Sergio G. Sanchez Valencia

**Chapter five**

**Iterative processing with loops**

Oracle can hand thre different iterative processes

* Simple loop
* For Loop
* While Loop

These ones share a similar structure

**Declaring loop:** It is composed by words that initiliaze the loop, the end condition (in this case END LOOP) indicates where a loop finishes.

**Body:** The code is going to be executed in each iteration.

**SIMPLE LOOP**

The general syntax as below

**LOOP**

código a ejecutar

**END LOOP**

To finish it, it’s necessary to agregate the reserved Word **EXIT** or **EXIT WHEN(condition)** otherwise it becomes a infinite loop. This words only can be used inside loops.

SQL does not have a Do while loop, however it can be emulated by a simple loop. To do this we write a **EXIT WHEN** before the **END LOOP**

**WHILE FOR**

This loop executes indefinitely the code while a evaluated condition comes true. This kind of loop shouldn’t be used if it’s known the number of iterations it’s going to be repeated, or even if is not requiered execute code at least once.

The general syntas must be written as next.

**WHILE** condition

**LOOP**

**NUMERIC FOR LOOP**

Hay dos diferentes tipos de ciclos For, el numérico y el cursor. El primero es en donde el número de iteraciones es conocido, especifica el rango de iteraciones que se van a realizar.

SQL has two different types of for loops, numeric and cursor. On first one programmer knows the number of iterations and specifies the range of cycles are going to be executed..

It’s general syntax for booth is:

**FOR** loop index in [REVERSE] inicio .. fin

**LOOP**

code

**END LOOP**

**CONSTRAINTS:**

* You must not declare iteration variable, SQL do this automatically
* Ranges can not be modified when cycle has started
* We use reseverd Word **REVERSED** to decrease from max number to min number

Para ejecuciones que no se requieran de uno en uno, se debe usar una condición dentro del ciclo, ya que este se realiza avanzando de uno en uno, por lo cual si se requiere que se ejecute solo de 2 en dos, se utilizaría el comando **MOD e.g. IF MOD(loop\_index, 2) = 0**

By default Oracle makes iterations one per one, this is to say, it do the number of cycles declared on preamble. For example for execute code each two iterations we use the **MOD** functions, e.g. IF

**EL CICLO CURSOR MOD e.g. IF MOD(loop\_index, 2) = 0 THEN.**

Un ciclo de este tipo está asociado a un cursor o un comando **SELECT** incorporado directamente en la declaración del ciclo. Se usa este ciclo solamente si se necesita encontrar y procesar cada registro de un cursor.

This kind of loop it’s asociated with a **SELECT** command written directally in the loop initialization.

Basic syntax is

**FOR** record **IN** { cursor\_name | (explicit **SELECT** statement) }

**LOOP**

código a ejecutar

**END LOOP**

Also , we can use **CONTINUE** or **CONTINUE WHEN** this is for skip or continue the cycle. We can use this to change from a loop to another, for this it’s necessary they be nested.

**Chapter Eight**

**Strings**

En SQL existen distintos tipos de datos para los Strings, sin embargo los más importantes y más utilizados son los tipos **VARCHAR2** que es el equivalente a un String y el tipo **CHAR** que se puede usar como una cadena de caracteres o como un solo caracter. Sus sintáxis son las siguiente

In SQL language there are different data types for Strings, most known are **VARCHAR2** (this is equivalent to a String in other languages) and **CHAR** type, this can be used as a characters array or as a single character. For both syntax is written like.

name **VARCHAR2** (max\_size [CHAR| Byte])

name **CHAR**(100 char) - - Fixed array size

En el primero se debe especificar el tamaño en Bytes.

Max\_size parameter must be specified on bytes

For comparations between two strings we only use a IF clause, as next.

**IF** cadena1 = cadena2 **THEN**

**String manipulations**

All programming languages has reserved symbols for using in code, but sometimes we need to show them, for this we use the **q** command together with delimiters. E.g. i fis necessary print a String with doublé quotation marks you can write code like this way, **q’{**texto**}’,** or just writing quotation marks twice. Strings can be concatenated using **CONCAT** statement. General syntax is:

**CONCAT**(cadena1, cadena2)

Also you can get the same result by using two vertical bars as follow x := cad1 || cad2 evenfurther is recommended to use **CONCAT** this because EBDIC servers usually have problems with bars.

We can use next functions to formar our texts

* **LOWER(**‘palabra’) - - Convert to Lowercase
* **UPPER**(cadena) - - Convert to Uppercase
* **INITCAP**(‘palabra’) - - Make a first letter Uppercase

For make comparassion between two strings taking out if booth are lowercase, uppercase or combined. We need to make SBDM unsensible. To get this we alter session with respective statement.

**ALTER\_SESSION** **SET** **NLS\_SORT** = BINARY\_CY

**ALTER\_SESSION SET NLS\_COMP** = LINGUISTIC

Asi también se pueden hacer búsquedas extracción y reemplazo mediante las siguientes funciones

Likewise for searching, extraction and replacing exist next functions.

**Searching**

**cont := INSTR**(string, ‘symbol’, position+1)

Is necessary a counter variable for storage array position.

**Extraction**

For this is necessary use loops, where stop condition is when **SUBSTR** function returns 0.

* position\_symbol := **SUBSTR**(fit, ‘symbol’ , position\_symbol);

**Replacing**

* **REPLACE**(string, ‘symbol’, replacing\_symbol)

**Substrate some fragments of a string**

Also is possible extract a number of characters using **SUBSTR** function, for example **SUBSTR(**string, -10)returns last 10 characters.

**TRIMMING**

Las funciones **LTRIM, RTRIM,** y **TRIM** permiten eliminar caracteres que estén demás, tanto por la izquierda o por la derecha o usando **TRIM** para hacerlo por ambos lados. La sintáxis es la siguiente

**LTRIM, RTRIM,** y **TRIM** has been made to extract characters and delete from strings. You can make this by left, right or booth sides. (Left TRIM Right Trim). General syntax as follows.

**TRIM**(char\_to\_erase, place)

**REGULAR EXPRESSIONS**

We can found some patterns inside text, this is useful for example to emails, telephone numbers etc. We achieve this by using

**REGEXP\_INSTR**

General syntax is:

**REGEXP\_INSTR** (source\_string, patter [, start [,ocurrence [,return\_option[, math\_modifier, [, subexpression]]]]])

Also SQL has inside **REGEXP\_REPLACE** which le tus modify text what condition evaluation comes true. General syntax is:

**REGEXP\_REPLACE** (source\_string, pattern [, string\_to\_replace [, possition[, ocurrence [, match\_modifier]])

Además podemos comparar dos cadenas de texto, se debe hace un hincapié en este aspecto ya que una variable **CHAR** y **VARCHAR2** cuando son inicializadas a vacío. Sólo la que es String **(VARCHAR2)** es inicializada a NULL. Sin embargo, no podemos hacer comparaciones de **VARCHAR2** con cadenas **CHAR** si estas no tienen el mismo tamaño, ya que si por ejemplo evaluamos la palabra subsecuencia con una cadena de 15 elementos, los últimos 3 no coincidirán por lo cual, retornará falso.

Finally we if we want to compare we must to take in count variables **CHAR** and **VARCHAR2** when initialize, VARCHAR2 is initialized NULL. Furthermore we can not compare VARCHAR with arrays CHAR if booth are not the same size. For example, if we evaluates subsequence, with a string and an array of 15 elements, last thre ítems will not match, then result will be False.

**Chapter Nine**

**Numbers**

On SQL language exist five different data types, NUMBER, PLS\_INTEGER and BINARY INTEGER, SIMPLE\_INTEGER, BINARY\_FLOAT and BINARY\_DOUBLE, SIMPLE\_FLOAT and BINARY\_DOUBLE y SIMPLE\_FLOAT and SIMPLE\_DOUBLE.

**NUMBER:** It is the simplest numerical data type we found on SQL, this works for integers as floats, and it can storage values with a maximum value of 40 digits for any number of integers, for floating number, size is fit to value is assigned. Ranges are from 10E-130 to

0^125 - (1.9E126-1). The general sintaxis is show as below.

**NUMBER (precission, scale);**

**PLS\_INTEGER:** This data type allows you to storage signed integers in range from 2,147,483 648 to 2, 147, 483, 647. Using this, has some advantages inasmuch as it allows portability on hardware. It is preferable to use the specific type for each numerical data type.

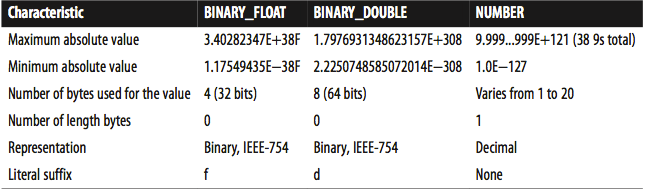
**BINARY\_INTEGER:** The BINARY\_INTEGER datatype also allows you to store signed integers in a binary format. BINARY\_INTEGER is equivalent to PLS\_INTEGER.

**SIMPLE\_INTEGER:**

This datatype is a performance-enhanced version of PLS\_INTEGER with a few caveats. The SIM‐ PLE\_INTEGER datatype has the same range of values as PLS\_INTEGER (−2,147,483,648 through 2,147,483,647), but it does not support NULL values or check for overflow conditions. It is recommended for situation where is such that your variable will never be NULL and will never overflow, then the SIMPLE\_INTEGER type will scream with better performance.

**FLOATING BINARY**: Oracle 10g, introduced to new floating numerical data types. These expanded their range to double range.

For this we can encapsulate these on table below.



These datatypes also have some variations like **BINARY\_FLOAT\_INFINITY** with inifinty precission.

We must to be carefully when using some functions that works with datatypes. Inasmuch as with respect to precission and memory assigned, comparations can be evaluated to False.

**TO\_NUMBER FUNCTION:**

This function converts fixed strings and variable fix, to numbers specifying the data type. The general syntax is:

**TO\_NUMBER (string** [formato [,nls\_params]])

Format parameter is the way it manipulates the string, for example, if function have to start converting from the string start.

We also have **TO\_CHAR** function. It allows you to apply the inverse process and convert a number into a string. However, it is able to handing strings with format, for example.

b := TO\_CHAR(123.01,’L999G999G999D99');

the output is:

$123,456,789.01

**FUNCIÓN CAST**

Es utilizada para convertir números a cadenas y viceversa, como su nombre lo indica, permite realizar un casting común y corriente. Para su uso solo se requiere escribirlo de la siguiente forma

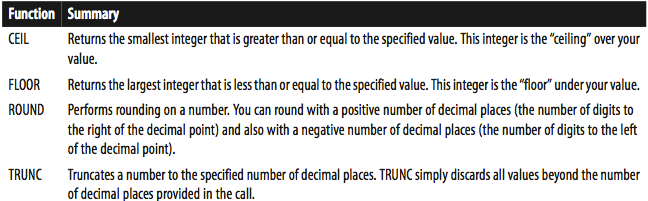
It is used to convert number into strings and vice versa, as name indicates, it is able to cast datatypes. To use this function we only requiere to write it as follows

a := CAST(a AS tipo\_dato);

However this functions has the disadvantage of not supporting the uso of number format model. On the other hand it has the advantage that it is part of the ISO SQL standard.

We also have NUMERIC OPERATORS, they’re as we know, with some differences in exponentiation that it’s represented with double asterisks **\*\*** and for not equal values, we use **<>, !=, ~=, ^=.**

Finally for rounding, we use for functions, they’re described in follows.



**Chapter 10**

**Dates and Timestamps**

Most applications requiere manipulations of dates and times, for example if we have a bank and we make several transactions per day and an rob occurs we can estimate how much money we lost, from time transactions. Dates are quite complicated: not only are they highly formatted data, but there are myriad rules for determining valid values and valid calculations (leap days and years, daylight saving time changes, national and company holidays, date ranges, etc.).

For any datetime value, the database stores some or all of the following information:

• Year

• Month

• Day

• Hour

• Minute

• Second

• Time zone region

• Time zone hour offset from UTC

• Time zone minute offset from UTC

TIMESTAMP stores a data and time without respect to time zone. Except for being able to resolve time to the billionth of a second. This is equivalent to DATE. As variants of this we also have “ ” with time zone, and with local time zone.

The nuances of these types, especially the TIMESTAMP WITH LOCAL TIME ZONE type, can be a bit difficult to understand at first. To help illustrate, let’s look at the use of TIMESTAMP WITH LOCAL TIME ZONE in a calendaring application for users across multiple time zones.

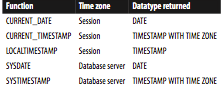
Choosing a Datetime datatype is not a hard work, we can let us lead for this manual.

• Use one of the TIMESTAMP types if you need to track time down to a fraction of a second.

• Use TIMESTAMP WITH LOCAL TIME ZONE if you want the database to auto‐ matically convert a time between the database and session time zones.

• Use TIMESTAMPWITHTIMEZONE if you need to keep track of the session time zone in which the data was entered.

• You can use TIMESTAMP in place of DATE.ATIMESTAMP that does not contain subsecond precision takes up 7 bytes of storage, just like a DATE datatype does. When your TIMESTAMP does contain subsecond data, it takes up 11 bytes of storage.

Oracle have some functions for making queries to TIMESTAMPS they’re return only data types related with dates. On the other hand we must know about the interval DataTypes, it let us record specific points in time.

If we need to change between some data types, we can use functions **CAST and EXTRACT**, these are standard SQL functions that are sometimes useful when you are working with datetimes.

For casting with respecto to date and time, we can use function to

* Convert a character to string to a datetime value
* Convert a datetime value to a character string
* Convert one datetime tyoe to another (e.g Date to TimeStamp)

When it is used to convert datetimes to and from character strins, CAST respects the NLS parameters. We need to change the NLS settings, they can be:

**NLS\_DATE\_FORMAT**

When casting to or from a DATE

**NLS\_TIMESTAMP\_FORMAT**

When casting to or from a TIMESTAMP or a TIMESTAMP WITH LOCAL TIME ZONE

**NLS\_TIMESTAMP\_TZ\_FORMAT**

When casting to or from a TIMESTAMP WITH TIME ZONE

**THE EXTRACT FUNCTION:**

This functions is used to extract date components from a datetime value the general syntax is invoked with the reserved word, EXTRACT as follows.

EXTRACT (*component\_name*, FROM {*datetime* | *interval*})

Years, months, days, hours, minutes, seconds, TIMEZONE\_HOUR, TIMEZONE\_MINUT return a NUMBER only TIMEZONE\_REGION and TIMEZONE\_ABBR returns strings.

E.g.

**IF EXTRACT** (MONTH FROM SYSDATE) = 11 **THEN**

**DBMS\_OUTPUT.PUT\_LINE(**'It is November');

**ELSE**

**DBMS\_OUTPUT.PUT\_LINE**('It is not November');

We also can use date arithmetic with DateTypes, for example if we need to express and hour into the London timezone we use SYSDATe + 6.

**MIXING DATES AND TIMESTAMPS**

The result of a subtraction involving two **TIMESTAMP**s is a value of type **INTERVAL DAY TO SECOND**. The result of a subtraction involving two DATEs is a numeric value. Consequently, if you want to subtract one DATE from another and return an **INTERVAL DAY TO SECOND** value, you will need to **CAST** your DATEs into **TIMESTAMPs**.

Unlike the case with datetime values, it makes perfect sense to add one interval to an‐ other. It also makes sense to subtract one interval from another. The one rule you need to keep in mind is that whenever you add or subtract two intervals, they must be of the same type.

**Chapter 17**

**Procedures, Functions, and parameters**

We’ve analized some code but no the way it is organized, for example we use **modularization**. In the process by which you break up large blocks of code into smaller pieces (modules) that can be called by other modules. Modularization of code is analogous to normalization of data, with many of the same benefits and a few additional advantages. As more reusable, manageable, readable and also reliable.

PL/SQL offers the following types of structure that modularize your code in different ways:

**PROCEDURE**

A program that performs one or more actions and is called as an executable PL/SQL statement. You can pass information into and out of a procedure through its parameter list.

We use this general syntax as follow

**CREATE OR REPLACE PROCEDURE** [*schema*.]*name*[( *parameter*[, *parameter*...] ) ] [AUTHID DEFINER | CURRENT\_USER ] [ACCESSIBLE BY (*program\_unit\_list*)]

**IS** [*declarations*]

**BEGIN**

*executable statements*

[ **EXCEPTION** *exception handlers*]

**END** [*name*];

To execute we call it with EXEC and the name of procedure, or in a BEGIN AND END we just write the name and parameters. As it is a procedure.

**FUNCTION**

A program that returns data through its RETURN clause, and is used just like a PL/ SQL expression. You can pass information into a function through its parameter list. You can also pass information out via the parameter list, but this is generally considered a bad practice. As it is a function it necessary needs to return something with the **RETURN** clause.

It is really necessary to know the schema of functions ‘cause it can save time for you.

**FUNCTION** [schema.]name[( parameter[, parameter...] ) ] **RETURN** return\_datatype

[AUTHID DEFINER | CURRENT\_USER]

[DETERMINISTIC]

[PARALLEL\_ENABLE ...]

[PIPELINED] [RESULT\_CACHE ...]

[ACCESSIBLE BY (program\_unit\_list) [AGGREGATE ...]

[EXTERNAL ...]

**IS**

[declaration statements] BEGIN

executable statements

**[EXCEPTION** exception handler statements]

**END** [name];

Where:

**AUTHID clause:** Determines whether the function will execute with the privileges of the definer (owner) of the procedure or of the current user. The former (the default) is known as the definer rights model, the latter as the invoker rights model.

**DETERMINISTIC** **clause:** Defines this function to be deterministic, which means that the value returned by the function is determined completely by the argument values. If you include this clause, you will be able to use the function in a function-based index and the SQL engine may be able to optimize execution of the function when it is called inside queries..

**PARALLEL\_ENABLE clause**

Is an optimization hint that enables the function to be executed in parallel when called from within a SELECT statement.

**PIPELINED clause**

Specifies that the results of this table function should be returned iteratively via the PIPE ROW command.

**RESULT\_CACHE clause**

Specifies that the input values and result of this function should be stored in the new function result cache.

**AGGREGATE clause**

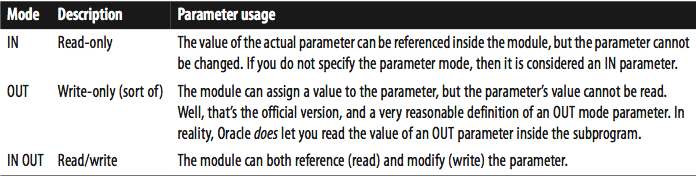
Is used when you are defining your own aggregate function.

**exception handler statements**

Specify any exception handlers for the function. If you do not explicitly handle any exceptions, then you can leave out the EXCEPTION keyword and simply terminate the execution section with the END keyword.

Talking about again the **return** statement it can return anything that be compatible with the return statement indicated in function declaration.

Passing parameters to a functions, you sometimes can specify the way they’re treated.



**DATABASE TRIGGER**

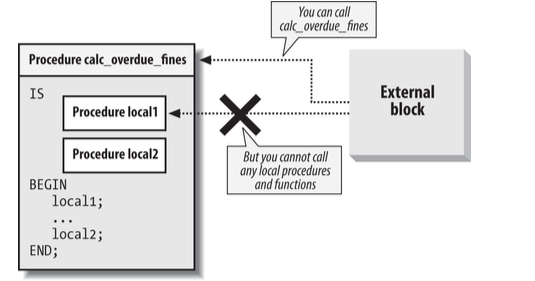
This is composed by a set of commands that are triggered to execute (e.g., log in, modify a row in a table, execute a DDL statement) when an event occurs in the database.

**PACKAGE**

A named collection of procedures, functions, types, and variables. A package is not really a module (it’s more of a metamodule), but it is so closely related that I mention it here.

Object type or instance of an object type

On the other we have modules, a derivation this we can find local or nested modules. In resume it is a procedure or function that is defined in the declaration section of a PL&SQL block (anonymus or named). This module is considered local because it is defined only within the parent PL/SQL block. It cannot be called by any other PL/SQL blocks defined outside that enclosing block.

The image shows this.

Local modules must be located after all of the other declaration statements in the dec‐ laration section. You must declare your variables, cursors, exceptions, types, records, tables, and so on before you type in the first PROCEDURE or FUNCTION keyword.

**SUBPROGRAM OVERLOADING**

When more than one subprogram in the same scope shares the same name, the sub‐ programs are said to be overloaded. PL/SQL supports the overloading of procedures and functions in the declaration section of a block (named or anonymous), package specifications and bodies, and object type definitions. Overloading is a very powerful feature, and you should exploit it fully to improve the usability of your software.

SDBM infers what function you want to use, checking parameters you’re sending on, nevertheless althought is possible functions can be overloaded, parameters can not be the same type, because, the SDBM could call one you don’t want to.

There are three different scenarios that benefit from overloading:

**Supporting many data combinations**

When you are applying the same action to different kinds or combinations of data, overloading does not provide a single name for different activities so much as it provides different ways of requesting the same activity. This is the most common motivation for overloading.

**Fitting the program to the user**

To make your code as useful as possible, you may construct different versions of the same program that correspond to different patterns of use. This often involves overloading functions and procedures. A good indicator of the need for this form of overloading is when you find yourself writing unnecessary code. For example, when working with DBMS\_SQL, you will call the DBMS\_SQL.EXECUTE function, but for DDL statements, the value returned by this function is irrelevant.

**Overloading by type, not value**

This is the least common application of overloading. In this scenario, you use the type of data, not its value, to determine which of the overloaded programs should be executed. This really comes in handy only when you are writing very generic software. DBMS\_SQL.DEFINE\_COLUMN is a good example of this approach to overloading. You need to tell DBMS\_SQL the type of each of your columns being selected from the dynamic query.

Oracle implements also streaming and pipelined function. First accepts as a parameter a result (set via cursor expression) and returns a result ser in the form collection. The second one, is a table function that returns a set as a collection but does so asynchronously to the termination function.

By last a deterministic functions, is considered to be deterministic if it returns the same result value when is called.