. Now, a different result R3 is generated. Once again, the dotted lines portray the frame.

R3 = R /// C per RANGE

| R3 | (K | A | C | D | E | FID) |
|---|---|---|---|---|---|---|
| | k2 | a1 | 9 | c2 | 87 | 1 |
| | k5 | a3 | 13 | c3 | 81 | 2 |
| | k1 | a1 | 13 | c1 | 93 | 2 |
| | k4 | a3 | 24 | c2 | 76 | 2 |
| | k3 | a2 | 37 | c1 | 52 | 4 |

Column FID identifies the interval and makes it unnecessary for the DBMS to keep the row ordering illustrated. Note that the values in FID determine membership in various elements of the partition. Thus, there is no need for the DBMS to preserve the ascending ordering based on column FID as illustrated in the preceding table.

### 5.1.4 Applying Aggregate Functions to a Framed Relation

Assume that the relation R1 discussed in Section 5.1.2 is framed on column C according to the ternary relation RANGE discussed in Section 5.1.3. Let the result be R3. Normally, applying the function SUM to column E in any of the relations R1, R2, R3 (whether framed or not) yields the sum of all the values in column E. If, however, a relational command requests that the function SUM be applied to column E of either relation R2 or R3 *according to the frame implied by column FID,* then SUM is applied to each member of the partition; that can yield as many resulting values as there are distinct values in column FID.

| SUM | R1.E |
|---|---|
| | 389 |

| SUM-per-FID | R2.E |
|---|---|
| | 87 |
| | 174 |
| | 76 |
| | 52 |

| SUM-per-FID | R3.E |
|---|---|
| | 87 |
| | 250 |
| | 52 |

Of course, it is quite likely that in the third case the user would like to have the pertinent range from the RANGE relation with each of the three totals. This can easily be accomplished by requesting column R3.FID along with the SUM according to FID(R3.E), and then requesting either the **natural join** or the **equi-join** of this relation, with the RANGE relation, using the pair of FID columns (one from each of the operand relations) as the comparand columns.

The result is relation R4:

| R4 | (FROM | TO | FID | SF(R3.E)) |
|---|---|---|---|---|
| | 1 | 11 | 1 | 87 |
| | 12 | 25 | 2 | 250 |
| | 33 | 48 | 4 | 52 |

SF denotes the function SUM-per-FID.

The following example illustrates partitioning and applying an aggregate function to the members of the partition. Assume that the following base relation provides the identification EMP# and immediate properties of employees:

| EMP1 | (EMP# | ENAME | DEPT# | SALARY | H_CITY ) |
|---|---|---|---|---|---|
| | E107 | Rook | D12 | 10,000 | Wimborne |
| | E912 | Knight | D10 | 12,000 | Poole |
| | E239 | Knight | D07 | 12,000 | Poole |
| | E575 | Pawn | D12 | 11,000 | Poole |
| | E123 | King | D01 | 15,000 | Portland |
| | E224 | Bishop | D07 | 11,000 | Weymouth |

Consider the following two steps:

1. Partition the relation EMP1 according to the DEPT# column:

   EMP2 ← EMP1 /// DEPT#

| EMP2 | (EMP# | ENAME | DEPT# | SALARY | H_CITY | FID) |
|---|---|---|---|---|---|---|
| | E123 | King | D01 | 15,000 | Portland | 1 |
| | E224 | Bishop | D07 | 11,000 | Weymouth | 2 |
| | E239 | Knight | D07 | 12,000 | Poole | 2 |
| | E912 | Knight | D10 | 12,000 | Poole | 3 |
| | E107 | Rook | D12 | 10,000 | Wimborne | 4 |
| | E575 | Pawn | D12 | 11,000 | Poole | 4 |

2. Find for each department the department serial number and the total salary earned by all employees assigned to that department:

DSAL(DEPT#, TOTSAL) ← EMP2(DEPT#, SUM-per-FID(SALARY))

| DSAL | (DEPT# | TOTSAL) |
|---|---|---|
| | D01 | 15,000 |
| | D07 | 23,000 |
| | D10 | 12,000 |
| | D12 | 21,000 |

Although the GROUP BY feature of SQL is quite concisely expressed, it is neither as powerful nor as flexible as the framing feature of RM/V2.

## 5.2 ■ Auxiliary Operators

One reason to introduce the auxiliary operators here is to keep the definition of the three outer set operators in the next section reasonably concise. Another reason is that these operators can be useful in other contexts.

Common to all three outer set operators—**outer union, outer difference,** and **outer intersection**—is an initial step that makes the two operands union-compatible. This step is explained by introducing the operator discussed in Section 5.2.1. (see [GOOD]).

### 5.2.1 Extending a Relation

---

**_RZ-2_ Extend the Description of one Relation to Include all the Columns of Another Relation**

The relation cited first in the command is the one whose description is altered to include all the columns of the second-cited relation that are not in the first. The columns thus introduced into the first relation are filled with A-marked values, unless the VALUE qualifier RQ-13 (see Chapter 10) is applied to specify a particular value.

---

Considerable care must be taken in using the **extend** operator. A column of one of the operands may have the same name and certain other properties (such as the domain) as a column of the second operand, but the two columns may have different meanings. Such columns are called *homographs* of one another.

The **extend** operator may not be able to distinguish between the different meanings, and may incorrectly assume that they are identical. The only known solution to this problem is for the DBA to be continually concerned about the possibility of homographs and try to avoid them altogether. Homographs can be deadly in other contexts also.

It is possible, although unlikely, for the DBMS to discover that no new columns need be added to the first-cited relation. To make a pair of relations (say S and T) mutually union-compatible, it is normally necessary to extend the columns of both relations by requesting

St ← S **per** T and Ts ← T **per** S,

where "**per**" denotes the **extend** operator. Note that, in general, union compatibility is attained only after two applications of the **extend** operator.

Two applications of this kind constitute the first step in each of the outer set operators.

This **extend** operator is used in defining the **outer joins** and the **outer set** operators. Of course, it may be used independently of these operators.

It is quite common for some banks to record accounts in more than one way. For example, the following two types might have been established (they are simpified for exposition):

| | | | | |
|---|---|---|---|---|
| | | | % | $ |
| ACCOUNT | (ACCOUNT# | NAME | INTEREST_RATE | BALANCE) |
| | | | | $ |
| ACCT | (ACCOUNT# | NAME | MATURITY_DATE | BALANCE) |

The first type is unique in having INTEREST_RATE as a column, while the second is unique in having MATURITY_DATE as a column. Thus, these two relations are *not* union-compatible. They can be converted into a pair of relations that are union-compatible by applying the **extend** operator to each one.

A1 ← ACCOUNT EXTEND per ACCT

A2 ← ACCT EXTEND per ACCOUNT

Both A1 and A2 have as their columns ACCOUNT#, NAME, MATURITY_DATE, INTEREST_RATE, and BALANCE. The relation A1 UNION A2 may be exactly what the headquarters planning staff needs for analysis and planning purposes.

An unlikely special case, which is not given special treatment, does not permit the operands (relations S and T) to have any domains in common. Hence, there are no comparable columns at all.

### 5.2.2 The Semi-theta-join Operator

The idea for the **semi-join** operator has been circulating for many years; it is not clear who originated the concept. One discussion is found in a paper by Bernstein and Chiu [1981]. In this section, a slight generalization of the **semi-join** operator is discussed: the EQUALITY comparator that is normally assumed is replaced by **theta,** where "theta" can be any one of the 10 comparators:

1. EQUALITY
2. INEQUALITY
3. LESS THAN
4. LESS THAN OR EQUAL TO
5. GREATER THAN

6. GREATER THAN OR EQUAL TO
7. GREATEST LESS THAN
8. GREATEST LESS THAN OR EQUAL TO
9. LEAST GREATER THAN
10. LEAST GREATER THAN OR EQUAL TO

Once again, it is important to recall that the relational model provides a very important safety feature, first cited in Chapter 3:

### *RT-1* Safety Feature when Comparing Database Values

When comparing a database value in one column with a database value in another, the DBMS merely checks that the two columns draw their values from a common domain, unless the domain check is overridden (see Feature RQ-9 in Chapter 10). When comparing (1) a computed value with a database value or (2) one computed value with another computed value, however, the DBMS checks that the basic data types (not the extended data types) are the same.

Several of the advanced operators involve comparing of database values. The following operators are examples of this.

Let n = 3,4, . . . ,12. Then the RZ feature with n as its number is a semi-theta-join that makes use of the comparator numbered n-2 in the list of comparators cited at the beginning of this section.

### *RZ-3 through RZ-12* Semi-Theta-Join

Suppose that the operands of a **theta-join** are S and T, where **theta** is any one of the 10 comparators listed earlier, and the columns to be compared are simple or composite column A of S with simple or composite column B of T. Suppose that relation T is projected onto column B. The result of this projection contains only those values from B that are distinct from one another. The **semi-join** of S on A with T on B yields that subrelation of S whose values in column A are restricted to just those that qualify in accordance with the comparator **theta** with respect to the projection of T onto B.

Suppose that, when **theta** happens to be EQUALITY, the operator **semi-theta-join** is denoted by **sem=**. Then, S **sem=** T is that subrelation of S containing all of the rows of S that match rows of T with respect to the

comparand columns. The remaining rows of S are those that fail to match any row of T in accordance with the comparand columns. When the comparand columns are not explicitly specified (as in the preceding case), the DBMS assumes that the set of these columns is maximal with respect to the given operands, disregarding keyhood. (Incidentally, I use the term "match" only when values are being compared for equality.)

The **semi-join** can be useful when relations S and T happen to be located at different sites as part of a distributed database. Suppose that relation T has many more rows than relation S. Then, the load on the communication lines between the site containing S and the site containing T can often be reduced by (1) transmitting the projection of S onto A to the site containing T, (2) executing the semi-join of T with S [A] at this site, yielding a subset of relation T, and then (3) transmitting this subset of relation T back to the site containing S for the full **join** to be completed there.

It is the responsibility of the optimizer in the DBMS to select this method of handling a **join,** whenever it happens to be the most efficient. Such a selection certainly should not be a burden on end users or on application programmers.

The following example shows the **semi-join** operator in action. The operands are as follows:

| S | (EMP# | ENAME | H_CITY ) |
|---|---|---|---|
| | E107 | Rook | Wimborne |
| | E912 | Knight | Poole |
| | E239 | Pawn | Poole |

| T | (WHSE# | W_CITY ) |
|---|---|---|
| | W17 | Wareham |
| | W34 | Poole |
| | W92 | Poole |

Consider this query: find the employee information and the warehouse information for every case in which an employee's residence is in the same city as a company warehouse. The formula for the **join** is

$$U \leftarrow S\ [\ H\_CITY\ =\ W\_CITY\ ]\ T,$$

and the result obtained assuming the values just cited is as follows:

| U | (EMP# | NAME | H_CITY | WHSE# | W_CITY) |
|---|---|---|---|---|---|
| | E912 | Knight | Poole | W34 | Poole |
| | E912 | Knight | Poole | W92 | Poole |
| | E239 | Pawn | Poole | W34 | Poole |
| | E239 | Pawn | Poole | W92 | Poole |

## 5.3 ■ The Outer Equi-join Operators

Several operators referred to as the **outer join** are supported in the relational model. The **join** operators introduced in Chapter 4 are henceforth called **inner joins,** when a need arises to refer to them as a collection.

The **outer joins** are based on a proposal made in 1971 by Ian Heath, then of the IBM Hursley Laboratory in England [Heath 1971]. In this section and Section 5.4, examples of the **inner** and **outer equi-join** with and without the MAYBE qualifier are described, and the close relationship of the **inner** and **outer joins** is explained. The MAYBE qualifier pertains to the four-valued logic supported by a relational DBMS, assuming its fidelity to the model with respect to the treatment of missing information. This qualifier is discussed in detail in Chapters 8, 9, and 10.

The following two simple relations are used as sample operands in explaining the **outer join** operators. Notice that values in column B of relation S are to be compared with values in column C of relation T. More concisely, S.B and T.C are the comparand columns. Further, b1 occurs in S.B but not in T.C, while b4 occurs in T.C but not in S.B.

| S (A | B ) |
|---|---|
| a1 | b1 |
| a2 | b2 |
| a3 | b3 |

| T (C | D ) |
|---|---|
| b2 | d1 |
| b2 | d2 |
| b3 | d3 |
| b4 | d4 |

Columns S.B and T.C draw their values from the same domain. Thus, it is meaningful to compare values from S.B with those from T.C.

There are three kinds of **outer joins: left outer equi-join** (Feature RZ-13), **right outer equi-join** (Feature RZ-14), and **symmetric outer equi-join** (Feature RZ-15).

## *RZ-13* Left Outer Equi-join

The **left outer join** of S on B with T on C, denoted U = S [ B /= C ] T, is defined in terms of the inner **equi-join** (IEJ) and the left outer increment (LOI). LOI is defined as follows: pick out those tuples from S whose comparand values in the comparand column S.B do not participate in the **inner join,** and append to each such tuple a tuple of nothing but missing values and of size compatible with T.

In more formal terms,

LOI = (S − IEJ [A, B]) **per** IEJ.

Then, U is defined by

U = IEJ ∪ LOI.

Its extension is as follows:

| U | (A | B | C | D ) | |
|---|---|---|---|---|---|
| | a1 | b1 | — | — | } left outer increment |
| | a2 | b2 | b2 | d1 | } inner equi-join |
| | a2 | b2 | b2 | d2 | |
| | a3 | b3 | b3 | d3 | |

## *RZ-14* Right Outer Equi-join

The **right outer join** of S on B with T on C, denoted V = S [ B = \ C ] T, is defined in terms of the **inner equi-join** (IEJ) and the right outer increment (ROI). ROI is defined as follows: pick out those tuples from T whose comparand values in the comparand column T.Y do not participate in the inner join, and append to each such tuple a tuple of nothing but missing values and of size compatible with S.

In more formal terms,

ROI = (T − IEJ [B, C]) **per** IEJ.

V is then defined by

V = IEJ ∪ ROI.

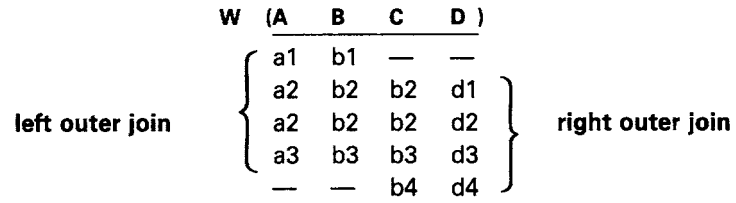
Its extension is as follows:

| V | (A | B | C | D ) | |
|---|---|---|---|---|---|
| | a2 | b2 | b2 | d1 | } inner equi-join |
| | a2 | b2 | b2 | d2 | |
| | a3 | b3 | b3 | d3 | |
| | — | — | b4 | d4 | } right outer increment |

## *RZ-15* Symmetric Outer Equi-join

The **symmetric outer equi-join** of S on B with T on C, denoted W = S [ B /=\ C ] T, is defined by W = LOI **union** IEJ **union** ROI. This implies that W = U **union** V.

Its extension is as follows:




Note that this particular result contains an A-marked value in every column, which is not necessarily true of other examples of **symmetric outer join.** Also note that the following identity holds for **outer join** results:

**outer join = left outer join ∪ right outer join.**

It is important to observe that the result of a **symmetric outer join** is likely to contain one or more missing values in each and every column, although no single row contains a missing value in every column. This is why such a result cannot have a primary key that satisfies the entity-integrity rule—namely, that a primary key must have no missing values (see Chapter 13).

Because duplicate rows are prohibited in every relation of the relational model, however, it is still true that every row is distinct from every other row in the result of a **symmetric outer join.** Thus, for every relation R of this type, an identifier is defined that consists of every column of R; this is called the *weak identifier* of R.

Even though **outer joins** are not well supported in many of today's DBMS products, they are frequently needed and heavily used. Consider the following example.

A database contains information about suppliers and shipments received from these suppliers. The supplier relation S contains the serial number S# of all of the suppliers in the database, their names SNAME, and other immediate properties. The shipment relation SP contains the serial number S# of the supplier making each shipment, the serial number P# of the part shipped, the date SHIP_DATE the shipment was received, and other immediate properties of the shipment, such as the quantity received SHIP_Q.

A company executive requests a report listing all the shipments received in the first six months of 1989. The report must include, for each shipment, the supplier's serial number and name, together with the serial number of the part, the quantity shipped, and the date received. The executive also requests that the report include those suppliers on record in the database from which the company received no shipments at all, accompanied by an indication that each one shipped nothing at all during the specified period. Such a request can be expressed in terms of a **left outer join.**

Suppose that the supplier relation S is as shown below, and relation SP″ is derived from the shipment relation SP by row selection, retaining exclu-

sively those rows that pertain to shipments with SHIP_DATE between the dates 89-01-01 and 89-06-30 inclusive.

| S | S# | SNAME | . . . |
|---|---|---|---|
| | s1 | Jones | . . . |
| | s2 | Smith | . . . |
| | s3 | Clark | . . . |
| | s4 | Rock | . . . |
| | s5 | Roth | . . . |

| SP″ | S# | P# | SHIP_DATE | SHIP_Q |
|---|---|---|---|---|
| | s1 | p1 | 89-05-31 | 1000 |
| | s1 | p2 | 89-03-20 | 575 |
| | s2 | p7 | 89-02-19 | 150 |
| | s4 | p2 | 89-06-15 | 900 |
| | s4 | p4 | 89-04-07 | 250 |
| | s4 | p8 | 89-02-28 | 650 |

The **left outer join** of S on S# with SP on S# is denoted,

$$SSP \leftarrow S\ [S\# \not= S\#]\ SP,$$

in which the relation S is the left operand. Alternatively, the operands may be switched and the **right outer join** may be used,

$$SSP \leftarrow SP\ [S\# =\backslash S\#]\ S,$$

in which the relation S is the right operand.

In either case the extension of SSP is as follows:

| SSP | . . . | SNAME | S# | S# | P# | SHIP_DATE | SHIP_Q |
|---|---|---|---|---|---|---|---|
| | . . . | Jones | s1 | s1 | p1 | 89-05-31 | 1000 |
| | . . . | Jones | s1 | s1 | p2 | 89-03-20 | 575 |
| | . . . | Smith | s2 | s2 | p7 | 89-02-19 | 150 |
| | . . . | Clark | s3 | — | — | — | — |
| | . . . | Rock | s4 | s4 | p2 | 89-06-15 | 900 |
| | . . . | Rock | s4 | s4 | p4 | 89-04-17 | 250 |
| | . . . | Rock | s4 | s4 | p8 | 89-02-28 | 650 |
| | . . . | Roth | s5 | — | — | — | — |

Note that, to conform with the request, the suppliers Clark and Roth in this report have designations of missing items in the shipments-half of their rows—no shipment was received from either of these suppliers in the first six months of 1989.

In Section 17.5.4, it is argued that, in certain circumstances, the **outer equi-join** operator is clearly superior as a view to its inner counterpart. I am confident that this use of the **outer equi-join** was not conceived when the operator was invented.

## 5.4 ■ Outer Equi-joins with the MAYBE Qualifier

A common characteristic of real databases is that values are missing in various rows and columns for a variety of reasons. As a result, a DBMS

that has only two truth values (TRUE and FALSE) designed into it may be unable to determine in a non-guessing mode the truth value of a truth-valued expression in the condition part of a relational request. A relational DBMS that supports all of the features of RM/V2 has four truth values designed into it:

TRUE (t), FALSE (f), MAYBE-APPLICABLE (*a*),

and

MAYBE-INAPPLICABLE (*i*)

Both of the latter two truth values reflect the fact that missing data can make it impossible for the DBMS to determine whether the truth value is TRUE or FALSE. These truth values are distinguished by whether a value is missing but applicable (simply unknown at this time) or missing and inapplicable (e.g., the sales commission earned by an employee who is not a salesman).

While describing the **outer equi-joins,** it is worthwhile to consider the effect of the MAYBE qualifier on the operator. The MAYBE qualifier should be distinguished from the MAYBE truth values. For data to be retrieved, it is normally required that the specified condition evaluate to TRUE. The MAYBE qualifier alters the truth value that is required for data to be retrieved. The alteration is from the truth value TRUE to one of the MAYBE truth values (see Chapters 8 and 10 for more details). In other words, the data retrieved is that for which the condition is evaluated to be neither TRUE nor FALSE.

Let a1 and b3 in S be missing but applicable (A-marked). Let d1 and the second occurrence of b2 in T be missing but applicable (A-marked). Thus, the new operands are as follows:

| S″ (A | B ) |
|---|---|
| — | b1 |
| a2 | b2 |
| a3 | — |

| T″ (C | D ) |
|---|---|
| b2 | — |
| — | d2 |
| b3 | d3 |
| b4 | d4 |

The 12 possible comparisons between the values from S″.B and the values from T″.C have the following truth-values:

| S″.B | b2 | b1 | b2 | — | — | — | — | b1 | b1 | b1 | b2 | b2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T″.C | b2 | — | — | — | b2 | b3 | b4 | b2 | b3 | b4 | b3 | b4 |
| truth | t | m | m | m | m | m | m | f | f | f | f | f |

where t, f, m respectively denote the truth values TRUE, FALSE, and MAYBE.

The **symmetric outer equi-join** of S on B with T on C accompanied by the MAYBE qualifier is denoted

U″ = S″ [ B/=\C ] T″ MAYBE.

Its extension is as follows:

| U″ | (A | B | C | D ) |
|---|---|---|---|---|
| | — | b1 | — | d2 |
| | a2 | b2 | — | d2 |
| | a3 | — | b2 | — |
| | a3 | — | — | d2 |
| | a3 | — | b3 | d3 |
| | a3 | — | b4 | d4 |

Note that < a2 b2 b2 — > is the only tuple that belongs to the **inner equi-join** with no MAYBE qualifier. It does *not* belong to either the **inner** or **outer equi-join** with the MAYBE qualifier.

In this example, the **outer join** with MAYBE happens to be equal to the **inner join** with MAYBE. In other words, the left outer increment and the right outer increment happen to be empty in the MAYBE case. In the next example, all four of the following results are distinct:

**inner equi-join** TRUE, **inner equi-join** MAYBE,
**outer equi-join** TRUE, **outer equi-join** MAYBE.

To provide some additional explanation of these operators, consider the **outer equi-join** of S on X with T on Y, where the comparand columns are S.X and T.Y.

The increment over the **inner equi-join** contributed by the *left outer increment* (LOI) is defined as follows:

> LOI with or without the MAYBE qualifier: pick out those tuples from S whose comparand values in the comparand column S.X do not participate in the **inner join,** and append to each such tuple a tuple of nothing but missing values and of size compatible with T.

Non-participation of a comparand value in the MAYBE case means that no comparison involving that value yields the truth-value m. Non-participation in the true case (reflected by the absence of the MAYBE qualifier) means that no comparison involving that value yields the truth value t.

The increment over the **inner equi-join** contributed by the **right outer increment** (ROI) is defined as follows:

> ROI with or without the MAYBE qualifier: pick out those tuples from T whose comparand values in the comparand column T.Y do not participate in the **inner join,** and append to each such tuple a tuple of nothing but missing values and of size compatible with S.

Now for the promised example. Assume the operands are as follows:

| S2 (A | B ) |
|---|---|
| a1 | — |
| a2 | b2 |
| a3 | b3 |

| T2 (C | D ) |
|---|---|
| b2 | d1 |
| b2 | d2 |
| b3 | d3 |
| b4 | d4 |

The 12 possible comparisons between the occurrences of values in S2.B and T2.C have the following truth-values:

| S2.B | b2 | b2 | b3 | — | — | — | — | b2 | b2 | b3 | b3 | b3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| T2.C | b2 | b2 | b3 | b2 | b2 | b3 | b4 | b3 | b4 | b2 | b2 | b4 |
| truth | t | t | t | m | m | m | m | f | f | f | f | f |

The results obtained by applying the four operators are as follows:

**Inner equi-join TRUE**

| | | | |
|---|---|---|---|
| a2 | b2 | b2 | d1 |
| a2 | b2 | b2 | d2 |
| a3 | b3 | b3 | d3 |

**Inner equi-join MAYBE**

| | | | |
|---|---|---|---|
| a1 | — | b2 | d1 |
| a1 | — | b2 | d2 |
| a1 | — | b3 | d3 |
| a1 | — | b4 | d4 |

**Outer equi-join TRUE**

| | | | | |
|---|---|---|---|---|
| a1 | — | — | — | } LOI |
| a2 | b2 | b2 | d1 | } inner |
| a2 | b2 | b2 | d2 | |
| a3 | b3 | b3 | d3 | |
| — | — | b4 | d4 | } ROI |

**Outer equi-join MAYBE**

| | | | | |
|---|---|---|---|---|
| a2 | b2 | — | — | } LOI |
| a3 | b3 | — | — | |
| a1 | — | b2 | d1 | } inner |
| a1 | — | b2 | d2 | |
| a1 | — | b3 | d3 | |
| a1 | — | b4 | d4 | |

Note that ROI is empty in the **outer equi-join** with MAYBE. All four of these relations are distinct.

## 5.5 ■ The Outer Natural Joins

Consider two relations S and T that happen to have extensions as follows:

| S (P | A ) |
|---|---|
| k1 | a1 |
| k2 | a2 |
| k3 | a2 |
| k4 | a3 |

| T (Q | B ) |
|---|---|
| m1 | a2 |
| m2 | a2 |
| m3 | a4 |

Suppose that columns S.A and T.B draw their values from a common domain, and it is therefore meaningful to compare values from one column

with values from the other. Consider two kinds of joins: the **symmetric outer natural join** of S on A with T on B, and the **symmetric outer equi-join** of S on A with T on B.

U = S [ A /*\ B ] T
V = S [ A /=\ B ] T

**Symmetric outer natural join**

U (P AB Q)

| | | | | |
|---|---|---|---|---|
| | k1 | a1 | — | |
| | k4 | a3 | — | |
| **left outer join** | k2 | a2 | m1 | |
| | k3 | a2 | m1 | **right outer join** |
| | k2 | a2 | m2 | |
| | k3 | a2 | m2 | |
| | — | a4 | m3 | |

**Symmetric outer equi-join**

V (P A B Q)

| | | | |
|---|---|---|---|
| k1 | a1 | — | — |
| k4 | a3 | — | — |
| k2 | a2 | a2 | m1 |
| k3 | a2 | a2 | m1 |
| k2 | a2 | a2 | m2 |
| k3 | a2 | a2 | m2 |
| — | — | a4 | m3 |

In table U, the **left outer** and **right outer natural joins** are shown as subrelations of the **symmetric outer natural join.** The three **outer natural joins** are defined constructively (Features RZ-16–RZ-18)—that is, in terms of an algorithm that will generate the appropriate result. An implementation can make use of this algorithm, but is not required to do so. It is only necessary that the implementation generate the same result as the defining algorithm.

## *RZ-16* Left Outer Natural Join

First, form the **inner natural equi-join** W of S on A with T on B. Then, form the **relational difference** W1 = S − W [P, A]. Then, **extend** W1 **per** S to yield W2. Finally, form the **left outer natural join** LONJ = W **union** W2.

## *RZ-17* Right Outer Natural Join

First, form the **inner natural equi-join** W of S on A with T on B. Then, form the **relational difference** W3 = T − W [A, Q]. Then, **extend** W3 **per** S to yield W4. Finally, form the **right outer natural join** RONJ = W **union** W4.

## *RZ-18* Symmetric Outer Natural Join

First, form W and W2 as in the first three steps of Feature RZ-16. Then, form W4 as in the first three steps of Feature RZ-17. Finally,

form the **symmetric outer natural join** by taking the **union:** SONJ = W2 **union** W **union** W4. Alternatively, **symmetric outer join** = LONJ **union** RONJ. Note that **union** in the relational model always includes removal of duplicate rows from the result.

---

It may be recalled that the **inner natural join** is a simple projection of the **inner equi-join,** in which one of two mutually redundant comparand columns is removed. The **outer joins,** however, are not related to one another so simply.

The columns in the **outer equi-join** that stem from the comparand columns in the operands are not necessarily mutually redundant columns. In fact, in this example columns A and B are clearly *not* mutually redundant. Thus, the **outer natural join** is not necessarily a projection of the **outer equi-join**—a fact that may decrease the usefulness of the **outer natural join.**

## 5.6 ■ The Outer Set Operators

In this section I define the three outer set operators in the relational model—**union, set difference,** and **set intersection**—and compare them with their inner counterparts. A close correspondence is shown to exist between an identity that pertains to the inner operators and one that pertains to the outer operators.[1]

### 5.6.1 The Inner Operators Revisited

In the relational model, each of the inner set operators—**union, set difference,** and **set intersection**—is applied exclusively to a pair of relations of precisely the same type. In other words, between the two operands (relations S and T, say) there must exist a one-to-one correspondence between the columns of S and the columns of T, such that each of the pair of columns in this correspondence draws its values from a common domain. Any pair of relations that are of precisely the same type are said to be *union-compatible.* When restricted in this way, these operators are called **relational union, relational difference,** and **relational intersection,** respectively.

This correspondence must be specified in the expression that invokes the pertinent relational operator. The reason is as follows. The domains from which the columns of S and T draw their values are, in general, inadequate to establish such a correspondence, because two or more of the columns of either S or T may draw their values from the same domain.

1. I am grateful to Nathan Goodman [1988], who contributed to the definitions in their final form.

An important relationship called the *inner identity* holds between these three inner relational operators:

$$(S \cup T) = (S - T) \cup (S \cap T) \cup (T - S)$$

for any relations S and T that are union-compatible, where the minus sign denotes relational difference.

### 5.6.2 The Outer Set Operators

With the inner set operators, the operands S and T are required to be union-compatible, that is, of exactly the same type. One important reason for the outer set operators is to allow the operands S and T to differ somewhat in type, and even in degree. Thus, S may include columns not found in T, and T may include columns not found in S.

In the context of the inner set operators, two rows (one from S, the other from T) are duplicates of one another if there is pairwise equality between corresponding components. In the context of the outer set operators, however, equality between a row in S and a row in T is seldom encountered, because S and T are not required to be union-compatible. Therefore, it is necessary to include the following concept, which is more general than row equality.

A row from relation S is a *close counterpart* of a row from relation T if all the following conditions hold:

- the operands S and T have primary keys defined on the same domain;
- the two rows (one from S, one from T) have equal primary-key values;
- pairwise equality in non-missing values holds for those properties of S that correspond to properties of T.

This concept is heavily used in the outer set operators: **union, difference,** and **intersection.**

The outer set operators are potentially important in distributed database management. For example, consider a bank that stores customer accounts in a distributed database. Suppose that customer accounts are represented using logical relations of different types in different cities or in different states. The differences may be slight, or may be quite significant. An extreme case, not likely to be found in banks, and not part of this example, is that S and T have no domains at all in common. In the discussion following Feature RZ-19, the bank example is pursued in more detail with explicit data.

For each of the three outer set operators, a precise definition is presented followed by an example and some informal discussion. The definitions of the outer operators are crafted so that the identity cited for the three inner operators applies also to the outer operators.

The following sample relations are used as operands to illustrate the outer set operations:

| S (A | B) |
|---|---|
| a1 | b1 |
| a1 | b2 |
| a2 | b3 |

| T (E | C) |
|---|---|
| a2 | c3 |
| a3 | c4 |

| T″ (E | C) |
|---|---|
| a1 | c1 |
| a1 | c2 |
| a2 | c3 |
| a3 | c4 |

For generality, columns A, B, C, and E may be either simple or composite. B is assumed to consist of all—and nothing but—the simple columns whose domains do not occur in T. C is similarly assumed to consist of all—and nothing but—the columns whose domains do not occur in S. Thus, A consists of all—and nothing but—the simple columns whose domains occur in both S and T. A similar remark applies to column E.

In the examples, either the pair S and T or the pair S and T″ is used as the two operands. All of the columns S.A, T.E, T″.E draw their values from a common domain, whether simple or composite. Columns B and C draw their values from domains that are different from one another and from the domain of A.

## *RZ-19* Outer Union

Suppose the operands of **outer union** are S and T. As the first step, apply the **extend** operator to both S and T: **extend** S **per** T and call it St; **extend** T **per** S and call it Ts. Now, St and Ts are of the same degree, and each contains columns based on all the domains in S and all the domains in T. In fact, St and Ts are completely union-compatible. As the second and final step, form St **union** Ts, which yields the **outer union** S\U/T.

The **outer union** S\U/T of relation S with relation T is generated by means of the following three steps:

1. form St = S **per** T;
2. form Ts = T **per** S;
3. form S \U/ T = St ∪ Ts.

The close-counterpart concept (see p. 116 for its definition) is used instead of row equality to remove duplicate rows.

By definition,

S \U/ T = (S **per** T) ∪ (T **per** S).

And clearly,

S \U/ T = T \U/ S.

Take the sample relations as operands, and apply the **outer union.** The results are as follows:

| S \U/ T (A | B | C) |
|---|---|---|
| a1 | b1 | — |
| a1 | b2 | — |
| a2 | b3 | — |
| a2 | — | c3 |
| a3 | — | c4 |

| S \U/ T″ (A | B | C) |
|---|---|---|
| a1 | b1 | —(1) |
| a1 | b2 | — |
| a2 | b3 | — |
| a1 | — | c1(2) |
| a1 | — | c2 |
| a2 | — | c3 |
| a3 | — | c4 |

Note that rows of S\U/T″ marked (1) and (2) in the preceding R-table are not coalesced into < a1, b1, c1 >, primarily because the operands S and T″ do not have primary keys with a common domain. Judging from their present extensions, S and T″ merely have weak identifiers. Lacking a common primary key means that a typical row of S and a typical row of T″ represent quite different types of objects in the micro-world. Under these circumstances, it would be very risky to assume that the missing B-component of row < a1, —, c1 > of operand T″ is equal to b1, and that the missing C-component of row < a1, b1, — > of operand S is equal to c1.

In the following example, the coalescing of rows is acceptable. In this example, a bank has accounts of two different types. Suppose that one type is recorded in relation ACCOUNT; the other, in relation ACCT. The primary key of each relation is ACCOUNT#. No claim is made, of course, that the few columns in each relation are adequate for any bank. The small number was selected to keep the example simple.

| ACCOUNT | (ACCOUNT# | NAME | (ANNUAL) INTEREST_RATE | ($) BALANCE) |
|---|---|---|---|---|
| | 121-537 | Brown | 7.5 | 10,765 |
| | 129-972 | Baker | 8.0 | 25,000 |
| | 126-343 | Smith | 7.5 | 15,000 |
| | 302-888 | Jones | 8.0 | 18,000 |

| ACCT | (ACCOUNT# | NAME | MATURITY_DATE | BALANCE) |
|---|---|---|---|---|
| | 645-802 | Green | 95-12-31 | 35,680 |
| | 645-195 | Hawk | 94-09-30 | 50,000 |
| | 640-466 | Shaw | 96-03-31 | 22,500 |
| | 642-733 | Piper | 97-10-30 | 30,900 |
| | 302-888 | Jones | 96-07-31 | 18,000 |

Note that the first table is unique in having INTEREST_RATE as a column, while the second table is unique in having MATURITY_DATE as a column. Thus, these two relations are not union-compatible. Part of

the **outer union** operation is to convert them into a pair of relations that are union-compatible by applying the **extend** operator to each one.

For purposes of exposition, not implementation, the **outer union** A of these two relations is developed in two stages. First is the generation of a temporary result A′:

| A′ (ACCOUNT# | NAME | MATURITY_DATE | (ANNUAL) INTEREST_RATE | ($) BALANCE) |
|---|---|---|---|---|
| 121-537 | Brown | — | 7.5 | 10,765 |
| 129-972 | Baker | — | 8.0 | 25,000 |
| 126-343 | Smith | — | 7.5 | 15,000 |
| 645-802 | Green | 95-12-31 | — | 35,680 |
| 645-195 | Hawk | 94-09-30 | — | 50,000 |
| 640-466 | Shaw | 96-03-31 | — | 22,500 |
| 642-733 | Piper | 97-10-30 | — | 30,900 |
| 302-888 | Jones | — | 8.0 | 18,000 |
| 302-888 | Jones | 96-07-31 | — | 18,000 |

The final result A differs from A′ in only one respect: the DBMS attempts to coalesce the two rows describing accounts held by Jones, because the two operands have a primary key in common, and these rows have a common primary-key value. The attempt succeeds because each of the corresponding non-missing properties in the two rows has pairwise equal values. Thus, the two Jones rows in A′ are close counterparts. The end result A contains the row < 302-888, Jones, 96-07-31, 8.0, 18,000 > instead of the two Jones rows in A′.

## *RZ-20* Outer Set Difference

The **outer set difference** S \ − / T between relations S and T, with S as the information source and T as the reducing relation, is generated by means of the following steps:

1. form St = S **per** T;
2. form Ts = T **per** S;
3. form the **semi-equi-join** U = St**[sem=]**Ts;
4. form S \ − / T = St − U.

The close-counterpart concept (p. 116) is used instead of row equality.

By definition,

S \ − / T = St − (St **sem=** Ts).

And clearly,

$$S \setminus\!-\!/\ T \neq T \setminus\!-\!/\ S.$$

Take the sample relations as operands, and apply the **outer set difference:**

**S \ − / T**

| A | B | C |
|---|---|---|
| a1 | b1 | — |
| a1 | b2 | — |

**S \ − / T″**

| A | B | C |
|---|---|---|
| empty | | |

**T \ − / S**

| A | B | C |
|---|---|---|
| a3 | — | c4 |

**T″ \ − / S**

| A | B | C |
|---|---|---|
| a3 | — | c4 |

Once again, consider the operands ACCOUNT and ACCT in the bank example. Suppose that a user requests all of the accounts information from the ACCOUNT relation, but excluding those rows that have close counterparts in the ACCT relation. The DBMS responds by extending each operand in accordance with the other, and then removing those rows in the extended ACCOUNT relation that have close counterparts in the extended ACCT relation. The result is defined by

$$\text{DIFF} \leftarrow \text{ACCOUNT} \setminus\!-\!/\ \text{ACCT}.$$

Its extension is as follows:

| DIFF (ACCOUNT# | NAME | MATURITY_DATE | (ANNUAL) INTEREST_RATE | ($) BALANCE) |
|---|---|---|---|---|
| 121-537 | Brown | — | 7.5 | 10,765 |
| 129-972 | Baker | — | 8.0 | 25,000 |
| 126-343 | Smith | — | 7.5 | 15,000 |

## *RZ-21* Outer Set Intersection

The **outer set intersection** $S \setminus\cap/\ T$ of relations S and T is generated by means of the following steps:

1. form St ← S **per** T;
2. form Ts ← T **per** S;
3. form U ← St **sem=** Ts;
4. form V ← Ts **sem=** St;
5. form $S \setminus\cap/\ T \leftarrow U \cap V$.

The close-counterpart concept (p. 116) is used instead of row equality.

By definition,

$S \setminus\cap/ T = ((S \textbf{ per } T) \textbf{ sem} = (T \textbf{ per } S)) \cup ((T \textbf{ per } S) \textbf{ sem} = (S \textbf{ per } T))$

and clearly,

$S \setminus\cap/ T = T \setminus\cap/ S.$

Take the sample relations as operands, and apply the **outer set intersection:**

$S \setminus\cap/ T$

| A | B | C |
|---|---|---|
| a2 | b3 | — |
| a2 | — | c3 |

$S \setminus\cap/ T''$

| A | B | C |
|---|---|---|
| a1 | b1 | — |
| a1 | b2 | — |
| a2 | b3 | — |
| a1 | — | c1 |
| a1 | — | c2 |
| a2 | — | c3 |

$T \setminus\cap/ S$

| A | B | C |
|---|---|---|
| a2 | b3 | — |
| a2 | — | c3 |

$T'' \setminus\cap/ S$

| A | B | C |
|---|---|---|
| a1 | b1 | — |
| a1 | b2 | — |
| a2 | b3 | — |
| a1 | — | c1 |
| a1 | — | c2 |
| a2 | — | c3 |

The sample operands S and T, displayed at the beginning of this section, will now be used again to show that the **symmetric outer join** yields a quite different result from that generated by the **outer union, difference,** and **intersection** operators. The composite columns labeled A and E are used as comparands:

$U = S [A \setminus=/ E] T.$

U

| A | B | E | C |
|---|---|---|---|
| a1 | b1 | — | — |
| a1 | b2 | — | — |
| a2 | b3 | a2 | c3 |
| — | — | a3 | c4 |

Once again consider the operands ACCOUNT and ACCT in the bank example. Suppose that a user requests all the accounts information that is common to the ACCOUNT relation and the ACCT relation. The DBMS responds by extending each operand per the other, and then preserving those rows in the extended ACCOUNT relation that have close counterparts in the extended ACCT relation.

The result is defined by

INT ← ACCOUNT \∩/ ACCT.

Its extension is as follows:

| INT (ACCOUNT# | NAME | MATURITY_DATE | ANNUAL INTEREST_RATE | $ BALANCE) |
|---|---|---|---|---|
| 302-888 | Jones | 96-07-31 | 8.0 | 18,000 |

### 5.6.3 The Relationship between the Outer Set Operators

It should now be clear that, for any pair of relations S and T, the following identity holds:

S \∪/ T = (S \−/ T) ∪ (S \∩/ T) ∪ (T \−/ S).

This outer identity is very similar to the relationship between the inner set operators; the latter identity was defined at the end of Section 5.6.1.

This outer identity can be seen in action by applying it to the two cases of **outer union.** The first case makes use of the sample relations S and T.

S \∪/ T

| (A | B | C ) | |
|---|---|---|---|
| a1 | b1 | — | } S \−/ T |
| a1 | b2 | — | |
| a2 | b3 | — | } S \∩/ T = T \∩/ S |
| a2 | — | c3 | |
| a3 | — | c4 | } T \−/ S |

The second case makes use of the relations S and T″:

S \∪/ T″

| (A | B | C ) | |
|---|---|---|---|
| | | | } S \−/ T″ empty |
| a1 | b1 | — | } S \∩/ T″ = T″ \∩/ S |
| a1 | b2 | — | |
| a2 | b3 | — | |
| a1 | — | c1 | |
| a1 | — | c2 | |
| a2 | — | c3 | |
| a3 | — | c4 | } T″ \−/ S |

## 5.7 ■ The Inner and Outer T-join

The well-known **inner joins** are based on the 10 comparators:

1. EQUALITY
2. INEQUALITY
3. LESS THAN
4. LESS THAN OR EQUAL TO
5. GREATER THAN
6. GREATER THAN OR EQUAL TO
7. GREATEST LESS THAN
8. GREATEST LESS THAN OR EQUAL TO
9. LEAST GREATER THAN
10. LEAST GREATER THAN OR EQUAL TO

These **inner joins,** together with the corresponding **outer joins,** are readily accepted today. The **T-join** operators about to be described are new kinds of joins, principally based on the four ordering comparators (numbered 3–6 in the preceding list). The **inner T-join** produces a subset of that produced by the corresponding **inner join;** the **outer T-join** produces a subset of that generated by the corresponding **outer join.** The **T-joins** should be regarded as a proposed enrichment of the relational model, not a replacement for any of the original **inner joins** or **outer joins.**

The topic of **T-joins** is a complicated one. The reader may wish to skip to Section 5.8, which is much simpler.

### 5.7.1 Introduction to the T-join Operators

The four ordering comparators are as follows:

| | | |
|---|---|---|
| Strict: | LESS THAN | GREATER THAN |
| Non-strict: | LESS THAN OR EQUAL TO | GREATER THAN OR EQUAL TO |

Full joins based on these comparators frequently yield a result that is not very informative because it includes too many concatenations of the tuples of the operands. For example, consider the following relations S and T:

S (P A)

| P | A |
|---|---|
| k1 | 4 |
| k2 | 6 |
| k3 | 12 |
| k4 | 18 |
| k5 | 20 |

T (Q B)

| Q | B |
|---|---|
| m1 | 3 |
| m2 | 5 |
| m3 | 9 |
| m4 | 11 |
| m5 | 13 |
| m6 | 15 |

Suppose that A and B draw their values from the same domain, and that the comparator LESS THAN (<) is applicable on this domain. One of the full **joins** is

U = S [ A < B ] T.

Its extension is as follows:

U

| (P | A | B | Q ) |
|---|---|---|---|
| k1 | 4 | 5 | m2 |
| k1 | 4 | 9 | m3 |
| k1 | 4 | 11 | m4 |
| k1 | 4 | 13 | m5 |
| k1 | 4 | 15 | m6 |
| k2 | 6 | 9 | m3 |
| k2 | 6 | 11 | m4 |
| k2 | 6 | 13 | m5 |
| k2 | 6 | 15 | m6 |
| k3 | 12 | 13 | m5 |
| k3 | 12 | 15 | m6 |

Of the items being compared (A and B), if both are calendar dates, times of day, or combinations of dates and times, a **join** is often needed that is "leaner" than this full **join** ("leaner" in the sense that it has fewer rows or tuples). **T-joins** are intended to fill this role.

Each of the new **joins** is defined constructively—that is, in terms of an algorithm that will generate the appropriate result. An implementation can make use of this algorithm, but is not required to do so. All that is necessary is that the implementation generate the same result as the defining algorithm.

An important first step in the defining algorithm is to order the rows in each operand on the basis of the values in the comparand column of that operand. If the comparand column in each of the operands is guaranteed by a declaration in the catalog to contain no duplicate values, then precisely the same ordering will be generated again if the command is re-executed later, provided the data in the operand relations has not changed. If, however, duplicate values are permitted in one or both of these columns, the DBMS must be able to make use of values in other columns to resolve ties in the comparand columns.

These other columns are called *tie-breaking columns.* The need for resolving these ties stems from the need to make the operation precisely repeatable, if it should be re-executed later with the operands in exactly the same state at the relational level, but not necessarily in the same state at the storage level. Precise re-executability of relational commands is required by Fundamental Law 15 (see Chapter 29).

To be a good tie-breaker, a tie-breaking column should be of a *highly discriminating character:* that is, the number of distinct values in such a column divided by the number of rows in the operand must be as close as

possible to one. Clearly, the best tie-breaking columns are the primary-key column (or for certain kinds of views, the weak identifier) and any other column for which there is a declaration in the catalog that each of its values must be unique within that column.

Every datum in a computer-supported database is represented by a bit string of some length. Now, every bit string of length L bits (say) can be interpreted as a binary integer, whether that bit string represents a number, a string of logical truth-values, or a string of characters. This integer can therefore be arithmetically compared with every other bit string of length L if this latter string is also interpreted as a binary integer.

Thus, it might be proposed that, whenever a tie is encountered in which two equal values from the comparand columns are competing to qualify as representatives of two candidate rows of one of the operands, the tie can be broken by (1) descending to the bit level in other items of data and (2) comparing the corresponding binary integers arithmetically. To achieve precise repeatability of the operation, however, this approach assumes too much—namely, that the representation of data by the DBMS in storage remains constant. This approach departs from the principle that all actions in the relational model should be explicable either within the model or, at the very least, at the same level of abstraction as the model.

To avoid complexity, the approach taken in RM/V2 is to assume that each of the operands includes a primary key, and that the DBMS has information concerning which column of that operand, simple or composite, constitutes that key. This assumption is consistent with the expected use of **T-joins** for generating schedules. After all, when scheduling activities, it is necessary to know precisely for each line in the schedule which activity is involved.

In addition to discussing the repeatable generation of orderings, it is necessary to classify the comparators as in the beginning of this section. The comparators involving ordering are called the *ordering comparators*. The two strict ordering comparators are LESS THAN and GREATER THAN, and the two non-strict ordering comparators are LESS THAN OR EQUAL TO and GREATER THAN OR EQUAL TO. The two non-ordering comparators are EQUAL TO and NOT EQUAL TO, but these two do not participate in the proposed new joins.

These new **joins** are introduced step by step in Sections 5.7.2 and 5.7.3. The sample relations S and T cited earlier are used to illustrate various points.

### 5.7.2 The Inner T-join

#### *RZ-22 through RZ-25* Inner T-joins

The four new inner joins, called the **inner T-joins**, are each based on one of the four ordering comparators: LESS THAN, LESS

THAN OR EQUAL TO, GREATER THAN, or GREATER THAN OR EQUAL TO.

---

Suppose that the inner **T-join** is distinguished from the 10 full **inner joins** by doubling the square brackets around the comparing expression. For example,

V = S [[ A < B ]] T.

Let n = 22,23,24,25. Then the RZ feature with n as its number is a T-join that makes use of the comparator numbered n + 19 in the list of comparators cited in the first part of Section 5.7. An interesting property of V is that each tuple of S contributes to either no tuple at all in V or else to exactly one tuple in V. A similar remark can be made about contributions from tuples of T.

Next, **T-joins** are examined using the strict-ordering comparators LESS THAN and GREATER THAN, followed by an examination of **T-joins** using the non-strict-ordering comparators LESS THAN OR EQUAL TO and GREATER THAN OR EQUAL TO.

**Strict Ordering in T-joins** The result V is formed by the DBMS in two major steps, which are described here to enable DBMS vendors and users to understand **T-joins.** Because **T-joins** are expected to be built into DBMS products, it should never be necessary for the user to program these steps.

1. Suppose that column P is the primary key of relation S, and thus its tie-breaker, while column Q is the primary key and tie-breaker of relation T. Suppose that relation S is ordered by *increasing values* of A; relation T, by *increasing values* of B. Whenever repetitions of a value are encountered in A, break the tie by selecting the corresponding rows from S in an order determined by increasing values of P. Whenever repetitions of a value are encountered in B, select the corresponding rows from T in an order determined by increasing values of Q.
2. Take the first tuple from S. This is the tuple with the least value from column A of relation S and, in case of ties, the least value of P. Note that the DBMS applies the comparator LESS THAN (<) to P *even if* there is a declaration in the catalog that < is not meaningfully applicable to the domain of P. Concatenate it with the first available tuple from T that satisfies A < B. This is the tuple with the least value from column B of T, and in case of ties, the least value of Q. Assuming one such tuple is found in T, mark that tuple in T as used and unavailable. To complete this first minor part of Step 2, contribute the concatenated tuple to V.

   Now take the second tuple from S and concatenate it with the first available tuple in T for which A < B. Assuming one such tuple is found

in T, mark that tuple as used and unavailable. To complete this second minor part of Step 2, contribute the concatenated tuple to V. These minor steps are repeated until the whole of S is scanned. The availability marks are then erased from the operands.

This explanation is constructive and is intended to define the **inner T-join** based on the comparator <. Naturally, when the **T-join** operator is implemented within a DBMS, it is not required that this particular algorithm be used. All that is required is that, whatever operands are given, exactly the same result must be generated by the implemented algorithm and by this algorithm.

The extension of V resulting from this procedure is as follows:

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

The full **join** U could be an intermediate product in the formation of V, but U is not required to be an intermediate product. Although the **T-join** V represents a feasible schedule (assuming that properties A and B are date- or time-oriented), it is important to observe that some values of A and some of B may be omitted altogether in the result.

B = 3 can be thought of as a non-participant because it is a value of B less than every value of A. Similarly, A = 18 and A = 20 can be thought of as non-participants because they are values of A that are greater than every value of B.

The non-participants encountered so far are terminal. There may exist one or more non-terminal non-participants. In the preceding example, B = 11 is a non-terminal non-participant.

Now, let us investigate

$$W = T\ [[\ B > A\ ]]\ S.$$

One might expect that W would have the same information content as V because the analagous full **joins** are equal to one another. However, this is not the case with **T-joins.**

To construct W conceptually,

- relation S should be ordered by *decreasing values* of B;
- relation R should be ordered by *decreasing values* of A.

Now, the same general procedure is followed as used in forming V. Relation S is ordered by *decreasing values* of A, and relation T is ordered by *decreasing values* of B. Take the first tuple from T. This is the tuple with the greatest value in column B of relation T and, in case of ties, the greatest

value of Q. Note that the DBMS applies the comparator > to Q *even if* there is a declaration in the catalog that < is not meaningfully applicable to the domain of P. Concatenate it with the first tuple from S that satisfies B > A. This is the tuple with the greatest value of A. Assuming that one such tuple is found in R, mark that tuple as used and no longer available. Contribute the concatenated tuple to W. These steps are repeated until the whole of T is scanned. The availability marks are then erased from the operands.

The extension of W resulting from this procedure is as follows:

| W | (Q | B | A | P ) |
|---|---|---|---|---|
| | m6 | 15 | 12 | k3 |
| | m5 | 13 | 6 | k2 |
| | m4 | 11 | 4 | k1 |

Suppose that column ordering is disregarded (this is quite normal in the relational model), but that the column headings are noted. Clearly, W is not identical to V in information content. In fact, *every tuple of W is different from every tuple of V* (a strong contrast not encountered in some other examples).

Moreover, the values that do not participate in W are

$$A = 18,\ A = 20,\ B = 3,\ B = 5,\ B = 9.$$

All of these non-participants are terminal. In contrast to V, in W there is no non-terminal non-participant. It is therefore important to take care in choosing the ordering of terms in an expression defining a **T-join;** otherwise, the meaning of an intended query may not be conveyed accurately to the DBMS.

Because the interesting case is that in which duplicate values actually occur in A, in B, or in both, this part is based on slightly altered extensions for S and T. The tuples marked with an asterisk ("*") have been added, and the two occurrences of 6 as a value of A are distinguished by the primary key values k2 and k6 in column P of relation S′.

| S′ (P | A) |
|---|---|
| k1 | 4 |
| k2 | 6 |
| k6 | 6* |
| k3 | 12 |
| k4 | 18 |
| k5 | 20 |

| T′ (Q | B ) |
|---|---|
| m1 | 3 |
| m2 | 5 |
| m7 | 7* |
| m3 | 9 |
| m4 | 11 |
| m5 | 13 |
| m6 | 15 |

Suppose as before that A and B draw their values from the same domain. One of the full **joins** is

$$U' = S' [\ A < B\ ] T'.$$

Its extension has 17 tuples (of course, more than U):

| Expository row number | U′ | (P | A | B | Q ) |
|---|---|---|---|---|---|
| 1 | | k1 | 4 | 5 | m2 |
| 2 | | k1 | 4 | 9 | m3 |
| 3 | | k1 | 4 | 11 | m4 |
| 4 | | k1 | 4 | 13 | m5 |
| 5 | | k1 | 4 | 15 | m6 |
| 6 | | k2 | 6 | 7 | m7 |
| 7 | | k2 | 6 | 9 | m3 |
| 8 | | k2 | 6 | 11 | m4 |
| 9 | | k2 | 6 | 13 | m5 |
| 10 | | k2 | 6 | 15 | m6 |
| 11 | | k7 | 6 | 7 | m7 |
| 12 | | k7 | 6 | 9 | m3 |
| 13 | | k7 | 6 | 11 | m4 |
| 14 | | k7 | 6 | 13 | m5 |
| 15 | | k7 | 6 | 15 | m6 |
| 16 | | k3 | 12 | 13 | m5 |
| 17 | | k3 | 12 | 15 | m6 |

Of course, the row number at the extreme left appears for expository purposes only.

Now consider

$$V' = S' [[ A < B ]] T'.$$

Its extension is as follows:

| V′ | (P | A | B | Q ) |
|---|---|---|---|---|
| | k1 | 4 | 5 | m2 |
| | k2 | 6 | 7 | m7 |
| | k6 | 6 | 9 | m3 |
| | k5 | 12 | 13 | m5 |

Note that, in this example, A = 18, A = 20, B = 3 remain terminal non-participants, while B = 11 remains a non-terminal non-participant. Moreover, if

$$W' = S' [[ B > A ]] R',$$

then W′ remains quite different from V′:

| W′ | (Q | B | A | P ) |
|---|---|---|---|---|
| | m6 | 15 | 12 | k3 |
| | m5 | 13 | 6 | k6 |
| | m4 | 11 | 6 | k2 |
| | m3 | 9 | 4 | k1 |

Clearly, W′ is quite different from V′ in information content. In fact, in this case every tuple of W′ is different from every tuple of V′.

**Non-strict Ordering in T-joins** Now, it is appropriate to consider the non-strict comparators LESS THAN OR EQUAL TO (<=) and GREATER THAN OR EQUAL TO (>=). The introduction of equality as part of the comparing brings with it a new problem: the possibility of *cross-ties*, in which two or more comparand values from one comparand column are not only equal to one another, but are also equal to two or more comparand values from the other comparand column. These occurrences of equality both within and between values in the comparand columns contribute at least four rows to the result of a full **join.** The question arises: In what way are certain rows selected to participate in a **T-join** result, while other rows are rejected?

In Chapter 17, "View updatability," the term *quad* is defined as 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 any full **join** based on a comparator that involves equality must consist of a number of rows that is the product of the two integers *m* and *n*. Since each integer is at least 2, this product cannot be less than 4: hence, the name "quad." Clearly, a quad contribution cannot consist of 3, 5, 7, 11, or any prime number of rows. When quads can occur in **T-join** operands, the selection of cross-ties that survive in the result becomes an issue that must be handled by the DBMS (*not* the user).

The following example illustrates a quad. Suppose that relations S and T contain rows as indicated:

| S ( P | A ) |
|---|---|
| k7 | 13 |
| k5 | 13 |
| .... | |

| T (B | Q ) |
|---|---|
| 13 | m5 |
| 13 | m8 |
| .... | |

Then, the full LESS THAN OR EQUAL TO join of S on A, with T on B, includes the following four rows because of the cross-ties arising from the multiple occurrences of the value 13 in both A and B:

| Row label | U | (P | A | B | Q ) |
|---|---|---|---|---|---|
| row t | | k5 | 13 | 13 | m5 |
| row u | | k5 | 13 | 13 | m8 |
| row v | | k7 | 13 | 13 | m5 |
| row w | | k7 | 13 | 13 | m8 |
| Other rows | | ..... | | | |

Note that the row labels are purely expository.

In a **T-join,** each row of each operand may be used once only. This

means that each of the rows < k5, 13 > and < k7, 13 > from S can be used once only, not twice as indicated in the preceding full **join.** Thus, in the **T-join** result for this example the DBMS must choose between row t and row u. The DBMS must also choose between row v and row w.

The defining algorithm for **T-joins** resolves cross-ties by selecting those two rows from any quad that contains in the primary key columns the combination of values that are greatest within the quad that remain unused in the result.

Remember that, in general, a quad contains $m \times n$ rows, where $m$ and $n$ are at least two.

In the example, columns P and Q are the primary-key columns. Row t contains the combination of least keys in the quad < k5, m5 >, while row w contains the combination of the greatest keys in the quad < k7, m8 >. Thus, row t and row w are selected to be the only participants in the LESS THAN OR EQUAL TO **T-join** of S on A with T on B.

This algorithm is designed to make execution of all **T-joins** repeatable in the sense that, if the operands remain unchanged, so does the result—even if access methods and representations of the operands have been changed in storage.

If the examples of S and T were those illustrated in Section 5.7.1 and the comparators were changed from strict to non-strict, the resulting relations U, V, W would be unchanged, because there were no occurrences of equality when comparing values from A with values from B. A similar remark applies to S′ and T′ in the preceding discussion. Therefore, relations S″ and T″ are introduced. Each of these relations has ties in the comparand columns; the pair of relations also has cross-ties:

| S″ (P | A) |
|---|---|
| k1 | 4 |
| k2 | 6 |
| k6 | 6 |
| k3 | 8 |
| k4 | 9 |
| k5 | 10 |

| T″ (Q | B) |
|---|---|
| m1 | 3 |
| m2 | 5 |
| m7 | 6 |
| m8 | 6 |
| m3 | 9 |
| m4 | 11 |
| m5 | 13 |
| m6 | 15 |

Consider the **T-join** V″ of S″ on A with T″ on B using the comparator LESS THAN OR EQUAL TO (<=). The defining expression for this **join** is

$$V'' = S'' \; [[\; A <= B \;]] \; T''.$$

The extension of V″ is as follows:

| V″ | (P | A | B | Q) |
|---|---|---|---|---|
| | k1 | 4 | 5 | m2 |
| | k2 | 6 | 6 | m7 |
| | k6 | 6 | 6 | m8 |
| | k3 | 8 | 9 | m3 |
| | k4 | 9 | 13 | m5 |

Note that, in resolving cross-ties, the DBMS did not select the rows < k2, 6, 6, m8 > and < k6, 6, 6, m7 > to be members of V″.

The new problem that arises with the non-strict ordering comparators is the need to resolve cross-ties between comparand values by use of columns other than the comparand columns. The technique built into **T-joins** provides a systematic resolution.

The following practical example exhibits some of the limitations in the present version of the **T-join** operator. Suppose that students have registered for certain classes that are scheduled to run concurrently in various rooms and buildings. The relation ROOM identifies and describes each room that is available for classes. The relation CLASS identifies and describes each class.

| | | |
|---|---|---|
| ROOM | R# | Room serial number |
| | BLDG | Building name |
| | SIZE | Number of seats for students |
| CLASS | C# | Class identifier |
| | STUDENTS | Number of students registered for a class |

Assume the following extensions for these two relations:

| ROOM | (R# | BLDG | SIZE) |
|---|---|---|---|
| | r1 | lab | 70 |
| | r2 | lab | 40 |
| | r3 | lab | 50 |
| | r4 | tower | 85 |
| | r5 | tower | 30 |
| | r6 | tower | 65 |
| | r7 | tower | 55 |

| CLASS | (C# | STUDENTS) | |
|---|---|---|---|
| | c1 | 80 | 75 |
| | c2 | 70 | 65 |
| | c3 | 65 | 60 |
| | c4 | 55 | 50 |
| | c5 | 50 | 45 |
| | c6 | 40 | 35 |

alternative column of data

To assign any class to a room, it is required that the room have a number of seats in excess of the number of students in the class. The **T-join** operator can be used to assign classes to rooms in two ways:

U1 ← CLASS [[ STUDENTS < SIZE ]] ROOM
U2 ← ROOM [[ SIZE > STUDENTS ]] CLASS.

Let us illustrate these two approaches. First, the rows of the operand CLASS are ordered by enrollment in each class, and the rows of ROOM are ordered by room size—both in *ascending* order, in preparation for the derivation of U1.

CLASS″

| C# | STUDENTS |
|---|---|
| c6 | 40 |
| c5 | 50 |
| c4 | 55 |
| c3 | 65 |
| c2 | 70 |
| c1 | 80 |

ROOM″

| R# | BLDG | SIZE |
|---|---|---|
| r5 | tower | 30 |
| r2 | lab | 40 |
| r3 | lab | 50 |
| r7 | tower | 55 |
| r6 | tower | 65 |
| r1 | lab | 70 |
| r4 | tower | 85 |

U1 ← CLASS [[ STUDENTS < SIZE ]] ROOM

U1

| C# | STUDENTS | SIZE | R# | BLDG |
|---|---|---|---|---|
| c6 | 40 | 50 | r3 | lab |
| c5 | 50 | 55 | r7 | tower |
| c4 | 55 | 65 | r6 | tower |
| c3 | 65 | 70 | r1 | lab |
| c2 | 70 | 85 | r4 | tower |

In the second attack on this problem, the rows of the operand ROOM are ordered by room size, and the rows of the operand CLASS are ordered by enrollment in each class, both in *descending* order, in preparation for the derivation of U2.

ROOM″

| R# | BLDG | SIZE |
|---|---|---|
| r4 | tower | 85 |
| r1 | lab | 70 |
| r6 | tower | 65 |
| r7 | tower | 55 |
| r3 | lab | 50 |
| r2 | lab | 40 |
| r5 | tower | 30 |

CLASS″

| C# | STUDENTS |
|---|---|
| c1 | 80 |
| c2 | 70 |
| c3 | 65 |
| c4 | 55 |
| c5 | 50 |
| c6 | 40 |

U2 ← ROOM [[ SIZE > STUDENTS ]] CLASS

U2

| R# | BLDG | SIZE | STUDENTS | C# |
|---|---|---|---|---|
| r4 | tower | 85 | 80 | c1 |
| r1 | lab | 70 | 65 | c3 |
| r6 | tower | 65 | 55 | c4 |
| r7 | tower | 55 | 50 | c5 |
| r3 | lab | 50 | 40 | c6 |

In this example, the two results U1 and U2 are different from one another. Neither of the **T-joins** using LESS THAN and GREATER THAN, respectively, assigns all of the classes to rooms. Class c1 is omitted from assignment in the first **T-join,** and class c2 in the second. However, if the comparators are changed to LESS THAN OR EQUAL TO and GREATER THAN OR EQUAL TO each one of these **T-joins** assigns all of the classes to rooms. In general, given any collection of classes and any collection of rooms, and the requirement that distinct classes be assigned to distinct rooms, there is no guarantee that each and every class can be assigned.

### 5.7.3 The Outer T-join

The **outer T-joins,** like the **inner T-joins** defined in Section 5.7.2, are each based on one of the four ordering comparators.

For each **inner T-join,** three kinds of **outer T-joins** are potentially useful: the **left outer T-join,** the **right outer T-join,** and the **symmetric T-join.** These **outer T-joins** are now defined in a constructive manner, but with no restriction intended on how they are implemented.

#### *RZ-26 through RZ-37* Outer T-joins

The **outer T-join** of relations S on A with T on B consists of the **inner T-join** U of S on A with T on B, together with additional sets of tuples, called the *outer increments.* The **inner T-join** of S on A with T on B is denoted

V = S [[ A @ B ]] T,

where "@" stands for one of the four ordering comparators. S is called the *left operand;* T, the *right operand.*

There are two distinctly defined outer increments. To construct the *left outer increment,* collect those tuples of the left operand S that do not happen to participate in the **inner T-join;** to each of these, append a sufficient number of marked values to indicate that the value of each component of a tuple from T is missing but applicable. To construct the *right outer increment,* collect those tuples of the right operand T that do not happen to participate in the **inner T-join;** to each of these, append a sufficient number of marked values to indicate that the value of each component of a tuple from S is missing but applicable.

Each **outer T-join** is the **union** of the corresponding **inner T-join,** together with the following:

- the left outer increment for the 4 **left outer T-joins** RZ-26 through RZ-29;

- the right outer increment for the 4 **right outer T-joins** RZ-30 through RZ-33;
- both increments for the 4 **symmetric outer T-joins** RZ-34 through RZ-37.

---

Suppose that the **outer joins** of S on A with T on B, using the comparator @, are denoted as follows:

| | |
|---|---|
| **Left outer T-join** | VL = S [o[ A @ B ]] T |
| **Right outer T-join** | VR = S [[ A @ B ]o] T |
| **Symmetric outer T-join** | VS = S [o[ A @ B ]o] T |

Note that the lowercase letter "o" is inserted between the left square brackets (for the **left outer join**), between the right square brackets (for the **right outer join**), or between both pairs of square brackets (for the **symmetric outer join**).

Taking the sample operands S, T presented in Section 5.2, the following results are obtained for the **outer T-joins** based on the LESS THAN comparator (<). Missing information is represented by a hyphen ("—") in these examples, and tuples from the **inner T-join** based on < are marked with an asterisk.

**Left VL**

| | P | A | B | Q |
|---|---|---|---|---|
| * | k1 | 4 | 5 | m2 |
| * | k2 | 6 | 9 | m3 |
| * | k3 | 12 | 13 | m5 |
| | k4 | 18 | — | — |
| | k5 | 20 | — | — |

**Right VR**

| | P | A | B | Q |
|---|---|---|---|---|
| | — | — | 3 | m1 |
| * | k1 | 4 | 5 | m2 |
| * | k2 | 6 | 9 | m3 |
| | — | — | 11 | m4 |
| * | k3 | 12 | 13 | m5 |
| | — | — | 15 | m6 |

**Symmetric VS**

| | P | A | B | Q |
|---|---|---|---|---|
| | — | — | 3 | m1 |
| * | k1 | 4 | 5 | m2 |
| * | k2 | 6 | 9 | m3 |
| | — | — | 11 | m4 |
| * | k3 | 12 | 13 | m5 |
| | — | — | 15 | m6 |
| | k4 | 18 | — | — |
| | k5 | 20 | — | — |

## 5.7.4 Summary of T-joins

There are four simple **inner T-joins** corresponding to the following four ordering comparators:

RZ-22 LESS THAN
RZ-23 LESS THAN OR EQUAL TO
RZ-24 GREATER THAN
RZ-25 GREATER THAN OR EQUAL TO.

There are 12 simple **outer T-joins,** three for each of the four ordering comparators. The three types are the **left outer T-joins** (Features RZ-26–RZ-29), the **right outer T-joins** (Features RZ-30–RZ-33), and the **symmetric outer T-joins** (Features RZ-34–RZ-37).

In Table 5.1, which summarizes the 16 simple **T-joins,** the following notation is used:

| | | | |
|---|---|---|---|
| I | Inner | 3 | LESS THAN |
| O | Outer | 4 | LESS THAN OR EQUAL TO |
| L | Left | 5 | GREATER THAN |
| R | Right | 6 | GREATER THAN OR EQUAL TO |
| S | Symmetric | | |
| C | Comparator | | |

The **inner** and **outer T-joins** can be applied effectively when the values being compared happen to be (1) date intervals, time intervals, or combinations of both, or (2) loads and capacities. The 16 simple **T-joins** may also be useful in some other situations. They are not useful, however, if the comparator < is declared in the catalog to be meaningfully inapplicable to the values being compared in the principal comparand columns.

The **T-joins** represent a step toward a relational operator that will probably appear in the next version of the relational model (RM/V3). This operator transforms two union-compatible relations involving a sequence of non-contiguous time intervals needed on some machines into a result that can be interpreted as a merged schedule for the two activities on those machines.

Before leaving the subject of **T-joins,** it is interesting to consider a counterpart to the **semi-theta-join,** namely the **semi-T-join.** It will be recalled from Chapter 4 that, under certain conditions, **semi-theta-join** can be useful in the efficient execution of inter-site **theta-joins** in a distributed database

***Table 5.1*** **Summary of Simple T-joins**

| **Feature RZ-** | **22** | **23** | **24** | **25** | **26** | **27** | **28** | **29** | **30** | **31** | **32** | **33** | **34** | **35** | **36** | **37** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| I or O | I | I | I | I | O | O | O | O | O | O | O | O | O | O | O | O |
| L, R, or S | — | — | — | — | L | L | L | L | R | R | R | R | S | S | S | S |
| Comparator | 3 | 4 | 5 | 6 | 3 | 4 | 5 | 6 | 3 | 4 | 5 | 6 | 3 | 4 | 5 | 6 |

management system. In the same way, **semi-T-joins** can be useful in the efficient execution of inter-site **T-joins.**

## 5.8 ■ The User-defined Select Operator

The main reason for this operator is to introduce a more powerful version of the **select** operator than the built-in version described in Chapter 4. This operator permits the selection of rows from a specified relation based on any user-defined function that transforms one or more row-components into a truth value.

The *built-in* operator **select** with operand relation S involves comparing the values in a specified column of S (say A) with either

1. some specified constant or host variable (say x), or
2. values in a second specified column of S (say B).

In Case 2, each pair of values that are compared (an A-value and a B-value) must be drawn from the same row of S. The distinction between Cases 1 and 2 does not apply to the **user-defined select** operator.

### *RZ-38* User-defined Select

This operator is denoted S [i; p(A); t], where i is an initializing function (optional), p is a truth-valued function (required), and t is a terminating function (optional). The argument A of the function p denotes one or more simple columns of the relation S. However, the truth value of p(A) must be computable for each row using only the A-components of that row. If A is a collection of columns, more than one component of each row is involved.

Note that the comparators in any user-defined **select** are hidden in the function p. Therefore, there is only one Feature RZ-38.

Specifying i, t, or both can be omitted in any user-defined **select** command. If included in the command, the initializing function i is executed to completion at the very beginning of the **select,** and delivers what is called the temporary version of S. If included in the command, the terminating function t is executed at the very end of the **select,** at which point all rows that qualify to be selected from S or from its temporary version have been selected. The operand of i is the relation S. The operand of t is the relation resulting from all the rows of S (or its temporary version) that happen to be selected.

The languages in which the functions may be expressed should include one of the host languages supported by the DBMS, together with retrieval

operators and the qualifier ORDER BY of the principal relational language, constrained to apply to the specified operand relation only.

The following example is intended to illustrate the practical use of the user-defined **select** operator. Suppose that a company has sales teams in various parts of the world. A database keeps track of sales by team in a relation called TEAM. Each team has an identifier TID that is unique with respect to teams. TID is the primary key of TEAM. The immediate properties of a team include year-to-date sales (all expressed in a single currency), total sales for each of the preceding five years, and number of members on that team.

Once each month, the company makes a statistical analysis of the year-to-date sales in relation to the sales of previous years. The function F is applied to measure long-term and short-term growth. F combines these two growths in some simple way to arrive at a performance rating. The function yields as its result the truth-value TRUE for about 10% of the sales teams, those that have achieved the best performance rating.

Let the relation describing each sales team be called TEAM. Suppose that TID denotes the team identifier (the primary key of TEAM). It is possible to use the function F to select the sales teams that have performed the best on a year-to-date basis:

WINNERS ← TEAM [;F(TID);].

Note that no initiating or terminating function is involved. Note also that F probably has several arguments (this fact is not shown).

## 5.9 ■ The User-defined Join Operator

This **join** operator is to a large extent user-defined, but not completely so. There are two main reasons for this:

1. the objective of continued support for optimization by the DBMS using techniques similar to those applicable to the built-in **joins;**
2. the objective of reducing, if not eliminating, the need for users to construct iterative programming loops.

### *RZ-39* User-defined Join

The user-defined join is more powerful than the built-in joins. It concatenates a row from one relation with a row from another whenever a user-defined function p transforms specified components of these rows into the truth value TRUE. If included in the command, the initializing function i is executed to completion at the very beginning of the **join,** before any rows of the first operand are

concatenated with any rows of the second operand. Temporary versions of the operands are delivered as the result of executing i. If included in the command, the terminating function t is executed at the very end of the **join,** at which point all rows that are to be concatenated have been concatenated.

---

A **user-defined join** of relations S on A with T on B using functions i, p, t may be specified by means of the following expression:

S [ i; p( A,B ); t ] T.

The operands for the initializing function i are S and T. One practical use of i is to generate temporary relations from S and T ordered by the values in their respective comparand columns A and B. The operand for the terminating function t is the result of the **join** up to that point. Incidentally, function p will rarely have an inverse, and an inverse is not required.

The languages in which the functions may be expressed should include one of the host languages supported by the DBMS, together with retrieval operators and the qualifier ORDER BY of the principal relational language, constrained to apply to the specified pair of operand relations only.

The following example extends the sales-analysis example cited in Section 5.8 for **user-defined select.** This extended example illustrates the practical use of the **user-defined join** operator.

Suppose that the previously described database also contains a relation CUST describing its large customers. One of the columns of CUST is the customer identifier CID—naturally, the primary key of CUST. Another property of each customer is the identifier TID of the sales team assigned to the customer. In this case TID is a foreign key. Other columns of CUST contain sales information similar to that in the TEAM relation, except that in each row the information applies to one customer only.

Suppose that another, different monthly analysis is required by the company for its large customers. The intent is to find *contra-flow situations*—that is, situations in which regional sales are increasing, while sales to one or more large customers in the region are decreasing. Let G be a function that is applied to customer sales data and team-oriented regional sales data. G yields the truth-value TRUE when growth of sales is positive for the region, but negative for a large customer. Team information such as team identifier TID and team manager TMGR, along with the customer name CNAME and customer location CLOC, is requested. This request can be expressed in terms of a **user-defined join** between the relations TEAM and CUST, followed by a projection:

CONTRA ← ( TEAM [; G ;] CUST ) [TID, TMGR, CNAME, CLOC].

The function G probably has several arguments (these are not shown in the request).

## 5.10 ■ Recursive Join

It has been asserted in a public forum that "the relational algebra is incapable of **recursive join.**" In fact, such an assertion is astonishingly erroneous. The **recursive join** was introduced 10 years ago in one of my technical papers [Codd 1979].

### *RZ-40* Recursive Join

The **recursive join** is an operator with one operand. This operand is a relation that represents a directed graph. One of the columns of this relation plays a subordinate (SUB) role, while another plays a superior (SUP) role. Each tuple represents an edge of a directed graph, and by convention this edge is directed from the node identified by the SUP component down to the node identified by the SUB component. Because **joins** are normally applied to pairs of relations, it is convenient to think of the single operand as two identical relations. The **recursive join** acts on this pair of identical relations by matching each SUB value in one operand to a SUP value in the second operand. It yields all of the pairs of identifiers for nodes that are connected by paths in the acyclic graph, no matter what the path lengths are.

It is useful to compare the regular **equi-join** with this recursive join. Note that a regular **equi-join** of such a relation matching each SUB value in one operand to the SUP in the second operand yields all of the paths that are precisely two edges in length. The distinction between regular **equi-join** and **recursive join** should therefore be clear: regular **join** is terminated by completion of a simple scan of one of the two relations, whether real or virtual; on the other hand, recursive **join** with respect to a path of the underlying acyclic graph is terminated only when a node is encountered which has no node that is subordinate to it. An equivalent way of expressing this termination with respect to a path is that it occurs when the path ends.

There are several versions of this **recursive join** and they differ principally in the information content of the result that is delivered. The simple version described above was presented in RM/T [Codd 1979, page 427] as the CLOSE operator. A more powerful version suitable for the bill-of-materials type of application and not yet published is likely to be included in the next version (RM/V3) of the relational model.

Some relations represent directed graphs. Relation S is a *directed graph relation* if it is of degree at least two and has the following properties:

- two of its columns are defined on a common domain;
- one of these columns has a superior role, termed SUP;
- the other column has a subordinate role, termed SUB;
- no other columns have the SUP or SUB role.

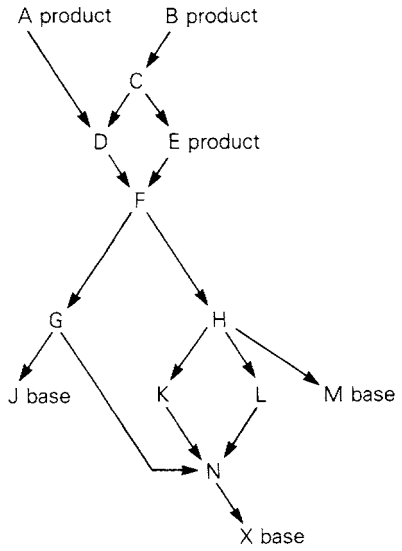
An interpretation of such a relation is that there is an edge of the graph that connects from the SUP component of any row down to the SUB component of that same row. Two edges are connected if the SUB value of one (the higher of the two edges) is the SUP value of the other (the lower of the two edges).

Suppose that a directed graph includes a sequence of edges, each connected to its successor and with the property that, if the successive edges are traversed according to the directedness of the graph, the traversal returns to the same node at which it started. Then, such a graph is cyclic, and the sample sequence of edges just described is called a *cyclic path*. An *acyclic graph* has no cyclic paths whatsoever.

An example of an acyclic graph is discussed briefly here and in Section 28.4. Figure 5.1 is a diagram of an acyclic directed graph.

An acyclic path in a directed graph consists of a sequence of edges, each of which is connected to one edge lower (except the lowest edge in the path)

***Figure 5.1*** **An Acyclic Directed Graph G1 Representing Product Structure**



and each of which is connected to one edge higher (except the highest edge in the path). All of the paths that exist in an acyclic directed graph are acyclic. Any traversal of a path in compliance with its direction is called *downward.* Any traversal in the opposite direction is called *upward.*

Note that nothing in the definition of the acyclic directed graph concept prevents a single SUB value from being associated with more than one SUP value. In other words, nothing prevents two or more nodes from acting as superiors to a single subordinate node. A *hierarchy* is a special case of an acyclic graph in which each node may have at most one immediate superior node.

In the connectivity part of the bill-of-materials type of problem, an example of this type is the product structure graph G1, shown in Figure 5.1. In this relation, single letters are used as part serial numbers to identify parts. To save space, the acyclic graph relation AG is listed "on its side" and the immediate properties of each edge are represented by a lowercase letter:

| AG | SUP | A | B | C | C | D | E | F | G | G | B | F | H | H | H | K | L | N |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SUB | D | C | D | E | F | F | G | J | N | H | H | K | L | M | N | N | X |
| | P | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q |

The graph corresponding to the relation AG appears in Figure 5.1.

Whenever product structure for two or more products is represented by a directed graph, each node represents a component and each edge represents the fact that one component is an immediate component of another. The graph in Figure 5.1 is clearly acyclic and nonhierarchic. Even if the graph of product structure begins its existence as a pure hierarchy, it is unlikely to remain that way. Thus, a general solution to the bill-of-materials problem should not assume the hierarchic structure.

There is a comprehensive solution to the general bill-of-materials problem based on the relational model. The solution is very concise, protects the user from iterative and recursive programming, and provides pertinent integrity constraints as well as manipulative capability. The **recursive join** now being described, however, is not a complete solution to this problem. (The more complete solution will be published later.)

The **recursive join** of RM/V2 has four arguments:

1. a single relation that represents an acyclic directed graph;
2. one column of node identifiers with the SUP role;
3. one column of node identifiers with the SUB role;
4. an identifier for a node from which all downward paths are to be traversed.

The result of **recursive join,** a relation that identifies every one of these downward paths, is therefore of degree three, with the columns SUB, SUP,

and PI (path identifier). In each row the SUB and SUP components together uniquely identify an edge on one of the downward paths, while PI uniquely identifies that path by means of an integer generated by the DBMS. If the acyclic graph includes a total of $N$ distinct paths downward from the specified node, the integers assigned to each of these paths is from the set $1, 2, \ldots, N$.

Some paths are likely to be composed of several edges. However, a particular edge may be part of two distinct paths, and may therefore occur with two distinct path identifiers. Remember that the graph is not necessarily hierarchical. The particular integer assigned to identify a path is meaningful only in the sense that it is distinct from all the other path identifiers.

A reasonable notation for **recursive join** is exhibited in the following example:

T ← S [ SUB | SUP ; PI ].

Note that, if this operator is to be applied to graphs that may include cycles, it must have a minor extension in its definition to avoid the peril of unending looping around the cycles. As execution proceeds, whenever the operator traverses an edge of the graph, it should temporarily mark that edge as traversed, and avoid traversing it again. RM/V2 includes only one version of **recursive join.** It works on relations that represent directed acyclic graphs. RM/V2 does not include an extended version that works on relations representing directed graphs that can have cycles in them. This extended version is a clear candidate for inclusion in RM/V3.

One interesting application of this cyclic version is that of recording contacts between criminals and suspects in a database for use by the police. In this case, contact between Person X and Person Y implies contact between Person Y and Person X, whereas the fact that Part p is an immediate component of Part q implies that Part q is *not* an immediate component of Part p.

## 5.11 ■ Concluding Remarks

There is no claim that the operators discussed in Chapters 4 and 5 represent all the operators that users will ever need. In fact, four more operators are introduced in Chapter 17, "View Updatability." When introducing any new operator, the reader is advised to remain within the discipline of the relational model (see Chapter 28).

## Exercises

5.1 What is the **framing** operator? What are its operands and results? What is it used for? Supply an example.

5.2 What is the **extend** operator and what is it used for? If the description of S extended per T is the same as the description of S, what is true of the relations S and T?

5.3 Describe an example that illustrates **outer union,** and state how this operator is likely to be used in practice.

5.4 What are the three kinds of **outer join?** Supply an example for each kind.

5.5 Define **outer T-join** and supply an example. How is **outer T-join** likely to be used in practice?

5.6 How does **inner T-join** differ from **outer T-join?** How is **inner T-join** likely to be used in practice?

5.7 The definitions of **theta-join, semi-theta-join,** and **T-join** can be found in Chapters 4 and 5. Supply a definition of **semi-T-join.**

5.8 A need arises for an **equi-join** involving two currency columns as comparand columns. However, the values in one of the comparand columns happen to be expressed in dollars, whereas the values in the other comparand column happen to be expressed in British sterling. Explain how you would apply a **user-defined join** to solve this problem.

5.9 Describe an example that illustrates **recursive join,** and state how it is likely to be used in practice. (See also Chapter 28.)

5.10 Develop two operand relations S and T with the following properties:

- they are joinable by both **theta join** and **T-join;**
- when S and T are combined by **theta join** using the comparator GREATEST LESS THAN, the result is U, say;
- when S and T are combined by **T-join** using the comparator LESS THAN, the result is V, say;
- the relations U and V are not identical.

## ■ CHAPTER 6 ■

# Naming

In this chapter, the topic of naming is discussed with respect to the management of non-distributed databases. Naming is taken up again in Chapters 24 and 25 with respect to the management of distributed databases.

When initially establishing a database, much of the naming is concerned with the database description, and hence belongs in the catalog. This naming is determined by the DBA staff who are designing the database. Later, during the interrogation and manipulation stage, much of the naming is concerned with the columns of intermediate and final results. This naming is initiated by the DBMS according to rules described in this chapter. Users must know these rules when combining several operators into one or more commands, whether these commands are executed interactively or from an application program.

In establishing or expanding a relational database, names must be assigned to domains, R-tables, columns, and functions. The features listed in the naming class (class N) make this activity reasonably systematic and in accordance with other features of the relational model—for example, protection of users from having to be aware of positioning within the database and "nextness" applied to rows and columns.

When a user attempts to insert names into the catalog, the DBMS must check whether the names are compatible with the features of class N. Because the user may be unaware of these features, the DBMS must be prepared to catch simple errors. All of the naming features discussed in this chapter apply to any single relational database, and are intended to make the database easy to understand and the interactions unambiguous.

One of the principles underlying these naming features is that, when deciding which names should be selected by the DBMS for the columns of every relation that is an intermediate or final result, interchangeability of the operands must not be reduced or in any way damaged. For example, **union** is an operator for which

$S \cup T = T \cup S.$

This commutativity could easily be damaged if the automatic naming of columns in the result were dependent upon which operand is cited first.

## 6.1 ■ Basic Naming Features

### *RN-1* Naming of Domains and Data Types

All domains (extended data types)—whether simple or composite, whether built-in or user-defined—must be assigned names that are distinct from one another, and distinct from the names of relations and functions.

The description of each domain must be stored in the catalog before any use is made of that domain.

### *RN-2* Naming of Relations and Functions

All relations, whether base or derived, and all functions, whether built-in or user-defined, must be assigned names that are distinct from one another, as well as distinct from all of the names of domains, data types, and columns.

The description of each relation and each function must be stored in the catalog before any use is made of either object.

### *RN-3* Naming of Columns

All columns, whether simple or composite, within any single relation must be assigned names that are distinct from one another, and distinct from the names of relations and functions.

Note that this feature does not require that all column names in the entire database be distinct from one another. Such a rule is not only unnecessary, but may also be counter-productive.

A guideline for naming that tends to make programs easier to read and understand is that the DBA and users abide by two simple rules:

1. If one considers the names of all domains, all relations, and all functions as a single collection of names, then in that collection every name is distinct from every other name.
2. Every column name is a combination of a role name and a domain name, where the role name designates in brief the purpose of the column's use of the specified domain.

For example, if the domain is QUANTITY OF PARTS (abbreviated Q) and a particular column designates the quantity-on-hand of parts, it would be appropriate to select Q as the domain name, OH as the role name, and OH_Q as the column name. Similarly, the quantity on order would be named OO_Q, and the quantity shipped would be named SHIP_Q. A DBMS that supports these guidelines should be regarded as supporting the somewhat less stringent features RN-1–RN-3.

The DBA may wish to impose the additional constraint that names of different kinds of objects should begin with a letter that designates the kind of object. For example,

| | |
|---|---|
| Relations | R |
| Domains | D |
| Columns | C |
| Role prefix | P |
| Functions | F |

While this additional constraint is not a requirement, compliance with this convention would make programs—and perhaps the database—easier to understand.

An important consequence of Features RN-2–RN-3 is that any combination of relation name and column name denotes precisely one column in the entire database, provided the column name is the name of a column within that relation. This fact is ignored in the design of the language SQL, which includes the clause SELECT . . . FROM . . . WHERE. One result is that **joins** are awkward to express in that language.

A simple syntax for such a composite name is a relation name, followed by a period, followed by the name of a column within that relation.

### *RN-4* Selecting Columns within Relational Commands

The combination of relation name and column name is an unambiguous way to select a particular column in a relational database. The syntax of RL must avoid separating column names from relation names, which causes (1) difficulty in extending the language and (2) either ambiguity or needless difficulty for users in understanding relational commands.

The end user or programmer must have the option of specifying the order in which columns are to be presented in a report.

### *RN-5* Naming Freedom

Success of the DBMS in executing any RL command (e.g., a **join**) that involves comparing database values from distinct columns must not depend on those columns having identical column names.

At the time of this writing, the NOMAD product includes the undesirable and unnecessary constraint on **joins** cited at the end of Feature RN-5. It must be remembered that in some **joins,** both comparand columns may belong to a single relation. No pair of columns in a single relation are permitted to have the same name (Feature RN-3). Thus, a DBMS that supports this undesirable naming constraint may be unable to execute **joins** of a relation with itself using two distinct columns as comparands.

In contrast, the constraint on **joins** (see Chapter 4) that is part of the relational model—namely, that the comparand columns must draw their values from a common domain—guards against user errors in conceiving **join** commands without the adverse consequences just outlined.

## 6.2 ■ Naming Columns in Intermediate and Final Results

### *RN-6* Names of Columns Involved in the Union Class of Operators

In RL, when the user requests the operation R UNION S, he or she need not specify which columns of R are aligned with which columns

of S, except for those columns of R and S where two or more columns of R (or two or more columns of S) draw their values from a common domain. The same applies to **intersection, difference,** and the three outer counterparts: **outer union, outer intersection,** and **outer difference.**

Of course, the language RL does permit the user to specify which columns of R are to be associated with which columns of S whenever two or more columns of R or of S draw their values from a common domain (see Chapter 3).

One reason for using domains to determine associativity of columns in the **union**-type of operator is that this reduces the burden on the user and also reduces the occurrence of errors. If the degree of either operand is *N*, the user would be burdened with specifying *N* associations. Each association is a pair of columns, one column from one operand, one column from the other operand. A second reason for using domains in this way is that they ensure that the command is meaningful.

In supporting **union,** most existing relational DBMS products check no more than basic data types. This check is inadequate to ensure meaningfulness of the **union** operation, and can easily result in incorrect data in the database.

Consider the example of two relations A1 and A2 that identify and describe customer accounts pertaining to two different services provided by a company. Suppose that A1 and A2 have identical descriptions (see Table 6.1).

Note that there are five domains (extended data types) and six columns. The two currency columns and the days-of-service column all have the same basic data type, namely, non-negative integers.

***Table 6.1*** **Description of Relations A1 and A2**

| **Columns** | | **Domains** | |
|---|---|---|---|
| A# | Account number | Account numbers | A# |
| CNAME | Customer name | Company names | NAME |
| PDATE | Date of last payment | Calendar dates | DATE |
| PD1 | Year-to-date paid type 1 | U.S. currency | U |
| PD2 | Year-to-date paid type 2 | U.S. currency | U |
| SERV | Days of service | Days | D |

As shown in the following R-table, the five domains are

| A | A# (A# | NAME CNAME | DATE PD | U PD1 | U PD2 | DAYS SERV) |
|---|---|---|---|---|---|---|
| | c1 | Smith | 88-12 | 500 | 300 | 60 |
| | c2 | Jones | 89-01 | 800 | 0 | 105 |
| | c3 | Blake | 88-07 | 400 | 200 | 55 |
| | c4 | Adams | 88-10 | 1200 | 0 | 200 |
| | c5 | Brook | 88-08 | 150 | 150 | 35 |
| | c6 | Field | 87-12 | 120 | 200 | 30 |
| | c7 | Wild | 88-06 | 200 | 50 | 45 |

For successful action by the **union** operator, present versions of relational DBMS products merely require those columns that are paired off to have the same *basic data type.* This means that these DBMS would accept the following pairing of columns:

A1 ( A# CNAME PDATE PD1 PD2 SERV )
A2 ( A# CNAME PDATE SERV PD2 PD1 ).

The relational model, however, requires those columns that are paired off to have the same *extended data type.* Thus, it would not allow the SERV column of A1 to be paired with either PD1 or PD2 of A2. The model would allow PD1 of A1 to be paired with either PD1 or PD2 of A2. This safety feature is one of several in the model that carry some of the meaning of the data; such features are said to be *semantic.* I avoid applying the term "semantic" to the whole model, however, because this would be making a very extravagant claim.

## *RN-7* Non-impairment of Commutativity

Given any one of the relational operators that happens to have two operands and to be commutative, the rule built into the DBMS for naming the columns of the result must not impair this commutativity. Similarly, this naming rule must not impair any other simple identities that apply to the operators.

An example of a commutative operator is **union,** since (as just pointed out), for any pair of relations R, S,

$$R \cup S = S \cup R.$$

Thus, in this case, a rule that names the columns of the result in a way that depends on whether R or S is cited first in a relational command is unacceptable.

**Outer join** is an example of an operator to which a simple, but different, identity applies. For any pair of relations R, S, the **left outer join** of R on A with S on B yields the same result as the **right outer join** of S on B with R on A. One simple way to ensure that the DBMS supports Feature RN-7 is to design it to choose, from any two alternative names for a column, the name that comes first alphabetically using a standard collating sequence. This choice, however, would be troublesome for users who are unaccustomed to the Roman alphabet.

### *RN-8* Names of Columns of Result of the Join and Division Operators

When the user requests in RL a **join** (inner or outer) or a **relational division,** if (1) any one name of any pair of column names in the result is inherited from one operand of the command, (2) the other name is inherited from the second operand of the command, and (3) the two column names happen to be identical, then that name is in each case prefixed by the name of the relation that is the source of the column.

A feature of this kind is necessitated by the fact that no two columns of the result can have the same name.

### *RN-9* Names of Columns of Result of a Project Operator

The column names and sequencing of such names in the result of a **project** operator are precisely those specified in the pertinent command.

### *RN-10* Naming the Columns whose Values are Function-generated

A column whose values are computed using a function acquires a name composed of the name of the function followed by a period followed by the name of its first argument.

If the function has only one argument, that one is treated as its first argument. If two or more columns have values that are generated by the same function, and could be assigned the same name as a result, the DBMS resolves the potential ambiguity in names by assigning in each case a suffi-

ciently large substring of the function-invoking expression that ambiguity is resolved. Columns whose values are computed using an arithmetic expression (not an explicitly named function) are treated similarly.

Such a substring must exist; otherwise, the pertinent columns would be identical in content.

### *RN-11* Inheritance of Column Names

Every intermediate result and every final result of an RL command for interrogation or manipulation inherits column names from its operands (the **join** class of operators and the **union** class of operators), except for those columns covered by Feature RN-10. Such results also inherit column sequencing, except in the case of the **project** operator.

Rules for the naming by the DBMS of all columns in intermediate and final results are needed partly because of the rejection of positioning and nextness concepts in the relational model (see Chapter 1). It is worth remembering the following example: a single relational command may form the **union** of several **joins.** The user needs to know how the DBMS assigns names to columns of the **joins** (which are intermediate results) in order to be able to determine the desired alignment of columns when the **union** is executed.

## 6.3 ■ Naming Other Kinds of Objects

Data from a database can be archived, but only as one or more relations. Each of these relations can be base or derived. Most often, relations that are archived are derived relations. In either case, the archived relation has an *associated source relation*—that is, the relation whose name is alphabetically first of the one or more relations from which the archived relation is copied or derived.

### *RN-12* Naming Archived Relations

When archiving a relation, the user, normally the DBA, may choose to assign a name to it himself or herself; if not, the DBMS assigns a name. The name assigned by the DBMS is the name of the associated source relation concatenated with the eight-digit date of archiving (four-digit year first, then two-digit month, then two-digit day), followed by an integer *n* identifying the archived data as the *n*th version that day.

### *RN-13* Naming of Integrity Constraints

Each and every integrity constraint, regardless of its type, must be declared in the catalog and must be assigned a unique name.

This feature is necessary for the support of DBA-initiated integrity checks (see Feature RI-21 in Chapter 13). It is recommended that the naming of integrity checks be clearly distinguishable from the naming of domains, relations, functions, and columns. Note that, for a given primary key, there are likely to be many integrity constraints of the referential type. Each of these constraints must be given a distinct name.

The DBMS can make good use of these distinct names for integrity constraints when reporting on the failure of one of them. It should be remembered that a single row may contain two or more foreign keys. Thus, in the case of failure of referential integrity, it is insufficient to identify the row containing the foreign key that is giving trouble.

Frequently, a user begins his or her interaction with a database not knowing precisely what information he or she must retrieve from it. The user begins by posing some simple queries, and basing subsequent queries on information obtained from preceding ones. From time to time, it is necessary to treat results from preceding queries as operands in subsequent queries. This kind of querying is called the *detective mode* because detectives seeking information about criminal acts normally question witnesses and suspects in this way.

### *RN-14* Naming for the Detective Mode

A user's request for a query must include an option for the user to supply a name to be attached to the result of this query. If such a name is supplied, the DBMS checks that it does not conflict with any other names in its catalog, and, if so, stores the result of the query under the name supplied.

## Exercises

6.1 Must the name of each column in the entire database be distinct from the name of every other column in the entire database? If yes, discuss why. If no, discuss why not.

6.2 Must any two columns that are to act as comparands in a relational operation be identically named? Explain your answer.

6.3 Consider an **equi-join** of S with T. Assume that one of the columns of

S has the same name as one of the columns of T. What are the implied names of columns of the result? Use a simple example to explain your answer.

6.4 Why is it useful to include the domain name as a distinctive part of a column name?

6.5 How is the domain concept used in the **union** operator (1) to make the request more meaningful and (2) to reduce the column-pairing burden on the user?

6.6 What does naming have to do with possible impairment of commutativity?

6.7 When a relation S is archived and no name is supplied for this version by the user, how is that version named by the relational model? How does this feature relate to version support (where "version" means version of the data)?

6.8 Why should each integrity constraint be distinctly named?

## ■ CHAPTER 7 ■

# Commands for the DBA

The main purpose of the commands discussed in this chapter is to support certain tasks that are often the responsibility of the database administrator. Examples of such tasks are finding all occurrences of values in a specified domain (see Chapter 3); introducing new kinds of information into the database; loading and unloading R-tables from various sources (e.g., virtual storage access method files); archiving and re-activating R-tables; and creating, renaming, and dropping various parts of the database description. The features presented here do not specify the syntax that might be adopted; they are intended to convey the semantics.

Use of the commands described here requires special authorization, and is normally restricted to the DBA and his or her staff. These commands are not intended to support all of the tasks that are normally within the DBA's responsibility. Among such tasks not supported by these commands, and not supported in RM/V2, are changes in storage representation and in access paths to gain improved performance on the current traffic. Such changes are likely to depend heavily upon the design of the particular DBMS product involved. It is appropriate that these differences between DBMS products exist: different vendors may use quite different storage and access techniques in attaining good performance in the execution of high-level relational commands. The relational model remains unaffected due to its high level of abstraction.

Another typical task for a DBA or a security officer is assigning appropriate authorization to users so that they may access parts of the database

and possibly engage in data insertion, updating, and deletion. Support for this kind of task is included in RM/V2 (see Chapter 18, "Authorization").

Two of the commands for the DBA were introduced in Section 3.4 at the end of Chapter 3. These were the FIND commands (Features RE-1 and RE-2) for locating all occurrences of *all* active values drawn from any specified domain (FAO_AV) and locating all occurrences of *just those values* that occur both in a specified list and in a specified domain (FAO_LIST).

Of course, the term "locating" is used here in a sense that is meaningful to users of relational systems (see Chapter 3), and therefore has nothing to do with disk addresses as far as the user is concerned.

## 7.1 ■ Commands for Domains, Relations, and Columns

When dealing with domains and columns, it is useful to keep in mind the kinds of information declared for each one. Consider domain D; let col(D) denote the collection of all of the columns that draw their values from this domain. One aim is to include in the declaration of D every property that is shared by all of the columns in col(D). Then, the declaration of each column in col(D) need not repeat any of these common properties. It must, however, include the properties that are peculiar to that column, and these properties only.

Thus, a domain declaration normally includes the following:

- the basic data type;
- the range of values that spans the ranges permitted in all of the columns drawing their values from this domain;
- whether the comparator LESS THAN (<) is meaningfully applicable to such values.

A column declaration normally includes the following:

- an additional range constraint (if relevant) that provides a narrower range than that declared in the underlying domain;
- whether values are permitted to be missing from the column;
- whether the values in the column are all required to be distinct from one another.

For details, see Chapter 15, "The Catalog."

### *RE-3* The CREATE DOMAIN Command

This command establishes a new domain as an extended data type. (For more information on this topic, see Chapters 3 and 15.) The

information supplied as part of the command includes the name (selected by the DBA), the basic data type (as in programming languages such as COBOL, FORTRAN, and PL/1), a range of values, and whether it is meaningful to apply the comparator < to these values.

---

For example, it is often the case that applying the comparator < to part serial numbers is meaningless. Note that, if < is applicable, then so are all of the other comparators. That is the reason why only the comparator < is cited in Feature RE-1. Note also that the basic data type indicates whether arithmetic operators are applicable.

## *RE-4* The RENAME DOMAIN Command

This command re-names an already existing domain without changing any of its characteristics. The old name and the new name are supplied as part of the command. In addition, the DBMS finds every occurrence in the catalog of a column that draws its values from the specified domain (identified by its old name), and updates the name of that domain in the column description.

---

Since large parts of the catalog may have to be locked during the latter process, the DBA would be well advised to make this kind of request only during periods of low activity.

References by application programs to the cited domain by its old name are not automatically updated in RM/V2, but may be in RM/V3. There should be little impairment of application programs because normally these programs do not make direct reference to any domain.

## *RE-5* The ALTER DOMAIN Command

A suitably authorized user can employ this command to alter an already declared domain (extended data type) in various ways. An alteration of this kind is likely to impair application programs logically. Thus, such action must be undertaken with great care, and only when absolutely necessary. The items that might be changed are the basic data type, the range of values, and the applicability of <.

## *RE-6* The DROP DOMAIN Command

This command drops an existing domain, provided no columns still exist that draw their values from this domain. If such a column still exists, an indicator is turned on to indicate that this is the case, and that the command has been aborted (see Feature RJ-7 in Chapter 11). If there is an index based on the specified domain (see Feature RE-15) and if that domain is dropped, then the index is dropped.

## *RE-7* The CREATE R-TABLE Command

This command stores the declaration for a base R-table or a view in the catalog. All domains cited in such a command must be already declared. Otherwise, the command is aborted and the domain-not-declared indicator is turned on (see Feature RJ-5 in Chapter 11).

The following information is supplied as part of this command.

- The name of the R-table.
- If it is a view, its definition in terms of base R-tables and other views.
- For each column, its name.
- For each column, the name of the domain from which it draws its values.
- Which combination of columns constitutes the primary key or weak identifier. (The weak identifier pertains to certain kinds of views only; see the discussion of **outer equi-join** in Chapter 5.)
- For each foreign key, which combination of columns constitute that key and which primary keys (usually only one) are the target. This item is vital for base R-tables, but less critical for views.

---

It would be helpful for a DBMS that uses indexes to establish a domain-based index on the domain of the primary key of the R-table being created, if such an index does not already exist. Remember that another R-table may already have a primary key on the same domain, and an index based on this domain. If the DBMS does not yet support domain-based indexes, but does support the more common type of indexes, then it would be helpful if the system created an index on the primary key. Automatic creation of indexes on the foreign keys, or corresponding expansion of existing domain-based indexes, should also be considered.

## *RE-8* The RENAME R-TABLE Command

This command renames an existing base R-table or a view. The DBMS then examines all view definitions and authorizations recorded in the catalog without deleting any of them. The purpose is to make changes from the old name to the new name wherever that relation is cited. The old name and the new name are supplied as part of the command.

References by application programs to the cited R-table by its old name are not automatically updated in RM/V2, although they may be in RM/V3. Of course, the catalog would have to be expanded to become more like what is usually called a *dictionary*.

The DBA would be well advised to use this command only during periods of low activity.

## *RE-9* The DROP R-TABLE Command

When a base R-table or a view, say S, is dropped, several parts of the database description may be affected: integrity constraints, views, and authorization constraints. It should be remembered that an integrity constraint may straddle two or more R-tables. Thus, such a constraint may involve not only the R-table S, but also one or more other R-tables. The definition of a view may also cite several R-tables, of which S is only one. It may also be necessary to drop a bundle of authorization constraints based on the R-table.

The total effect of a normal drop of a specified R-table, say S, is abandonment of three types of specifications.

1. All of the integrity constraints citing S.
2. All of the views whose definitions cite S.
3. All of the authorization constraints citing S.

Collectively, the dropping of these specifications is called the *cascading action* that is expected from the DROP request, if such action is not explicitly postponed or avoided altogether. Type 1 applies principally to base R-tables, while Types 2 and 3 apply to both base R-tables and views.

Taking all of these factors into account, dropping such a table can cause a significant impact on users of that database. This action therefore requires special authorization (see Features RA-5 and RA-6 in Chapter 18). Normally, only the DBA and his or her staff are so authorized.

Sometimes the aim of the user is to replace the dropped R-table by other tables, preserving the integrity constraints, views, and authorization constraints. The sheer bulk of these items makes them worth preserving, even if they need minor editing. For this purpose, RM/V2 provides the *catalog block* (Feature RM-7 in Chapter 12) to postpone the cascading actions. The catalog block is a sequence of commands, each of which operates on the catalog only. Certain ones of these commands may normally have a cascading effect. This cascading action is postponed. It is the catalog that is allowed to leave a state of integrity during execution of the catalog block. The postponed cascading is re-examined by the DBMS at commit time to see whether any of it must be re-executed immediately prior to the execution of the commit that terminates this catalog block.

For safety reasons, the DROP R-TABLE command is executed in three steps. However, only Step 1 is applied if the R-table is not a base relation.

In the *first step,* the DBMS checks that the table name is recorded in the catalog as either a base relation or a view. If the specified R-table happens to be only a temporary R-table, it is immediately and unconditionally dropped.

If the relation being dropped is a base R-table or a view, the DBMS checks to see whether the catalog block indicator (Feature RJ-11 in Chapter 11) is on. If it is, the DBMS drops the relation and omits any cascading action.

If RJ-11 is off, the DBMS checks to see whether there is any potential cascading action. If not, the DBMS again drops the relation. If there is potential cascading action, the user is warned of the type of such action, and notified that the cascading action can be postponed by requesting a catalog block. If the user responds "go ahead anyway," the DBMS not only drops the relation, but also takes all of the necessary cascading action. On the other hand, if the user requests that the command be aborted, the DBMS cancels its attempt to drop the specified R-table. This ends Step 1.

In the *second step,* applicable to base R-tables only, if the DBMS has decided to initiate the drop procedure, it archives the specified R-table for either a specified or a default period. This period is at least seven days (the default value). See Features RA-5 and RA-6 in Chapter 18 for the authorization aspects.

Upon expiration of the archiving period, the DBMS takes the *third and final step,* applicable only to base R-tables by deleting all rows of the data in the specified R-table. It then drops the description of that R-table from the catalog. At any time during the archiving period, the DBA can restore the R-table to its state immediately before the execution of the DROP request.

### *RE-10* The APPEND COLUMN Command

This command specifies the name of an existing base R-table. The DBMS appends to the description of that table in the catalog the name supplied for a new column that draws its values from an already declared domain; the name of this domain is also supplied as part of the command. Each row of that table is extended to include a value for the named column. For the time being, however, each such value is A-marked as missing, unless the VALUE qualifier RQ-13 (see Chapter 10) is specified in the command.

The VALUE qualifier is one way of handling the case in which missing values are prohibited. Another way is by utilizing Feature RI-19 (see Chapter 13).

The domain cited in the command imposes certain constraints upon the values permitted in the new column. Additional constraints for the new column may be imposed by means of column-integrity assertions. Both domain integrity and column integrity are defined and discussed in Chapter 13.

### *RE-11* The RENAME COLUMN Command

This command renames an existing column of some existing R-table. The name of the pertinent R-table, the old name of the column, and the new name of this column must be supplied. If an index has been created on this column, any reference within the DBMS to this column by its old name is updated.

References by application programs to the cited column by its old name are not automatically updated in RM/V2, but may be in RM/V3.

### *RE-12* The ALTER COLUMN Command

Occasionally, it may be necessary to make changes in the properties assigned to a column. For example, for a specific column, the DBA may decide to change from one domain to another or to alter the range of values permitted in the column.

### *RE-13* The DROP COLUMN Command

This command makes those component values in each row that fall in the specified column inaccessible to all users. These component

values are actually removed, but at a reorganization time that is convenient for the DBMS.

---

Except for the special cases discussed next, this command then drops from the description of the pertinent R-table both (1) the column name and (2) the column description, including any reference therein to its domain.

If it happens that the column being dropped is part of the primary key of that R-table, the DBMS requests that a new primary key be declared. Since that and other R-tables may include numerous foreign keys drawn from the same domain as the primary key being dropped, the possibility of cascading action exists. Use of the catalog block may be appropriate in order to postpone any cascading action and to update these foreign keys.

If the column being dropped is part of a foreign key, the foreign-key declaration is dropped. If the column is simply indexed, the corresponding index is dropped. In the case of a domain-based index, only the contribution from this column is dropped.

References by application programs to the dropped column are not automatically found and reported by RM/V2, but may be by RM/V3.

## 7.2 ■ Commands for Indexes

Features RE-14–RE-16 apply only to relational DBMS that exploit indexes to attain good performance. The DBMS designer should remember that indexes in the relational context are tools for obtaining improved performance, and they should be used *for that purpose only*. In early releases of some relational DBMS products, uniqueness of values within a column could be accomplished, only if that column was indexed. Consequently, if the DBA dropped that index, the control over uniqueness of values was lost. Therefore, for these releases performance could not be the sole criterion for choosing whether a column is indexed. In the context of the relational model this coupling with the DBMS of semantic properties of the data with performance in making index decisions is an abuse of the index concept and a DBMS design error.

Uniqueness of values within any column should be specified as one of the properties of that column, not as a property of an index. Similarly, the kinds of marks permitted or prohibited in any column should be specified as a property of the column, not as a property of an index.

DBMS products with other kinds of performance-oriented access paths should have DBA commands similar to Features RE-14–RE-16.

### *RE-14* The CREATE INDEX Command

This command is intended to be designed into a relational DBMS that exploits indexes. It creates the description of an index and

stores this description in the catalog. It also creates the index, although not necessarily immediately. If the DBMS receives several successive requests for indexes to be created on a single relation, it attempts to process them all in a single pass over the data. The purpose is improved performance.

### *RE-15* The CREATE DOMAIN-BASED INDEX Command

This command is also intended to be designed into a relational DBMS that exploits indexes. It creates an index based on the specified domain. It provides the DBMS with the storage location of each active value drawn from this domain. Such an index refers to all of the columns in the database that draw their values from the specified domain or a subset of these columns, provided such a subset can be specified conveniently by the DBA. An index of this kind may therefore straddle two or more base R-tables.

When such an index is applied to a primary domain, it yields improved performance not only on retrieval of data, including the evaluation of **joins,** but also on referential integrity. When the database is distributed and a domain-based index is created on a primary domain, that index should exist at the site or sites where the primary keys are located.

### *RE-16* The DROP INDEX Command

This command is also intended to be designed into a relational DBMS that exploits indexes. It drops an existing index, whose name is supplied, or reports the non-existence of an index with that name.

## 7.3 ■ Commands for Other Purposes

### *RE-17* The CREATE SNAPSHOT Command

A query is embedded in this command. The query part yields a derived R-table, whose name is supplied as part of the command. The DBMS stores this derived R-table in the database, and stores its description (including the date and time of creation) in the catalog.

Suppose that a snapshot is created from a base relation S. Unlike a view, a snapshot of S does not reflect the insertions, updates, and deletions applied to S after the snapshot was created.

This command is likely to be used heavily in the management of distributed databases. For example, if a bank has branches in several cities and a computer-managed database in each city, the planning staff at headquarters might require a weekly snapshot of the accounts data in each city. These snapshots contain data that could be as much as a week out-of-date, whereas a view would reflect all of the transactions at the branches as they occur, and therefore be much more up-to-date. Snapshots, however, are much cheaper than views, and place a much smaller load on the communications network.

### *RE-18* The LOAD AN R-TABLE Command

This command invokes a user-defined loading program in accordance with a name specified in the command. The function may, and very likely will, convert the data from a non-relational form into a relation.

### *RE-19* The EXTRACT AN R-TABLE Command

This command can unload a copy of a relation (base or derived) in whatever form the DBMS delivers the relation. Alternatively, it can invoke a user-defined unloading program by including its name. This program may (and very likely will) convert the unloaded data from a relation into some non-relational form.

---

If no ordering of columns is explicitly specified, it may be useful to order the columns alphabetically by column name. This procedure will improve communications between two or more DBMS, possibly at different sites, and between a relational DBMS and non-relational recipients.

The following two features would be good options on the load utility. If supported in such a utility, they need not be supported in the DBMS itself.

### *RE-20* The CONTROL DUPLICATE ROWS Command

This command has as its single operand a table that may have duplicate rows in it. In other words, the operand need not be a true relation. It generates a true relation (an R-table) that contains only those rows of the operand that are distinct with respect to each

other, and appended to each such row is the number of occurrences of that row in the operand. The column that contains these counts is named by the DBMS as column ZZZ or given some equally unlikely name.

Its principal use is on tables loaded from non-relational sources. Such tables are likely to contain duplicate rows.

---

An example of the usefulness of the CONTROL DUPLICATE ROWS command can be found in supermarkets. The customer selects a wide variety of items from the supermarket shelves and places them in a shopping cart. Then he or she pushes the cart to a check-out line to enable a cashier to accumulate the bill and complete the transaction.

The cashier takes each item one by one and draws it across a device that electronically reads the bar code on the item. For speed of execution of check-out, it is important that it be unnecessary for either the cashier or customer to have to arrange items in any specific sequence. Thus, if the customer happens to have five cans of tuna fish randomly scattered in his or her shopping cart, it is highly unlikely that these cans are drawn across the bar code reader consecutively. Thus, it requires more than a bar code reader to add up the cans of tuna fish.

Suppose the bar code readings are automatically entered into and recorded in a computer system, partly for the purpose of adding up the bill for the complete transaction, and partly to insert this new information into a database that keeps track of inventory and, from time to time, places orders with one or more wholesale suppliers of food and housewares. Suppose also that, as the bar code for each item is entered into the computer, the computer converts it into a digital code, searches a table for descriptive properties of the item (including its price), and records the digitized bar code and properties as one more row in a table.

Such a table is bound to contain duplicate rows from time to time. For example, there are likely to be five separate rows for the five cans of tuna, but these rows are duplicates of one another and therefore not distinguishable rows. The question arises: how can duplicate rows be avoided if the database is relational?

Resolution of this question depends on what distinctions and identifications the supermarket manager deems to be useful for his or her business. One possibility is that the manager wants to keep track of what purchases are frequently coupled together in customers' habits. This requirement suggests that the collection of items purchased by one customer in one transaction be kept separate from those purchased by another customer in another transaction.

Does this mean that each customer should be required to provide his or her social security number to the cashier? Certainly not: such a requirement would be unacceptable to most customers. In addition, the manager is not likely to be interested in *identifying* each customer uniquely. Thus,

the social security approach represents a serious confusion between *distinctiveness* and *identification.*

The following is one solution to this problem, and I am not claiming that it is the best. However, it does avoid corrupting a relation by storing duplicate rows within it.

What has to be maintained as a distinct collection of records or rows is the collection of items purchased in a single customer transaction. To maintain this distinctiveness does not require unique identification of each customer. Instead of placing any burden on the customer to identify himself or herself uniquely, it is the system that should bear an equivalent burden, namely that of attaching the cash register identification and time of day to each transaction.

If the system initially records each and every item of a customer's transaction in a table, then duplicate rows should be removed from this table before it is planted in the database. Of course, this removal of duplicate rows must avoid loss of information.

The transformation needed is one that counts the number of occurrences of each distinct row and develops a revised table in which each row is distinct from every other row, and each row contains the number of occurrences of its counterpart in the initially generated table. This means that the transformation needed is precisely that provided by the CONTROL DUPLICATE ROWS command.

## 7.4 ■ Archiving and Related Activities

From time to time data must be removed from disk storage, because it has become inactive in the database (either completely or almost completely inactive). The main reason for this is to avoid the expense of having the entire database consume too much disk storage. However, either for government reasons (e.g., tax audit) or for business reasons (e.g., internal audit), the data thus removed must normally be saved in inactive status for a certain period in an archive. Such an archive is usually supported in a storage medium with very large capacity and relatively slow access—for example, by recording the data on magnetic tape. Sometimes it is required that archived data be reactivated in a separate database for use by analysts, planners, or accident investigators.

The DBA needs to plan this archiving and reactivating of data so that it becomes a routine activity handled by the DBMS: an activity that is repeated at various intervals specified by the DBA. Some data in the database may have a very short period of activity, while other data may have a very long period of activity. For simplicity and adequate generality, RM/V2 permits *any derived relation* to be archived. Note the emphasis on any derived relation. In a relational database a derived relation may consist of any combination of rows, providing they are all of the same extended data type.

An archived relation may later be reactivated in the same database from which it came or in some other database. Alternatively, an archived relation may be dropped altogether. Reactivation in a different database is quite likely whenever a company-related accident occurs in which employees or members of the public are injured. Reactivation in a different database is also quite likely if the data reflects operations that the company requires to be regularly analyzed "off-line" for planning purposes.

In RM/V2 actions such as archiving, reactivating, and dropping may be triggered by a calendar event, by a non-calendar event, or by the expiration of a specified period of calendar days after the occurrence of some specified event. The actions and their triggering conditions are specified by the DBA as relational commands. Since these requests concern the community of users, it is inappropriate to incorporate them in an application program. Instead, the DBMS stores these commands in the catalog. Further examination of these requests and their triggering conditions must be postponed until user-defined integrity constraints are considered (see Chapter 14).

As a general rule, relational requests such as retrievals, insertions, updates, and deletions, do not touch the archived data, and are therefore unaffected by it. The following two requests, however, do involve the archived data in a very explicit way.

### *RE-21* The ARCHIVE Command

This command stores a specified R-table in the archive storage, and attaches to it either the specified name or, if such a name is not supplied, the name of source R-table with the present date appended (see feature RN-12 in Chapter 6). It also attaches the name or identification of the source database from which it was archived.

This command is normally applied to a derived R-table. If an R-table with the same name already exists in archive storage, it is over-written by the new version.

### *RE-22* The REACTIVATE Command

This command, invoked for an R-table that was previously archived, copies the specified R-table from archive storage into storage that is more readily accessible. The reactivated copy becomes part of the database specified in the REACTIVATE command. Alternatively, if the database name is omitted, it once again becomes part of the source database. If in the process an R-table with the same name (including archiving date where applicable) is encountered in the receiving database, it is over-written.

Additional commands for enabling the DBA to maintain better control over the integrity of the database are discussed in Chapter 13 (Feature RI-21) and Chapter 14 (Features RI-31, RI-32, RI-33).

## Exercises

7.1 What relational objects can you create, rename, and drop using the DBA commands?

7.2 Why are the following two properties considered to be columnar, rather than domain-oriented?

1. Whether values are permitted to be missing from the pertinent column.
2. Whether all the values in the column are required to be distinct from one another.

Give examples of a type of domain, and two types of columns based on it, to support your argument.

7.3 Consider the task of appending new columns to two of the base relations. Is it necessary first to bring all the traffic to a halt? If not, explain how RM/V2 handles this problem.

7.4 Are there any commands in RM/V2 for keeping indexes consistent with whatever columns of data are indexed? Can semantic properties, such as uniqueness of values or keyhood, be associated with indexes? Explain your answer.

7.5 What is a domain-based index? How can it help improve the speed of execution of joins and of referential-integrity checks? (See also Chapter 13.)

7.6 If the occurrence of duplicate rows within a relation is banned by the relational model, why is the CONTROL DUPLICATE ROWS command needed (1) to check the existence of duplicate rows and (2) to remove the redundant duplicate rows?

7.7 What is an important reason for requiring that a user-defined function comes into effect during loading or unloading data? Does RM/V2 require such a function to be invoked when using the LOAD or EXTRACT command (Features RE-18 and RE-19), or is this optional?