CHAPTER 9

Codd's Rules for Relational DBMSs

Abstract

This chapter examines each of Codd's rules for a "fully relational" database design. The discussion presents practical reasons for adhering to the rules. It also looks at why, in some cases, the rules have to be extended to include developments in database technology since the rules were first published.

Keywords

E.F. Codd
Codd's rules
rules for relational databases
relational database design
relational database design rules

In October 1985, E. F. Codd published a series of two articles in the computer industry weekly called *Computerworld*. The first article laid out 12 (or 13, depending on how you count them) criteria to which a "fully relational" database should adhere. The second article compared current mainframe products to those rules, producing a flurry of controversy over whether it was important the DBMSs be theoretically rigorous, or that they simply work effectively.

Note: If you read Appendix A, then you will be aware of a product based on the simple network data model called IDMS/R. When Codd rated IDMS/R—which was then being marketed as a relational DBMS—it met none (0) of the 12 rules. DB/2, IBM's flagship relational product, met 10 of the rules.

To help you understand the issues raised and why Codd's rules for relational databases for the most part make sense, in this chapter we will look at those criteria along with the implications of their implementation. Should you then choose not to adhere to one or more of the rules, you will be doing so with full understanding of the consequences. In some cases, the consequences are minimal; in others, they may significantly affect the integrity of the data in a database.

Rule 0: The Foundation Rule

Before getting into specifics, Codd wanted to ensure that a relational DBMS was truly using the relational data model and relational actions (for example, relational algebra) to process data:

"A relational database management system must manage its stored data using only its relational capabilities."

This basic rule excludes products that have non-relational engines underlying tools and/or query languages that allow users to function as if the product were relational. Products such as IDMS/R, mentioned in the note above, were eliminated at this point.

Rule 1: The Information Rule

The first criterion for databases deals with the data structures that are used to store data and represent data relationships:

"All information in a relational database is represented explicitly at the logical level in exactly one way—by values in tables."

The purpose of this rule is to require that relations (two-dimensional tables) be the *only* data structure used in a relational database. Therefore, products that require hard-coded links between tables are not relational.

At the time Codd's article was published, one of the most widely used mainframe products was IDMS/R, a version of IDMS that placed a relational-style query language on top of a simple network database. The simple network data model requires data structures, such as pointers or indexes, to represent data relationships. Therefore, IDBMS/R, although being marketed as relational, was not relational according to the very first rule of a relational database. It was this product that was at the heart of the "who cares about rules if my product works" controversy.

Regardless of which side you take in this particular argument, there are several very good reasons why creating a database from nothing but tables is a good idea:

- Logical relationships are very flexible. In a simple network or hierarchical database, the only relationships that can be used for retrieval are those that have been predetermined by the database designer who wrote the schema. However, because a relational database represents its relationships through matching data values, the join operation can be used to implement relationships on the fly, even those that a database designer may not have anticipated.
- Relational database schemas are very flexible. You can add, modify, and remove individual relations without disturbing the rest of the schema. In fact, as long as you are not changing the structure of tables currently being used, you can modify the schema of a live database. However, to modify the schema of a simple network or hierarchical database, you must stop all processing of data and regenerate the entire schema. In

modify the schema of a simple network or hierarchical database, you must stop all processing of data and regenerate the entire schema. In many cases, modifying the database design also means recreating all the physical files (using a dump and load process) to correspond to the new design.

- The meaning of tabular data is well understood by most people. For example, you usually don't need specialized training to teach people that all the data in a row apply to the same item.

Note: DBMSs that require you to specify "relationships between files" when you design a database fail this first rule. If you read Appendix A, then you know that a number of PC-only products work in this way and that although they are marketed as relational, they really use the simple network data model. Keep in mind that the ER diagrams for simple networks and 3NF relational database are identical. The differences come in how the relationships between the entities are represented. In a simple network, it is with hard-coded relationships; in a relational database, it is with primary key–foreign key pairs.

When Codd originally wrote his rules, databases couldn't store anything other than text and numbers. Today, many DBMSs store images, audio, and video in a variety of formats or store the path names (or URL) to media in external files. Technically, path names or URLs of external files are pointers to something other than tables and therefore would seem to cause a DBMS to violate this rule. However, the spirit of the rule is that relationships between entities—the logical relationships in the database—are represented by matching data values, without the use of pointers of any kind to indicate entity connections.

Note: This is not the only rule that needs to be stretched a bit to accommodate graphics in a database environment. See also rule 5 later in this chapter.

Rule 2: The Guaranteed Access Rule

Given that the entire reason we put data into a database is to get the data out again, we must be certain that we can retrieve every single piece of data:

"Each and every datum (atomic value) in a relational database is guaranteed to be logically accessible by resorting to a combination of table name, primary key value, and column name."

This rule states that you should need to know only three things to locate a specific piece of data: the name of the table, the name of the column, and the primary key of the row containing the data.

Note: With today's DBMSs, the definition of a table name can mean many things. For example, if you are working with IBM's DB/2, a table name is the table creator's loginName.tableName. If you are working with Oracle, then a complete table name may include a catalog name, schema name, and Oracle owner name, as well as the name of the individual table.

There is no rule in this set of 12 rules that specifically states that each row in a relation must have a unique primary key. However, a relation cannot adhere to the guaranteed access rule unless it does have unique primary keys. Without unique primary keys, you will be able to retrieve *some* row with the primary key value used in a search, but not necessarily the exact row you want. Some data may therefore be inaccessible without the ability to uniquely identify rows.

Early relational DBMSs did not require primary keys at all. You could create and use tables without primary key constraints. Today, however, SQL will allow you to create a table without a primary key specification, but most DBMSs will not permit you to enter data into that table.

Note: A DBMS that requires "relationships between files" cannot adhere to this rule because you must specify the file in which data reside to locate data.

Rule 3: Systematic Treatment of Null Values

As you know, null is a special database value that means "unknown." Its presence in a database brings special problems during data retrieval. Consider, for example, what happens if you have an *employee* relation that contains a column for salary. Assume that the salary is null for some portion of the rows. What, then, should happen if someone queries the table for all people who make more than $60,000? Should the rows with null be retrieved or should they be left out?

When the DBMS evaluates a null against the logical criterion of salary value greater than 60,000, it cannot determine whether the row containing the null meets the criteria. Maybe it does; maybe it doesn't. For this reason, we say that relational databases use *three-valued logic*. The result of the evaluation of a logical expression is either true, false, or maybe.

Codd's third rule therefore deals with the issue of nulls:

"Null values (distinct from the empty character string or a string of blank characters or any other number) are supported in the fully relational DBMS for representing missing information in a systematic way, independent of data type."

First, a relational DBMS must store the same value for null in all columns and rows where the user does not explicitly enter data values. The value used for null must be the same, regardless of the data type of the column. Note that null is not the same as a space character or zero; it has its own, distinct ASCII or UNICODE value. However, in most cases when you see a query's result table on the screen, nulls do appear as blank.

Second, the DBMS must have some consistent, known way of handling those nulls when performing queries. Typically, you will find that rows with nulls are not retrieved by a query such as the salary greater than 60,000 example unless the user explicitly asks for rows with a value of null. Most relational DBMSs today adhere to a three-valued logic truth table to determine retrieval behavior when they encounter nulls.

The inclusion of nulls in a relation can be extremely important. They provide a consistent way to distinguish between valid data such as a 0, and missing data. For example, it makes a great deal of difference to know that the balance in an account payable is 0 instead of unknown. The account with 0 is something we like to see; the account with an unknown balance could be a significant problem.

Note: The concept of unknown values is not unique to relational databases. Regardless of the data model it uses, a DBMS must contend with the problem of how to behave when querying against a null.

Rule 4: Dynamic Online Catalog Based on the Relational Model

Earlier in this book, you read about relational database data dictionaries. Codd very clearly specifies that those dictionaries (which he calls *catalogs*) should be made up of nothing but relations:

catalogs) should be made up of nothing but relations:

"The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to the interrogation as they apply to regular data."

One advantage of using the same data structures for the data dictionary as you do for data tables is that you have a consistent way to access all elements of the database. You need to learn only one query language. This also simplifies the DBMS itself, since it can use the same mechanism for handling data about the database (*metadata*) as it can data about the organization.

When you purchase a DBMS, it comes with its own way of handling a data dictionary. There is rarely anything you can do to change it. Therefore, the major implication of this particular rule comes in selecting relational software: You want to look for something that has a data dictionary that is made up of nothing but tables.

Note: Because of the way in which their schemas were implemented, it was rare for a prerelational DBMS to have an online data dictionary.

Rule 5: The Comprehensive Data Sublanguage Rule

A relational database must have some language that can maintain database structural elements, modify data, and retrieve data. Codd included the following rule that describes his ideas about what such a language should do:

"A relational system may support several languages and various modes of terminal use (for example, fill-in-the-blanks mode). However, there must be at least one language whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all of the following items:

- Data definition
- View definition
- Data manipulation (interactive and by program)
- Integrity constraints
- Transaction boundaries (begin, commit and rollback)"

The current SQL language does meet all of these rules. (Versions earlier than SQL-92 did not include complete support for primary keys and referential integrity.) Given that most of today's relational DBMSs use SQL as their primary data manipulation language, there would seem to be no issue here.

However, a DBMS that does not support SQL, but uses a graphic language, would technically not meet this rule. Nonetheless, there are several products today whose graphic language can perform all the tasks Codd has listed, without a command-line syntax. Such DBMSs might not be theoretically "fully relational," but, since they can perform all the necessary relational tasks, you lose nothing by not having the command-line language.

Note: Keep in mind the time frame in which Codd was writing. In 1985, the Macintosh—whose operating system legitimized the graphic user interface—was barely a year old. Most people still considered the GUI-equipped computers to be little more than toys.

Rule 6: The View Updating Rule

As you will read in more depth in Chapter 21 some views can be used to update data. Others—those created from more than one base table or view, those that do not contain the primary keys of their base tables, and so on—often cannot be used for updating. Codd's sixth rule speaks only about those that meet a DBMS's criteria for updatability:

"All views that are theoretically updatable are also updatable by the system."

This rule simply means that if a view meets the criteria for updatability, a DBMS must be able to handle that update and propagate the updates back to the base tables.

Note: DBMSs that used prerelational data models included constructs similar in concept to views. For example, CODASYL DBMSs included "subschemas," which allowed an application programmer to construct a subset of a schema to be used by a specific end user or by an application program.

Rule 7: High-Level Insert, Update, Delete

Codd wanted to ensure that a DBMS could handle multiple rows of data at a time, especially when data were modified. Therefore, the seventh rule requires that a DBMS's data manipulation facilities be able to insert, update, and delete more than one row with a single command:

"The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update, and deletion of data."

SQL provides this capability for today's relational DBMSs. What does it bring you? Being able to modify more than one row with a single command simplifies data manipulation logic. Rather than needing to scan a relation row by row to locate rows for modification, for example, you can specify logical criteria that identify rows to be affected and let the DBMS find the rows for you.

Rule 8: Physical Data Independence

One of the benefits of using a database system rather than a file processing system is that a DBMS isolates the user from physical storage details. The physical data independence rule speaks to this issue:

"Applications and terminal activities remain logically unimpaired whenever any changes are made in either storage representation or

"Applications and terminal activities remain logically unimpaired whenever any changes are made in either storage representation or access methods."

This means that you should be able to move the database from one disk volume to another, change the physical layout of the files, and so on, without any impact on the way in which application programs and end users interact with the tables in the database.

Most of today's DBMSs give you little control over the file structures used to store data on a disk. (Only the very largest mainframe systems allow systems programmers to determine physical storage structures.) Therefore, in a practical sense, physical data independence means that you should be able to move the database from one disk volume or directory to another, without affecting the logical design of the database and, therefore, the application programs and interactive users remain unaffected. With a few exceptions, most of today's DBMSs do provide physical data independence.

Note: Prerelational DBMSs generally fail this rule to a greater or lesser degree. The older the data model, the closer it was tied to its physical data storage. The tradeoff, however, is performance. Hierarchical systems are much faster than relational systems when processing data in tree traversal order. The same can be said for a CODASYL database. When traversing in set order, access will be faster than row-by-row access within a relational database. The tradeoff is flexibility to perform ad hoc queries, something at which relational systems excel.

Rule 9: Logical Data Independence

Logical data independence is a bit more subtle than physical data independence. In essence, it means that if you change the schema—perhaps adding or removing a table or adding a column to a table—then other parts of the schema that should not be affected by the change remain unaffected:

"Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit unimpairment are made to the base tables."

As an example, consider what happens when you add a table to a database. Since relations are logically independent of one another, adding a table should have absolutely no impact on any other table. To adhere to the logical data independence rule, a DBMS must ensure that there is indeed no impact on other tables.

On the other hand, if you delete a table from the database, such a modification is not "information preserving." Data will almost certainly be lost when the table is removed. Therefore, it is not necessary that application programs and interactive users be unaffected by the change.

Rule 10: Integrity Independence

Although the requirement for unique primary keys is a corollary to an earlier rule, the requirement for nonnull primary keys and for referential integrity is very explicit:

"Integrity constraints specific to a particular relational data base must be definable in the relational data sublanguage and storable in the catalog, not in the application programs.

A minimum of the following two integrity constraints must be supported:

1. Entity integrity: No component of a primary key is allowed to have a null value.
2. Relational integrity: For each distinct non-null foreign key value in a relational data base, there must exist a matching primary key value from the same domain."

Notice that the rule requires that the declaration of integrity constraints must be a part of whatever language is used to define database structure. In addition, integrity constraints of any kind must be stored in a data dictionary that can be accessed while the database is being used.

When IBM released its flagship relational database—DB/2—one of the two things users complained about was the lack of referential integrity support. IBM and other DBMS vendors at that time omitted referential integrity because it slowed down performance. Each time you modify a row of data, the DBMS must go to the data dictionary, search for an integrity rule, and perform the test indicated by the rule, all before performing an update. A referential integrity check of a single column can involve two or more disk accesses, all of which takes more time than simply making the modification directly to the base table.

However, without referential integrity, the relationships in a relational database very quickly become inconsistent. Retrieval operations therefore do not necessarily retrieve all data because the missing cross-references cause joins to omit data. In that case, the database is unreliable and virtually unusable. (Yes, IBM added referential integrity to DB/2 fairly quickly!)

Note: One solution to the problem of a DBMS not supporting referential integrity was to have application programmers code the referential integrity checks into application programs. This certainly works, but it puts the burden of integrity checking in the wrong place. It should be an integral part of the database, rather than left up to an application programmer.

Note: Most DBMSs using prerelational data models provided some types of integrity constraints, including domain constraints, unique entity identifiers, and required values (non-null). CODASYL could also enforce mandatory relationships, something akin to referential integrity.

Rule 11: Distribution Independence

As you will remember from Chapter 1 a distributed database is a database where the data themselves are stored on more than one computer. The database is therefore the union of all its parts. In practice, the parts are not unique, but contain a deal of duplicated data.