

## 2 Values, Types, Variables, Operators

### 2.1 Introduction

This chapter in the theory book looks at the four fundamental concepts on which most computer languages are based and establishes the related terminology that is used in the book. Here we examine, at a general level, SQL's treatment of those four concepts, but unfortunately SQL doesn't stick with just those four and in fact has two more. One of these is so pervasive in its effects on the other four that we had better start to examine it right away...

#### SQL's Fifth Concept, NULL

Unfortunately, SQL embraces a fifth concept, called NULL, an apparently simple little thing but one that has pervasive effects on our usual understanding of the other four. NULL denotes a kind of nothingness. It is somewhat akin to a value in that it can be the result of evaluating an expression, but it has no type, cannot appear everywhere a value can appear, and lacks certain other properties that values have, as we shall see.



**LIGS University**  
based in Hawaii, USA

is currently enrolling in the  
Interactive Online **BBA, MBA, MSc,**  
**DBA and PhD** programs:

- ▶ enroll **by October 31st, 2014** and
- ▶ **save up to 11%** on the tuition!
- ▶ pay in 10 installments / 2 years
- ▶ Interactive Online education
- ▶ visit [www.ligsuniversity.com](http://www.ligsuniversity.com) to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education. More info [here](#).



Although SQL is not the only computer language to include a special construct representing non-existence, its own variety, NULL is accompanied by a departure from classical logic that is not found in any other well-known languages and gives rise to some deviations from relational database theory that students and users have to be well aware of. The departure in question is the introduction of a third truth value, denoted by the key word UNKNOWN. For example, a comparison of the form  $x = y$  in SQL evaluates to UNKNOWN if either  $x$  or  $y$  evaluates to NULL, even when they both do! (The comparison operator IS NOT DISTINCT FROM is available in place of  $=$ , such that  $x$  IS NOT DISTINCT FROM  $y$  evaluates to FALSE if just one of  $x$  and  $y$  is NULL, TRUE if they both are, and is otherwise equivalent to  $x = y$ .)

I shall reveal as we go along the effects of NULL and UNKNOWN that cause SQL to deviate from normally expected behaviour.

### Historical Note

SQL's NULL and UNKNOWN are generally considered to have their origins in a 1979 paper by E.F. Codd (reference [6]), who was an employee of IBM at that time and thus readily available for consultation by the early SQL DBMS developers in that company. It is perhaps rather unfortunate, then, that these constructs, which have been subject to so much criticism and controversy, thus received a cachet of approval from such an eminent authority. In his famous 1970 paper (reference [3]) Codd was quite clear in requiring every attribute of a tuple contained in a relation to be assigned a value in the “domain” (i.e., declared type) of that attribute. It seems likely (to me, at least) that in 1979 Codd was reacting to critics of his Relational Model of Data who complained that it appeared to offer no good, practical method of dealing with the so-called “problem of missing information”, though an earlier paper by him (reference [4]) does contain a couple of oblique mentions of something called “null”. That criticism might still be to some extent valid. For example, several genuinely relational approaches to the problem that have been advanced are described in reference [11], yet none of these is likely to appeal to those who are beguiled by the superficial attraction of just “leaving it blank”, so to speak.

It is interesting that Codd later rejected his 1979 proposal as being inadequate (rather than as being flawed, as some would have it) and replaced it in 1990 (reference [5]) by a proposal involving two different kinds of null and a fourth truth value. The point was that his original proposal was for a special marker to appear, in place of an attribute value, to denote “a value exists but which particular value really belongs here is not known”. SQL's NULL is indeed used for that purpose but, as we shall see, it is used for other purposes too. One of those other purposes is to signify “no value belongs here”, and that is the additional purpose Codd was addressing in 1990.

As we shall see as we go along, the beguiling simplicity of “just leaving it blank” (i.e., assigning NULL) is more than compensated for—its detractors such as myself would argue—by the complications and difficulties that arise, for example, in defining constraints and expressing queries.

Now, although I have claimed that NULL is a “fifth concept”, the current (2011) SQL standard’s definition of NULL appears to directly contradict that claim, and also to take a different view on my claim that NULL has no type:

Every data type includes a special value, called the *null value*, sometimes denoted by the keyword NULL. This value differs from other values in the following respects:

- Since the null value is in every data type, the data type of the null value implied by the keyword NULL cannot be inferred; hence NULL can be used to denote the null value only in certain contexts, rather than everywhere that a literal is permitted.
- Although the null value is neither equal to any other value nor not equal to any other value — it is *unknown* whether or not it is equal to any given value — in some contexts, multiple null values are treated together; for example, the <group by clause> treats all null values together.

Note the term *the null value*, and that although NULL is called that, it “differs from all other values”. In particular it compares neither equal to itself nor unequal to everything bar itself, except in certain contexts as vaguely noted in the second bullet. IS NOT DISTINCT FROM is one of these exceptions. It, and its converse IS DISTINCT FROM, were added to SQL in 1999 and they remain an optional conformance feature. Note also the notion that there is only one NULL, causing us to wonder how it can be that the NULL that is a member of type INTEGER is the very same thing as the NULL that is a member of type DATE, for example. I stick by my “fifth concept” claim.

I have also mentioned UNKNOWN as the key word denoting SQL’s third truth value. In the light of the text I have just cited, to the effect that NULL is a value of every type in SQL, the question should arise in your mind: what is the difference between UNKNOWN and the NULL that is a member of type BOOLEAN? Here is the official answer (again, as of 2011):

The data type boolean comprises the distinct truth values *True* and *False*... [T]he boolean data type also supports the truth value *Unknown* as the null value. This specification does not make a distinction between the null value of the boolean data type and the truth value *Unknown*...they may be used interchangeably to mean exactly the same thing.

Notice in passing “the null value of the boolean data type”, strongly suggesting that this is not, after all, the very same thing as the null value of the integer data type. Clearly, NULL is a very strange and difficult thing to speak or write about, especially for the painstaking members of the international committee responsible for drafting the SQL standard (and I can attest to that difficulty, having been one of them for 15 years!).

## 2.2 Anatomy of A Command

Figure 2.1, showing a simple SQL command, is almost identical to its counterpart in the theory book. The only difference arises from the fact that SQL uses a different notation for assignment to a local variable. (The SQL international standard doesn't actually use the terms *update operator* and *read-only operator*, but SQL does embrace those distinct concepts and for convenience I shall continue to use those terms in this book.)

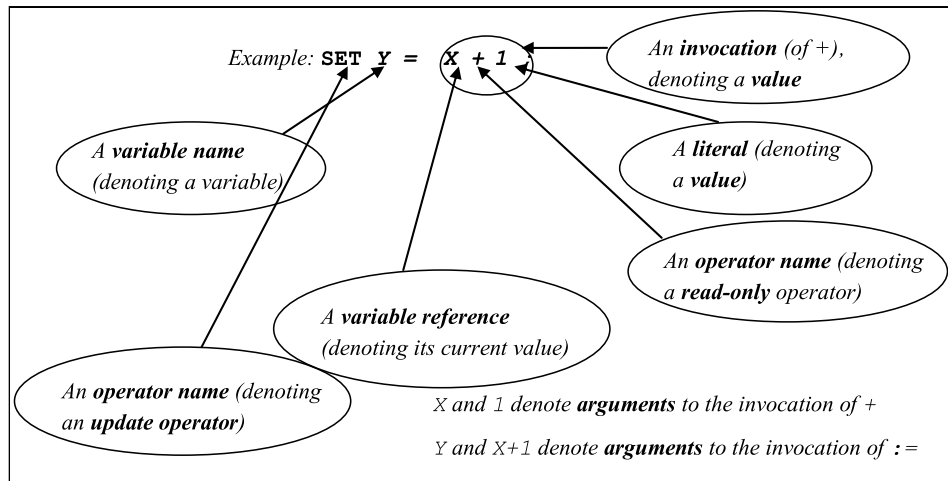



Figure 2.1: Some terminology.

.....Alcatel-Lucent 

[www.alcatel-lucent.com/careers](http://www.alcatel-lucent.com/careers)

What if you could build your future and create the future?

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".



The remaining text of Section 2.2 in the theory book holds good here, subject to the two small syntactic differences, the initial key word `SET` as the operator name for assignment and the use of `=` in place of `:=`.

### Effects of `NULL`

The numeric variable `X`, perhaps of type `INTEGER`, might be assigned `NULL`. In that case the result of evaluating `X + 1` is `NULL`, and so `SET Y = X + 1` assigns `NULL` to `Y`. A curious consequence of this is that immediately following this assignment the comparison `Y = X + 1` does not evaluate to `TRUE`. For that matter, neither does the comparison `Y = Y`. In fact they both evaluate to `UNKNOWN`.

## 2.3 Important Distinctions

Again the theory book text for this section holds good here too. The list of important distinctions is repeated here. You might wish to check your own understanding of them before re-reading the text in the theory book.

- Syntax versus semantics
- Value versus variable
- Variable versus variable reference
- Update operator versus read-only operator
- Operator versus invocation
- Parameter versus argument
- Parameter subject to update versus parameter not subject to update

## 2.4 A Closer Look at a Read-Only Operator (+)

In the theory book I equate the term *read-only operator* to the mathematical term *function*. Here I just need to add that the SQL standard reserves the term *function* for read-only operators that are invoked using prefix notation: an operator name followed by a commalist of argument expressions enclosed in parentheses, as in for example `SQRT(2)`, denoting the positive square root of 2, or `SUBSTRING(X FROM 1 FOR 2)` denoting the first two characters of the character string denoted by `X`. The term *operator* is used for those invoked using infix notation, like `+` for example.

## 2.5 Read-only Operators in SQL

Like **Tutorial D** and many other computer languages, SQL distinguishes between *system-defined* (or *built-in*) operators and *user-defined* operators.



I do not give a list of SQL's system-defined operators. For one thing it would be too overwhelming for present purposes. For another, it would inevitably be incomplete sooner or later, as the SQL standard is revised every few years and implementers are naturally inclined to add new ones on demand. We will meet some of them when we need them for illustrative purposes.

Section 2.5 of the theory book gives Example 2.1, defining in **Tutorial D** an operator named `HIGHER_OF` to give the value of whichever is the higher of two given integers. Here I simply translate that definition into SQL.

**Example 2.1: A User-Defined Operator in SQL**

```
CREATE FUNCTION HIGHER_OF ( A INTEGER, B INTEGER )
    RETURNS INTEGER
IF A > B THEN RETURN A ;
    ELSE RETURN B ;
END IF ;
```

*Points to note:*

- the key word `FUNCTION` in place of **Tutorial D**'s `OPERATOR`;
- some minor differences concerning semicolons;
- the lack of a counterpart to **Tutorial D**'s `END OPERATOR` (which some observers have criticised as being redundant);
- above all, the initial key word, `CREATE-SQL` uses this for all sorts of database objects that users can create. For example, `CREATE TABLE` is SQL's counterpart of **Tutorial D**'s `VAR ... BASE RELATION` for creating a database relvar, and `CREATE PROCEDURE` is for creating a user-defined update operator.

Example 2.1 doesn't illustrate all of the features in SQL connected with user-defined operator definitions. For example, the code that implements the operator can be written separately, in a language other than SQL. Also, the key word `FUNCTION` can be replaced by some special syntax to denote a special kind of function referred to as a *method* (as used in several object-oriented programming languages), but that and certain other optional ingredients are beyond the scope of this book.

### Effects of NULL

As a general rule—but not a universal one—if NULL is an argument to an invocation of a system-defined read-only operator, then NULL is the result of that invocation. As you can see, the code for HIGHER\_OF includes `A > B`, an invocation of the system-defined read-only operator “>”, which does follow the general rule. Hence, if NULL is substituted for either or both of the parameters A and B, then NULL—which in this case we can also call UNKNOWN because “>” is a Boolean operator—is the result of the invocation. You are perhaps now wondering how SQL handles the IF statement when the specified condition yields UNKNOWN: is the THEN clause evaluated, or is it the ELSE clause?

As you know, other programming languages are normally based on classical logic. In keeping with the existence of just two truth values, TRUE and FALSE, the syntax for IF statements (and IF expressions) in such languages has just the two forks, THEN for when the condition is TRUE, ELSE for when it is not (i.e., is FALSE). You might therefore reasonably expect a language that embraces  $n$  truth values to support a variety of IF that has  $n$  forks—under a language design principle that Fred Brooks in reference [1] referred to as *conceptual integrity*, which means adhering rigorously to the language’s adopted concepts. Instead, SQL retains just the two forks, keeping the normal treatment of THEN as being the one for when the condition is TRUE and arbitrarily lumping UNKNOWN in with FALSE for the ELSE fork.

You should now be able to see that the general rule (“NULL in, NULL out”) for system-defined operators cannot be said to apply to user-defined ones. If `A > B` evaluates to UNKNOWN, then the result of the HIGHER\_OF invocation is the argument substituted for B, which might or might not be NULL.

If we wanted to make HIGHER\_OF adhere to “NULL in, NULL out”—let’s call it the NiNo rule—we would have to write something like what is shown in Example 2.1a.

#### Example 2.1a: NiNo version of HIGHER\_OF

```
CREATE FUNCTION HIGHER_OF ( A INTEGER, B INTEGER )
    RETURNS INTEGER
CASE
    WHEN A IS NULL OR B IS NULL
        THEN RETURN CAST ( NULL AS INTEGER ) ;
    WHEN A > B
        THEN RETURN A ;
    ELSE RETURN B ;
END CASE ;
```

**Explanation 2.1a:**

- **IS NULL** is a monadic Boolean operator that evaluates to TRUE when its argument is “the null value”, otherwise FALSE. Note, therefore, that it is our first exception to the NiNo rule, which would require it to evaluate to NULL (UNKNOWN) when its argument is “the null value”.
- **CAST ( NULL AS INTEGER )** denotes “the null value” of type INTEGER. As in **Tutorial D**, the operand of RETURN must be an expression that has a declared type and NULL, on its own, does not have a declared type. The CAST operator takes an expression and a type name, separated by the “noise” word AS, and normally expresses a “type conversion”—a function that maps elements of one type to those of another that are considered to be in some defined sense equivalent. In the special case of NULL it is used to confer a type, so to speak, on something that doesn’t otherwise have one.
- **AS**, appearing where you might have expected just a comma, is explained by a matter of policy in SQL whereby invocations of user-defined functions, which always use commas between arguments, can be syntactically distinguished from invocations of system-defined functions.
- **OR** is SQL’s counterpart of the usual logical operator of that name. In `A IS NULL OR B IS NULL` its operands are clearly restricted to just TRUE and FALSE and that expression therefore results in TRUE if either of those operands does, otherwise FALSE. I deal with the treatment of UNKNOWN in SQL’s counterparts of the logical operators in Chapter 3, “Predicates and Propositions”.

**Indeterminacy in SQL**

Some SQL expressions are actually not function invocations at all in the mathematical sense, being indeterminate—invocations operating on identical input do not always yield the same value. The indeterminate expressions are among those that the standard defines as *possibly non-deterministic*, but—*caveat lector*—not all expressions defined as possibly non-deterministic are actually indeterminate from a mathematical viewpoint. For example, the key word `USER` denotes the userid (officially, the *authorization identifier*) of the session in which an expression containing that key word is evaluated. Such an expression is defined as possibly non-deterministic by virtue of the appearance of `USER`, even though invocations in different sessions, in which `USER` stands for different userids, are clearly different invocations. For a more general example, a user-defined function can be explicitly declared as either `DETERMINISTIC` or `NOT DETERMINISTIC`. In the latter case the “function” is flagged such that all invocations of it are treated as possibly non-deterministic.



Regardless of the appropriateness of the term non-deterministic, there is a good reason for categorizing references to the current user (or, for another example, the current time) along with genuine cases of indeterminacy. It concerns constraint declarations. We clearly want to outlaw a constraint condition whose result, when evaluated, depends on the properties of the session in which, or on the time at which, the evaluation takes place. Of course we must also outlaw genuinely indeterminate conditions—a database might satisfy such a condition but later fail to satisfy it even though the database has not been updated in the meantime! The question then arises as to how it is possible for indeterminacy to arise: surely a computer program always gives the same result when invoked with the same input?



**Maastricht University** *Leading in Learning!*

**Join the best at the Maastricht University School of Business and Economics!**

**Top master's programmes**

- 33<sup>rd</sup> place Financial Times worldwide ranking: MSc International Business
- 1<sup>st</sup> place: MSc International Business
- 1<sup>st</sup> place: MSc Financial Economics
- 2<sup>nd</sup> place: MSc Management of Learning
- 2<sup>nd</sup> place: MSc Economics
- 2<sup>nd</sup> place: MSc Econometrics and Operations Research
- 2<sup>nd</sup> place: MSc Global Supply Chain Management and Change

Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012

**Visit us and find out why we are the best!**  
**Master's Open Day: 22 February 2014**

**Maastricht University is the best specialist university in the Netherlands**  
(Elsevier)

[www.mastersopenday.nl](http://www.mastersopenday.nl)



One root cause of indeterminacy in SQL lies in its implementation of comparison for equality. For certain system-defined types it is possible for distinct values to compare equal (note the contradiction). One such type is CHARACTER, described later in Section 2.6. Like COBOL, SQL ignores trailing “pad characters” when comparing character strings. The pad character is normally the space obtained by depressing the space bar on a keyboard. Thus, for example, the comparison 'SQL' = 'SQL ' evaluates to TRUE, even though `CHAR_LENGTH('SQL') = CHAR_LENGTH('SQL ')`, comparing the lengths of those two strings, 3 and 6, evaluates to FALSE. Now consider the relational projection of ENROLMENT over just its Name attribute: `ENROLMENT{Name}` in **Tutorial D**. As we shall see in Chapter 4, SQL has a counterpart of projection, but suppose the two rows for student S1 in the ENROLMENT table had 'Anne' and 'Anne ' for S1's name. If both of those values were to appear in the result, that would be inconsistent with the fact that they compare equal in SQL. If just one of them appears, then which one? The SQL standard declares such an expression to be possibly non-deterministic and permits a conforming implementation to give any value that compares equal to 'Anne'—possibly one that doesn't even appear in the table—and does not require it to give the same value every time the expression is evaluated. As a consequence, there are several SQL operators whose use on character strings is not permitted to appear in constraint declarations.

The SQL standard lists a multitude of conditions that cause an expression to be defined as possibly non-deterministic. Perhaps the most alarming is the assumption that equals comparison of values of user-defined types is assumed to suffer from the same problem as I have described for character strings: it is assumed that distinct values can compare equal, even if the type definition is such that this cannot possibly be the case.

### Historical Note

Support for user-defined operators, along with a programming language for their code, didn't arrive in the SQL standard until 1996, 17 years after the appearance of the first commercial SQL product. Before that time some products had provided such support as proprietary (i.e., nonstandard) extensions. Even though those products may have later adopted the standard syntax, they will not have abandoned their nonstandard syntax and so these products may suffer from undesirable complications and redundancy as a result.

It is interesting to note that the designers of Business System 12 (reference [8]) recognised the need for user-defined operators, and a programming language to write them in, in 1978, before the first SQL product hit the scene. It's perhaps a pity that their colleagues in the System R team didn't take a closer look at Business System 12, considering that there was some contact between those two teams in the 1970s. In fact, the same requirement had been recognized even earlier by the designers of PRTV (reference [17]) at IBM UK's Scientific Centre in Peterlee, who were consulted by the Business System 12 team and also in contact with the System R team.

PRTV stands for “Peterlee Relational Test Vehicle”, a notably unglamorous and unassuming title. In fact it was based on an earlier prototype—possibly the very first implementation of Codd’s ideas—developed at the same centre back in 1971 with the much more impressive name, IS/1. The switch to the more modest name was dictated by IBM, as they did not wish to appear at all committed to the relational model at that time.

## 2.6 What Is a Type?

SQL’s concept does not differ significantly from that defined in the theory book, apart from that business concerning NULL. However, the theory book equates *type* with the term *domain* used in much of the relational database literature. SQL is at odds with this equation because it uses *domain* for a defined subset of a given type that is not itself a type. For example, the domain WEEKDAY might be defined to consist of the values 'Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', and 'Saturday', but the declared type of a column defined on that domain is that on which the domain itself is defined, perhaps VARCHAR(9). Also, whereas a domain is defined by specifying a constraint (on some underlying type), a constraint cannot be used to specify a user-defined type.



> **Apply now**

REDEFINE YOUR FUTURE  
**AXA GLOBAL GRADUATE  
PROGRAM 2015**

redefining / standards 

agence.cdg. © Photonistop



The system-defined types of SQL at the time of writing are:

- CHARACTER or, synonymously, CHAR, for character strings. When this type is to be the declared type of something (e.g., a column), the permissible values are further constrained by a maximum length specification given in parentheses and optionally by the key word VARYING, indicating that values shorter than the maximum, including the empty string ' ', are also permissible. Examples: CHAR(5) for strings of five characters only, CHARACTER VARYING(100), which can be abbreviated to VARCHAR(100), for strings of up to one hundred characters. The last two are alternative spellings for the same declared type. The type CHARACTER LARGE OBJECT, or CLOB, allows for inclusion of strings that are longer than the longest supported by the other CHARACTER types. Note that in the terminology of the theory book CHARACTER is a kind of *type generator*. The key word does not of itself denote a type, but only does so when qualified by a length specification. A similar remark applies to some of the other type names used in SQL.
- DECIMAL, NUMERIC, REAL, FLOAT and various other terms for various sets of rational numbers. When these key words are specified for the declared type of something, they are usually accompanied by *precision* and *scale* specifications. DECIMAL and NUMERIC are synonymous and are used for types with uniform scales, where the difference between any number and its immediate successor or predecessor is constant. For example, DECIMAL(5, 2) consists of rational numbers with a precision of 5 (i.e., having at most 5 decimal digits) and a scale of one hundredth (i.e., 2 places of decimals), such as 372.57 and -1.00. (Note that SQL always uses the period “.” as the decimal separator.)
- INTEGER or, synonymously, INT, for integers within a certain range. SQL additionally has types SMALLINT and BIGINT for certain ranges of integers. The exact sets of values denoted by these type names are not specified in the SQL standard, which states merely that “[t]he precision of SMALLINT shall be less than or equal to the precision of INTEGER, and the precision of BIGINT shall be greater than or equal to the precision of INTEGER”.
- TIMESTAMP for values representing points in time on a specified uniform scale. DATE is used for timestamps on a scale of one day, such as DATE '2012-09-25' (a literal denoting September 25th, 2012). TIME is used for values denoting time of day only, such as TIME '16:30:00' (a literal denoting half-past four in the afternoon).
- INTERVAL for values denoting, not intervals (!) but durations in time, such as 5 years, 3 days, 2 minutes, and so on.
- BOOLEAN, consisting of the values TRUE and FALSE, with UNKNOWN being an alternative name to NULL for “the null value” of this particular type.

- `BINARY LARGE OBJECT` for arbitrarily large bit strings.
- `XML` for XML documents and fragments.
- `ARRAY` types for arrays.
- `ROW` types and `MULTISET` types as described later in this Chapter.

At this point, having previously introduced `NULL` as SQL's "fifth concept", I need to mention SQL's sixth: *pointers*. Like `NULL`, pointers are expressly excluded from relational database theory and in fact the difficulty they give rise to was one of Codd's stated motivations for his Relational Model. A pointer is a value that somehow identifies a repository, such as a variable for example, where some other value is stored. The object identifiers (oids) used in object-oriented programming languages are pointers of a kind. SQL's so-called *REF values* are another and so are its so-called *datalinks*. A `REF` value points to a row in a base table and is otherwise rather like an oid in the operations that can be performed on it. A *datalink* is a url, allowing SQL database management to be synchronized with management of related objects that are strictly outside of the database. To complete the list of SQL's system-defined types I therefore need to add:

- `REFERENCE` types for `REF` values. A `REFERENCE` type comes into existence automatically when a certain kind of user-defined type is defined, as I describe in Section **2.10, Types and Representations**.
- `DATALINK` for *datalinks*. I say no more about *datalinks* in this book.

### Historical Note

The `CHARACTER` and numeric types have been in SQL since the appearance of the first commercial products. The same is true of truth-valued expressions but `BOOLEAN` did not become a "first-class" type in the SQL standard until 1999 and it remains an optional conformance feature. (In common parlance, a type is "first-class" only if it has a name and can thus be used for any of the purposes described in Section 2.7, **What Is a Type Used For?**, in the theory book.) `TIMESTAMP` types and `INTERVAL` types arrived in standard SQL in 1992. `LARGE OBJECT` types, `DATALINK`, `ARRAY` types, user-defined types, and `REF` types first appeared in the 1999 edition. Type `XML` and `MULTISET` types were introduced in the 2003 edition.

Rows and tables obviously appeared in the original SQL products but `ROW` types as first-class types didn't appear until 1999. Table types became almost first-class with the introduction in SQL:2003 of `MULTISET` types, as described in Section 2.8, **The Type of a Table**, but SQL:2011 still does not support `TABLE` types *per se*. `ROW` types and `MULTISET` types are both optional conformance features.

## 2.7 What Is a Type Used For?

In SQL, as in most computer languages, a type can be used for *constraining* the values that are permitted to be used for some purpose. In particular, it can be used for constraining:

- the values that can be assigned to a variable
- the values that can be substituted for a parameter
- the values that an operator can yield when invoked
- the values that can appear in a given column of a table

In each of the above cases, the type used for the purpose in question is the *declared type* of the variable, parameter, operator, or column, respectively. As a consequence, every expression denoting a value has a declared type too, whether that expression be a literal, a reference to a variable, parameter, or column, or an invocation of an operator. Thus SQL is able to detect errors at “compile time”—by mere inspection of a given script—that would otherwise arise, and cause much more inconvenience, at run time (when the script is executed).

Unfortunately that’s not the end of the type story in SQL. SQL adds a fifth use for types, though this one is not available with all types but only for one of its two kinds of user-defined types, namely, the so-called *structured types*. A structured type is defined using something similar to **Tutorial D**’s “possrep” construct. Thus, type POINT might be defined in SQL in terms of a representation consisting of components X and Y, each of type REAL. Such a type can be used for any of the four purposes already listed but it can also be used as a basis for defining a base table (or view), consisting of columns corresponding to the components of its structure, plus one further column whose declared type is a REFERENCE type (REFERENCE ( POINT ) if POINT is the name of the structured type). The REFERENCE column is a relational key for the table thus defined, which is variously called a *referenceable table* and a *typed table* (the two terms are synonymous). The REF values can be used in the way object identifiers are used in object-oriented languages.

### Effects of NULL

Although an SQL type does have those constraining purposes, it must be remembered that in each case NULL might appear in place of a true value of the type in question, except where explicit measures are taken to avoid such appearances. Appearance of NULL in a column of a base table can easily be avoided by specifying NOT NULL in the column definition, but no such declaration is available in connection with local variables, parameters, or results of operator invocations.



### Historical Notes

Support for local variables didn't arrive in SQL until 1996, when support for user-defined operators first appeared. Until that time "the values that can be assigned to a variable" applied only to base tables (SQL's counterparts of base relvars), "the values that can be substituted for a parameter" applied only to parameters of system-defined operators, and "the values that an operator can yield" similarly applied only to system-defined operators.

Support for user-defined types, including the structured types mentioned in this section, was added in 1999, as part of the so-called "object-relational" support that was billed as the main theme of SQL:1999.

## 2.8 The Type of a Table

This section in the theory book is headed **The Type of a Relation** and it uses its running example, repeated again here as Figure 2.3 as a basis for its discussion.

StudentId	Name	CourseId
S1	Anne	C1
S1	Anne	C2
S2	Boris	C1
S3	Cindy	C3
S4	Devinder	C1

Figure 2.3: Enrolments again

**Empowering People. Improving Business.**

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

[www.bi.edu/master](http://www.bi.edu/master)

**BI NORWEGIAN BUSINESS SCHOOL**

EFMD **EQUIS** ACCREDITED



As a relation in **Tutorial D** the type name for the value shown in Figure 2.3 can be written as

```
RELATION { StudentId SID, Name NAME, CourseId CID }
```

or, equivalently (for recall that there is no ordering to the elements of a set),

```
RELATION { Name NAME, StudentId SID, CourseId CID }
```

(and so on).

Of course Figure 2.3 could also be illustrating some SQL table, but SQL has no names for table types, so no counterparts of relation type names. It does, however, have names for row types and in fact the standard uses term *row type* as a property of a table—it is of course the type of each of the rows in the table, though it is defined even when the table is empty.

A row type name is the key word ROW followed by a commalist of field name/type name pairs in similar style to the attribute name/type name pairs in **Tutorial D** except that the commalist is enclosed in parentheses rather than braces (SQL uses the term *field* for the components of a row). It would be nice, then, if we could say that the row type of our enrolments table was

```
ROW ( StudentId SID, Name NAME, CourseId CID )
```

or, equivalently,

```
ROW ( Name NAME, StudentId SID, CourseId CID )
```

but there are two important reasons why we can't. First, those two type names would in fact define different types in SQL, because the order of the elements is significant. It follows that the order of columns in an SQL table is significant, contrary to one of the important precepts you have learned in relational theory, and we will discover various complications that arise from this significance as we go along. Secondly, even if type names SID, NAME, and CID are defined, the types so named cannot be equivalent to their **Tutorial D** counterparts, for reasons given in Section 2.10, **Types and Representations**. The row type of our enrolments table is therefore more likely to be something like:

```
ROW ( Name VARCHAR(50), StudentId VARCHAR(5),
      CourseId VARCHAR(5) )
```

SQL's row types exhibit one further deviation from relational theory: there is no such type as ROW ( ). It follows that every table in SQL has at least one column. TABLE\_DEE and TABLE\_DUM, defined in Chapter 4 of the theory book, do not exist in SQL.

**MULTISET types**

An SQL *multiset* is what in mathematics is also known as a *bag*—something like a set except that the same element can appear more than once. The body of an SQL table is in general a bag of rows, rather than a set of rows, because SQL does indeed permit the same row to appear more than once in the same table. Although SQL has no names for table types, it does support multisets in general and it does have names for multiset types. A multiset type name consists of a type name followed by the key word `MULTISET`. For example, `INTEGER MULTISET` is the name of the type each of whose values is either (a) a bag, consisting of zero or more appearances of each value of type `INTEGER` and zero or more appearances of the null value of type `INTEGER`, or (b) the null value of type `INTEGER MULTISET`.

It would seem at first glance, then, that we perhaps do have a type name for a table type after all. For example, our enrolments table could perhaps be of type

```
ROW ( Name VARCHAR(50), StudentId VARCHAR(5),
      CourseId VARCHAR(5) ) MULTISET
```

In fact one could declare a *local* variable to be of this type and its value could indeed consist of the rows shown in Figure 2.3. However, such a type cannot be the declared type of a base table, in spite of the fact that the elements of a base table are indeed rows of the same type. Moreover, as I have already mentioned, there is such a thing as the null value of that multiset type, whereas `NULL` can never appear in place of a table—no table expression in SQL can ever evaluate to `NULL`—nor can `NULL` appear in place of a row in a table. So the set of values of a multiset type whose element type is a row type includes bags that are not tables as well as bags that are.

**2.9 Table Literals**

One might expect SQL to support table literals in the manner illustrated in Example 2.2, but in fact that is not a legal SQL expression.

**Example 2.2: Not a Table Literal**

```
TABLE (
  ROW ( StudentId 'S1', CourseId 'C1', Name 'Anne' ),
  ROW ( StudentId 'S1', CourseId 'C2', Name 'Anne' ),
  ROW ( StudentId 'S2', CourseId 'C1', Name 'Boris' ),
  ROW ( StudentId 'S3', CourseId 'C3', Name 'Cindy' ),
  ROW ( StudentId 'S4', CourseId 'C1', Name 'Devinder' )
)
```

It is illegal because row literals in SQL do not use column names. Instead, the column values must be written in the appropriate order, reflecting the order of the columns of the table, as in

```
ROW ( 'S1', 'C1', 'Anne' )
```

Moreover, the word `VALUES` is used in place of `TABLE`, parentheses are not used around the list of row literals, and the key word `ROW` is in fact optional, so that the most common form is that shown in Example 2.3.

**Example 2.3:** A Table Literal (correct version)

```
VALUES
( 'S1', 'C1', 'Anne'      ),
( 'S1', 'C2', 'Anne'      ),
( 'S2', 'C1', 'Boris'     ),
( 'S3', 'C3', 'Cindy'     ),
( 'S4', 'C1', 'Devinder'  )
```

Now, the question arises, what is the (table) type of the table shown in Example 2.3? For that matter, what is the (row) type of ( 'S1', 'C1', 'Anne' )? In particular, what are the field names of those three fields, which would become column names for the containing table? The short answer is that they are determined by the context in which the expression appears. Because the components are distinguished anyway by ordinal position, the field names sometimes serve little or no purpose. In fact several fields are permitted to acquire the same name. Also, sometimes the context does not provide any names at all, in which case, according to the standard, each field is assigned a unique but unpredictable name. Examples arising as we go along will make this issue a little clearer. I shall use the term *anonymous column* to refer to a column whose name is unpredictable and therefore effectively undefined.

Note carefully that if the word `ROW` is omitted and the row consists of a single field, then the parentheses can also be omitted. Thus, `VALUES 'S1'` denotes a table consisting of a single column and a single row, the SQL counterpart of `RELATION { TUPLE { StudentId 'S1' } }` (though the SQL counterpart has nothing corresponding to the attribute name). Furthermore, recall from Chapter 1, Section 1.15 **Updating Variables** that `VALUES 'S1', 'S2'` denotes a table consisting of a single column and two rows—not a single row and two columns!

The declared type of a `VALUES` expression is determined by that of its contained row expressions and cannot be explicitly stated. As a consequence, SQL has no literals for empty tables—no counterparts of **Tutorial D**'s expressions such as `RELATION { StudentId SID, Name CHAR, CourseId CID } { }`, where heading and body are both specified explicitly.

### Effects of NULL

When a `VALUES` expression appears as the source value for an SQL `INSERT` statement, the key word `NULL` can appear as a field value, such that for example `INSERT INTO ENROLMENT VALUES ( 'S1', NULL, 'C1' )` is permitted as short for `INSERT INTO ENROLMENT( StudentId, Name, CourseId ) VALUES ( 'S1', CAST ( NULL AS VARCHAR(50) ), 'C1' )`. (I omitted the column name list in the short form just to remind you of the significance of column order in SQL.)

### Historical Note

The history surrounding `VALUES` expressions is described in the historical notes for Section 1.15, [Updating Variables](#), in Chapter 1.

## 2.10 Types and Representations

This section in the theory book introduces the concept *possible representation*, abbreviated *possrep*, and explains how these can be used in conjunction with constraints to define types. Example 2.4 from that book is copied here, along with its explanation.

## Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now



Go to [www.helpmyassignment.co.uk](http://www.helpmyassignment.co.uk) for more info



Helpmyassignment



**Example 2.4:** A Type Definition

```

TYPE SID POSSREP SID { C CHAR
                        CONSTRAINT LENGTH(C) <= 5
                        AND
                        STARTS_WITH(C, 'S')
                        AND
                        IS_DIGITS(SUBSTRING(C,1))
                        } ;

```

**Explanation 2.4:**

- **TYPE SID** announces that a type named SID is being defined to the system.
- **POSSREP SID** announces that what follows, in braces, specifies a possible representation for values of the type. It means that the operators defined for type SID behave as if values of type SID were represented that way, regardless of how they are physically represented “under the covers”. That is why we use the word “possible”—the values might possibly be represented internally that way (but they don’t have to be and we don’t even know if they are). In this case the name of that possrep is the same as the type name, SID (as it would be if we omitted the name). (**Tutorial D** allows more than one possrep to be given for the same type, but this feature is beyond the scope of this book. I do not deal with types in any depth. It is sufficient for present purposes just to understand how they exist, what they are for, and how to use them.)
- **C CHAR** defines the first and only component of the possrep, naming it C and specifying CHAR as its declared type. This definition causes an operator, `THE_C`, to come into existence. `THE_C` takes a value, *s*, of type SID and returns the value of the C component of *s* under the possible representation SID.
- **CONSTRAINT** announces that the expression following it (up to but excluding the closing brace) is a condition that must be satisfied by all possrep values that do indeed represent values of type SID. Note that the expression itself uses the logical connective AND, with its usual meaning, to connect three expressions, two of which are *comparisons* and each of which is a *truth-valued* expression—one that, when evaluated, yields either TRUE or FALSE.

The operators `LENGTH`, `STARTS_WITH`, `SUBSTRING`, and `IS_DIGITS`, invoked in the constraint expression, are not defined as built-in operators in **Tutorial D**. I am assuming their existence as user-defined operators. Happily, their definitions are contained in the file `OperatorsChar.d` included in the download package for *Rel*.

- **LENGTH(C) <= 5** expresses a rule to the effect that the total length of a value for the C possrep component must never exceed 5.



- **STARTS\_WITH(C, 'S')** returns TRUE if and only if the string given as the first operand starts with the string given as the second operand—in this case the string consisting of just the capital letter S.
- **SUBSTRING(C, 1)** denotes the string consisting of the whole of the value of the C posrep component apart from the first character. This is given as the argument to an invocation of **IS\_DIGITS**, which takes a string and yields TRUE if every character in the given string is a numeric digit, otherwise FALSE.

Although SQL has more than one way of defining user-defined types, it has no true counterpart of **Tutorial D**'s **TYPE** statement. We will examine three attempts to emulate Example 2.4 in SQL, showing in each case where the attempt falls short. To avoid falling out of synch with the theory book's example numbers, these three are labelled 2.4a, 2.4b, and 2.4c.

**Example 2.4a:** First attempt at defining type SID in SQL

```
CREATE TYPE SID AS ( C VARCHAR(5) ) ;
```



**Brain power**

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations. Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.  
Visit us at [www.skf.com/knowledge](http://www.skf.com/knowledge)

**SKF**



**Explanation 2.4a:**

- **TYPE SID** announces that a type named SID is being defined to the system.
- **AS ( C VARCHAR(5) )** defines SID as a *structured type*, whose values are represented as a structure consisting in this case of just a single *attribute*, named C, of type VARCHAR(5). (The term *attribute* here is not to be confused with its use in relational theory.) The attribute definition C VARCHAR(5) causes an operator to come into existence that takes a value, s, of type SID and returns the value of the C component of s. The operator, SQL's counterpart of THE\_C, is invoked using dot notation: s.C and is termed the *observer function* for the component C.

SQL does not have an immediate counterpart of **Tutorial D's selector**. Instead, a so-called *constructor function*, in this case a niladic operator named SID, is generated by the structure definition, such that SID() denotes the value of type SID whose only component is the “default value” for the attribute C, probably NULL. To emulate **Tutorial D's SID selector** we need to use the so-called *mutator function* for the attribute C, which is also invoked using dot notation: SID().C('S1'). The mutator function takes a value of type SID as its left operand and a value of the declared type of C as its right operand (in parentheses). In general, if s is a value of type SID, then s.C('S1') denotes the SID value that is obtained from s by replacing its C component by the string 'S1'. If s had any other components (it doesn't, of course), they would be retained in s.C('S1'). By the way, don't be misled by the term “mutator”: an SQL mutator function is a read-only operator.

Note that although we can restrict the length of the character strings as in Example 2.4, we cannot apply those further constraints concerning the string value. That explains how defining a structured type for student identifiers falls short of emulating Example 2.4.

It is probably unusual in SQL for the structure of a structured type to consist of a single attribute. One would more likely decide to define a *distinct type*, as shown in Example 2.4b.

**Example 2.4b:** Second attempt at defining type SID in SQL

```
CREATE TYPE SID AS VARCHAR(5) ;
```

**Explanation 2.4b:**

- **TYPE SID** announces that a type named SID is being defined to the system.
- **AS VARCHAR(5)** defines SID as a *distinct type*, whose values are represented by values of type VARCHAR(5). In this form of type definition the given representation must be a system-defined type.

Under this definition, `SID( 'S1' )` is exactly equivalent to the same expression in **Tutorial D**, and the SQL expression `CAST( s AS VARCHAR( 5 ) )`, where *s* is a value of type `SID`, is equivalent to **Tutorial D**'s `THE_C( s )`. However, as in Example 2.4a, we have no way of further constraining the string values representing student identifiers; so `SID( '34x.1' )`, for example, also denotes a value of type `SID`, as does `SID( '' )`.

Finally, we could try defining a domain, as in Example 2.4c.

**Example 2.4c:** Third attempt at defining type `SID` in SQL

```
CREATE DOMAIN SID AS VARCHAR( 5 )
CHECK ( VALUE IS NOT NULL AND
        SUBSTRING( VALUE FROM 1 FOR 1 ) = 'S' AND
        CAST( '+' || SUBSTRING( VALUE FROM 2 ) AS INTEGER ) >= 0 );
```

**Explanation 2.4c:**

- **DOMAIN SID** announces that a *domain* named `SID` is being defined to the system.
- **AS VARCHAR( 5 )** specifies that values in domain `SID` are certain values of type `VARCHAR( 5 )`.
- **CHECK ( ... )** specifies a constraint defining exactly which values of type `VARCHAR( 5 )` are in the domain `SID`. Note that the key word `VALUE`, which is available only in domain constraints, refers (in this particular example) to an arbitrary value of type `VARCHAR( 5 )`.
- **VALUE IS NOT NULL** specifies that the null value of type `VARCHAR( 5 )` is not a value in the domain. This is needed because the other conjuncts evaluate to `UNKNOWN` if `VALUE` is the null value and a domain constraint is deemed to be violated only when it evaluates to `FALSE`.
- **SUBSTRING( VALUE FROM 1 FOR 1 ) = 'S'** specifies that every value in the domain must begin with `S`. Note SQL's deliberate use of “noise” words in the invocation of `SUBSTRING`, the justification for which is to distinguish invocations of system-defined operators from those of user-defined ones.
- **CAST( '+' || SUBSTRING( VALUE FROM 2 ) AS INTEGER ) >= 0** is an attempt to emulate an invocation of `IS_DIGITS`, perhaps showing how a user-defined operator of that name might be implemented in SQL. The character “+” is concatenated to the putatively numeric portion of the string in order to exclude values such as `'S+123'` from the domain (the string `'+123'` can be cast as an integer but `'++123'` cannot).

So the domain `SID` gives us the required values, but they are values of the system-defined type `VARCHAR(5)`, not a user-defined type. Thus, the operators defined for these values are exactly those defined for character strings in SQL. Moreover, as SQL does not allow domain names to be given as declared types of parameters or operators, we cannot define any special operators for student identifiers. So defining `SID` as a domain also falls short of emulating Example 2.4.

The syntax for `CREATE TYPE` includes a plethora of optional extras that are not illustrated in Examples 2.4a and 2.4b. Most of them are beyond the scope of this book but it is worth mentioning the `UNDER` clause, whereby a structured type can be specified as a subtype of another structured type. Note first that subtyping is available only with structured types, so in SQL we cannot define, for example, type `POSINT` (positive integers) as a subtype of `INTEGER`. Note also that SQL's model of subtyping is quite different from **Tutorial D's** and is more akin to that found in object-oriented languages, whereby a subtype actually “extends” its supertype by adding extra values (so to speak), rather than being a proper subset of it as specified by a declared constraint. The theory book doesn't have much to say about models of subtyping because relational database theory is generally considered to be independent of type theory, in the sense that it is silent with regard to which types are permitted as declared types of attributes of relations.

### Effect of `NULL`

One effect has already been seen in Example 2.4c, where the declared constraint has to explicitly rule out `NULL` from the domain. In Example 2.4a `SID()` denotes the `SID` value whose `C` component is “the null value of type `VARCHAR(5)`”, but note carefully that `SID() IS NULL` evaluates to `FALSE`—it does not denote “the null value of type `SID`”! And yet that null value does exist in SQL, being denoted by `CAST(NULL AS SID)`. By contrast, in Example 2.4b, where `SID` is a distinct type as opposed to being a structured type, `SID(CAST(NULL AS VARCHAR(5)))` is indeed “the null value of type `SID`”.

### An Example of Type versus Representation Confusion in SQL

The theory book describes how a value might have two or more distinct representations. For example, user-defined type `POINT` might have a declared `possrep` based on Cartesian coordinates and another based on polar coordinates. SQL has a system-defined type, `TIMESTAMP`, for values representing points in time, a timestamp being represented by a date, a time of day, and—optionally—a time zone, expressed as a displacement from UTC. Clearly any timestamp can be expressed in several different ways, using different time zones. Three o'clock in the afternoon of December 31st, 2011 in UK, for example, is the same time as two o'clock in the morning of January 1st, 2012 in New Zealand.

SQL treats those two representations as distinct values that compare equal, in like fashion to its treatment of character strings with trailing blanks mentioned in Section 2.5 under the heading **Indeterminacy in SQL**, with similar consequences. If instead it had treated them as distinct representations of the same value, then the issue of indeterminacy would not have arisen.

### Historical Note

`CREATE DOMAIN` was added to the SQL standard in 1992 but it is still an optional conformance feature and it has not been taken up by many SQL products. Support for user-defined operators arrived in 1996, but nobody in any national body represented on the committee was willing and able to complete the work that had been started on specifications for supporting domain names in the new places where data types are needed, such as parameter definitions, `RETURNS` clauses of operator definitions, and local variable declarations. That's because most substantive change proposals are drafted by experts employed by leading SQL vendors and none of these vendors' products supported domains at the time.

`CREATE TYPE` was added in 1999. Apart from the simple form shown in Example 2.4b it is an optional conformance feature. For the reasons already given, a domain name cannot be used to specify the declared type of an attribute or the representation type of a distinct type.

## 2.11 What Is a Variable?

SQL's support for variables is very similar to **Tutorial D's**, except that the syntax for creating persistent variables—base tables—is quite different from that used to declare local variables. Example 2.5 is SQL's counterpart of that example in the theory book but, as you know, `CREATE TABLE` is used for base tables.



"I studied English for 16 years but...  
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



**Example 2.5:** A Variable Declaration

```
DECLARE SN SID DEFAULT SID ( 'S1' ) ;
```

This differs from Example 2.5 in the theory book only in the two key words, `DECLARE` in place of `VAR` and `DEFAULT` in place of `INIT`. The effect is exactly the same apart from the fact that, as already noted, SQL's type `SID` (here assumed to be a distinct type) cannot be the same as **Tutorial D**'s. The key word `DEFAULT` is perhaps a strange choice as that word normally suggests action to be taken by the system when no specific action is explicitly demanded by the user. Here it is used to state explicitly an immediate assignment to the variable being declared.

Example 2.6 in the theory book illustrates the use of a `VAR` statement in **Tutorial D** to create a persistent relation variable and declare a (key) constraint to be satisfied by every relation ever assigned to it. That example is actually a repetition of Example 1.1. For completeness, Example 1.1 of the present book is repeated here as Example 2.6, but slightly revised to make it a truer counterpart of the **Tutorial D** example. The revision is marked in bold.

**Example 2.6:** Creating a base table.

```
CREATE TABLE ENROLMENT
    ( StudentId  SID ,
      Name      VARCHAR(30) NOT NULL,
      CourseId  CID ,
      PRIMARY KEY ( StudentId, CourseId )
    ) ;
```

**Explanation 2.6:**

- **CREATE TABLE ENROLMENT** announces that what follows defines a variable in the database, named `ENROLMENT`. A variable in an SQL database is necessarily a table variable, just as in a relational database every variable is a relation variable. SQL does not use the term variable, instead referring to the variable as a *base table* (its value being called a table, of course).
- **StudentId SID** defines the first column of `ENROLMENT`, giving its name and either its declared type (a user-defined type) or its domain—we cannot tell which. If `SID` is a domain, then the definition of that domain specifies the declared type of the column `StudentId`. Similarly, `Name VARCHAR(30)` and `CourseId CID` define the second and third columns of `ENROLMENT`, respectively. A system-defined type is explicitly given for the column `Name` but the remarks on the declared type of `StudentId` apply in similar fashion to `CourseId`. Note carefully that in SQL it is correct, in ordinary prose, to identify columns by their ordinal position. By contrast there is no such thing as “the first attribute” of a relation or a relation variable.



- **NOT NULL**, appended to the definition of `Name`, specifies a constraint to the effect that the table assigned to `ENROLMENT` cannot contain a row in which “the null value of type `VARCHAR(30)`” appears for that column. The constraint is needed for accurate emulation of Example 2.6 in the theory book because relational theory does not admit any counterpart of SQL’s `NULL` (so nor does **Tutorial D**). See the next bullet for an explanation of why `NOT NULL` is not appended to the other two column definitions.
- **PRIMARY KEY ( StudentId, CourseId )** specifies that at no time can two distinct rows appear in the current value of `ENROLMENT` having the same value for `StudentId` and also the same value for `CourseId`. In enterprise terms, no two enrolments can involve the same student and the same course. In addition, it implies that the `NOT NULL` constraint applies to each those two columns, which explains why those constraints are not explicitly declared in Example 2.6 (though there would be no harm in doing so). Unlike **Tutorial D**, SQL again attaches significance to the order in which the columns of a key are specified. There is more to be said about key constraints in SQL but we defer that to Chapter 6, “Constraints and Updating”. Also unlike **Tutorial D**, an SQL base table whose definition includes no key constraints can exhibit the “duplicate row” phenomenon, allowing its current value to include more than one appearance of the same row.

No initial value for `ENROLMENT` is specified in Example 2.6. SQL does allow one to be specified but does not use the `DEFAULT` syntax of local variable declarations for that purpose. Rather, it allows an initial value to be specified *in place* of the list of column definitions as shown in Example 2.6a—a counterpart of **Tutorial D**’s `VAR` declaration in which the declared type is omitted, being implied by the `INIT` specification. When no initial value is specified, the initial value is the empty table of the specified type.

### Effect of Anonymous Columns

Now, recall that a `VALUES` expression denotes a table with undefined column names. If an initial value is to be specified when a base table is created, column definitions have to be implied by that initial value, so the question arises, how can a `VALUES` expression provide the initial value for a base table? The answer is that you have to learn an extra syntactic construct for that purpose, shown in Example 2.6a.

**Example 2.6a:** Specifying an initial value for a base table.

```
CREATE TABLE ENROLMENT ( StudentId, Name, CourseId )
      AS ( VALUES ( SID('S1'), 'Anne', CID('C1') ),
              ( SID('S2'), 'Boris', CID('C1') ) )
      WITH DATA ;
```

**Explanation 2.6a:**

- ( **StudentId**, **Name**, **CourseId** ) provides the names, positionally corresponding to the anonymous columns of the AS table.
- **WITH DATA** specifies that the given table is indeed to be the initial value. Curiously, this is required, unless **WITHOUT DATA** is written instead, in which case the AS table serves only to determine the declared types of the columns of the base table, thus providing something of a counterpart to **Tutorial D**'s **SAME\_TYPE\_AS** construct (though no such facility is available in local variable declarations).
- SQL does not allow constraints to be declared if AS is used in place of explicit column definitions. Example 2.6b shows how the ones given in Example 2.6 could be added subsequently.

**Example 2.6b:** Adding table constraints later

```
ALTER TABLE ENROLMENT ADD CONSTRAINT NameNotNull
        CHECK ( Name IS NOT NULL ) ;
```

```
ALTER TABLE ENROLMENT ADD CONSTRAINT PK_StudentId_CourseId
        PRIMARY KEY ( StudentId, CourseId ) ;
```



What do you want to do?

No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site [www.volvogroup.com](http://www.volvogroup.com). We look forward to getting to know you!

**VOLVO**  
AB Volvo (publ)  
[www.volvogroup.com](http://www.volvogroup.com)

VOLVO TRUCKS | RENAULT TRUCKS | MACK TRUCKS | VOLVO BUSES | VOLVO CONSTRUCTION EQUIPMENT | VOLVO PENTA | VOLVO AERO | VOLVO IT  
VOLVO FINANCIAL SERVICES | VOLVO 3P | VOLVO POWERTRAIN | VOLVO PARTS | VOLVO TECHNOLOGY | VOLVO LOGISTICS | BUSINESS AREA ASIA



**Explanation 2.6b:**

- **ALTER TABLE ENROLMENT** specifies that the definition of base table ENROLMENT is to be modified in some way, such as adding or dropping a column, adding or dropping a constraint (among other things).
- **ADD CONSTRAINT NameNotNull** specifies that in fact a constraint is being added to the definition of ENROLMENT, and the name by which that constraint can subsequently be referred to is NameNotNull. For example, the name would be needed if the constraint were later to be dropped, or perhaps it could appear in an error message when an attempted update is rejected for violating that constraint. Similar comments apply to **ADD CONSTRAINT PK\_StudentId\_CourseId**.
- **CHECK ( Name IS NOT NULL )** specifies a truth-valued expression, Name IS NOT NULL, that must be satisfied by each row of every table that is assigned to ENROLMENT. If the current value of ENROLMENT fails to satisfy this constraint, then the **ALTER TABLE** statement fails.
- **PRIMARY KEY ( StudentId, CourseId )** has the same meaning as in Example 2.6. Again, the **ALTER TABLE** statement fails if the constraint is not satisfied by the current value of ENROLMENT.

Note that it is not possible to specify more than one alteration in a single **ALTER TABLE** statement.

**Effects of NULL**

If *LV* is a local variable and *TLV* is its declared type, then “the null value of type *TLV*” can appear as the value of *LV*, and is indeed its initial value by default.

As already noted in Section 2.8 **The Type of a Table**, there is no null value of any table type, so it is not possible for NULL to appear as the value of a base table. Also, NULL cannot appear in place of a row in any table. However, if *MRT* is a multiset type whose element type is a row type *RT*, then “the null value of type *MRT*” exists and can appear as the value of a local variable of that type, as the argument substituted for a parameter of that type, as the result of an invocation of an operator of that type, as the value of an attribute of that type in a value of some structured type, and as the value of a field of that type in a row. Moreover, “the null value of type *RT*” can appear as one or more elements of a value of type *MRT*.

## Historical Notes

The SQL syntax in Examples 2.6 and 2.6b was defined in SQL:1992, apart from the use of user-defined types. Support for local variables was added in 1996, being included in the definition of the SQL standard's programming language, SQL/PSM. PSM stands for Persistent Stored Modules—user-defined functions and procedures akin to **Tutorial D**'s user-defined read-only operators and update operators, respectively, though actually it defines just the additional language features that are needed to write the implementation code for such modules. There are historical reasons for the inappropriate name for this programming language—the first edition of Part 4, in 1996, included, along with the programming language constructs, the specifications for `CREATE FUNCTION` and `CREATE PROCEDURE` statements, functions and procedures being referred to collectively as “server modules”. When these were later moved to Part 2, SQL/Foundation, the name for Part 4 was not changed.

The `AS` option in Example 2.6a was added in 2003 and is an optional conformance feature. In connection with that, it should be noted that SQL:1992 saw the first appearance of syntax in `CREATE TABLE` whereby some or all of the required column definitions could be copied from an existing base table (or view). For example, `LIKE T2` appearing in place of the  $i$ -th column definition for base table `T1` specifies that the definitions of the columns of `T2` are to appear, in order, as the  $i$ -th,  $i+1$ -th, ... columns of `T1`. Thus, in 2003 the following two statements became equivalent:

```
CREATE TABLE T1 ( LIKE T2 ) ;
CREATE TABLE T1 AS ( SELECT * FROM T2 ) WITHOUT DATA;
```

The `LIKE` option, added in 1992 and still an optional conformance feature, might have had its origins (though I somehow doubt it) in a similar feature of Business System 12, devised in 1978, but BS12's `LIKE` took a table expression of arbitrary complexity rather than being restricted to the name of some existing variable. SQL:1992's `LIKE` was criticised by some (including myself, a member of the standards committee at that time) for being so restrictive, but as we have already seen in the case of `VALUES` expressions, not every table expression in SQL denotes a table whose type meets the special requirements of a base table—in particular, no column of a base table can be anonymous and no two columns can have the same name. This fact would have slightly complicated matters if the restriction on `LIKE` had been removed in 1992, but we have already seen how the problem was addressed when `AS` was added in 2003.

## 2.12 Updating a Variable

Like **Tutorial D**, SQL supports an explicit assignment operator, illustrated in Example 2.7, but this requires the target to be a local variable, not a base table (or updatable view).

### Example 2.7: A Simple Assignment

```
SET SN = SID ( 'S2' ) ;
```

This can obviously be read as “set the variable SN to be equal in value to SID ( 'S2' )”. For completeness we show also SQL’s counterpart of the theory book’s Example 2.8.

**Example 2.8:** Assignment Source Not a Literal

```
SET SN = SID ( SUBSTRING ( SN.C FROM 1 FOR 1 ) || '5' ) ;
```

Example 2.9 is an exact copy of its counterpart in the theory book.

**Example 2.9:** Invocation of a user-defined update operator

```
CALL SET_DIGITS ( SN , 23 ) ;
```

Example 2.10 in the theory book illustrates assignment to a “pseudovalue” (that term being taken from the old IBM language PL/I). SQL doesn’t use that term but does have a counterpart of that particular example.

**Example 2.10:** Assignment of an attribute value in a variable of a structured type

```
SET SN.C = 'S2' ;
```

As in Example 2.10 in the theory book, the entire statement is equivalent to a regular assignment, in this case

```
SET SN = SID().C('S2') ;
```

—and the remarks in the theory book apply equally well here.

## 2.13 Conclusion

In the theory book I concluded Chapter 2 by reminding you of the important distinctions I drew to your attention again here in Section 2.3. They are still important, of course, so I repeat them here, with the illustrative examples translated into SQL:

- syntax and semantics (expressions and their meaning)  
An expression (syntax) *denotes* either a value or a variable.
- values and variables  
A value such as the integer 3, the character string 'London', or the table VALUES ( 3, 'London' ) is something that exists independently of time or space and is not subject to change. A variable *is* subject to change (in value only), by invocation of update operators. A variable reference, such as the expression X, denotes either the variable of that name or its current value, depending on the context in which it appears.
- values and representations of values  
The character string value denoted by 'S1' is the *representation* of the student identifier S1, a value of type SID, denoted by SID( 'S1' ) if SID is a distinct type, or by SID( ) .C( 'S1' ) if it is a structured type. (*Note: “the representation”, not just “a possible representation” as the theory book has it. SQL doesn’t cater for multiple possible representations.*)

**gaiTEYE**  
Challenge the way we run

EXPERIENCE THE POWER OF  
FULL ENGAGEMENT...

.....

RUN FASTER.  
RUN LONGER..  
RUN EASIER...

READ MORE & PRE-ORDER TODAY  
WWW.GAITEYE.COM





- types and representations

For example, if `SID` is a structured type, then `( C VARCHAR ( 5 ) )` defines the representation for all values of type `SID`. (Note “the” again.)

In connection with type `TIMESTAMP`, SQL’s support for time zone displacements appears to suffer from a failure to distinguish a type from its possible representations, giving rise to needless indeterminacy.

- read-only operators and update operators

“+” is a *read-only operator* because, when it is invoked, it returns a value. “SET” is an *update operator* because, when it is invoked, it has the effect of replacing the current value of a variable (i.e., *updating* the variable by *changing* its value)—and does *not* return a value.

- operators and invocations

`SID` is an *operator*, as it happens, an operator of the same name as the type whose definition brings this operator into existence. Assuming the type `SID` to be a distinct type now, the *signature* of the operator `SID` is `SID ( C VARCHAR ( 5 ) )`. `SID ( ' S1 ' )` denotes an *invocation* of `SID`. Similarly, “+” is an operator, with signature `" + " ( A REAL , B REAL )`, and `x+y` denotes an invocation of “+”. (As in **Tutorial D**, “+” has other signatures, defining it for other numeric types such as `INTEGER`.)

- parameters and arguments

`C` is a *parameter* (and in fact the only parameter) of the operator `SID`. `VARCHAR ( 5 )` is the declared type of `C`. Similarly, `x` and `y` denote the arguments substituted for the parameters `A` and `B`, respectively, in the invocation `x+y`.