nderstanding Numerical Data ypes in SQL

data types, numerical data types April 18, 2017  Aldo Zelen

Working with databases of any kind means working with data. This data can take a couple of predefined formats. As you start on your learning path with Vertabelo Academy, you will start to understand SQL's different data types. In this article, we will cover the numeric data types of ANSI SQL. We'll also examine some functions that convert data from one type to another.

Creating tables is the first step in any SQL coding project. You do this using DDL (Data Definition Language) statements like `CREATE` and `DROP`, which you can learn about in our Creating Tables in SQL course. Once you've set up your table, you start listing column names and data types in SQL. Data types tell your database what information to **expect** for that column.

Let's say you have a table of "`users`". Each user has information in a name column and a phone number column. Names would be stored in a *character* column. A phone *number* would be in a *numerical* column. So, numerical columns store numbers and all numbers are the same, right? Guess again.

There are different types of numbers, and there are different types of numerical columns. Anyone with a background in statistics knows that there are four different scales that apply to numbers: nominal, ordinal, interval, and ratio. Unlike other data types, numerical types can represent all of these scales. But how are these scales different from each other?

***Nominal* values differentiate by "name" only.** Nominal values aren't treated like numbers; you can't add or subtract them, and they have no inherent order. For example, student ID numbers and telephone numbers are nominal values. They function more like a label than a number. We can say that our phone numbers are equal (same importance, same length) but we cannot compare them or say that one is first. We cannot add them and get a meaningful phone number.

***Ordinal* values provide order or rank.** In ordinal values, the order between them is the significant thing. Suppose you had a scale of 1 to 3 that rated your mood. Feeling unhappy would get you a "1"; content gets a "2", and happy has a value of "3". You know that happy ranks higher than content or unhappy, but that's about it. You can't add "content" and "unhappy" and get a "happy".

***Interval* values show exact differences.** In an interval scale, the differences between values are what is important. If you subtract 90 degrees centigrade from 100 degrees centigrade, you get a 10-degree difference. Years, dates, and most personality measures are interval measures. Interval values are numerical and are represented as numerical in the database.

***Ratio* values are intervals with a defined zero value.** Like an interval scale, a ratio has measurable differences between values. In a ratio scale, though, a zero value means there is nothing to be measured. For example, think about mass, length, and duration. If any of these have a zero value, there's nothing there. Ratio scales are very important in science.

Numerical types are used to represent all of the above values, especially intervals and ratios. You can compare character values in SQL, so one could argue that

character values can also represent interval data. However, that's a topic for another article.

In SQL, numbers are defined as either **exact** or **approximate**.

The exact numeric data types are `SMALLINT`, `INTEGER`, `BIGINT`, `NUMERIC(p,s)`, and `DECIMAL(p,s)`. Exact types mean that the values are stored as a literal representation of the number's value.

The approximate *numeric data types* are `FLOAT(p)`, `REAL`, and `DOUBLE PRECISION`. These represent real numbers, but they are not represented as exact numbers in the database. Rather, they are an approximation of the real number because of the way that computer systems represent numbers. If this sounds confusing, rest assured that we'll explain it in detail later.

Let's start our consideration of SQL numerical types with the exact or *numeric* data types.

The Numeric Data Types

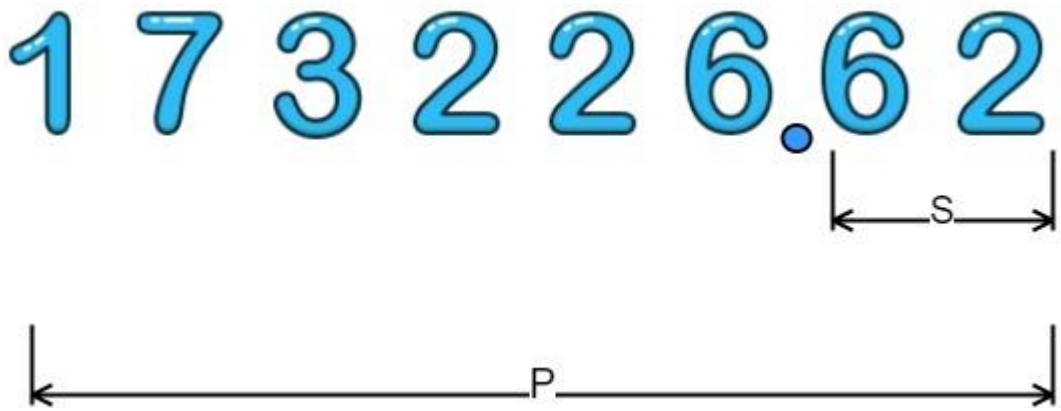
SQL's exact numeric data types consist of **NUMERIC**(p,s) and **DECIMAL**(p,s) subtypes. They are exact, and we define them by precision (p) and scale (s). Precision is an integer that represents the total number of digits allowed in this column. These digits are in a particular radix, or number base – i.e. binary (base-2) or decimal (base-10). They are usually defined with a *decimal point*. The scale, also an integer value, represents the number of *decimal places* to the left (if positive) or right (if negative; this is rarely used) of the *decimal point*.

Let's look at an example. Suppose that you defined a "`balance`" column as `NUMERIC` with a precision of 8 and a scale of 2.

The DDL would look like this:

```
CREATE TABLE account (
accountNo integer,
balance numeric(8,2)
);
```

The "`balance`" column can safely store the number 173226.62.



P represents the total number of all digits and *s* represents the two digits after the decimal.

There is a small difference between `NUMERIC(p,s)` and `DECIMAL(p,s)` types. `NUMERIC` determines the **exact precision and scale**. `DECIMAL` specifies **only the exact scale**; the precision is equal or greater than what is specified by the coder. `DECIMAL` columns can have a larger-than-specified precision if this is more convenient or efficient for the database system.

In other words, `DECIMAL` gives you some leeway.

Keep in mind that **financial data** such as account balances **must be stored as NUMERIC or DECIMAL data types**.

Also, be aware that many top database management systems have vendor-specific representations of numeric types (e.g. Oracle's `NUMBER` data type). These implementations usually do not differentiate between `NUMERIC` and `DECIMAL` types. (In Oracle, both are the `NUMBER` type).

Common Numeric-Type Mistakes

When inserting data into a `NUMERIC` column, remember its precision limits. If you try to insert too large a number, you might get an error. For example, we want to insert the following:

```
INSERT INTO account(accountNo, balance) VALUES(1313,12331411.23);
```

This will produce an error. Why? We try again with a similar number:

```
INSERT INTO account(accountNo, balance) VALUES(1313,123314.1123);
```

The second attempt works. This is because the RDBMS rounds the inserted number by discarding of any "extra" digits to the right of the decimal point. In this case, it kept ".11" but discarded the "23" following. Note that If the first discarded digit is a 5 or above, the RDBMS will round up the leftmost digit.

In our example, that would mean if you inserted:

```
INSERT INTO account(accountNo, balance) VALUES(1313,123314.1153);
```

...the balance in the database would be 123314.12.

To learn more about rounding and common numerical functions, check out the **SQL Basics** course.

The Integer Data Types

Integer data types hold numbers that are whole, or without a decimal point. (In Latin, *integer* means whole.) ANSI SQL defines `SMALLINT`, `INTEGER`, and `BIGINT` as integer data types. The difference between these types is the size of the number that they can store.

Below, we can see Microsoft SQL's definition of various integer data types:

Data type	Range	Storage
bigint	-2^63 (-9,223,372,036,854,775,808) to 2^63-1 (9,223,372,036,854,775,807)	8 Bytes
int	-2^31 (-2,147,483,648) to 2^31-1 (2,147,483,647)	4 Bytes
smallint	-2^15 (-32,768) to 2^15-1 (32,767)	2 Bytes
tinyint	0 to 255	1 Byte

For these types, the default size of the column is important. Defining a smaller column size for smaller integer types (if you know the max size in advance) can help keep your tables as small as possible.

Common Integer-Type Mistakes

INTEGER columns round decimal points. To explain, let's modify our table a little:

```
CREATE TABLE account (
accountNo integer,
balance integer
);
```

If we run this statement:

```
INSERT INTO account(accountNo, balance) VALUES(1313,123314.3153);
```

... the inserted values are rounded to the first digit before the decimal point. Suppose we insert 123314.5 into the balance column:

```
INSERT INTO account(accountNo, balance) VALUES(1313,123314.5);
```

Because we changed the value behind the decimal point to a value equal to or greater than 5, we would get 123315.

If you're using integer data types in formulas, know that **rounding can cause inconsistencies in formulas**. If you subtract 123314.3153+123314.3153:

```
INSERT INTO account(accountNo, balance)    VALUES(1313,123314.3153+123
```

.. the inserted value would be 123314. If you subtract the same values we added before from the balance:

```
SELECT BALANCE - 123314.3153 FROM ACCOUNT;
```

... the result, now a decimal number, would be 123314.3147. This is a clear inconsistency. You can avoid it by **defining appropriate column data types according to what operations will be done on the columns**.

The Float Data Types

Float and float-related data types hold *approximate* numeric values. They consist of a significant (a signed numeric value) and an exponent (a signed integer that specifies the magnitude of the significant). These data types have a *precision*, or a positive integer that defines the number of significant digits (exponent of the base of the number).

This type of data representation is commonly called floating-point representation.

If we were to represent 173226.62 in this notation (with a base of 10), it would look like this.

The value of an approximate numeric value is its significant multiplied by 10 to the exponent.

To truly understand the floating point data type, you will have to dig into a little bit of computer science. It can be fun, but at this stage of your SQL journey I believe it is overkill. For now, just remember that there are three ANSI-standard SQL approximate types: `FLOAT(p)`, `REAL`, and `DOUBLE PRECISION`.

The difference between `FLOAT(p)` and `REAL` is that `FLOAT` has a binary (not decimal) precision equal to or greater than the defined value. `REAL` has a predefined precision based on the database implemented. In normal working life, `FLOAT` is rarely used; `REAL` and `DOUBLE PRECISION` are tied to particular system implementations and developers tend to pass system implementation work to the DBAs and Sysadmins.

The difference in `REAL` and `DOUBLE PRECISION` is that `REAL` represents numbers in 34 bits and `DOUBLE PRECISION` in 64 bits.

Working with Approximate Types

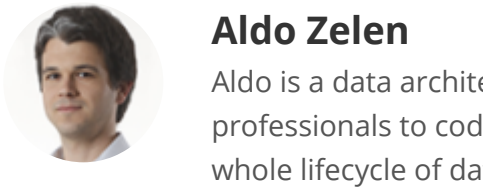
It's very important to remember that approximate numerical data types sacrifice precision for range, thus the name *approximate*.

In calculations, approximate types may give you weird results – like 204.000000056 where the exact result should be 204. If you are building your database for an engineering or scientific application, floating data types should be fine. There is also the difference in speed; if you are doing an exceptionally large number of complicated computations (e.g. trigonometric functions, etc.) float types should be

much faster than other numerical data types. On the other hand, if you are working on a financial, banking, or other business application, using decimal representation is more appropriate.

SQL's numerical data types are not just integer- and decimal-related. They are reflections of the need to store data in a way that's safe, predictable, and usable. As with any programming language, they remind us of the computer science aspect of databases and SQL.

To practice numerical data types and come to grips with their possibilities, usage, constraints, and common mistakes, go to **Vertabelo Academy** and have fun!



Aldo Zelen

Aldo is a data architect with a passion for the cloud. From leading a team of data professionals to coding a data warehouse in the cloud, Aldo has experience with the whole lifecycle of data-intensive projects. Aldo spends his free time forecasting geopolitical events in forecasting tournaments.

« 

Previous Post:
5 Functions for Manipulating SQL Strings

Next Post:
Using LIKE to Match Patterns in SQL

»

Related Posts:

Spooky Scary NULL: Unexpected Danger Lurking in Your Database

SUBSCRIBE

Support

If you need assistance, drop us a line at academy@vertabelo.com

Legals

Terms of Service
Privacy Policy

Powered by