Whitepaper

embarcadero

I Do Declare

By Joe Celko

For Embarcadero Technologies

September, 2015

Table of Contents

Most of us began programming in procedural languages. But SQL is a declarative language. Chris Date described the difference as “What, not How” in one of his books. SQL has three sub-languages; the DDL or Data Declaration Language, the DML or Data Manipulation Language and the DCL or Data Control Language.

The DCL is the sub-language where we control user and admin access to the database. It is a procedural language whose major commands are GRANT (give a privilege to a user) and REVOKE (remove a privilege from a user). SQL Server also has a DENY command that permanently blocks a user's access to a database resource. This sub-language is the smallest and the one that nobody learns in school.

The DML is where you do the queries that return the data from the schema (SELECT) and statements that change the data (for users UPDATE, INSERT, DELETE, and for admin use ALTER, DROP, CREATE) in the schema. This is where we spend most of our time, both in the classroom and on the job.

The DDL gets some attention in the classroom, but most working programmers are not allowed add or remove schema objects. This is an administrative power, as it should be. I have worked in a shop where everyone could do an ALTER TABLE and seen the chaos that an unpredictable schema creates.

But most of the hard work in SQL should be done in the DDL. This short paper is an attempt to fill the gaps in the education of working programmers.

Data Types

The first place to start in a database language is with the data types it allows. SQL deliberately picked a wide range of data types because we had no idea which host languages would use a database. We had to cover everything! In particular, back when we started the ANSI X3H2 Database Standards committee we had what were called “the X3J” programming languages. The programming languages were under the jurisdiction of that ANSI committee in those days. And to be honest, about all that we had was FORTRAN and COBOL, with an effort made at standardizing BASIC.

The proper mental model of numbers in SQL is not to worry about the "bits and bytes" level of the physical representation, but to think in abstract terms. We do not care about if the hardware is twos-complement, ones-complement, high end or low end and all the various word sizes. The idea is that any host language can find an SQL numeric type that matches one of its own, or we can cast from the SQL internal formats to the host language formats. Remember that SQL is meant to be used with a host language and not by itself. That is one reason we say "SQL means Scarcely Qualifies as a Language" on the committee.

Numeric data types in SQL are classified as either exact or approximate. They are for computations and should never be used as identifiers. Be aware that the rules for computations in SQL might not match the host language. This is why you limit yourself to the most basic numeric types whenever possible.

Exact Numeric Data Types

An exact numeric value has a precision, p, and a scale, s. The precision is a positive integer that determines the number of significant digits in a particular radix. The standard said the radix can be either binary or decimal, but today, everyone uses decimal math. The scale is a non-negative integer that tells you how many radix places the number has.

The data types INTEGER, BIGINT, SMALLINT, NUMERIC(p, s), and DECIMAL(p,s) are exact numeric types. An integer has a scale of zero but the syntax simply uses the word INTEGER or the abbreviation INT, but if you use the abbreviation you will look like a C family programmer.

SMALLINT has a scale of zero, but the range of values it can hold are less than or equal to the range that INTEGER can hold in the implementation. Likewise, BIGINT has a scale of zero, but the range of values it can hold are greater than or equal to the range that INTEGER can hold in the implementation.

BIGINT can be really huge, so you should not need it very often in your entire career. It is often a bad code smell that tells you it has been used as an as identifier.

DECIMAL(p,s) can also be written DEC(p,s) . For example, DECIMAL(8,2) could be used to hold the number 123456.78, which has eight digits and two decimal places.

The difference between NUMERIC(p,s) and DECIMAL(p,s) is subtle. NUMERIC(p,s) specifies the exact precision and scale to be used. DECIMAL(p,s) specifies the exact scale, but the precision is implementation-defined to be equal to or greater than the specified value. That means DECIMAL(p,s) can have some room for rounding and NUMERIC(p,s) does not. Mainframe COBOL programmers can think of NUMERIC(p,s) as a PICTURE numeric type, whereas DECIMAL(p,s) is like a BCD. The use of BCD is not common today, but was popular on older mainframe business computers. I recommend using DECIMAL(p,s) because it might enjoy some extra precision when doing math.

The INCITS/H2 Database Standards Committee debated about defining precision and scales in the standard in the early days of SQL and finally gave up. This means I can start losing high-order digits, especially with a division operation, where it is perfectly legal to make all results single-digit integers.

Nobody does anything that stupid in practice. In the real world, some vendors allow you to adjust the number of decimal places as a system parameter, some default to a known number of decimal places, and some display as many decimal places as they can so that you can round off to what you want. You will simply have to learn what your implementation does by experimenting with it.

When an exact or approximate numeric value is assigned to an exact numeric column, it may not fit. SQL says that the database engine will use an approximation that preserves leading significant digits of the original number after rounding or truncating. The choice of whether to truncate or round is implementation-defined, however. This can lead to some surprises when you have to shift data among SQL implementations, or storage values from a host language program into an SQL table. It is probably a good idea to create the columns with more decimal places than you think you need.

Truncation is defined as truncation toward zero; this means that 1.5 would truncate to 1, and -1.5 would truncate to -1. This is not true for all programming languages; everyone agrees on truncation toward zero for the positive numbers, but you will find that negative numbers may truncate away from zero (i.e., -1.5 would truncate to -2).

SQL is also indecisive about rounding, leaving the implementation free to determine its method.

What about NULLs in this data type? To be an SQL data type, you have to have NULLs. The general rule is that NULLs propagate, so you need to watch your DDL.

Heuristics for Exact Numeric Data:

Make the column NOT NULL first, then come back and figure out what NULL means for that attribute.

Add a CHECK() constraint for the actual range of the attribute. One of my favorite horror story was an order entry system that allowed negative quantities. The application treated negative quantity as a return or refund and sent a check. Order -5000 jars of boiled buzzard eggs at $50 per jar and wait for the check in the mail!

The most common constraints that should be used, but are usually skipped, are to establish cardinals:

```
x INTEGER NOT NULL CHECK (x > 0)
```

or non-negative values

```
x INTEGER NOT NULL CHECK (x >= 0)
```

and enforce ranges and subsets if they apply:

```
x INTEGER NOT NULL CHECK (x BETWEEN 1 AND 10)

x INTEGER NOT NULL CHECK (x IN (0, 1, 2, 9))
```

These are simple predicates that can pass information to the optimizer, so they both maintains data quality and improves performance. It also saves you the trouble of doing this same validation in hundred or thousands of programs in the system.

More elaborate constraints are useful and should be used as appropriate. But those complex constraints require planning; these can be done almost immediately.

Approximate Numeric Data Types

An approximate numeric value consists of a significand or mantissa, and an exponent. The term "mantissa" may cause confusion, however, because it can also refer to the fractional part of the common logarithm. and an exponent. The significand is a signed numeric value; the exponent is a signed integer that specifies the magnitude of the significand for a particular radix or base. In English, it means we represent all numbers in the form:

$$\text{significand} \times \text{base}^{\text{exponent}}$$

In SQL, we have FLOAT(p), REAL, and DOUBLE PRECISION for the approximate numeric types. There is a subtle difference between FLOAT(p), which has a binary precision equal to or greater than the value given, and REAL, which has an implementation-defined precision.

In the real world REAL and DOUBLE PRECISION are the IEEE Standard 754 for floating point numbers; FLOAT(p) is almost never used. IEEE math functions are built into processor chips so they will run faster than a software implementation. IEEE Standard 754 is binary and uses 32 bits for single precision and 64 bits for double precision, which is just right for personal computers and most Unix and Linux platforms.

Floating point math is truly elaborate and complicated. Zero cannot be directly represented in this format, so it is modeled as a special value denoted with an exponent field of zero and a fraction field of zero. The sign field can make this either -0 and +0, which are distinct values that compare as equal.

On top of this, we have NaN ("Not a Number") values for bit configurations that do not represent valid numbers. NaN's are represented by a bit pattern with an exponent of all ones and a non-zero fraction. There are two categories of NaN: QNaN (Quiet NaN) and SNaN (Signaling NaN).

A QNaN is a NaN with the most significant fraction bit set. QNaN's propagate freely through most arithmetic operations. These values pop out of an operation when the result is not mathematically defined, like division by zero.

An SNaN is a NaN with the most significant fraction bit clear. It is used to signal an exception when used in operations. SNaN's can be handy to assign to uninitialized variables to trap premature usage.

Semantically, QNaN's denote indeterminate operations, while SNaN's denote invalid operations.

The two values "+infinity" and "-infinity" are denoted with an exponent of all ones and a fraction of all zeroes. The sign bit distinguishes between negative infinity and positive infinity. Being able to denote infinity as a specific value is useful because it allows operations to continue past overflow situations.

Operations with infinite values are well defined in IEEE floating point, but do not exist in SQL or most other programming languages. Much of the SQL Standard allows implementation defined rounding, truncation and precision so as to avoid limiting the language to particular hardware platforms. If the IEEE rules for math were allowed in SQL, then we need type conversion rules for infinite and a way to represent an infinite exact numeric value after the conversion.

Other Numeric Types

There are a few surprises in converting from one numeric type to another. The SQL Standard left it up to the implementation to answer a lot of basic questions, so the programmer has to know his SQL package. Try not to use exotic, proprietary numeric data types unless necessary. You want to have portable data and not a single vendor data store.

My heuristic is never to use floating point. We have DECIMAL(p,s). People have enough trouble with NULLs, so let's not go there. If you actually need floating point math, then you probably should be using a specialized math package with all of the software needed and run it on a machine with the proper hardware.

In essence, I am advocating giving up your 1960's slide rule for a digit pocket calculator. The worst use of approximate numeric data types is for currency. It is illegal under GAAP, EU regulations and other laws. The rules and financial practices specify the number of decimal places and the rules for doing math, such as commercial rounding.

Character Data Types

SQL-89 defined a CHARACTER(n) or CHAR(n) data type, which represents a fixed-length string of (n) printable characters, where (n) is always greater than zero. Some implementations allow the string to contain control characters, but this is not the usual case. Originally, we had to dealt with EBCDIC, but today this means ASCII and UNICODE character sets and we use their collation sequences for sorting.

Starting with SQL-92, we added the VARYING CHARACTER(n) or VARCHAR(n), which was already present in many implementations. A VARCHAR(n) represents a string that varies in length from 1 to (n) printable characters. This is important; SQL does not allow a string column of zero length, but you may find vendors who do, so that you can store an empty string.

SQL-92 also added NATIONAL CHARACTER(n) and NATIONAL VARYING CHARACTER(n) data types (or NCHAR(n) and NVARCHAR(n), respectively), which are made up of printable characters drawn from ISO-defined UNICODE character sets. The literal values use the syntax N'<string>' in these data types.

SQL-92 also allows the database administrator to define collation sequences and do other things with the character sets. A Consortium (http://www.unicode.org/) maintains the Unicode standards and makes them available in book form (UNICODE STANDARD, VERSION 5.0; ISBN-13: 978-0321480910) or on the website.

When the Standards got to SQL:2006, we had added a lot of things to handle Unicode and XML data, but kept the basic string manipulations pretty simple compared to what vendors have. Unicode requires that all national character sets include a small basic subset of ASCII characters (Latin letters, digits and some punctuation marks). This subset is used for International standards. That is why you see Latin letters for the names of metric units and other ISO standards in databases in Japanese, Chinese, Arabic and other non-Latin scripts. Think about that when you design encoding schemes.

Problems with Strings

Different programming languages handle strings differently. You simply have to do some un-learning with you get to SQL. Here are the major problem areas for programmers.

In SQL, character strings are printable characters enclosed in single quotation marks. Many older SQL implementations and several programming languages use double quotation marks or make it an option so that the single quotation mark can be used as an apostrophe. SQL uses two apostrophes together to represent a single apostrophe in a string literal.

Double quotation marks are reserved for column names that have embedded spaces or that are also SQL reserved words.

Problems of String Equality

No two programming languages agree on how to compare character strings as equal unless they are identical in length and match position for position, exactly character for character.

The first problem is whether uppercase and lowercase versions of a letter compare as equal to each other. Only Latin, Greek, Cyrillic and Arabic have cases; the first three have upper and lower cases, while Arabic is a connected script that has initial, middle, terminal and stand-alone forms of its letters. Most programming languages, including

SQL, ignore case in the program text, but not always in the data. Some SQL implementations allow the DBA to set uppercase and lowercase matching as a system configuration parameter. Other programming languages ignore upper- and lowercase differences. The Standard SQL has two folding functions (yes, that is the name) that change the case of a string:

`LOWER(<string expression>)` shifts all letters in the parameter string to corresponding lowercase letters;

`UPPER(<string expression>)` shifts all letters in the string to uppercase. Most implementations have had these functions (perhaps with different names) as vendor library functions.

In SQL, equality between strings of unequal length is calculated by first padding out the shorter string with blanks on the right-hand side until the strings are of the same length. Then they are matched, position for position, for identical values. If one position fails to match, the equality fails.

In contrast, the Xbase languages (FoxPro, dBase, and so on) truncate the longer string to the length of the shorter string and then match them position for position.

Because of the padding, you may find doing a GROUP BY on a VARCHAR(n) has unpredictable results. Which of the possible equal strings represents the group? Shortest? Longest?

Problems of String Ordering

The usual >, <, =, <=, >= , <> theta operators apply using the collation in the SQL engine. SQL-89 was silent on the collating sequence to be used. In practice, almost all SQL implementations used either ASCII or EBCDIC, which are both Latin I character sets in ISO terminology. A few implementations have a Dictionary or Library order option (uppercase and lowercase letters mixed together in alphabetic order: {A, a, B, b, C, c, …} and many vendors offer a national-language option that was based on the appropriate local national Standard.

National language options can be very complicated. The Nordic languages all shared a common character set for newspapers, but they do not sort the same letters in the same position. German was sorted differently in Germany and in Austria. Spain decided to quit sorting 'ch' and 'll' as if they were single characters. You really need to look at the ISO Unicode implementation for your particular product.

The Standard SQL allows the DBA to define a collating sequence that is used for comparisons. The feature is becoming more common as we come more globalized, but you have to see what the vendor of your SQL product actually supports.

Heuristics for String

These are much like the heuristics I use for numeric values.

Try to avoid VARCHAR(n) in favor of CHAR(n) data types. The length (n) of a column is part of the validation of the data. Do not use a default size; think and plan your data. We know that a ZIP code is five digits, no more, no less. This lets us allocate space on screens and paper forms. But setting this column to VARCHAR(50), VARCHAR(256) or whatever some package gave you as a default size. This is a waste, it invites garbage data and destroys the code as documentation.

Make the column NOT NULL first, then com back and figure out what NULL means for that attribute. Add a CHECK() constraint for the string. The most common constraints that should be used, but are usually skipped, are simple Regular Expressions. SQL originally had only the LIKE predicate which could only match to patterns with % (the zero or more characters wildcard) and _ (the single character wildcard). The choice of the underscore was not a good idea; when we got good printers and clean screen displays, it became hard to tell how many underscores you had in a row. But since LIKE is simple, it works very fast.

Vendors added some grep() extensions to the LIKE predicate, but the ANSI/ISO Standard predicate is <> `SIMILAR TO <regular expression>`, a version of grep() based on the POSIX operating system.

These predicates can pass information to the optimizer, but they maintain data integrity on insertion in a single place in the schema. It also saves you the trouble of doing this same validation in hundred or thousands of programs in the system.

```
airport_code CHAR(3) NOT NULL

 CHECK (airport_code SIMILAR TO '[A-Z][A-Z][A-Z]')
```

As an aside, IATA airport codes are the three-letter code which is used in passenger reservation, ticketing, and baggage-handling systems. However, there is also the ICAO airport code, a four-letter code which is used by air-traffic control systems and for airports that do not have an IATA airport code.

```
zip_code CHAR(5) NOT NULL

 CHECK (zip_code SIMILAR TO '[0-9][0-9][0-9][0-9][0-9]')

something_name VARCHAR(25) NOT NULL

 CHECK (something_name = UPPER(zip_code))
```

If you need complicated regular expression, you can Google the patterns on various websites (http://regexlib.com/). But these sites might have extensions not supported in your SQL. Consider this example for email addresses:

```
^([a-zA-Z0-9_\-\.]+)@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.)|(([a-
zA-Z0-9\-]+\.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(\]?)$
```

This email validator allows for everything from ipaddress and country-code domains, to very rare characters in the username part.

Default

The default clause is an underused feature, whose syntax is

```
<default clause> ::=

 [CONSTRAINT <constraint name>] DEFAULT <default option>

<default option> ::= <literal> | <system value> | NULL

<system value> ::= CURRENT_DATE | CURRENT_TIME | CURRENT_TIMESTAMP |
SYSTEM_USER | SESSION_USER | CURRENT_USER | NEXT VALUE FOR <sequence
name>
```

Almost nobody uses a constraint name, since there is only one per column. This might be a good way to document your code, however. Whenever the system does not have an explicit value to put into this column, it will look for its DEFAULT clause and use that value. The default option can be a literal value of the relevant data type, or something provided by the system, such as the current timestamp, current date, current user identifier, the exit value from a SEQUENCE and so forth. If you do not provide a DEFAULT clause and the column is NULL-able, the system will provide a NULL as the default. If all that fails, you will get an error message about missing data.

The DEFAULT is a little known and little used option for inserting a column into a table. This is funny, since most programmers know they can use NULL in a <row value constructor list> when they do an insertion. Just use the keyword DEFAULT instead of NULL! But there is also a short hand

```
INSERT INTO <table name> DEFAULT VALUES;
```

The DEFAULT VALUES clause is a shorthand for VALUES (DEFAULT, DEFAULT, ..., DEFAULT), so it is just shorthand for a particular single row insertion. Almost nobody knows it exists. Any column not in the list will be assigned NULL or its explicit DEFAULT value.

This is a good way to make the database do a lot of work that you would otherwise have to code into all the application programs. The most common tricks are to use a zero in numeric columns, a string to encode a missing value ('{{unknown}}') or an application defined default ('same address') in character columns, and the system timestamp to mark transactions.

The heuristics are obvious. Make sure the default value is the same as the data type of the column. Casting the default from integer to decimal or vice versa, short string to long string, etc. is a waste of time and resources. It is also poor documentation.

Temporal Data

There are three temporal data types; DATE, TIME and TIMESTAMP. The first two explain themselves and the third one is a combination of a date and a time. We use the Common Era calendar and the ISO standard UTC time. We have the option to keep local time instead of the UTC value in the table.

DATE is a string of digits that are made up of the four digit year, a dash, two digit month, dash and a two digit day within the month. Example: '2015-06-25' for June 25th of 2015. This is the only date display format allowed in ANSI/ISO Standard SQL.

TIME(n) is made up of a two digit hour between '00' and '23', colon, a two digit minute between '00' and '59', colon, and a two digit second between '00' and '59' or '60', if the leap second is still in use. Seconds can also have decimal places shown by (n) from zero to an implementation defined accuracy. The FIPS-127 standard requires at least five decimal places after the second and modern products typically go to Nanoseconds (seven decimal places).

TIMESTAMP is a DATE string followed by a space and then a time string. It attempts to model a point in the time continuum.

The term field in SQL has nothing to do with records and files. It refers to parts of a temporal value, namely, {YEAR, MONTH, DAY, HOUR, MINUTE, SECOND}

Because temporal data was late in being standardizes, there is a lot of vendor implementations and syntax. In particular, many vendors have functions for doing display formatting in the database. Never use them; they exist because we had to keep the COBOL people happy by getting the data into strings that match local dialect conventions. I am sure you immediately recognized the month names from Czech: leden, únor, březen, duben, květen, červen, červenec, srpen, září, and říjen. No? This is why the ISO standards use digits. And of course you immediately know if 12/01/2015 is in January or December without knowing if this is the US or UK convention.

SQL is a tiered architecture, so display formatting in done in a presentation layer, not the database. This means that temporal data has to be in one and only standard format, so all the presentation layers (current and future) can handle the data the same way.

There is also an INTERVAL data type in the ANSI/ISO Standard. This is a "temporal unit of duration" which may or may not be supported in your SQL, and if it is supported, the syntax might not be standard.

There are two types of intervals: year-month, which stores the year and month (YYYY-MM); and day-time (DD HH:MM:SS), which stores the days, hours, minutes, and seconds. Intervals can be positive or negative. All INTERVAL components are integers, except seconds, which can contain decimal fractions of a second. You can only compare year-month intervals to other year-month intervals and day-time values to other day-time values.

Temporal periods have to be modeled with (start_timestamp, end_timestamp) pairs. Since we use the ISO-8601 half-open interval model, the end_timestamp has to be NULL-able to model a period that is still open. In English, we know when you checked into the hotel, but have no idea when you will check out in the future.

My heuristics are not so simple:

Use DATE. Most of the real work in the world is done with DATE precision, and not to nanoseconds. Dates also avoid the question of local timezones, Daylight Saving Time, etc.

Remember your system defaults:

```
t TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL

dt TIMESTAMP DEFAULT CURRENT_DATE NOT NULL
```

We need a CHECK (start_timestamp <= end_timestamp) constraint. It is also a good idea to restrict the range on temporal columns. It is surprisingly easy to accept absurd future or past timestamps. But it is also important to assure that multiple temporal values have the correct ordering. This means named constraints for the orderings.

Do not use INTERVAL in the columns. It is not very portable and its purpose is really to do temporal math, not to record an event.

Do not use TIMESTAMP data types unless you really need it. Most work is not done to nanosecond precision. Rounding and truncating among timestamps also has an overhead.

Not Null

All data types in SQL are created NULL-able. This constraint should be the default and removed only after you have defined exactly what s NULL in that columns means in the data model. For example, a NULL end_timestamp in a (start_timestamp, end_timestamp) pair means "eternity" and needs to be treated that way in the application code. But a NULL in a "cellphone_nbr" means that this guy has no cell phone.

The NULL gives us two slightly different three valued logics. NULLs return UNKNOWN in most predicates; the idea is that you cannot determine if a predicate is TRUE or FALSE if there is no known value.

In the DML, a predicate in the WHERE or ON clauses that returns a FALSE or UNKNOWN rejects that row in the table. We only look at TRUE facts.

But in the DDL, if a predicate in a CHECK() clause returns a TRUE or UNKNOWN ("benefit of the doubt" rule) then the row is accepted. We did this so that the DDL would not be overloaded with IS NULL conditions.

Check

I have already talked about adding check constraints to columns to control ranges and formats. But a CHECK() can be declared at the table level. I also talked about ordering constraints on multiple temporal columns.

But there are most complex relationships among and within columns. As long as they can be expressed with theta operators, they can be passed to the optimizer. However, a good trick is to use a CASE expression instead of a TRIGGER, or other procedural code. In effect, the CASE replaces nested if-then-else statements. The WHEN clauses test in order, and this lets you sequence predicates; return a 'T' for a true result or return an 'F" for a false one.

```
CHECK

(CASE

WHEN <expression 1> THEN 'T'

WHEN <expression 2> THEN 'T'

…

WHEN <expression n> THEN 'F'

ELSE 'F' END = 'T')
```

Writing a check digit validation for an identifier string can get a bit long, but it is do-able. Use a SUBSTRING () to pull out each digit, use a CASE expression to the get the appropriate weight for that digit, sum all these weights and take the MOD() function and compare it to the final digit.

```
CASE SUBSTRING (x FROM 1 FOR 1)
WHEN '1' THEN 1
WHEN '2' THEN 3
WHEN '3' THEN 7
WHEN '4' THEN 1
WHEN '5' THEN 3
WHEN '6' THEN 7
WHEN '7' THEN 1
WHEN '8' THEN 3
WHEN '9' THEN 7
ELSE 0 END
```

References

The <references specification> is the simplest version of a referential constraint definition, which can be quite tricky. For now, let us just consider the simplest case:

```
<references specification> ::=

 [CONSTRAINT <constraint name>]

 REFERENCES <referenced table name>[(<reference column>)]
```

What this says is that the value in this column of the referencing table must appear somewhere in the referenced table's column that is named in the constraint. Notice the terms referencing and referenced. This is not the same as the parent and child terms used in network databases. Those terms were based on pointer chains that were traversed in one direction; that is, you cannot find a path back to the parent from a child node in the network. Another difference is that the referencing and referenced tables can be the same table. Self-references can be a useful trick.

Furthermore, the referenced column must have either a UNIQUE or a PRIMARY KEY constraint. For example, you can set up a rule that the Orders table will have orders only for goods that appear in the Inventory table.

If no <reference column> is given, then the PRIMARY KEY column(s) of the referenced table is assumed to be the target. There is no rule to prevent several columns from referencing the same target column. For example, we might have a table of flight crews that has pilot and copilot columns that both reference a table of certified pilots.

A circular reference is a relationship in which one table references a second table, which in turn references the first table. The old gag about "you cannot get a job until you have experience, and you cannot get experience until you have a job!" is the classic version of this.

Referential Actions

The REFERENCES clause can have two sub-clauses that take actions when a database event changes the referenced table.

<referential triggered action> ::=

<update rule> [<delete rule>] | <delete rule> [<update rule>]

<update rule> ::= ON UPDATE <referential action>

<delete rule> ::= ON DELETE <referential action>

<referential action> ::= NO ACTION | CASCADE | SET DEFAULT SET NULL

When the referenced table is changed, one of the referential actions is set in motion by the SQL engine.

1) The NO ACTION option explains itself. Nothing is changed in the referencing table and some error message about reference violation will be raised. If a referential constraint does not specify any ON UPDATE or ON DELETE rule, update rule, then NO ACTION is implicit.

2) The CASCADE option will change the values in the referencing table to the new value in the referenced table. This is a very common method of DDL programming that allows you to set up a single table as the trusted source for an identifier. This way the system can propagate changes automatically.

This removes one of the arguments for non-relational system generated surrogate keys. In early SQL products that were based on a file system for their physical implementation, the values were repeated both the referenced and referencing tables. Why? The tables were regarded as separate units of storage, like files.

Later SQL products regarded the schema as a whole. The referenced values appeared once in the referenced table, and the referencing tables obtained them by following pointer chains to that one occurrence in the schema. The results are much faster update cascades, a physically smaller database, faster joins and faster aggregations.

3) The SET DEFAULT option will change the values in the referencing table to the default value of that column. Obviously, the column needs to have some DEFAULT declared for it for this to work.

4) The SET NULL option will change the values in the referencing table to a NULL. Obviously, the referencing column needs to be NULL-able.

The SET NULL and SET DEFAULT options are not actually used much. The workhorse is CASCADE, which was created to replace about 70-80% of the Triggers we had to write in the early days of SQL.

Full ANSI/ISO Standard SQL has more options about how matching is done between the referenced and referencing tables. Nobody uses them since they are tricky and if you have a clean simple design you do not need them.

Nested UNIQUE Constraints

One of the basic tricks in SQL is representing a one-to-one or many-to-many relationship with a table that references the two (or more) entity tables by their primary keys. This third table has several popular names such as "junction table", “Associative Entity” or "join table", but we know that it is a relationship. The terms "junction table" is a pointer structure from Network databases, not part of RDBMS. For example given two tables,

```
CREATE TABLE Boys
(boy_name VARCHAR(30) NOT NULL PRIMARY KEY
 ...);
CREATE TABLE Girls
(girl_name VARCHAR(30) NOT NULL PRIMARY KEY,
 ... );
```

Yes, I know using names for a key is a bad practice, but it will make my examples easier to read. There are a lot of different relationships that we can make between these two tables. If you don't believe me, just watch the Jerry Springer Show sometime. The simplest relationship table looks like this:

```
CREATE TABLE Couples
(boy_name INTEGER NOT NULL
   REFERENCES Boys (boy_name)
   ON UPDATE CASCADE
   ON DELETE CASCADE,
 girl_name INTEGER NOT NULL,
    REFERENCES Girls(girl_name)
    ON UPDATE CASCADE
    ON DELETE CASCADE);
```

The Couples table allows us to insert rows like this:

```
INSERT INTO Couples

VALUES

('Joe Celko', 'Miley Cyrus'),

('Joe Celko', 'Lady GaGa'),

('Alec Baldwin', 'Lady GaGa'),

('Joe Celko', 'Miley Cyrus');
```

Opps! I am shown twice with 'Miley Cyrus' because the Couples table does not have its own compound key. This is an easy mistake to make, but fixing it is not an obvious thing.

```
CREATE TABLE Orgy

(boy_name INTEGER NOT NULL

  REFERENCES Boys (boy_name)

  ON DELETE CASCADE

  ON UPDATE CASCADE,

 girl_name INTEGER NOT NULL,

   REFERENCES Girls(girl_name)

   ON UPDATE CASCADE

   ON DELETE CASCADE,

 PRIMARY KEY (boy_name, girl_name)); -- compound key
```

The Orgy table gets rid of the duplicated rows and makes this a proper table. The primary key for the table is made up of two or more columns and is called a compound key because of that fact. These are valid rows now.

('Joe Celko', 'Miley Cyrus')

('Joe Celko', 'Lady GaGa')

('Alec Baldwin', 'Lady GaGa')

But the only restriction on the couples is that they appear only once. Every boy can be paired with every girl, much to the dismay of the traditional marriage advocates. I think I want to make a rule that guys can have as many gals as they want, but the gals have to stick to one guy.

The way I do this is to use a NOT NULL UNIQUE constraint on the girl_name column, which makes it a key. It is a simple key since it is only one column, but it is also a nested key because it appears as a subset of the compound PRIMARY KEY.

```
CREATE TABLE Playboys

(boy_name INTEGER NOT NULL

  REFERENCES Boys (boy_name)

  ON UPDATE CASCADE

  ON DELETE CASCADE,

 girl_name INTEGER NOT NULL UNIQUE, -- nested key

   REFERENCES Girls(girl_name)

   ON UPDATE CASCADE

   ON DELETE CASCADE,

 PRIMARY KEY (boy_name, girl_name)); -- compound key
```

The Playboys is a proper table, without duplicated rows, but it also enforces the condition that I get to play around with one or more ladies, thus.

('Joe Celko', 'Miley Cyrus')

('Joe Celko', 'Lady GaGa')

The ladies might want to go the other way and keep company with a series of men.

```
CREATE TABLE Playgirls
(boy_name INTEGER NOT NULL UNIQUE -- nested key
  REFERENCES Boys (boy_name)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
 girl_name INTEGER NOT NULL,
REFERENCES Girls(girl_name)
   ON UPDATE CASCADE
   ON DELETE CASCADE,
 PRIMARY KEY (boy_name, girl_name)); -- compound key
```

The Playgirls table would permit these rows from our original set.

('Joe Celko', 'Lady GaGa')

('Alec Baldwin', 'Lady GaGa')

Think about all of these possible keys for a minute. The compound PRIMARY KEY is now redundant. If each boy appears only once in the table or each girl appears only once in the table, then each (boy_name, girl_name) pair can appear only once. However, the redundancy can be useful in searching the table because the SQL engine can use it to optimize queries.

The traditional marriage advocates can model their idea of stable couples. With this code.

```
CREATE TABLE Marriages
(boy_name INTEGER NOT NULL UNIQUE -- nested key
  REFERENCES Boys (boy_name)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
 girl_name INTEGER NOT NULL UNIQUE -- nested key,
   REFERENCES Girls(girl_name)
   ON UPDATE CASCADE
   ON DELETE CASCADE,
 PRIMARY KEY(boy_name, girl_name)); -- redundant compound key
```

The Couples table allows us to insert these rows from the original set.

('Joe Celko', 'Miley Cyrus')

('Alec Baldwin', 'Lady GaGa')

Making special provisions for the primary key in the SQL engine is not a bad assumption on the part of vendors because the REFERENCES clause uses the PRIMARY KEY of the referenced table as the default. Many new SQL programmers are not aware that a FOREIGN KEY constraint can also reference any UNIQUE constraint in the same table or in another table. The following nightmare will give you an idea of the possibilities. The multiple column versions follow the same syntax.

```
CREATE TABLE Foo
(foo_key INTEGER NOT NULL PRIMARY KEY,
 ...
 self_ref INTEGER NOT NULL
   REFERENCES Foo(fookey),
 outside_ref_1 INTEGER NOT NULL
    REFERENCES Bar(bar_key),
 outside_ref_2 INTEGER NOT NULL
   REFERENCES Bar(other_key),
 ...);
CREATE TABLE Bar
(bar_key INTEGER NOT NULL PRIMARY KEY,
 other_key INTEGER NOT NULL UNIQUE,
 ...);
```

Overlapping Keys

But getting back to the nested keys, just how far can we go with them? My favorite example is a teacher's schedule. The rules we want to enforce are:

1) A teacher is in only one room each period.

2) A teacher teaches only one class each period.

3) A room has only one class each period.

4) A room has only one teacher in it each period.

Stop reading and see what you come up with for an answer. It is not as easy as you first think. Each UNIQUE constraitn will have overhead, usually an index, so you want to use the fewest possible constraints.

```
CREATE TABLE Schedule -- corrected version
(teacher_name VARCHAR(15) NOT NULL,
 class_title CHAR(15) NOT NULL,
 room_nbr INTEGER NOT NULL,
 period_nbr INTEGER NOT NULL,
 UNIQUE (teacher_name, period_nbr), -- rules #1 and #2
 UNIQUE (room_nbr, period_nbr)); -- rules #3 and #4
```

If a teacher is in only one room each period, then given a period and a teacher I should be able to determine only one room, i.e. room is functionally dependent upon the combination of teacher and period. Likewise, if a teacher teaches only one class each period, then class is functionally dependent upon the combination of teacher and period. The same thinking holds for the last two rules: class is functionally dependent upon the combination of room and period, and teacher is functionally dependent upon the combination of room and period.

Try to imagine enforcing this with procedural code. This is why I say that most of the work in SQL is done n the DDL.

Heuristics

If the list of legal values for an attribute is short and static, then use an CHECK (..)) constraint. For example, the ISO sex code valeus for “unknown”, “male”, “female” and “lawful person”(corporations, organizations, etc) are protected by this declaration.

```
sex_code CHAR(1) DEFAULT '0' NOT NULL

         CHECK (sex_code IN ('0','1','2','9'))
```

If the list of legal values for an attribute is long or dynamic, then use an auxiliary table or look-up table with a reference. Remember to add the CASCADE for the dynamic data.

Earlier, I gave this constraint for the IATA airport codes are the three-letter code which is used in passenger reservation, ticketing, and baggage-handling systems. Here sis full version:

```
airport_code CHAR(3) NOT NULL

 CHECK (airport_code SIMILAR TO '[A-Z][A-Z][A-Z]')

 REFERENCES IATA (airport_code)-- download from Internet

  ON UPDATE CASCADE,
```

The actual list of valid codes changes and it is longer than you might want to put into an IN() predicate.   Instead of, or in addition to this simple CHECK(), you can add a REFERENCES to a look-up or auxiliary table with IATA (International Air Transport Association) airport codes. This table can also have the name, country and the four-letter ICAO (International Civil Aviation Organization) airport code, which is used by air-traffic control systems and for airports that do not have an IATA airport code.

A table always has a PRIMARY KEY on a table. We actually considered requiring this in the Standards, but decided that the then-existing implementations could not handle it; they were built on old file systems based on magnetic tapes and punch cards.

The PRIMARY KEY is the default in the referenced table of DRI actions, but explicitly name those columns. If anything changes, you are safe and it is good documentation.

While NO ACTION is implicit, put it on the REFERENCES clauses for documentation. It leaves a "hook" for later changes.

Remember that a UNIQUE() constraint allows NULLs unless you explicitly add NOT NULL constraints to all the columns.

Remember that (UNIQUE(a) and UNIQUE(b)) is not the same as UNIQUE(a, b).

About the Author

Mr. Joe Celko serves as Member of Technical Advisory Board of Cogito, Inc. Mr. Celko joined the ANSI X3H2 Database Standards Committee in 1987 and helped write the ANSI/ISO SQL-89 and SQL-92 standards. He is one of the top SQL experts in the world, writing over 700 articles primarily on SQL and database topics in the computer trade and academic press. The author of six books on databases and SQL, Mr. Celko also contributes his time as a speaker and instructor at universities, trade conferences and local user groups.