

## Re: float bug? perl 5.8, DBI and oracle 10.2.0

---

*Source:* <http://coding.derkeiler.com/Archive/Perl/perl.dbi.users/2007-07/msg00109.html>

---

- *From:* [Tim.Bunce@xxxxxxxxxx](mailto:Tim.Bunce@xxxxxxxxxx) (Tim Bunce)
  - *Date:* Wed, 18 Jul 2007 00:01:36 +0100
- 

On Tue, Jul 17, 2007 at 11:11:49AM -0400, Christopher Sarnowski wrote:

For what it's worth, I'd say "not a bug." If you want to store high precision numbers in oracle, you've got 38 decimal digits to play with, and with minimal coaxing perl (and DBI) will handle them as strings at the appropriate points so that the exact values go in and come out.

For the record, DBD::Oracle binds parameters and fetches values as strings.

Once you start doing any sort of math with them, I'd say all bets are off. I haven't done any numerical work in 10 years or so, but I seem to recall that one can reasonably expect 6 or so decimal significant digits from a 32 bit floating point number – I'll go out on a limb and hazard that one can expect 12 or so digits from a 64 bit floating point number – at any rate I'd be very surprised to get 18 significant digits. And of course these expectations will shrink depending on the number and order of manipulations.

And as <http://www.lahey.com/float.htm> points out, that's just the way it is.

Yeap.

I'd have to ask, when you put in 1.73696, how did you derive it in the first place? Are your measurements and calculations such that you "really" have 1.7369600000000000?

This is a key point. When you're on this kind of investigation you must assume nothing and check everything with great care.

Funnily enough I wrote a section on this topic back in May 2006 for the 2nd edition of DBI book (which is currently shelved, by the way). I've appended the relevant chunk of the rough draft. Comments welcome.

Tim.

## =head0 Handling Database Data Types

We've talked a lot about fetching data. Fetching it all at once, fetching it row by row, fetching into arrays, and fetching into hashes. But fetching what? What is this stuff we've been fetching?

Data, I hear you say. Strings and numbers and stuff like that. Well, strings and numbers may seem simple, and usually they are, but even here there are issues you should be aware of. Then there are more complex types like dates and LOBs with their own set of complications to consider. Finally we need to discuss NULLs and how the absence of a value is represented and handled.

[...string sections skipped...]

## =head1 Numeric Types

Let's take a look at integer, fixed point, and floating point values now. These types are more straightforward than character strings but there are still issues you should be aware of.

### =head2 The Numbers of Perl

Before we talk about databases and drivers and such like I should outline the way perl handles numeric values. This is very much an outline as there are dragons lurking in the details. The intrepid and curious might like to look at the perl source code. Especially the comments for NV\_PRESERVES\_UV in sv.c and PERL\_PRESERVE\_IVUV in pp\_hot.c

Internally perl has three basic types that are relevant here: integer values (known as IV), floating point values (known as NV), and strings.

### =head3 Perl Integer Values

Integers are typically stored as 32 or 64 bit (4 or 8 byte) values depending on how perl was configured. You can check the size of integers in your perl by running `C<perl -V>` and looking for `C<ivsize=>` in the output. The range of a 32 bit integer is `-2,147,483,648 to 2,147,483,647` (10 digits of precision). The range of a 64 bit integer is `-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807` (19 digits of precision).

### =head3 Perl Floating Point Values

Technically the term "floating point" refers to a number representation consisting of a `I<mantissa>`, `C<M>`, and an `I<exponent>`, `C<E>`. The number represented is the value of `C<M ** E>`. But what does that mean?

## Re: float bug? perl 5.8, DBI and oracle 10.2.0

Basically, a floating point value is represented internally by two values. One value, the mantissa, holds a binary I<approximation> of the significant digits and another value, the exponent, is used to indicate where the decimal point should be. It may be within the significant digits but it may also be way off to the right (positive mantissa) or left (negative mantissa).

Floating point values are typically stored in 64 bits or sometimes 96 bits (that's 8, or 12 bytes) depending on how your perl was configured. You can check the size used in your perl by running `C<perl -V>` and looking for `C<nvsize=>` in the output. The 64 bit floats are known as I<doubles> and have approximately 15 digits of precision between  $1e-307$  to  $1e+308$ , and the 96 bit floats are known as I<long doubles> and have approximately 18 digits of precision between  $1e-4931$  to  $1e+4932$ . Some systems support 128 bit I<quad doubles> with even greater precision and scale.

It's becoming more common for perl to be configured with 64 bit integers but still using 64 bit floating point values. But a 64 bit integer has 19 digits of precision whereas a 64 bit floating point value only has approximately 18. This is important to know because it means that a large integer may lose precision if it's involved in a calculation that causes it to be converted to a floating point value (which is basically anything more involved than addition or subtraction of another integer).

If you ask perl to print a value where the decimal point is outside the significant digits it'll format the value using exponential notation, where the mantissa and exponent are written separately with an `C<e>` between them. You can get the same effect using `C<printf "%e">`.

Here are a few examples to help illustrate all this:

```
print 1000000000000000.0, "\n";
print 1000000000000000.0 * 10, "\n"; # shift decimal place
print 0.0000000000000001, "\n";
print 0.0000000000000001 * 10, "\n"; # shift decimal place
print 1 / 299792458, "\n";
printf "%e\n", 42.1;
```

prints:

```
1e+15
1e+16
1e-16
1e-15
3.33564095198152e-09
4.210000e+01
```

=head3 Beware the Creeping Errors

## Re: float bug? perl 5.8, DBI and oracle 10.2.0

Floating point values are so common, so pervasive, and so effective that it's easy to forget that their great utility comes with a price: subtle loss of precision, even for small values.

Here's a little script that adds 42.1 one thousand times:

```
$a = 0;
$a += 42.1 for (1..1000);
print "$a\n";
```

Well,  $42.1 * 1000$  is 42100, so that's what it should print. But it doesn't, it prints:

```
42099.9999999992
```

This sort of thing gives accountants nightmares. Your mileage may vary as floating point systems can vary, leading to further nightmares. Try it for yourself.

You might be lucky and find it prints the right number, but don't think that means all is well. All it means is that your perl was configured with I<long doubles>. The underlying problem is still there, we just need to push a little harder to make it show itself. Change the C<1000> above to C<1000000>, or more, and you'll see the loss of precision. [XXX check that 1000000 does exhibit the problem on long double systems.]

You may remember I said that the mantissa part of the internal floating point representation holds a I<binary approximation> of the significant digits. It's that binary approximation that's the cause of the problems.

I don't want to delve further into the intricacies of why floating point values behave this way. There's no need. There are just two things you need to know:

Firstly, that floating point values often contain I<imperceptably> small errors (that don't show up when you print them) so you need to take care when working with them to avoid accumulating those errors into nasty surprises like the one above.

Secondly, that those 'imperceptably small errors' mean that just because two floating point values I<appear> to be the same doesn't mean they are. Take a look at this example:

```
$a = 42.1; $a += 0.1 for (1..30); # add 3 one way
$b = 42.1; $b += 0.3 for (1..10); # add 3 another way
print "a=$a b=$b\n";
($a == $b) ? print "Equal\n" : print "Not equal!\n";
```

it prints:N<Again, your mileage may vary. Especially on I<long double> systems.>

a=45.1 b=45.1  
Not equal!

This is because the test for equality checks the underlying binary representation of the values and thereby includes those imperceptable errors.

### =head3 Perl String Values

Whenever a string is used in a numeric context perl will try to interpret the contents of the string as a number. Before perl 5.8 strings were always converted to floating point values. From perl 5.8 onwards a string will become an integer value if appropriate. Strings that don't look like numbers are converted to 0 and a warning generated if perl is running with the `-w` flag (which is highly recommended).

What if you need to work with larger values, or greater precision, than your perl can handle? With perl there is always a way and the way here is to use the `Math::BigInt` and/or `Math::BigFloat` modules which come with perl. Those modules let you work with arbitrary sized integers and floating point values.

Now, back to your regularly scheduled database programming...

### =head2 Integer Values

Integers. 0, 1, 2, 3, -7, 42. You can't get much simpler than integers. But do you I<get> integers at all?

Some databases support very large integers. Oracle, for example, supports integers with 38 digits of precision. That's far beyond the 10 digits of a simple 32bit integer and even the 19 digits of a 64bit integer. Because of this the `DBD::Oracle` driver returns integers, and indeed all other numeric types, as strings.

By returning a string the driver avoids the issue of how best to deal with values outside the range supported by Perl's native types. The application is then free to do whatever is most appropriate. Often that's to simply duck the issue as well, in which case perl will convert the strings to floating point values and everything will be, or seem to be, just fine. However, if the application does care about precision then it can use the `Math::BigInt` and/or `Math::BigFloat` modules mentioned above.

Perhaps you don't think it would matter much if 1234567890123456 becomes 1.23456789012346e+15. After all it's only lost one digit of precision in a very large number. Who'd notice? Apart from your boss you'll find that databases are picky about accuracy as well. If 1234567890123456 was a value fetched from the database and you tried to update it by executing

```
UPDATE table SET foo = foo - 42 WHERE foo = 1.23456789012346e+15
```

you'll be disappointed.

The DBI will probably gain a way to hook into the fetching of a value from the database so that the use of `Math::BigInt`, for example, can be made transparent and not clutter up the code. But that hasn't happened yet. XXX

Many databases support multiple sizes of integer types from 1 to 8 bytes in size with `INTEGER` (4 bytes) and `SMALLINT` (2 bytes) being the most common. Oracle is a little unusual in that it only has one underlying numeric type which is variable width and all other numeric types are aliases for it.>

The standard integer types are `INTEGER` and `SMALLINT`, with `BIGINT` and `TINYINT` being newer additions. The `TYPE` numbers are 4, 5, and -5, -6 respectively.

=head2 Fixed Point Values

Databases often provide a fixed point numeric type called `NUMERIC`, or `DECIMAL`, or both. Officially the only difference between `NUMERIC` and `DECIMAL` is that `NUMERIC` has a precision `I<equal>` to the one specified in the type declaration and `DECIMAL` has the same `I<or more>` precision than the declared one. It's a classic fudge from the early days of SQL standardization. Most databases just use the same type for both.>

You can think of fixed point values as being like a string of digits of a fixed maximum length with a decimal point at a fixed position. Hence the name: fixed point. The key, er, point about fixed point values is that they never lose precision. Unlike floating point, which I'll describe next, there's no fuzzyness in the value. That makes them very useful where accuracy is required, which is commonly the case with money, for some reason.

Consider this table:

```
CREATE TABLE pay {
id INTEGER,
amount DECIMAL(5,2)
}
```

In this example, 5 is the `I<precision>` and 2 is the `I<scale>`. The precision is the `I<total>` number of significant decimal digits that will be stored for values, and the scale is the number of those digits that will be stored `I<following>` the decimal point.

The value 42.1 would be stored in a `DECIMAL(5,2)` field as:

```
<- precision -> 5  
[ ] [4] [2] . [1] [0]  
<- scale -> 2
```

If the scale is 0 then the value will have no decimal point or fractional part.

Standard SQL requires that the amount column be able to store any value with 5 digits in total with 2 of those after the decimal point. In this case, therefore, the range of values that can be stored in the amount column is from -999.99 to 999.99.

Drivers should typically return these values as strings for the same reasons that some return integers as strings: it avoids the driver being the cause of a loss of precision. Also take note of the cautions in the Floating Point Values section below as they can also apply to fixed point values.

One more thing to keep in mind about fixed point types: zero isn't false. A zero stored in a DECIMAL(5,2) field will probably be returned as "C<0.00>" so code like this:

```
$hourly_rate = $row->{hourly_rate} || 0.42; # if zero then use default value
```

won't default a zero value to 0.42. I've seen this cause bugs in production systems more than once. The "zero is false" concept runs deep in Perl developers and can make it hard to spot this problem.

The standard fixed point types are NUMERIC and DECIMAL, with TYPE codes of 2 and 3 respectively.

## =head2 Floating Point Values

Having discussed I<fixed> point values you ought to be able to guess what I<floating> point means: you still have a precision of a certain number of digits but instead of the decimal point being fixed at a certain location, it 'floats'. But what does that mean?

Take a moment to reread "Perl Floating Point Values" on page XXX if you haven't just read it. That's where I discuss some subtle but important issues with floating point values in general. Here on I'll just focus on how those issues apply in the context of the DBI.

The standard floating point types are FLOAT, REAL, and DOUBLE, which have standard TYPE codes of 6, 7, and 8 respectively.

In most databases, the REAL type is half the size of the DOUBLE type and has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The DOUBLE type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits.

### =head3 Floating Ambiguity

The FLOAT type is typically a 'smart alias' for either REAL or DOUBLE depending on the value of an optional value in parenthesis: C<FLOAT(I<n>)>.

It's best avoided for applications wishing to be portable because databases differ in how they interpret the value of I<n>. Some interpret it as the precision in I<binary> digits, which matches the SQL standard, while others interpret it as the precision in I<decimal> digits.

### =head3 Beyond the Fringe

Values that are too large or too small may cause an error, or may be capped at infinity (positive or negative infinity, as appropriate).

Values with more significant digits than can be represented by the type used may be rounded to fit, or truncated to fit, or cause an error.

Numbers too close to zero that are not representable as distinct from zero may cause an underflow error, or may be silently changed to 0.

May this, may that. It all depends on the database. Are you having fun yet?

### =head3 Behind the String

Back in "Perl Floating Point Values" on page XXX we looked at how two numbers that appear to be the same may not be equal. Here's another variation on the same theme, but this one is more directly relevant to databases and the DBI:

```
$a = 42.1; $a += 0.1 for (1..30); # add 3
$b = "$a"; # cross the client/server interface as text
print "a=$a b=$b\n";
($a == $b) ? print "Equal\n" : print "Not equal!\n";
```

I'm sure you can guess what that prints:

```
a=45.1 b=45.1
Not equal!
```

Depending on the database and driver used the conversion two/from text at the client/server interface can apply in either direction and happen at the client, or at the server, or not at all. Or perhaps you did it yourself without noticing:

```
# select the lowest value of the foo field
$min_foo = $dbh->selectrow_array("SELECT MIN(foo) FROM TABLE");
# delete all records with that value
$dbh->do("DELETE FROM TABLE WHERE foo = $min_foo");
```

Using a placeholder I<may> help:

```
$dbh->do("DELETE FROM TABLE WHERE foo = ?", undef, $min_foo);
```

but isn't guaranteed to, and it almost certainly won't if the driver emulates placeholders by substituting values into the SQL statement. (We'll talk about placeholders in "Adding Parameters to Statements" on page XXX.)

XXX xref this above from optimistic locking

### Getting the Point

If you live in a country that uses a full stop ("C<.>") as the decimal point character it may come as a surprise to you that large parts of the world don't. Many countries use a comma ("C<,>") instead.

The rules about how numbers and other types are formatted in different places are known as I<locales>.

All is fine so long as your application and the database you're talking to both agree on the format to use. But if they don't agree, perhaps because they're in different countries, or are using different locales for some other reason, then you'll get some unpleasant surprises.

When sending floating point values to the database as strings some databases will complain about invalid characters in the value with an error. That's good, or at least it's better than other databases which will silently truncate or otherwise mangle the number.

When fetching floating point values from the database as strings you'll get a string formatted by the database server according to its rules. If you use that value in a numeric context perl will try to convert it to a number for you. You'll only get a warning about any invalid characters if you're running perl with warnings enabled (via the C<-w> option or the C<use warnings;> pragma).

### Getting the Value

DBI drivers for databases which use standard floating point types, and can return floating point values as floating point values, ought to do so in order to avoid an obvious cause of imperceptible changes in the value.

It would be tempting to think that that would completely avoid the problem. Sadly that's not the case. Yes, it does significantly reduce it, but don't be lulled into a false sense of security. There are still other ways imperceptible differences can creep in.

### In Summary

The bottom line here is to avoid working with floating point values in a

Re: float bug? perl 5.8, DBI and oracle 10.2.0

way that accumulates the hidden errors, and avoid any logic that assumes two floating point values will be equal.

I've only touched on the main issue that you need to be aware of when working with floating point values but there are others. Lots of them! For example, the standard floating point format, known as IEEE 754N<U<<http://grouper.ieee.org/groups/754/>> >, defines special values like "C<Inf>" for Infinity, "C<NaN>" for Not a Number, even the humble zero has a positive and negative version (though they are treated as being equal, thankfully). But, you can go a long way without knowing more than that, so I'll stop here.

=cut

.