■ CHAPTER 22 ■

Principles of Design for Relational Languages

The question has often been asked, "Why not extend a relational language to become a general-purpose programming language?" More recently, often the following is asked: "What are you going to do about image data and the large numeric arrays that are common in science and engineering?"

Early in my work on database management, I decided not to try to modify any of the well-established programming languages, such as FORTRAN, COBOL, and PL/1, to include the kind of statements needed in database management. Experience in dealing with the standards committees for these languages had convinced me that the members were not very interested in technical issues of any depth. Thus, I concentrated on database-oriented sublanguages, languages intended for every aspect of database management only. The term "sublanguage" clearly indicates that the language is specialized, and is not intended to support the computation of all computable functions.

This direction has proved to be sound. It was subsequently pursued with great success by the developers of text editing on micro-computers. After all, how many secretaries would now be using text editors if they had to learn COBOL or PL/1 first?

Programming languages such as PL/1 and ADA have reached a total complexity that is staggering. Therefore, it seems reasonable to predict that there will be continued growth in specialized sublanguages, each capable of being used interactively alone, and of communicating with programming languages of a more general nature in order to participate in application programs.

The processing of image data and large numeric arrays, especially matrices, involves many specialized operators and functions. Two examples of specialized operators are the inversion and transposition of matrices, while the computation of their determinants involves a specialized function.

As a user option, it should be possible to embed the corresponding retrieval expressions and functions in the target part or the condition part of a request in a relational language. It should not be necessary, however, for a commercial or industrial user to learn about these separable concepts unless he or she needs them on the job. Thus, a relational language should be designed to *accommodate* a wide variety of specialized operators and functions, not all of which can be anticipated at the time the language is designed.

In Features RL-1–RL-17, following, the abbreviation RL continues to denote whatever relational language is the principal one supported by a relational DBMS.

RL-1 Data Sublanguage: Variety of Users

RL is a data sublanguage intended to support users of all types, including both programmers and non-programmers, in all logical aspects of managing databases. RL contains no commands for branching, looping, manipulating pointers, or manipulating indexes.

The reasoning behind this feature is that the introduction of commands of this type would convert RL into an overly complicated language usable by programmers, but by hardly anyone else.

RL-2 Compiling and Re-compiling

RL commands must be compilable separately from the host-language context in which they may appear. The DBMS must support the compilation of RL commands, even if it also supports interpreting them. Moreover, the DBMS must support automatic re-compilation of RL commands whenever any change in access paths, access methods, or indexing invalidates the code developed by a previous compilation.[1]

RL-3 Intermixability of Relational- and Host-language Statements

In application programs, statements in RL can be freely intermixed with statements in the host language.

[1]This feature of RM/V2 is based on work done by Raymond Lorie while he was a member of the System R team in IBM Research, San Jose.

RL-4 Principal Relational Language Is Dynamically Executable

Any request expressed as an RL command can be pieced together as a character string using RL and/or HL. This character string can then be compiled and executed either immediately or later (user's choice) as if it were a command that has been entered from a terminal.

RL-5 RL Both a Source and a Target Language

RL is designed for two modes of use, as a source language and as a target language. In the first mode, statements must be easy for human beings to conceive correctly. In the second mode, statements must be easy for a computer-based system to generate.

Frequently, a language is designed as a source language only. However, almost every source language in the computing field becomes a target language for software packages on top. This is particularly true for relational languages because many services must make use of the information in relational databases.

RL-6 Simple Rule for Scope Within an RL Command

The scope of operators, comparators, functions, logical connectives, qualifiers, and indicators within any single RL expression or command must conform to a simple and readily comprehensible rule.

RL-7 Explicit BEGIN and END for Multi-command Blocks

The scope of multi-command blocks such as the transaction block and the catalog block (see Feature RM-7 in Chapter 12) must be explicitly stated by a BEGIN type command and an END type command—except where the intended extent of the block is a single command, in which case both the BEGIN and the END are omitted. Each of the commands BEGIN and END includes some identification of the type of block.

The SQL of prototype System R [Chamberlin et al. 1981] had both an explicit BEGIN and an explicit END for transactions. The IBM product DB2 fails to support the BEGIN command; this means that it is difficult to pin down the scope of a transaction block.

RL-8 Orthogonality in Language Design

The features of RL are expressed orthogonally. When a semantic feature is supported in one context, it should be supported in every RL context in which it can be interpreted sensibly and unambiguously. No single semantic feature is expressed in two or more distinct ways with the choice of expression being context-dependent. Wherever a constant can appear, an expression can replace it, provided it yields a value that is type-compatible with the type needed in the given context.

This feature means that there is no unnecessary coupling of one feature with another. Moreover, to take an example violated by SQL, the mode of expressing the removal of duplicate rows from the result of an operation should not be dependent upon whether the operation is **projection** or **union.** In the case of **projection,** SQL requires the presence of the qualifier DISTINCT. In the case of **union,** SQL requires the omission of the qualifier ALL. Of course, duplicate rows should not have been supported as an option in the first place.

RL-9 Predicate Logic versus Relational Algebra

RL is more closely related to the relational calculus of the language ALPHA [Codd 1971a] than to the relational algebra. The purpose is to encourage users to express their requests in as few RL commands as possible, and hence improve the optimizability of these RL commands taken one at a time.

To support this feature, RL must include **join** terms as in ALPHA [Codd 1971a]. RL must also include a simple way of expressing the universal quantifier. This avoids the kind of circumlocutions and circumconceptions that require the user to translate

FOR_ALL x Px

into

NOT THERE_EXISTS x NOT Px.

Such a translation must be made by users of the SQL of IBM's current release of DB2, which lacks a simple way to express relational division.

RL-10 Set-oriented Operators and Comparators

RL includes certain set-oriented operators such as **set union, set intersection,** and **set difference.** These operators are subject, of course, to the usual constraints of the relational model that the operands and the results must be relations that are type-compatible with one another. RL also includes set comparators such as SET INCLUSION. These set operators and set comparators must be defined, at least in technical papers available to the public, in terms of the predicate logic supported in RL.

The requirement for definitions in terms of the predicate logic within RL is based on the following reasons:

- to simplify the interface between the DBMS and inferential systems designed to operate on top of the DBMS;
- to simplify the DBMS optimizer by making it easy, as a first step, to convert any query or integrity constraint into a canonical form.

The second reason is important in attaining good performance no matter how the user chooses to express his or her needs. In this way, the burden of gaining good performance is transferred from the user to the DBMS, where it belongs.

RL-11 Set Constants and Nesting of Queries Within Queries

RL includes certain relation or set constants, such as the empty set. It may also include, but is not required to include, the nesting of subqueries within queries, as in SQL. However, unlike the present SQL of either ANSI or IBM, if nesting is supported, it must be defined in terms of simple basic expressions of predicate logic, and it must be an optional way of expressing the request, not a required way under any circumstance.

The last two sentences in Feature RL-11 are vital to ensure effective optimization that gives the best performance regardless of the way any condition is expressed. They also simplify the task of software vendors developing software based on the DBMS as a platform. For example, such

vendors may want to develop inferential packages, such as expert systems, that must interface with the DBMS using RL.

The introduction of set-oriented operators and comparators, plus set constants and nesting, frequently makes complex logical conditions easier to express, and therefore supports user productivity. On the other hand, one of the penalties of this introduction is that a given condition can now be expressed by the user in several different ways. In this case, it is extremely undesirable to burden users or software (on top of the DBMS) with the task of selecting a specific one of these ways so as to obtain the best performance with respect to the database traffic, storage representations, access paths, and access methods currently in effect.

RL-12 Canonical Form for Every Request

There must be a single canonical form for every request expressed in RL, whether interrogative or manipulative. Thus, no matter how a user chooses to express a query or manipulative action, the first step taken by the DBMS is to convert the source request into this canonical form.

This is another feature intended to enable the DBMS to assume the whole burden of finding the most efficient way to handle the request. The programmer or interactive user is then left with the sole task of determining how to express his or her logical and semantic needs.

RL-13 Global Optimization

RL statements are compiled or interpreted into target machine language. The optimization is carried out entirely within the DBMS. It is not split into a sequence of local suboptimizations such that the total sequence is less optimal than a corresponding single global optimization.

This global optimization includes at least the following:

- determining the alternative sequences in which the relational operations can be correctly executed;
- for each such sequence, selecting access paths that yield the least possible use of resources for that sequence, given the access paths currently in effect;
- finally, selecting that combination of a sequence of operations and pertinent access paths—a combination that yields overall the least possible use of resources.

In some products, optimization is implemented partly in the DBMS and partly in separate, additional software packages. As a consequence, performance suffers. Usually, a sequence of two or more local optimizations is not as good as a single global optimization.

RL-14 Uniform Optimization

For any query or data manipulation, whenever RL permits that activity to be expressed in two or more alternative (but logically equivalent) RL statements, the optimizer converts them to a canonical form, thereby ensuring that these alternatives yield target codings that are either identical or equally efficient.

This feature is intended to relieve users of the burden of selecting detailed source coding in RL for performance reasons. If not removed, such a burden would significantly reduce the adaptability of the relational approach to changes in storage representation and access paths resulting from changes in the total database traffic or from changes in the statistics of data in the database.

This feature also applies to distributed database management (see Chapters 24 and 25).

RL-15 Constants, Variables, and Functions Interchangeable

Wherever a constant can occur in an RL command, it can be replaced by an RL or host-language variable of a type suited to the context. Wherever an RL or host-language variable can occur, it can be replaced by an expression invoking a function (either built-in or user-defined), provided that function yields a result of a type suited to the context. In both cases, the substitution must yield a clearly meaningful and unambiguous command.

Regarding user-defined functions, see Chapter 19.

RL-16 Expressing Time-oriented Conditions

Time-oriented conditions can be included in any condition specified in an RL command, along with any other conditions that may be specified and oriented toward database content.

For example, RL would handle the requirement that a particular authorization be in effect for some specified interval of the day by attaching a time-oriented condition to the authorization command. See Feature RA-13 in Chapter 18, "Authorization."

RL-17 Flexible Role for Operators

In a relational command, any one of the basic operators can yield either an intermediate or a final result. More specifically, no basic operator is excluded from use in either of the following roles: (1) subordinate within an RL command to any other basic operator, or (2) superordinate within an RL command to any other basic operator.

SQL does not conform to this requirement because **union** cannot be used in a subordinate role to a **join,** although **join** can be subordinate to **union.** In other words, within a single SQL command the **union** of two or more **joins** can be requested. The **join** of two or more **unions** cannot be requested. This kind of complexity places an unnecessary load on the user's memory and makes SQL a frustrating tool to use.

In the design of a relational language, it is clearly desirable to make as few distinctions as possible between the interactive use of that language at a terminal and its use as a language for application programming. An example of a departure from this principle is the asterisk of SQL which is trouble-free when used interactively, but not trouble-free when used in application programs.

In SQL an asterisk can be used to denote all columns of a specified relation. This asterisk is intended to alleviate the burden of naming every column whenever the need arises for all columns to be involved in a query or manipulative command.

Interactive use of this feature of SQL at a terminal appears to present no special problem. Incorporation of an SQL asterisk in an application program is, however, another matter entirely. The asterisk damages the immunity of the program to such changes as the addition of new columns to the relation and the dropping of columns that already participate in that relation. This difficulty stems largely from the need to interface relational DBMS to old, single-record-at-a-time, host languages such as FORTRAN, COBOL, and PL/1. These languages deal with record structure in a rather inflexible way.

The 17 design principles introduced in this chapter are aimed at cleanliness of design and ease of use by the whole community of users. They are *not* aimed at reducing either the creative originality of individual implementors or the degree of competition among their respective companies.

- RL-1 Data sublanguage: variety of users
- RL-2 Compiling and re-compiling
- RL-3 Intermixability of RL and host-language statements
- RL-4 Principal relational language is dynamically executable
- RL-5 RL is a source language and a target language
- RL-6 Simple rule for scope within an RL command
- RL-7 Explicit BEGIN and END for multi-command blocks
- RL-8 Orthogonality in language design
- RL-9 Predicate logic versus relational algebra
- RL-10 Set-oriented operators and comparators
- RL-11 Set constants and nesting of queries within queries
- RL-12 Canonical form
- RL-13 Global optimization
- RL-14 Uniform optimization
- RL-15 Constants, variables, and functions are interchangeable
- RL-16 Expressing time-oriented conditions
- RL-17 Intermediate result from any basic operator

Exercises

22.1 Why does the relational model treat the language aspect of database management as a separate sublanguage, instead of promoting the advancement of programming languages to include database management?

22.2 What does it mean for a sublanguage to be able to communicate well with a host language?

22.3 The features in a well-designed language are orthogonal with respect to each other. Give three examples, each illustrating a distinct and serious lack of orthogonality in SQL.

22.4 What are multi-command blocks, and how are their boundaries made explicit?

22.5 Which is preferable, a data sublanguage based on the relational algebra, or one based on predicate logic? Why?

22.6 Of two sublanguages based on predicate logic, which is preferable, one that uses tuple variables, or one that uses domain variables? Why?

■ CHAPTER 23 ■

Serious Flaws in SQL

Most of the database management systems now being introduced as products on the world market are based on the relational model. Each of these products supports the Structured Query Language (SQL) [IBM], or, more accurately, some version of this language. Very few of these versions, and possibly no two of them, are identical. In the early 1980s, the American National Standards Institute (ANSI) rapidly adopted their own version of SQL as a standard.

It appears that all present versions share the following three flaws:

1. they permit duplicate rows to exist in relations;
2. they fail to separate psychological features from logical features;
3. they fail to provide adequate support for the use of either three-valued or four-valued logic (i.e., logics with truth-values in addition to TRUE and FALSE).

The devastating consequences of these three properties are explained in this chapter. The following kinds of steps are suggested:

- steps that vendors should take to remedy the problems;
- precautionary steps that users can take to avoid severe difficulties before vendors take action; **and**
- steps to avoid compatibility problems when vendors make the necessary changes in SQL.

These remarks have no effect on the relational model, since SQL is not part of that model. Nevertheless, a discussion of SQL's major flaws may help the reader acquire an improved understanding of the relational model.

23.1 ■ Introduction to the Flaws

The criticisms of SQL in this chapter are certainly not intended to be interpreted as criticisms of the relational approach to database management. SQL departs significantly from the relational model, and where it does, it is clearly SQL that falls short. Neither are the criticisms intended to be interpreted as wholesale criticism of IBM's relational DBMS products DB2 and SQL/DS. Although both of these products support SQL, they are good products when compared with other products on the market today. The flaws are serious enough to justify immediate action by vendors to remove them, and by users to avoid the consequences of the flaws as far as possible.

What, then, are the flaws in SQL that have such grave consequences? We shall describe just three:

1. SQL permits duplicate rows in relations;
2. it supports an inadequately defined kind of nesting of a query within a query;
3. it does not adequately support three-valued logic, let alone four-valued logic.

My position on these three "features" is as follows:

1. duplicate rows within relations ought to be prohibited;
2. even though I am not totally opposed to nesting, it requires precise definition and extensive investigation *before being included in a relational language,* so that a canonical form can be established for all requests;
3. four-valued logic should be fully supported within the DBMS and its language.

Criticisms of SQL have been plentiful; many are cited in this book. See, for example, [Date 1987], in which 20 or more serious errors are listed. Date's article, however, does not deal with the three most serious flaws, which are the main focus of this chapter.

23.2 ■ The First Flaw: Duplicate Rows and Corrupted Relations

When the idea was introduced that relations could be perceived as flat files or tables, the converse notion was adopted by numerous people as a true statement—namely, that any flat file or table can be perceived as a relation. This converse is totally incorrect. The flat files and tables of the past were

highly undisciplined structures. Frequently, not all of the rows (or records) were required to have the same type, and duplicate rows were permitted. Design of relations became corrupted by this false idea.

Relations in the relational model and in mathematics do not have duplicate rows. There may, of course, be duplicate values within a column. I shall refer to relations in which duplicate rows are permitted as *corrupted relations*.

At first glance, permitting relations to have duplicate rows appears to be a disarmingly simple and harmless extension. When this extension was conceived, I indicated that, before any such extension was made, it would be necessary to investigate the effect of duplicate rows on the definitions of each and every relational operator, as well as on the mutual interaction of these operators. It is worth noting that I originally defined the relational operators and their mutual interaction assuming that relations had no duplicate rows (as in mathematics). In all of my subsequent technical papers on database management, I have continued to take this position.

Research into the effects of duplicate rows was simply not done by any prototype- or product-development group. Moreover, the problem was not addressed by the ANSI committee X3H2. It is now clear that the consequences are devastating.

23.2.1 The Semantic Problem

The *first and perhaps most important concern* is a semantic one: the fact that, when hundreds (possibly thousands) of users are sharing a common database, it is essential that they 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.*

The contention that the DBMS must permit duplicate rows if its statistical functions (such as SUM and AVERAGE) are to deliver correct answers is quite incorrect. Clearly, duplicate values must be permitted within columns. For example, it is impossible to rule out the following possibilities:

- two values of currency happen to be the same (for example, the cost of two distinct investments);
- two employees happen to have the same birthdate;
- two employees happen to have the same gender (male or female);
- the inventory levels for two distinct kinds of parts happen to be identical.

Consider two or more rows in some corrupted relation that happen to be duplicates of each other. One may well ask what the meaning of each occurrence of these duplicate rows is. If they represent distinct objects (abstract or concrete), why is their distinctiveness not represented by distinct values in at least one component of the row (the primary key component) as required by the relational model?

If they do not represent distinct objects, what purpose do they serve? A fact is a fact, and in a computer its truth is adequately claimed by one assertion: the claim of its truth is not enhanced by repeated assertions. In database management, repetition of a fact merely adds complexity, and, in the case of duplicate rows within a relation, uncontrolled redundancy.

Consider how data in a database should be interpreted by users. The main reason for establishing any database is that it is an organized (and hopefully systematic) way of sharing data amongst many users (perhaps hundreds, perhaps thousands). To make such sharing successful, it is necessary for all users to understand exactly one common meaning of all of the data they are authorized to access. "*Common*" in this context means common to all users, and shared by all of them. Now, if an operand or a result contains duplicate rows, there is no standard conception of what the duplicate rows mean. There may be some private (unshared) conception, but that is just not good enough for the successful management of shared data.

23.2.2 Application of Statistical Functions

A *second concern* is the correct application of statistical functions. It is often claimed that projection should not eliminate duplicate rows. To support this claim, an example may be cited in which information about employees is stored in a relation called EMP, and this relation has a SALARY column that contains the present salary of each employee. To provide an analyst with details of present salaries, but not the items that identify employees, the projection of EMP onto SALARY is claimed to be necessary as the first step. It is also claimed that, in this projection, duplicate salaries must not be eliminated, since then the analyst is likely to deduce wrong answers to those queries of a statistical nature.

Statistical functions in relational DBMS can and should operate in the context of relations that do not have duplicate rows. This means that the relation name as well as the column name are arguments for a statistical function applied to that column.

Each of the statistical functions built into the DBMS should have two flavors: one that treats each row as it occurs (just once) ignoring any degree-of-duplication component (if such exists); the other that treats each row as if it occurred n times, where n is the degree-of-duplication component of that row (see Section 23.2.4 and Chapter 19).

23.2.3 Ordering of the Relational Operators

A *third concern* is the interchangeability in ordering of the relational operators. When manipulating non-corrupted relations (duplicate rows *not* permitted) using the relational operators of the relational model, there is a high degree of immunity to the specific ordering chosen for executing these operators. To illustrate, let us consider the operators **projection** and **equi-join.** Suppose that the **projection** does not discard any of the columns whose

values are compared in the **join.** Then, provided no duplicate rows are allowed, the same result is generated whether the **projection** is executed first and then the **join,** or the **join** is executed first and then the **projection.**

Note that, if as usual the projection cites the columns to be saved (instead of those to be dropped), there must be a change of this list of columns depending on whether the **projection** preceded or followed the **join.** If, however, the **projection** cites the columns to be discarded, there need be no change in the list of these columns. Both forms of **projection** are useful.

This degree of immunity to the sequence of operators is lost when duplicate rows are permitted within relations. Consider an example involving **join** and **projection.** Suppose that duplicate rows are allowed in the result of **projection,** but not in the result of **join.** In SQL this means that the qualifier DISTINCT is used in the **join** command only.

R (A B C)

A	B	C
a1	1	c1
a2	1	c1
a3	1	c2
a4	2	c2
a5	2	c1

S (D E)

D	E
d1	1
d2	1
d3	2

Taking the **projection** R [B,C] first and retaining duplicate rows, we obtain the following result. Then let us take the **equi-join** of this relation, with S comparing column B with column E, permitting duplicate rows in the operands, but not in the result.

R [B,C] (B C)

B	C
1	c1
1	c1
1	c2
2	c2
2	c1

R [B,C][B = E] S (B C D E)

B	C	D	E
1	c1	d1	1
1	c1	d2	1
1	c2	d1	1
1	c2	d2	1
2	c2	d3	2
2	c1	d3	2

The final result has just six rows and no duplicate rows.

Now let us reverse the sequence of operators, executing the **equi-join** first to generate relation T, and then executing the **projection** of T onto B,C,D,E.

R [B = E] S (A B C D E)

A	B	C	D	E
a1	1	c1	d1	1
a2	1	c1	d1	1
a3	1	c2	d1	1
a1	1	c1	d2	1
a2	1	c1	d2	1
a3	1	c2	d2	1
a4	2	c2	d3	2
a5	2	c1	d3	2

T (B C D E)

B	C	D	E
1	c1	d1	1
1	c1	d1	1
1	c2	d1	1
1	c1	d2	1
1	c1	d2	1
1	c2	d2	1
2	c2	d3	2
2	c1	d3	2

The final result has eight rows, including two cases of duplicate rows. Clearly, when duplicate rows are permitted, the result obtained by executing the **projection** first and then the **join** is different from that obtained by executing the **join** first and then the **projection.** If duplicate rows had not been permitted, the results would have been identical, whichever sequence of relational operations was adopted. (The reader may wish to check this himself or herself.) What this example shows is that changing the sequence in which relational operations are executed can yield different results if the DBMS permits duplicate rows within a relation.

It is useless for an advocate of duplicate rows to dismiss the difference between these results as nothing more than two rows being duplicated—that suggests that duplicate rows are meaningless to the DBMS and to users. If duplicate rows have no meaning (Case 1), they should certainly be prohibited by the DBMS. If they have a meaning (Case 2), then this is surely a private (unshared) meaning, applicable only in some special context. There is no general meaning for duplicate rows that is accepted. Thus, once again, duplicate rows should be prohibited by the DBMS.

Another possible argument from the advocates of duplicate rows is, "Why not express the **projection** and **join** combined into a single SQL command? Then it will be impossible to use the qualifier DISTINCT on one of the operators without it becoming effective on the other."

A first reply to this is that one operator may define a view and the other a query on that view, and two users may have defined these items independently of one another. When executing such a query, it is the DBMS (and not a user) that combines the view definition with the query definition to make the query effective on base relations. A second reply is that the DBMS undoubtedly does not prevent a programmer from expressing these operators in separate SQL statements, whether one of the statements is a view definition or not.

It is worth noting here that, if the DBMS permits duplicate rows in results, it must also permit duplicate rows in operands because of the operational closure feature of relational database management systems: "The principal relational language is mathematically closed with respect to the operators it supports" (see Feature RM-5 in Chapter 12). This means that, in the principal relational language, the results of manipulative operations must always be legal as operands. If corrupted relations are permitted as results, then they must also be permitted as operands. This closure feature is intended to make it possible for users to make investigative inquiries in which it is occasionally necessary to use as operands the results of previous queries.

In case the reader thinks this is just an isolated example, let us look at a quite different one (communicated to me by Nathan Goodman) involving three simple relations, each concerned with employees—first their names, second their qualifications, and third their ages:

E1 (E# ENAME) E2 (E# QUAL) E3 (E# AGE).

As usual, E# stands for employee serial number. Using SQL, we can find the names of employees who have the degree Ph.D. or whose age is at least 50 (or who satisfy both conditions). One of the distinct ways in which this query can be expressed in SQL involves using logical OR. Another way involves using **union** on the serial numbers for employees that satisfy each of the conditions. *These two approaches should always yield the same result—but do they?*

The answer is that, if SQL is used, it depends on when and in what context the user requests that duplicate rows be retained or eliminated. If **union** ALL is used in this context, the result contains the names of employees duplicated whenever each employee satisfies *both* conditions (that is, he or she has the Ph.D. degree and is at least 50 years old).

The reduction in interchangeability of the sequence in which relational operations are executed can adversely affect both the DBMS and users of the DBMS. As we shall see, it damages the production by the DBMS of efficient target code (this process is usually called optimization) and substantially increases the user's burden in determining the sequence of relational commands, when the user chooses to make this sequence explicit.

Application programmers tell me they find it a confusing phenomenon that some **joins** yield duplicate rows, while others do not. They also tell me that for their applications it is both necessary and difficult to eliminate duplicate rows efficiently.

Optimization by the DBMS A relational command usually consists of a collection of basic relational operators. Part of the optimizer's job is to examine the various alternative sequences in which these basic operations can be executed. For each such sequence it determines the most efficient way of exploiting the existing access paths. Finally, it determines which of the alternative sequences consumes the least resources. Clearly, then, any reduction in the interchangeability of ordering of the basic relational operations will reduce the alternatives which can be explored by the DBMS, and this in turn can be expected to reduce the overall performance and efficiency attainable by the DBMS.

User's Burden in Choosing an Ordering of Commands Occasionally, the user may (for various reasons) express in two or more relational commands what could have been expressed in just one. For example, he may decide to express a projection in one command and a join in another command. Because the sequence of these commands can affect the ultimate result when duplicate rows are permitted, the user must give the matter much more careful thought than would have been necessary if duplicate rows had not been permitted. One consequence will be a proliferation of unnecessary bugs in programs and terminal activities.

The extra thinking and the extra bugs will undoubtedly cause an unnecessary reduction in the productivity of users. A far more serious consequence is that undiscovered bugs may lead to poor business decisions.

Violation of the Fundamental Law #1 As discussed in Chapter 29, the relational model is based on at least twenty fundamental laws. One of them is as follows: each object about which information is stored in the database must be uniquely identified, and thereby distinguished from every other object. This fundamental law is violated if duplicate rows are permitted in base relations. This is an important part of the job of maintaining the database in a state of integrity. The DBMS must help the DBA in this responsibility.

23.2.4 The Alleged Security Problem

It has been alleged that duplicate rows are needed to support correct statistical analysis of data in a relation that has a sensitive primary key—that is, a key that cannot be made available to the analyst. There are two ways in which this alleged need for duplicate rows can be avoided.

The first approach is by use of the degree-of-duplication (DOD) qualifier, which the DBMS should support. This qualifier enables the DBA to grant a user access to enough information from the database and enough computed information (appended to each row, a count of the number of occurrences of precisely similar rows) to enable him or her to make correct statistical analysis without access to some primary keys that happen to be sensitive. See Feature RQ-11 in Chapter 10 for more detail.

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

In the second approach, the DBMS vendor changes the authorization mechanism in the database management system. Suppose that a view is defined as a suitable projection that includes the primary-key column(s). The user is authorized to access all of the columns in this view, except the primary-key column(s) that are blocked by the DBMS. See Chapter 18 for further information on this topic, and on the approach the DBA and DBMS can take, if this feature is supported.

23.2.5 The Supermarket Check-out Problem

In 1988 I published my contention [Codd 1988b] that duplicate rows should be avoided altogether in a relational database management system. Shortly thereafter, attempted rebuttals began to appear. In one of these attempts [Beech 1989], the example of checking out a customer at a cash register in a supermarket was described. In this example, the customer had picked up five cans of cat food, and the cashier registered each one separately and not consecutively, a common occurrence in supermarkets.

In this supermarket all the cash registers were connected to a computer with a database management system, so that all purchases were recorded in

the database. The rebuttal claimed that the purchase of the five cans of cat food would have to be stored as five separate rows in the database, and that these five rows would be duplicates of one another.

What Beech and others seem to have overlooked is that part of the design of databases and rules concerning their use depends quite heavily on what the business managers consider worth recording. In particular, some semantic distinctions are beneficial to the business, while others are not. In the supermarket example, the manager of the supermarket is not likely to be interested in distinguishing one can of cat food from another if they are of the same brand. That same manager is, however, likely to be interested in determining the average number of cans of a particular brand purchased by individual customers, because he or she can then determine how much shelf space to allot to each brand. For this distinction it is necessary to distinguish each customer check-out from every other customer check-out.

Does this distinction require that the customer present a unique identification (such as a Social Security number) to the cashier? Certainly not. Does it require that each can be distinctly labelled even if it is of the same brand? Certainly not. The distinction can be made if each brand is distinctly labelled, and if each customer transaction (the purchase of all items by a particular customer) is somehow distinguished from every other customer transaction. This distinctiveness is easily achieved for transactions by means of the following steps:

1. The cash register must send its identifier automatically into the computer at the beginning of each transaction.
2. At this point, the computer must append the date and time.
3. When each item is entered by the cashier, the system must examine whether an identically identified item has been entered at some time before within the current transaction—if it has, the count of items of that brand is increased by one in the row pertaining to that brand; if it has not, a new row is recorded in the appropriate relation.

It is useful to refer to this kind of analysis as the *analysis of semantic distinctiveness*. This is an aspect of the meaning of the data that is strongly supported by the relational model. I have yet to encounter any other approach to database management that supports this aspect adequately.

Considering all the adverse consequences and incorrect allegations just cited, I still find that duplicate rows in any relation are unacceptable.

23.3 ■ The Second Flaw: The Psychological Mix-up

23.3.1 The Problem

As used here, the term "psychological" refers to what is often called the human-factors aspects of a language. The term "logical" refers to the logical

power of a language, especially the power achievable without resorting to the usual programming tricks, such as iterative loops.

Normally, if proper relations are employed, a manipulative command or query expressed in terms of nesting and using the term IN can be re-expressed in terms of an **equi-join.** Let us look, however, at an example involving corrupted relations. Suppose we are given the relations EMP and WAREHOUSE:

EMP (E#	ECITY)
E1	A
E2	B
E3	C

WAREHOUSE (WNAME	WCITY)
W1	A
W1	A
W2	D
W3	C
W4	E

In this example, EMP is intended to list all the employees by employee serial number and city in which the employee lives; WAREHOUSE is intended to list all warehouses by serial number and city where located. Suppose we wish to find each employee name and the city in which he or she lives whenever that city is one in which the company has a warehouse. One might reasonably expect that this query could be handled equally well either by an **equi-join** or by a nesting that uses the IN term as follows:

Equi-join

```
SELECT E#, ECITY
FROM EMP, WAREHOUSE
WHERE ECITY = WCITY
```

Nesting

```
SELECT E#, ECITY
FROM EMP
WHERE ECITY IN
      (SELECT WCITY
      FROM WAREHOUSE)
```

The results, however, are not identical:

Equi-join Result

E#	ECITY
E1	A
E1	A
E3	C

Nesting Result

E#	ECITY
E1	A
E3	C

Once again, we have a problem that arises in part from permitting duplicate rows. This case, however, is somewhat more complicated than the ones considered earlier. Whenever the DBMS encounters a query in nested form, it must transform such a query into a non-nested form in order to simplify the task of the optimizer. Some excellent work on this transformation has been done [Kim 1982, Ganski and Wong 1987].

There appear, however, to be two major omissions from the works just cited and from other related work. First, the question of duplicate rows is

not discussed. Second, even if duplicate rows were prohibited, the remaining question is whether the coverage in the work is complete with respect to *all nested versions permitted* in SQL.

My position on the nesting of SQL is that, when conceived in the early 1970s, it was an attractive idea, but one that needed careful scrutiny and investigation. This nesting was advocated by its proponents as: (1) a replacement for predicate logic in the relational world, and (2) a more user-friendly language than the preceding relational database sublanguage ALPHA [Codd 1971a].

The first-cited reason is simply not true. As time has elapsed, it has been found necessary to incorporate bits of predicate logic in the language, although errors have been made in this activity. The second-cited reason has some credibility, but would turn SQL into a curious mixture of the logical and the psychological aspects of a relational language. There are two reasons why these two kinds of aspects should be kept separate from one another:

1. a relational language must be effective both as a source language and as a target language because of the myriad of subsystems expected on top (e.g., application development systems, database design systems, expert systems and natural-language subsystems);
2. the relational approach is intended to serve a great variety of users, and therefore different users may have entirely different education, training, and background—this means that just one approach to psychological support is very unlikely to be adequate.

Accordingly, all of the statements in each of the several distinct languages providing *psychological support* should be translatable into the single language providing *logical support.* Until that translatability is demonstrated for SQL by means of a rigorous proof, serious problems in using that language will continue to arise.

Even when the translatability problem is solved, published, and implemented, there is the danger that a feature will be added to the nested queries that will introduce non-translatability. Theoretical investigation is sorely needed in this aspect of language definition.

While on the subject of nesting queries within queries, there are two features of IBM's SQL that I feel drastically reduce both the comprehensibility and the usability of that language. Let us illustrate these features by making small modifications to the examples concerning employees and warehouses (introduced earlier in this section).

Some city names occur several times in the United States, but only once in any selected state. For example, Portland occurs both in Maine and in Oregon. Suppose that to each relation, EMP and WAREHOUSE, we add a column pertaining to the state in which the city is located. Then let us try the following query:

```
SELECT E#, ECITY, ESTATE
FROM EMP
WHERE (ECITY, ESTATE) IN
  (SELECT (WCITY, WSTATE) FROM WAREHOUSE)
```

The DBMS refuses to handle this query, even though it is just like the original, except that in this case the IN clause involves a combination of columns instead of a single column. To a user, this seems totally inappropriate behavior for a DBMS. The ability of DB2 to concatenate the name of a city with the name of a state can be used to alter this query into one that can be executed. However, this is neither a general nor a natural solution to the problem.

Returning to the original relations, suppose that the query is altered to elicit more columns of data:

```
SELECT E#, ECITY, WNAME, WCITY
FROM EMP, WAREHOUSE
WHERE ECITY IN
  (SELECT WCITY FROM WAREHOUSE)
```

This time, the DBMS yields the **Cartesian product** of EMP with WAREHOUSE, except that rows that contain the cities that fail to qualify are excluded. This result is clearly not what was requested. Like the previous example, this kind of surprise is the hallmark of a poorly designed language.

23.3.2 Adverse Consequences

Optimization by the DBMS When the prototype System R [Chamberlin et al. 1981] was passed from IBM Research to the product developers, the question of whether SQL could be translated from a nested query to a non-nested version had not been investigated. Subsequently, when the IBM products DB2 and SQL/DS were built, the problem was found too difficult to handle in the optimizer. As a result, the first three releases of DB2 perform poorly on nested queries compared with non-nested queries. This is truly ironic, because SQL had been sold to IBM's management on the basis of its alleged ease of use and power due to the nesting feature.

User's Burden in Choosing Nested versus Non-nested The difference in performance between nested and non-nested versions of the same query puts an unnecessary performance-oriented burden on users, one that will not disappear until nesting is prohibited, or the translatability problem is completely solved and incorporated into DBMS optimizers. In nested as in non-nested queries, duplicate rows must be prohibited to avoid the additional burden of unexpected discrepancies in the results.

23.4 ■ The Third Flaw: Inadequate Support for Three- and Four-Valued Logic

DB2 is one of the few relational DBMS products that *represents* missing information independently of the type of the data that is missing—a requirement of the relational model and a requirement for ease of use. DB2 uses an extra byte for any column in which missing values are permitted, and one bit of this byte tells the system whether the associated value should be taken seriously or whether that value is actually missing.

DB2, however, completely fails to meet one more requirement of the relational model—namely, that missing information should be *handled* in a manner that is independent of the type of the information that is missing, and that the user should be relieved of the burden of devising three-valued logic. Representation of missing information is one thing, but handling it is quite another [Codd 1986a and 1987c]. Actually, three-valued logic is built into DB2, but is used only to pass over values for which the condition evaluates to MAYBE (neither true nor false). Thus, only one of the many uses of the fact that a value is missing is supported by DB2. This is mainly due to the weak treatment of missing values in SQL.

23.4.1 The Problem

How to Support Three-Valued Logic Usually the occurrence of cases of missing information in a practical database is unavoidable—it is a fact of life. I believe that, when interrogating a database for information, users prefer the DBMS as its normal behavior to take a conservative position and to avoid guessing the correct answer. Whenever the system does not know some requested fact or condition, it should admit a lack of knowledge.

The DBMS should also support, as exceptional behavior explicitly requested, the extraction of all the items that could satisfy a request if unknown values were replaced by information that yielded as many values as possible in the target list of the query.

A database retrieval may, of course, include several conditions like

DATE > 66-12-31,

where the DATE column has values of extended data type DATE, and

AMOUNT < 20,000,

where the AMOUNT column has values of extended data type U.S. CURRENCY. The conditions may be combined in many different logical combinations (including the logical connectives AND, OR, and NOT and the quantifiers UNIVERSAL and EXISTENTIAL). Suppose, as an example, that both expressions just noted participate in some condition. Also suppose that both columns are allowed to have missing database values. How does

the DBMS deal with a query involving the following combination of conditions,

(DATE > 66-12-31) OR (AMOUNT < 20,000),

where the date condition, the amount condition, or both may evaluate to MAYBE? Clearly, the DBMS must know the truth-value of such combinations as MAYBE OR TRUE, TRUE OR MAYBE, and MAYBE OR MAYBE. This means that the DBMS must support at least three-valued logic. If not, then the user must do the following:

1. request the primary-key values of those orders for which the DATE > 66-12-31 is TRUE;
2. request the primary-key values of those orders for which the AMOUNT < 20,000 is TRUE; **and**
3. request the **union** of the two sets generated by Steps 1 and 2.

In the case of AND instead of OR, the user would have to request in Step 3 the **intersection** of the two sets of primary keys. Users are liable to make numerous mistakes if they are forced to support three-valued logic mentally because the DBMS provides inadequate support. Who knows what crucial business decisions might be made incorrectly as a consequence?

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

In the following truth tables for the three-valued logic [Codd 1979] of the relational model RM/V1, the symbols P and Q denote propositions, each of which may have any one of the following truth values: t for TRUE *or* m for MAYBE *or* f for FALSE.

P	NOT P
t	f
m	m
f	t

P OR Q		Q		
		t	m	f
P	t	t	t	t
	m	t	m	m
	f	t	m	f

P AND Q		Q		
		t	m	f
P	t	t	m	f
	m	m	m	f
	f	f	f	f

In the relational model, the universal and existential quantifiers are applied over finite sets only. Thus, the universal quantifier behaves like the logic operator AND, and the existential quantifier behaves like OR, both operators being extended to apply the specified condition to each and every member of the pertinent set.

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

If to the entire condition part of this query we add the keyword MAYBE, then the result consists of all the cases in which this condition evaluates to MAYBE, and no other cases. The MAYBE qualifier is used only for exploring possibilities. Special authorization would be necessary if a user is to incorporate it in a program or a terminal interaction.

Actually, the relational model calls for the DBMS to support the attachment of the MAYBE qualifier to any truth-valued expression, since a view is normally defined not using this qualifier, while a query on the view may involve it. The normal action of the DBMS is to combine the view condition with the query condition using logical AND. This action, of course, would give rise to a more comprehensive condition involving the MAYBE qualifier attached to just one truth-valued expression within that condition.

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

(B < 66-12-31) OR (B = 66-12-31) OR (B > 66-12-31).

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

To avoid this type of error, there are two options:

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

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

Treatment of Missing Values in SQL The only concession in SQL commands to the existence of missing values is the clause IS NULL, which enables the user to pick up from any column those cases in which there are missing values. Flexible use of three-valued logic (let alone four-valued) is not

supported. An example of inflexibility is the action of DB2 when the condition part of a query is evaluated as unknown. It simply does not retrieve the corresponding instances of the target data. Although in practice this is one of the options that users need, they also need others. One such option is the temporary replacement of missing values by user-specified values, where "temporary" means just for the execution of the pertinent command.

Other DBMS products that fail to go beyond present SQL, whether it be the IBM or the ANSI version, are unable to provide adequate support of three-valued logic. As a result, we can expect users to make many errors, some of which are bound to go undetected.

A somewhat separate problem is the effect of missing values on aggregate functions. The relational model supports the following options: request the missing items to be ignored, or temporarily replace each missing item by a specified value, where "temporarily" again means just for the execution of the pertinent command. SQL appears to support only one of these options: it always ignores the missing items.

23.4.2 Adverse Consequences

Overall, the SQL approach to handling missing values is quite disorganized and weak. This will lead to disorganized thinking on the part of users, an increased burden for them to bear, and many unnecessary errors. Errors that are not discovered can lead to incorrect business decisions based on incorrectly extracted data.

The SQL approach also causes some users to wish for the old approaches like "default values" that were at least familiar, even if more disorganized. Of course, the old approaches are completely out of place in any DBMS based on the relational model.

In some cases of inadequate handling of missing information, the problem is incorrectly perceived to be a problem of the relational model. In fact, the problem stems from the inadequacies of SQL and its non-conformance to the relational model.

23.5 ■ Corrective Steps for DBMS Vendors

Let us discuss the three problems in turn. First, consider the problem of duplicate rows.

23.5.1 Corrective Steps for Duplicate Rows

This correction should be handled in three stages:

1. warn users that support for duplicate rows is going to be phased out in about two years' time;
2. within the first year, install in some new release a "two-position switch" (i.e., a DBA-controlled bit) that permits the DBMS to operate in two

modes with respect to duplicate rows: (1) accepting them and (2) rejecting them;

3. drop the support for duplicate rows within a relation altogether, and improve the optimizer accordingly.

With regard to loss of integrity from databases, it is well-known that prevention is much better than cure. For this reason, the DBMS should check that duplicate rows are not being generated whenever an operator is executed that could generate duplicate rows. Three of the operators that are defined to remove duplicate rows are **projection, union,** and **appending** rows to a relation, including initial loading. Most DBMS products today fail to conform to the definitions of these operators.

To provide assistance in the loading of data into relations from tables that may contain duplicate rows, the command CONTROL DUPLICATE ROWS was introduced as Feature RE–20 in Chapter 7. Using this command, the duplicate rows are removed without loss of information.

23.5.2 Corrective Steps for the Psychological Mix-up

The most recent version of IBM's SQL (even if the duplicate row concept has been removed) should be treated as a language that stands or falls on its psychological or ease-of-use properties. A new relational language should be created with features that are highly orthogonal to one another. The language should be readily extensible, include all of the logical properties necessary to manage a relational database, be readily compilable, and be convenient to use as a target language by all the software packages that interface on top of the DBMS.

23.5.3 Corrective Steps in Supporting Multi-Valued Logic

DBMS vendors should start the work required to introduce support for four-valued logic [Codd 1986a and 1987c]. The three-valued logic just cited is a sublogic of the four-valued logic. Implementing the four-valued logic is not noticeably more difficult or time-consuming than implementing the three-valued. The four-valued logic treats information that is missing for a second reason—namely, that a particular property happens to be inapplicable to certain objects represented in the pertinent relation. With adequate support for three or four-valued logic, the IS NULL clause in SQL becomes redundant and should be phased out.

23.6 ■ Precautionary Steps for Users

While these three flaws are being corrected by the DBMS vendors, there are several steps users can take to protect their databases and hence their business. The *first step* is to avoid duplicate rows within relations at all times

by insisting on continued adherence to the programming and interactive discipline:

- exactly one primary key must be specified for each base relation;
- the DISTINCT qualifier must immediately follow the keyword SELECT in every SQL command that includes SELECT;
- the ALL qualifier must never accompany any **union.**

The *second step* is to avoid nested versions of SQL statements whenever there exists a non-nested version. The *third step* is to take extra care in manipulating relations that have columns that may contain missing values, and as far as possible separate the handling of missing information into easily identifiable pieces of code that can be readily replaced later.

23.7 ■ Concluding Remarks

Is it too extreme to call these SQL blunders serious flaws? I do not think so, in view of the fact that more and more business and government institutions are becoming dependent on relational DBMS products for the continued success of their operations. In my view, the three flaws described in this chapter *must* be repaired, even though the repair may cause some users to have to change some SQL portions of their programs.

DBMS vendors should immediately begin putting the corrective steps outlined in Section 23.5 into effect. Such action could easily give them a substantial competitive advantage in the eyes of their prospective customers.

The proposed changes in SQL described in this chapter also represent a great opportunity for ANSI to take the lead.

Users are strongly advised to take the precautionary steps outlined in Section 23.6. Then, the changes in subsequent releases of their DBMS will prove to be far less traumatic.

How did SQL reach the undesirable state described in this chapter? I believe that the reason can be traced to inadequate theoretical investigation.

Exercises

23.1 At first glance, SQL offers an attractive feature: the capability of nesting a query within a query. What are the problems arising from this feature of SQL?

23.2 Most, but not all, relational DBMS products violate the very basic property of the relational model that a relation, whether base or derived, must not have duplicate rows. Why, and in what ways, does this violation give rise to serious problems?

23.3 Consider the language SQL. It contains a feature called GROUP BY. Treat SQL, with this feature dropped as SQLX. Are there any queries

expressible in SQL that are *not* expressible in SQLX? If your answer is yes, supply two examples. If your answer is no, supply a proof.

23.4 SQL has a nesting feature that permits a query to be nested within another query. Treat SQL with this feature dropped as SQLZ. Are there any queries expressible in SQL that are *not* expressible in SQLZ? If your answer is yes, supply two examples. If your answer is no, supply a proof.

■ *CHAPTER 24* ■

Distributed Database Management

24.1 ■ Requirements

For many reasons, it is necessary to understand distributed database management clearly. Very few database management systems will survive in the 21st century if they are not capable of managing distributed databases with the extensive capabilities described in this chapter and the next.

The first necessity is to be able to distinguish clearly *distributed database management* from *distributed processing*. Some vendors claim support for distributed database management, when their products actually support nothing more than distributed processing. The relational model addresses many of the problems associated with the management of distributed databases, but offers very little help in distributed processing.

A simple, but superficial, way of distinguishing these two services is as follows. Distributed database management is the coordinated management of data distributed in various separate but interconnected computer systems. Distributed processing, which is based on a collection of programs that are distributed in various separate but interconnected computer systems, permits a program at any site to invoke a program at any other site in the network as if it were a locally resident subprogram.

There is no claim that it is a simple task to implement the level of support for distributed database management in RM/V2. Such implementation, although quite a challenge, closely represents what users need. A distributed database satisfies at least the following four conditions:

1. The database consists of data dispersed at two or more sites;

2. the sites are linked by a communications network that may be as modest as a local area network, as impressive as a satellite-based network, or anything in between;
3. at any site X, the users and programs can treat the totality of the data as if it were a *single global database* residing at X;
4. all of the data residing at any site X and participating in the global database can be treated by the users at site X in exactly the same way as if it were a local database isolated from the rest of the network.

Normally, these sites are geographically dispersed, and the communications network includes the telephone lines of at least one telephone company. For convenience, the term *network* will often be applied to the total collection of sites in a distributed database. In a widely dispersed situation, the databases are located in different countries, possibly on different continents. At the other extreme, the database sites may all happen to be located within a single city or even within a single building. It is worthwhile to observe here that more and more companies are becoming trans-nationals (the United Nations term for multi-nationals).

Condition 4 in the preceding list is required because some vendors have DBMS products that support Condition 3 but violate Condition 4. To understand this, suppose that R is one of the relations in the global database, that F is a fragment of R, and that F is allocated to one of the sites, while the remaining parts of R are stored at other sites. In the products that violate Condition 4, but not Condition 3, fragment F is permitted to be a non-relation, even though it is a table. More specifically, F consists of some of the columns of R, but does not include the primary key of R, and, as a result of the corrupted version of **projection** being used, F can contain duplicate rows. The corrupted version of **projection** fails to eliminate duplicate rows. The penalties for permitting duplicate rows were described in Chapter 23.

It is reasonable to consider the chief responsibilities of the systems that manage a distributed database. Perhaps the single most important responsibility of such systems is support for *distribution independence* [Codd 1985]. This term means that a program developed to handle data that is distributed in one way should continue to operate correctly without change when the data is re-distributed. In an extreme case covered by this definition, a program is developed to run successfully on data that is initially located completely at one site (i.e., not distributed at all). Such a program must continue to operate correctly without change when the data is dispersed to multiple sites. This is an important step in protecting the investment by DBMS customers in the development of application programs.

To support distribution independence, a necessary but not sufficient condition is that it must be possible to retrieve data without referring to any location or locations in which it may reside. This is sometimes called *location independence* and sometimes *location transparency*, although I much prefer

the former phrase.This subject is dealt with in more detail in Sections 24.4 through 24.8.

For performance, fault tolerance, or other reasons, some of the data may be duplicated at different sites. If so, end users and programs must be insulated from this redundancy. This is sometimes called *replication independence*. This topic is discussed in more detail in Section 24.4.5.

A third requirement for adequate support of distribution independence is that it must be possible for a *single relational command* to operate on data located at two or more sites. This falls naturally within the scope of relational DBMS, since the optimizer breaks down each command into basic relational operators even if the data is not distributed. Note that this requirement is more stringent than merely supporting any single *transaction* operating upon data located at two or more sites.

RX-1 Multi-site Action from a Single Relational Command

In a distributed database, a single relational command, whether query or manipulative, can operate on data located at two or more sites.

Some of the older pre-relational DBMS products that were claimed by their vendors to support the management of distributed databases could transmit a transaction to a remotely located site, but the whole transaction had to be executable at that site on data located entirely at that site. This capability is called *transaction routing*, and is, of course, far less than what is needed for supporting distribution independence. In fact, the present situation is that the *only* prototypes and products that have been able to demonstrate the capability of supporting distribution independence are relational DBMS.

24.2 ■ The Optimizer in a Distributed DBMS

As we have seen, within a relational DBMS there is a component called the *optimizer*. This component is responsible for translating the high-level relational commands into the most efficient target code. Now, an optimizer may be adequate for managing local databases, but, if that is its only capability, it is likely to be quite inadequate for managing distributed databases.

Nevertheless, it is a simple task to extend a local-only optimizer to handle the distributed case. Whether a database is totally local or distributed, managing it efficiently entails finding an ordering of the basic operators within whatever relational command is to be executed (an ordering that may

include concurrent execution of these operators), together with access paths for each operator, so as to develop efficient target code.

In managing local databases only, the goal is minimal use of processing power, on the one hand, and of input-output devices, on the other hand. In managing distributed databases, the consumption of inter-site communication power is a third factor that must be considered, because it may be quite significant. In fact, when the sites are far apart, this third factor is likely to be dominant.

There is likely to be a wide variation between the requirements of any two companies or institutions using a distributed DBMS. For this reason, the usual approach in DBMS products is to request the DBA at each site to insert three coefficients: one for the local processing load, one for the local input-output load, and one for the inter-site communication load. The optimizer can then calculate a single rating for each combination of a sequencing of operations and access paths.

24.3 ■ A DBMS at Each Site

One of the goals in managing distributed data is to promote *local autonomy* to the extent that is compatible with distribution independence and with the necessary inter-site integrity constraints. When one or more sites are experiencing malfunction of the hardware or software, the other sites must be able to continue to execute all of their workloads, except those portions that involve the sites that are temporarily out of action.

Why is the adjective "necessary" used to qualify the phrase "inter-site integrity constraints" in the preceding paragraph? One reason is that, even if the initial deployment of data across sites does not require such constraints, the deployment will undoubtedly evolve and require a growing number of such constraints as it evolves. A second reason is that the options in redeployment would be severely restricted if all integrity constraints had to affect local data only.

RX-2 Local Autonomy

In distributed database management, whenever the DBMS at any site goes down, each site X that is still functioning must be capable of continuing to operate successfully and in a relational mode on data at each and every site that is still functioning, including X, provided X is still in communication with that site.

One clear consequence of Feature RX-2 is that there is no reliance on a single site in a distributed relational DBMS, whether that site be labeled "central" or not.

One aspect of this feature is that the system must support continued access to all of the data at site A by users at site A, even if the inter-site communications network is either completely or partially out of action. A more precise phrase for this subgoal is *local management of local data.*

To attain local management of local data by means of a relational DBMS, the first requirement is that, at every site, all of the data stored at that site must be perceived by end users and application programmers as a *relational database*—that is, a collection of proper relations of assorted degrees. As indicated in Chapter 1, a proper relation is one that has no duplicate rows. This requirement corresponds to Condition 4 cited earlier.

The second requirement is that there be a *relational database management system* at each and every site. During most of this chapter, the homogeneous case is assumed for simplicity—that is, at each site there is the same hardware and software, including the DBMS software. At the end of this chapter, a few remarks are made about the difficulties encountered in the heterogeneous case. The optimizing capability in a distributed DBMS should itself be distributed. Every site must be capable of controlling every request entered at that site, so that there is no reliance on a central coordinator. Every site has to be capable of (1) global planning and optimization; and (2) local planning and optimization.

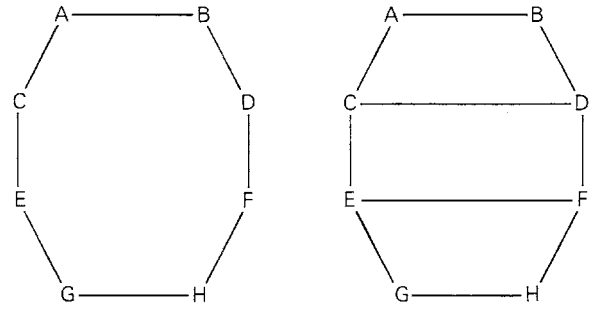
An important factor in organizing the distribution of data over several sites is that the communication channels between sites be sufficiently redundant that the failure of one or two channels does not reduce all sites to managing locally resident data only. When the channel organization avoids this catastrophic behavior, it is called *fault-tolerant.* A channel organization, of course, can exhibit a high degree of fault tolerance, a modest degree, or none at all.

For example, if there were eight sites and the communication channels were unidirectional only, then the channel organization shown in Figure 24.1(a) would not be fault-tolerant at all. On the other hand, if the channels were bidirectional, this organization would be fault tolerant to a small degree. The organization shown in Figure 24.1(b) is fault-tolerant to a much higher degree, assuming that the channels continue to be bidirectional.

24.4 ■ The Relational Approach to Distributing Data

There are several widespread but false notions concerning the initial planning of how to distribute data. One false notion is that a collection of relational databases that already exist in diverse locations can simply be inter-connected by means of communications lines, and that the result is a distributed database. Unfortunately, there is more to this problem than meets the eye. An example of a potentially serious problem that is quite likely to arise is the case in which two columns, one from one relation, and one from a second, are homographs having a different meaning. See Section 6.2 for more detail.

Figure 24.1 **(a) Simple Network with Eight Sites. (b) More Stable Network with Eight Sites**



Another false notion is that, once the initial plan is put in place, redistribution of the data will never be necessary.

When deciding how to distribute data between various sites, it is convenient to think of the totality of data initially—and, as a matter of fact, at all later times also—as a single collection Z of relations. This totality is called the *global database*. It is the importance of this concept that strongly suggests employing a global database administrator (GDBA), who is responsible for the global database, as well as a local database administrator (LDBA) for each site.

Since the global database is really a virtual database (an abstract concept), it could be said that the GDBA has a "virtual job," but I doubt whether he or she would appreciate such a comment because of the multiple meanings attached to the English word "virtual." Nevertheless, this job might well be one of the most important positions in any company or institution. The title "chief information officer" seems quite appropriate.

In any distributed relational network, the GDBA needs as a tool for his or her job a description of the entire global database. Such a description is contained in what I call the *global catalog*. This catalog is, of course, distinct from the on-line catalogs at each site, and more comprehensive, since the local on-line catalogs tend to concentrate on local data only. One of the reasons for having a global catalog is that it is a vital tool in conceiving and declaring integrity constraints, views, and authorization that straddle sites.

RX-3 Global Database and Global Catalog

Associated with each distributed database is the concept of a *global database* that covers all the data stored at each site. Associated with

the global database is the *global catalog*. This catalog contains three parts:

GC1 The declarations for every domain, every base relation, every column, every view, every integrity constraint, every authorization constraint, and every function in the global database *as it is actually distributed,* including the sites at which they are stored.

GC2 A concise description of this totality of information as if it were a single, non-distributed database cast in fifth normal form with minimal partitioning.

GC3 Expressions for each relation located at each site defining how that relation is defined in terms of the relations declared in GC2.

It is more useful to call GC1 the *composite global catalog* and GC2 the *normalized global catalog.*

The source code of application programs can contain local names, but these are converted by the DBMS at bind time into global names as found in GC2. It is this *globalized source code* that is retained in the system and remains unaffected by redeployment of the data, partly because it contains no local names.

RX-4 *N* Copies of Global Catalog ($N > 1$)

The network contains N copies ($N > 1$) of the global catalog, in the form of N small databases at N distinct sites to avoid too much reliance on whichever site is normally used by the global database administrator. These N sites can also be N of the sites in the network for regular data and the corresponding local catalogs. At these sites (at least two of them), the power supply should be mutually independent; that is, failure of electric power at one site does not normally mean failure at the other.

In a distributed relational database and, in particular, in its global database Z, it is necessary to have names for domains, relations, columns, views, integrity constraints, and functions that (1) conform to the standard naming rules for relational databases (see Chapter 6), and (2) apply to the totality of data in the network as if it were a single database. These names are called *global names* to distinguish them from any names in use locally at each of the sites. Names that are continued in local use are called *local names.*

RX-5 Synonym Relation in Each Local Catalog

To support the continued use at each site of names local to that site, each local catalog should include a synonym relation containing each local name, the type of object named by it, and the correspondence between the local name and the global database (see RX-7).

The job of interpreting global names used in relational commands is not the sole responsibility of the N global catalogs, because they would then become major bottlenecks. That job is dispersed throughout the network by means of the birth-site concept, which is discussed here and also in Section 24.8. This concept is due to Bruce Lindsay [1981] of the System R* team of IBM Research [Williams et al. 1981]. One of the advantages of the birth-site concept is that it does not require a centralized dispenser of names that are unique within the network. However, this concept is not a complete solution to the problem of decentralizing all resolution of names. Frequently, redeployment is not confined to simple movement of entire relations from one site to another. Redeployment can include decomposing some relations and recombining others. Then, the name of a relation on which a program operates may have to be transformed into a relation-valued expression.

Each DBMS in the network has the responsibility of keeping its portion of the global catalog consistent with the global catalog itself. The many DBA sites, each of which contains a copy of the global catalog, must send a message to whatever sites are affected whenever the GDBA makes a change in the global catalog. These messages are requests to the sites to bring their catalogs up-to-date and consistent with the global catalog. The sending of these messages can be, and should be, automatically triggered by the DBMS at the GDBA site.

Ideally, of course, any changes made by the GDBA in the global catalog should be wholeheartedly supported by the local database administrators. Similarly, any changes made by a LDBA should be wholeheartedly approved by the GDBA.

Therefore, the DBMS must support changes in a database at a specific site that are initiated by the DBA at that site, assuming he or she has been granted that privilege. Such changes cause the local DBMS to send a message immediately to the global catalog to make it consistent with these changes, along with a confirming message to the GDBA.

24.4.1 Naming Rules

Although there is every reason to believe that the naming rules introduced here actually work and would satisfy most users' needs, a distributed database management system can support alternative features, provided it can

be shown that these features are at least as powerful, flexible, and comprehensible. The first and most obvious feature follows.

RX-6 Unique Names for Sites

Whenever an object is created—whether it be a domain, base relation, view, integrity constraint, or function—it is asigned a five-part name:

1. the name or identification of the user creating that object;
2. the name of the object as if it was a local object;
3. the name of the site at which the object was created (called its *birth site*);
4. the formula or command that translates from the global database to the local object;
5. the inverse formula or command (see RX-8).

The next feature is explained partly below and partly in Section 24.8, which deals with the distributed catalog.

RX-7 Naming Objects in a Distributed Database

Whenever an object is created—whether it be a domain, base relation, view, or function—it is assigned a three-part name: (1) the name or identification of the user creating that object, (2) the name of the object as if it were a local object, and (3) the name of the site at which the object was created, called the object's *birth site*.

If the object is a domain, considerable care should be taken to determine whether that domain already exists somewhere in the network. Even if exactly the same domain does not exist elsewhere, it may be possible, and is probably preferable, to expand the range of some existing domain to cover the one now needed, and introduce some column constraints that will keep the old definition applicable to those columns already existing.

Of course, an object created at a site X may at some later time be moved to site Y. Later still, it may be moved again to site Z. Therefore, there is one more name associated with each object—namely, the name of the site at which that object is residing at present. The association of this sixth name component with the first five is maintained by the DBMS at the birth site, and by the N copies of the global catalog at the GDBA sites. The goal is to protect users, programmers, and application programs from having to know this sixth name component.

In establishing a distributed database, the starting point may be several databases that have been operated independently up to that time. Then, the first step is to take a copy of the description of each database from its catalog, and examine the consequences of putting all these descriptions together as the description of one global database. This examination should include the complete naming scheme, about which the following questions should be asked:

- Is it devoid of the kind of duplication of names that would give rise to ambiguity? (For example, is the name CAP at one site given to a relation that represents the capabilities of suppliers in supplying parts, while at another site that same name is used for a relation with an entirely different meaning—say, for actual shipments made by suppliers?)
- Are there any instances of an object, such as a domain or a relation, common to two or more sites, and having two or more distinct names? (For example, is a relation that describes shipments at one site given a different name at another site, even though the two relations have the same meaning and are union-compatible?)

The DBMS provides considerable help in resolving these naming problems by supporting in the catalog at each site (as noted earlier) a synonym relation that contains global names or relation-valued expressions as synonyms for all the local names. This facility must be applicable to at least the names of domains, base relations, columns, views, integrity constraints, and functions.

After naming problems are resolved, the following question arises: are the databases that are being coalesced into a single global database interrelated? This problem was discussed in Section 3.1, but not in the context of distributed database. However, the approach was based on the use of domains by relations, and this is clearly applicable without change to the global database Z. If any one of the databases being coalesced is not interrelated by domains to any of the other databases, one may reasonably question the reason for coalescing—perhaps there is a plan to add new relations to one or more sites in order to inter-relate the databases at these sites, where previously they were not.

The information in the global database Z is to be distributed in some way to a collection of sites, with a collection of relations at each site. Thus, at first glance, it appears necessary to consider as candidate relations all of the relations derivable from the given collection using the relational operators.

The class of all derivable relations, however, is quite large; for a large commercial database, it may run into the millions. Thus, it is necessary to find some sensible means of reducing the options. Fortunately, there are two important concerns that quickly narrow down the options to be considered.

Perhaps the most obvious concern is that data should be distributed according to the frequency of its use. To express it another way, data should be local with respect to the users who need to access it most frequently and with the shortest response time.

For example, consider a bank that has branches in several cities, and customers who execute most of their transactions with the bank at a particular branch selected by each customer. When establishing a distributed DBMS, the usual decision by the bank is to require the data stored in each major city to be precisely that which reflects the customer accounts established in the branches of the bank in that city.

A second concern is *reversibility* of the transformations applied to the global database Z (the totality of data at all sites) in determining which of these relations (or, alternatively, which of the relations derivable from them by means of the relational operators only) are stored at which sites. This reversibility is in many ways similar to that used in deciding the ways in which a view can be updated (see Chapter 17). As shown in the discussion of decomposition in Section 24.4.3, this second requirement can also narrow down the options significantly.

As far as users are concerned, reversibility is needed primarily to facilitate re-distribution of data at some later time. DBMS vendors are advised to make this a requirement in distributing data because it simplifies the design of the optimizer. Even then, however, there is no claim that designing the optimizer is easy.

RX-8 Reversibility and Redistribution

In assigning part of the global database Z to a particular site, each relation assigned to that site must be derivable from the relations in Z by a combination of relational operators that has an inverse. In other words, each relation at each site is reversibly derivable from Z.

It is quite common for managers of data processing and information system to claim that, if the job is done correctly when first introducing a distributed DBMS, there will be no need to redistribute the data. This can be true only if the business or institution remains unchanged forever! However, there is one thing in life that is inevitable for everyone, and that is change.

24.4.2 Assignment of Relations from the Global Database Z

The most elementary distribution is to assign some of the relations in Z without any transformation to site A, to assign other relations in Z without any transformation to site B, and so on until all the relations have been assigned to sites. In some cases of database distribution, this simple approach may be adequate, but in other cases it will not be adequate. In the next two sections, other options in distributing data are considered.

24.4.3 Decomposition of Relations from the Global Database Z

Notice that, in the simple distribution just discussed, no relation in Z is decomposed into two or more pieces, and the pieces then scattered to two or more distinct sites. It is this step of decomposition that now must be discussed.

When a relation is decomposed into pieces for distribution of the pieces to various sites, the term *fragmentation* is often applied. I shall not use this term, however, because it is often used and interpreted as the breaking up of a table by rows and columns, without concern as to whether the objects resulting from this break-up are proper or corrupted relations. At least two distinct distributed DBMS prototypes—IBM's R* and Relational Technology's INGRES STAR—and one distributed DBMS product—Tandem's NonStop SQL—support vertical fragmentation of any kind, without concern regarding whether each fragment is a proper relation or not. To paraphrase Woody Allen, this bug is as big as a Buick!

In applying relational technology to the management of a distributed database, it is essential that only the relational operators be used to decompose relations in the global database into relations for the various sites. One important reason is the goal of preserving the correctness of programs when the data is re-distributed.

Consider the example of a relation R with columns A1, A2, A3, A4. Suppose that the primary key of R is A1. The fragmentation approach would permit R to be split into two tables S and T, where S has the columns A1 and A2 of R, and T has the columns A3 and A4 of R. It is assumed here that, from the standpoint of the values found in R, the columns are preserved intact. Notice that the primary key of R (i.e., A1) is now a column in S, and that this very column is now the primary key of S. Therefore, the rows of S are all distinct and S is a proper relation. The same cannot be said of T. Because the primary key of R has *not* been preserved in T, there is no guarantee that the rows of T are all distinct. Thus, T may be a corrupted relation. The difficulties stemming from this kind of table are described in Chapter 23.

What kind of column-oriented decomposition is considered correct according to the relational model? The relational operator to apply is **project.** Each projection to be stored at some site, possibly remote, must include the primary key of the relation from which the projection is made. Then, each and every projection is a proper relation with no duplicate rows, and it will be managed at the pertinent site by the DBMS at that site. Thus, the relation R just examined could be split into two or three projections, each of which includes column A1. The three-fold split would be as follows:

S (A1 A2)	T (A1 A3)	U (A1 A4)
at site B	at site C	at site D

Note that, at any time, the original relation R can be recovered from relations S, T, and U by using **equi-join** with respect to the primary keys of the relations being joined. For each relation in this example, the primary key is A1. This recovery capability permits the relation R to be re-distributed in an entirely different way at some later time, without causing any application program to be damaged logically.

There are bound to be complaints about the duplication of keys across sites that stems from the preservation of the primary key at each site that receives a projection. This duplication is a small price to pay for the power from retaining keys (e.g., non-traumatic re-distribution of the data when needed). The power is especially cheap because primary keys are seldom updated.

RX-9 Decomposition by Columns for Distributing Data

When some columns of a relation R in the global database Z are assigned to one site, and some to one or more other sites, the pertinent operator is **project** (as defined in Section 4.2), and the primary key of R *must be included* in each and every projection.

An alternative or additional decomposition is by rows. Here again, however, the selection of rows cannot be arbitrary—such as, store the first 50 at site B, the next 50 at site C, and so on. Relational DBMS are intentionally not aware of the meaning of "the first 50" and "the next 50." These concepts are leftovers from the days of file-management systems and early pre-relational DBMS products, in which the user's perception of the data was very close to the way the data happened to be stored.

A selection of rows must be made using the **select** operator. For example, one may store at site B those rows of relation R whose numeric values in column A3 range from 557722 to 999999. Another, quite distinct, example is the selection of those rows of R whose alphanumeric values in column A4 range from HHH12JJ to NNN11KK as a relation to be stored at site C. It may be worth noting that alphanumeric ordering must be based on either a declared or a standard collating sequence such as ASCII or EBCDIC.

When a relation R is partitioned by rows into several subrelations, each to be stored at distinct sites, it is possible to recover the original relation R by means of the operator **union.** The **union** ALL of SQL is not usable in this context because it is a non-relational operator, one that generates a result that may contain duplicate rows.

If, as is usually the case and as is recommended, relation R is partitioned into several *disjoint* subrelations (i.e., no pair of subrelations has any rows in common), then it will not be necessary to update the mutually redundant

rows at two or more sites (because there will be no such rows) whenever an update is requested on one row.

RX-10 Decomposition by Rows for Distributing Data

When some rows of a relation R in the global database Z are assigned to one site, and other rows to other sites, the pertinent operator is **select** (see Section 4.2), using ranges of values applied to a simple or composite column of R. The ranges should not overlap, which means that no row is assigned to more than one site.

Why is the capability of recovering the original relation important? This recovery is needed if at some later time a decision is made to re-distribute the data in a different way, possibly using different decompositions and/or different combinations. For example, the initial distribution of the sample relation might be by rows based on values in column A3. Later a re-distribution might be necessary; it might be by rows, but based on values in column A4. Still later, it might be by columns instead of by rows.

24.4.4 Combination of Relations from the Global Database Z

While the **project** and **select** operators can be used to split a single relation from Z into several relations, the **join, union, relational intersection, relational difference,** and **relational divide** operators can be used to combine several relations from Z into one relation. Clearly, if the transformations performed on relations from Z are to be reversible, these operators must be applied with considerable care. For example, just as every **projection** of relation R must include the primary key of R, so every **join** should involve the following in the comparand columns:

- the primary key on domain D (say) of one relation; **and**
- either a primary key on domain D of the other relation, or a foreign key on domain D of the other relation.

RX-11 General Transformation for Distributing Data

Any combination of relational operators (whether of the decomposing or the combining type) is applicable to determining how the data should be distributed, provided that the total transformation is reversible.

24.4.5 Replicas and Snapshots

Sometimes, two or more sites need frequent access to common information in certain relations in the global database, and the GDBA may be tempted to assign copies of these relations to these sites. The type of copy that is kept up-to-date by the distributed DBMS whenever any modification (insertion, update, or deletion) is executed on one of the copies is called a *replica*. The type of copy that is not kept in such a high state of accuracy, but is merely refreshed from time to time (either by a specific DBA request or by a DBA request to refresh at specified time intervals), is called a *snapshot*. An example of a DBA command for this purpose is

```
CREATE SNAPSHOT R REFRESH EVERY 7 DAYS
```

where R denotes an expression that evaluates to a relation (including the simple special case that it is the name of a base relation).

The GDBA should decide whether a copy is to be held as a replica or a snapshot. This decision should be made with great care because replicas are significantly more expensive than snapshots in terms of performance. If only two sites are involved in the decision, and one is oriented toward production and the other is oriented toward planning, the choice is clear: accurate on-line data for the production site and snapshots for the planning site; no replicas are needed. Then, the GDBA must decide how frequently these snapshots must be refreshed.

_RX-12_ Replicas and Snapshots

The DBMS must support all declared replicas by dynamically maintaining them in an up-to-date state. End users and application programs can operate independently of whether these replicas exist and how many there are. The DBMS also supports snapshots that are updated to conform to the distributed database with a frequency declared by the DBA.

Why are replicas significantly more expensive in performance than snapshots? The root of the performance problem with replicas is that application programs must be protected from the burden of explicitly modifying all replicas of a given relation. This protection is necessary in order to achieve replication independence. In other words, these application programs must continue to be logically correct when a new replica of an old relation is introduced or when an old replica is discontinued. The programs must therefore modify just one copy, and the DBMS must assume responsibility for modifying the remaining copies of the pertinent relation in exactly the same way.

24.5 ■ Distributed Integrity Constraints

One more reason to perceive the totality of distributed data as a single global database is to establish all the integrity constraints that happen to be appropriate. Since certain types of integrity constraints are subject to change over the years (especially those that are DBA-defined), there must be a GDBA—a database administrator who maintains the global perspective and is responsible for the global database.

Some of the integrity constraints in a distributed database can be enforced completely at just one site, but many will straddle the data at multiple sites. These straddlers require inter-site cooperation of two kinds:

1. between DBAs to establish appropriate declarations;
2. between DBMS at the various sites for their enforcement.

RX-13 Integrity Constraints that Straddle Two or More Sites

The DBMS must support both referential integrity and user-defined integrity constraints when they happen to straddle two or more sites. Inter-site cooperation for enforcement must not involve any special action by users, but rather must be built into the DBMS at each site. The support for integrity constraints that straddle sites must protect users from having to be aware of the straddling in any way, even after one or more redeployments of the data.

24.6 ■ Distributed Views

One more reason to perceive the totality of distributed data as a single global database is to establish all of the views needed by application programmers and terminal users. Views are still defined in terms of the base relations and other views, but some of these relations may be located at distinct sites. In addition, the actual views that are needed may change over the years. Thus, once again a database administrator (the GDBA) who maintains the global perspective, and who is responsible for the global database, is needed.

All view definitions applicable to the entire database are stored in each of the global databases for use by the DBA. However, this collection of definitions is not used by the DBMS at individual sites to handle each relational request successfully.

To avoid the global databases becoming a traffic bottleneck, view definitions are scattered around the sites with whatever degree of duplication is necessary to provide good performance. One possibility is to store at each

site X only those views that refer to one or more relations stored at X. Such definitions can, of course, refer to relations at sites other than site X also.

Full support of views that straddle the data at multiple sites requires inter-site cooperation of two kinds:

1. between people to establish appropriate declarations;
2. between DBMS at the various sites.

The second kind of inter-site cooperation is required due to the fact that *it is not normally true* that the view definitions for the entire distributed database are stored in each and every catalog.

RX-14 Views that Straddle Two or More Sites

The DBMS must support views when they happen to straddle two or more sites. Inter-site cooperation for this support must not involve any special action by users. It must be built into the DBMS at each site. The support for views that straddle sites must protect users from having to be aware of the straddling in any way, even after one or more redeployments of the data.

Support of inter-site views is essential to full support of distribution independence.

24.7 ■ Distributed Authorization

One more reason to perceive the totality of distributed data as a single global database is to establish all of the authorization needed by application programmers and terminal users. Authorization is still defined in terms of the base relations and views, but some of these relations may be located at different sites. In addition, the authorization that is needed may change over the years. Thus, once again there needs to be a database administrator (the GDBA) who maintains the global perspective, and who is responsible for the global database.

All declarations of authorization for the entire database are stored in each of the global databases for use by the global DBA. However, this collection of definitions is not used by the DBMS at individual sites to handle each relational request successfully.

To avoid the global databases becoming a traffic bottleneck, declarations of authorization are scattered around the sites with whatever degree of duplication is necessary to provide good performance. One possibility is to store at each site X only those declarations that refer to one or more relations stored at site X.

Full support of declarations of authorization that straddle the data at multiple sites requires inter-site cooperation of two kinds:

1. between people to establish appropriate declarations;
2. between DBMS at the various sites.

The second kind of inter-site cooperation is required due to the fact that *it is not true that* every declaration of authorization for the entire distributed database is stored in each and every catalog.

RX-15 Authorization that Straddles Two or More Sites

The DBMS must support declarations of authorization when they happen to straddle two or more sites. Inter-site cooperation for this support must not involve any special action by users. It must be built into the DBMS at each site. The support for authorization constraints that straddle sites must protect users from having to be aware of the straddling in any way, even after one or more redeployments of the data.

Support of authorization that straddles two or more sites is essential to full support of distribution independence.

24.8 ■ The Distributed Catalog

Every site has its own relational DBMS. Therefore, every site has its own catalog. What does this catalog contain in the case of a distributed database?

At each site, there must be a catalog that includes at least the description of all of the data stored at that site. If nothing more is done, then, when any DBMS at one site attempts to find data located at other sites, this may entail searching the catalog at each and every site. This task is unnecessarily burdensome for both the particular DBMS, which does the searching, and for the network, which must support a heavy amount of inter-site communication.

One solution that is unacceptable is to store the complete description of the global database Z at just one central site, and use this description to determine which sites are involved for every reference to data in the network. If this solution were adopted and the central site goes down, then the entire distributed database network would become inoperable. Further, the catalog at this central site would be a continual bottleneck in all database activities. Thus, there can be no dynamic dependence upon a single site that happens to have a catalog that describes the global database Z and indicates at which site each part of Z is stored.

At the other extreme, however, the catalog at each and every site would contain the description of the entire global database Z. In this case, there is a very high degree of redundancy between the catalogs. Any change in the database at just one site would involve making changes in the catalog at each and every site. These changes would have to be coordinated so that all the catalogs remained at all times consistent with one another and with respect to each transaction. This consistency would probably entail locking up all catalogs until any requested catalog change had been received by each and every site, and completed by each and every DBMS. This, in turn, means that virtually all the other traffic on the network would have to be stopped for however long would be required to get these catalogs into synchronism with the global catalog.

A compromise between the two extremes was invented by the R* team, then located at the IBM research laboratory in San Jose (most team members are now located at the IBM Almaden Research Center). Suppose that a relation is created at site X. Then, site X is called its *birth site*. From the user's perspective, the name of the relation has only two parts: the local name at the time of creation, and the name of the birth site. Although the local name alone is temporarily adequate for any interactive user who happens to be at the site where the relation is stored, one fact of life must be faced: the pertinent relation might at some future time be moved from its birth site to some other site; then, the local name is inadequate.

In contrast, there is a significant advantage to combining the local name with the name of the birth site to form the global name; this combined name permanently identifies the relation uniquely, no matter at what site the relation happens to be stored. Consequently, to develop application programs that need not be changed whenever a relation is moved from one site to another, it is necessary to use the global name for each relation.

RX-16 Name Resolution with a Distributed Catalog

Each base relation stored at each site has a global name that is a combination of its local name at that site, together with the name of its birth site. The catalog at its birth site includes the name of the site at which it may now be found.

An inexpensive tool for gaining improved performance is a *global-name cache* at each site. This cache can be conceived as an extension of the synonym relation described in Section 24.4. For items in the global database that are not stored at site X, but are frequently accessed from site X, the synonym relation at site X is extended to include the global name of each such item, along with the identification of the site where the corresponding object is now located.

24.8.1 Inter-site Move of a Relation

When a relation is moved from one site to another, the user making such a request must be appropriately authorized. Such a user is likely to be the global DBA or someone on his or her staff. Four sites are involved in such a move: the FROM site, the TO site, the BIRTH site, and the GDBA site. Of course, either the FROM site or the TO site (but not both) may be identical to the BIRTH site, and the GDBA site may be identical with any one of the other three sites.

Thus, when an inter-site move of a relation is requested, four sites are normally involved, each of which makes changes to its catalog contents. These changes reflect not only the fact that one or more relations have been moved, but also the changes in local and inter-site integrity constraints that result from that move. Normally, in the case of the global catalog, there are no resulting changes in integrity constraints.

The DBMS at the birth site of the pertinent relation (say R) is responsible for ensuring that its catalog contains identification of the present site of R along with its birth site. Thus, using the global name of the relation R enables any DBMS in the network that happens to be the source of a request on R to ask of its birth site where that relation is now located. The DBMS at any site can find every relation in the network by querying the catalog at exactly one or two sites: one site only if the relation happens to have remained at its birth site; otherwise, two sites.

RX-17 Inter-site Move of a Relation

A user who moves a relation from one site to another must be authorized to do so, or else the authorization mechanism will not permit the move to be executed. In the MOVE command, the user must specify the global name. By this means the DBMS at the birth site can and does update its catalog to record the new site for this relation.

24.8.2 Inter-site Move of One or More Rows of a Relation

As indicated in Section 24.8.1, a relation in the global database can be dispersed to several sites by rows using the range of values in a simple or composite column. As an example, consider a relation R that has a column drawing its values from the currency domain. Suppose that the rows of R are distributed by assigning to sites A, B, C those rows that have values in this currency column within the following ranges:

Site A:	0 to 999
Site B:	1,000 to 9,999
Site C:	10,000 to 99,999

Suppose that a suitably authorized user at some site requests an update on a row of R that happens to be located at site A, because it happens to have a currency value in the range 0 to 999. Suppose also that this update is an increment that takes the currency value into the range for site B (i.e., 1,000 to 9,999). Then, this row is automatically moved from site A to site B, and the initiative for this action is taken by the DBMS at site A.

RX-18 Inter-site Moves of Rows of a Relation

Suppose that rows of a relation R are distributed according to the range of values in some simple or composite column of R, and that an update is applied to this column of a row at site X1. Suppose also that this update takes the value out of the range of values pertaining to site X1, and into the range pertaining to site X2. Then, if an indicator in the catalog signals that moving a row can be triggered by such an update, the DBMS automatically moves the row from site X1 to site X2. A single updating relational command may result in zero, one, two or more inter-site moves of rows of a relation.

In one sense, inter-site moves of rows are simpler for the DBMS to handle than inter-site moves of relations. No changes are necessary in any catalog. Authorization for inter-site moves triggered by updates can be handled by means of the *N*-person turn-key Feature RA-5 (see the remarks following Feature RA-6 in Chapter 18).

24.8.3 More Complicated Re-distribution

Now, it is reasonable to ask, "What if a re-distribution of data is more complicated than the move of an entire relation from one site to another? What if a relation in the global database Z is decomposed or combined with some other relation in a new way—by using different operators, by using different columns as comparand columns, or by both means?"

Clearly, the one object that remains constant in such a change is the global database Z. Therefore, to protect application programs from damage under these circumstances, they should be developed to operate upon data in the global database or upon views that are defined on that database, making use of the information in the global catalog. The burden of using global names can be assumed to a large degree by the DBMS if it supports globalization of the source code as defined shortly after Feature RX-3. Local names can be used by end users when interacting at terminals with a local database. In all other cases, however, local names should not be used.

24.8.4 Dropping Relations and Creating New Relations

What if the global database changes because some kinds of data in the network have been dropped entirely? This type of change is similar to the dropping of information from a non-distributed database. Those application programs that make use of the dropped information are very likely to become inoperable unless some changes in them are made. The other application programs will remain unaffected. The global catalog must be contracted so that it no longer includes any mention of the kinds of items that have been removed. Those local catalogs that are affected by the drop must also be changed to be consistent with the global catalog.

RX-19 Dropping a Relation from a Site

When a suitably authorized user drops a relation R stored at site X, the following catalogs are adjusted to reflect the loss of information from the entire distributed database: (1) the local catalog at site X, (2) all copies of the global catalog, and (3) the birth-site catalog. The only application programs adversely affected are those that use the data in R.

Now, suppose that instead of dropping certain kinds of data, new kinds of data are added to one or more sites in the network. The description of the global database must be enlarged to cover the new domains, new relations, new columns, new views, new integrity constraints, new authorization constraints, and any new functions. All the application programs developed before these new additions should be capable of operating correctly without any changes whatsoever. Once again, those local catalogs that are affected by the new items introduced must also be changed to be consistent with the global catalog.

RX-20 Creating a New Relation

When a suitably authorized user creates a new relation R at site X, the following catalogs are adjusted to reflect the new information in the network: (1) the local catalog (X is declared to be the birth site), and (2) all copies of the global catalog.

24.9 ■ Abandoning an Old Site

Occasionally, because of business or institutional conditions, one of the sites managed by a distributed database system must be either abandoned, or

detached from the network. Two cases must be considered. In the simpler case, all of the data at that site is to be abandoned or detached also. In the more complicated one, some or all of the data at that site is to be retained in the network, but moved to other sites in that network.

24.9.1 Abandoning the Data as Well as an Old Site

Suppose that site X is being abandoned, along with all the data stored there now. In this case, the GDBA must remove from the N copies of the global catalog all references to the relations at site X, except references to X as the birth site of any relation that happens to be located now at some site other than X. These references must remain viable if application programs are to remain as logically correct as possible.

A site other than X—one that is surviving in the network—must be chosen by the GDBA to act as if it were the birth site X. This places a responsibility on the chosen site to behave as if it were (1) the birth site of relations created at that site, and (2) the birth site of relations actually created at site X. Each DBMS at each site in the network must have the capability of assuming this kind of additional responsibility.

24.9.2 Retaining the Data at Surviving Sites

Together with the local database administrators, the GDBA must determine how the data now stored at site X (the site being abandoned) is to be re-distributed. In the simplest case of re-distribution, each relation currently stored at X is moved in its entirety to some new site. Several sites may be involved as recipients. The more complicated case of re-distribution involves the following:

- decomposition of relations presently stored at X; or
- combination of these relations with others in the network; **or**
- both decomposition and combination.

This task is similar to that of establishing the distributed database in the first place. When this is done, the GDBA must select one of the surviving sites to act as the virtual birth site X for all of the relations created at X that are still in the network. (This process was discussed in more detail in the previous section.)

RX-21 Abandoning an Old Site and Perhaps Its Data

A suitably authorized user (usually the global database administrator) may detach a site X completely from the network, and also

abandon the data stored at that site. Some other site must be designated to carry on the duties of X in keeping track of the present whereabouts of relations created at X (birth site = X), but already moved from X to some other sites. If some or all of the data is to be retained at other sites, then the GDBA should be involved, and the *N* copies of the global catalog and the local catalogs at receiving sites must be adjusted to reflect the re-distribution of the data formerly at site X.

In both the cases discussed in Section 24.9 and this section, application programs remain logically correct, except if they refer to data that has been abandoned or detached altogether from the network.

24.10 ■ Introducing a New Site

It must be possible to introduce a new site without adversely affecting the logical correctness of any of the application programs. The tasks for the GDBA and the local DBA for the newly introduced site are similar to those involved in establishing the distributed database in the first place.

Suppose that the new site is site X. If there is no DBMS at site X, one must be selected and installed. Care must be taken in selecting it to ensure that it is compatible with the rest of the network. Today's advertisements by DBMS vendors contain many false claims concerning compatibility.

If there is no database at site X, it will be necessary to create some domains and relations there; data may be re-distributed from other sites. As these domains and relations are created, the local catalog at site X is kept up-to-date by the DBMS at that site. Changes will also be necessary in all *N* copies of the global catalog and these changes are made by DBMS.

RX-22 Introducing a New Site

A suitably authorized user, usually the GDBA, can attach a new site X, to the network. The GDBA must decide whether data that is already in the network is now to be moved to X, or what new data is to be stored at X. The *N* copies of the global catalog, together with the local catalog at site X and possibly the local catalogs at other sites also, must be adjusted to reflect these decisions. Introduction of a new site does not adversely affect any existing application programs.

Occasionally, local DBAs and the global DBA may have to make a combination of changes in their catalogs. This is facilitated by the following feature.

RX-23 Deactivating and Reactivating Items in the Catalog

A suitably authorized user can deactivate any selected items in the catalog under his or her control. Later, and possibly within the same CAT block, this user can reactivate the deactivated items.

For details concerning the CAT block, see Feature RM-7 in Chapter 12.

Exercises

24.1 What is the optimizer supposed to do in a non-distributed DBMS? State three stages. What additional action is expected if the DBMS is claimed to manage distributed databases? (See Chapter 25.)

24.2 A distributed database has been in operation for a while. Assume that the data is distributed according to RM/V2. Because of changes in the business, it is now necessary to discontinue some sites, introduce others, and generally redistribute the data in a non-loss way. Will it be necessary to change the application programs in order to ensure that these programs operate correctly on the redistributed data? Provide reasons for your answer.

24.3 Assume you have a distributed DBMS that supports replication independence. In determining how data should be distributed to various sites, what is the problem in deciding whether some data should be replicated at two or more sites? What are the alternatives to replicas?

24.4 In distributed database management, suppose that you are determining how data should be deployed to the various sites, and that you have decided to use decomposition by rows based on specified ranges of values in some column. Why is it desirable to ensure that the range of values for any one site does not overlap the range of values applicable to any other site?

24.5 Consider the reversibility condition applied by RM/V2 to whatever transformations are used on the global database in planning the deployment of data in a distributed database. Consider also the reversibility condition applied in defining views that are intended to be updatable. Are these two conditions identical? If not, what are the differences and why? All parts of your answer should be precise. (See Chapter 17.)

24.6 A single relational command happens to refer to data located in several sites. Assume that it has not been compiled yet. Supply a list of three inter-site activities that the DBMS must support if the pre-execution stage is to be completed correctly. Make no special assumptions about the command.

24.7 Supply one or more reasons why is it important that the DBMS retain the source code for every relational request in distributed database management. These reasons must not include those applicable to non-distributed data.

■ *CHAPTER 25* ■

More on Distributed Database Management

25.1 ■ Optimization in Distributed Database Management

To provide high performance on a variety of query and manipulative commands, the optimizer is an important component, even in a non-distributed version of a relational database management system. The optimizer is even more important in a distributed version. In fact, I would describe it as a *sine qua non.*

It is not hard for a user to devise tests that show how good an optimizer is in any given relational DBMS product. The basic idea is as follows. A query is created that makes use of several of the basic relational operators: for example, one or two **joins,** a **union,** a **selection,** and one or two **projections.** Using the relational language supported by the product, the query is cast in at least two distinct forms, which can be expected to give markedly different performance if the optimizer is not doing its job.

These forms differ primarily in the ordering of terms within them. If the DBMS yields quite different performance on these two forms, then it is not executing the first important step in optimization—namely, converting each query into a single canonical form so that the performance attained does not depend on how the user expressed the request in the relational language.

A question that arises is: Why is the optimizer so important in handling distributed databases? The following example should answer this question. It demonstrates the radical difference in performance that can be achieved depending on the quality of the optimizer in a distributed database management system.

25.1.1 A Financial Company Example

Consider a sample distributed database that includes relations concerning customers, investments that are offered by the company, and investments held by the customers:

CUSTOMER C (stored at site X)

- C# Customer serial number
- CN Customer name
- CC Customer city
- CS Customer state

CUSTOMER-INVESTMENT CI (stored at site Y)

- C# Customer serial number
- I# Investment serial number
- CV Value in U.S. dollars

INVESTMENT I (stored at site Y)

- I# Investment serial number
- IT Investment type
- IU Unit of investment

Suppose that the following assumptions hold:

- 100,000 customers in relation C.
- 2,000 distinct types of investments in relation I.
- 200,000 customer-investments in relation CI.
- 2,000 bits per row (in any relation).
- 10,000 bits per second over the communications links, a rate equivalent to five rows per second.
- 1-second delay in gaining access to the communications links.
- 10 type-G investments.
- 4,000 customers in Illinois.

Consider a sample query, to be expressed in SQL: find the identifiers and names of customers in the state of Illinois who have investments of type G. One way of expressing this in SQL is as follows:

```
SELECT  C#, CN
FROM    C, CI, I
WHERE   C.CS = 'Illinois'
AND     C.C# = CI.C#
AND     CI.I# = I.I#
AND     IT = 'G'
```

There are at least six alternative methods that can be used in executing this SQL command; only these six are presented. In what follows, only the load on the inter-site communications lines is computed for each case, since this is likely to be the dominant factor. An adequate optimizer would, of course, go further and calculate processing loads at each site and input-output loads at each site.

Method 1: Move C to site Y
100K rows @ 5 per sec = 20K seconds
= 5 hours 33 mins 20 secs

Method 2: Move CI and I to site X
202K rows @ 5 per sec = 11 hours
→ GREATEST COMMUNICATION TIME

Method 3: Execute (CI * I)[IT = 'G'] at site Y,
where * denotes **natural join** using I# as the comparand
For each row, check site X to see if customer state = 'Illinois'
2 messages for each type-G investment held
= 200 messages at 1 sec delay and 0.2 sec transmission
= 240 secs = 4 minutes

Method 4: Execute C[CS = 'Illinois'] at site X
For each row, check site Y to see if customer has investment of type G
2 messages for each Illinois customer
= 8K messages at 1 sec delay,
and 0.2 sec transmission
= 9600 secs = 160 minutes
= 2 hours 40 mins

Method 5: Execute (CI * I)[CT = 'G'][C#, I#] at site Y,
where * denotes **natural join** using I# as the comparand
Move result to site X to complete the query
1 row for each type G investment held
= 100 × 0.2 secs
(ignoring single delay of 1 sec)
= 20 secs → LEAST COMMUNICATION TIME

Method 6: Execute C [CS = 'Illinois'] at site X
Move result to site Y
1 row for each Illinois customer
= 4000 × 0.2 secs = 800 secs
(ignoring single delay of 1 sec)
= 13 mins 20 secs

Of these six methods, Method 2 consumes the most communication time—11 hours—while Method 5 uses up the least—20 seconds. A high-quality optimizer, together with adequate statistics on the database, would select Method 5.

Exercise 25.2 at the end of this chapter requests the reader to (1) construct a sample database that is distributed to only two sites, (2) construct a sample query that requires information be retrieved from both sites and combined by means of a join, and (3) show that the longest time in communication across the network is N days and the shortest time is N seconds.

This exercise is not particularly difficult and demonstrates the importance of designing a high-quality optimizer in each DBMS of a distributed database management system. Such an optimizer normally selects the approach with the least communication time, since this time dominates the loading on all resources when a command happens to involve two or more sites. If the communication happens to be trans-Atlantic or trans-Pacific, the differences in communication time mean very large differences in communication cost.

These examples suggest that, if the optimizer in a distributed database management system is either weak or missing entirely, there is a serious risk that certain commands will consume an unacceptable time on the communication lines and will also make the costs of communication too heavy. In other words, a distributed relational DBMS with a weak or non-existing optimizer is a DBA's nightmare.

The examples also suggest the following question: What is an adequate collection of statistics about the database? The DBMS should occasionally generate, and use in every optimization, at least a minimal collection of statistics. This collection consists of the number of rows in each base relation, together with the number of distinct values in every column of every base relation. From these statistics and the assumption that within every column the distribution of values is uniform, the DBMS can calculate for every column the expected number of occurrences of each distinct value within that column.

It is important to realize that statistics about a database do not normally change significantly whenever a single insertion, deletion, or update is executed. Therefore, the DBMS need not modify the statistics whenever an insertion, deletion, or update is executed. There would be a severe loss of performance if the DBMS attempted to keep the statistics as up-to-date as that. In many situations, it is quite adequate if the DBMS (1) generates statistics about a base table when that table is created and loaded, and (2) updates the statistics either once every week, or only when they have changed significantly.

It is known that the usual assumption of uniform distribution within each column of the distinct values in that column is quite often far from the actual distribution. Adoption of this assumption, however, is a significant step in the right direction.

RX-24 Minimum Standard for Statistics

The DBMS must maintain at least simple statistics for every relation and every distinct column stored in a distributed database. The statistics should include the number of rows in each relation and the number of distinct values in each column, whether or not that column happens to be indexed.

In early versions of relational DB2 products, statistics were generated for only those columns that happened to be indexed. Thus, statistics on non-indexed columns were simply not available to the optimizer. This serious design flaw was corrected in later versions.

To appreciate the seriousness of this flaw, suppose that one is given the description of a distributed database, including which columns of the base relations are indexed. It is then possible to conceive sample queries whose performance is heavily dependent on statistics being available concerning non-indexed columns. This exercise demonstrates the total inadequacy of a DBMS that does not maintain statistics on *every column*, and whose optimizer fails to make full use of these statistics.

25.1.2 More on Optimization in the Distributed Case

Assume that the DBMS is receiving query and manipulative commands expressed in a relational language in a logic-based style. Optimization involves the following five steps.

1. Convert the given command into a canonical form based on first-order predicate logic.
2. Convert the command into a sequence of relational operators, a sequence that is simply and directly related to the canonical form.
3. Examine all the ways in which this sequence can be altered without altering the final result of the command.
4. Deduce, for each viable sequence of operators,
   a. which of the operators can be concurrently executed at different sites, and
   b. which access paths provide the shortest execution time for each operator.
5. Calculate for each viable sequence of operators, in combination with the best access paths for that sequence, the consumption of resources using a linear combination of processing load at each site involved, input-output load at each site involved, and communication load on the network. The coefficients in this linear combination are those established (and seldom changed) by the DBA.

RX-25 Minimum Standard for the Optimizer

The optimizer in a distributed DBMS must be capable of estimating the resources consumed in executing a relational command in a variety of ways. To generate the most effective target code, the optimizer must combine the three main components (processor time, input-output time, and time on the communication system) using a linear function in which these times appear with coefficients selected initially by the system, but alterable by the DBA.

It should be noted that relational algebra plays a significant role in the second step in optimization. My work in the period 1968–1972 [Codd 1969–1971d], during which I developed both the algebra and the logic (and, incidentally, their inter-relatedness) for querying and manipulating relations of arbitrary degree, lays the foundation for this step.

The **semi-theta-join** operator was described in Section 5.2.2. This operator can prove useful in efficiently executing **joins** between relations that happen to be stored at different sites, but it is not always the most efficient technique, because in some cases it can involve an increased amount of inter-site communication. This fact, however, should not discourage DBMS vendors from incorporating **semi-theta-join** into their designs. At present, I do not know of any distributed DBMS product that uses it.

A final note on performance distinguishes the site at which a relational request is entered from the site or sites at which it is executed. The following feature was suggested by Professor Michael Stonebraker of the University of California at Berkeley.

RX-26 Performance Independence in Distributed Database Management

In a distributed relational DBMS, the performance of a relational request is to a large extent independent of the site at which the request is entered.

25.2 ■ Other Implementation Considerations

Two types of concurrency must be supported by a relational DBMS. The first kind, called *intra-command concurrency,* consists of treating various portions of a single relational command as independent tasks, and executing these tasks concurrently. The second kind, called *inter-command concurrency,* consists of executing two or more relational commands concurrently.

RX-27 Concurrency Independence in Distributed Database Management

The DBMS supports concurrency of execution of relational operators between all of the sites in the network. Application programs and activities by end users at terminals must be logically independent of this inter-site concurrency, whether the DBMS supports intra-command concurrency or inter-command concurrency or both. These programs and activities must also be independent of the controls (usually locking) that protect any one action from interfering with or damaging any concurrent execution.

As previously mentioned, execution of a *single relational command* can involve activity at multiple sites. Therefore, the execution of a *single transaction* can certainly involve activity at multiple sites. Aborting such a transaction would therefore involve recovery at multiple sites.

RX-28 Recovery at Multiple Sites

If it is claimed that the DBMS provides full support for distributed database management, then without user intervention the DBMS must support and coordinate recovery involving multiple sites whenever it has been necessary to abort a transaction at multiple sites. Application programs and activities by end users at terminals must be independent of this inter-site recovery.

The locking scheme implemented in the DBMS must detect deadlocks that may occur between actions at distinct sites. These are often called *global deadlocks,* but a better term is *inter-site deadlocks.*

RX-29 Locking in Distributed Database Management

The DBMS detects inter-site deadlocks, selects one of the contending activities, backs it out to break the cycle of contention, and forces it to wait until a fresh occurrence of the deadlock is avoided as a result of one contender completing or absorbing its transaction. Relational languages contain no features specifically for the handling of deadlocks.

For further information on the intricate controls needed to support the management of distributed data safely and reliably, see [Williams et al. 1981].

25.3 ■ Heterogeneous Distributed Database Management

Today, many companies make use of several computer systems in their business. It is quite likely that these systems were acquired from different vendors, and that each is operated independently of the others. There is a growing demand for software that supports the sharing of data across these systems.

In database terms, the software needed is called a *heterogeneous distributed database management system.* Ideally, such a system should be able to support all the user-oriented features of the homogeneous case. This means that the system must be able to translate correctly, and in a single uniform way, any statements expressed in a single relational language, with the following additional kinds of independence:

- hardware independence;
- operating-system independence;
- network independence;
- DBMS independence; **and**
- catalog independence.

Today, there exists a great diversity in the hardware and software of the various vendors, in spite of all the effort that has gone into creating information processing standards. For example, no two versions of SQL from different vendors are completely compatible. Sometimes this is even true of any two versions from a single vendor. Another example: no two versions of the DBMS catalog from different vendors are compatible with one another.

The lack of appropriate and enforced standards, and the non-conformity of products with respect to existing standards, make the heterogeneous distributed DBMS an extremely ambitious goal. Except in quite simple cases of heterogeneity, the products that emerge are likely to have an extremely large number of cases of user-unfriendly exceptions, which make some of them impractical and unacceptable.

As an aside, the occurrence of duplicate rows within a relation in either the homogeneous or heterogeneous case presents a quite unnecessary additional problem.

Occasionally, one hears that a non-relational DBMS, such as IMS, is to be in the same network as a relational DBMS. Unless the non-relational DBMS is very simple, the problems encountered in trying to make this work are enormous [Date 1984].

25.4 ■ Step by Step Introduction of New Kinds of Data

There are several ways in which new kinds of data can be introduced into a distributed database. The following example stresses initiative at a particular site (site X), and how that initiative can be introduced step by step into the network using the features that have been described in Chapters 24 and 25.

In this example new kinds of data are introduced in a sequence of six steps. The abbreviations I, V, A mean integrity constraints, view definitions, and authorization constraints respectively.

The six steps for introducing new kinds of data into site X are:

1. create the necessary domains and relations—an addition to the local catalog at site X;
2. load some test data in these relations;
3. create whatever I,V,A in the local catalog are needed for purely local use, principally for testing, and make the tests;
4. obtain clearance from the global DBA for some users at site X to make read-only use of data in the network to participate in conditions for relational requests that confine their modifying activities to the new data only;
5. load regular (non-test) data into the new relations;
6. negotiate with the global DBA the introduction of these new relations into the network as site X participants.

The negotiation in Step 6 can be broken down into five parts:

6.1 examine whether any new domains created at site X can be identified as domains that already exist on the network—if so, use the global domain name and definition;

6.2 determine the correspondence between the local names newly introduced at site X and the existing collection of global names;

6.3 define necessary views
6.4 define integrity constraints
6.5 define authorization constraints

} these may straddle new and old data and possibly several sites

25.5 ■ Concluding Remarks

The relational model represents the best existing technology for supporting distributed database management. So far, it is the only approach that supports a language of adequate level, a language in which the user is able to issue a request without dictating to the system how it is to be carried out. To quote Dr. Bruce Lindsay of IBM Almaden Research Center, San Jose:

> Single-record-at-a-time DBMS products (the old approach) in which the user or programmer has to navigate his way through the database are the kiss of death in managing distributed databases.

The level of language is more than a matter of how many records or rows can be retrieved using a single command. It is also more than the complexity of the logical condition that can be expressed in a single command to determine which pieces of information are to be retrieved. Comprehensibility of statements expressed in that language represents an extremely important concern. This applies not only to end users, who may be nonprogrammers, but also to application programmers, who often must maintain programs written by people other than themselves. In such a task, comprehensibility of the statements in those programs is a *sine qua non.*

Here are some more specific reasons why the relational model lends itself to the management of distributed data. The first reason, *decomposition flexibility,* is applicable to two important tasks: (1) the task of distributing a large database to multiple sites (discussed at length in Section 24.4), and (2) the task of decomposing logically expressed relational commands into sequences of operators of the relational algebra.

From the discussion in Section 24.4, it should be obvious that the relational operators provide an extremely flexible tool for carving up the information for distribution purposes. A single relational command can touch many columns scattered over many relations. Such a command can be decomposed into the basic relational operators that are involved. The power of decomposition of relational commands should also be obvious from the discussion of optimization in Section 25.1.

The second reason why the relational model lends itself to the management of distributed data is *recomposition power.* After a relational command has been decomposed into basic relational operators acting on data at various sites, these sites return data in the form of derived relations to the requesting site. This site now has the task of recombining these returned relations into the single relation that is the overall result that was requested. All the relational operators are available for specifying this recombining activity.

The third reason is *economy of transmission.* As discussed earlier, upon receipt of a relational command, a distributed DBMS decomposes it into several basic relational operators that can be applied to the relations stored at various sites. These operators, expressed as simple relational commands, are sent as messages to the appropriate sites. Each command may involve the processing of hundreds, thousands, or possibly millions of rows in relations. If the DBMS had been an old single-record-at-a-time product, a message across the network would have been necessary for each one of the hundreds, thousands, or millions of records involved. Thus, a relational distributed database management system can be orders of magnitude cheaper than a non-relational DBMS in terms of inter-site communication costs, as well as orders of magnitude faster.

The fourth reason is *analyzability of intent* and *optimizability*. A query or manipulative command expressed in a relational language tells the DBMS what kinds of information the user wants and under what conditions, but does not specify how the system is to find and extract this information. It is therefore reasonable to say that (1) such a command expresses the user's intent, and (2) the methods used by the system to satisfy this intent are left entirely up to the system. These facts give the DBMS a very wide choice of methods from which to make a selection.

Thus, the scope for optimization is significantly wider than that of any other approach known today. In turn, this means that, with respect to the management of distributed data, it is very difficult for any non-relational DBMS to compete in combination of cost and performance with a relational DBMS that is equipped with a well-designed optimizer. Note that it is essential for an optimizer to be able to re-optimize commands in a transaction that touches parts of the database where the statistics have changed significantly. Then the DBA can feel confident that the system is really helping him or her to meet the DBA's responsibilities.

The fifth reason, *distribution independence*, was explained in Chapter 20 as one of the means by which a user's investment is protected if a relational DBMS is acquired. First of all, application programs can be developed for a non-distributed version of a relational DBMS. A distributed version of that same DBMS (if it is completely language-compatible with the non-distributed version) can then be installed. The database can be distributed to multiple sites that are geographically separated. The application programs that were originally developed for the non-distributed version will, without change, run correctly on the distributed version.

Also, the data may be re-distributed across these sites and possibly others. Once again, application programs will continue to run correctly, without changing them. This ease of redistribution is an important requirement for every company that acquires a distributed database management system, a requirement that companies often overlook.

It has been proved that distribution independence is supportable by the relational model by means of prototype relational database management systems, such as System R (non-distributed) and System R* (distributed) [Williams et al. 1981]. Not one of the non-relational database management systems has been proven to be effective in this respect. The principal reasons for this superiority in the relational approach are (1) the very high level of relational languages, and (2) the sharp separation between the user's perception and manipulation of the data, on the one hand, and the storage representation and access methods used by the DBMS, on the other.

To recapitulate, the five reasons why the relational approach lends itself to the interrogation, manipulation, and control of distributed data are as follows:

1. decomposition flexibility;
2. recomposition power;

3. economy of transmission;
4. analyzability of intent; **and**
5. distribution independence.

Exercises

25.1 Why does a DBMS need any statistics about the database? Why does a DBMS need statistics about each column of each relation in the database? Why is it insufficient for the DBMS to have statistics about indexed columns only? What are the minimum statistics required by RM/V2, and where are they stored?

25.2 Construct a sample database which is simple (three relations are adequate) and which is distributed to only two sites (two of the relations at one site, one at the other site). Construct a sample query that requires information be retrieved from both sites and then combined by means of a **join.** Select a set of realistic parameters for the communication rate across the network, the time to access the network for each message, and the number of bits per row of a relation. For some integer N of your own choosing, show that:

1. If the optimizer does a bad job, the time spent in communication across the network is approximately N days.
2. If the optimizer does a good job, the communication time is approximately N seconds.

Hints:

1. The three relations can be the usual suppliers S, parts P, and capabilities C.
2. S and C can be stored in site # 1, and P in site # 2.
3. Consider the query: Which suppliers are based in London and can supply instruments for airplanes?
4. An example of parameters and their values that may be assumed:
   - Network access time for each message = 1 second.
   - Transmission speed = 10,000 bits per second.
   - Each record consists of 10,000 bits.

   Note that there are 86,400 seconds in a 24-hour day.

25.3 What does it mean to assert that application programs and terminal activities must be independent of inter-site concurrency?

25.4 Suppose that a transaction T straddles two or more sites. Consider the following five assertions:

1. T must involve retrieval only;
2. T must involve insertion only;
3. T must involve update only;

4. T must involve deletion only;
5. T can involve any combination of retrieval, insertion, update, and deletion commands.

Which of these five assertions is true, and which is false? Which of the currently available relational DBMS products supports such multi-site execution of a transaction?

25.5 Is it possible for the execution of a single relational command to straddle two or more sites? Supply an example showing the need for this. Which of the currently available DBMS products supports such multi-site execution of a single command? Are there any constraints on the type of command?

■ CHAPTER 26 ■

Advantages of the Relational Approach

The advantages of the relational approach over other approaches to database management are so numerous that I do not claim that the 15 advantages discussed in this chapter constitute a complete list. My opinion regarding the various pre-relational approaches is that the only advantage they enjoy is that some large-scale users have a very large investment in those systems—not only in the form of large quantities of company data represented in a way peculiar to the pertinent DBMS, but also in the form of application programs that appear to work correctly against that data. Such programs are very difficult to translate into the corresponding programs needed on a relational database, a difficulty largely due to the lack of a discipline in the design and use of pre-relational DBMS products.

Even though the conversion from pre-relational to relational DBMS products is very labor-intensive and costly, I believe that users should start now to plan such a conversion, and execute the plan step by step. In this way, users can realize at an earlier time the benefits in cost, efficiency, and integrity of managing their databases by means of a more modern, relational DBMS.

When a company postpones conversion to a relational DBMS, it incurs the cost of conversion sometime in the future, when the cost will be significantly higher because of the labor-intensive nature of conversion. During the period of delay, the company also loses the productivity, safety, and security of the relational approach.

26.1 ■ Power

The relational approach is very powerful and flexible in access to information (by means of ad hoc queries from terminals) and in inter-relating information without resorting to programming concepts (e.g., iterative loops and recursion). The power stems from the fact that the relational model is based on four-valued, first-order predicate logic.

26.2 ■ Adaptability

Errors are often made in both logical and physical database design. When a database is created, it is virtually impossible to predict all the uses that will be made of it. With regard to changes in use of the data and changes in the database traffic, the relational approach is much more forgiving than any other approach. (There is no claim, of course, that it is totally forgiving.)

The features that make the relational approach more capable of accommodating change are the immunity of the application programs and terminal activities to the following types of changes:

1. the storage-representation and access methods;
2. the logical design of base relations;
3. integrity constraints;
4. the deployment of data at various sites.

Database design is still necessary. When it becomes necessary to change either the logical or the physical design, however, a relational database is much more adaptive to the changes because of these four features.

26.3 ■ Safety of Investment

How safe is an investment in a relational DBMS? Will the relational approach be replaced by some new, incompatible approach in the near future?

Many people attach considerable importance to the fact that the relational model has a sound theoretical foundation. It is based on predicate logic and the theory of relations, parts of mathematics that have taken about two thousand years to develop. Thus, it is highly unlikely that the theoretical foundation will be replaced overnight. This observation makes the relational approach a reasonably safe investment on the part of DBMS vendors and DBMS users.

Furthermore, the relational approach is the only one to offer the four important investment-protection features cited in Chapter 20: (1) physical data independence, (2) logical data independence, (3) integrity independence, and (4) distribution independence. The kinds of investments protected by these features include investments in the development of application

programs and in the training of the programmers and end users. In acquiring any new DBMS product—whether relational or not, and whether hardware, software, or both are involved—a company is likely to invest more money in application programming and training than in actually purchasing the DBMS product.

26.4 ■ Productivity

Early users of relational DBMS products report a substantial increase in the productivity of their application programmers. This advantage can be traced to several facts:

- Application programs developed to run on top of a relational DBMS contain significantly fewer database statements than corresponding application programs developed to run on a non-relational DBMS.
- These statements convey intent, and are therefore easier to understand by people responsible for the maintenance of the programs, who may not be the original developers.
- The database statements are clearly separate from the non-database statements, and can be separately developed and speedily debugged using terminals.
- The burden of achieving the best performance is largely removed from the application programmer and interactive user, and is instead assumed by the DBMS.

These early users also report that end users on their systems are able to make extensive use of the information in relational databases (including the generation of requested reports) without requiring help from application programming staff. This is mainly because, in constructing the relational model, I rejected the need for users to have programming skills in retrieving and modifying data in the database (skills such as designing iterative and recursive loops and creating I/O channel commands).

This ability of end users to make direct use of information in relational databases without assistance is undoubtedly the primary reason why the relational DBMS market has expanded so quickly. In just a few years, it has overtaken the market for all its predecessors. One of the many reasons that users need substantially increased productivity is that it enables them to plan and launch new products much more rapidly.

26.5 ■ Round-the-Clock Operation

The relational approach is designed for round-the-clock operation of the database management system. Pre-relational DBMS products often required the traffic on the database to be brought to a halt if changes were to be

made in the access methods or access paths, or in any aspect of the database description (e.g., the record types in the database).

In a relational DBMS, this interruption of service is unnecessary for the following reasons:

- the sophisticated nature of the locking scheme;
- the automatic recompiling of just those database manipulation commands adversely affected by any changes in the database description;
- the inclusion of both data-definition commands and data-manipulation commands within any relational language.

Indexes may therefore be dynamically created and dropped. New columns may be introduced dynamically into a base relation. Old columns may be dynamically dropped. Authorization may be dynamically granted and dynamically revoked. Domains, views, integrity constraints, and functions may be dynamically created and dropped.

26.6 ■ Person-to-Person Communicability

One of the many problems with pre-relational DBMS products was that the application programmers had to have extensive training of a very narrow kind, training oriented toward the particular DBMS installed. This meant that it was virtually impossible for a company executive to find out for himself or herself what kind of information was stored in the database. To do so, the executive had to ask a database specialist to interrogate either the system or manually prepared documents related to the particular database, and then translate his or her discovery into terms comprehensible to the executive.

With the relational approach, an executive can have a terminal on his or her desk from which answers to questions can be readily obtained. He or she can readily communicate with colleagues about the information stored in the database because that information is perceived by users in such a simple way. The simplicity of the relational model is intended to end the company's dependence on a small, narrowly trained, and highly paid group of employees.

It is important to note that, if the relational approach is being used, end users and application programmers can at last talk to one another about both the content of the database (because of its simple structure) and database actions (because end users and application programmers employ a common database sublanguage).

26.7 ■ Database Controllability

The relational model was designed to provide much stronger machinery than any pre-relational DBMS for maintaining the integrity of the database. The

motivation was that many companies store in their databases information that is vital to the continued success of the company. The accuracy of this information is therefore of great concern. Now, it is well-known that preventing the loss of accuracy or integrity is much more readily mechanized than is the cure of such loss.

As explained in Chapters 13 and 14, the relational model not only supports the two types of integrity (entity integrity and referential integrity) that apply to every relational database, but also supports domain integrity, column integrity, and user-defined integrity. In this last category, the model provides the power of four-valued, first-order predicate logic in the creation by users (principally the database administrator, of course) of integrity constraints that chiefly reflect the company policy, governmental regulations, and certain semantic aspects of the data used in designing the database.

The environment in which the company operates is bound to change as time passes. Therefore, it is unrealistic to expect that company policy, government regulations, and the semantics of data will somehow remain unchanged. For this reason, the relational model supports integrity independence, which permits integrity constraints to be changed without changing application programs. This integrity independence makes it much less costly to implement changes in integrity constraints, and therefore makes the company much more adaptable to environmental changes.

Of course, in many of the relational DBMS products on the market today, support for the integrity features of the relational model is quite weak. This weakness reflects irresponsibility on the part of DBMS vendors.

26.8 ■ Richer Variety of Views

Pre-relational DBMS products were quite weak in their support of views. For example, the CODASYL-proposed DBTG standard[1] of the 1970s supported nothing more than those views that just one of the relational operators **project** can generate. The relational model, on the other hand, supports the full power of four-valued, first-order predicate logic in defining views.

Unfortunately, the relational DBMS products available today do not yet possess the strength of the relational model in regard to defining views. Within the next decade or two, however, we can expect the necessary product improvements.

26.9 ■ Flexible Authorization

Pre-relational DBMS products were embarrassingly weak in their support for permitting or denying access to parts of the database. Usually the access control was based on explicit denial of access by specified users to specified

[1]From the Report of Data Base Task Group of CODASYL Programming Language Committee, April 1971. Available from ACM, BCS, and IAG.

record types or specified fields. Usually these DBMS products also failed to support access control that is dependent on values encountered in the database.

The relational model, on the other hand, uses view definitions based on four-valued, first-order predicate logic to determine the portions of the database to which access will be permitted; these portions can easily be defined to be value-dependent. A user is then permitted by the system to access one or more specified views only, and to use certain specified relational operators only on each view.

26.10 ■ Integratability

On top of a DBMS, a user is likely to need products such as application development aids, report generation support, terminal screen painting support, graphics support, support for the creation and manipulation of business forms, and support for logical inference. Pre-relational DBMS products offered nothing more than a low-level, single-record-at-a-time interface to such products.

Relational DBMS products, on the other hand, offer a powerful, multiple-record-at-a-time language for this purpose, making it significantly easier to develop the products on top. As a result, we can expect to see a vast proliferation of products that interface with relational DBMS products and make use of the data supported in the databases.

Those products that support logical inference (e.g., a few of the so-called *expert systems*) can easily exploit the relational language interface, since a language for logical inference must be closely related to predicate logic, which is the most powerful known tool for making precise logical inferences.

26.11 ■ Distributability

Now that vendors have discovered the relational approach to database management, numerous systems capable of managing distributed databases are beginning to appear on the market. One DBMS product supports not only retrieval from remote sites, but also insertion, update, and deletion at remote sites, as well as full-scale transactions, each of which may straddle multiple remote sites without the user being aware of which sites were involved.

In Chapter 25, I discussed five principal reasons why the relational approach has been far more successful than any non-relational approach in managing distributed databases:

1. decomposition flexibility;
2. recomposition power;
3. economy of transmission;

4. analyzability of intent; **and**
5. distribution independence.

26.12 ■ Optimizability

The ability of a DBMS to assume a large portion of the burden of achieving good performance on all database interactions depends on its ability to generate the best quality target code from the source code in which the user expresses these interactions. This translation from source code to efficient target code is usually called the *optimization problem.*

Although present relational DBMS products differ significantly in their abilities to handle the optimization problem, almost all of them have far superior capabilities in this area when compared with pre-relational DBMS products. This is because of both (1) the high level of relational languages and (2) the sharp separation of the user's perception of the data from the representation of this data in storage and from the access methods presently in effect.

There are several reasons why companies should avoid depending on their application programmers, however skilled, for obtaining good performance from the DBMS for their application programs. Even the most highly skilled programmers sometimes find it difficult to concentrate on the job at hand. In addition, the DBMS is in a far better position than the programmer to keep track of DBMS performance, and know when it is necessary to make changes in access methods and access paths and to recompile certain relational commands to cope efficiently with these changes.

These remarks apply whether the database is distributed or not. In the distributed case, however, the DBMS can provide an additional service. It can help determine at appropriate times whether and how the database should be re-distributed. I do not know of any current product that provides this service. Perhaps the problem will be a good subject for one or more doctoral dissertations.

26.13 ■ Concurrent Action by Multiple Processing Units to Achieve Superior Performance

For many years, people in the computer field have been aware of the vast difference in speeds of processing units, on the one hand, and secondary storage such as disks, on the other. There have been suggestions that what was needed in commercial data processing was a high-level language for input and output, but no such proposal has been forthcoming. Now, the relational approach to database management offers vast new opportunities to exploit concurrent action by multiple processing units—not just 2, 4, or 6 units, but 50, 100, or more. For example, Tandem has shown that, by

adding processing units, it is possible to obtain an improvement in speed of the whole system that is linear with cost.

Tandem's NonStop SQL is more powerful in speed than any non-relational DBMS products because automatic concurrent actions are made possible by the relational approach, and Tandem exploits this concurrency opportunity to the hilt. By "automatic," I mean that the concurrency is not programmed by either the user or the application programmer.

Incidentally, Tandem's announcement and the audited benchmark finally laid to rest the ill-conceived notion that the relational approach would never be accepted because it "performed poorly." No other architectural approach is known for achieving very high performance on database management: for example, over 1000 simple transactions per second, coupled with fast response (e.g., less than 2 seconds per transaction). The IBM product IMS Fastpath, once considered a leader in performance among database management systems, has been overtaken by Tandem's NonStop SQL coupled with Tandem's NonStop architecture. Moreover, IMS and IMS Fastpath cannot catch up in this race, since they are single-record-at-a-time systems, in which opportunities for concurrent action are quite limited.

It is worthwhile to distinguish between two types of concurrency:

1. inter-command concurrency—concurrency between the execution of distinct relational commands;
2. intra-command concurrency—concurrency between tasks that are part of a single relational command.

The present release of Tandem's NonStop SQL concentrates on the first type, while the Teradata DBMS product concentrates on the second type of concurrency.

26.14 ■ Concurrent Action by Multiple Processing Units to Achieve Fault Tolerance

Today, as more and more companies become international in scope, they must operate effectively in multiple time zones. Such companies tend to need continuous, round-the-clock operation of their complete systems. Thus, if a processing unit, channel, or disk unit fails, the system as a whole should continue to function, even though its performance may be reduced.

Tandem Corporation has shown that this fault tolerance can be achieved in data processing through its NonStop architecture (hardware and software), and in database management through its NonStop SQL (software). Thus, it is clear that one advantage of the relational approach is that it lends itself to a high degree of concurrent action, which, with an appropriate architecture, in turn leads to a high degree of fault tolerance.

In scientific computing, arrays and matrices offer significant opportun-

ities for concurrent execution; this has been exploited to obtain performance improvements. In commercial data processing, relations offer similar opportunities. The need in commercial data processing is greater, however, in that not only must performance improvements be obtained, but also significant improvements in fault tolerance must be achieved.

26.15 ■ Ease of Conversion

If and when the relational approach to database management becomes obsolete, it will be much easier to convert to whatever approach replaces the relational model. There are two chief reasons:

1. all information in a relational database is perceived in the form of values;
2. the language used in creating and manipulating a relational database is much higher in level than the languages used in pre-relational database management.

26.16 ■ Summary of Advantages of the Relational Approach

To recapitulate, the relational approach is the leading approach to database management today because of its sound theoretical foundation plus the following 15 major advantages it has over other approaches.

1. powerful approach;
2. adaptability;
3. safety of investment;
4. productivity;
5. round-the-clock operation:
   - dynamic tuning
   - dynamic change of database description;
6. person-to-person communicability;
7. control capability, especially integrity constraints;
8. richer variety of views;
9. flexible authorization;
10. integratability;
11. distributability;
12. optimizability;
13. radically increased opportunities for concurrent action by multiple processing units to achieve better performance;

14. radically increased opportunities for concurrent action by multiple processing units to achieve fault tolerance; **and**
15. ease of conversion to any new approach.

Exercises

26.1 The introduction of relations into the management of large databases has spurred the development of commercial computer systems with large numbers of processing units capable of executing many commands concurrently. Identify two types of concurrency, and give two practical reasons why the multiplicity of processing units means improved service to users.

26.2 A designer of DBMS products wrote a lengthy memorandum in the mid-1970s asserting that data models were a waste of time. He asserted that all that any vendor needed to supply were access methods. Take a position on this and compare:

- the old access methods (SAM, ISAM, PAM, DAM, etc.);
- the old database management systems (IMS, IDMS, ADABAS);
- the relational model.

26.3 Cite four ways in which RM/V2 is adaptable to change.

26.4 Cite four reasons why a relational DBMS yields substantial increases in productivity.

26.5 What capabilities does RM/V2 provide to help the DBA keep the database well-controlled and accurate?

26.6 Cite five reasons for expecting the relational model to be more capable in managing distributed databases than any single-record-at-a-time DBMS.

26.7 Why were pre-relational DBMS products unable to support optimizing as part of the translation from source language into efficient target language?

■ *CHAPTER 27* ■

Present Products and Future Improvements

The extent of support of the relational model (even Version 1) in present DBMS products is disappointingly low. In Chapter 27 we discuss the major errors of omission and errors of commission in these products. These errors have a negative impact not only on the DBMS products themselves, but also on products developed to run on top of these DBMS products. We will discuss the future of these products on top from two viewpoints:

1. The future if logically based.
2. What is to be expected.

The next subject to be discussed is the relatively new area of exploiting the many opportunities for concurrency offered by the relational approach in order to achieve top performance and fault tolerance. The last topic is the ability to communicate between machines of different architectures, including IBM's SAA (Systems Application Architecture) strategy.

27.1 ■ Features: the Present Situation

Present relational DBMS products and languages, including the language SQL, support at most only half of the relational model, and consequently fail to provide some of the significant benefits of the relational approach. I have encountered among the staff of vendors a curious tendency to continue supporting obsolete methods because they are familiar. To some, familiarity appears more important than technical progress.

27.1.1 Errors of Omission

The most important errors of omission in present versions and releases are domains as extended data types, primary keys, and foreign keys. The IBM product DB2 is one of the few that provides partial support for the keys. DB2, however, still fails to require exactly one primary key for each base R-table on a mandatory basis. It does not support the existence in the entire database of two or more primary keys on a single primary domain. These keys, of course, would necessarily occur in distinct R-tables. One consequence of two or more primary keys on a common domain is that a given foreign key may have two or more target primary keys.

Finally, updating a primary key with the same update being applied to all corresponding foreign keys (1) requires the participation of the host language, (2) is far too complicated, and (3) depends on some application programmer knowing on an up-to-the-millisecond basis the state of the foreign keys for a given primary key. If the DBMS is relational, only the system can hold such knowledge.

Additional features that are not supported in present versions of most relational DBMS products include referential integrity (although this is partially supported in DB2 Version 2), user-defined integrity constraints, and user-defined functions.

Because of the inextricably interwoven features of the relational model, each of these omissions negatively impacts numerous benefits of relational DBMS. In particular, since domains are part of many features of the model, none of these features can be fully supported without first supporting domains. To cite just three examples:

1. The DBMS should require that each foreign key be subjected to referential integrity with respect to all of the primary keys with the same domain as the foreign key. To manage this on a highly dynamic basis, the system must know all the keys that draw their values from a common domain, and this knowledge must be current on a millisecond basis.
2. Each pair of comparand columns in **joins, relational division,** and certain **selects** should be based on a common domain. The DBMS should always check this safety feature, unless the user requests the rarely used DOMAIN CHECK OVERRIDE.
3. A DBMS should check whether a requested **union** is meaningful or not. To do this, the system must be able to check the domains of all columns involved in the **union.** The same applies, of course, to **relational difference** and **intersection.**

Finally, view updatability in present products is extremely weak and inadequately investigated. Consequently, logical data independence is hardly supported at all.

27.1.2 Errors of Commission

In present DBMS products, there are not only numerous errors of omission, but also significant errors of commission, only three of which are discussed here. Among the errors of this type, SQL allows duplicate rows within a relation, whether base or derived. (See Chapter 23 for the resulting penalties.) Apparently, many vendors and ANSI have not noticed this and other flaws in SQL.

A second major flaw in SQL is the inadequately investigated nesting of queries within queries. It should always be possible for the DBMS to translate a nested version into a non-nested version. One important reason is that the optimizer can then do an equally good job, no matter which version is presented by the user. Failure in this regard places a major part of the performance burden right back in the user's lap, just as in pre-relational DBMS products—and this is *not* what I intended. Clearly, there must exist a canonical form into which all relational requests can be cast.

A third major flaw lies in the way that present DBMS represent and treat missing database values. In RM/V2, both the representation of the fact that a database value is missing and the treatment of each missing value are independent of the type of value that is missing (see Chapter 8). Several relational DBMS products fail the representation requirement by representing the fact that a numeric value is missing differently from the fact that a character-string value is missing. A few relational DBMS products satisfy the representation requirement, but fail the treatment requirement.

The only feature in SQL related to the treatment question is the clause IS NULL. This feature is clearly inadequate. Moreover, four-valued logic should be provided under the covers in any relational DBMS product; the logic supported by most relational products today is not even three-valued.

27.2 ■ Products Needed on Top of the Relational DBMS

It is clear that the rapidly expanding market for relational DBMS products opens up a substantial market for products that operate on top of these systems. Such products must interface with relational DBMS; they do so by using whatever relational language the relational DBMS supports, usually SQL. Thus, in recent years vendors, especially software vendors, have announced many new products that operate on top of the more popular relational DBMS products.

Examples of such products follow:

dictionaries	forms support
database design aids	screen painting
application development aids	graphics support
expert-system shells	natural-language support

computer-assisted software engineering (CASE) tools

re-engineering tools

27.3 ■ Features of the Relational DBMS and Products on Top Assuming that the Future is Logically Based

If a relational DBMS is to provide all of the benefits of the Relational Model, it is important that the errors of omission and commission be fixed first. This is the only way to avoid giving these errors a permanence that results from the substantial investment users make on the assumption that the features of a system will continue to exist.

Then, it will be possible for both vendors and users to proceed on a secure foundation with (1) all the products on top and (2) distributed relational DBMS products.

27.4 ■ Features of Relational DBMS and Products on Top, Assuming that Vendors Continue to Take a Very Short-term View

Vendors, however, are forging ahead with both products on top and distributed relational DBMS products, disregarding errors in present relational DBMS products. All the evidence indicates that they will continue to do so.

An inevitable result is that existing errors will become more difficult to fix, because more products and more users will be affected. Over time, the defects and deficiencies in the present versions of SQL will become totally embedded in relational DBMS products.

It is important to be aware that, *first, the language SQL is not part of the relational model. Second, the defects and deficiencies in SQL correspond closely to the various departures of SQL from the relational model.*

27.5 ■ Performance and Fault Tolerance

Many relational DBMS products offer excellent performance; some can outperform non-relational DBMS products. Improvements in performance are being introduced rapidly as each new version or release comes to market.

Today's leader in general fault tolerance, including DBMS fault tolerance, is Tandem, with its NonStop architecture and NonStop SQL. Moreover, the adaptable performance of the NonStop architecture will prove attractive to many companies. If a user's workload grows, he or she can cope with the growth by adding onto the installed system several closely coupled processing units and additional disk-storage units. The user need not replace the entire installed system with a completely new, higher-performance system.

27.6 ■ Performance and Fault Tolerance Assuming that the Future Is Logically Based

As more and more companies become international and operate in multiple time zones, there is a greater need for improved fault tolerance. There is also a growing need for very high transaction rates. Therefore, DBMS vendors can be expected to recognize the growing market for increased DBMS performance and fault tolerance. Eventually, they will also recognize that relational DBMS, coupled with architecture that exploits concurrency, can exceed the performance and fault tolerance of non-relational DBMS by a wide margin.

It would be reasonable for each DBMS vendor eventually to exploit both inter-command and intra-command concurrency in its DBMS products (see Section 26.13 for an explanation of these terms).

27.7 ■ Performance and Fault Tolerance Assuming that the Vendors Continue to Take a Very Short-term View

Why can we expect a logically based future in performance and fault tolerance? Competition will force this, and vendors seem more comfortable in competing in this arena than in any questions related to the services provided by their products.

Relational DBMS vendors will gradually see the advantages of exploiting the concurrency opportunities offered by the relational approach—specifically, performance (usually measured in simple transactions per second), performance adaptability, and fault tolerance.

27.8 ■ Communication between Machines of Different Architectures

It is not easy to design effective communication links between machines of different architectures. Two major problems are (1) the concise representation of control messages and (2) the representation of large blocks of data. Clearly, standards are needed in both areas. Some work has been done on the first, but it appears that no attention is being given by standards committees to the second.

The relational model offers a partial solution to the representation of large blocks of data—namely, how those blocks should be organized and structured. In addition, however, we need standards dealing with the bit-level representation of the various types of atomic data. I believe that the standards committees should have considered the communication of large blocks of data before trying to standardize on the relational language SQL.

IBM has introduced a much-needed standard for its own systems of different architectures. The IBM term is the *systems application architecture* (abbreviated SAA). A major part of this IBM standard is communication between databases managed on these different architectures. This communication involves relational commands and relational blocks of data, an approach I advocated in 1970 in an internal memorandum addressed to a senior IBM manager in Poughkeepsie, New York.

Communication concerning databases between machines acquired from different vendors involves numerous problems of substantial difficulty. Nevertheless, some software vendors are beginning to work on these problems. If they are successful in solving them, the market for their products will be very substantial. The relational approach to database management appears to be the cornerstone of every current attempt to solve this communication problem.

Exercises

27.1 Present relational DBMS products fail to support numerous features of the relational model. List six such features. For each omitted feature, cite two benefits that users lose as a result.

27.2 Many relational DBMS products violate a very fundamental feature in the relational model (a part of that model from its conception 20 years ago). What is this feature, and what harm does this violation create?

27.3 Why will vendors probably be slow to correct the infidelities to the relational model in their products, but will improve their architectures rapidly to support concurrent processing of data from databases?

27.4 What kinds of products are needed on top of a relational DBMS? In what ways is SQL an inadequate database-oriented language for these products to use in communicating with the DBMS?

■ *CHAPTER 28* ■

Extending the Relational Model

28.1 ■ Requested Extensions

I am frequently asked how the relational model can be extended to handle (1) very large quantities of image data, (2) very large quantities of text, and (3) computer-aided engineering design.

These kinds of data appear to require specialized user-perceived representation and specialized kinds of retrieval capability. In other words, the representation and retrieval are different from those of the relational model. At present, I believe that the details of handling these kinds of data should not be explicitly incorporated in the relational model, because (at present and in the foreseeable future) only a minority of businesses and other institutions are concerned with these three kinds of data.

Note, however, that the version support required in computer-aided engineering is provided to a limited degree by the library check-out and return features, Features RM-19 and RM-20 (see Chapter 12).

Instead of expanding the relational model to handle every specialized need, the interfacing features of the relational model should be exploited so that the usual kinds of data handled by the model can be enriched by developing specialized invokable functions. It then becomes unnecessary to introduce new, specialized features into the model. With respect to the relational language, this means the ability to incorporate data extracted from image bases, text bases, or design bases into the target part, the condition part, or both of a relational query. Note also that each image base, text base, and engineering base is likely to have descriptive data associated with it that is relational in character.

What are these interfacing features? First, Features RF-4–RF-7 in Chapter 19 support user-defined functions that can be exploited in both the target part and the condition part of a relational request. Second, if necessary, the names of these functions, the names of the built-in functions, the names of their arguments, and the values of the arguments can all be stored as regular data in a relational database. Numerous other features of the model are user-defined. Two of the most important are user-defined integrity constraints and user-defined extended data types.

Incidentally, the term "user" in "user-defined" includes database administrators and even hardware and software vendors. Such vendors are quite likely to offer a package of functions for use with a relational DBMS in accessing image data.

28.2 ■ General Rules in Making Extensions

Now let us discuss the introduction of new features into the relational model. When extending the representation, manipulative, or integrity aspects of either a DBMS or a relational language beyond the capabilities of RM/V2, I strongly recommend that the problem be examined first at the level of abstraction of the relational model. This means treating the relational model, together with any necessary and relevant extensions, as a tool for solving the problem.

This approach is recommended because it is more likely to yield an extension that is minimal in complexity with respect to (1) user comprehension and (2) implementation. It is also more likely to yield an elegant solution that avoids an unwarranted number of exceptions, each of which must be handled by additional pieces of code in the DBMS, and many of which burden users with exceptions that must be remembered. Such exceptions require different cases to be handled in quite different ways.

Suppose that a simple extension to the relational model (e.g., adding a new authorization feature) will suffice. Then, the extension should be made unless in its present form it runs counter to other features of the relational model. If and when this inconsistency is discovered, a revised version should be created that is not inconsistent.

When examining the usefulness of the model together with any necessary extensions for more complicated types of applications, the following 15 questions should be answered:

1. What is a precise statement of the general problem?
2. What mathematical tools are known to be relevant, and, using examples, how can these tools be used?
3. Does any collection of these tools solve the whole problem without the need for programming skill?
4. Is a collection of relations of the relational model an *adequate, simple* representation tool for the problem?

5. Can a combination of relations, relational operators, and functions solve the manipulative aspects of the problem, while avoiding programming concepts such as pointers and iterative loops? Are any new operators needed? If so, which ones?
6. Even though the functions may have to be coded by a programmer in one of the host languages, can the interface between the functions, their arguments, and the relational language protect the user from programming concepts?
7. Is there any need for the names of invokable functions to be part of the database? Can Feature RF-9 in Chapter 19 (repeated next) be helpful?

RF-9 Domains and Columns Containing Names of Functions

One of the domains (extended data types) that is built into the DBMS is that of function names. Such names can be stored in a column (possibly in several columns) of a relation by declaring that the column(s) draw their values from the domain of function names. Both RL and the host programming language support the assemblage of the arguments together with the function name, followed by the invocation of that function to transform the assembled arguments.

8. Is there any need for the names of arguments for these functions to be part of the database? Can Feature RF-10 in Chapter 19 (repeated next) be helpful?

RF-10 Domains and Columns Containing Names of Arguments

One of the domains (extended data types) that is built into the DBMS is that of argument names. Such names can be stored in a column (possibly in several columns) of a relation by declaring that the column(s) draw their values from the domain of argument names. These arguments have values that can be retrieved either from the database or from storage associated with a program expressed in the HL.

9. Which integrity constraints must be supported?
10. Can a combination of relations, relational operators, and functions solve the integrity aspects of the problem, while avoiding programming concepts such as pointers and iterative loops? Are any new operators needed? If so, which ones?

11. From the standpoint of (a) end users, (b) application programmers, and (c) the DBA, what are the advantages and disadvantages of the relational approach?
12. From each of these standpoints, what are both the manipulative and integrity aspects of the problem?
13. How does this approach compare with other approaches (e.g., hierarchic and network-structured approaches)?
14. Does a relational language that includes the extensions to support DBA-defined integrity constraints (described in Chapter 14) require even more extensions? If so, what are these extensions, and can programming concepts such as iterative loops be avoided?
15. From the standpoint of the DBA, what are the advantages and disadvantages of the relational approach to solving the integrity aspects of the problem when compared with other approaches?

With regard to Question 6, it is neither necessary nor desirable to require that any functions that are involved be coded entirely in the relational language. Such a requirement would necessitate extending the relational language to become just another programming language that supports the coding of all computable functions. The functions can be coded in the host language, in the RL, or a combination.

In executing these steps, it is very important to use proper relations (e.g., no duplicate rows), and to avoid the ordering of rows in any relation whenever the ordering represents information that is not also represented by values in the operand or result relations. It is also important to make sure that the extensions comply with the mathematical closure feature, Feature RM-5 (see Chapter 12). If any one of these recommendations is ignored, the extensions will be incompatible with the relational model.

In this chapter, a bill-of-materials (BoM for brevity) type of problem is used from time to time as an example. This problem is of significant scope because it is a problem of wide ranging application. It is also a good example to use as an illustration of how the model can be extended because there is a very large market for a sound solution, and hence the extensions described for this example are very likely to be made.

In 1987, I developed extensions of the relational model to handle the BoM-type of problem, both from the manipulative point of view and from the integrity preservation point of view. In 1988, I prepared a rather complete technical paper on this subject. One of my motivations for this work was that many people were glibly and falsely claiming that the relational model was incapable of solving this kind of problem. Although I have solved the BoM problem within the relational approach, the emphasis here is not on the solution, but instead on the method by which the solution was created. (The solution will be published elsewhere at a more appropriate time.)

In tackling the BoM problem, I bring to bear the powerful tools of the relational model and propose extensions to this model, especially additional manipulative techniques and integrity-preservation operators. It should be noted that the relational model requires that at least three aspects be covered: structural, manipulative, and integrity. These aspects are now discussed in turn.

28.3 ■ Introduction to the Bill-of-Materials Problem

In dealing with the BoM problem and similar problems, it is necessary to establish a precise theoretical foundation, allowing the methods developed to be handled correctly by computers. Therefore, relevant mathematical tools should be selected, such as graphs, matrices, and relations, along with their various operators.

The main purpose of using these tools is to understand the problem and then invent commands to solve it. However, such commands should be usable by people who do not understand either regular programming or the underlying mathematics.

As in Chapter 5, the terms "graph" and "network" are used to denote a set of points (called *nodes*) together with lines (called *edges*) that connect pairs of these points. When every edge of the graph has an associated direction, the graph is called a *directed graph* (or *digraph*, for brevity). It is then possible to speak of the *starting node* of each edge and the *terminating node* of that same edge. In the bill-of-materials problem, it is appropriate to make the following assumptions:

- No edge has a single node that is both the starting node and the terminating node of that edge.
- The number of edges that start at any selected node and provide an immediate link to any other single node is either zero or one (no other number is either necessary or acceptable).

A principal aim is to extend the relational model so that it can manage a database that happens to contain (but not necessarily exclusively) the kind of information to which the BoM-type of problem can be applied. Of the tools just mentioned, if software packages for presentation purposes are ignored, users will continue to see relations only. Graphs and matrices are used to explain the relational operators and the integrity constraints, as well as to show that implementation is feasible and potentially very efficient.

28.4 ■ Constructing Examples

To attack the problem and later to explain the solution, it is necessary to devise one or more examples that are simple enough to be readily under-

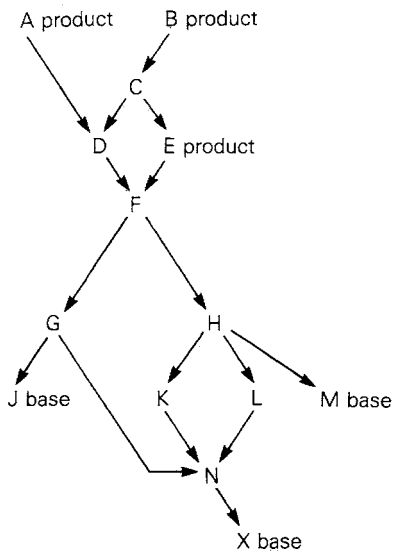
stood, but sufficiently complicated to contain the most serious problems encountered in finding the solution. Even when, for explanatory reasons, the examples are relatively simple, the methods described should be applicable to very large relations such as would pertain in the industrial world.

In the connectivity part of the BoM type of problem, an example of this type is the product-structure graph shown in Figure 28.1. This acyclic graph represents the structure of several products by showing which parts are components of which. For the sake of simplicity, single letters are used as identifiers of distinct kinds of parts as they are assembled (including final products). An edge of the graph indicates that a certain part is an immediate component of some other part. The nodes labeled "product" represent products that are constructed within the company and then shipped out to customers. The nodes labeled "base" represent parts that are *not* constructed in the company. Base parts are likely to be purchased from outside sources and shipped in.

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. That graph is certainly acyclic, but it is very unlikely to be hierarchic. Even in the unlikely case that it begins life 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.*

I claim to have a solution to the general bill-of-materials problem, one

Figure 28.1 **The Structure of Several Products**



that is very concise, that protects the user from iterative and recursive programming, and that provides pertinent integrity constraints as well as manipulative power. However, the **recursive join** described in Chapter 5 is *not* a complete solution to this problem. (The more complete solution will, of course, be published later.)

28.5 ■ Representation Aspects

The problem should be carefully examined to determine if the relations of the relational model are adequate as a representation tool. If a hierarchy or network is involved in the problem, it is not necessarily true that a hierarchic or network data structure is essential, or even necessarily best.

In the BoM problem, the representation issue is how to represent product structure; such structure is often treated as if it were a pure hierarchy. Many observers draw the immediate conclusion that a DBMS is needed that exposes hierarchically structured data to users. Each hierarchic link would represent the fact that one type of part is an immediate component of another. Recent proposals to ANSI are clear evidence of this.

A hierarchy may be an adequate representation in a few manufacturing environments, but in many—probably most—it is not adequate. In these latter environments, a particular type of part may be an immediate component of several types of parts, not just one. A second, all-too-rapid conclusion is that a DBMS is needed that exposes network-structured data to users.

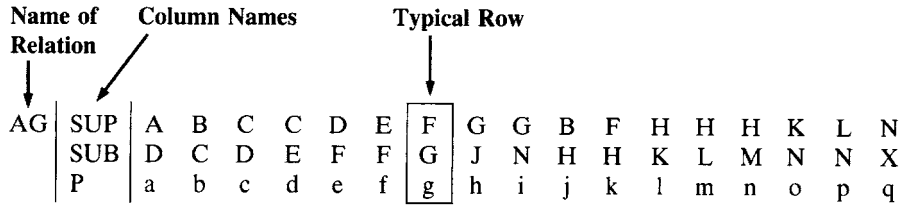
In fact, in 1970 I presented [Codd 1970] an extremely simple representation of product structure in the relational model by means of the COMPONENT relation:

COMPONENT (SUB_P# SUP_P# Q1 Q2 ... Q*n*),

where SUB_P# denotes subordinate part number, SUP_P# denotes superior part number, and Q1, Q2, ..., Q*n* denote immediate properties of each particular subordination.

Note that the columns SUB_P# and SUP_P# draw their values from the same domain, that of part numbers. Incidentally, if (p1, p2, q1, q2, ..., q*n*) is a row of the COMPONENT relation, then part p1 is an *immediate* component of part p2. The fact that part p is a non-immediate component of a part p7 (say) is not directly represented in the COMPONENT relation. A fact of this type can be easily derived by the **recursive join** operator (see Feature RZ-40 in Chapter 5 and [Codd 1979]).

The following relational representation of product structure represents the structure illustrated in Figure 28.1. To save space, the COMPONENT relation is abbreviated AG (for acyclic graph), the relation is listed "on its side," and the immediate properties of each edge are represented by a single lowercase letter.



In a computer-oriented sense, this kind of representation in a relation is adequate for all kinds of networks, whether they happen to be pure hierarchies, acyclic nets, or nets in which cycles may occur. It is also a very simple representation for a computer to manage. For the product-structure network, the acyclic net is adequately general.

In the preceding paragraph, I use the term "computer-oriented sense" because the relational representation is probably not the best for use by human beings, for whom graphs drawn as pictures appear to be more comprehensible and suitable. However, that subject can be discussed separately, and handled by separate code, when presenting data to people in a form more consumable by people (e.g., the formatting of reports)—it has very little relevance to mechanizing the management of data.

28.6 ■ Manipulative Aspects

It was clear at least 10 years ago [Codd 1979] that extensions to the relational model would be required to handle the manipulative and integrity aspects of the BoM application. This application involves manipulating relations that represent acyclic directed graphs. Such manipulations usually involve *transitive closure.* In the context of relations, transitive closure is expressed in the form of **recursive join.** Feature RZ-40 in Chapter 5 is a simple version of this type of **join.**

A slightly more complicated version of **recursive join** is needed for the BoM application. This is because this application has a connectivity aspect that deals with the path finding, and a computational aspect that deals with machine times, personnel times, and costs of assembling each batch of superior parts from a batch of subordinate parts. These two aspects should be handled together in as few passes as possible over the COMPONENT relation and other relations.

Whatever new operators are introduced, they should have relations as both their operands and their results, in compliance with the operational closure feature, Feature RM-5 (see Chapter 12). The attempt by the Oracle Corporation to extend SQL to handle the BoM application is quite inadequate in that the total effort is represented by the CONNECT command. This command not only violates the closure feature, RM-5, but also fails to handle many aspects of the BoM problem, including the integrity aspects. Finally, in early releases the CONNECT command failed to work on views.

28.7 ■ Integrity Checks

Normally, it is necessary to establish the kinds of integrity constraints that are pertinent to the problem at hand. If these integrity constraints cannot be expressed in terms of those types already supported in RM/V2, it becomes necessary to invent extensions of RL and algorithms to support these extensions under the covers of the DBMS.

In the case of the BoM example, RL extensions are needed, along with sample algorithms for supporting these extensions. When a database contains one or more relations, each of which happens to represent an acyclic graph, certain types of integrity checking are needed. Three of these types are discussed here: maintaining the acyclic constraint, checking for isolated subgraphs, and, when applicable, checking hierarchic structure. However, the discussion does not include the RM/V2 extensions or sample algorithms.

28.7.1 Checking for Unintended Cycles

If, as a result of insertions or modifications, a relation that represents an acyclic graph changes in such a way as to reflect a cycle in the graph, this indicates that one part p is a component (not necessarily immediate) of another q, and that q is at the same time a component (not necessarily immediate) of p. This condition is normally deemed unacceptable.

Hence, an overall method of checking the whole relation is needed to see whether it is acyclic. Furthermore, it is desirable to have a more localized method of establishing incrementally that any insertion into or modification of an acyclic relation does not introduce a cycle. In this way, the DBMS can efficiently ensure that a graph that is initially acyclic remains acyclic.

28.7.2 Isolated Subgraphs

Let us take the following definition: a subgraph of a graph G is any collection of edges all of which occur in the graph G. In the relational representation of graph G, each and every subset of that relation represents a subgraph of G.

The two extremes are the largest subgraph (i.e., G itself) and the smallest subgraph (i.e., the empty graph with no edges). If g and G are graphs, then the complement of g in G is the set of edges in G but not in g; it is denoted G - g. Note that G - g is a subgraph of G.

An isolated subgraph of G is any subgraph g of G that has no edges connecting it to the complement of g in G, namely, G - g. Therefore, in this case, the graphs g and G - g have no nodes in common. Clearly, an isolated subgraph may itself contain an isolated subgraph. Hence, one may sensibly speak of decomposing any given graph into a collection of minimal isolated subgraphs, each of which has no isolated subgraph itself. Two kinds of programs are needed: one intended merely to detect whether any subgraph

of G is isolated, and a second that identifies all the isolated subgraphs of G.

Actually, when a graph G has an isolated subgraph g, then G has at least two isolated subgraphs: g and its complement with respect to G. In practice, from a purely logical standpoint, it may not be necessary to separate the isolated subgraphs by casting them into distinct relations. However, such separation may yield a noticeable performance advantage if large amounts of data are involved.

28.7.3 Strict Hierarchy

A *strict hierarchy* is not only free from cycles, but also has the property that each node has exactly one parent, except for the topmost node which has no parent at all. Occasionally, it may be necessary to check whether the strict hierarchic structure has been maintained.

28.8 ■ Computational Aspects

A general requirement is that a user be able to supply some of the arguments, that the database supplies the others, and that the DBMS invokes the appropriate DBA-defined functions. It should be possible to execute this activity by the user, together with the development of function definitions by the DBA, without programming tricks such as loops. This can be achieved within relational operators that scan graphs, whether these graphs are acyclic or not.

The main requirement in the BoM application is to compute the cost, the time, or both for manufacturing one quantum of the kind of part at the terminating node, where a quantum is the smallest number of such parts built in a single run. The result should be a relation that indicates not only the previously cited cost, time, or both, but also the contribution to these amounts from each of the edges that must be traversed in manufacturing the pertinent part at node k. Then, the cost or time for N quanta can be computed by a simple final computation.

Since each edge in the product structure graph is represented by a single row in the COMPONENT relation, it is easy to arrange for appropriate functions and arguments to be available for the edge-based computations by including as columns in this relation the names of pertinent functions, together with the names and/or values of arguments for these functions. This represents an exploitation of Features RF-9 and RF-10 (re-introduced in Section 28.2).

It may also be necessary to exploit node-based functions and their arguments. This can be arranged similarly by including as columns in a PART relation the names of pertinent functions, together with the names and/or values of arguments for these functions. The PART relation has part number P# as its primary key. This column contains at least all of the distinct part numbers that occur in the COMPONENT relation.

28.9 ■ Concluding Remarks

Once again, there has been *no* attempt in this chapter to give a full account of the extensions of the relational model planned for the bill-of-materials application. Instead, the focus was on the general task of making extensions, using the BoM as an example.

To make sensible and acceptable extensions of the relational model, it is necessary to be thoroughly familiar with the model. It is also necessary to make a thorough mathematical investigation of the problem that these extensions are designed to solve. One purpose of such an investigation is to determine whether any extensions are needed at all. Thus, it should be clear that, if a brilliant idea concerning an extension suddenly comes to mind, there remains a substantial piece of work before an extension should be proposed.

Exercises

28.1 What are the 15 questions that should be answered when extending the relational model to handle a particular type of problem?

28.2 Discuss whether conserving the mathematical closure of the relational operators is (1) crucial, (2) fairly important, or (3) of no concern whatsoever.

28.3 It has been claimed that the bill-of-materials problem involves the processing of purely hierarchic data, and that therefore a hierarchic DBMS (such as IBM's IMS) is ideally suited to the problem. Take a position on this and defend it.

28.4 In what specific way does the CONNECT command of the Oracle Corporation fail to satisfy the operational closure feature, Feature RM-5? In what additional respect is the Oracle approach to solving the bill-of-materials problem inadequate?

28.5 Supply two reasons why mathematical tools other than relations should be explored whenever a significant extension of the relational model is being contemplated.

■ CHAPTER 29 ■

Fundamental Laws of Database Management

This chapter could be regarded as summarizing in a new framework what has been presented earlier. Actually, however, it is an attempt to generalize upon the relational approach by stating concisely some 20 principles with which *any* new approach to database management should comply. My development of these principles was motivated by various object-oriented approaches to database management.

What impressed me the most in these proposals was the clear lack of knowlege of the relational model on the part of their authors. In any attempt to invent an approach that is superior to the relational model, the first step must be to learn about that model. Moreover, one should not assume that the relational DBMS products of today or the corresponding manuals fairly represent the model. As a corollary, it is highly unlikely that anyone who has learned to use a relational DBMS solely from the vendor's manual really knows how to use it.

The fundamental laws outlined in this chapter are principles to which the relational model adheres. Any new approach to database management intended to compete with the relational model should adhere to these principles. It appears unlikely that any competitor will seriously challenge the dominant position of the relational model today, because the model is based on first-order predicate logic. Predicate logic took 2,000 years to develop, beginning with the ancient Greeks who discovered that the subject of logic could be intelligently discussed separately from the subject to which it might be applied, a major step in applying levels of abstraction.

This chapter attempts to discourage the outrageous claims that have been made regarding "semantic data models." It is also an attempt to encourage researchers to direct their attention to the overall problem of database management, instead of considering only one small part, such as data structures.

The fundamental laws are as follows:

1. object identification;
2. objects identified in one way;
3. unrelated portions of a database;
4. community issues;
5. three levels of concepts;
6. same logical level of abstraction for all users;
7. self-contained logical level of abstraction;
8. sharp separation;
9. no iterative or recursive loops;
10. parts of the database inter-related by value-comparing;
11. dynamic approach;
12. extent to which data should be typed;
13. creating and dropping performance-oriented structures;
14. adjustments in the content of performance-oriented structures;
15. re-executable commands;
16. prohibition of cursors *within the database;*
17. protection against integrity loss;
18. recovery of integrity;
19. re-distribution of data without damaging application programs;
20. semantic distinctiveness.

Now, let us consider each of these laws in turn.

1. **Object identification**

   A database models a micro-world. Each object about which information is stored in the database must be uniquely identified, and thereby distinguished from every other object. The DBMS must enforce this law.

The unique identifier in the relational model is the combination of the relation name and the primary-key value.

2. **Objects identified in one way**

   Both programming and non-programming users perceive all objects to be identified in exactly one way, whether these objects are abstract or concrete and whether they are so-called entities or relationships.

So far, no one has come forward with definitions for the concepts in the "whether" clauses that are reasonable, objective, precise, non-overlapping, and unambiguous. It is extremely doubtful that the task is worthwhile pursuing. In the relational model, such distinctions are avoided altogether.

3. **Unrelated portions of the database**

   If the database can be theoretically split into two or more mutually unrelated parts without loss of information, whether stored or derived, there exists a simple and general algorithm, independent of access paths, to make this split.

The database-splitting algorithm that is part of the relational model was described in Chapter 3. It is heavily based on the domain concept.

4. **Community issues**

   All database issues of concern to the community of users (except, for the time being, performance goals and advice) should be:

   a. Removed from application programs, if incorporated therein;

   b. openly and explicitly declared in the catalog or in some part of the database to which all suitably authorized users have access; **and**

   c. managed by the DBMS. Such management includes enforcement in the case of integrity constraints and authorization.

Much of the relational model is based on Fundamental Law 4. Such support is clearly visible in the techniques used for the retention of database integrity and in the authorization mechanism.

5. **Three levels of concepts**

   Three levels of concepts must be distinguished: (1) psychological (the user's level), (2) logical and semantic (the logical level), and (3) storage-oriented and access-method (the physical level).

A data model should address the requirements of the logical level first and foremost. Any attempt to define Level 1 or Level 3 must be accompanied by transformations of concepts, structure, data, and actions upon data from Level 2 to and from whatever level is added to Level 2. For a single logical Level 2, there may be many instances of psychological Level 1 and many instances of physical Level 3. Levels 1 and 2 do not represent different levels of abstraction.

The relational model specifies the properties required at the logical level (Level 2), and in such a way as to leave to the DBMS vendors how to treat the physical level (Level 3) and the psychological level (Level 1). This model defines the boundaries between the three levels very sharply; and it may be the only existing approach that does this.

6. **Same logical level of abstraction for all users**

   The level of abstraction supported by the DBMS for end users must be the same as that supported for application programmers.

This law runs counter to the practices of the past, when end users were offered query products that were defined separately and packaged separately from the DBMS. The designers of these query products tried to offer end users a higher level of abstraction than the DBMS offered to application programmers.

7. **Self-contained logical level of abstraction**

   The logical level must be sufficiently complete that there is no need to proceed to a lower level of abstraction to explain how a command at the logical level works.

An example of an unsuccessful departure from this law was a DBMS product that required those users with the responsibility of defining views to know how the information was represented at a lower level of abstraction. In effect, the product limited the defining of views to the DBA staff. While these people should have the specialized skills for this job, they would rapidly become overloaded with work that users should be able to do for themselves.

8. **Sharp separation**

   In the services offered by the DBMS, there must be a sharp separation between aspects of Type 1 (the logical and semantic aspects), and those of Type 2 (the storage-representation and performance aspects, including access methods).

It is this sharp separation that makes the relational model a standard that vendors can live with, while not restricting their freedom and inventiveness to design competitive products. This separation is also of great value to users since it protects their investment in training and in the development of application programs.

9. **No iterative or recursive loops**

   In order to extract any information whatsoever in the database, neither an application programmer nor a non-programming user needs to develop any iterative or recursive loops.

This law significantly reduces the occurrences of bugs in programs, and sharply improves the productivity of application programmers and end users. Great care has been taken to uphold this law in developing the relational model, more than in any other approach.

10. **Parts of the database inter-related by value comparing**

    All inter-relating is achieved by means of comparisons of values, whether these values identify objects in the real world or indicate properties of those objects. A pair of values may be *meaningfully compared* if and only if these values are of the same extended data type. Inter-relating

parts of the database is *not* achieved by means of pointers visible to users.

It is safe to assume that all kinds of users understand the act of comparing values, but that relatively few understand the complexities of pointers. The relational model is based on this fundamental principle. Note also that the manipulation of pointers is more bug-prone than is the act of comparing values, even if the user happens to understand the complexities of pointers.

11. **Dynamic approach**

    Performance-oriented structures can be created and dropped dynamically, which means without bringing traffic on the database to a halt. Automatic locking by the DBMS permits such activities without damage to the information content of the database and without impairing any transactions.

With pre-relational DBMS products, there was a significant dependence on utilities—programs that changed the performance-oriented structures and access methods, but that could be executed in the off-line mode only. In other words, execution of any one of these utilities required that the database traffic be brought to a complete halt. In contrast, the relational approach, which is highly dynamic, can support the kind of concurrency architecture designed to cope with non-stop traffic on the database and provide various degrees of fault tolerance.

12. **Extent to which data should be typed**

    The types of data seen by users should be strict enough to capture some of the meaning of the data, but not so strict as to make the initially planned uses and applications the only viable ones.

When a new database is created or when new kinds of information are incorporated into an already existing database, the creator is almost always unable to foresee all the uses to which the new kinds of data will be applied. While suited to the development of programs, the object-oriented approach probably imposes too many restrictions on the use of data through its typing of data.

13. **Creating and dropping performance-oriented structures**

    Performance-oriented structures, such as indexes, must be capable of being created and dropped by either the DBA or the DBMS without adversely affecting the semantic information, all of which should be in the database or in the catalog.

At some time in the future, probably early in the next century, even the DBA will be eliminated from this law as an acceptable agent for creating and dropping performance-oriented structures. DBMS vendors will have invented ways in which performance-oriented structures can be automatically

adjusted to support changes in the database traffic, changes that last for a reasonable time.

14. **Adjustments in the content of performance-oriented structures**

    When data is inserted into a database, updated, or deleted from a database by a user, it must not be necessary for that user or any other user to make corresponding changes in the information content of performance-oriented structures. It is the responsibility of the DBMS to make these adjustments dynamically.

Most existing index-based relational DBMS products comply with this law by adjusting the content of their indexes automatically whenever an **insert, update,** or **delete** occurs on the data in the database.

15. **Re-executable commands**

    Every command at the logical level of abstraction must be re- executable and yield exactly the same results, provided the data operated upon by the command remains unchanged in information content (i.e., unchanged at the logical level). This means that results must be independent of the data organization and access methods that are in effect at lower levels of abstraction.

Most of the relational operators comply with this law. None of them is affected in its results by the data organization and access methods that are in effect at the time of execution.

16. **Prohibition of cursors within the database**

    At the logical and psychological levels, users are not required to manipulate cursors that traverse data *within the database.* However, cursors that traverse data extracted from the database are acceptable as a means of providing an interface to single-record-at-a-time host languages.

Cursors within the database, a nightmare in the CODASYL approach to database management, are the source of severe bugs that are very hard to track down. In the debate in 1974 [Codd 1974], I used an example developed by members of the CODASYL DBTG committee that was intended by them to show off their approach. I demonstrated how the manipulative activity in this example could be reduced from eight pages of DBTG and COBOL code to just one statement in a relational language. It is interesting to note that, five years later, someone discovered that there were two bugs in the CODASYL program that were directly related to cursor manipulation.

Cursors that traverse data extracted from a relational database are much easier to manage correctly. Such cursors are supported in several DBMS products to enable single-record-at-a-time host languages such as FORTRAN, COBOL, and PL/1 to interface with relational DBMS. However, these cursors are troublesome in the management of distributed databases. The file is a

more promising package for a relational DBMS to use in delivering data to programs written in these languages.

17. **Protection against integrity loss**

    With the help of the DBA, the DBMS must provide strong protection against loss of data integrity.

18. **Recovery of integrity**

    The support provided for *correction* of any integrity loss actually experienced must include an audit log that is readily transformable into a database of the same kind as that handled by the pertinent approach.

Laws 17 and 18 reflect the fact that it is easier to prevent loss of database integrity than to correct such loss. Even with an audit log, trying to correct loss of integrity is often a painful task. Without such a log, it is an impossible task in most cases. Much of the emphasis in the relational model is on prevention of such loss.

19. **Re-distribution of data without damaging application programs**

    In a DBMS capable of managing distributed data, it should be possible to re-distribute the data in a highly flexible manner without affecting the logical correctness of any application programs.

The relational model appears to be the only approach known today that is capable of supporting this law. For details, see Chapters 24 and 25.

20. **Semantic distinctiveness**

    Semantically distinct observations, whether derived or not, must be represented distinctly to the users. In any database, all of the data redundancy *that is made visible to the users* must be both introducible and removable by those users who are authorized to do so, without affecting the logical correctness of any application programs and the training of interactive users.

In all base and derived relations of the relational model, duplicate rows are prohibited. Note that data may superficially appear to be redundant, but still not be redundant. The crucial question is: If the data were removed, would information be lost?

By now, the reader should be in a good position to counter the frequently held opinion that the relational model is nothing more than its structures, and that these structures are merely tables. How could the single concept of tables comply with all 20 of the principles just outlined? A thorough reading of the preceding 28 chapters should enable the reader to determine exactly how the relational model adheres to each and every one of these laws.

Exercises

29.1 Taking each of the 20 fundamental laws in turn, list the features of RM/V2 that ensure compliance with the law.

29.2 Choose any approach to database management other than the relational model. Repeat the same 20 exercises for the approach selected. Then compare this approach with the relational model.

■ *CHAPTER 30* ■

Claimed Alternatives to the Relational Model

After the publication of my papers on Version 1 of the relational model in the period 1969–1973, numerous articles began to appear proposing new approaches to database management. Frequently, the articles began with the false claim that the relational model contains no features for representing the meaning of the data.

Often these new approaches were no more than new kinds of data structure or data typing—often new to the database management field only. In other words, the authors overlooked the need to specify query and manipulative operators, integrity constraints, authorization, commands for the DBA, a counterpart to the catalog, distributed database management, user-defined data types, and user-defined functions. That is why I call each of them an "approach," and avoid using the term "data model."

Occasionally, mistakes of the past are revisited, apparently by authors who have no knowledge of the DBMS products of the past. Examples of such mistakes are repeating groups and representing information in many distinct ways. These mistakes add complexity but not generality.

In this chapter, only five kinds of approaches are discussed:

1. the universal relation approach (UR);
2. the binary relation approach (BR);
3. the entity-relationship approaches (ER);
4. the semantic data approaches (SD);
5. the object-oriented approaches (OO).

The main objective of this chapter is to improve understanding of the relational model, and especially to indicate why the relational model is the way it is. A lesser objective is to review these approaches as replacements for or alternatives to the relational model. The purpose is not to dismiss all the ideas contained in these approaches. Each one, except ER, contains some good ideas, some of which are quite eligible to be attached to the relational model.

My comments about UR and BR tend to be quite precise because these approaches are precisely defined. On the other hand, my comments about ER, SD, and OO tend to be quite imprecise because at present these approaches are imprecisely defined with respect to database management.

30.1 ■ The Universal Relation and Binary Relations

These two approaches to database design and management represent opposite extremes. UR takes all the relations in a regular relational database and glues them together by means of one operator (e.g., **natural join** based on primary and foreign keys) to form a single relation of very high degree that is claimed to contain all the information in the given database. BR splits every relation into a collection of binary relations (i.e., relations of degree two). Thus, the universal relation can be regarded as a "macro" approach; the collection of binary relations, as a "micro" approach.

Both approaches are examined with the principal aim of shedding more light on why the relational model is based on a middle-of-the-road approach—namely, a collection of relations of assorted degrees. This means that any base or derived relation of the relational model can be of *any* degree n, where n is a strictly positive integer ($n > 0$).

In contrast to the relational model, UR requires just one relation of a very large degree (the degree must be large enough to accommodate all of the information in the database). BR requires relations, each of which is of degree one or two only.

As is apparent from the referenced papers (and perhaps this chapter), it is questionable whether either UR or BR is really comprehensive enough in tackling the total problem of database management to be treated as a data model. In the case of UR, it is also questionable whether the approach deserves the label "universal."

In what follows, I point out claims that are clearly false. In doing so, my aim is not to discourage university researchers from pursuing their lines of investigation, but rather to clarify these approaches and their relationship to the relational model.

30.2 ■ Why the Universal Relation Will Not Replace the Relational Model

The "universal relation" is just one of the very many views supported by the relational model. For detailed information, see [Maier, Ullman, and

Vardi 1984]. The assertion has been made [Vardi 1988]—and the very title of [Maier, Ullman, and Vardi 1984] makes this same claim—that the universal relation can replace the entire relational model. This assertion is quite preposterous. I now present eight solid reasons for stating this.

In [Vardi 1988], the author complains about the need for users to "navigate" through the logical parts of a relational database, and proposes the Stanford University "universal relation" as a means of protecting users from this burden. The "universal relation" fails completely to provide an alternative to the relational model. I am not arguing, however, that Stanford University should never have undertaken research into the "universal relation"; there may be some useful by-products of this research. Nevertheless, I do think that the term "universal" is a complete misnomer—the reason why it is enclosed in quotation marks in this chapter.

In [Codd 1971d], I proved that collectively the algebraic operators of the relational model are as powerful as first-order predicate logic in retrieving information from a relational database. Indeed, if the several relations in a relational database are transformed into a single relation, the resulting relation, together with operators that can interrogate a single relation only, is not as powerful as the relational model. In the following subsections, eight reasons are presented for asserting that UR will not replace the relational model.

30.2.1 The Operators

First, let us look at an ordinary relational database containing several relations, and consider how it might be transformed into a single "universal relation." Vardi and others suggest the use of either **natural join** or **equi-join** (it does not matter which is chosen) as the "connecting function." This function is presumably key-based—in other words, it joins by comparing a primary key with a foreign key. Immediately, we are struck with the notion that there are 10 distinct kinds of **theta-joins** based on the following 10 comparators:

1. EQUAL TO
2. NOT EQUAL TO
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 constructing the "universal relation" using **equi-join,** what happened to the nine other kinds of **theta-joins?** Moreover, what happened to **relational division,** the algebraic counterpart of the universal quantifier?

30.2.2 Joins Based on Keys

The construction of a "universal relation" from a given relational database involves the repeated application of either **natural join** or **equi-join** based on the keys of the relational model. The "universal relation" is based on the false assumption that two classes of entities (e.g., suppliers and types of parts) have only one relationship between them. This assumption is proved false by citing just one counter-example: each supplier can be related to each type of part by its capability of supplying that part; each supplier can also be related to each type of part by its possibly several, actual deliveries of that part in response to a sequence of orders for the part.

30.2.3 Joins Based on Non-keys

If only key-based joins are used in constructing the "universal relation," then it is possible that hundreds of **joins** that are not based on keys have been overlooked. An example would be the query: "Find the employees who reside in a city in which the company owns one or more warehouses." Assume that this query is applied to a frequently encountered database in which city is not a primary key because of the company's lack of interest in cities as objects whose properties need to be recorded. In the relational model, this query then involves an **equi-join** based on a non-key. I fail to see how a user would make this request against a "universal relation" without applying **equi-join** as an operator to two parts of the allegedly "universal relation."

Vardi claims the "universal relation" reduces the burden on users of choosing which operators to apply, and choosing the relations and attributes to which these operators should be applied. This claim may be true when only one operator is involved and it happens to match the one used in constructing the "universal relation." However, when there may be several operators involved in a single query and at least one of them does not match the construction operator, the user is faced with significantly more complexity than with the relational model.

30.2.4 Cyclic Key States

It is not at all clear how the "universal relation" copes with what are often called *cyclic key states.* Suppose that a relation R1 has a primary key PK1, while a relation R2 has a foreign key FK1 drawing its values from the same domain as PK1. Suppose also that the primary key of R2 is PK2, and that R1 contains a foreign key FK2 drawing its values from the same domain as

PK2. Then, R1 and R2 participate in a cyclic key state, in which the cycle is of size two.

An example may clarify this situation. Suppose that we are using the relational model and we have the following two relations:

EMP (E# . . . DEPT# . . .),

with primary key E#, and

DEPT (DEPT# . . . MGR# . . .),

with primary key DEPT#. EMP identifies and describes employees, E# is the employee serial number, DEPT# is the department identifier, DEPT identifies and describes departments, and MGR# is the employee serial number of the department manager.

Suppose that (1) DEPT# in EMP is a foreign key with respect to the primary key DEPT# of the DEPT relation, and that (2) MGR# in DEPT is a foreign key with respect to the primary key E# of the EMP relation. Then, these two relations have a two-step cyclic key state.

Clearly, such cycles can be of size greater than two. Cycles are not only possible, they occur rather frequently. Any solution to this problem in the context of the "universal relation," which requires data to be repeated in different parts of such a relation, is unacceptable as a confusing and unnecessary form of redundancy.

30.2.5 Insertion, Deletion, and Updating

As a vehicle for inserting information, deleting, and updating, the "universal relation" is replete with problems. It is a relation that is not even in third normal form, let alone fifth. One can therefore expect to encounter update anomalies galore [Codd 1971b]. If the connecting function used in constructing the "universal relation" were other than a key-based **join,** there would be the serious possibility that it could not be updated at all.

30.2.6 Coping with Change

The ability of a data model to cope with change must be taken into account. Nothing is as certain as change in requirements as time goes on. A particular relation in the relational model may become obsolete through lack of use or for other reasons (perhaps it is going to be replaced by two or more new relations with different descriptions).

The relation may then be simply dropped, and all remaining users of that relation warned of the drop. Unless that relation happens to be a boundary member of the "universal relation"—which is improbable—the data will have to be extracted from some non-boundary position of the "universal relation" and it will be necessary to reconstruct this relation

completely. A similar remark applies to the counterpart in the "universal relation" of a newly introduced relation in the relational model. In either case, does this reorganization impair the application programs?

30.2.7 No Comprehensive Data Model

No data model has been published for the UR approach. To be comprehensive, such a data model must support all of the well-known requirements of database management. Until this occurs, companies intending to acquire a DBMS product should be concerned about the risk of investing in the "universal relational" approach.

As an aside, Vardi's use of the term "access path" is a complete departure from the usual use of this term. Usually, it can be assumed that, when two or more alternative access paths can be used to extract certain data from a database, the only difference between those access paths is performance. There is no semantic distinction between the paths. In a relational DBMS, it is the optimizer that selects access paths with the objective of good performance. Vardi's "access paths" are quite different because distinct paths yield distinct results (all of the paths to which he refers are semantically distinct).

30.2.8 UR Not Essential for Natural Language

The allegation in [Vardi 1988, page 85] that a "universal relation" is essential as a natural-language interface is quite incorrect. Curiously, advocates of the binary relational approach make the same claim. Certainly, at least one of the claims must be incorrect. It is my belief that both are incorrect.

During the period 1974–1977, I led the development of a prototype translator from English to a relational language, and from the relational language back into precise English. This two-way translator was accompanied by a third component that supported clarification dialogue. Incidentally, I believe that it is very risky to use a natural-language package with any kind of database unless the package supports (1) clarification dialogue, and (2) before database access, a routine check by the system that it understands the user's request by telling the user in the same natural language precisely its interpretation of the user's request.

All three components were based on the relational model and on predicate logic. The prototype, called Rendezvous [Codd 1978], was successfully tested in 1977 against 30 subjects with wide variations in their knowledge of computers and of English. Some subjects tried as hard as they could to beat the system, but failed. This evidence suggests that the natural-language claims of both the UR and the BR approaches are false.

30.2.9 Concluding Remarks Regarding UR

In conclusion, I would not rule out the "universal relation" as one of the many views that should be supported in a relational DBMS, but I consider it incapable of replacing the relational model. I also believe that for many purposes it is too complicated as a relational view, and it is not likely to be popular even in that restricted role. The question remains: What does the 'universal relation' accomplish that simpler views in the relational model do not?

30.3 ■ Why the Binary Relation Approach Will Not Replace the Relational Model

In Chapter 1, a false claim found in many mathematical textbooks was briefly discussed—namely, the assertion that every problem expressed in terms of relations of degree higher than two can be reduced to an equivalent problem in which the relations are of degree either one or two. This false idea has appealed to several people doing research in database management. Why not perceive and manipulate the information in the database as a collection of unary and binary relations?

I believe that this approach was first proposed in the IBM Hursley Laboratory in England; a prototype [Titman 1974] was built there about 1973. The approach recently re-surfaced at the University of Maryland [Mark 1988]. One attractive feature is that it is easy to get a prototype into operational state because relatively few operators must be implemented, and these few are quite simple to implement.

Users are faced with serious problems, however, if the approach is applied to the kind of large-scale databases encountered in the commercial and industrial world. If the binary relations are perceived as tables, they are two-column tables.

In what follows, nine reasons are discussed for the assertion that the binary-relation approach cannot replace the relational model. For this purpose, it is useful to have two examples, each representing just a portion of a database.

Consider a simple example: if an insurance-policy relation in the relational model has a single-column primary key (usually the policy number), together with 100 columns each containing a simple immediate property of the policy, then in the binary relationship approach there will be 100 tables each with two columns. Each table carries the policy number to identify the policy, together with just one simple immediate property.

Now consider a more complicated example: in a suppliers and parts database, suppose that there is a capability relation indicating in each row that a specified supplier can supply a specified quantum of a specified kind of part within a specified time at a specified cost. Note that the primary key

of this relation is composite. It consists of the combination of supplier serial number and part serial number.

To represent this *n*-ary relation in terms of binary relations, the simple approach adopted in the insurance-policy case (i.e., repetition of the primary key along with each of the simple immediate properties) is no longer viable because each of those relations would be ternary (i.e., of degree three). One solution, perhaps the simplest, is to introduce an extra (artificial) single-column primary key in the *n*-ary version, and then convert that relation into a collection of binary relations, just as in the insurance case. Unfortunately the users will have to know about this artificial primary key in order to manipulate these binary relations properly.

30.3.1 Normalization Cannot Be Forgotten

In [Mark 1988], the author claims,

> "The model [he is referring to the BR approach] seems to be easy for non-technical people because it avoids normalization and because schemata defined in terms of the model can be read almost like natural language."

In database design, normalization can certainly be relegated to an analytical or checking stage, but it cannot be avoided altogether if surprising anomalies in insertions, updates, and deletions are to be avoided when use of the database begins. **Join** dependencies, difficult to discover even in the regular relational model, are even more difficult to discover in BR.

Corresponding to any given *n*-ary relational version of a database, there is clearly a binary relational counterpart. Consider two *n*-ary versions of a common conceptual database, one thoroughly normalized (say B1), and the other incompletely normalized (say B2). Each of these databases has a binary relational counterpart (say b1 and b2, respectively).

Unfortunately, in the binary relational form it is extremely difficult to see that b2 is effectively incompletely normalized, and that under certain conditions insert, update, and delete anomalies will occur. Thus, the claim that "normalization of *n*-ary relations can be forgotten" [Mark 1988] is false. Moreover, it is harder to cope with this aspect if the database designer is constrained to think in terms of binary relations only.

30.3.2 Much Decomposition upon Input

When information about an insurance policy is entered into the insurance database, a good proportion is entered in a single operation. Otherwise, questions such as, "To whom does this policy belong?," "Does the policy-holder satisfy our minimum requirements for this kind of policy?," and

"How frequently does the policyholder have to be billed?" could arise and be unanswerable.

If users are to perceive this policy information as part of a collection of binary relations, it is likely that the DBMS, either once upon entry or else many times, whenever the information is manipulated, must decompose the policy information into small pieces to fit into the large collection of binary relations. Even the policyholder's home address and work address each must be decomposed into at least six separate items: (1) apartment or suite number within a building, (2) building number within a street, (3) street name within a city, (4) city name within a state, (5) name of state, and (6) zip code. Presumably this task of decomposition is a burden on the DBMS, not the user. However, no matter where the burden falls such decomposition is completely unnecessary.

An important part of the difficulties encountered with the BR approach is the fact that composite domains, composite columns, composite primary keys, and composite foreign keys all must be abandoned (see Section 30.3.5). From the user's point of view, the units of information that are atomic with respect to a DBMS based on the BR approach are frequently too small to support concise and clear thinking.

30.3.3 Extra Storage Space and Channel Time

DBMS prototypes of the BR type tend to store the data in the form of a two-column table for each binary relation. The net result is that the storage space consumed for a BR database is about double what would be required for a regular relational database.

The extra bits have to be transmitted to and from the processing units across channels. Therefore, the channel load will be significantly greater than that experienced if the structure and operators of the relational model were used.

30.3.4 Much Recomposition upon Output

For obvious reasons, business and government reports are rarely, if ever, presented as a collection of two-column tables. Thus, the development of reports and replies to queries by any DBMS based on the binary relational approach entails putting together many items that are perceived by users as separately tabulated items residing in tables that have just two columns. This recomposition entails either many key-based **natural joins** of carefully selected binary relations, or else an extended version of **natural join** that collects all the desired properties into a single n-ary relation [Codd 1979].

Considering together the reasons in Sections 30.3.2 and 30.3.3, one wonders what the purpose of decomposition was, if it is followed later by as many recompositions as there are reports to be generated and queries for which replies are needed.

30.3.5 Composite Domains, Composite Columns, and Composite Keys Abandoned

The concepts of composite domains, columns, and keys fit quite naturally into the relational model. These concepts appear to have been abandoned in the binary relational approach. Any attempt to fit them into this approach is bound to result in bending the approach and in unnecessary complexity. The example of a person's home address or work address cited earlier illustrates the need for composite domains and columns. The example of the capability relation cited earlier also illustrates the need for composite domains and composite primary keys, and therefore for composite foreign keys.

The absence of these concepts from the binary relational approach means that users must manipulate information in the database in terms of pieces that are smaller than the user's customary perception. Thus, it should be expected that user productivity will decrease.

30.3.6 The Heavy Load of Joins

The binary relational approach places an unnecessarily heavy load of joining on the DBMS. In the regular relational model, database designers occasionally must de-normalize parts of the database to reduce the time spent in executing **joins** and, in this way, obtain good performance. The binary relational approach is likely to cause an order-of-magnitude increase in the execution of **joins** over that required by the relational model.

30.3.7 Joins Restricted to Entity-based Joins

In [Mark 1988], the author says that in the binary relation approach, "Relationships between object types are derived through entity-joins rather than symbol-joins." This is equivalent to taking the regular relational model and permitting key-based **joins** only, since keys denote objects, while non-keys denote properties of objects. All **joins** based on non-keys would be prohibited. This is a very severe restriction that would be hard to justify.

Consider the query: Find the employees who live in a combination of city and state in which the company has a warehouse. Given Mark's ground rules, this query could not be handled by a binary relational DBMS, unless every city and state combination were already treated as an object (having its own immediate properties), and consequently this combination would constitute the primary key in some relation.

When designing a database, it is quite likely that the designer will choose to treat the city and state combination as a combination of properties of other kinds of objects. Normally, this combination of city and state will be treated as an object only if the company's business heavily depends upon data maintained about cities (e.g., population, size of market, crime rate, and distribution of wealth).

30.3.8 Integrity Constraints Harder to Conceive and Express

In the relational model, each case of referential integrity frequently involves two distinct relations (although more than two can be involved). Many user-defined integrity constraints can also be expected to involve two or more distinct relations. In the binary relational approach, each of these integrity constraints is likely to involve even more distinct binary relations, and therefore be harder to conceive, more cumbersome to express, and entail more overhead for the DBMS, thus reducing its performance.

As an example, consider any referential integrity constraint that involves a composite key in the relational model. As a second example, consider a user-defined integrity constraint that requires the company's salespeople to be based in city and state combinations for which the market is at least 10% of the company's total market in the immediately preceding year.

30.3.9 No Comprehensive Data Model

So far, the BR approach lacks a solid foundation. No data model for it has been published that supports all the well-known requirements of database management. Until this occurs, companies intending to acquire a DBMS product should be concerned about the risk of investing in the binary relational approach.

30.4 ■ The Entity-Relationship Approaches

Numerous approaches based on splitting objects into two types, entities and relationships, have been proposed. Many of the authors of these approaches identify P.P. Chen as their source of inspiration [Chen 1976], even though he was by no means the first to propose such a split. In fact, this kind of split was an inherent part of the thinking that went into all single-record-at-a-time, pre-relational approaches to database management. Of the five approaches discussed in this chapter, this one is clearly the winner in terms of its lack of precise definitions, lack of a clear level of abstraction, and lack of a mental discipline. The popularity of ER may lie in its multitude of interpretations, as well as its use of familiar but obsolete modes of thought.

The major problem in the entity-relationship approach is that one person's entity is another person's relationship. There is no general and precisely defined distinction between these two concepts, even when discussion is limited to a particular part of a business that is to be modeled by means of a database. If there are 10 people in a room and each is asked for definitions of the terms "entity" and "relationship," 20 different definitions are likely to be supplied for each term.

A good example is an airline flight. An accountant is likely to think of this as an entity. Someone responsible for airplane scheduling or crew

scheduling is likely to think of it as a relationship between a type of aircraft, a flight route, a crew, and a date.

A second problem with this approach is that a relationship between objects is not supposed to have immediate properties that are recorded in the database. It should be very obvious that relationships can have any number of immediate properties. Consider as an example a database containing information about parts and suppliers. Two quite distinct relationships between parts and suppliers are as follows:

1. the CAPABILITY relation, in which each assertion states that a particular supplier *can supply* a particular kind of part;
2. the SHIP relation, in which each assertion states that a particular supplier *has supplied* a particular kind of part to the pertinent company.

Each of these relations is likely to have a distinct collection of numerous immediate properties. For example, CAPABILITY may have estimated speed of delivery, the number of units supplied as a non-divisible package, and the cost of each such package. SHIP may have date of shipment, quantity of parts shipped, and an identifier for the destination warehouse.

Even though a relationship may begin life with no immediate properties, it is extremely unwise to establish the database design and the development of application programs on the assumption that it will stay that way forever.

If it is proposed to handle the manipulation of entities and relationships by means of distinct commands, then the vocabulary for retrievals, insertions, updates, and deletions is doubled over the vocabulary in the relational model. If no manipulative distinctions are made between entities and relationships, why are they conceived as two different kinds of information? Is this just one more example of a distinction that leads to an increase in complexity, but no increase whatsoever in generality?

No data model has yet been published for the entity-relationship approach. To be comprehensive, it must support all of the well-known requirements of database management. Until this occurs, companies intending to acquire a DBMS product should be concerned about the risk of investing in the entity-relationship approach.

30.5 ■ The Semantic Data Approaches

The claim that an approach is semantic is a very strong claim indeed, strong enough to be considered extravagant. One test that I believe should be made to check such a claim is as follows. Imagine that the computer system is equipped with the five human senses: touch, smell, vision, hearing, and taste. If such a system were also equipped with a DBMS based on some approach that is claimed to be semantic, together with a database concerning suppliers, parts, warehouses, projects, and employees, could this system use its five senses and the database to distinguish these objects from one another in its environment, and recognize the type of each object?

While there have been numerous approaches to database management claimed to be semantic, the one discussed here is that of Hammer and McLeod [1981]. This particular case is chosen, because one United States vendor, Unisys, claims that its product INFOEXEC is based on it, and that "it will become the preferred approach to database management in the 1990s" [Balfour 1988]. The major difficulties encountered by any approach that is claimed to be exclusively semantic are two-fold:

1. there is no known, totally objective boundary to the world of semantics;
2. there is no known way to replace predicate logic by semantic machinery.

Of course, either or both of these states of affairs could change in the future, but in neither case is it likely to be an overnight change. Until such a change occurs, however, both of these states represent sound reasons for all of us to avoid claims that a semantic approach can replace the relational model.

In [Hammer and McLeod 1981, page 353], the authors make the mistake of characterizing the relational model as "record-oriented." They proceed to declare that "it is necessary to break with the tradition of record-based modeling, and to base a database model on structural constructs that are highly user-oriented and expressive of the application environment." This completely overlooks all the integrity constraints of the relational model, as well as their definition by linguistic means independently of application programs. The linguistic approach to defining these aspects of the meaning of data is much more powerful than any known structural approach. It represents a strong step forward from the old hierarchic and network-structured approaches.

In [Balfour 1988], the author lists several properties of the relational model that he alleges are "fundamental weaknesses." I find his supporting case for the allegations to be quite shallow and completely unconvincing. He interposes some criticisms of current SQL and current implementations of the relational model. I agree with his criticism of SQL, but I believe his assertion that "current implementations of the relational model have not performed as well as the older database technologies in high-volume on-line transaction-oriented environments" is not only false [Codd 1987a], but also irrelevant to his defense of semantic data approaches.

No data model has yet been published for the semantic approach. To be comprehensive, it must support all of the well-known requirements of database management. Until this occurs, companies intending to acquire a DBMS product should be concerned about the risk of investing in the semantic approach.

30.6 ■ The Object-oriented Approaches

There are several different approaches in the object-oriented category, and no vendor has yet announced a database management product based on one

of them. The ideas in this kind of approach stem from the need in programming languages for more thoroughly defined and more abstract data types. In particular, the concept known as *abstract data type* is the key to most of the research in this area. Such ideas are clearly a step forward in the area of programming languages, but it is not at all clear that they represent a step forward in the technology for database management.

As time progresses, a database is bound to change in terms of the types of information stored in it. In fact, there is likely to be a significant expansion in the kinds of information stored in any database. Such expansion should not *require* changes to be made in application programs. Some kinds of information may be dropped. When any drop occurs, there should be a simple way to detect (1) all the application programs that are adversely affected, and (2) all the commands within each such program that are adversely affected.

If each of these requirements is met, the approach can be claimed with some credibility to be *adaptable to change*. When applied to database management, the object-oriented approaches take a very restrictive and non-adaptable approach to the interpretation and treatment of data.

It is important to ask whether any object-oriented language exists that is as high in level as the relational languages. Without a language that conveys the user's intent at a high level of abstraction, how can the system optimize the sequence of minor operations and the choice of access paths when executing a request? The answer to this question is particularly crucial when distributed database mangement is involved.

Can the OO approach to database management support distribution independence? In other words, can application programs remain unchanged and correct when a database is converted from centralized to distributed, and later when the data must be re-distributed? What support does the OO approach provide for built-in and user-defined integrity constraints that are not embedded in the application programs?

No comprehensive data model has yet been published for the object-oriented approach. To be comprehensive, it must support all of the well-known requirements of database management. Until this occurs, companies intending to acquire a DBMS product should be concerned about the risk of investing in the object-oriented approach.

30.7 ■ Concluding Remarks

Both the SD and the OO approaches emphasize the need for DBMS products to support generalization or type hierarchies. I agree that such support is necessary, and showed with RM/T [Codd 1979] how type hierarchies could be supported without making the data structure concepts more complicated. This is a feature of RM/T that is very likely to drop down into RM/V3 within the next decade.

The five approaches to database management examined in this chapter are just five of many that are now competing for a place in the sun.

Development of the relational model has made researchers aware of the impact a data model can have on the field of data processing. Each new model that comes along must be carefully examined from the standpoint of its technical merit, usability, and comprehensiveness. New theoretical contributions to the field should also be examined carefully, not glibly cast aside as just theory, and therefore not practical—a judgment made many times in the last 20 years about the relational model.

Exercises

30.1 **Joins** and **relational division** are occasionally criticized for requiring users to engage in "logical navigation." This presumably means "finding their way through a logical data model." Discuss the claimed alternatives to logical navigation, and your position regarding how complete each alternative is and its technical pros and cons.

30.2 Does the "universal relation" provide a means of protecting users from the multi-relation operators of the relational model? Defend your position on this issue.

30.3 When insertions, deletions, and updates are applied to the "universal relation," what problems are encountered?

30.4 When a new relation is created in a relational database, what is the counterpart activity in a "universal relation?" Create an example such that the addition cannot be attached to the outer boundaries of the "universal relation."

30.5 Consider this query: find the suppliers, and 10 immediate properties of these suppliers, each of whom can supply every part listed by part serial number in a given unary relation. Assume that the database contains information about suppliers including their supplier serial numbers and 50 other immediate properties. Assume also that the database contains capability information about suppliers and parts, together with 20 immediate properties that apply to each combination of supplier and part. Outline a binary relational database, and express the query in terms of **joins, relational division, projects,** and so forth (but only as these operators apply to binary and unary relations). Do not assume that any relation of degree greater than two can be generated as a derived relation, except as the final "report-presentation" step.

30.6 Take an example of a relation of degree five in which there exists a **join** dependency that is not a multi-valued or functional dependency. Cast this relation into a collection of binary relations that is equivalent in information content. Comment on whether the **join** dependency is easier to detect in this form or in the original form.

30.7 A bank has branches in several cities, and each city has its own DBMS storing the customer accounts. Assume that the accounts are

all the same in the kinds of identifying properties and other properties stored for each. Thus, they are union-compatible. Suppose that the DBMS in each city is part of the bank's overall control of its distributed database. If each DBMS is based solely on the binary relation approach and each customer account has 20 distinct properties recorded, how would you obtain at bank headquarters the **union** of all customer accounts?

30.8 Supply six reasons why the "universal relation" will not replace the relational model.

30.9 Supply eight reasons why the binary relation approach will not replace the relational model.

30.10 How prevalent is the *property-not-applicable* mark in a "universal relation?" Take a simple example involving 50 suppliers with 20 distinct properties, 1,000 parts with 40 distinct properties, and 2,000 capabilities with 15 distinct properties. Assume the usual condition that none of the properties of an object of any one type is applicable to objects of the other two types, where the three types are suppliers, parts, and capabilities. In this database, how many items of data are marked *property inapplicable?*

■ *APPENDIX A* ■

RM/V2 Feature Index

A.1 ■ Index to the Features

This index of the 333 features of Version 2 of the relational model described in this book is intended to make it easy to find each feature of RM/V2.

The features are labeled with consecutive numbers within each of 18 classes. The following table indicates the letters denoting each class and the chapter(s) each class falls in:

Chapter	Class	
18	A	Authorization
4	B	Basic operators
15	C	Catalog
21	D	Principles of DBMS design
3,7	E	Commands for the DBA
19	F	Functions
13,14	I	Integrity
11	J	Indicators
22	L	Principles of language design
12	M	Manipulation
6	N	Naming
20	P	Protection
10	Q	Qualifiers
2	S	Structure

Chapter	Class	
3	T	Data types
16,17	V	Views
24,25	X	Distributed database management
5,17	Z	Advanced operators

In the "priority" column, the letter F or B appears (F denotes fundamental, and hence top priority, while B denotes basic). The phrase "multiple rows" includes zero and one row as special cases that are not given special treatment.

Note that DBMS products can be classified by the features they support. In the early 1990s, a DBMS product that fully supports all the features of both Type F and Type B deserves to be called advanced.

Structure-Oriented and Data-Oriented Features (Chapter 2)

Feature Label	Priority	Feature Title	Page
RS-1	F	The information feature	30
RS-2	F	Freedom from positional concepts	32
RS-3	F	Duplicate rows prohibited in every relation	32
RS-4	F	Information portability	33
RS-5	B	Three-level architecture	34
RS-6	F	Declaration of domains as extended data types	34
RS-7	F	Column descriptions	35
RS-8	F	Primary key for each base R-table	35
RS-9	B	Primary key for certain views	36
RS-10	F	Foreign key	36
RS-11	B	Composite domains	37
RS-12	B	Composite columns	37
RS-13	F	Missing information: representation	39
RS-14	B	Avoiding the universal relation	40

Domains as Extended Data Types (Chapter 3)

Feature Label	Priority	Feature Title	Page
RT-1	F	Safety feature when comparing database values	46
RT-2	F	Extended data types built into the system	49
RT-3	F	User-defined extended data types	50

Feature Label	Priority	Feature Title	Page
RT-4	B	Calendar dates	50
RT-5	B	Clock times	52
RT-6	B	Coupling of dates with times	53
RT-7	B	Time-zone conversion	53
RT-8	B	Non-negative decimal currency	53
RT-9	B	Free decimal currency	54

The ten comparators in theta-select and theta-join are as follows:

1	EQUAL TO	=
2	NOT EQUAL TO	≠
3	LESS THAN	<
4	LESS THAN OR EQUAL TO	<=
5	GREATER THAN	>
6	GREATER THAN OR EQUAL TO	>=
7	GREATEST LESS THAN	G<
8	GREATEST LESS THAN OR EQUAL TO	G<=
9	LEAST GREATER THAN	L>
10	LEAST GREATER THAN OR EQUAL TO	L>=

The Basic Operators (Chapter 4)

Feature Label	Priority	Feature Title	Page
RB-1	F	De-emphasis of Cartesian product as an operator	66
RB-2	F	The **project** operator	67
RB-3	F	**Theta select** using =	69
RB-4	F	**Theta select** using ≠	69
RB-5	F	**Theta select** using <	69
RB-6	F	**Theta select** using <=	69
RB-7	F	**Theta select** using >	69
RB-8	F	**Theta select** using >=	69
RB-9	F	**Theta select** using G<	69
RB-10	F	**Theta select** using G<=	69
RB-11	F	**Theta select** using L>	69
RB-12	F	**Theta select** using L<=	69

Feature Label	Priority	Feature Title	Page
RB-13	B	The Boolean extension of **theta-select**	72
RB-14	F	**Theta join** using =	73
RB-15	F	**Theta join** using ≠	73
RB-16	F	**Theta join** using <	73
RB-17	F	**Theta join** using <=	73
RB-18	F	**Theta join** using >	73
RB-19	F	**Theta join** using >=	73
RB-20	F	**Theta join** using G<	73
RB-21	F	**Theta join** using G<=	73
RB-22	F	**Theta join** using L>	73
RB-23	F	**Theta join** using L<=	73
RB-24	B	The Boolean extension of **theta-join**	76
RB-25	F	The **natural join** operator	77
RB-26	F	The **union** operator.	78
RB-27	F	The **intersection** operator	81
RB-28	F	The **difference** operator	82
RB-29	F	The **relational division** operator	83
RB-30	F	**Relational assignment**	87
RB-31	F	The **insert** operator	88
RB-32	F	The **update** operator	89
RB-33	B	**Primary-key update** with cascaded update of foreign keys and optional update of sibling primary keys	90
RB-34	B	**Primary-key update** with cascaded marking of foreign keys	92
RB-35	F	The **delete** operator	92
RB-36	B	The **delete** operator with cascaded deletion	93
RB-37	B	The **delete** operator with cascaded A-marking and optional sibling deletion	94

The Advanced Operators (Chapters 5 and 17)

Feature Label	Priority	Feature Title	Page
RZ-1	F	Framing a relation	98
RZ-2	F	Extend the description of one relation to include all the columns of another relation	103
RZ-3	B	**Semi-theta join** using =	105
RZ-4	B	**Semi-theta join** using ≠	105
RZ-5	B	**Semi-theta join** using <	105

Feature Label	Priority	Feature Title	Page
RZ-6	B	**Semi-theta join** using <=	105
RZ-7	B	**Semi-theta join** using >	105
RZ-8	B	**Semi-theta join** using >=	105
RZ-9	B	**Semi-theta join** using G<	105
RZ-10	B	**Semi-theta join** using G<=	105
RZ-11	B	**Semi-theta join** using L>	105
RZ-12	B	**Semi-theta join** using L<=	105
RZ-13	B	**Left outer equi-join**	107
RZ-14	B	**Right outer equi-join**	108
RZ-15	B	**Symmetric outer equi-join**	108
RZ-16	B	**Left outer natural join**	114
RZ-17	B	**Right outer natural join**	114
RZ-18	B	**Symmetric outer natural join**	114
RZ-19	B	**Outer union**	117
RZ-20	B	**Outer set difference**	119
RZ-21	B	**Outer set intersection**	120
RZ-22	B	**Inner T-join** using <	125
RZ-23	B	**Inner T-join** using <=	125
RZ-24	B	**Inner T-join** using >	125
RZ-25	B	**Inner T-join** using >=	125
RZ-26	B	**Left outer T-join** using <	134
RZ-27	B	**Left outer T-join** using <=	134
RZ-28	B	**Left outer T-join** using >	134
RZ-29	B	**Left outer T-join** using >=	134
RZ-30	B	**Right outer T-join** using <	134
RZ-31	B	**Right outer T-join** using <=	134
RZ-32	B	**Right outer T-join** using >	134
RZ-33	B	**Right outer T-join** using >=	134
RZ-34	B	**Symmetric outer T-join** using <	134
RZ-35	B	**Symmetric outer T-join** using <=	134
RZ-36	B	**Symmetric outer T-join** using >	134
RZ-37	B	**Symmetric outer T-join** using >=	134
RZ-38	B	**User-defined select**	137
RZ-39	B	**User-defined join**	138
RZ-40	B	**Recursive join**	140
RZ-41	B	**Semi-insert** operator	320
RZ-42	B	**Semi-update** operator	321
RZ-43	B	**Semi-archive** operator	321
RZ-44	B	**Semi-delete** operator	321

Naming (Chapter 6)

Commands for the DBA (Chapters 3 and 7)

Missing Information (Chapters 8 and 9)

Qualifiers (Chapter 10)

Feature Label	Priority	Feature Title	Page
RQ-3	F	The MAYBE qualifier	210
RQ-4	B	Temporary replacement of missing database values (applicable)	210
RQ-5	B	Temporary replacement of missing database values (inapplicable)	210
RQ-6	B	Temporary replacement of empty relation(s)	211
RQ-7	F	The ORDER BY qualifier	211
RQ-8	B	The ONCE ONLY qualifier	214
RQ-9	B	The DOMAIN CHECK OVERRIDE (DCO) qualifier	215
RQ-10	B	The EXCLUDE SIBLINGS qualifier	216
RQ-11	F	The appended DEGREE OF DUPLICATION (DOD) qualifier	216
RQ-12	B	The SAVE qualifier	218
RQ-13	B	The VALUE qualifier	218

Indicators (Chapter 11)

Feature Label	Priority	Feature Title	Page
RJ-1	B	Empty-relation indicator (result indicator)	223
RJ-2	B	Empty-divisor indicator (argument indicator)	223
RJ-3	B	Missing-information indicator (result indicator)	223
RJ-4	B	Non-existing argument indicator (argument indicator)	224
RJ-5	B	Domain-not-declared indicator (argument indicator)	224
RJ-6	B	Domain-check-error indicator (argument indicator)	224
RJ-7	B	Domain not droppable, column still exists indicator (argument indicator)	224
RJ-8	B	Duplicate-row indicator (argument indicator)	225
RJ-9	B	Duplicate-primary-key indicator (argument indicator)	225
RJ-10	B	Non-redundant ordering indicator (result indicator)	225
RJ-11	B	Catalog block indicator (result indicator)	226
RJ-12	B	View not tuple-insertible	226
RJ-13	B	View not component-updatable	226
RJ-14	B	View not tuple-deletable	227

Data Manipulation (Chapter 12)

Integrity Constraints (Chapter 13)

Feature Label	Priority	Feature Title	Page
RI-8	B	Determining applicability of constraints	248
RI-9	B	Retention of constraint definitions for Types R and U	248
RI-10	B	Activation of constraint testing	249
RI-11	F	Violations of integrity constraints of Types D, C, and E	249
RI-12	F	User-defined prohibition of missing database values	250
RI-13	B	User-defined prohibition of duplicate values	251
RI-14	B	Illegal tuple	251
RI-15	B	Audit log	252
RI-16	F	Non-subversion	252
RI-17	B	Creating and dropping an integrity constraint	253
RI-18	B	New integrity constraints checked	253
RI-19	F	Introducing a column integrity constraint (Type C) for disallowing missing database values	253
RI-20	B	Minimal adequate scope of checking	254
RI-21	B	Each integrity constraint executable as a command	255
RI-22	B	On-the-fly, end of command, and end of transaction timing	255

User-Defined Integrity Constraints (Chapter 14)

Feature Label	Priority	Feature Title	Page
RI-23	B	Information in a user-defined integrity constraint	260
RI-24	B	Triggering based on AP and TU actions	260
RI-25	B	Triggering based on date and time	261
RI-26	B	Insertion involving I-marked values	267
RI-27	B	Update involving I-marked values	268
RI-28	B	Functional dependency constraint	272
RI-29	B	Multi-valued dependency constraint	272
RI-30	B	**Join** dependency constraint	273
RI-31	B	Inclusion dependency constraint	273
RI-32	F	The REJECT command	274
RI-33	F	The CASCADE command	274
RI-34	F	The MARK command	274

The Catalog (Chapter 15)

Views (Chapters 16 and 17)

Authorization (Chapter 18)

Feature Label	Priority	Feature Title	Page
RA-7	F	Authorizable database-control activities	331
RA-8	F	Authorizable query and manipulative activities	332
RA-9	B	Authorizable qualifiers	333
RA-10	F	Granting and revoking authorization	333
RA-11	B	Passing on authority to grant	334
RA-12	B	Cascading revocation	334
RA-13	B	Date and time conditions	334
RA-14	B	Resource consumption (anticipated or actual)	335
RA-15	B	Choice of terminal	335
RA-16	B	Assigning authorization	335

Functions (Chapter 19)

Feature Label	Priority	Feature Title	Page
RF-1	F	Built-in aggregate functions	338
RF-2	F	The DOD versions of built-in statistical functions	340
RF-3	B	Built-in scalar functions	340
RF-4	B	User-defined functions: their use	341
RF-5	B	Inverse function required, if it exists	341
RF-6	B	User-defined functions: compiled form required	341
RF-7	B	Functions can access the database	342
RF-8	F	Non-generation of marked values by functions	342
RF-9	B	Domains and columns containing names of functions	343
RF-10	B	Domains and columns containing names of arguments	344

Protection of Investment (Chapter 20)

Feature Label	Priority	Feature Title	Page
RP-1	F	Physical data independence	345
RP-2	B	Logical data independence	346
RP-3	B	Integrity independence	347
RP-4	B	Distribution independence	347
RP-5	B	Distributed database management: decomposition and recomposition	349

DBMS Design (Chapter 21)

Language Design (Chapter 22)

Feature Label	Priority	Feature Title	Page
RL-12	B	Canonical form for every request	366
RL-13	B	Global optimization	366
RL-14	B	Uniform optimization	367
RL-15	B	Constants, variables, and functions interchangeable	367
RL-16	B	Expressing time-oriented conditions	367
RL-17	F	Flexible role for operators	368

Distributed Database Management (Chapters 24 and 25)

Feature Label	Priority	Feature Title	Page
RX-1	B	Multi-site action from a single relational command	393
RX-2	B	Local autonomy	394
RX-3	B	Global database and global catalog	396
RX-4	B	N copies of global catalog ($N > 1$)	397
RX-5	B	Synonym relation in each local catalog	398
RX-6	B	Unique names for sites	399
RX-7	B	Naming objects in a distributed database	399
RX-8	B	Reversibility and redistribution	401
RX-9	B	Decomposition by columns for distributing data	403
RX-10	B	Decomposition by rows for distributing data	404
RX-11	B	General transformation for distributing data	404
RX-12	B	Replicas and snapshots	405
RX-13	B	Integrity constraints that straddle two or more sites	406
RX-14	B	Views that straddle two or more sites	407
RX-15	B	Authorization that straddles two or more sites	408
RX-16	B	Name resolution with a distributed catalog	409
RX-17	B	Inter-site move of a relation	410
RX-18	B	Inter-site moves of rows of a relation	411
RX-19	B	Dropping a relation from a site	412
RX-20	B	Creating a new relation	412
RX-21	B	Abandoning an old site and perhaps its data	413
RX-22	B	Introducing a new site	414

Feature Label	Priority	Feature Title	Page
RX-23	B	Deactivating and reactivating items in the catalog	415
RX-24	B	Minimum standard for statistics	421
RX-25	B	Minimum standard for the optimizer	422
RX-26	B	Performance independence in distributed database management	422
RX-27	B	Concurrency independence in distributed database management	423
RX-28	B	Recovery at multiple sites	423
RX-29	B	Locking in distributed database management	423

A.2 ■ Summary of RM/V2 Features by Class

Table of features fundamental and basic

Feature Class	Fundamental (F)	Basic (B)	Totals		
RS	9	5	14		
RT	3	6	9		
RB	31	6	37	104	
RZ	2	42	44		
RN	7	7	14		221
RE	8	14	22		
RQ	3	10	13	63	
RJ	0	14	14		
RM	14	6	20		
RI	11	23	34	54	
RC	8	3	11		
RV	8	0	8	35	
RA	6	10	16		
RF	3	7	10		
RP	1	4	5		112
RD	8	8	16	77	
RL	8	9	17		
RX	0	29	29		
Totals	130	203	333	333	333

A.3 ■ Classes of Features and Numbers of Features in Each Class

Class	Number of Features			
Z	44			
B	37	115		
I	34			
X	29		186	
E	22	71		
M	20			
D	16			
L	17	77		333
A	16			
S	14			
N	14		147	
J	14			
Q	13	38		
C	11			
F	10			
T	9			
V	8	32		
P	5			

A.4 ■ Principal Objects and Properties in RM/V1

Structure

Domains
Relations (same as R-tables)
Attributes (same as columns)
Primary keys
Foreign keys
Information portability

Data Types

See Features RT-1–RT-6.

Operators

Each operator can be expressed in at most one command, and without circumlocution or circumconception.

Theta-select (restrict)

Project

Theta-join

(Where **theta** is any of the comparators 1–6 for theta-select, and any of the comparators 1–10 for **theta-join**.)

Union

Set difference

Intersection (inner types)

Left, right, and **symmetric outer join**

Relational division

Relational assignment

Qualifiers

Qualifiers include the temporary replacement of missing elementary database values, as well as the MAYBE qualifiers.

Indicators

Empty relation

Empty divisor in relational division

Manipulative Features

See Features RM-1–RM-5.

Integrity Constraints

Each integrity constraint can be expressed in one command per constraint, and without circumlocution or circumconception. Each such constraint is stored in the catalog, not in an application program.

D Domain integrity

C Column integrity

E Entity integrity

R Referential integrity
U User-defined integrity

Catalog

See Features RC-1–RC-4.

Views

See Features RV-1–RV-5.

Authorization

See Features RA-1–RA-4.

A.5 ■ Functions

See Features RF-1–RF-3.

A.6 ■ Investment Protection

See Features RP-1 and RP-2.

DBMS Design

See Features RD-1, RD-2, RD-4, RD-6, RD-11, and RD-13.

Language Design

See Features RL-1–RL-6.

A.7 ■ The Rules Index

The following table shows the features of RM/V2 that correspond to the twelve rules published in 1985 [Codd 1985]. The text of any selected rule can be found by first using this table to determine which feature(s) of RM/V2 corresponds to the selected rule, and then by using the index to the features at the beginning of this Appendix to find the text in the book.

1985 Rule	RM/V2 Feature	Name
1	RS-1	Information rule
2	RM-1	Guaranteed access

1985 Rule	RM/V2 Feature	Name
3	RS-13, RM-10	Missing information
4	RC-1	Active catalog
5	RM-3	Comprehensive data sublanguage (DSL)
6	RV-4, RV-5	View updatability
7	RM-4	High level language
8	RP-1	Physical data independence
9	RP-2	Logical data independence
10	RP-3	Integrity independence
11	RP-4	Distribution independence
12	RI-16	Non-subversion

■ APPENDIX B ■

Exercises in Logic and the Theory of Relations

The following exercises are included in this book to help the reader test his or her own knowledge in these two branches of mathematics. Although these topics are not covered in this book, knowledge of them is important for the designers of DBMS products and for database administrators.

B.1 ■ Simple Exercises in Predicate Logic

In the following exercises, P, Q are predicates, x is an individual variable, c is a constant, $(\exists)$ denotes the existential quantifier, $(\forall)$ denotes the universal quantifier, and $\rightarrow$ denotes logical implication.

Which of the following pairs of formulas in predicate logic are logically equivalent?

1a.	$\forall x(Px \vee Qx)$	1b.	$(\forall xPx \vee \forall xQx)$
2a.	$\forall x(Px \wedge Qx)$	2b.	$(\forall xPx \wedge \forall xQx)$
3a.	$\neg (Pc \vee Qc)$	3b.	$(\neg Pc) \wedge (\neg Qc)$
4a.	$Pc \rightarrow Qc$	4b.	$(\neg Pc) \vee Qc$
5a.	$\neg \forall xPx$	5b.	$\exists x(\neg Px)$
6a.	$\forall x(Px \rightarrow Qx)$	6b.	$(\exists xPx \vee \forall xQx)$
7a.	$\exists x(Px \rightarrow Qx)$	7b.	$\exists x(\neg Px) \vee \exists xQx$
8a.	$\neg \exists xPx$	8b.	$\neg \forall x\, Px$

B.2 ■ Simple Exercises in Relational Theory

In the following exercises, R, S, and T are union-compatible relations. The three operators are **relational union** (denoted ∪), **relational difference** (denoted −), and **relational intersection** (denoted ∩).

Which of the following pairs of expressions are guaranteed to yield identical results?

1a.	R ∪ S	1b.	S ∪ R
2a.	R − S	2b.	S − R
3a.	R ∩ S	3b.	S ∩ R
4a.	(R ∪ S) ∩ T	4b.	(R ∩ T) ∪ (S ∩ T)
5a.	(R ∪ S) − T	5b.	(R − T) ∪ (S − T)
6a.	(R ∩ S) − T	6b.	(R − T) ∩ (S − T)
7a.	(R ∩ S) ∪ T	7b.	(R ∪ T) ∩ (S ∪ T)
8a.	(R − S) ∩ T	8b.	(R ∩ T) − (S ∩ T)

B.3 ■ Exercises Concerning the Inter-relatedness of RM/V2 Features

No claim has been made that all 333 features of RM/V2 are independent of one another; such a claim would be false. This kind of independence would make the relational model, whether Version 1 or Version 2, difficult to understand. It would be similar to trying to learn an IBM 3090 by first learning a universal Turing machine.

Suppose that F and G are two features of RM/V2. The notation F → G means that the DBMS must support F if it is to provide full support for G.

1. Let F be view updatability and G be logical data independence. Show that F → G.
2. Let F be domains as extended data types and G be view updatability. Show that F → G.
3. Let F be primary keys and G be view updatability. Show that F → G.
4. Take each distinct pair of features F and G in RM/V2. Examine whether F → G or G → F, or neither. Explain your answers.

■ REFERENCES ■

This list of references is not claimed to be complete. The list, however, does include most of the works by the author of this book that concern database management (Codd 1969–Codd 1988b). (Codd 1968) illustrates levels of abstraction, but applied to a different subject.

The reference list includes the following database management papers by other authors: (Beeri, Fagin, and Howard 1977), (Bernstein and Chiu 1981), (Bernstein, Hadzilacos, and Goodman 1987), (Buff 1986), (Casanova, Fagin, and Papadimitriou 1984), (Chen 1976), (Date 1984 and 1987), (Dayal and Bernstein 1982), (Ganski and Wong 1987), (Goodman 1988), (Hammer and McLeod 1981), (Heath 1971), (Keller 1986), (Kim 1982), (Klug 1982), (Lindsay 1981), (Lipski 1978), (Maier, Ullman, and Vardi 1984), (Mark 1988), (Sweet 1988), (Vardi 1988), (Vassiliou 1979), and (Wiorkowski and Kull 1988).

Several texts dealing with predicate logic also appear in the reference list. In descending order of difficulty, they are as follows: (Church 1958), (Suppes 1967), (Exner and Rosskopf 1959), (Stoll 1961), and (Pospesel 1976).

The remaining entries in the reference list consist of database management system prototypes and products. (IMS n.d.) was a leading DBMS product in the late 1960s and early 1970s. Note that the Peterlee relational test vehicle (Todd 1976), listed as operational at the end of 1972, preceded all other systems based on the relational model. These entries are far from complete because of the recent development of many relational DBMS products.

Balfour, A. (1988) *INFOEXEC: The First Practical Implementation of a Semantic Database,* Unisys, Europe-Africa Division. December.

Beech, D. (1989) "The Need for Duplicate Rows in Tables." *Datamation,* January.

Beeri, C., R. Fagin, and J. H. Howard (1977) "A Complete Axiomatization for Functional and Multi-Valued Dependencies in Database Rela-

tions." In *Proc. ACM SIGMOD 1977 Int. Conf. on Management of Data* (Los Angeles, 1977).

Bernstein, P. A., and D. W. Chiu (1981) "Using Semi-joins to Solve Relational Queries." *JACM* 28:1.

Bernstein, P. A., V. Hadzilacos, and N. Goodman (1987) *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

Buff, H. W. (1986) "The View Update Problem Is Undecidable." Personal communication, Zurich, August 4. Published as "Why Codd's Rule No. 6 Must Be Reformulated." *ACM SIGMOD Record* 17:4 (1988).

Casanova, N., R. Fagin, and C. Papadimitriou (1984) "Inclusion Dependencies and Their Interaction with Functional Dependencies." *JCSS* 28:1.

Chamberlin, D. D., Astrahan, M. M., Blasgen, M. W., Gray, J. N., King, W. F., Lindsay, B. G., Lorie, R., Mehl, J. W., Price, T. G., Putzolu, F., Selinger, P. G., Schkolnick, M., Slutz, D. R., Traiger, I. L., Wade, B. W., Yost, R. A. (1981) "A History and Evaluation of System R." *Comm. ACM* 24:10.

Chen, P. P. (1976) "The Entity-Relationship Model—Toward a Unified View of Data, *ACM TODS* 1:1.

Church, A. (1958) *Introduction to Mathematical Logic.* Princeton University Press.

Codd, E. F. (1968) *Cellular Automata.* Academic Press.

Codd, E. F. (1969) *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks.* San Jose, IBM Research Report RJ599.

Codd, E. F. (1970) "A Relational Model of Data for Large Shared Data Banks." *Comm. ACM* 13:6.

Codd, E. F. (1971a) "ALPHA: A Data Base Sublanguage Founded on the Relational Calculus." In *Proc. 1971 ACM SIGFIDET Workshop* (San Diego, Nov. 11–12, 1971).

Codd, E. F. (1971b) "Further Normalization of the Data Base Relational Model." In *Courant Computer Science Symposia 6, Data Base Systems.* (New York, May 24–25). Prentice-Hall.

Codd, E. F. (1971c) "Normalized Data Base Structure: A Brief Tutorial." *Proc. 1971 ACM SIGFIDET Workshop* (San Diego, Nov. 11–12, 1971).

Codd, E. F. (1971d) Relational Completeness of Data Base Sublanguages." In *Courant Computer Science Symposia 6, Data Base Systems* (New York, May 24–25). Prentice-Hall.

Codd, E. F. (1974a) "Interactive Support for Non-Programmers: The Relational and Network Approaches" and "Data Models: Data-Structure-Set versus Relational." *Proc. 1974 ACM SIGMOD Debate* (Ann Arbor, May 1–3).

Codd, E. F. (1974b) "Recent Investigations in Relational Data Base Systems." *Proc. IFIP* (Stockholm, August). Also published in *Information Processing 1974.* North-Holland.

Codd, E. F. (1974c) "The Relational Approach to Data Base Management: An Overview." Third Annual Texas Conference on Computing Systems (Austin, November 7–8).

Codd, E. F. (1974d) "Seven Steps to Rendezvous with the Casual User." *Proc. IFIP TC2 Working Conf. on Data Base Management Systems* (Cargese, Corsica, April 1–5).

Codd, E. F. (1978) "How about Recently?" *Proc. Int. Conf. on Databases: Improving Usability and Responsiveness.* (Haifa, Israel, August 2–3). Academic Press.

Codd, E. F. (1979) "Extending the Database Relational Model to Capture More Meaning." *ACM Trans. on Database Systems* 4:4.

Codd, E. F. (1980) "Data Models in Database Management." ACM SIGMOD, SIGPLAN, SIGART meeting (Pingree Park, Colorado, November 1980).

Codd, E. F. (1981) "The Capabilities of Relational Database Management Systems." Proc. Convencio Informatica Llatina (Barcelona, June 9–12).

Codd, E. F. (1982) "Relational Database: A Practical Foundation for Productivity." *Comm. ACM* 25:2.

Codd, E. F. (1985) "How Relational Is Your Database Management System?" *Computerworld,* October 14 and 21. Also available from The Relational Institute, San Jose.

Codd, E. F. (1986a) "Missing Information (Applicable and Inapplicable) in Relational Databases." *ACM SIGMOD Record,* Vol. 15, no. 4.

Codd, E. F. (1986b) *The Twelve Rules for Relational DBMS.* San Jose, The Relational Institute, Technical Report EFC-6.

Codd, E. F. (1987a) "The Beginning of a New Era in Data Processing" (review of Tandem's NonStop SQL). *InfoWeek,* May 4.

Codd, E. F. (1987b) *Fundamental Laws in Database Management.* San Jose, The Relational Institute, Technical Report EFC-27.

Codd, E. F. (1987c) "More Commentary on Missing Information in Relational Databases." *ACM SIGMOD Record,* Vol. 16, no. 1.

Codd, E. F. (1987d) *Principles of Design of Database Management Systems.* San Jose, The Relational Institute, Technical Report EFC-18.

Codd, E. F. (1987e) *View Updatability in Relational Databases: Algorithm VU-1.* Unpublished paper.

Codd, E. F. (1988a) "Domains, Primary Keys, Foreign Keys, and Referential Integrity." *Info DB,* May.

Codd, E. F. (1988b) "Fatal Flaws in SQL (Both the IBM and the ANSI Versions)." *Datamation,* August and September. Also available from The Relational Institute, San Jose (Technical Report EFC-28).

Date, C. J. (1984) "Why Is It So Difficult to Provide a Relational Interface to IMS?" *Info IMS* 4:4.

Date, C. J. (1987) "Where SQL Falls Short," (abridged). *Datamation,* May 1. Unabridged version, *What Is Wrong with SQL,* available from The Relational Institute, San Jose.

Dayal, U., and P. A. Bernstein (1982). "On the Correct Translation of Update Operations on Relational Views." *ACM TODS* 7:3.

Exner, R. M., and M. F. Rosskopf (1959) *Logic in Elementary Mathematics.* McGraw-Hill.

Ganski, R. A., and H. K. T. Wong (1987) "Optimization of Nested SQL Queries Revisited." *Proc. ACM SIGMOD Annual Conference* (San Francisco), May 27–29.

Goodman, N. (1988) Letter regarding the outer set operators, July 15.

Hammer, M., and D. McLeod (1981) "Database Description with SDM: A Semantic Database Model." *ACM TODS* 6:3.

Heath, I. J. (1971) Memorandum introducing the outer join. Fall.

Hutt, A. T. F. (1979) *A Relational Data Base Management System.* Wiley.

IBM (n.d.) *IMS Reference Manual.* IBM Corporation, White Plains, N.Y. (any post-1971 edition).

IBM (1988) *SQL Reference Manual.* IBM Corporation.

Keller, A. M. (1986) "Choosing a View Update Translator by Dialog at View Definition Time." 12th Conference on Very Large Databases (Tokyo, August 25–28, 1986).

Kim, Won (1982) "On Optimizing an SQL-like Nested Query." *ACM TODS* 7:3.

Klug, A. (1982) "Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions." *JACM* 29:3.

Lindsay, B. (1981) "Object Naming and Catalog Management for a Distributed Database Manager." *Proc. Second Int. Conf. on Distributed Computing* (Paris, April 1981). Also IBM Research Report RJ2914, 1980.

Lipski, W. (1978) *On Semantic Issues Connected with Incomplete Information Data Bases.* Institute of Computer Science, Polish Academy of Sciences, Warsaw.

Maier, D., J. D. Ullman, and M. Y. Vardi (1984) "On the Foundations of the Universal Relational Model." *ACM TODS,* Vol. 9, no. 2.

Mark, L. (n.d.) *The Binary Relationship Model.* Institute of Advanced Computer Studies, University of Maryland (probably 1988).

Meldman, M. J., D. J. McLeod, R. J. Pellicore, and M. Squire (1978) *RISS: A Relational DBMS for Minicomputers.* Van Nostrand Reinhold.

Ozkarahan, E. A., S. A. Schuster, and K. C. Smith (1975) "RAP: An Associative Processor for Data Base Management." *Proc. AFIPS,* Vol. 44.

Pospesel, H. (1976) *Predicate Logic.* Prentice-Hall.

Relational Technology (1988) *INGRES/QUEL Reference Manual.* Alameda, California, Relational Technology Inc.

Stoll, R. R. (1961) *Sets, Logic, and Axiomatic Theories.* Freeman.

Stonebraker, M., ed. (1986) *The INGRES Papers.* Addison-Wesley.

Suppes, P. (1967) *Introduction to Logic.* Van Nostrand.

Sweet, F. (1984) "What, if Anything, Is a Relational Database?" *Datamation (USA)* Vol. 30, no. 11.

Titman, P. J. (1974) "An Experimental Data Base System Using Binary Relations." *Proc. IFIP TC-2 Working Conf. on Data Base Management Systems,* eds. Klimbie J. W., and Koffeman, K. I., North-Holland.

Todd, S. J. P. (1976) "The Peterlee Relational Test Vehicle: A System Overview." *IBM Systems Journal* 15:4.

Vardi, M. Y. (1988) "The Universal-Relational Data Model for Logical Independence." *IEEE Software,* March.

Vassiliou, Y. (1979) "Null Values in Database Management: A Denotational Semantics Approach," *Proc. ACM SIGMOD 1979 Int. Conf. on Management of Data* (Boston, May 30–June 1).

Williams, R., Daniels, D., Haas, L., Lapis, G., Lindsay, B., Ng, P., Obermarck, R., Sclinger, P., Walker, A., Wilms, P., and Yost, R. (1981) *R*: An Overview of the Architecture.* IBM Research Report RJ3325.

Wiorkowski, G., and D. Kull (1988) *DB2 Design and Development Guide.* Addison-Wesley.

Zloof, M. M. (1975) "Query by Example." *Proc. of AFIPS,* Vol. 44.

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Dr. Codd received his M.A. in Mathematics from Oxford University and his M.S. and Ph.D. in Communication Sciences from the University of Michigan. He is an elected member of the National Academy of Engineering, a Fellow of the British Computer Society, as well as a long-time member of the Association of Computing Machinery, Phi Beta Kappa, and Sigma Xi. In 1976, Dr. Codd became an IBM Fellow, and in 1981 he received the ACM Turing Award.

DATABASE

The RELATIONAL MODEL for DATABASE MANAGEMENT: VERSION 2

E. F. Codd

This book, written by the originator of the relational model, chronicles its development and presents the current state-of-the-art, Version 2. It defines the features that must be implemented in order for database management systems (DBMS) to be described as truly relational, and gives the motivation behind them. The book includes more than 300 features of the model organized into 18 classes, together with clear, practical explanations of each feature.

The relational model, invented by Codd in 1969, was the first model for database management. This book collects, together in one complete document, much of what has appeared in technical papers since its invention. While the new, expanded Version 2 of the relational model is highlighted, the book covers both Versions 1 and 2. Some of the important new features of Version 2 include:

- Support for all kinds of integrity constraints, especially the user-defined type.
- Support for the management of distributed databases.
- A new treatment of data that is missing because certain objects have properties that are inapplicable.
- A more detailed account of view updatability, which has been neglected by DBMS vendors.
- Some newer principles of design applied to DBMS products and relational languages.
- Some of the fundamental laws on which the relational model is based.
- A more detailed account of what should be in the catalog.

This book is essential reading for anyone with an interest in relational database management systems, including data processing specialists, DBMS developers, database administrators, and DBMS users.

ADDISON-WESLEY PUBLISHING COMPANY