

## Table of Contents

1. Bind variable demo .....	1
1.1. Define a variable, assign a value, and display .....	1
1.2. Using the variable in a row filter .....	2
2. What is happening and some syntax issues .....	3
2.1. Defining a bind variable and data types .....	3
2.2. Assigning a value to a bind variable .....	3
2.3. Unassigned variables .....	4
2.4. Using variables to define other variables .....	4
2.5. Displaying the bind variables .....	5
2.6. Using the bind variable, scope, lifespan .....	5
3. Why does this matter .....	5
4. Some advantages in using variables .....	6
5. How does this work (optional) .....	7

We can use another Oracle technique - declaring and using a variable. Most computer languages have a way to define variables. A variable is a named memory location that can store a value that we can use in our SQL queries. The name/identifier gives us a way to refer to the stored value. To use a variable:

- We need a way to define the variable giving it a name
- We need to consider the data type of the variable
- We need a way to assign a value to that variable and change the value
- We need to consider the scope of the variable- what parts of your code can refer to that variable
- We need to consider the lifetime of the variable- how long does it keep its value

We will limit this discussion to assigning a value to a user-defined variable, here called a bind variable, and then using that variable in an SQL statement.

The variables we are considering here are scalar variables- which means they hold a single data value.

## 1. Bind variable demo

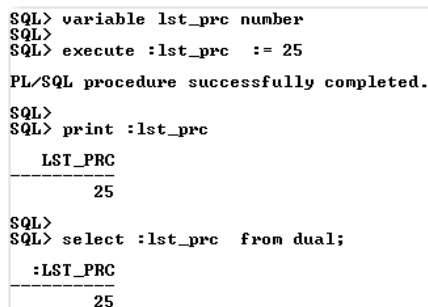
We will start with a demo of a bind variable and then discuss what it is and why you might want to use these.

### 1.1. Define a variable, assign a value, and display

Demo 01: Run the following steps in SQL\*Plus (SQL Developer does not do as well with bind variables)

```
variable lst_prc number
execute :lst_prc := 25
print :lst_prc
select :lst_prc from dual;
```

The following screen shot shows the results of running the above commands in SQL\*Plus.



```
SQL> variable lst_prc number
SQL>
SQL> execute :lst_prc := 25
PL/SQL procedure successfully completed.
SQL>
SQL> print :lst_prc
  LST_PRC
  -----
       25
SQL>
SQL> select :lst_prc from dual;
  :LST_PRC
  -----
       25
```

We are defining a numeric variable, giving it a value and then printing it with a print command and also displaying it with a select query.

## 1.2. Using the variable in a row filter

Now try the following SQL statement.

Demo 02: Use the bind variable in the Where clause

```
select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = :lst_prc
;
```

PROD_ID	PROD_NAME	PROD_LIST_PRICE
1080	Cornpopper	25

The value that was assigned to the bind variable is used in the SQL statement.

Change the bind variable and rerun the sql statement in the sql buffer. We are using the same query with a different value being supplied to the row filter. The slash command in SQL\*Plus reruns the sql statement that is in the sql buffer. Even though we issued a command to set a variable, the SQL buffer is not changed and you can rerun the SQL command with the slash.

Demo 03:

```
execute :lst_prc := 29.95
/
```

PROD_ID	PROD_NAME	prod_list_price
1030	Basketball	29.95
4576	Cosmo cat nip	29.95
4577	Cat leash	29.95

It might not be obvious but you are rerunning the same SQL statement that you ran before, with a different value being used in the filter. The work that Oracle has to do to prepare your SQL statement to be run does not have to be repeated when you use the bind variables.

Demo 04: We could set a variable for a tax rate and then use it in a calculation

```
variable sales_tax_rate number

execute :sales_tax_rate := 0.095;

select prod_id
, quantity_ordered as Quantity
, quoted_price as Price
, quantity_ordered * quoted_price as AmtDue
, quantity_ordered * quoted_price * :sales_tax_rate as SalesTaxDue
from oe_orderDetails
where rownum <=5
;
```

PROD_ID	QUANTITY	PRICE	AMTDUE	SALESTAXDUE
1020	5	12.95	64.75	6.15125
1110	1	49.99	49.99	4.74905
1020	3	12.95	38.85	3.69075
2747	3	12.95	38.85	3.69075
1080	2	22.5	45	4.275

## 2. What is happening and some syntax issues

A variable is a named location in memory that can hold a data value. The bind variable is declared at the SQL\*Plus client level. The variable statement declared a name and a data type for the variable. This is not an SQL statement; it is an SQL\*Plus command- note that it did not end with a semicolon.

The next statement used an execute command to assign a value to that bind variable. The message that was provided indicated that this ran a PL/SQL procedure. We have not talked about PL/SQL yet but it is a programming language provided by Oracle. Two things to note about the syntax:

- in this statement we had to prepend a colon to the variable name so that it is not confused with other identifiers that we might have.
- the assignment operator in PL/SQL is written with a colon and an equals character `:=`

Now that the variable has a value, we can print it. Print is a SQL\*Plus level command; we do not need a semicolon at the end of this command (it is not an SQL statements) and we do not need the colon in front of the variable at the SQL\*Plus level but we can include it.

Then we use that variable (with the colon) in an SQL statement.

So this variable is being passed back and forth between SQL\*Plus and the Oracle engines which run SQL and PL/SQL. The variable could also get its value from an application program that connects with the database.

### 2.1. Defining a bind variable and data types

To declare a variable you need to supply a name and a data type. The command to define a variable is `variable`.

The data types can be one of these (there are others we won't discuss); date is not an allowed type here.

```
number
char
char(99)
varchar2(99)
```

You cannot provide a precision for number; you must provide a length for varchar2; you may supply a length for char- it defaults to one character.

By itself, the command `variable` will show you the names and data types of your bind variables.

### 2.2. Assigning a value to a bind variable

If you need to assign a value to a single bind variable use the execute command as shown above. This has the advantage of not changing your SQL buffer contents.

If you have several bind variables to assign, you can use the following syntax.

Demo 05: Using an anonymous block to assign a value to the variable.

```
variable lst_prc number
variable catg varchar2(10)

begin
  :lst_prc := 15.987;
  :catg := 'APL';
end;
/
```

The syntax here is that we start with the keyword `begin`. Each assignment is done on a new line which ends with a semicolon. Then we finish with the word `end` followed by a semicolon and a slash to run this section of code. However this does replace the SQL buffer with this PL/SQL block of code.

```
select :lst_prc, :catg from dual;

-----
15.987 APL
```

Demo 06:

```
variable target varchar2(20)
exec :target := 'Shingler Hammer';

select prod_id, prod_list_price, prod_name
from prd_products
where prod_name = :target;
```

PROD_ID	PROD_LIST_PRICE	PROD_NAME
5005	45	Shingler Hammer

### 2.3. Unassigned variables

Demo 07: Using a variable that has been declared but not assigned a value gives us a null

```
variable quantity number
print :quantity
```

QUANTITY
-----

```
select :quantity from dual;
```

:QUANTITY
-----

### 2.4. Using variables to define other variables

Demo 08: You can use one variable to define another.

```
variable varb number;
variable varb_2 number;

exec :varb := 45 * 3;
exec :varb_2 := :varb + 100;
```

```
select :varb, :varb_2 from dual;
```

:VARB	:VARB_2
-----	-----
135	235

Demo 09: This builds up a variable to use with a Like operator.

```
variable target varchar2(20);
variable target2 varchar2(20);
exec :target := 'Hammer';
exec :target2 := '%' || :target || '%';
```

```
select prod_id, prod_list_price, prod_name
from prd_products
where prod name Like :target2;
```

PROD_ID	PROD_LIST_PRICE	PROD_NAME
5002	23	Ball-Peen Hammer
5004	15	Dead Blow Hammer
5005	45	Shingler Hammer

## 2.5. Displaying the bind variables

You can use the print command to show a single bind variable or use the print command by itself to show the values of all of your bind variables

variable

```
variable  sales_tax_rate
datatype  NUMBER

variable  lst_prc
datatype  NUMBER

variable  quantity
datatype  NUMBER

variable  catg
datatype  VARCHAR2(10)
```

print

```
SALES_TAX_RATE
-----
          .095

      LST_PRC
-----
      15.987

    QUANTITY
-----

CATG
-----
APL
```

## 2.6. Using the bind variable, scope, lifespan

When you use the bind variable in an sql statement, remember to refer to it with a leading colon. You can use the bind variable in Where clauses, Select clauses, calculated columns, Insert statements, etc.

You cannot use bind variables for things such as table names or columns names- only for the literals.

Although you won't see any difference in speed of execution with our small tables and single user demo accounts, the difference in a production system can be significant when you use variables.

Here we are assigning values to the variables; commonly the values for these variables would come from the application level programs.

Lifespan: The bind variables are part of your session; they disappear when you close your session. If there is a need to keep the values of these variables, you should store them in a table created for that purpose.

Scope: Other people who are running other sessions cannot see your bind variables.

## 3. Why does this matter

Suppose you have an application running where users were running that sql statement multiple times with different literals for the price.

```
select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = 25;
```

```
select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = 500;
```

```
select prod_id, prod_name, prod_list_price
from prd_products
where prod_list_price = 150;
```

The way Oracle works is that each of these SQL statements would be seen as a brand new statement and Oracle would do a lot of work for each of these to create an execution plan to run this efficiently. This takes time to do. Also each of these SQL statements in its parsed form would be stored in memory (in the shared pool) in case someone needed to use it again; if the statement were issued again then Oracle has less work to do in terms of setting the statement up to run. But since we have different numbers in each statement, Oracle sees these as different statements.

But we can abstract this a bit- these are all the same SQL statement- they just have different numbers for the filter test. We can let Oracle build the execution plan for the SQL statement with the bind variable and set that up in memory and when the statement is executed, the bind variable is replaced with its current value. This is more efficient.

## 4. Some advantages in using variables

One advantage in the use of variables in your queries is when you need to test your queries for different values for the filter tests.

1) Suppose you have a long complex query that needs to test the product category value several times. If you store the value for the product category in a variable, then you can change that variable one time and have all references to it in the query change. If you wrote the literal value of the category in the query, then you have to find and change each of those values.

2) Suppose you had a query that needs to filter on the list price of an item and also filter on the quantity of the item. And your first test was

```
Where prod_list_price > 50 and quantity > 50;
```

If you then need to change the filter for quantity to 60 ( and remember you have this test several times in your query) you have to be certain to change only the quantity test and not the price test.

Setting up two variables would make maintenance easier and less error prone.

```
Where prod_list_price > :priceLimit and quantity > :quantityLimit
```

3) Another example you will have in assignments is a filter that tests a date value compared to the current date. We have not discussed date expressions to any detail yet, but suppose you have this test. We want to find orders with the same month number as the current month. So if we run this query in August, it will find orders placed in August, ignoring the month and year.

```
Where extract ( month from ord_date) = extract ( month from sysdate)
```

How do you test that query for validity if you need to be able to run in at a later time (remember the actual test may be much more complex).

Suppose you set up a variable.

```
Variable dtm varchar2(20);
exec :dtm := sysdate;
```

And use that in your query

```
Where extract ( month from ord_date) = extract ( month from to_date(:dtm))
```

Now you can change the value of the variable to other date values and run the same query using a different "current date".

You might not have noticed that I used to\_date(:dtm) in that Where clause. I could also use cast(:dtm as date). The reason for this is that :dtm was defined as a varchar2 type. The exec took the date value returned by sysdate, in its default format as a string and assigned that string to :dtm. In order to get the month from that string we have to cast it back to a date value. ( I know- it is annoying to have to do that but if you don't ,you get an error message and that is even more annoying.)

4) Using variables can help you think more methodically about your queries. With the previous example you could write:

```
Variable dtm varchar2(20);  
exec :dtm := sysdate;  
Variable curr_month number  
exec :curr_month := extract ( month from to_date(:dtm));
```

and then use the following filter which is closer to the wording of the task.

```
Where month(ord_date) = :curr_month
```

You can also display the value of :curr\_month to check that part of the calculation.

## 5. How does this work (optional)

Parsing an SQL statement makes it ready to be executed. One of the concerns with executing a query against a database is efficiency and the parser tries to find the most efficient technique for executing that query. The first time a query is parsed is called a hard parse and it produces an execution plan. Parsing takes time and computer resources. So the execution plan for the parsed query is stored in memory. If you run the identical query again, the execution plan can be reused if it is still in memory.

But in order to use the execution plan again, the second query has to be identical to the first (even white space counts). Suppose you have a query that counts how many cats we saw at the vet clinic last month. Then you want to run a query that counts how many dogs we saw at the vet clinic last month. That is not the same query and a new execution plan is created.

Now suppose you have a query that counts how many animals we saw last month of type XXX where you will fill in the value for XXX every time you run the query. There can be one execution plan created and reused. This is not going to save any time if the query plan is used only once- but it will if you are running several queries of this type.

The way to handle this is with bind variable- in the discussion above XXX represented the bind variable. Your code has to set up the bind variable, give it a value, and then use it. The bind variable is a place holder for values to be passed during execution.

If you think about the most efficient way for this query to be expected this might depend on the data we have. Suppose that we see mostly cats and dogs and pretty much in the same numbers. But there are a few examinations of hedgehogs. When you are searching for cats or dogs, the use of an index might not be the most efficient way; but you are searching for a hedgehog, then an index will help. So what is the execution plan that is set up? The execution plan is set up with the first query and what Oracle does is sneak a peek at the value of the bind variable the first time it is used and uses that. So if your data has the same relative distribution things will work out ok; if you have these wide variations in distribution this might not be a good way to handle things. Some people avoid bind variables in these cases, forcing the parser to create a new execution plan.

With Oracle 11g there is a new approach. With Oracle if there is an execution plan with a bind variable, the Oracle parser marks it and keeps track of the bind variable values being used to decide if it should consider using a different execution plan. This is done automatically by Oracle 11g.