

## WRITING BASIC SQL STATEMENTS -- CHAPTER 1 --

- **SELECT** statement retrieves information from the database by doing one of the following: (1) Selection to choose the *rows* in a table that you want returned by a query, (2) Projection to choose the *columns* in a table that you want returned by the a query, (3) Join to bring together data that is stored in different tables by creating a link between them.
- In its simplest form, a **SELECT** statement must include a **SELECT** clause, which specified the columns to be displayed, and a **FROM** clause, which specifies the table containing the columns listed in the **SELECT** clause.

```
SELECT      [DISTINCT] {*, COLUMN [alias],...}
FROM table;
```

- In the syntax:
  - **SELECT** is a list of one of more columns
  - **DISTINCT** suppresses duplicates
  - **\*** selects all columns
  - **column** selects the named column
  - **alias** gives selected columns different headings
  - **FROM table** specifies the table containing the columns
- Writing SQL statements:
  - SQL statements are not case sensitive.
  - SQL statements can be one or more lines.
  - Keywords cannot be abbreviated or split
  - Clauses are usually placed on separate lines
  - Tabs and indents are used to enhance readability.
- When selecting multiple columns from a table, these must be delimited by a comma:

```
SQL> select deptno, dname
      2  from dept;
```

DEPTNO	DNAME
10	ACCOUNTING
20	RESEARCH
30	SALES
40	OPERATIONS
50	DEVELOPMENT

\* Note that the order of the column names does affect the order in which the output is produced...if se say select dname, deptno..., then the dname column will appear first.

- Using (**\***), we can select all columns simultaneously:

```
SQL> select *
      2  from dept;
```

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON
50	DEVELOPMENT	DETROIT

- Note the use of an ALIAS (Victor3434) to manipulate the column headings in the output display:

```
SQL> select dname Victor3434, loc
2 from dept;
```

VICTOR3434	LOC
ACCOUNTING	NEW YORK
RESEARCH	DALLAS
SALES	CHICAGO
OPERATIONS	BOSTON
DEVELOPMENT	DETROIT

- Using arithmetic expressions in SELECT statements:

```
SQL> select ename, sal, sal+300
2 from emp;
```

ENAME	SAL	SAL+300
SMITH	1500	1800
ALLEN	1000	1300
WARD	1000	1300
JONES	2000	2300
BOBBY	3000	3300

- Note that the resulting column SAL+300 is for display purposes only and does not affect the internal structure of the emp table.
- Order of operations is as usual, \* / + - (left to right) is the order or precedence. Further, operators of the same priority are evaluated from left to right.
- Parenthesis are used to force prioritize evaluation and to clarify statements.
- Here's an example of operator precedence...

```
SQL> select ename, sal, 12*sal+100
2 from emp;
```

ENAME	SAL	12*SAL+100
SMITH	1500	18100
ALLEN	1000	12100
WARD	1000	12100
JONES	2000	24100

- Compare to the following grouping with parenthesis...

```
SQL> select ename, sal, 12*(sal + 100)
2 from emp
3 /
```

ENAME	SAL	12*(SAL+100)
SMITH	1500	19200
ALLEN	1000	13200
WARD	1000	13200

```
JONES          2000          25200
```

- Defining a NULL value: A null value is a value that is unavailable, unassigned, unknown, or inapplicable. Note that a null value is not the same thing as zero.
- Whenever any arithmetic is performed on a value that is NULL, the resulting outcome will also be either NULL or undefined. This eliminates the problem/error that could occur if one tries to divide by zero.
- The **Concatenation** operator is: '||' and is used as follows:

```
SQL> select ename||job AS "Name||Job"  
2 from emp;
```

```
Name||Job  
-----  
SMITHANALYST  
ALLENSALESMAN  
WARDSALESMAN  
JONESMANAGER  
BOBBYSALESMAN  
BLAKEMANAGER
```

- As you can see, the resulting string of ename||job is a concatenated value. In addition, we make use of the alias "Name||Job" and employ the optional 'AS' clause which simply makes for easier readability.
- Literal Character Strings are characters, numbers or Dates included in a SELECT statement. Dates and character literal values must be enclosed inside single quotation marks.
- The following example shows how to use literal character strings...

```
SQL> select ename || ' is a ' || job  
2 from emp;
```

```
ENAME||' ISA' ||JOB  
-----  
SMITH is a ANALYST  
ALLEN is a SALESMAN  
WARD is a SALESMAN  
JONES is a MANAGER  
BOBBY is a SALESMAN
```

- Note that we leave a space before and after 'is a' in order to pad with spaces between the two items.

```
SQL> select rownum||': ' ||ename  
2 from emp;
```

```
ROWNUM||': ' ||ENAME  
-----  
1: SMITH  
2: ALLEN  
3: WARD  
4: JONES  
5: BOBBY  
...
```

- Note our use of keyword rownum which evaluates to the row number of a particular field. Also note that the select statement allows for pre-pending the concatenated value of rownum and string literal ": ".
- Eliminating Duplicate Rows involves the use of the keyword DISTINCT. Note that this keyword must be used immediately following the keyword SELECT. Also, you can specify multiple columns after the DISTINCT qualifier. The DISTINCT qualifier affects all the selected columns, and the result represents a distinct combination of the columns.

- SQL\*Plus commands can be divided into the following main categories:
  - Environment Affects the general behavior of SQL statements for the session
  - Format Formats query results
  - File Manipulation Saves, loads, and runs script files.
  - Execution Sends SQL statements from SQL buffer to Oracle8 Server
  - Edit Modified SQL statements in the buffer
  - Interaction Allows you to create and pass variables to SQL statements, print variable values, and print messages to the screen.
  - Miscellaneous Has various commands to connect to the database, manipulate the SQL\*Plus environment, and display column definitions.
- In order to display the structure of a table... use the keyword “DESC[RIBE]”. In other words, you may use the whole word or simply use the abbreviation “DESC” followed by the table name.

```
SQL> desc emp;
Name                               Null?    Type
-----
EMPNO                               NOT NULL NUMBER(4)
ENAME                               VARCHAR2(10)
JOB                                 VARCHAR2(9)
MGR                                 NUMBER(4)
HIREDATE                            DATE
SAL                                 NUMBER(7,2)
COMM                                NUMBER(7,2)
DEPTNO                              NUMBER(2)
```

- Note that those columns that have NOT NULL means that such column must have a value. In the case above, since empno represents a the primary key of the table, it is required to have a value.
- Some data types used in ORACLE database are as follows:
  - NUMBER(p,s) Number value having a maximum number of *p* digits, the number of digits to the right of the decimal points is *s*.
  - VARCHAR2(s) Variable-length character value of maximum size *s*.
  - DATE Date and time value between January 1, 4712 B.C. and December 31, 9999 A.D.
  - CHAR(s) Fixed-length character value of size *s*.
  -
- SQL\*PLUS Editing Commands:
  - A[PPEND]text Adds text to the end of the current line.
  - C[HANGE]/old/new Change the old text to new in the current line
  - C[HANGE]/text/ Deletes text from the current line
  - CL[EAR]BUFF[ER] Deletes all lines from the SQL buffer.
  - DEL Deletes current line
  - I[NPUT] Inserts an indefinite number of lines.
  - I[NPUT]/text Inserts a line consisting of text
  - L[IST] Lists all lines in the SQL buffer
  - L[IST]/n Lists one line (specified by n)
  - R[UN] Displays and runs the current SQL statement in the buffer
  - n Specifies the line to make the current line
  - n text Replaces line n with text
  - 0 text Inserts a line before line 1
- Before we can use these commands above, note that every time you enter a command, it is stored in the **SQL buffer** where it can be accessed by typing the command ED or EDIT which causes the Notepad editor to open a window in which the contents of the current buffer are stored. Note that if the buffer is empty, Notepad will not be invoked. Also, if

you type a number (if it corresponds to a line number in the SQL\*Plus buffer) and [ENTER], it will SQL\*Plus will display the contents of such line from the buffer. For example, if the buffer contains:

...see below

```
select
ename
from
emp
/
```

Then, entering 4 [ENTER] causes the following to happen...

```
SQL> 4
      4* emp
```

- SQL\*Plus File Commands
  - **SAVE[E]*filename*.[ext][REPL[ACE]APP[END]]** Saves current contents of SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql.
  - **GET *filename*.[ext]** Writes the contents of a previously saved file to the SQL buffer. The default extension is .sql.
  - **STA[RT]*filename*.[ext]** Runs a previously saved command file.
  - **@*filename*** Runs a previously saved command file (same as START).
  - **ED[IT]** Invokes the editor and saves the buffer contents to a file named afiedt.buf.
  - **ED[IT][*filename*.[ext]]** Invokes editor to edit contents of a saved file.
  - **SPO[OL][*filename*.[ext]]OFF|OUT** Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the system printer.
  - **EXIT** Leaves SQL\*Plus.

## RESTRICTING AND SORTING DATA -- CHAPTER 2 --

- Limiting the rows selected involves the use of the special WHERE clause as follows:

```
SELECT      [DISTINCT] {*} | column [alias], ...}
FROM        table
WHERE       condition(s);
```

- The condition above refers to any combination of column names, expressions, constants, and a comparison operator (>, <, =, >=, <=, <> (not equal) etc.).

```
SQL> run
  1  select ename, job, deptno
  2  from emp
  3* where job='SALESMAN'
```

ENAME	JOB	DEPTNO
ALLEN	SALESMAN	30
WARD	SALESMAN	30
BOBBY	SALESMAN	
TURNER	SALESMAN	30
BLACK	SALESMAN	10

- Note that 'SALESMAN' in the where clause IS case-sensitive!
- The default DATE format is: DD-MON-YY [01-JAN-95].
- Oracle stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds.
- Number values should not be enclosed in quotation marks as string literals should.

```
SQL> r
  1  select ename, job, deptno
  2  from emp
  3  where job='SALESMAN' AND
  4*      deptno=30
```

ENAME	JOB	DEPTNO
ALLEN	SALESMAN	30
WARD	SALESMAN	30
TURNER	SALESMAN	30

- Note the use of the AND operator to further restrict the selection criteria.
- Other comparison operators:
  - BETWEEN ...AND... Between two values (inclusive)
  - IN(list) Match any of a list of values
  - LIKE Match a character pattern
  - IS NULL Is a null value.
- The following example shows how to use the BETWEEN operator...

```
SQL> run
  1  select ename, job, deptno
  2  from emp
  3* where deptno between 30 AND 50
```

ENAME	JOB	DEPTNO
ALLEN	SALESMAN	30
WARD	SALESMAN	30
BLAKE	MANAGER	30
TURNER	SALESMAN	30
MOLITOR	ANALYST	40
SHANK	ANALYST	40

6 rows selected.

- In this case, we use two constants, 30 and 50, but we could have used a reference to a value stored in the table. It is also very important to note the order of the 30 and 50. The left of the 'AND' keyword represents the lower limit (30) while the number at the right of the 'AND' represents the upper limit. For this reason, a statement like this: ...WHERE DEPTNO BETWEEN 50 AND 30; would find no rows since there would be no intersection between upper and lower value.
- The following is an example on how to use the IN operator...

```
SQL> run
 1 select ename, job, deptno
 2 from emp
 3* where job IN ('SALESMAN', 'MANAGER', 'victor3434')
```

ENAME	JOB	DEPTNO
ALLEN	SALESMAN	30
WARD	SALESMAN	30
JONES	MANAGER	20
BOBBY	SALESMAN	
BLAKE	MANAGER	30
CLARK	MANAGER	10
TURNER	SALESMAN	30
BLACK	SALESMAN	10

- Note that as long as one of the fields is found, the whole row is selected.
- Now look at the following example using the LIKE operator...

```
SQL> select ename
 2 from emp
 3 where ename LIKE 'S%';
```

ENAME
SMITH
SCOTT
SIMMSON
SHANK

- Note that the % symbol is a 'Wildcard' character...in other words, in the example above, all those values whose ename start with an 'S' followed by zero or more characters, regard less of what they are. Note also that the pattern to be matched IS case-sensitive meaning that those of ename that start with a lower-case 's' will not be matched!

```
SQL> r
 1 select empno, ename
 2 from emp
 3* where ename LIKE 'S_'
```

EMPNO	ENAME
-------	-------

```
-----
3434 SI
```

- In the example above, we use “\_” to match a single character (as opposed as any number of them with ‘%’). SI is the only ename that matches this description, despite that there are other values of ename that start with a capital ‘S’.
- Further, these two operators ( ‘\_’, and ‘%’) can be combined in a single statement...

```
SQL> select ename
2 from emp
3 where ename like ‘_A%’;
```

```
ENAME
-----
WARD
MARKELL
HAYES
HAYES2
```

- Which returns all those names whose second letter is a capital ‘A’.
- Finally, in the event that we are looking specifically for patterns that contain the symbols ‘\_’ or ‘%’, we can use the ESCAPE identifier which allows us to specify any character as an ESCAPE character in order to match those special symbols.

**SEE BELOW...**

```
SQL> r
1 select empno, ename, job
2 from emp
3* where ename like ‘%*_%’ ESCAPE ‘*’
```

```
EMPNO ENAME      JOB
-----
3535 VIC_SANC
```

- Here, we designate ‘\*’ to be our escape character. Note that the escape character must be of type character string and must have a length of 1.
- The following example shows how to use the NULL operator...

```
SQL> select empno, ename, job
2 from emp
3 where job IS NULL;
```

```
EMPNO ENAME      JOB
-----
3434 SI
3535 VIC_SANC
```

- In this case, we select those items having a NULL value in the JOB field. Recall that some of these columns must contain a value and NULL values are not allowed...such is the case with rows representing primary key (and possibly foreign key values). Testing for NULL cannot be done with the ‘=’ operator!
- Logical operators AND, OR, and NOT can be applied to further restrict search criteria. Here’s an example using the OR operator...(all of these can be used in combination).

```
SQL> r
1 select empno, ename, job
2 from emp
3 where empno=3434 or
4* job is null
```

EMPNO	ENAME	JOB
3434	SI	
3535	VIC_SANC	

- Below is an example using the NOT operator...

```
SQL> r
1 select empno, ename, job
2 from emp
3 where not empno=3434 or
4* job is null
```

EMPNO	ENAME	JOB
7369	SMITH	ANALYST
7499	ALLEN	SALESMAN
7521	WARD	SALESMAN
7566	JONES	MANAGER
321	BOBBY	SALESMAN

...

- Note that list goes on and on...but I truncated it here...
- Here's a few other examples of proper syntax using the NOT operator...

- o ...WHERE job NOT IN('CLERK', 'ANALYST')
- o ...WHERE sal NOT BETWEEN 1000 AND 1500
- o ...WHERE ename NOT LIKE '%A%'
- o ...WHERE IS NOT NULL

- There are rules of precedence when using logical operators in conjunction. In order of most to least importance, these operators are evaluated as follows: All comparison operators-->NOT-->AND-->OR.
- However, it is always a good idea to use parentheses to force the evaluation of certain pairs of items and it is also easier to read.
- ORDER BY clause can be used to sort the rows. If you use the ORDER BY clause, it must be used at the end of all SQL statements. You can specify an expression or an alias to sort. Here's the syntax:

```
SELECT          expression
FROM            table
[WHERE          condition(s)]
[ORDER BY      {column, expression}[ASC|DESC]];
```

- where ORDER BY specifies the order in which the retrieved rows are displayed. ASC orders the rows in ascending order (this is the default order) and DESC (orders rows in descending order).
- The following is an example of using the ORDER BY clause...

```
SQL> r
1 select empno, ename, job
2 from emp
3 where empno > 3000 and
4     empno < 8000
5* order by empno desc
```

EMPNO	ENAME	JOB
-------	-------	-----

```

7876 ADAMS      ANALYST
7844 TURNER    SALESMAN
7839 KING      PRESIDENT

7096 BLACK     SALESMAN
7044 MOLITOR   ANALYST
3535 VIC_SANC
3434 SI

```

7 rows selected.

- Notice that we use the 'DESC' keyword in order to list the results in descending order according to the empno field.
- Note that we can also sort using a user-defined column alias as follows:

```

SQL> select empno, ename, sal*12 annsal
2 from emp
3 order by annsal;

```

EMPNO	ENAME	ANNSAL
8555	HAYES	1188
7499	ALLEN	12000
7521	WARD	12000
7844	TURNER	12000
...		

- Here, we have annsal as the column alias and it is ordering in ascending order, since it is the default manner in which ORDER BY orders rows.
- Further, it is possible to generate a list of values that are sorted using a column that is not included in the select list (i.e., a column that exists on the table, but is not displayed in the results).
- Finally, it is possible to sort based on multiple columns...

```

SQL> run
1 select ename, deptno, sal
2 from emp
3* order by deptno, sal DESC

```

ENAME	DEPTNO	SAL
MARKELL	2	1500
H	3	1500
KING	10	3000
CLARK	10	2000
BLACK	10	1000
JONES	20	2000
SCOTT	20	2000
SMITH	20	1500
...		

## SINGLE ROW FUNCTIONS -- CHAPTER 3 --

- There are two main types of functions, single-row and multi-row functions.
- MULTIPLE ROW functions manipulate groups of rows to give one result per group of rows.
- SINGLE ROW functions operate on single rows only and return one result per row. There are different types of these functions and they include those that process Characters, Number, Data, and Conversion.
  - Can manipulate data
  - Accept arguments and return one value
  - Act on each row returned
  - Return one result per row
  - Can Modify the data-type (type casting).
  - Can be nested.
- The syntax for single-row functions is as follows:

```
Function_name      (column | expression, [arg1, arg2,...])
```

- Where column is any database named column, expression is any character string or calculated expression and arg1, arg2, etc is any argument to be used by the function.

- **Character Functions:**

- (1) Case conversion functions and
  - LOWER – Converts alpha characters to lower case.
    - LOWER(column | expression)
  - UPPER – Converts alpha characters to upper case.
    - UPPER(column | expression)
  - INITCAP – Capitalizes the first alpha character of a single word.
    - INITCAP(column | expression)

```
SQL> run
1 select empno, initcap(ename), lower(ename), upper(ename)
2* from emp
```

EMPNO	INITCAP (EN	LOWER (ENAM	UPPER (ENAM
7369	Smith	smith	SMITH
7499	Allen	allen	ALLEN
7521	Ward	ward	WARD
7566	Jones	jones	JONES
321	Bobby	bobby	BOBBY

- (2) Character manipulation functions.
  - CONCAT – Concatenates the first character value to the second character value. Note that this is equivalent to using the concatenation operator (||).
    - CONCAT(column1 | expression1, column2 | expression2,...)

```
SQL> r
1 select empno, concat('Hello ', concat('Employee: ', ename))
2* from emp
```

EMPNO	CONCAT ('HELLO ', CONCAT ('EMP
7369	Hello Employee: SMITH
7499	Hello Employee: ALLEN
7521	Hello Employee: WARD

7566 Hello Employee: JONES

- **SUBSTR** – Returns specified characters from character value starting at character position m, n characters long (if m is negative, the count starts from the end of the character value. If n is omitted, all characters to the end of the of the string are returned.

- SUBSTR(column | expression, m[n])

- 

SQL> r

```
1 select ename, substr(ename, 0,2), substr(ename, 3)
2* from emp
```

ENAME	SU	SUBSTR(E
SMITH	SM	ITH
ALLEN	AL	LEN
WARD	WA	RD
JONES	JO	NES

- **LENGTH** – Returns the number of characters in value

- LENGTH(column | expression)

SQL> run

```
1 select ename, length(ename)
2* from emp
```

ENAME	LENGTH(ENAME)
SMITH	5
ALLEN	5
WARD	4

- **INSTR** – Returns the numeric position of a named character

- INSTR(column | expression, m)

ENAME	INSTR(ENAME, 'M')
-------	-------------------

SMITH	2
ALLEN	0
ADAMS	4
SIMMSON	3

- Note that when the character does not exist, this function returns a zero. Note also that in the event that the character exists more than once only the position of the first one (left to right) is returned.

- **LPAD** – Pads the character value right-justified to a total width of n character positions. Note that there IS also an **RPAD** function that works similarly but pads from the right...

- PAD(column | expression n, 'string')

SQL> run

```
1 select ename, lpad(ename, 20, '# ')
2* from emp
```

ENAME	LPAD(ENAME, 20, '#')
SMITH	# # # # # # # #SMITH
ALLEN	# # # # # # # #ALLEN
WARD	# # # # # # # # WARD
JONES	# # # # # # # #JONES

BOBBY # # # # # # # #BOBBY

- o **TRIM** – Enables to trim heading or trailing characters (or both) from a character string. If trim\_chracter or trim\_source is a character literal, you must enclose it in single quotes. This is a feature available from Oracle8i onward.

- TRIM(leading | trailing | both, trim\_character FROM trim\_source)

```
SQL> r
1 select trim('R' from 'RRRKR MITH')
2 from emp
3* where empno > 9000
```

```
TRIM('
-----
KRMITH
KRMITH
KRMITH
```

- Note that this function is acting on a string literal in this case (but it could be a column containing a string) and removes all occurrences of 'R' that are contiguous. Note that the right-most 'R' in the string is protected by the 'K' that precedes it.

- **Number Functions:**

- o **ROUND** – Rounds a value to a specified decimal
  - ROUND(45.926, 2) → 45.93
- o **TRUNC** – Truncates a value to specified decimal
  - TRUNC(45.926, 2) → 45.92
- o **MOD** – Returns remainder of division
  - MOD(1600, 300) → 100

```
SQL> r
1 select trunc(45.845, 1), round(45.845), round(45.845,2),
2 mod(45, 6)
3* from emp
3* where empno > 9000
```

```
TRUNC(45.845,1) ROUND(45.845) ROUND(45.845,2) MOD(45.5,2.3)
-----
45.8 46 45.85 1.8
```

- Note that the MOD function is not restricted to integer values. Any floating point values are allowed as arguments.
- **Working With DATES:**
- Oracle stores date in an internal numeric format: century, year, month, day, hours, minutes, seconds. The default date format is DD-MON-YY (example: 22-02-75 for February 22, 1975?)
- The function **SYSDATE** returns the current time/date.
- **DUAL** is a dummy table used to view SYSDATE. ( this will come in handy a lot!). It contains only one column and one row and is useful for returning single values of functions.

```
SQL> select SYSDATE
2 from DUAL;
```

```
SYSDATE
-----
29-AUG-02
```

- Arithmetic with DATES:
  - o OPERATION RESULT DESCRIPTION

- o Date + number    Date                    Add a number of days to a date
- o Date - number        Date                    Subtracts a number of days to a date
- o Date - Date            # of Days                Subtracts one date form another
- o Date + number/24     Date                    Adds a number of hours to a date

```
SQL> r
 1 select ename, round((SYSDATE - hiredate)/7) WEEKS
 2 from emp
 3* where deptno = 10
```

ENAME	WEEKS
CLARK	1107
KING	1084
BLACK	46

- Date Definitions: Date functions operate on Oracle date. All date function s return a value of DATE type with the exception of MONTHS\_BETWEEN which returns a numeric value.
  - o MONTHS\_BETWEEN (date1, date2) : Finds the number of months between date1 and date2. Note that the result can be either negative or positive depending on the argument's values.
  - o ADD\_MONTHS (date, n) : Adds a number of calendar months to date. The number of n must be an integer value and as such, may be either negative or positive.
  - o NEXT\_DAY (date, 'char') : Finds the date of the next specified day of the week ('char') following date. The number of char may be a number representing a day or character string.
  - o LAST\_DAY (date): Finds the date of the last day of the month that contains date.

```
SQL> r
 1 select months_between ('22-feb-1975', '28-aug-2002') *-1
 2      "NumberOfMontsIHaveLivedSoFar",
 3      round (months_between ('22-feb-1975', '28-aug-
 4      2002'))/12*-1
 5      "YearsIHaveLivedSoFar",
 6      next_day (SYSDATE, 'FRIDAY')
 7      "Tomorrow:"
 7* from dual
```

NumberOfMontsIHaveLivedSoFar	YearsIHaveLivedSoFar	Tomorrow:
330.19355	27.5	30-AUG-02

- Note in the example above, that the order in which the dates appear is important...in this case , we had to multiply the result by -1 in order to get positive results.
- o ROUND (date [, 'fmt' ]) : Returns date rounded to the unit specified by the format model fmt. If the format model fmt is omitted, date is rounded to the nearest day.
- o TRUNC (date [, 'fmt' ]) : Returns date with the time portion of the day truncated to the unit specified by the format model fmt. If the format model fmt is omitted, date is truncated to the nearest day.

```
SQL> r
 1 SELECT empno, hiredate,
 2 ROUND (hiredate, 'MONTH'), TRUNC (hiredate, 'MONTH')
 3 FROM emp
 4* WHERE hiredate LIKE '%81%'
```

EMPNO	HIREDATE	ROUND (HIR	TRUNC (HIR
7499	20-FEB-81	01-MAR-81	01-FEB-81
7521	22-FEB-81	01-MAR-81	01-FEB-81

```

7566 02-APR-81 01-APR-81 01-APR-81
7698 01-MAY-81 01-MAY-81 01-MAY-81
7782 09-JUN-81 01-JUN-81 01-JUN-81
7839 17-NOV-81 01-DEC-81 01-NOV-81

```

- **Conversion Functions:**

- In addition to Oracle datatypes, columns of tables in an Oracle8 database can be defined using ANSI dB2, and SQL/DS datatypes. However, the Oracle Server internally converts such datatypes to Oracle8 datatypes.
- In some cases, Oracle Server allows data of one datatype where it expects data of a different datatype. This is allowed when Oracle Server can automatically convert the data to the expected datatype. This datatype conversion can be done implicitly by Oracle Server or explicitly by the user.
- **Implicit** datatype conversions work according to the rules explained in the following sections.
- **Explicit** datatype conversions are done by using the conversion functions. The first datatype is the input datatype; the last datatype is the output datatype. It is recommended to always explicitly convert datatypes for clarity.
  - TO\_CHAR(number | Date, [fmt], [nlsparams]): Converts a number or data value to a VARCHAR2 character string with format model fmt. For number conversions, the nlsparams parameter specifies the following characters, which are returned by number format elements:
    - Decimal Character
    - Group Separator
    - Local Currency Symbol
    - International Currency Symbol
  - TO\_NUMBER(char, [fmt], [nlsparams]): Converts a character string containing digits to a number in the format specified by the optional format model 'fmt'.
  - TO\_DATE(char, [fmt], [nlsparams]): Converts a character string representing a date to a date value according to the fmt specified. If fmt is omitted, the default format is used.
- NOTE: There is a [list of time/date formats in 3-30 and 3-31](#) of Study Guide.

```

SQL> r
1 select ename, TO_CHAR(hiredate, 'fmDD Month YYYY') "Hire Date"
2* from emp

```

ENAME	Hire Date
SMITH	17 December 1980
ALLEN	20 February 1981

Another example:

```

SQL> r
1 select ename,
2 TO_CHAR(hiredate, 'fmDdspth "of" MONTH YYYY fmHH:MI:SS AM')
3* from emp

```

ENAME	TO_CHAR(HIREDATE, 'FMDDSPTH"OF"MONTHYYYYFMHH:
SMITH	Seventeenth of DECEMBER 1980 12:00:00 AM
ALLEN	Twentieth of FEBRUARY 1981 12:00:00 AM

- Note that the 'fm' that precedes Ddspth is used to remove padded blanks or suppress leading zeroes. Also note that 'Dd' in Ddspth Cause the Day to be displayed with the first character capitalized. Further, the 'sp' means 'spelled out' and 'th' means 'display ordinal'.
- Using TO\_CHAR() with numbers...(see 3-33 of Study Guide for formatting syntax).

```

SQL> r
1 select TO_CHAR(sal, '$99,999')
2 from emp
3* where sal between 1000 and 3000

```

```

TO_CHAR (
-----
$1,500
$1,000
$1,000
$2,000

```

- Note that Oracle Server will display (#'s) if the value in a field is greater than that allowed by the formatting model.
- NVL Function converts null to an actual value. Datatypes that can be used as arguments are Date, Character, and Number. The syntax is: NVL(expr1, expr2) where expr1 is the source value or expression that may contain a null value, and expr2 is the target value for converting null into.

```

SQL> r
1 select empno, ename, job, NVL(job, 'No Job Yet')
2* from emp

```

EMPNO	ENAME	JOB	NVL(JOB, 'No Job Yet')
7566	JONES	MANAGER	MANAGER
9600	BUSH	ANALYST	
3434	SI		No Job Yet
3535	VIC_SANC		No Job Yet

```

SQL> r
1 select empno, sal, sal*100, NVL(sal, 2), NVL(sal, 2)*100
2 from emp
3* where empno = 3434 or empno = 3535

```

EMPNO	SAL	SAL*100	NVL(SAL, 2)	NVL(SAL, 2)*100
3535			2	200
3434			2	200

- Note here that by using NVL, we transform NULL into a number 1, which is then multiplied by 100 resulting in a result of 200.

- DECODE Function: Facilitates conditional inquiries by doing the work of a CASE or IF-THEN-ELSE statement...here's the syntax:

```

DECODE (col/expression, search1, result1,
        [, search2, result2, ..., ], [, default])

```

- This function decodes *expression* after comparing it to each search value. If the *expression* is the same as *search*, result is returned.

```

SQL> SELECT job, sal,
2 DECODE(job, 'ANALYST', SAL*2.0,
3         'SALESMAN', SAL*3.0,
4         'MANAGER', SAL*5.0,
5         SAL)
6 REVISED_SALARY
7* FROM emp

```

JOB	SAL	REVISED_SALARY
ANALYST	1500	3000

SALESMAN	1000	3000
MANAGER	2000	10000

- Notice how REVISSED\_SALARY is incremented differently according to title as prescribed by the DECODE function. In this function, the 'job' inside the DECODE parentheses is the argument whose values you are to decode (i.e., 'ANALYST', 'SALESMAN', etc...), then the value that SAL\*2.0 evaluates to is used as the resulting value placed under column 'REVISSED\_SALARY' which is itself an alias. The last 'SAL' inside the DECODE parentheses is the default value in the event that a value of 'job' is not matched...for example, if there is a title not included in the list of values to match.
- Functions can be nested and I know how to do that.

**DISPLAYING DATA FROM MULTIPLE TABLES**  
**-- CHAPTER 4 --**

- Obtaining data from multiple tables: Sometimes you need to use data from more than one table. In the slide example, the report displays data from two separate tables.
- **Join:** When data from more than one table in a database is required, a join condition is used. Rows in one table can be joined to rows in another table according to common values existing in corresponding columns, that is, usually primary and foreign key columns. The syntax used within the WHERE clause to join information from two tables is as follow...

```
WHERE...
...Table1.column...
...table1.column1 = table2.column2;
```

- In the statements above, Table1.column denotes the column from which the information is to be extracted, and the statement table1.column1 = table2.column2 is the condition that must be true before an element from table1.column is selected.
- Cartesian Product: The combination of all rows between two table. It is formed when...
  - A join condition is omitted
  - A join condition is invalid
  - All rows in the first table are joined to all rows in the second table
- Types of joins: Equijoin, Non-equijoin, Outer join, Self join, Set Operators.
  - **Equijoin:** values in the two tables are equal...(i.e., the use of a primary key and a foreign key).
    - Qualifying column names with table name scan be very time consuming, particularly if table names are lengthy. You can use table aliases instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name.

```
SQL> r
1 select e.empno, e.ename, e.deptno, d.deptno, d.loc
2 from emp e, dept d
3* where e.deptno = d.deptno
```

EMPNO	ENAME	DEPTNO	DEPTNO	LOC
7369	SMITH	20	20	DALLAS
7499	ALLEN	30	20	CHICAGO
7521	WARD	30	30	CHICAGO
...				

- Note that the table alias is declared in the second line in the “FROM” clause. While table aliases can be up to 30 characters long, the point is usually to make them as small as possible for efficiency.
  - **Non-Equijoins:** Table joins where there isn’t necessarily a shared column between tables. These tables are united by using an operator other than ‘=’ such as “>, <, BETWEEN” etc. Here’s a small example...

**SEE BELOW...**

```
SQL> r
1 select e.ename, e.sal, s.grade
2 from emp e, salgrade s
3* where e.sal between s.losal and s.hisal
```

ENAME	SAL	GRADE
ALLEN	1000	1
WARD	1000	1
TURNER	1000	1
BLACK	1000	1
...		

- Recall that when using the BETWEEN clause, the order of the arguments does matter.
  - Outer Joins:** Can be used to return records with no direct match between tables. The operator used for these joins is the plus operator (+). Here's the syntax used...

```

SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.colulmn(+) = table2.column;
Or
SELECT table1.column, table2.column
FROM table1, table2
WHERE table1.colulmn = table2.column(+);

```

- The plus operator is placed on the side of the join that is deficient of information. This operator has the effect of creating one or more null rows, to which one or more rows from the non-deficient table can be joined. Let's look at an example...

```

SQL> r
1 select e.deptno, e.ename, d.deptno, d.dname
2 from emp e, dept d
3* where e.mgr(+) = d.deptno

```

DEPTNO	ENAME	DEPTNO	DNAME
		10	ACCOUNTING
		20	RESEARCH
		30	SALES
		40	OPERATIONS
		50	DEVELOPMENT

- In the example above, none of the records meet the criteria, but since this is an outer join, the all of the records from e.mgr are included with their respective null columns. Note that DEPTNO is a column shared by both tables, and therefore, this column does appear here.
- Note that only one side in the WHERE clause may have the outer join operator, and this is the side that lacks the information. Note also that a condition involving the outer join, cannot use the IN operator or be linked to another condition using the OR operator.
  - Self Joins:** Sometimes you need to join a table to itself. To find the name of each employee's manager, you need to join the emp table to itself, or perform a self join.

```

SQL> r
1  select worker.empno, worker.ename, worker.mgr,
2  worker.ename || ' works for ' || manager.ename
3  from emp worker, emp manager
4* where worker.mgr = manager.empno

```

EMPNO	ENAME	MGR	WORKER.ENAME    'WORKSFOR'    MANAG
7499	ALLEN	7698	ALLEN works for BLAKE
7521	WARD	7698	WARD works for BLAKE
7566	JONES	7839	JONES works for KING
7698	BLAKE	7839	BLAKE works for KING
7782	CLARK	7839	CLARK works for KING
7788	SCOTT	7566	SCOTT works for JONES
7844	TURNER	7698	TURNER works for BLAKE
7876	ADAMS	7788	ADAMS works for SCOTT
7096	BLACK	7782	BLACK works for CLARK
9666	SHANK	9300	SHANK works for REDCORNER

- Note that not all of the managers are listed here. For example, SHANK at the bottom works for REDCORNER, but REDCORNER does not appear on the left...this is because REDCORNER does not have a manager listed, according to this report. Only those employees who have a manager appear here.

**AGGREGATING DATA USING GROUP FUNCTIONS**  
**-- CHAPTER 5 --**

- What are group functions? These are functions that operate on sets of rows to give one result per group.
  - **AVG** ([DISTINCT] | ALL] n):
    - Average value of n, ignoring null values.
  - **COUNT** ({\* | [DISTINCT] | ALL] expr):
    - Returns the number of rows where expr evaluates to something other than null.
  - **MAX** ([DISTINCT] | ALL] expr):
    - Returns the largest value ignoring all null values.
  - **MIN** ([DISTINCT] | ALL] expr):
    - Returns the smallest value ignoring all null values.
  - **STDDEV** ([DISTINCT] | ALL] x):
    - Standard deviation of n, ignoring null values.
  - **SUM** ([DISTINCT] | ALL | n):
    - Returns the sum of all values ignoring all null values.
  - **VARIANCE** ([DISTINCT] | ALL] x):
    - Returns the variance of n, ignoring all null values.

```
SQL> select avg(sal), max(sal), min(sal), sum(sal), variance(sal), stddev(sal), count(sal)
2 from emp;
```

AVG(SAL)	MAX(SAL)	MIN(SAL)	SUM(SAL)	VARIANCE(SAL)	STDDEV(SAL)	COUNT(SAL)
2032.3235	5500	99	69099	1123584.2	1059.9925	34

- The above example, shows an example of all these group functions.
- Note that the **DISTINCT** clause in all these above functions makes the function consider only non-duplicate values, while **ALL** makes it consider all values. Furthermore, the data-types for the arguments may be **CHAR**, **VARCHAR2**, **NUMBER**, or **DATE** where expr is listed.
- Also, all group functions with the exception of **COUNT(\*)** ignore null values. To substitute values in for the null values, use the **NVL** function. Recall that the **NVL** function forces group functions to include null values.

```
SQL> r
1 select avg(NVL(sal,0)), max(sal), min(sal), sum(sal), variance(sal), stddev(sal), count(*)
2* from emp
```

AVG(NVL(SAL,0))	MAX(SAL)	MIN(SAL)	SUM(SAL)	VARIANCE(SAL)	STDDEV(SAL)	COUNT(*)
1919.4167	5500	99	69099	1123584.2	1059.9925	36

- Compare the **AVG** values with the ones on top...in this case, those records having null values are treated as if their null values were '0's' and therefore, they are included in the calculation, bringing the overall average down. Note that '0' was arbitrary here...we could have substituted those null values with any other value that would have applied.
- **Creating Groups Of Data:** You can use the **GROUP BY** clause to divide the rows in a table into groups. You can then use the group functions to return summary information about each group.

```
SQL> r
1 select deptno, avg(sal), count(*)
2 from emp
3* group by deptno
```

DEPTNO	AVG(SAL)	COUNT(*)
2	1500	1
3	1500	1
10	2000	3

20	1750	4
30	1250	4
40	5250	2
	1978.8947	21

- When using the GROUP BY clause, make sure that all columns in the SELECT list that are not in the group functions are included in the GROUP BY clause.
- Grouping By More Than One Column: Sometimes there is a need to see results for groups.

```
SQL> select deptno, job, sum(sal)
2 from emp
3 group by deptno, job;
```

DEPTNO	JOB	SUM(SAL)
2	ANALYST	1500
3	ANALYST	1500
10	MANAGER	2000
10	PRESIDENT	3000
10	SALESMAN	1000
20	ANALYST	5000
20	MANAGER	2000
30	MANAGER	2000
30	SALESMAN	3000
40	ANALYST	10500
	ANALYST	34599
	SALESMAN	3000

- In the above example, the columns are organized first in order of magnitude based on the value of DEPTNO and secondly in alphabetical order according to the string in found in JOB.
- Excluding Group Results: In the same way that you use the WHERE clause to restrict the rows that you select, you used the HAVING clause to restrict groups. For example, to show the maximum salary of each department, but show only the departments that have a maximum salary of more than \$2900, you need to do the following.
  - Find the maximum salary for each department by grouping the department number
  - Restrict the groups to those departments with a maximum salary greater than \$2900.

```
SQL> r
1 select deptno, max(sal)
2 from emp
3 group by deptno
4* having max(sal) > 2900
```

DEPTNO	MAX(SAL)
10	3000
40	5500
	4000

- Nesting Group Functions: Group functions can be nested to a depth of two only! Here's an example...

```
SQL> select max(avg(sal))
2 from emp
3 group by deptno;
```

MAX(AVG(SAL))

5250

- The following shows an error that occurs when you try to nest at a depth greater than two...

```
SQL> r
  1  select count((max(avg(sal)))
  2  from emp
  3* group by deptno
select count((max(avg(sal)))
                *
ERROR at line 1:
ORA-00935: group function is nested too deeply
```

## SUBQUERIES -- CHAPTER 6 --

- A Subquery is a SELECT statement that is embedded in a clause of another SELECT statement. You can build powerful statements out of simple ones by using subqueries. They can be very useful when trying to select rows from a table with a condition that depends on the data in the table itself.
- You can place the subquery in a number of SQL clauses:
  - WHERE clause
  - HAVING clause
  - FROM clause
- Here's the syntax:

```
SELECT      select_list
FROM        table
WHERE expr operator
              (SELECT      select_list
                FROM        table);
```

- Note that comparison operators fall under one of two classes: single-row operators (>, =, >=, <, <>, <=) and multiple-row operators(IN, ANY, ALL).
- Note that a subquery must appear on the right of a comparison operator and must be enclosed within a pair of parentheses.
- There's different types of subqueries:
- Single-row queries return one row from the inner SELECT statement and uses a 'single-row' operator (mentioned earlier).
  - **Single-row** subquery --> CLERK

```
SQL> SELECT ENAME, JOB
2 FROM EMP
3 WHERE JOB =
4           (SELECT JOB
5            FROM EMP
6            WHERE EMPNO = 7369);
```

ENAME	JOB
SMITH	ANALYST
SCOTT	ANALYST
ADAMS	ANALYST
H	ANALYST
"HELLO"	ANALYST
FREY	ANALYST
...	

- Here's an example using the HAVING clause with a single-row subquery...

```
SQL> SELECT deptno, min(sal)
2 FROM emp
3 GROUP BY deptno
4 HAVING min(sal) >
5           (SELECT min(sal)
6            FROM emp
7            WHERE deptno = 20);
```

DEPTNO	MIN(SAL)
40	5000

- As shown here, and as mentioned earlier, you can have subqueries in the WHERE clause as well as in the HAVING clause.

- **Multiple-row** subquery --> CLERK  
MANAGER

- These queries return more than one row. For these, you must use the multiple-row comparison operators which are IN, ANY, ALL, as mentioned earlier.

```
SQL> SELECT ENAME, SAL, DEPTNO
2 FROM EMP
3 WHERE SAL IN (SELECT MIN(SAL)
4 FROM EMP
5 GROUP BY DEPTNO);
```

ENAME	SAL	DEPTNO
HAYES	99	50
ALLEN	1000	30
WARD	1000	30
TURNER	1000	30
HAYES2	1000	50
...		

- **Multiple-column** subquery --> CLERK 7900  
MANAGER 7698

- Using the ANY operator in Multiple-Row subqueries...The ANY operator (and its synonym SOME operator) compares a value to each value returned by a subquery. See the following example...

```
SQL> SELECT EMPNO, ENAME, JOB
2 FROM EMP
3 WHERE SAL < ANY
4 (SELECT SAL
5 FROM EMP
6 WHERE JOB = 'CLERK')
7 AND JOB <> 'CLERK';
```

no rows selected

- Although no rows were returned, the query is well formed.
- Finally let's look at how the ALL operator is used in Multiple-Row Subqueries...

```
SQL> SELECT EMPNO, ENAME, JOB
2 FROM EMP
3 WHERE SAL > ALL
4 (SELECT AVG(SAL)
5 FROM EMP
6 GROUP BY DEPTNO);
```

EMPNO	ENAME	JOB
9666	SHANK	ANALYST

- The ALL operator compares the value to every value returned by a subquery.
- NOTE that the NOT operator may be used in conjunction with any of IN, ANY, ALL operators.

## MULTIPLE-COLUMN SUBQUERIES -- CHAPTER 7 --

- Multiple-Column Subqueries: So we have written single-row subqueries where only one column was compared in the WHERE clause or HAVING clauses of the SELECT statement. If you want to compare two or more columns, you must write a compound WHERE clause using logical operators.
- Let's take a look at the syntax:

```
SELECT  column, column, ...
FROM    table
WHERE   (column, column, ...) IN
                                   (SELECT column, column, ...
                                   FROM    table
                                   WHERE   someCondition);
```

- Check out the following multiple-column query with subquery...

```
SQL> r
1  select empno, ename, job, sal, deptno
2  from emp
3  where (sal, deptno) in
4  (select sal, deptno
5  from emp
6  where sal between 1000 and 2000 and
7  deptno between 20 and 40)
8* order by empno
```

EMPNO	ENAME	JOB	SAL	DEPTNO
7369	SMITH	ANALYST	1500	20
7499	ALLEN	SALESMAN	1000	30
7521	WARD	SALESMAN	1000	30
7566	JONES	MANAGER	2000	20
7698	BLAKE	MANAGER	2000	30
7788	SCOTT	ANALYST	2000	20
7844	TURNER	SALESMAN	1000	30
7876	ADAMS	ANALYST	1500	20

- In this case, we selected empno, ename, job, sal, and deptno from the 'emp' table such that all those employees whose salary was between 1000 and 2000 and their deptno was between 20 and 40. A glance at the data allows us to verify that all records returned by the query satisfy those conditions. Note also that the records are ordered by the empno as dictated by the last line of the subquery.
- In the following example, we add a couple of more lines to the previous query to further restrict the search, and thereby eliminate the records for SMITH and JONES...

```
SQL> r
1  select empno, ename, job, sal, deptno
2  from emp
3  where (sal, deptno) in
4  (select sal, deptno
5  from emp
6  where sal between 1000 and 2000 and
7  deptno between 20 and 40)
8  and ename <> 'SMITH'
9  and ename <> 'JONES'
10* order by empno
```

EMPNO	ENAME	JOB	SAL	DEPTNO
7499	ALLEN	SALESMAN	1000	30
7521	WARD	SALESMAN	1000	30
7698	BLAKE	MANAGER	2000	30
7788	SCOTT	ANALYST	2000	20
7844	TURNER	SALESMAN	1000	30
7876	ADAMS	ANALYST	1500	20

6 rows selected.

- Column Comparisons: Pairwise Versus Nonpairwise Comparisons
- The above examples are called 'Pairwise Comparisons' since in order to meet the criteria, the records being compared had to meet two conditions simultaneously...i.e., their both their sal 'AND' their deptno had to be in a certain range. If you want a nonpairwise comparison (a cross product), you must use a WHERE clause with multiple conditions. A candidate row must match the multiple conditions in the WHERE clause but the values are compared individually.

```
SQL> r
1 select empno, ename, job, sal, deptno
2 from emp
3 where sal in
4 (select sal
5 from emp
6 where sal between 1000 and 2000)
7 and empno in
8 (select empno
9 from emp
10 where deptno between 20 and 40)
11* order by empno
```

EMPNO	ENAME	JOB	SAL	DEPTNO
7369	SMITH	ANALYST	1500	20
7499	ALLEN	SALESMAN	1000	30
7521	WARD	SALESMAN	1000	30
7566	JONES	MANAGER	2000	20
7698	BLAKE	MANAGER	2000	30
7788	SCOTT	ANALYST	2000	20
7844	TURNER	SALESMAN	1000	30
7876	ADAMS	ANALYST	1500	20

8 rows selected.

- Unfortunately, this is not the best example since it yielded the same results as the pairwise comparison, but theoretically, this is just a coincidence...other times the results may differ between these two types of comparisons.
- NULL values in a subquery. When SQL encounters a NULL value in a comparison of some sort, SQL will simply not return any rows as output. All conditions that compare a null value result in a null. So, whenever null values are likely to be part of the resultant set of a subquery, do not use the NOT IN operator. As an example, compare the following two queries...

```
SQL> select employee.ename
2 from emp employee
3 where employee.empno NOT IN
4 (select manager.mgr
5 from emp manager);
```

no rows selected

Compared to this one...

```
SQL> select employee.ename
2   from emp employee
3   where employee.empno IN
4   (select manager.mgr
5   from emp manager);
```

```
ENAME
-----
JONES
BLAKE
CLARK
SCOTT
KING
REDCORNER
```

6 rows selected.

- Notice that the null value as part of the resultant set of a subquery will not be a problem if you are using the IN operator.
- Finally, let's look at how we can use a subquery in the FROM clause...

```
SQL> r
1  select a.ename, a.sal, a.deptno, b.salavg
2  from emp a, (select deptno, avg(sal) salavg
3              from emp
4              group by deptno) b
5  where a.deptno = b.deptno
6* and a.sal > b.salavg
```

ENAME	SAL	DEPTNO	SALAVG
KING	3000	10	2000
SCOTT	2000	20	1750
JONES	2000	20	1750
BLAKE	2000	30	1250
SHANK	5500	40	5250
BOBBY	3000	50	1978.8947
"HELLO"	2000	50	1978.8947
BUSH	2000	50	1978.8947
HOMER	2000	50	1978.8947
RED	2000	50	1978.8947
REDCORNER	2000	50	1978.8947
ALBERT	4000	50	1978.8947
TESTTIME	2000	50	1978.8947
NKLGNFGLKB	2000	50	1978.8947
HOLLYO	2000	50	1978.8947
GREEN	2000	50	1978.8947
	2000	50	1978.8947
KWIATKI	2000	50	1978.8947
FREY	2000	50	1978.8947
GORDON	2000	50	1978.8947
DUTT	2000	50	1978.8947

21 rows selected.

- You can use a subquery in the **FROM** clause of a **SELECT** statement, which is very similar to how views are used. A subquery in the **FROM** clause of a **SELECT** statement defines a data source for that particular **SELECT** statement, and only that **SELECT** statement.

**PRODUCING READABLE OUTPUT WITH SQL\*PLUS**  
**-- CHAPTER 8 --**

- Interactive Reports: Using SQL\*Plus, you can create reports that prompt the user to supply their own values to restrict the range of data to be returned. To create interactive reports, you can embed *substitution variables* in a command file or in a single SQL statement. A variable can be thought of as a container in which the values are temporarily stored.
- In SQL\*Plus, you can use & (the ampersand) to temporarily store values. And you can use the DEFINE and ACCEPT commands to predefine variables. ACCEPT reads a line of user input and stores it in a variable.
- The following example shows the user of substitution variables...

```
SQL> select empno, ename, sal, deptno
2  from emp
3  where empno = &employee_num
4  /
Enter value for employee_num: 7369
old 3: where empno = &employee_num
new 3: where empno = 7369
```

EMPNO	ENAME	SAL	DEPTNO
7369	SMITH	1500	20

- Basically, we created a query in which we made a comparison to a variable, which SQL\*Plus then inquired about its value, and once it was provided, it was able to continue with the query. In the following example, we accomplish the same query using all variable names...

```
SQL> select &EmployeeNumber, &EmployeeName, &Salary, &DepartmentNumber
2  from &TableName
3  where &EmployeeNumber = &EmployeeNumber1
4  /
Enter value for employeenumber: empno
Enter value for employeename: ename
Enter value for salary: sal
Enter value for departmentnumber: deptno
old 1: select &EmployeeNumber, &EmployeeName, &Salary, &DepartmentNumber
new 1: select empno, ename, sal, deptno
Enter value for tablename: emp
old 2: from &TableName
new 2: from emp
Enter value for employeenumber: empno
Enter value for employeenumber1: 7369
old 3: where &EmployeeNumber = &EmployeeNumber1
new 3: where empno = 7369
```

EMPNO	ENAME	SAL	DEPTNO
7369	SMITH	1500	20

- Note that SQL\*Plus will inquire about the value of each one of the variable names before it will be able to return any meaningful results.
- Using the SET VERIFY command: Toggling the display of the text before and after SQL\*Plus replaces substitution variables with values. To change the display, simply issue the following command: set verify ON/OFF. When ON, it will show the old and new values of any variables used in the query.
- Character and Date Values with Substitution Variables.

```
SQL> select ename, deptno, sal*12
```

```

2  from emp
3  where job='&job_title'
4  /
Enter value for job_title: ANALYST

```

ENAME	DEPTNO	SAL*12
SMITH	20	18000
SCOTT	20	24000
ADAMS	20	18000
H	3	18000
"HELLO"	50	24000

- Note that in the above example, we need to either include single quotes around the variable name or if we want to omit them, we have to include them when SQL\*Plus prompts for the character value, so instead of entering ANALYST, we would enter 'ANALYST'.
- Note that in order for this query to find any records...ANALYST must be all in caps...in order to avoid this, you may enter [analyst] so as long as you use the UPPER() function as shown below...

```

SQL> select ename, deptno, sal*12
2  from emp
3  where job=upper('&job_title')
4  /
Enter value for job_title: analyst

```

ENAME	DEPTNO	SAL*12
SMITH	20	18000
SCOTT	20	24000
ADAMS	20	18000
H	3	18000
"HELLO"	50	24000

- LOWER() is also available for this purpose....
- Note that substitution variables can pretty much take the place of anything including WHERE, FROM, ORDER BY, Column expressions, table-Names, basically, the entire SELECT statement...To show this, take a look at the following example:

```

SQL> select &first, &second
2  &fromClause &tableName
3  &whereClause sal &theOperator 3000;
Enter value for first: ename
Enter value for second: sal
Enter value for fromclause: from
Enter value for tablename: emp
Enter value for whereclause: where
Enter value for theoperator: >

```

ENAME	SAL
MOLITOR	5000
SHANK	5500
ALBERT	4000

- Now, let's take a look at the && Substitution Variable: Use the double-ampersand (&&) if you want to reuse the variable value without prompting the user each time. The user will get a single prompt regardless of how many times the variable

name appears in the SELECT statement. Interesting to note is that once defined, the variable name retains its value across select statement sessions.

- Defining User Variables: You can predefine variables using one of two SQL\*Plus commands:
  - DEFINE: Create a CHAR datatype user variable
  - ACCEPT: Read user input and store it into a variable;
- If you need to predefine a variable that includes empty spaces, make sure you use quotation marks when using the DEFINE statement.
  - DEFINE variable = value
    - Creates a CHAR datatype user variable and assigns a value to it.
  - DEFINE variable
    - Displays the variable, its value, and its datatype
  - DEFINE
    - Displays ALL user variables with value and datatype
  - ACCEPT
    - Reads a line of user input and stores it in a variable
- Note that when defining variables, you do not need to include the & symbol. In fact, if you do, it does not work! Also, the string of characters you define as, does not need quotes, unless it needs to have empty spaces embedded in between.
- The ACCEPT command:
  - Creates a customized prompt when accessing user input
  - Explicitly defines a NUMBER or DATE datatype variable
  - Hides user input for security reasons
- **SYNTAX:**

```
ACCEPT variable [datatype] [FORMAT format] [PROMPT text] [HIDE]
```

- IN the syntax, variable is the name of the variable that stores the value (if it does not exist, SQL\*Plus will create one). Datatype is NUMBER, CHAR, or DATE(CHAR has a maximum length of 240 bytes. DATE checks against a format model, and the datatype is CHAR). FOR[MAT] format specifies the format model – for example, A10 or 9.999. PROMPT text displays the text before the user can enter the value. HIDE suppresses what the user enters – for example, a password.
- Here’s an example on how to use the ACCEPT command...

```
SQL> accept dept prompt 'Provide the department name: '  
Provide the department name: Sales  
SQL> select *  
2 from dept  
3 where dname = upper('&dept');  
old 3: where dname = upper('&dept')  
new 3: where dname = upper('Sales')
```

DEPTNO	DNAME	LOC
30	SALES	CHICAGO

- Now look at Customizing the SQL\*Plus Environment:
  - Use the SET commands to control the current session.
    - SET system\_variable value
  - Use the SHOW command to verify what you SET...

```
SQL> set echo on  
SQL> show echo  
echo ON
```

- To see all set variables, use the SHOW ALL command.

```
SQL> SHOW ALL  
appinfo is ON and set to "SQL*Plus"
```

```

arraysize 15
autocommit OFF
autoprint OFF
autotrace OFF
shiftinout INVISIBLE
blockterminator "." (hex 2e)
btitle OFF and is the 1st few characters of the next SELECT statement
cmdsep OFF
colsep " "
compatibility version NATIVE
concat "." (hex 2e)
copycommit 0
COPYTYPECHECK is ON
define "&" (hex 26)
echo ON
editfile "afiedt.buf"
embedded OFF
escape OFF
FEEDBACK ON for 6 or more rows
flagger OFF
flush ON
heading ON
headsep "|" (hex 7c)
linesize 100
lno 5
loboffset 1
long 80
longchunksize 80
newpage 1
null ""
numformat ""
numwidth 9
pagesize 24
PAUSE is OFF
pno 1
recsep WRAP
recsepchar " " (hex 20)
release 801050000
repfooter OFF and is NULL
repheader OFF and is NULL
serveroutput OFF
showmode OFF
spool OFF
sqlcase MIXED
sqlcode 904
sqlcontinue "> "
sqlnumber ON
sqlprefix "#" (hex 23)
sqlprompt "SQL> "
sqlterminator ";" (hex 3b)
suffix "sql"
tab ON
termout ON
time OFF
timing OFF
trimout ON
trimspool OFF
ttitle OFF and is the 1st few characters of the next SELECT statement
underline "-" (hex 2d)
USER is "SCOTT"
verify ON
wrap : lines will be wrapped

```

- **Set Command Variables:**

- ARRAYSIZE{20|N}
  - Sets database to data fetch size
- COLSEP{\_|text}
  - Sets text to be printed between columns (default is single space)
- FEEDBACK{6|n|OFF|ON}

- Displays the number of records returned by a query when the query selects at least n records.
  - HEADING{OFF|ON}
    - Determines whether column headings are displayed in reports or not.
  - LINESIZE{80|n}
    - Sets the number of characters to n for reports
  - LONG{80|n}
    - Sets the maximum length for displaying LONG values.
  - PAGESIZE{24|n}
    - Determines the number of lines per page of output.
  - PAUSE{OFF|ON|text}
    - Allows you to control scrolling of your terminal (You must press [RETURN] after seeing each pause.
  - TERMOUT{OFF|ON}
    - Determines whether output is displayed on screen.
- **IMPORTANT: SAVING CUSTOMIZATIONS in the `login.sql` File:**
- The `login.sql` file contains standard SET and other SQL\*Plus commands that are implemented at login
- You can modify the `login.sql` to contain additional SET commands.
- The `login.sql` file is located at `C:\orant\DBS\login.sql` in this computer. If this is not the location in your system, just have the operating system look for it for you...do a search!
- SQL\*Plus Format Commands and their corresponding Options...
  - **COLUMN** [column option]
    - Controls column formats.
      - CLEAR: Clears any column formats
      - FOR[MAT] format: Changes the display of the column data
      - HEA[DING]: text: Sets the column heading (a vertical line (|) will force a line feed in the heading if you do not use justification).
      - JUS[TIFY]{ALIGN}: Aligns the column heading to be left, center, or right.
      - NOPRI[NT]: Hides the column.
      - NUL[L]: Specifies text to be displayed for null values.
      - PRI[NT]: Shows the column.
      - TRU[NCATED]: Truncates the string at the end of the first line of display.
      - WRA[PPED]: Wraps the end of the string to the next line.
  - **TTITLE** [text|OFF|ON]
    - Specifies a header to appear at the top of each page.
  - **BTITLE**[text|OFF|ON]
    - Specifies a footer to appear at the bottom of each page of the report
  - **BREAK**[ON report\_element]
    - Suppress duplicate values and sections rows of data with line feeds.
- Note that all formatting settings remain in effect until the end of the setting.
- Using the TTITLE and BTITLE commands: Display headers and footers...
  - TTI[TLE] [text|OFF|ON]
  - TTITLE 'Salary|Report' -- Headers
  - BTITLE 'Confidential' -- Footers
- The following example demonstrates the use of setting pagesize, tttitle, btitle, and break.

```
SQL> set pagesize 25
SQL> tttitle 'Salary Report|September, 2002'
SQL> btitle 'Salary Report'
SQL> break on job skip 1
SQL> r
 1 select empno, ename, job, mgr, sal, deptno
 2 from emp
 3 where sal between &Low and &High
 4* order by job
Enter value for low: 1000
```

Enter value for high: 2000  
old 3: where sal between &Low and &High  
new 3: where sal between 1000 and 2000

Fri Sep 06 page 1

**Salary Report  
September, 2002**

EMPNO	ENAME	JOB	MGR	SAL	DEPTNO
7788	SCOTT	ANALYST	7566	2000	20
8001	"HELLO"			2000	50
8060	FREY			2000	50
9200	KWIATKI			2000	50
9100	DUTT			2000	50
8090	GORDON			2000	50
9600	BUSH			2000	50
1010	HOMER			2000	50
9230	GREEN			2000	50
9712	TESTTIME			2000	50
9310	NKLGNFGLKB			2000	50
9326	HOLLYO			2000	50
9300	REDCORNER			2000	50
9120	RED			2000	50
1000				2000	50
7566	JONES	MANAGER	7839	2000	20

**Salary Report**

Fri Sep 06 page 2

**Salary Report  
September, 2002**

EMPNO	ENAME	JOB	MGR	SAL	DEPTNO
7698	BLAKE	MANAGER	7839	2000	30
7782	CLARK		7839	2000	10
7839	KING	PRESIDENT		3000	10
321	BOBBY	SALESMAN		3000	50

**Salary Report**

20 rows selected.

- Notice that each page is 25 line long and that there are line breaks (of 1 in this case) between records that have a job-transition (i.e., from MANAGER to PRESIDENT).
- Creating a Script File to Run a Report
  - Create the SQL SELECT statement
  - Save the SELECT statement to a script file
  - Load the script file into the editor
  - Add formatting commands before the SELECT statements
  - Verify that the termination character follows the SELECT statement.
- How to create a Script File
  1. Create the SQL SELECT statements at the SQL prompt. Ensure that the data required for the report is accurate before you save the statements to a file and apply formatting commands. Endure that the relevant ORDER BY clause is included if you intend to use breaks.

2. Save the SELECT statement to a script file
3. Edit the script file to enter the SQL\*Plus commands
4. Add the required formatting commands before the SELECT statement. Be certain not to place SQL\*Plus commands within the SELECT statement.
5. Verify that the SELECT statement is followed by a run character, either a semicolon(;) or a slash (/).
6. Clear formatting commands after the SELECT statement.
7. Save the script file.
8. Enter "START filename" to run the script.

**sampleReport.sql**

```

SET PAGESIZE 37
SET LINESIZE60
SET FEEDBACK OFF
TTITLE 'Employee Report'
BTITLE 'Confidential'
BREAK ON job
COLUMN job HEADING 'Job|Category' FORMAT A15
COLUMN ename HEADING 'Employee' FORMAT A15
COLUMN sal HEADING 'Salary', FORMAT $99.999.99
REM ** Insert SELECT statement
SELECT job, ename, sal
FROM emp
WHERE sal < 3000
ORDER BY job, ename
/
SET FEEDBACK ON
REM clear all formatting commands..
/

```

\* The following is a sample report generated from the script above...

Fri Sep 06

page 1

Employee Report

Job Category	Employee	SAL
ANALYST	"HELLO"	2000
	ADAMS	1500
	BUSH	2000
	DUTT	2000
	FREY	2000
	GORDON	2000
	GREEN	2000
	H	1500

...

## MANIPULATING DATA -- CHAPTER 9 --

- A Data Manipulation Language (DML) statement is executed when you
  - Add new rows to a table
  - Modify existing rows in a table
  - Remove existing rows from a table
- A transaction consists of a collection of DML statements that form a logical unit of work.
- Adding a new row to a table is accomplished using the INSERT statement

```
INSERT INTO table [column, column, column]
VALUES (value, value, value);
```

- Because you can insert a new row that contains values for each column, the column list is not required in the INSERT clause. However, if you do not use the column list, the values must be listed according to the default order of the columns in the table.
  - You can insert NULL values by simply omitting the column value, or by specifying either ('') or NULL as the item to be inserted.
- Here's an example of an insert...

```
SQL> insert into emp
      2 values (2296, 'AROMANO', 'SALESMAN', 7782,
      3 TO_DATE('FEB 3, 1997', 'MON DD, YYYY'),
      4 1300, NULL, 10);

1 row created.
```

- Note that the TO\_DATE() function formats the string into a DATE datatype.
- Creating a Script with Customized Prompts
  - ACCEPT stores the value in the variable
  - PROMPT displays your customized text.
- Let's look at the following example...first, we create the following script and save it with the following name scriptsWithCustomizedPrompts.sql:

```
ACCEPT department_id PROMPT 'Please enter the -
department number:'
ACCEPT department_name PROMPT 'Please enter -
the department name:'
ACCEPT location PROMPT 'Please enter the -
location:'
INSERT INTO dept (deptno, dname, loc)
VALUES (&department_id, '&department_name', '&location');
```

- We then run the script using the START keyword...

```
SQL> START scriptsWithCustomizedPrompts
Please enter the          department number:90
Please enter          the department name:PAYROLL
Please enter the          location:HOUSTON
old  2: VALUES (&department_id, '&department_name', '&location')
new  2: VALUES (90, 'PAYROLL', 'HOUSTON')

1 row created.
```

- Notice that the ACCEPT keyword allows us to accept the value entered by the user and to store it in the variable name that follows it, which, by the way, does not require the substitution parameter (&). However, when we do make use of its contents in the INSERT statement, we must include the ampersand! When the script is run, the user is asked to provide the values for all three variables here defined: department\_id, department\_name, and location.
- Copying Rows from another Table: Write your INSERT statement with a subquery. Do not use the VALUES clause. Match the number of columns in the INSERT clause to those in the subquery.

```
SQL> create table managers(id number(4), name varchar2(10), salary
2  number(7,2), hiredate date)
3  /
```

Table created.

```
SQL> INSERT INTO managers(id, name, salary, hiredate)
2  SELECT empno, ename, sal, hiredate
3  FROM emp
4  WHERE job = 'MANAGER';
```

3 rows created.

```
SQL> select *
2  from managers
3  /
```

ID	NAME	SALARY	HIREDATE
7566	JONES	2000	02-APR-81
7698	BLAKE	2000	01-MAY-81
7782	CLARK	2000	09-JUN-81

- For changing date in a table, we make use of the UPDATE statement...

```
SQL> update managers
2  set id = 3434
3  where name='JONES';
```

1 row updated.

```
SQL> select *
2  from managers;
```

ID	NAME	SALARY	HIREDATE
3434	JONES	2000	02-APR-81
7698	BLAKE	2000	01-MAY-81
7782	CLARK	2000	09-JUN-81

- Notice that we make use of the keyword SET to modify the contents of a particular row. In this particular case, we updated only one row by specifically singling out the record for JONES using the WHERE clause. However, if left out, the UPDATE statement would have modified all rows. Note that if ID is a primary key, the update would fail since no two rows may have the same value. It is, however, possible to modify all ID values at once, so as long as the update statement allows for all of them to be different.
- Updating with Multiple-Column Subquery

```
SQL> UPDATE managers
2  SET (name, salary) =
```

```

3 (select ename, sal
4 from emp
5 where empno = 7698)
6 WHERE ID = 7782;

```

1 row updated.

```
SQL> select * from managers;
```

ID	NAME	SALARY	HIREDATE
3434	JONES	2000	02-APR-81
7698	BLAKE	2000	01-MAY-81
7782	BLAKE	2000	09-JUN-81

- Note here that the name CLARK (bottom) was changed to BLAKE and the salary seems to have remained unchanged since 2000 was replaced by 2000...however, the value was technically modified. Note that the information used to modify the managers table came from table emp.
- Removing a row from a table: Use the DELETE statement as follows:

```
SQL> delete from managers
2 where id = 7782;
```

1 row deleted.

```
SQL> select * from managers;
```

ID	NAME	SALARY	HIREDATE
3434	JONES	2000	02-APR-81
7698	BLAKE	2000	01-MAY-81

- Again, it is possible to delete rows based on the contents of a second table by using subqueries. This is likely a useful procedure.
- Database Transactions: Consist of one of the following statements:
  - DML statements that make up one consistent change to the data
  - One DDL statement
  - One DCL statement
- Transactions consist of DML statements that make up one consistent change to the date. For example, a transfer of funds between two accounts should include the debit to one account and the credit to another account in the same amount. Both actions should either fail or succeed together. The credit should not be committed without the debit.
- Database Transactions: Begin when the first executable SQL statement is executed and end with one of the following events: Commit or Rollback is issued.
- Controlling Transactions: You can control the logic of transaction by using the COMMIT, SAVEPOINT, and ROLLBACK statements.
  - COMMIT - Ends the current transaction by making all pending data changes permanent.
  - SAVEPOINT name - Marks a savepoint within the current transaction
  - ROLLBACK [TO SAVEPOINT name] - ROLLBACK ends the current transaction by discarding all pending data changes; ROLLBACK TO SAVEPOINT rolls back the current transaction to the specified savepoint, thereby discarding the savepoint and any subsequent changes. If you omit this clause, the ROLLBACK statement rolls back to the entire transaction.
- Implicit Transaction Processing: An automatic commit occurs under the following circumstances:
  - DDL statement is issued
  - DCL statement is issued
  - Normal exit from SQL\*Plus, without explicitly issuing COMMIT or ROLLBACK
  - An automatic rollback occurs under an abnormal termination of SQL\* Plus or a system failure.

- **Committing Changes:** Every data change made during the transaction is temporary until the transaction is committed. Other users cannot view the results of the data manipulation operations made by the current user. The Oracle Server institutes read consistency to ensure that each user sees data as it existed at a the last commit.
- **State of the Data After COMMIT:**
  - Data changes are made permanent in the database
  - The previous state of the data is permanently lost
  - All users can view the results
  - Locks on the affected rows are released; those rows are available for other users to manipulate.
  - All savepoints are erased.
- **State of the Data After Rollback:** Discard all pending changes by using the ROLLBACK statement.
  - Data changes are undone
  - Previous state of the data is restored
  - Locks on the affected rows are released.
- **Statement-Level-Rollback**
  - If a single DML statement fails during execution, only the statement is rolled back
  - The Oracle Server implements an implicit savepoint
  - All other changes are retained.
  - The user should terminate transactions explicitly by executing a COMMIT or ROLLBACK statement.
- **Read Consistency:**
  - Read consistency guarantees a consistent view of the data at all times.
  - Changes made by one user do not conflict with changes made by another user.
  - Read consistency ensures that on the same data:
    - Readers do not wait for writers
    - Writers do not wait for readers
- **Implementation of Read Consistency:** Read consistency is an automatic implementation. It keeps a partial copy of the database in rollback segments. When insert, update, or delete operation is made to the database, the Oracle Server takes a copy of the data before it is changed and writes it to a rollback segment. All readers, except the one who issued the change, still see the database as it existed before the changes started; they view the rollback segments' "snapshot" of the data.
- **Locking: Oracle locks:**
  - Prevent destructive interaction between concurrent transactions
  - Require no user action
  - Automatically use the lowest level of restrictiveness.
  - Are held for the duration of the transaction
  - Have two basic modes:
    - Exclusive – Prevents a resource from being shared. The first transaction to lock a resource exclusively, in the only transaction that can alter the resource until the exclusive lock is released.
    - Share – Allows the resource to be shared. Multiple users reading data can share the data, holding share locks to prevent concurrent access by a writer (who needs an exclusive lock) . Several transactions can acquire share locks on the same resource.

## CREATING AND MANAGING TABLES

### -- CHAPTER 10 --

- Database Objects: An Oracle database can contain multiple data structures. Each structure should be outlined in the database design so that it can be created during the build stage of database development.
  - Table: Stores data
  - View: Subset of data from one or more tables
  - Sequence: Generates primary key values
  - Index: Improves the performance of some queries
  - Synonym: Gives alternative names of objects
- Oracle8 Table Structures
  - Tables can be created at any time, even while users are using the database
  - You do not need to specify the size of any table. The size is ultimately defined by the amount of space allocated to the database as a whole. It is important, however, to estimate how much space a table will use over time.
  - Table structure can be modified online.
- Naming Conventions:
  - Must begin with a letter and must be no longer than 30 characters.
  - Must contain only A-Z, a-z, 0-9, \_, \$, and #
  - Must no duplicate the name of another object owned by the same user
  - Must not be an Oracle Server reserved word.
    - Note that names are case insensitive.
- The CREATE TABLE statement: You must have CREATE TABLE privilege and Storage Area...
  - Must specify Table Name
    - Must specify column name, column datatype, and column size;
- Referencing Another User's Tables:
  - Tables belonging to other users are not in the user's schema
  - You should use the owner's name as a prefix to the table.
- The DEFAULT Option
  - Specify a default value for a column during an insert
    - ...Hiredate DATE DEFAULT SYSDATE,...
  - Legal values are literal value, expression, or SQL function.
  - Illegal values are another column's name or pseudocolumn.
  - The default datatype must match the column datatype.
- Creating Tables

```
SQL> r
1 CREATE TABLE dept
2 (deptno NUMBER(2),
3  dname VARCHAR2(14),
4*  loc VARCHAR2(13))
```

Table created.

```
SQL> describe dept
Name                                Null?    Type
-----
DEPTNO                               NUMBER(2)
DNAME                                VARCHAR2(14)
LOC                                   VARCHAR2(13)
```

- Tables in the Oracle Database
  - USER TABLES
    - Collection of tables created and maintained by the user
    - Contain user information
  - DATA DICTIONARY
    - Collection of tables created and maintained by the Oracle Server

- Contain database information
- All data dictionary tables are owned by the SYS user. The base tables are rarely accessed by the user because the information in them is not easy to understand. Therefore, users typically access data dictionary views because the information is presented in a format that is easier to understand. Information stored in the data dictionary include names of the Oracle Server users, privileges granted to users, database object names, table constraints, and auditing information.
- There are four categories of data dictionary views, each category has a distinct previews which reflects their intended use.
  - USER\_ - These views contain information about objects owned by the user
  - ALL\_ - These views contain information about all of the tables (object tables and relational tables) accessible to the user.
  - DBA\_ - These views are restricted views. These views can be accessed only by people who have been assigned the role DBA.
  - V\$\_ - These views contain information about dynamic performance views, database server performance and locking.

- Querying the Data Dictionary

- SELECT \* FROM user\_tables;

```
SQL> select table_name
      2  from user_tables;
```

```
TABLE_NAME
-----
CONTACT
CUSTOMER
DEPT
ZIP_TABLE
```

- SELECT DISTINCT object\_type FROM user\_objects;

```
SQL> select distinct object_type
      2  from user_objects;
```

```
OBJECT_TYPE
-----
TABLE
```

- SELECT \* FROM user\_catalog;

```
SQL> select *
      2  from user_catalog;
```

TABLE_NAME	TABLE_TYPE
CONTACT	TABLE
CUSTOMER	TABLE
DEPT	TABLE
ZIP_TABLE	TABLE

- Datatypes

- VARCHAR2(size)
  - Variable-length character data (max size must be specified).
- CHAR(size)
  - Fixed-length character data of length size bytes.
- NUMBER(p,s)
  - Number having precision p and scale s (The precision is the total number of decimal digits, and the scale is the number of digits to the right of the decimal point. The precision can range from 1 to 38 and the scale can range from -84 to 127.
- DATE

- Date and time values between January 1, 4712 B.C. and December 31, 9999 A.D.
  - LONG
    - Variable-length character data up to 2 gigabytes.
  - CLOB
    - Single-byte character data up to 4 gigabytes.
  - RAW and LONG RAW
    - Raw binary data of length size (A maximum size must be specified. Maximum size is 2000).
  - BLOB
    - Binary data up to 4 gigabytes.
  - BFILE
    - Binary data stored in an external file; up to 4 gigabytes.
- Creating a Table by Using a Subquery: Create a table and insert rows by combining the CREATE TABLE statement and AS subquery option. You must make sure to match the number of specified columns with the number of subquery columns. You must define columns with columns names and default values.

```
SQL> CREATE TABLE dept30 AS
  2  SELECT tempo, ename, sal*12 ANNSAL, hiredate
  3  from emp
  4  WHERE deptno = 30;
```

Table created.

```
SQL> describe dept30
Name                               Null?    Type
-----
EMPNO                               NOT NULL NUMBER(4)
ENAME                               VARCHAR2(10)
ANNSAL                              NUMBER
HIREDATE                             DATE
```

```
SQL> select * from dept30;
```

EMPNO	ENAME	ANNSAL	HIREDATE
7499	ALLEN	12000	20-FEB-81
7521	WARD	12000	22-FEB-81
7698	BLAKE	24000	01-MAY-81
7844	TURNER	12000	08-SEP-81

- The previous table created a table containing details of all the employees working in department 30. Notice that all the data in dept30 came from table emp.
- The ALTER TABLE statement is used to: Add a new column, Modify an existing column, or Define a default value for the new column.
- You use the ADD clause to add columns to an existing table:

```
SQL> ALTER TABLE dept30
  2  ADD (job VARCHAR2(9));
```

Table altered.

```
SQL> select * from dept30;
```

EMPNO	ENAME	ANNSAL	HIREDATE	JOB
7499	ALLEN	12000	20-FEB-81	

```

7521 WARD          12000 22-FEB-81
7698 BLAKE        24000 01-MAY-81
7844 TURNER       12000 08-SEP-81

```

- You can add or modify columns, but you cannot drop them from a table. You cannot specify where the column is to appear. The new columns becomes the last column.
- Modifying a Column: You can change the column's data type, size, and value.

```

SQL> describe dept30
Name                               Null?    Type
-----
EMPNO                               NOT NULL NUMBER(4)
ENAME                               VARCHAR2(10)
ANNSAL                              NUMBER
HIREDATE                            DATE
JOB                                  VARCHAR2(9)

```

```

SQL> r
1 alter table dept30
2* modify (job VARCHAR2(20))

```

Table altered.

```

SQL> describe dept30
Name                               Null?    Type
-----
EMPNO                               NOT NULL NUMBER(4)
ENAME                               VARCHAR2(10)
ANNSAL                              NUMBER
HIREDATE                            DATE
JOB                                  VARCHAR2(20)

```

- Dropping a Column: You use the DROP COLUMN clause to drop columns you no longer need from the table.

```

SQL> alter table dept30
2 drop column job;

```

Table altered.

```

SQL> describe dept30;
Name                               Null?    Type
-----
EMPNO                               NOT NULL NUMBER(4)
ENAME                               VARCHAR2(10)
ANNSAL                              NUMBER
HIREDATE                            DATE

```

- SET UNUSED Option: You use the SET UNUSED option to mark one or more columns as unused. You use the DROP UNUSED COLUMNS option to remove the columns that are marked as UNUSED.

```

SQL> alter table dept30
2 set unused column hiredate;

```

Table altered.

```

SQL> select * from dept30;

```

EMPNO	ENAME	ANNSAL
7499	ALLEN	12000
7521	WARD	12000
7698	BLAKE	24000
7844	TURNER	12000

- In this book, it is not shown how to revert to undoing this table modification. Note that column hiredate was not dropped, merely marked as unused.
- Dropping a Table: All data and structure in the table is deleted. Any pending transactions are committed. All indexes are dropped. You cannot rollback this statement.
- **NOTE: Oracle will not question your request to drop a table and the modification is permanent and irreversible! It seems weird that you are not asked to confirm your request before making such a drastic modification, but such is the case!!!**
- Changing the Name of an Object: To change the name of a table, view, sequence, or synonym, you execute the RENAME statement.

```
SQL> rename dept30 to Departamento;
```

Table renamed.

```
SQL> describe Departamento;
```

Name	Null?	Type
ENAME		VARCHAR2 (10)
DEPTNO		NUMBER (2)
ANNSAL		NUMBER
MGR		NUMBER (4)

- The TRUNCATE TABLE statement: Removes all rows from a table, Releases the storage space used by that table. You cannot rollback row removal when using TRUNCATE. Of course, you can remove rows by using the DELETE statement.

```
SQL> TRUNCATE table Departamento;
```

Table truncated.

```
SQL> select * from Departamento;
```

**no rows selected**

- Finally, comments can be added to a table or column by using the COMMENT statement. Such comments can be viewed through the data dictionary views.

```
SQL> COMMENT ON TABLE Departamento
  2 IS 'This table is completely empty -
  3 since it was TRUNCATED';
```

Comment created.

## INCLUDING CONSTRAINTS -- CHAPTER 11 --

- What are constraints? Constraints enforce rules at the table level. Constraints prevent the deletion of a table if there are dependencies.
- The following constraint types are valid in ORACLE:
  - NOT NULL
    - Specifies that this column may not contain a null value.
  - UNIQUE
    - Specifies a column or combination of columns whose values must be unique for all rows in the table.
  - PRIMARY KEY
    - Uniquely identifies each row of the table.
  - FOREIGN KEY
    - Establishes and enforces a foreign key relationship between the column and the column of and a column of the referenced table.
  - CHECK
    - Specifies a condition that must be true.
- Note that if you do not give it a name, Oracle will generate a name with the format SYS\_Cn where n is an integer to create a unique constraint name.
- You can view the constraints defined for a specified table by looking g at the USER\_CONSTRAINTS data dictionary table.

```
SQL> r
1 create table scott.victor3434
2 (numero NUMBER(4),
3 name VARCHAR2(10),
4* CONSTRAINT PK PRIMARY KEY (numero))
```

Table created.

```
SQL> select * from victor3434;
```

no rows selected

```
SQL> describe victor3434;
```

Name	Null?	Type
NUMERO	NOT NULL	NUMBER(4)
NAME		VARCHAR2(10)

- Note that constraints can be added after the table has been created. Note also that constraints may be defined at either the column or the table level.
- More on Foreign Keys: These designate a column or combination of columns as foreign key and establishes a relationship between a primary key or a unique key in the same table or a different table. A foreign key value must match an existing value in the parent table or be NULL. Foreign keys are based on data values and are purely logical, not physical; they are pointers.

```
SQL> r
1 CREATE TABLE emp(
2 empno NUMBER(4),
3 ename VARCHAR2(10) NOT NULL,
4 job VARCHAR2(9),
5 mgr NUMBER(4),
6 hiredate DATE,
7 sal NUMBER(7,2),
8 comm NUMBER(7,2),
9 soc NUMBER(4) NOT NULL,
```

```

10 CONSTRAINT emp_SOC_FK FOREIGN KEY (soc)
11* REFERENCES victor3434 (soc))

```

Table created.

```
SQL> describe emp;
```

Name	Null?	Type
EMPNO		NUMBER(4)
ENAME	NOT NULL	VARCHAR2(10)
JOB		VARCHAR2(9)
MGR		NUMBER(4)
HIREDATE		DATE
SAL		NUMBER(7,2)
COMM		NUMBER(7,2)
SOC	NOT NULL	NUMBER(4)

- Notice now that the victor3434 table, which contains the foreign key in emp prevents such table from being deleted as shown below...

```

SQL> DROP TABLE VICTOR3434
2 /
DROP TABLE VICTOR3434
*
ERROR at line 1:
ORA-02449: unique/primary keys in table referenced by foreign keys

```

- FOREIGN KEY Constraint Keywords:
  - FOREIGN KEY: Defines the column in the child table at the table constraint level
  - REFERENCES: Identifies the table and column in the parent table.
  - ON DELETE CASCADE: Allows deletion in the parent table and deletion of the dependent rows in the child table.
- The CHECK constraint defines a condition that each row must satisfy. The condition can use the same constructs as query conditions, with the following exceptions:
  - References to CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudocolumns
  - Calls to SYSDATE, UID, USER, and USERENV functions
  - Queries that refer to other values in other rows.
- Note that a single column can have multiple CHECK constraints that reference the column in its definition. There is no limit to the number to the limit of CHECK constraints that you can define on a column. Further, CHECK constraints can be defined at either the column or table levels.
- Adding a constraint to a table is performed as shown in the following example:

```

SQL> alter table victor3434
2 add CONSTRAINT boss_ck CHECK(boss = 'Rick');

```

Table altered.

```

SQL> select * from victor3434
2 /

```

SOC	NAME	BOSS	SALARY
3434	Victor	Rick	3000
3535	Ryan	Rick	2500

```

SQL> insert into victor3434
2 values (3232, 'John', 'Rick', 3001);

```

```
1 row created.
```

```
SQL> insert into victor3434
  2 values (3131, 'Jane', 'Jack', 3001);
insert into victor3434
*
ERROR at line 1:
ORA-02290: check constraint (VSANCHEZ.BOSS_CK) violated
```

- Note in the previous example that we created a constraint that makes sure that all rows entered must contain the string 'Rick' under the BOSS column. If this is not the case, insertions are rejected and the appropriate error message is returned, i.e., **VSANCHEZ.BOSS\_CK**.
- Dropping a constraint is also easily accomplished by using the DROP keyword right before the CONSTRAINT keyword when ALTER-ing a table...

```
SQL> r
  1 ALTER table victor3434
  2* DROP CONSTRAINT boss_ck
```

```
Table altered.
```

- Notice that it is not necessary to include the table name ('victor3434.') before the name of the constraint ('boss\_ck').
- Note that the CASCADE option of the DROP clause causes any dependent constraints also to be dropped.

```
ALTER TABLE table
DROP PRIMARY KEY | UNIQUE (column) |
CONSTRAINT constraint [CASCADE];
```

- Execute the DISABLE clause of the ALTER TABLE statement to deactivate an integrity constraint. Apply the CASCADE option to disable dependent integrity constraints.

```
ALTER TABLE table
DISABLE CONSTRAINT constraint [CASCADE];
```

- Similarly, currently disabled constraints can be enabled using the ENABLE clause.

```
ALTER TABLE table
ENABLE CONSTRAINT constraint [CASCADE];
```

- Cascading CONSTRAINTS clause also drops all multicolumn constraints defined on the dropped columns.

```
SQL> ALTER TABLE victor3434 DROP (pk) CASCADE COSNTRAIANTS;
```

- Viewing constraints: After creating constraints, you can confirm its existence by issuing a DESCRIBE command. The only constraint that you can verify is the NOT NULL constraint. To view all constraints on your table, query the USER\_CONSTRAINTS table.

```
SELECT constraint_name, constraint_type, search_condition
FROM user_constraints
WHERE table_name = 'emp';
```

- You can also view the columns associated with constraints as follows:

```
SELECT constraint_name, column_name
FROM user_cons_columns
WHERE table_name = 'emp';
```

- this view is especially useful for constraints that use the system-assigned name.

## CREATING VIEWS -- CHAPTER 12 --

- What is a view? You can present logical subsets or combinations of data by creating views of tables. A view is a logical table based on another table or view. A view contains no data of its own but is like a window through which data from tables can be viewed or changed. The tables on which a view is based are called base tables. The view is stored as a SELECT statement in the data dictionary.
- Why use views? Views restrict access to the data because the view can display selective columns from the table. Views allow users to make simple queries to retrieve the results from complicated queries. For example, views allow users to view data from multiple tables without knowing how to write a join statement. Views provide data independence for ad hoc users and application programs. One view can be used to retrieve data from several tables. Views provide groups of users access to data according to their particular criteria.
- There are two main types of view: Simple and Complex
- Simple View:
  - Derives data from only one table
  - Contains no functions or groups of data
  - Can perform DML through a view
- Complex View:
  - Derives data from many tables
  - Contains functions or groups of data
  - Does not always allow DML through the view
- Creating a View: You can create a subquery within the CREATE VIEW statement.

```
CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW view [(alias[, alias]...)]
AS subquery
[WITH CHECK OPTION [CONSTRAINT constraint]]
[WITH READ ONLY];
```

- OR REPLACE:
  - Re-creates the view if it already exists.
- FORCE:
  - Creates the view regardless of whether or not the base tables exists.
- NOFORCE:
  - Creates the view only if the base table exists (this is the default).
- View:
  - Is the name of the view.
- Alias:
  - Specifies names for the expressions selected by the view's query (The number of aliases must match the number of expressions selected by the view).
- Subquery:
  - Is a complete SELECT statement. Note that this select statement cannot contain an ORDER BY clause.(You can use aliases for the columns in the SELECT list).
- WITH CHECK OPTION:
  - Specifies that only rows accessible to the view can be inserted or updated.
- Constraint:

- Is the name assigned to the CHECK OPTION constraint.
- WITH READ ONLY:
  - Ensures that no DML operations can be performed on this view.
- Here's an example of creating a view and using it to view its contents...

```
SQL> CREATE VIEW empvu10
  2 AS SELECT empno, ename, job
  3 FROM emp
  4 WHERE deptno=10;
```

View created.

```
SQL> select * from empvu10;
```

EMPNO	ENAME	JOB
7782	CLARK	MANAGER
7839	KING	PRESIDENT
7096	BLACK	SALESMAN
2296	AROMANO	SALESMAN
4296	AROMANO	SALESMAN
3354	AROMANO	SALESMAN

6 rows selected.

```
SQL> select *
  2 from empvu10
  3 where empno > 7000;
```

EMPNO	ENAME	JOB
7096	BLACK	SALESMAN
7782	CLARK	MANAGER
7839	KING	PRESIDENT

- Note that once the view is created, you may simply use a select statement to view its contents just as if the view was a table itself!
- You may also use the DESCRIBE statement to see the structure of the view as shown below:

```
SQL> describe empvu10;
Name                               Null?    Type
-----
EMPNO                               NOT NULL NUMBER(4)
ENAME                               VARCHAR2(10)
JOB                                  VARCHAR2(9)
```

- Also, once a view has been created, you can query the data dictionary table called USER\_VIEWS to see the name of the view and the view definition. The text of the SELECT statement that constitutes your views is stored in a LONG column.
- Views can be modified by using the CREATE OR REPLACE VIEW clauses. The following example shows how to add an alias name to an existing view...

```
SQL> CREATE OR REPLACE VIEW empvu10
  2 (employee_number, employee_name, job_title)
  3 AS SELECT empno, ename, job
  4 FROM emp
  5 WHERE deptno = 10;
```

View created.

```
SQL> select * from empvu10;
```

EMPLOYEE_NUMBER	EMPLOYEE_N	JOB_TITLE
7782	CLARK	MANAGER
7839	KING	PRESIDENT
7096	BLACK	SALESMAN
2296	AROMANO	SALESMAN
4296	AROMANO	SALESMAN
3354	AROMANO	SALESMAN

- **Creating a Complex View:** The following example shows how to create a complex view that contains group functions to display values from two tables...

```
SQL> r
  1 CREATE VIEW dept_sum_vu
  2 (name, minsal, maxsal, avgsal)
  3 AS SELECT d.dname, MIN(e.sal), MAX(e.sal), AVG(e.sal)
  4 FROM emp e, dept d
  5 WHERE e.deptno = d.deptno
  6* GROUP BY d.dname
```

View created.

```
SQL> select * from dept_sum_vu;
```

NAME	MINSAL	MAXSAL	AVGSAL
ACCOUNTING	1000	3000	1650
DEVELOPMENT	99	4000	1978.8947
OPERATIONS	5000	5500	5250
RESEARCH	1500	2000	1750
SALES	1000	2000	1250

- **Rules for performing DML Operations on a View**
  - You can perform DML operations on simple views
  - You cannot remove a row if the view contains the following:
    - Group functions
    - A GROUP BY clause
    - The DISTINCT keyword
    - The pseudocolumn ROWNUM keyword.
  - You cannot modify data in a view if it contains:
    - Any of the conditions mentioned in the previous rule
    - Columns defined by expressions
    - The ROWNUM pseudocolumn

- You cannot add data if:
  - The view contains any of the conditions mentioned above or in the previous rule
  - There are NOT NULL columns in the base tables that are not selected by the view.
- Using the WITH CHECK OPTION clause: It is possible to perform referential integrity checks through views. You can also enforce constraints at the database level. The view can be used to protect data integrity, but the use is very limited. The WITH CHECK OPTION clause specifies that INSERTS and UPDATES performed through the view are not allowed to create rows that the view cannot select, and therefore it allows integrity constraints and data validation checks to be enforced on data being inserted or updated.
- Denying DML Operations: You can ensure that no DML operations occur by adding the WITH READ ONLY option to your view definition. Any attempt to perform a DML on any row in the view will result in an ORACLE SERVER error.
- Removing a View: Remove a view without losing data because a view is based on underlying tables in the database.

```
SQL> select * from empvu10;
select * from empvu10
          *
ERROR at line 1:
ORA-00942: table or view does not exist
```

- Dropping views has no effect on the tables on which the views are based.
- Inline Views: An inline view is a subquery with an alias (correlation name) that you can use within a SQL statement. An inline view is similar to using a named subquery in the FROM clause of the main query. An inline view is not a schema object.
- “Top-N” Analysis
  - Top-N queries ask for the n largest or smallest values of all columns
    - What are the ten best selling products?
    - What are the ten worst selling products?
  - Both largest values and smallest values sets are considered Top-N queries.
- Performing “Top-N” Analysis

```
SQL> SELECT [column_list], ROWNUM
FROM (SELECT [column_list] FROM table
ORDER BY Top-N_column)
WHERE ROWNUM <= N;
```

- Top-N queries use a consistent nested query structure with the elements described below:
  - Subquery or an inline view to generate the sorted list of data. The subquery or the inline-view includes the ORDER BY clause to ensure that the ranking is in the desired order. For results retrieving the largest values, DESC parameter is needed.
- Outer query to limit the number of rows in the final result set. The outer query includes the following components:
  - The ROWNUM pseudo-column, which assigns a sequential value starting with 1 to each of the rows returned from the query.
  - WHERE clause, which specifies the n rows to be returned. The outer WHERE clause must use a < or <= operator.

- The following is an example of a Top-N analysis...

```
SQL> r
 1 SELECT ROWNUM as RANK, ename, sal
 2 FROM (SELECT ename, sal FROM emp
 3        ORDER BY sal DESC)
 4* WHERE ROWNUM <= 10
```

RANK	ENAME	SAL
1	SI	
2	VIC_SANC	
3		
4	SHANK	5500
5	MOLITOR	5000
6	ALBERT	4000
7	BOBBY	3000
8	KING	3000
9	JONES	2000
10	CLARK	2000

**OTHER DATABASE OBJECTS**  
**-- CHAPTER 13 --**

- What is a Sequence?
  - Automatically generates unique numbers
  - Is a sharable object
  - Is typically used to create a primary key value
  - Replaces application code
  - Speeds up the efficiency of accessing sequence values when cached in memory
- The CREATE SEQUENCE statement: Define a sequence to generate sequential numbers automatically.

```
CREATE SEQUENCE sequence
  [INCREMENT BY n]
  [START WITH n]
  [{MAXVALUE n | NOMAXVALUE}]
  [{MINVALUE n | NOMINVALUE}]
  [{CYCLE | NOCYCLE}]
  [{CACHE n | NOCACHE}];
```

- INCREMENT BY n
  - Is the name of the sequence generator
- START WITH n
  - Specifies the interval between sequence numbers where n is an integer (If this clause is omitted, the sequence will increment by 1).
- MAXVALUE n
  - Specifies the maximum value the sequence can generate.
- NOMAXVALUE
  - Specifies a maximum value of 10<sup>27</sup> for an ascending sequence and -1 for a descending sequence (This is the default option).
- MINVALUE n
  - Specifies the minimum sequence value.
- NOMINVALUE
  - Specifies a minimum value of 1 for an ascending sequence and 10<sup>26</sup> for a descending sequence (This is the default option).
- CYCLE | NOCYCLE
  - Specifies that the sequence continues to generate values after reaching either its maximum or minimum value or does not generate additional values (NOCYCLE is the default option).
- CACHE n | NOCACHE
  - Specifies how many values the Oracle Server will pre-allocate and keep in memory (By default, the Oracle Server will cache 20 values).
- The following is an example of the creation of a sequence named DEPT\_DEPTNO to be used for the primary key of the dept table. Do not use the CYCLE option.

```
SQL> CREATE SEQUENCE dept_deptno
2 INCREMENT BY 1
3 START WITH 91
```

- 4 MAXVALUE 100
- 5 NOCACHE
- 6 NOCYCLE;

Sequence created.

- Confirming your sequence values in the USER\_SEQUENCES data dictionary table.

```
SQL> SELECT sequence_name, min_value, max_value, increment_by, last_number
       2  from user_sequences;
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPARTMENT_ID_SEQ	1	1.000E+27	1	21
<b>DEPT_DEPTNO</b>	<b>1</b>	<b>100</b>	<b>1</b>	<b>91</b>
DEPT_ID_SEQ	1	80	1	78
MEMBER_ID	1	9999999	1	108
S_CUSTOMER_ID	1	9999999	1	216
S_DEPT_ID	1	9999999	1	51
S_EMP_ID	1	9999999	1	26
S_IMAGE_ID	1	9999999	1	1981
S_LONGTEXT_ID	1	9999999	1	1369
S_ORD_ID	1	9999999	1	115
S_PRODUCT_ID	1	9999999	1	50537
S_REGION_ID	1	9999999	1	6
S_WAREHOUSE_ID	1	9999999	1	10502
TEST_SEQ	1	1.000E+27	1	1
TITLE_ID	1	9999999	1	100
WORKER_ID_SEQ	1	9999999	1	204

16 rows selected.

- Note that there are 15 other sequences that were created by some one else in the Scott/Tiger schema...
- Using a sequence: Once you create your sequence, you can use the sequence to generate sequential numbers for use in your tables. Reference the sequence values by using the NEXTVAL and CURRVAL pseudocolumns.
- The NEXTVAL pseudocolumn is used to extract successive sequence numbers from a specified sequence. You must qualify ENXTVAL with the sequence name. When you reference sequence. ENXTVAL, a new sequence number is generated and the current sequence number is placed in CURRVAL.
- The CURRVAL pseudocolumn is used to refer to a sequence number that the current user has just generated. NEXTVAL must be used to generate a sequence number in the current user's session before CURRVAL can be referenced. You must qualify CURRVAL with the sequence name. WHEN sequence. CURRVAL is referenced, the last value returned to that user's process is displayed.
- Rules for using NEXTVAL and CURRVAL
  - You can use NEXTVAL and CURRVAL in the following:
    - The SELECT LIST OF A SELECT statement that is not part of the subquery
    - The SELECT list of a subquery in the INSERT statement.
    - The VALUES clause of an INSERT statement
    - The SET clause of an UPDATE statement

- You cannot use NEXTVAL and CURRVAL in the following:
  - A SELECT list of a view
  - A SELECT statement with the DISTINCT keyword
  - A SELECT statement with the GROUP BY, HAVING, or ORDER BY clauses
  - A subquery in a SELECT, DELETE, or UPDATE statement
  - A DEFAULT expression in a CREATE TABLE or ALTER TABLE statement
- Using a sequence: Insert a new department named “MARKETING” in San Diego.

```
SQL> INSERT INTO dept(deptno, dname, loc)
  2  values (dept_deptno.NEXTVAL,
  3          'MARKETING', 'SAN DIEGO');
```

1 row created.

```
SQL> select * from dept
  2  where deptno = 91;
```

DEPTNO	DNAME	LOC
91	MARKETING	SAN DIEGO

- View the current value for the dept\_deptno sequence...

```
SQL> select dept_deptno.CURRVAL
  2  from dual;
```

CURRVAL
91

- Here's another example...

```
SQL> r
  1  INSERT INTO dept(deptno, dname, loc)
  2* values (dept_deptno.NEXTVAL, 'SCIENCE', 'SANTA ANA')
```

1 row created.

```
SQL> select * from dept
  2  where deptno = dept_deptno.currval;
```

DEPTNO	DNAME	LOC
92	SCIENCE	SANTA ANA

- Note here that we selected the record whose deptno is the same as the current value selected in the sequence dept\_deptno.
- Modifying a sequence: Change the increment value, maximum, minimum, cycle option or cache option...\

```
SQL> ALTER SEQUENCE dept_deptno
  2  INCREMENT BY 5
```

```

3 MAXVALUE 999999
4 MINVALUE 20
5 NOCACHE
6 NOCYCLE;

```

Sequence altered.

```

SQL> SELECT sequence_name, min_value, max_value, increment_by, last_number
2 from user_sequences
3* where sequence_name = 'DEPT_DEPTNO'

```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	LAST_NUMBER
DEPT_DEPTNO	20	999999	5	97

- Guidelines for Modifying a Sequence:
  - You must be the owner of have the ALTER privilege for the sequence.
  - Only future sequence numbers are affected
  - The sequence must be dropped and re-created to restart the sequence at a different number
  - Some validation is performed.
- Removing a Sequence: Remove a sequence from the data dictionary by using the DROP SEQUENCE statement. Once removed, the sequence can no longer be referenced.

```
SQL> DROP SEQUENCE dept_deptno;
```

Sequence dropped.

- What a is an Index?
  - A schema object
  - Is issued y the Oracle Server to speed up the retrieval of rows by using a pointer
  - Can reduce disk I/O by using rapid path access method to locate the data quickly
  - Is independent of the table it indexes
  - Is used and maintained automatically by the Oracle Server.
- How are indexes created?
  - Automatically:
    - A unique index is created when you define a PRIMARY KEY or UNIQUE constraint in a table definition.
  - Manually:
    - Users can create non-unique indexes on columns to speed up access time to the rows.
- Creating an Index: Create an index on one or more columns.

```

SQL> r
1 CREATE INDEX emp_ename_idx
2* ON emp(ename)

```

Index created.

- More Is Not Always Better: More indexes on a table does not mean it will speed up queries. Each DML operation that is committed on a table with indexes means that the indexes must be updated. The

more indexes you have associated with a table, the more effort the Oracle Server must make to update all the indexes after a DML.

- When to Create an Index:
  - The column is used frequently in the WHERE clause or in a join condition
  - The column contains a wide range of values
  - The column contains a large number of null values.
  - Two or more columns are frequently used together in a WHERE clause or join condition.
  - The table is large and most queries are expected to retrieve less than 2-4% of the rows.
- Confirming Indexes: The USER\_INDEXES data dictionary view contains the name of the index and its uniqueness. The USER\_IND\_COLUMNS view contains the index name, the table name, and the column name.

```
SQL> r
 1 SELECT ic.index_name, ic.column_name,
 2 ic.column_position col_pos, ix.uniqueness
 3 FROM user_indexes ix, user_ind_columns ic
 4 WHERE ic.index_name = ix.index_name
 5* AND ic.table_name = 'EMP'
```

INDEX_NAME	COLUMN_NAME	COL_POS	UNIQUENES
SYS_C008971	EMPNO	1	UNIQUE
EMP_ENAME_IDX	ENAME	1	NONUNIQUE

- Function-Based Indexes: A function-based index is an index based on expressions. The index expression is built from table columns, constants, SQL functions, and user-defined functions.
- Removing an Index

```
SQL> DROP INDEX indexName;
```

- You cannot modify indexes. To change an index, you must drop it and then re-create it. Remove an index definition from the data dictionary by issuing the DROP INDEX statement. You must be the owner of the index or have the DROP ANY INDEX privilege.
- Synonyms: Simplify access to objects by creating a synonym. Refer to a table owned by another user. Shorten lengthy object names.

```
CREATE [PUBLIC] SYNONYM synonym
FOR object;
```

- PUBLIC: creates a synonym that is accessible to all users
- Synonym: is the name of the synonym to be created
- Object: Identifies the object for which the synonym is created.
  - The object cannot be contained in a package
  - A private synonym name must be distinct from all other objects owned by the same user.
- Creating and Removing Synonyms: Create a shortened name for the DEPT\_SUM\_VU view.

```
SQL> CREATE SYNONYM d_sum
 2 FOR dept_sum_vu;
```

Synonym created.

- **Removing a Synonym:** To drop a synonym, use the `DROP SYNONYM` statement. Only the DBA can drop a public synonym.

```
SQL> DROP SYNONYM d_sum;  
Synonym dropped.
```