Codd's 12 Rules for an RDBMS

While most people think that any database that supports SQL is automatically a relational database, this is seldom the case, at least not completely. In Chapter 3, you learned about the basics and foundations of relational theory, but no discussion on this subject would be complete without looking at the rules that were formulated in 1985 in a two-part article published by *ComputerWorld* (Codd, E. F., "Is Your DBMS Really Relational?" and "Does Your DBMS Run By the Rules?" *ComputerWorld*, October 14 and October 21, 1985). They are also outlined at http://newton.uor.edu/FacultyFolder/CKettemborough/Codd12R.html and in *Fundamentals of Database Systems (The Benjamin/Cummings Publishing Co., ISBN 0805317538).* These rules go beyond relational theory and define a more specific set of criteria that need to be met in a RDBMS, if it is to be truly relational.

It may seem like old news, but the same criteria are still used today to measure how relational a database is. These rules are frequently brought up in conversations when discussing how well a particular database server is implemented.

Here I present the rules, along with brief comments as to how SQL Server 2000 meets each of them, or otherwise.

Rule 1: The Information Rule

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

This rule is basically an informal definition of a relational database and indicates that every piece of data that you permanently store in a database is located in a table. In general, SQL Server 2000 fulfills this rule, as you can't store any information in anything other than in a table. The variables used in this code can't be used to persist any data and therefore they are scoped to a single batch.

Rule 2: Guaranteed Access Rule

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

This rule basically stresses the importance of primary keys for locating data in the database. The table name locates the correct table, the column name finds the correct column, and the primary key value finds the row containing an individual data item of interest. In other words, each (atomic) piece of data is accessible by the combination of table name, primary key value, and column name. This rule exactly specifies how you access data in SQL Server. In a table, you can search for the primary key value (which is guaranteed to be unique, based on relational theory) and once you have the row, the data is accessed via the column name.

Rule 3: Systematic Treatment of NULL Values

`NULL` values (distinct from an empty character string or a string of blank characters, and distinct from zero) are supported in the RDBMS for representing missing information in a systematic way, independent of data type.

This rule requires that the RDBMS support a distinct `NULL` placeholder, regardless of data type. `NULL`s are distinct from an empty character string or any other number, and are interpreted according to the `ANSI_NULL setting`. When the `ANSI_NULL` setting is set on, `NULL`s are treated properly, such that `NULL + 'String Value' = NULL`. If the `ANSI_NULL` setting is off, which is allowed for backward compatibility with SQL Server, `NULL`s are treated in a nonstandard way such that `NULL + 'String Value' = 'String Value'`.

Rule 4: Dynamic Online Catalog Based on the Relational Model

The database 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 its interrogation as they apply to regular data.

This rule requires that a relational database be self-describing. In other words, the database must contain certain system tables whose columns describe the structure of the database itself, or alternatively, the database description is contained in user-accessible tables.

This rule is actually becoming more of a reality in each new version of SQL Server, as with the implementation of the `INFORMATION_SCHEMA` system views. However, it isn't exactly true because your view of the system catalog is not in a normalized form; if it was, then you could access the data in the same manner as user tables. Consequently, you have to find information on the database schema by using the SQL Server system functions to mine out additional information from the catalog.

Rule 5: Comprehensive Data Sublanguage Rule

A relational system may support several languages and various modes of terminal use (for example, the fill-in-blanks mode). However, there must be at least one language whose statements are expressible, by some well-defined syntax, as character strings, and whose ability to support all of the following is comprehensible: data definition, view definition, data manipulation (interactive and by program), integrity constraints, and transaction boundaries (begin, commit, and rollback).

This rule mandates using a relational database language, such as SQL, although SQL is not specifically required. The language must be able to support all the central functions of a DBMS–creating a database, retrieving and entering data, implementing database security, and so on. Transact-SQL fulfills this function for SQL Server and carries out all the data definition and manipulation tasks required to access data.

Rule 6: View Updating Rule

All views that are theoretically updateable are also updateable by the system.

This rule deals with views, which are virtual tables used to give various users of a database different views of its structure. It is one of the most challenging rules to implement in practice, and no commercial product fully satisfies it today.

A view is theoretically updateable as long as it's made up of columns that directly correspond to real table fields. In SQL Server, views are updateable as long as you don't update more than a single table in the statement, nor can you update a derived or constant field. SQL Server 2000 also implements `INSTEAD OF` triggers that you can apply to a view (Chapter 12). Hence, this rule is technically fulfilled using `INSTEAD OF` triggers, but in a less than straightforward manner.

Rule 7: High-Level Insert, Update, and Delete

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, updating, and deletion of data.

This rule stresses the set-oriented nature of a relational database. It requires that rows be treated as sets in insert, delete, and update operations. The rule is designed to prohibit implementations that only support row-at-a-time, navigational modification of the database. This is covered in the SQL language via the `INSERT`, `UPDATE`, and `DELETE` statements.

Rule 8: Physical Data Independence

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

Applications must still work using the same syntax, even when changes are made to the way in which the database internally implements data storage and access methods. This rule is tougher to implement than other rules, and implies that the way the data is stored physically must be independent of the logical manner in which it's accessed. This is saying that users shouldn't be concerned about how the data is stored or how it's accessed. For example:

- **Adding indexes:** Indexes determine how the data is actually stored, yet the user, through SQL, will never know that indexes are being used.
- **Modifications to the storage engines:** From time to time, Microsoft will obviously have to modify how SQL Server operates. However, SQL statements must be able to access the data in exactly the same manner that they did in any previous version.

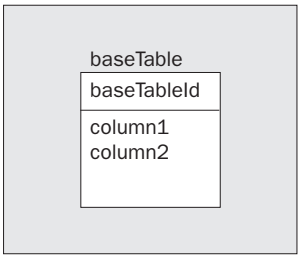
Microsoft has put a lot of work into this area, as they have a separate relational engine and storage engine and OLE DB is used to pass data between the two. Further reading on this topic is available in SQL Server 2000 Books Online in the *Database Engine Components* topic.

Rule 9: Logical Data Independence

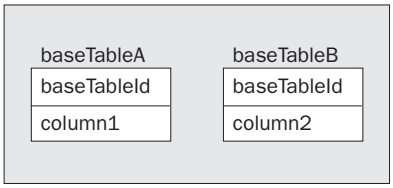
Application programs and terminal activities remain logically unimpaired when information preserving changes of any kind that theoretically permit unimpairment are made to the base tables.

Along with Rule 8, this rule insulates the user or application program from the low-level implementation of the database. Together they specify that specific access or storage techniques used by the RDBMS, and even changes to the structure of the tables in the database, should not affect the user's ability to work with the data. This rule is quite difficult to implement, since the goal is to be able to shield the table structures from the actual users of the table. In this way, if you add a column to a table, and if tables are split in a manner that doesn't add or subtract columns, then the application programs that call the database should be unimpaired.

For example, say you have the following table:



And you vertically break it up into two tables:



And if you then create this view:

```
CREATE VIEW baseTable
AS
SELECT baseTableId, column1, column2
FROM baseTableA
JOIN baseTableB
   ON baseTableA.baseTableId = baseTableB.baseTableId
```

the user should be unaffected. If you were to implement `INSTEAD OF` triggers on the view that had the same number of columns with the same names, you could meet this need. Note that the handling of identity columns can be tricky in views, as they require data to be entered, even when the data won't be used. See Chapter 12 for more details.

In Chapter 13, I discussed using distributed partitioned views to break up tables horizontally into identically structured tables. These new tables are then accessed using a view that joins the tables back together, thereby allowing them to be treated as a single table. This actually fulfills the criteria for Rule 9.

One thing of note: You can fulfill the requirements of this rule by implementing a layer of abstraction between the user and the data using stored procedures; the stored procedures would then become the "users" of the data.

Rule 10: Integrity Independence

Integrity constraints specific to a particular relational database 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:

- **Entity integrity: No component of a primary key is allowed to have a `NULL` value.**
- **Referential integrity: For each distinct non-`NULL` foreign key value in a relational database, there must exist a matching primary key value from the same domain.**

This rule says that the database language should support integrity constraints that restrict the data that can be entered into the database, and the database modifications that can be made. In other words, the RDBMS must support the definition and enforcement of entity integrity (primary keys) and referential integrity (foreign keys).

This rule requires that the database must be able to implement constraints to protect the data from invalid values and that careful database design is needed to ensure that referential integrity is achieved. SQL Server 2000 does a great job of providing the tools to make this rule a reality. You can protect your data from invalid values for almost any possible case using constraints and triggers. Most of Chapter 12 is spent covering the methods that you can use in SQL Server to implement integrity independence.

Rule 11: Distribution Independence

An RDBMS has distribution independence. Distribution independence implies that users should not have to be aware of whether a database is distributed.

This rule says that the database language must be able to manipulate data located on other computer systems. In essense, you should be able to split the data on the RDBMS out onto multiple physical systems without the user realizing it. This is beginning to become a reality with SQL Server 2000. With SQL Server 2000, the notion of federated database servers seamlessly sharing the load are becoming a reality. More reading on this subject can be found in SQL Server 2000 Books Online in the *Federated SQL Server 2000 Servers* topic.

Rule 12: Nonsubversion Rule

If an RDBMS has a low-level (single-record-at-a-time) language, that low-level language can't be used to subvert or bypass the integrity rules or constraints expressed in the higher-level (multiple-records-at-a-time) relational language.

This rule basically requires that alternate methods of accessing the data aren't able to bypass integrity constraints, which means that users won't be able to violate the rules of the database in any way. For most SQL Server 2000 applications, this rule is followed, as there are no methods of getting to the raw data and changing values other than by the methods prescribed by the database. However, there are two places where the rule is violated in SQL Server 2000:

- **Bulk Copy:** By default, the bulk copy routines can be used to insert data into the table directly and around the database server validations.
- **Disabling constraints and triggers:** There is syntax to disable constraints and triggers, thereby subverting this rule.

It's always good practice to ensure you use these two features carefully. They leave gaping holes in the integrity of your data, because they allow any values to be inserted in any field. Because you are expecting the data to be protected by the constraint you've applied, data value errors may occur in the programs that use the data without revalidating it first.

Conclusions

Codd's 12 rules for relational databases can be used to explain much about how SQL Server operates today. These rules were a major step forward in determining whether database vendors could call their system "relational," and presented stiff implementation challenges for database developers. Fifteen years on, even the implementation of the most complex of these rules is becoming achievable, though SQL Server (and other RDBMSs) still fall short of achieving all these objectives.