APPENDIX A

Codd’s 12 Rules for an RDBMS

Although most of us think that any database that supports SQL is automatically considered a relational database, this isn’t always the case—at least not completely. In Chapter 1, I discussed 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* magazine (“Is Your DBMS Really Relational?” and “Does Your DBMS Run By the Rules?” by E. F. Codd, *Computerworld*, October 14 and October 21, 1985). Many websites also outline these rules. These rules go beyond relational theory and defines more specific criteria that need to be met in an RDBMS, if it’s to be truly relational.

It might seem like old news, but the same criteria can still be 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. I present the rules in this appendix, along with brief comments as to whether SQL Server 2008 meets each of them, or otherwise. Relational theory has come a long way since these rules were first published, and “serious” theorists have enhanced and refined relational theory tremendously since then, as you’d expect. A good place for more serious learning is the website `http://www.dbdebunk.com`, run by C. J. Date and Fabian Pascal, or any of their books. If you want to see the debates on theory, the newsgroup `comp.databases.theory` is a truly interesting place. Of course, as the cover of this book states, my goal is practicality, with a foundation on theory, so I won’t delve too deeply into theory here at all. I present these 12 rules simply to set a basis for what a relational database started out to be and largely what it is and what it should be even today.

All these rules are based upon what’s sometimes referred to as the *foundation principle*, which states that for any system to be called a relational database management system, the relational capabilities must be able to manage it completely.

For the rest of this appendix, I’ll treat SQL Server 2008 specifically as a relational database engine, not in any of the other configurations in which it might be used, such as a plain data store, a document storage device, or whatever other way you might use SQL Server as a storage engine.

Rule 1: The Information Rule

> *All information in the relational database is represented in exactly one and only one way—by values in tables.*

This rule is an informal definition of a relational database and indicates that every piece of data that we permanently store in a database is located in a table.

In general, SQL Server fulfills this rule, because we cannot store any information in anything other than a table. We can’t use the variables in this code to persist any data, and therefore they’re scoped to a single batch.

Rule 2: Guaranteed Access Rule

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

This rule 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 we access data using an access language such as Transact-SQL (T-SQL) in SQL Server.

Using SQL, we can search for the primary key value (which is guaranteed to be unique, based on relational theory, as long as it has been defined), and once we have the row, the data is accessed via the column name. We can also access data by any of the columns in the table, though we aren't always guaranteed to receive a single row back.

Rule 3: Systematic Treatment of NULL Values

`NULL` *values (distinct from empty character string or a string of blank characters and distinct from zero or any other number) are supported in the fully relational 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 datatype. `NULL`s are distinct from an empty character string or any other number, and they are always to be considered as unknown values.

`NULL`s must propagate through mathematic operations as well as string operations. `NULL + <anything> = NULL`, the logic being that `NULL` means "unknown." If you add something known to something unknown, you still don't know what you have, so it's still unknown.

There are a few settings in SQL Server that can customize how `NULL`s are treated. Most of these settings exist because of some poor practices that were allowed in early versions of SQL Server:

- `ANSI_NULLS`: Determines how `NULL` comparisons are handled. When `OFF`, then `NULL = NULL` is `True` for the comparison, and when `ON` (the default), `NULL = NULL` returns `UNKNOWN`.
- `CONCAT_NULL_YIELDS_NULL`: When the `CONCAT_NULL_YIELDS_NULL` setting is set `ON`, `NULL`s are treated properly, such that `NULL + 'String Value' = NULL`. If the `CONCAT_NULL_YIELDS_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 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 becoming more of a reality in each new version of SQL Server, as with the implementation of the `INFORMATION_SCHEMA` system views. The `INFORMATION_SCHEMA` is a schema that has a set of views to look at much of the metadata for the tables, the relationships, the constraints, and even the code in the database.

Anything else you need to know can most likely be viewed in the system views (in the `SYS` schema). They're the system views that replaced the system tables in 2005 that we had used since the beginning of SQL Server time. These views are far easier to read and use, and most all the data is self-explanatory, rather than requiring bitwise operations on some columns to find the value.

Rule 5: Comprehensive Data Sublanguage Rule

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

This rule mandates the existence of a relational database language, such as SQL, to manipulate data. SQL as such isn't 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. T-SQL fulfils this function for SQL Server and carries out all the data definition and manipulation tasks required to access data.

SQL is a nonprocedural language, in that you don't specify "how" things happen, or even where. You simply ask a question of the relational server, and it does the work.

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's 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 columns. In SQL Server, views are updateable as long as you don't update more than a single table in the statement; neither can you update a derived or constant field. SQL Server 2000 also implemented `INSTEAD OF` triggers that you can apply to a view (see Chapter 6). Hence, this rule can be technically fulfilled using `INSTEAD OF` triggers, but in what can be a less-than-straightforward manner. You need to take care when considering how to apply updates, especially if the view contains a `GROUP BY` clause and possibly aggregates.

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, update, 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 support only row-at-a-time, navigational modification of the database. The SQL language covers this via the `INSERT`, `UPDATE`, and `DELETE` statements.

Even the CLR doesn't allow you to access the physical files where the data is stored, but BCP does kind of go around this. As always, you have to be extra careful when you use the low-level tools that can modify the data without going through the typical SQL syntax, because it can ignore the rules you have set up, introducing inconsistencies into your data.

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 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. In fact, users of the data need only be able to get the basic definition of the data they need.

Other things that shouldn't affect the user's view of the data are as follows:

- *Adding indexes*: Indexes determine how the data is stored, yet the user, through SQL, will never know that indexes are being used.
- *Changing the filegroup of an object*: Just moving a table to a new filegroup will not affect the application. You access the data in the same way no matter where it is physically located.
- *Using partitioning*: Beyond moving entire tables around to different filegroups, you can move parts of a table around by using partitioning technologies to spread access around to different independent subsystems to enhance performance.
- *Modifying the storage engine*: From time to time, Microsoft has to modify how SQL Server operates (especially in major version upgrades). However, SQL statements must appear to access the data in the same manner as they did in any previous version, only (we hope) faster.

Microsoft has put a lot of work into this area, because SQL Server has 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 2008 Books Online in the "Database Engine Components" topic or in *Inside Microsoft SQL Server 2005: The Storage Engine* by Kalen Delaney (Microsoft Press, 2006).

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—shouldn't affect the user's ability to work with the data.

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 table in Figure A-1.

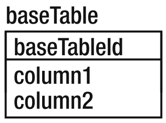

Figure A-1. *Small sample table*

Then, say you vertically break it up into two tables (see Figure A-2).

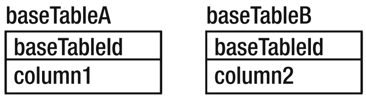

Figure A-2. *Small sample table split into two tables*

You then could 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 seamlessly meet the need to manage the view in the exact manner the table was managed. Note that the handling of identity columns can be tricky in views, because they require data to be entered, even when the data won't be used. See Chapter 6 for more details on creating `INSTEAD OF` triggers.

Of course, you cannot always make this rule work if columns or tables are removed from the system, but you can make the rule work if columns and data are simply added.

Tip Always access data from the RDBMS by name, and not by position or by using the `SELECT *` wildcard. The order of columns shouldn't make a difference to the application.

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.

The database must support a minimum of the following two integrity constraints:

- *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 internally support the definition and enforcement of entity integrity (primary keys) and referential integrity (foreign keys).

It requires that the database 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 2008 does a great job of providing the tools to make this rule a reality. We can protect our data from invalid values for most any possible case using constraints and triggers. Most of Chapter 6 was spent covering the methods that we can use in SQL Server to implement integrity independence.

Rule 11: Distribution Independence

The data manipulation sublanguage of a relational DBMS must enable application programs and terminal activities to remain logically unimpaired whether and whenever data are physically centralized or distributed.

This rule says that the database language must be able to manipulate data located on other computer systems. In essence, we should be able to split the data on the RDBMS out onto multiple physical systems without the user realizing it. SQL Server 2008 supports distributed transactions among SQL Server sources, as well as other types of sources using the Microsoft Distributed Transaction Coordinator service.

Another distribution-independence possibility is a group of database servers working together more or less as one. Database servers working together like this are considered to be *federated*. With every new SQL Server version, the notion of federated database servers seamlessly sharing the load is becoming a definite reality. More reading on this subject can be found in the SQL Server 2008 Books Online in the "Federated Database Servers" topic.

Rule 12: Non-Subversion Rule

If a relational system has or supports a low-level (single-record-at-a-time) language, that low-level language cannot 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 requires that alternate methods of accessing the data are not able to bypass integrity constraints, which means that users can't violate the rules of the database in any way. For most SQL

Server 2008 applications, this rule is followed, because there are no methods of getting to the raw data and changing values other than by the methods prescribed by the database. However, SQL Server 2008 violates this rule in two places:

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

It's always good practice to make sure 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 column. Because you're expecting the data to be protected by the constraint you've applied, data value errors might occur in the programs that use the data, without revalidating it first.

Summary

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 a database vendor could call his 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 their objectives.