PL/SQL 101 – DATATYPES – DATE

Author: BluShadow
Last Updated: 8th January 2016

CONTENTS

1. INTRODUCTION

Following from my article on the NUMBER datatype (https://community.oracle.com/docs/DOC-917993), other common questions we get on the community relate to people trying to store dates as VARCHAR2 to store them "*in a particular format*", or trying to write queries that are treating DATE values as strings, and confused as to why it's things aren't working. This article (certainly not as lengthy as the NUMBER datatype document you'll be pleased to hear) is going to look specifically at the DATE datatype and the importance of storing dates using the DATE datatype. The same principles apply to the TIMESTAMP datatype, but for the purposes of keeping things simple, we'll just look at DATE.

2. HOW CAN I STORE DATES IN A PARTICULAR FORMAT?

Briefly, the answer to this is simple… **you don't**.

For example, a date value representing 17th December 2015 is exactly the same date regardless of which date format you want to use to display it, e.g. in the UK we would display this as 17/12/2015, yet in America this would be displayed as 12/17/2015.

```
SQL> select sysdate from dual;

SYSDATE
----------
17/12/2015

1 row selected.
```

Just like the NUMBER datatype, the issue that is really trying to be resolved isn't one of how data is stored on the database, but rather one of how the data should be displayed to a user when it's queried; often being part of a requirement such as "The date should be shown in DD/MM/YYYY format". If the requirement has come from a business user stating "The date should be **stored**…" then consider that users shouldn't be providing the technical requirements, and they really mean that's how they want to see the data presented. How you store data is up to you as the technical expert, and when it comes to dates you have one choice… the DATE datatype (or TIMESTAMP if appropriate).

In these cases, the developer should consider that often the initial requirement may have been stipulated by one user or department, and later on there may be other requirements that want to use the same value, but display it differently, e.g. the Finance Department provided that initial requirement, but later the Payroll department say they want to provide the same data on cheques to be paid to people, and they don't want the date to show ambiguously. Thus the finance department of the company may work in UK date format DD/MM/YYYY, so expect all dates they see to be in that format. But cheques sent out by payroll may be going to employees worldwide, and someone in America receiving a cheque showing, for example, 12/04/2015, would read that as December 4th 2015; yet if the cheque was created on 12th April 2015 and received shortly after, it would appear to them that it had been post-dated to December. In that case, the payroll department want the dates showing in a specific format of DDth Month YYYY (e.g. 12th April 2015). If you'd stored it the date in one format in the database (as Finance had initially requested) what would you then do? You'd have to start splitting up the string, try and convert the month numbers to the month names, just to display it differently.

The thing to remember here is that, how data is stored and how it is displayed should be kept separate. Store the data in one consistent and most appropriate format, and then allow the displaying of the data to vary depending on requirements (or even individual regional or user settings!).

So why not store all DATEs in a VARCHAR2 datatype?

Well, if the above example hasn't made it clear for you, let's take a look and see…

When you try and store a date as a VARCHAR2, every digit or letter in that date becomes a character in the string, and is stored in its own byte. We can see this using SQL's DUMP function:

```
SQL> select '17 December 2015' as dt, dump('17 December 2015') as dmp from dual;

DT               DMP
---------------- --------------------------------------------------------------------
17 December 2015 Typ=96 Len=16: 49,55,32,68,101,99,101,109,98,101,114,32,50,48,49,53
```

Each of those bytes represents the ASCII character value of each digit, e.g. 49 = '1', 55 = '7', 32 = space etc. If we look at the same value as a DATE datatype:

```
SQL> select date '2015-12-17' as dt, dump(date '2015-12-17') as dmp from dual;

DT         DMP
---------- --------------------------------
17/12/2015 Typ=13 Len=8: 223,7,12,17,0,0,0,0
```

It doesn't take as many bytes to store it. So, that's the first benefit of using a NUMBER over a VARCHAR2, but what else?

Well, dates stored as strings cannot be sorted in the way you expect. Let's order some dates stored as VARCHAR2:

```
SQL> select dt from myvarchardates order by dt;

DT
-----------
01-Apr-2015
01-Feb-2015
01-Jan-2015
01-Jul-2015
01-Jun-2015
01-Mar-2015
01-May-2015
```

Why are they not sorted into their date order? Let's look at the bytes for them:

```
SQL> select dt, regexp_substr(dump(dt),'[^ ]+$') as dmp from myvarchardates order by dt;

DT          DMP
----------- --------------------------------------------------------------------------
01-Apr-2015 48,49,45,65,112,114,45,50,48,49,53
01-Feb-2015 48,49,45,70,101,98,45,50,48,49,53
01-Jan-2015 48,49,45,74,97,110,45,50,48,49,53
01-Jul-2015 48,49,45,74,117,108,45,50,48,49,53
01-Jun-2015 48,49,45,74,117,110,45,50,48,49,53
01-Mar-2015 48,49,45,77,97,114,45,50,48,49,53
01-May-2015 48,49,45,77,97,121,45,50,48,49,53
```

When you look at the ASCII values of each digit, you can see more clearly where the ordering has come from. Everything is ordered by the first character's ASCII value, then within those groups ordered by the second characters ASCII value etc. Ordering of VARCHAR2 strings is done on a character by character basis.

The ordering of the data is more relevant than just how the data is sorted in the results. "ordering" is also relevant to how data is compared when you search a range of values.

For example, let's query our string data to find dates that are less than or equal to 1st February. In our mind we know this should be 1st January and 1st February from the example data, but what happens when we query it…

```
SQL> select dt, regexp_substr(dump(dt),'[^ ]+$') as dmp from myvarchardates where dt <= '01-Feb-2015'
order by dt;

DT          DMP
----------- --------------------------------------------------------------------------------
01-Apr-2015 48,49,45,65,112,114,45,50,48,49,53
01-Feb-2015 48,49,45,70,101,98,45,50,48,49,53
```

For exactly the same reasons as we saw when ordering the data, we get the wrong results. 1st April appears to be before 1st February, simply because "A" comes before "F" in the alphabet (ASCII character values), and we don't even get January, because "J" comes after "F".

Let's look at the same ordering and querying done on the same date values, but stored as DATE datatype…

```
SQL> select dt, regexp_substr(dump(dt),'[^ ]+$') as dmp from mydatedates order by dt;

DT         DMP
---------- --------------------------------------------------------------------------------
01/01/2015 120,115,1,1,1,1,1
01/02/2015 120,115,2,1,1,1,1
01/03/2015 120,115,3,1,1,1,1
01/04/2015 120,115,4,1,1,1,1
01/05/2015 120,115,5,1,1,1,1
01/06/2015 120,115,6,1,1,1,1
01/07/2015 120,115,7,1,1,1,1

SQL> select dt, regexp_substr(dump(dt),'[^ ]+$') as dmp from mydatedates where dt <= date '2015-02-
01' order by dt;

DT         DMP
---------- --------------------------------------------------------------------------------
01/01/2015 120,115,1,1,1,1,1
01/02/2015 120,115,2,1,1,1,1
```

Now we're getting the correct results. The dates are given in the correct order, and when I query for a range style query, I get the data I expect to get. What happens though if I change my session's date format to appear differently…

```
SQL> alter session set nls_date_format = 'fmMonth DD, YYYY';

Session altered.

SQL> select dt, regexp_substr(dump(dt),'[^ ]+$') as dmp from mydatedates order by dt;

DT                 DMP
------------------ --------------------------------------------------------------------------------
January 1, 2015    120,115,1,1,1,1,1
February 1, 2015   120,115,2,1,1,1,1
March 1, 2015      120,115,3,1,1,1,1
April 1, 2015      120,115,4,1,1,1,1
May 1, 2015        120,115,5,1,1,1,1
June 1, 2015       120,115,6,1,1,1,1
July 1, 2015       120,115,7,1,1,1,1
```

Even though my session now displays the dates in a format I want to see it with the month names, the internal data is still stored in the same way, and the ordering still works in the same way.

The key thing here is that **HOW THE DATA IS DISPLAYED IS, AND SHOULD BE, INDEPENDENT OF HOW IT IS STORED**.

Apart from VARCHAR2 strings taking up more bytes, and the sorting of the data being broken, is there any other good reason? There certainly is… please read on…

Oracle provides in built validation and built in functionality for the datatype you use. If you try and store a non-date value in a DATE datatype, you'll correctly get an exception raised. Conversely, if you try and store a non-date value in a VARCHAR2 datatype, there won't be any exception… at least not until you try and use that value as a date and find your code breaks because it can't work out what, for example, '01-JAN-2015'+1 evaluates to.

Let's examine this a little more closely.

Built in Date validation

Firstly, let me set up my table to store DATE datatypes…

```
SQL> alter session set nls_date_format = 'DD/MM/YYYY';

Session altered.

SQL> create sequence id_seq;

Sequence created.

SQL> create table mydatedates (id number, dt date);

Table created.

SQL> create trigger ins_mydatedates before insert on mydatedates
  2  for each row
  3  begin
  4    :new.id := id_seq.nextval;
  5  end;
  6  /

Trigger created.
```

And if I insert a couple of valid dates to that table…

```
SQL> insert into mydatedates (dt) values (date '2015-01-01');

1 row created.

SQL> insert into mydatedates (dt) values (date '2015-02-01');

1 row created.
```

They insert without a problem. What about if I insert an invalid date…

```
SQL> insert into mydatedates (dt) values (date '2015-02-29');
insert into mydatedates (dt) values (date '2015-02-29')
                                           *
ERROR at line 1:
ORA-01847: day of month must be between 1 and last day of month
```

Oracle has validated the date and knows that 29th February 2015 isn't a valid day. How about 29th February 2016…

```
SQL> insert into mydatedates (dt) values (date '2016-02-29');

1 row created.
```

It's happy to insert that because Oracle knows that 2016 is a leap year and there are 29 days in February for that year.

We've not had to do any work ourselves to know whether a date is valid or not. If it's invalid, Oracle raises an exception and tells us. It knows the different number of days in different months, and leap years; even the change from Julian to Gregorian Calendar back in 1582 (if you really need to store such dates). If you are working on a database with different nationality settings, it will even take account of regional calendar differences. You don't get that with VARCHAR2.

Can invalid DATE values be stored?

Considering Oracle has in built validation for DATEs you'd expect that it's not possible to store invalid dates in the database. However, the validation Oracle performs is only done when DATE data is inserted through the most regular methods e.g. insert statements or SQL*Loader/External Tables etc.

Some methods of populating DATE values, such as database imports or some other API's, can allow for invalid dates to be stored. As an example I've created myself a function that gives me a DATE value that has bypassed the date validation (I won't supply the code for it here, as you don't really want or need to do this!). Let's insert an invalid DATE and see what happens…

```
SQL> alter session set nls_date_format = 'DD-MON-YYYY HH24:MI:SS';

Session altered.

SQL> delete from mydatedates;

3 rows deleted.

SQL> insert into mydatedates (dt) values (bypass_date_validation(120,100,255,255,1,1,1));

1 row created.

SQL> select dump(dt) from mydatedates;

DUMP(DT)
--------------------------------------------------------------------------------
Typ=12 Len=7: 120,100,255,255,1,1,1
```

So, I've certainly inserted the 7 byte values for my DATE into my table, and I've asked for my DATE value to be displayed in DD-MON-YYYY HH24:MI:SS format, meaning I should see the Day number, the short month name (e.g. JAN, FEB, MAR etc.), the 4 digit year, and the time shown in 24 hours clock as hours minutes and seconds. Let's query the date and see…

```
SQL> select dt from mydatedates;

DT
--------------------
01-NOVEMBER-2000 00:
```

Hmm, clearly I've upset the database. The month is shown as the full month name, and I've lost the minutes and seconds from the time. Perhaps the issue was something to do with my NLS date format parameter settings? I'll check by inserting a valid date/time and selecting again…

```
SQL> insert into mydatedates (dt) values (sysdate);

1 row created.

SQL> select * from mydatedates;

        ID DT
---------- --------------------
         6 01-NOVEMBER-2000 00:
         7 17-DEC-2015 10:10:31
```

Ok, that proves it. The valid date (sysdate) is displayed correctly, but the date I inserted by bypassing validation is clearly corrupted.

Worse than that, a corrupted date value that bypasses validation could make it so that I can't even select the data…

```
SQL> insert into mydatedates (dt) values (bypass_date_validation(0,0,255,255,1,1,1));

1 row created.

SQL> select dt from mydatedates;
ERROR:
ORA-01801: date format is too long for internal buffer



no rows selected
```

Ouch! That doesn't look good at all.

It's a rare exception, to find corrupted dates in the database, but it's worth bearing in mind that some processes or methods of importing data to tables, CAN bypass the built in DATE validation of the database; so if you're doing something out of the ordinary, or using some 3rd party tool or software to push data into the database, always consider that not all validation is done by the Oracle database and it can be up to that external process to ensure it's supplying the correct underlying data values.

Date Manipulation Functionality

As the final part of this section, I'm going to look at why the DATE datatype is far superior to VARCHAR2 when it comes to manipulating dates. It won't take long for you to see that treating dates as strings (varchar2) is certainly NOT a good idea.

We'll start with something basic… adding a day to a date.
If our dates are stored as VARCHAR2…

```
SQL> select dt
  2        ,to_char(to_number(substr(dt,1,2))+1,'fm00')||substr(dt,3) as new_dt
  3  from myvarchardates;

DT          NEW_DT
----------- -----------
01-Jan-2015 02-Jan-2015
01-Feb-2015 02-Feb-2015
01-Mar-2015 02-Mar-2015
01-Apr-2015 02-Apr-2015
01-May-2015 02-May-2015
01-Jun-2015 02-Jun-2015
01-Jul-2015 02-Jul-2015
```

We've had to do some pretty horrible string manipulation and conversion between varchar2 and number and back again, which is very specific to the format the date has been stored in (and you'll be in even more trouble if people are storing different formats in the same varchar2 column). What's worse is that we lose Oracle's built in date validation again…

```
SQL> insert into myvarchardates (dt) values ('28-Feb-2015');

1 row created.

SQL> select dt
  2        ,to_char(to_number(substr(dt,1,2))+1,'fm00')||substr(dt,3) as new_dt
  3  from myvarchardates
  4  where dt = '28-Feb-2015';

DT          NEW_DT
----------- -----------
28-Feb-2015 29-Feb-2015
```

… so now we get invalid dates.

Now, with our dates stored correctly as DATE datatype…

```
SQL> ed
Wrote file afiedt.buf

  1  select dt
  2        ,dt+1 as new_dt
  3  from mydatedates
  4* order by dt
SQL> /

DT         NEW_DT
---------- ----------
01/01/2015 02/01/2015
01/02/2015 02/02/2015
28/02/2015 01/03/2015
29/02/2016 01/03/2016
```

… adding a day is as simple as just adding 1 to the DATE column, AND Oracle automatically takes account of date validation, allowing for the number of days in each month to be accounted for, as well as the leap years etc.

You want to add months to a date? No problem…

```
SQL> ed
Wrote file afiedt.buf

  1  select dt
  2        ,add_months(dt,1) as new_dt
  3  from mydatedates
  4* order by dt
SQL> /

DT         NEW_DT
---------- ----------
01/01/2015 01/02/2015
01/02/2015 01/03/2015
28/02/2015 31/03/2015
29/02/2016 31/03/2016
```

… Oracle provides a function for that, which includes functionality that recognises the special case of the last day of a month, and accordingly adjusts the day part.

Or perhaps you want to determine the age of a person from their date of birth?

```
SQL> select floor(months_between(sysdate, date '1983-02-01')/12) as age
  2  from dual;

       AGE
----------
        32
```

… that's just the number of months between two dates divided by 12 (and rounded down).

Achieving this same sort of flexibility, along with all the validation required around dates, would be a mammoth task if the dates you are dealing with are stored in VARCHAR2. The DATE datatype makes it easy; no need to extract parts of the date; no need to try and account for how many days in a month, or the number of days in a year; and no need to convert parts to NUMBER datatype try and do numeric calculations and re-incorporate the numbers back into the date.

So, storing dates in a VARCHAR2 datatype is, as you can see, definitely a bad idea, and may introduce bugs or bad data that you won't know about until some user has entered it without any exception raised. Before we look at further aspects of the DATE datatype, let's take a look at how Oracle stores the date values internally…

3. How are DATE values stored?

As you've seen above, storing DATE values as DATE datatype is certainly preferred over using something like VARCHAR2, or any other character based datatype. But how does Oracle store the date values in those bytes?

If you've been observant with my above examples, you may have noticed that the DUMP values that have been produced seem to differ. Let me put those side by side so you can see…

```
SQL> insert into mydatedates (dt) values (trunc(sysdate));

1 row created.

SQL> select dump(trunc(sysdate)) as dmp, dump(dt) as dmp from mydatedates;

DMP                                  DMP
------------------------------------ ------------------------------------
Typ=13 Len=8: 223,7,12,17,0,0,0,0    Typ=12 Len=7: 120,115,12,17,1,1,1
```

Whilst both of these are DATE datatypes, the first of them shows an type number of 13 and length of 8 bytes, and the second one has an type number of 12 and length of 7. So what was the difference between the two dates we supplied?

The first of those dates is a date value supplied directly to the DUMP function, and the second is a stored DATE datatype on a table column.

The first you may find referred to as the external C representation of a Date or "External Type 13", which you won't easily find listed in the Internal Datatypes in Oracle documentation (because it's not there!). It is typically used behind the scenes of PL/SQL code and, as seen here, when dumping a non-stored date in SQL. In short, the byte representation of Type 13 dates can be looked at as:

Byte 1	low order byte of year
Byte 2	high order byte of year
Byte 3	month
Byte 4	day
Byte 5	hours
Byte 6	minutes
Byte 7	seconds
Byte 8	unused

All of those are fairly straightforward except for Bytes 1 and 2. However it will be clearer if we take the first two bytes of our Type 13 value and do the following:

(Byte 2 * 256) + Byte 1

giving us…

(7 *256) + 223 = 2015

That's the year! or will be if I finish this article before 2016 ;)

The one thing to note here is that you won't really see much of Type 13, certainly not in relation to the data you have stored on the database, as that is the Type 12 internal format. However, the format of Type 12 isn't the same as Type 13. So what are the 7 bytes of the Type 12 Date, let's take a look…

Byte 1	century + 100
Byte 2	year + 100
Byte 3	month
Byte 4	day
Byte 5	hours + 1
Byte 6	minutes + 1
Byte 7	seconds + 1

Thus our date (and time) is:

Year = ((Byte1-100)*100)+(Byte2-100)
Month = Byte3
Day = Byte4
Hours = Byte5-1
Minutes = Byte6-1
Seconds = Byte7-1

From our data above we get:

Year = ((120-100)*100)+(115-100) = 2015
Month = 12
Day = 17
Hours = 0
Minutes = 0
Seconds = 0

So, there's not much to the storage of DATEs within Oracle. It uses a proprietary byte format to store the values, which allows Oracle to perform its internal date calculations and validation as effectively as possible.

It doesn't matter whether you want your dates displaying in "DD/MM/YYYY HH24:MI:SS" format or in "Month DD YYYY HH:MI AM" format (or whatever you choose), the actual storage of dates in the database is always in the same 7 byte structure of Type 12.

The key thing is that **HOW YOU CHOOSE TO DISPLAY YOUR DATE HAS NO BEARING ON HOW THE DATE IS (OR SHOULD BE) STORED**.

For those with access to MOS:

"How does Oracle store the DATE datatype internally?" (Doc ID 69028.1)
https://support.oracle.com/epmos/faces/DocumentDisplay?_afrLoop=466847260824172&id=69028.1

To conclude this article, a handful of common issues that we see on the community…

4. NLS_DATE_FORMAT

In close relation to this article, and one of the reasons that some people seem to seek out how to store their dates in a particular format, is the effect of the NLS_DATE_FORMAT parameter. It's surprising how many developers (even some with apparently years of experience) aren't even aware of this parameter or how it can be used or overridden, so, when these developers are querying their DATE data, they see the results come out in a particular format and it's not the format they want… so they then start the elusive search for how to store it differently. For those who don't know about it, here's a quick overview…

The NLS_DATE_FORMAT is one of the database parameters, created when the database is installed.

```
SQL> select name, value from v$parameter where name like 'nls_date_for%';

NAME                                                                             VALUE
-------------------------------------------------------------------------------- ------------
nls_date_format                                                                  DD-MON-RR
```

(*run as the SYS user*)

…and by querying a date in the database, it will appear in that format by default…

```
SQL> select sysdate from dual;

SYSDATE
---------
18-DEC-15
```

As you can see, on a regular installation of the database, the default format isn't ideal, as it's using 2 digit years. (If you don't know what the format is you could assume that to be 15th December 2018 or 18th December 2015 – which we can obviously figure out in this particular case (as it's not 2018 yet!) but with other dates we likely wouldn't know)

Now, you'll be saying to yourself… "When I query a date I don't get it in that format?"
Sure. You may see the date format differently. That's going to be because of 1 of 3 reasons…

1) Your DBA has configured the database with a different NLS_DATE_FORMAT setting.
2) Your client computer's operating system has set the NLS_DATE_FORMAT as an environment variable.
3) Your client software may have set the NLS_DATE_FORMAT for the database session that's connected.

I've given you these in the order that they can override each other. Firstly the database parameter dictates the format, but if it's been set at the operating system level as an environment variable (how this is set varies depending on whether you are using Unix or Windows), then this will override the database setting, and lastly, if the database session you are connected with has set the parameter for that session, that will override the operating system setting.

Most commonly you will see the last one being used. You may have even noticed me already use this within this article…

```
SQL> alter session set nls_date_format = 'DD/MM/YYYY HH24:MI:SS';

Session altered.

SQL> select sysdate from dual;

SYSDATE
-------------------
18/12/2015 09:07:50
```

This above is being shown in SQL*Plus, but any client can usually set this in one way or another, either within the SQL script in the same way, or via some client interface settings.

The point here is that the NLS_DATE_FORMAT setting dictates how the DATE datatype is displayed to an end user (even if that end user is you as a developer), when a date is queried directly.

So, now you're thinking "Great, I'll use NLS_DATE_FORMAT in my code to change the format of my dates". Please! **DO NOT DO THAT**! Why not… let's look…

In my current session I have my date format set to a standard that I need for my work (or whatever purpose)…

```
SQL> select sysdate from dual;

SYSDATE
----------
18/12/2015
```

Now some clever person has created a function to provide a formatted string for a report for a given employee…

```
SQL> create or replace function formatted_salary(empno in number) return varchar2 is
  2    salaryString varchar2(4000);
  3  begin
  4    execute immediate 'alter session set nls_date_format = ''DD fmMonth YYYY''';
  5    select 'Employee: '||ename||' (Hired: '||hiredate||') - Salary £'||to_char(sal,'fm99,999')
  6    into   salaryString
  7    from   emp
  8    where  empno = formatted_salary.empno;
  9    return salaryString;
 10  exception
 11    when no_data_found then
 12      return 'Employee Not Found';
 13  end;
 14  /

Function created.

SQL> select formatted_salary(7788) from dual;

FORMATTED_SALARY(7788)
--------------------------------------------------------------------------------
Employee: SCOTT (Hired: 19 April 1987) - Salary £3,000
```

Well, that works, and gives the output required for that specific report.
Now, what about my session…

```
SQL> select sysdate from dual;

SYSDATE
-----------------
18 December 2015
```

It's been changed, and is going to affect anything I do in this session, or my enjoyment for browsing my data. ☺

Now the truth is, my own work shouldn't be relying on the NLS_DATE_FORMAT parameter anyway. If I need dates specified in a particular format, I should explicitly state the format I want to use, either with the TO_CHAR or TO_DATE functions (see section below on these functions). Likewise though, that function should explicitly convert the date to a string using the format it requires and the TO_CHAR function, rather than trying to alter the session's parameters.

My advice is, **DO NOT RELY ON NLS_DATE_FORMAT PARAMETER SETTINGS. ALWAYS USE EXPLICIT DATE FORMATTING**.

5. NLS_DATE_LANGUAGE

Now this doesn't crop up too often on the community, but when it does, we've seen some really ingenious (read that as "bad") code. Such questions are like "I'm getting data from an external source that is coming with French dates, and I need to get these into my database". E.g.

```
SQL> insert into mydatedates (dt) values (to_date('31-MAI-2010','DD-MON-YYYY'));
insert into mydatedates (dt) values (to_date('31-MAI-2010','DD-MON-YYYY'))
                                             *
ERROR at line 1:
ORA-01843: not a valid month
```

Then we get some amazingly bizarre responses where people try and split up the strings of the dates, and create case statements that try and convert the one language to English, then recombine that with the original values to actually create the date. The fact is that Oracle is more intelligent than that, and is quite capable of dealing with different languages, if you just know how.

If I go and look at my NLS_DATE% parameters I'll see I've got more than just NLS_DATE_FORMAT…

```
SQL> select name, value from v$parameter where name like 'nls_date%';

NAME                                                                             VALUE
-------------------------------------------------------------------------------- -----------
nls_date_language                                                                ENGLISH
nls_date_format                                                                  DD-MON-RR
```

… I also have an NLS_DATE_LANGUAGE. In my case this defaults to English, yours may be different.

So, how can I now insert my French date?
Well, I could alter my session parameter for this…

```
SQL> alter session set nls_date_language = 'french';

Session altered.

SQL> insert into mydatedates (dt) values (to_date('31-MAI-2010','DD-MON-YYYY'));

1 row created.
```

… and, as if by magic, Oracle now knows our French language.

However, as we saw with NLS_DATE_FORMAT, altering our session is not an ideal thing to do… especially if I now want to see my data in English…

```
SQL> alter session set nls_date_format = 'DD-MON-YYYY';

Session altered.

SQL> select * from mydatedates;

        ID DT
---------- -------------
         5 17-DΓC. -2015
         6 31-MAI  -2010
```

Apart from a conflict with my current client's language/characterset, I'm getting the dates shown in French, but hey I only wanted to insert some French dates, I don't want my whole session in French. So let's set things back to English, and look at another way of inserting the French data…

The function TO_DATE provides a 3rd optional argument that allows us to specify influencing parameters, such as our NLS_DATE_LANGUAGE. Let's see if that works…

```
SQL> alter session set nls_date_language = 'English';

Session altered.

SQL> select * from mydatedates;

        ID DT
---------- -----------
         5 17-DEC-2015
         6 31-MAY-2010

SQL> insert into mydatedates (dt)
  2  values (to_date('02-MAI-2010', 'DD-MON-YYYY', 'nls_date_language=french'));

1 row created.

SQL> select * from mydatedates;

        ID DT
---------- -----------
         7 02-MAY-2010
         5 17-DEC-2015
         6 31-MAY-2010
```

Well, that's better. I can insert a French string date, and specify that it should use an NLS_DATE_LANGUAGE of "French" without having to alter the parameter within my session or effect anything else. Instant translation from French to English.

And likewise, when querying data, you can do the same with the TO_CHAR function to translate between languages (let's try Irish for a change)…

```
SQL> select to_char(dt, 'fmDD Month YYYY', 'nls_date_language=irish') from mydatedates;

TO_CHAR(DT,'FMDDMONTHYYY
------------------------
2 Bealtaine 2010
17 Nollaig 2015
31 Bealtaine 2010
```

So again, my advice is, **DO NOT RELY ON A SESSIONS NLS PARAMETER SETTINGS. ALWAYS BE EXPLICIT WHERE YOU NEED TO SPECIFY IT**.

6. TO_CHAR AND TO_DATE

Ok, I really thought I wouldn't have to put this section in to such an article, but even recently on the community we keep on getting people who mis-use these functions, clearly showing they don't understand the difference or purpose of the DATE and VARCHAR2 datatypes. The number of times we see people doing things like…

```
SQL> select to_date(sysdate, 'DD/MM/YYYY') from dual;

TO_DATE(SYS
-----------
18-DEC-2015
```

… or …

```
SQL> select to_date(sysdate, 'Month DD, YYYY') from dual;
select to_date(sysdate, 'Month DD, YYYY') from dual
               *
ERROR at line 1:
ORA-01843: not a valid month
```

…and questioning why the date isn't being formatted as they've specified.

So, let's put this in simple terms…

- TO_DATE accepts a VARCHAR2 parameter in the format specified and converts it to a DATE datatype.
- TO_CHAR accepts a DATE parameter (in this context) and converts it to a VARCHAR2 datatype in the format specified.

I've highlighted these two things, as it's so important, yet so many people get it wrong.

You can see from these that the "format" of the date, in both cases, relates to the VARCHAR2 datatype. Thus, in the first, the format is telling the TO_DATE function what format the incoming string is in. In the second it's the opposite, it's telling the TO_CHAR function what format to return the string as.

In the above, failed code, the reason that it's not working is because we (well not me; you wouldn't catch me doing that) are providing the TO_DATE function with a value that is already a DATE datatype. As we know from this whole document, the DATE datatype has a single internal format and no amount of TO_DATEing it will change that format. So, what happens when we do the above code… Oracle says "Ah! you've passed a DATE datatype, and I'm expecting a VARCHAR2, but I can implicitly convert that for you as I have something called the NLS_DATE_FORMAT parameter that tells me what format to use to convert DATEs to VARCHAR2 strings, so I'll do that datatype change for you and pass the resultant VARCHAR2 string to the TO_DATE function instead."

In the first failed example, we still got a result, though it wasn't the right format. Oracle had implicitly converted the input date (SYSDATE) to a string using my NLS_DATE_FORMAT setting (DD-MON-YYYY), resulting internally in TO_DATE being called like this:

```
select to_date('18-DEC-2015', 'DD/MM/YYYY') from dual;
```

Now, you may be saying "But hang on, 18-DEC-2015 isn't in the format DD/MM/YYYY, so that should give an exception. Nope, Oracle's smarter than that. It doesn't care too much about the characters you choose to use separate the date components, so whether you specify "-" or "/" or "." or whatever it just recognises those as breaks in the format. After that the "DD" part matches ok and the "YYYY" part matches ok. The only surprise is that it is also quite happy to match "DEC" with the date format of "MM" which is usually the two numerical digit version of the month. In reality, it's just being clever… it knows you're supplying a month whether you've provided it as "12" as "DEC" or "December" etc.

However, the second example failed with an exception.

```
SQL> select to_date(sysdate, 'Month DD, YYYY') from dual;
select to_date(sysdate, 'Month DD, YYYY') from dual
               *
ERROR at line 1:
ORA-01843: not a valid month
```

Applying the same principle, an implicit conversion has taken place to convert the DATE datatype to a VARCHAR2 datatype before passing it in to the TO_DATE function. That results in…

```
select to_date('18-DEC-2015', 'Month DD, YYYY') from dual;
```

Now, while Oracle is clever, and it can pick out the year, it starts to struggle if the components aren't in the same order. So the 18 is assumed to be the month, and the "DEC" is assumes to be the day… and obviously confusion ensues.

Now, it is surely becoming clearer to you (if not already) that the correct way to convert a DATE datatype to a VARCHAR2 datatype isn't to use the TO_DATE function, but is rather to use the TO_CHAR function.

```
SQL> select to_char(sysdate, 'DD/MM/YYYY') from dual;

TO_CHAR(SY
----------
18/12/2015

SQL> select to_char(sysdate, 'fmMonth DD, YYYY') from dual;

TO_CHAR(SYSDATE,'F
------------------
December 18, 2015
```

Giving us the dates converted to VARCHAR2 strings in the format we asked for. And THAT is how you turn your DATE values into VARCHAR2 values for displaying on the client application(s).

The clue here is in the name of the functions. Trying to convert something to a DATE when it's already a DATE is stupid. Likewise, if you try and convert something that is a VARCHAR2 when it's already a VARCHAR2 will give you issues (just as stupid). Use the right function for the right datatype, and for the right purpose.

7. ANSI DATE LITERALS

What is an ANSI date literal you ask? (Ok, well you won't if you already know about it)

You'll probably have noticed in the first sections of this article, I gave examples such as…

```
SQL> select date '2015-12-17' as dt, dump(date '2015-12-17') as dmp from dual;

DT         DMP
---------- --------------------------------
17/12/2015 Typ=13 Len=8: 223,7,12,17,0,0,0,0
```

… in order to demonstrate the DATE datatype.

But hang on, I didn't use the TO_DATE function to convert my strings into DATE datatypes! So are they really Dates?
No, I didn't, and yes they are. I used ANSI Date Literals, by specifying the "date" keyword before my string.

Before you go running off and trying to put "date" before all your date strings, just read the following to ensure you understand the limitations of these literals…

1) ANSI date literals only work to specify a DATE, **they can NOT include a time** like you can with TO_DATE.
2) ANSI date literals **must specify the string in YYYY-MM-DD format**. No other choice, it has to be in that format.

Primarily they are useful if you are specifying a literal for a date-only string, as they can make your code a little more readable, as it saves you having to specify the date format string as part of e.g. TO_DATE('2015-12-17','YYYY-MM-DD'). In actual production code, there's probably not many parts of your code where you actually need a fixed date literal, so you probably won't find much use for this there, but when you're doing things 'on-the-fly', writing one-off queries or giving example code or date like I've done in this article, it can be very useful for specifying dates quickly and cleanly.

8. TWO DIGIT VS. FOUR DIGIT YEARS

It's shocking to still see how many people try to write code or queries using dates with 2 digit years. You may ask "what's the big deal?", after all you know that a year of 15 must mean 2015… doesn't it? Well, you just can't be sure. Consider a database that stores details of customers, clients, patients or whoever and their dates of birth. If you see a date of:

01/02/05

What date is that?

Ok, so we will likely know that our database is conforming to a particular county's date style, so if I assume I working on a UK database and this is UK format of DD/MM/YY then it's the 1st of February… but what year? Is it safe for me to assume that the year is 2005? Or was it a record of someone born in 1905?

When we worked on database back in the 1990's we were fairly safe as we could be sure that any computerised data relating to people (unless it was specifically a database of historical data) would include years that we could determine. E.g. in 1995, if we saw a year of birth of "05" we could be sure it was 1905, and if we saw a year of birth of "99" we could be sure it was 1899.

However, as we neared the year 2000, it was apparent there was a problem. This data would become ambiguous (and was already doing so), and there was the additional problem that servers that had operating systems and software that also only used 2 digit years would start to have problems when comparing the current date to the data, as most of it would suddenly appear to be in the future. This issue was known as the year 2000 bug or more affectionately the "Y2K bug".

Sure, some people out there think the whole "Y2K bug" thing was a hoax, that it never really existed, and all the computer systems didn't fall flat on their face as the clock ticked over from 1999 to 2000. However, I can assure you, it was very real. I, and thousands of other software developers and systems administrators, worked hard (very hard) in the years approaching 2000 to ensure that databases and software were updated, and operating systems were patched to prevent there being any such problem, and for the most part the hard work paid off; very few systems failed when the time came (and many of those that did fail often just required a manual update of the system date to bring it to 2000 rather than 1900). I personally spent the early hours of 1st January 2000 checking and testing a whole room full of servers to ensure everything was running smoothly… at great expense to... well… my company who had to pay me handsomely for giving up my early new year celebrations.

So, when you write code using dates… do everyone a favour (or favor if you're American) and please, always, use 4 digit years. The last thing we need is developers who weren't there doing all those fixes back in the late 1990's, breaking all the data again for us.

9. Aggregating a Series of Date Ranges

Aggregating or grouping a series of date ranges is something we often see asked on the community. This is something that I've already covered in another community document "Grouping Sequence Ranges", which also covers the grouping of sequences of numbers.

See the document here: https://community.oracle.com/docs/DOC-915680

10. Interval between two dates

Finally, another common question, for some reason, seems to be where people want to know the difference between two dates, expressed as an interval. In most cases, that's not a problem, but it becomes more difficult when we have to explain that you cannot easily specify an interval that spans the "months" and "days" part of an interval. E.g. you can express an interval in terms of "years and months", or in terms of "days, hours, minutes and seconds", but the moment you say you want the interval in "months and days" or "years, months and days" etc. then it becomes a problem… let's look at an example.

Consider we have two dates, 28th February 2014 and 30th July 2015.

We can easily determine how many **days** there are between them…

```
SQL> create table twodates(date1, date2) as (
  2         select date '2014-02-28', date '2015-07-30' from dual
  3       )
  4  /

Table created.

SQL> select date2-date1 as days_between
  2  from   twodates
  3  /

DAYS_BETWEEN
------------
         517
```

Or we could determine the number of **years** between them…

```
SQL> select floor(months_between(date2,date1)/12) as years_between
  2  from   twodates
  3  /

YEARS_BETWEEN
-------------
            1
```

We can even determine the number of **months**, or **years and months** between them…

```
SQL> select floor(months_between(date2,date1)) as mnths_between
  2        ,floor(months_between(date2,date1)/12)||' yr '||
  3         floor(mod(months_between(date2,date1),12))||' months' as yr_and_mnths_between
  4  from   twodates
  5  /

MNTHS_BETWEEN YR_AND_MNTHS_BETWEEN
------------- --------------------------------------------------------------------------------
           17 1 yr 5 months
```

But what happens now if we want **years, months and days** between the two…

```
SQL> ed
Wrote file afiedt.buf

  1  select floor(months_between(date2,date1)) as mnths_between
  2        ,floor(months_between(date2,date1)/12)||' yr '||
  3         floor(mod(months_between(date2,date1),12))||' months '||
  4         floor(mod(months_between(date2,date1),1)*31)||' days' as yr_mnth_day_between
  5* from   twodates
SQL> /

MNTHS_BETWEEN YR_MNTH_DAY_BETWEEN
------------- --------------------------------------------------------------------------------
           17 1 yr 5 months 2 days
```

Well that kind of looks ok for the dates we've given it, though you'll notice that I've made a massive assumption that there are 31 days in a month.

Consider some different dates too…

```
SQL> select date1, date2
  2        ,floor(months_between(date2,date1)) as mnths_between
  3        ,floor(months_between(date2,date1)/12)||' yr '||
  4         floor(mod(months_between(date2,date1),12))||' months '||
  5         floor(mod(months_between(date2,date1),1)*31)||' days' as yr_mnth_day_between
  6  from   twodates
  7  order by date1
  8  /

DATE1       DATE2       MNTHS_BETWEEN YR_MNTH_DAY_BETWEEN
----------- ----------- ------------- ------------------------------------------------------------
28-JAN-2014 28-FEB-2015            13 1 yr 1 months 0 days
29-JAN-2014 28-FEB-2015            12 1 yr 0 months 30 days
30-JAN-2014 28-FEB-2015            12 1 yr 0 months 28 days
31-JAN-2014 28-FEB-2015            13 1 yr 1 months 0 days
```

There's a few issues here:

a) On 28th Jan our query thinks the result is 1 year, 1 month and 0 days because our second date is also a 28th day of the month.
b) On 29th Jan we've dropped to 1 year, 0 months and 30 days, but then on 30th Jan we've jumped a day to it being 1 year, 0 months and 28 days. What happens to an interval of 1 year, 0 months and 29 days?
c) On 31st Jan we've gone back to the result being 1 year, 1 month and 0 days? That's because Oracle gives special consideration for the last day of each month so treats the difference between 31st Jan and 28th Feb as being whole months.

So, what's the best way to find intervals where we need to span the months and days?

Well, there isn't. It doesn't actually make sense to have some predefined way of stipulating how this should work, considering that there is not a fixed number of days in each month. Oracle makes some assumptions (and they're perfectly valid) about the way it treats the last day of the month as a special case, and that may or may not be what you require in particular cases.

The issue is even clearer when you look at Oracles INTERVAL literals:
https://docs.oracle.com/cd/B28359_01/server.111/b28286/sql_elements003.htm

which may be specified as YEAR TO MONTH or as DAY TO SECOND (and combinations within), but you cannot span the MONTH to DAY boundary, for exactly the same reason.

The answer is to define your requirements clearly first, then use the appropriate in-built date functionality to achieve it.

SUMMARY

Hopefully, you can now see the importance of using the DATE datatype (or even TIMESTAMP), when you want to deal with storing date or datetime information; and you recognise that a date "format" is only relevant when you are displaying date information or accepting strings as input. You should almost never have a need to try and store dates as VARCHAR2 strings.

Oracle provides many useful date functions and in-built date calculation abilities, so if you ever find yourself thinking of extracting parts of a date (string) to try and calculate something, just take a moment to step back and re-consider how you can achieve it using DATE functionality instead… e.g.

Instead of this…

```
SQL> select to_date('01-'||to_char(sysdate,'MM-YYYY'),'DD-MM-YYYY') as start_of_month
  2  from   dual;

START_OF_MO
-----------
01-DEC-2015
```

… to try and get to the first day of the current month, you should be considering this…

```
SQL> select trunc(sysdate,'MM')
  2  from   dual;

TRUNC(SYSDA
-----------
01-DEC-2015
```

… completely avoiding any string manipulation or conversion between DATE to VARCHAR and back to DATE again.

THE DATE FUNCTIONALITY IN ORACLE IS POWERFUL. USE IT!

If you're not familiar with the DateTime functions within Oracle, take some time to read through each of them and have a practice to really get a feel for what you can do. They're all nicely documented:

https://docs.oracle.com/database/121/SQLRF/functions002.htm#SQLRF51181