RIYOSU PTY LTD

Oracle SQL

Volume 1

Student Guide

Document ID : **OracleSQL10**
Revision : **1.0**
Date : **22 July 2003**

Contents

1. Introduction

1.1 Typographic Conventions

The below table indicates the typographic conventions fro all the examples used throughout this manual.

Convention	Meaning	Example
Bold, Uppercase	SQL Reserved Word	***SELECT* * *FROM* employees**
Bold, Lowercase	Database Objects	**SELECT * FROM employees**
Bold, Italic	Literal Value	**SELECT *"F"* from employees**

2. Basic SQL

2.1 Objectives

After this chapter you should be able to perform the following,

- Execute basic **SELECT** statement.
- Understand basic functionality and format of the **SELECT** statement.

2.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

TABLE : employees			
empno	**empname**	**salary**	**Manager**
AAAA	Greg		AAAC
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	John Paul	5000	AAAD

2.3 SELECT Statement Usage

The **SELECT** statement retrieves data from the database. The **SELECT** statement has the following usage,

- Joining Joins the data from multiple tables to form one set of results.
- Projection Choose specific columns from a table.
- Selection Choose specific rows from a table.

The following rules and guidelines apply to the **SELECT** statement,

- SQL statements can be split over multiple lines.
- SQL statements are not case sensitive.
- SQL reserved words cannot be split over multiple lines.
- SQL reserved word cannot be abbreviated.
- Clauses are split across multiple lines for easy legibility.
- Indents are used for easy legibility.
- Only SQL reserved word are entered in uppercase.

2.4 Basic SELECT Statement

All **SELECT** statements must include two clauses,

- The **SELECT** clause identifies which columns to select.
- The **FROM** clause identifies which table to select the columns from.

Example

Basic **SELECT** statement

SELECT empno, empname FROM employees	
empno	**empname**
AAAA	Greg
AAAB	Mathew
AAAC	John
AAAD	Barry
AAA_E	John Paul

The above example selects the **empno** and **empname** fields from the **employees** table.

2.5 Column Selection

As discussed in section 2.2 the **SELECT** clause identifies the list of columns to display. The SQL statement can display all columns or a list of specific columns.

- Displaying all columns, You can display all columns of a table by following the **SELECT** clause with an asterisk "*".
- Displaying specific columns, You can display specific columns of a table by following the **SELECT** clause with individual column names.

Example 1

The result of the below selection will display the information for every column in the table.

SELECT * FROM employees			
empno	**empname**	**salary**	**manager**
AAAA	Greg		AAAC
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	John Paul	5000	AAAD

Example 2

The result of the below selection will display the information for every column specified on the **SELECT** clause.

SELECT empno, empname FROM employees	
empno	**empname**
AAAA	Greg
AAAB	Mathew
AAAC	John
AAAD	Barry
AAA_E	John Paul

2.6 Null Values

A null value is a column that contains no data. A null value is not a zero or a space, zero and space have ASCII values. Any column can contain a null value except for columns that have constraints of "NOT NULL" and "PRIAMRY KEY". It is important to remember that any arithmetic expression that contains a null will result in a null.

Example 1

Arithmetic Expressions with null values

SELECT empno, empname, salary, salary+1000 FROM employees			
empno	**empname**	**salary**	**salary+1000**
AAAA	Greg		
AAAB	Mathew	10000	11000
AAAC	John	10000	11000
AAAD	Barry	10000	11000
AAA_E	John Paul	5000	5100

The first row in the above example return a null value in the **salary+1000** field due to the null value in the salary field.

2.7 Arithmetic Operators

SQL offers the add(+), subtract(-), multiply(*) and divide(/) arithmetic operators for use when entering SQL statements. The arithmetic operators can be used to calculate and display specific values based on rows retrieved from an SQL statement.

Example 1

Using arithmetic operators

SELECT empno, empname, salary, salary + 1000 FROM employees			
empno	**empname**	**salary**	**salary + 1000**
AAAA	Greg		
AAAB	Mathew	10000	11000
AAAC	John	10000	11000
AAAD	Barry	10000	11000

The above example evaluates the arithmetic expression **salary + 1000**.

If an arithmetic expression contains various operators the operator precedence is as follows,

- Multiplication and division occur before addition and subtraction
- Multiplication and division are of the same precedence.
- Addition and subtraction are of the same precedence.
- Same precedence operators are evaluated from left to right.

Overriding the rule of operator precedence is achieved by the use of parenthesis.

Example 2

Using parenthesis

SELECT empno, empname, salary, 2*(salary + 1000) FROM employees			
empno	**empname**	**salary**	**2*(salary + 1000)**
AAAA	Greg		
AAAB	Mathew	10000	22000
AAAC	John	10000	22000
AAAD	Barry	10000	22000

The above example overrides the rules of precedence and evaluates the part of the arithmetic expression within the parenthesis first.

2.8 Column Aliases

The results of an SQL statement are categorized under column headings. Sometimes these column headings are difficult to understand and are non descriptive. It is for this reason we introduce column aliases.

Rules for using column aliases are as per below,

- Must come immediately after the column name.
- Columns aliases must be enclosed in double quotes for case sensitivity or if they contain spaces or special characters (#,$,%,^,etc…).
- By default column aliases appear in uppercase.
- The optional reserved word **AS** is used to make the column alias easier to read. It appears between the column name and alias.

Example 1

Using column aliases

SELECT empno "Number", empname "Name" FROM employees	
Number	**Name**
AAAA	Greg
AAAB	Mathew
AAAC	John
AAAD	Barry

The above example replaces the column names from the table with the literal column aliases defined in the **SELECT** clause.

Example 2

Using column aliases

SELECT empno AS "Number", empname AS "Name" FROM employees	
Number	**Name**
AAAA	Greg
AAAB	Mathew
AAAC	John
AAAD	Barry

The above example is exactly the same as example 1 but is easier to read because of the use of the **AS** keyword.

2.9 Concatenation Operator

Columns can be merged together to form a single column in the SQL statement output. This is achieved by using the concatenation operator (||).

Example 1

Using the concatenation operator

SELECT empno||empname AS "Names" FROM employees
Names
AAAAGreg
AAABMathew
AAACJohn
AAADBarry

The above example concatenates the **empno** and **empname** columns into one column under the alias **Names**.

2.10 Literal Strings

Literals are free text that appear in the **SELECT** clause that do not identify any column name or column alias. The literal value included in the **SELECT** clause will be displayed for every row retuned from the SQL statement.

Rules for literals,

- Dates must be enclosed in single quotes.
- Strings must be included in single quotes.
- Numeric literals do not require single quotes.

Example 1

Using literal values

SELECT 'Number :'||empno, 'Name:'||empname FROM employees	
Number :||empno	**Name:||empname**
Number:AAAA	Name:Greg
Number:AAAB	Name:Mathew
Number:AAAC	Name:John
Number:AAAD	Name:Barry

The above example inserts the literal values 'Number:' and 'Name:' whilst using the concatenate operator.

2.11 Duplicate Rows

The default for SQL statement processing is to display all rows including duplicate rows. In order to eliminate duplicate rows we use the reserved word **DISTINCT** immediately after the **SELECT** clause.

Example 1

The below example displays all rows including all duplicate values

SELECT salary FROM employees
salary
10000
10000
10000

Example 2

The below example eliminates duplicate rows because **DUPLICATE** is used

SELECT DISTINCT salary FROM employees
salary
10000

2.12 Summary

- The **SELECT** statement is used for retrieving data from the database.
- The **SELECT** and **FROM** clauses form the basis of all SQL statements.
- List the columns to display by placing the column names after the **SELECT** clause.
- List the tables to read from by placing the table names after the **FROM** clause.
- Use the asterisk (*) to display all columns.
- Null values means no data.
- Arithmetic expressions using null values result in null values.
- Column aliases are used to improve the legibility of column headings.
- Use the concatenation operator (||) to merge columns in the SQL results.
- Literal strings are free form text displayed for every row returned in an SQL statement.
- Duplicate rows are removed with the use of the reserved word **DISTINCT**.

2.13 Exercises

1. Display all columns from the employees table.
2. Display the empno and empname from the employees table.
3. Display the empno and empanme columns of the employees table in a concatenated format with a column alias of “Emp Details”.
4. Display all distinct values from salary column in the of employees table.

3. Data Conditioning

3.1 Objectives

After this chapter you should be able to perform the following,

- Limit the number of rows returned from an SQL query.
- Sort the rows returned from an SQL query.

3.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

TABLE : employees			
empno	**empname**	**salary**	**manager**
AAAA	Greg		AAAC
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	Ben	5000	AAAD

3.3 Limiting Rows

The method of limiting the amount of data displayed by an SQL query is performed using the **WHERE** clause. The **WHERE** clause places constraints on the data being selected, this means that only rows that match a certain condition are displayed.

The **WHERE** clause comes immediately after the **FROM** clause and has the following elements,

- Column name
- Condition
- Column name, constant, literal values(s)

Example 1

The below example displays all rows that meet the condition in the WHERE clause

SELECT * FROM employees WHERE salary = 10000			
empno	**empname**	**salary**	**manager**
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	

The format of the condition used in the above example is as follows,
WHERE Column name(**salary**) Condition(**=**) Literal value(**10000**)

3.4 Using Literals

Literals are values used for searching when using the **WHERE** clause. When using literals in the **WHERE** clause certain rules must be followed as per below,

- Dates must be enclosed in single quotes.
- The default date format is DD-MON-RR.
- Strings must be included in single quotes.
- Numeric literals do not require single quotes.

Example 1

The below example displays all rows where the empname columns is equal to the literal 'Barry'

SELECT * FROM employees WHERE empname = 'BARRY'			
empno	**empname**	**salary**	**Manager**
0 rows selected			

The example failed to return any rows because the **WHERE** condition was not meet. This is because literal strings are case sensitive.

Example 2

The below example displays all rows that meet the condition in the **WHERE** clause

SELECT * FROM employees WHERE empname = 'Barry'			
empno	**empname**	**salary**	**Manager**
AAAD	Barry	10000	

The example returned rows because the WHERE condition was meet. This is because the expression **empname = 'Barry'** is using the correct case.

3.5 Conditional Operators

Conditional operators are used to determine the condition under which data is selected. The following conditional operators can be used within the WHERE clause,

- = Equal to.
- > Greater than.
- >= Greater than or equal to.
- < Less than.
- <= Less than or equal to.
- <> Not equal to.
- != Not equal to.
- ^= Not equal to.

Rules for usage in the WHERE clause,

- Column aliases cannot be used in the **WHERE** clause.

Example 1

The below examples list various ways to use conditional operators

WHERE salary > 1000
WHERE empname <> 'John'
WHERE salary <= 20000

Example 2

This example will return all rows that meet the expression in the **WHERE** clause

SELECT * FROM employees WHERE salary <= 10000			
empno	**empname**	**salary**	**manager**
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	Ben	5000	AAAD

Only the rows that have a **salary** less than or equal to **10000** are displayed.

3.6 Additional Conditional Operators

The additional conditional operators listed below can also be used to determine the condition under which data is selected.

- BETWEEN, AND Between two values.
- IN In a list of values.
- LIKE Matches a string sequence.
- IS NULL Is a null value.

3.6.1 The BETEEN Conditional Operator

The **BETWEEN**, **AND** conditional operators are used to select values that are between a range of two values. The **BETWEEN** conditional operator test if a value falls in the range of a lower limit and an upper limit as per example 1.

3.6.2 The IN Conditional Operator

The IN conditional operator (also known as the membership operator) is used to search for a specific value in a list of values as per example 2. The same literal rules mentioned in section 3.4 apply to the list of values used with the IN conditional operator. The list of values defined can be of any data type.

3.6.3 The LIKE Conditional Operator

The LIKE conditional operator is used to perform wildcard searches on literal string values as per example 3. The actual characters used to represent wildcards are as per below,

- % Indicates any sequence of characters.
- _ Indicates a single character.
- \ Escape Option, used to search for the actual % and _ characters.

The two wildcard characters can be combined to search for certain characters in a specific position as per example 4. If there is a requirement to search for the actual wildcard characters “%” or “_” then the **ESCAPE** option is used as per example 5.

3.6.4 The NULL/NOT NULL Conditional Operator

The **IS NULL** and the **IS NOT NULL** conditional operators are used to search for null/non null values as per example 6. Because nulls do not actually equal a value we cannot use the equals”=” or not equals“<>” operator.

Example 1

This example returns rows that fall between the range of 10000 and 20000

SELECT * FROM employees WHERE salary BETWEEN 10000 and 20000			
empno	**empname**	**salary**	**manager**
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	

Example 2

This example returns rows that exist in the list of values

SELECT * FROM employees WHERE empname IN ('Mathew','John')			
empno	**empname**	**salary**	**manager**
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD

Example 3

This example returns rows that have the character **'M'** starting in the **empname** column

SELECT * FROM employees WHERE empname LIKE ('M%')			
empno	**empname**	**salary**	**Manager**
AAAB	Mathew	10000	AAAC

Example 4

This example returns rows with **'a'** as the second character of the **empname** column

SELECT * FROM employees WHERE empname LIKE ('_a%%')			
empno	**empname**	**salary**	**manager**
AAAB	Mathew	10000	AAAC
AAAD	Barry	10000	

Example 5

This example returns rows that have the character **"_"** in the **empno** column

SELECT * FROM employees WHERE empno LIKE ('%_\%') ESCAPE '\'			
empno	**empname**	**salary**	**manager**
AAA_E	Ben	5000	AAAD

Example 6

This example returns rows that have a null value in the manager column

SELECT * FROM employees WHERE manager IS NULL			
empno	**empname**	**salary**	**manager**
AAAD	Barry	10000	

If the above example used the **IS NOT NULL** conditional operator then all rows that were not null would be displayed.

3.7 Logical Conditional Operators

Logical operators are used to combine multiple conditions of an SQL expression. Data is only returned if the overall results of the SQL expression is true.

The logical conditional operators are as follows,

- AND Evaluates to true if both conditions are true.
- OR Evaluates to true if either conditions are true.
- NOT Evaluates to true if the next condition is false.

Example 1

The AND conditional operator

SELECT * FROM employees WHERE salary = 1000 AND empname = 'Mathew'			
empno	**empname**	**salary**	**manager**
AAAB	Mathew	10000	AAAC

Example 2

The OR conditional operator

SELECT * FROM employees WHERE empno = 'AAA_E' or empname = 'Greg'			
empno	**empname**	**salary**	**manager**
AAAA	Greg		AAAC
AAA_E	Ben	5000	AAAD

Example 3

The NOT conditional operator

SELECT * FROM employees WHERE salary NOT IN (10000)			
empno	**empname**	**salary**	**manager**
AAAA	Greg		AAAC
AAA_E	Ben	5000	AAAD

3.8 Rules of Precedence

The rules of precedence are used to determine the order in which SQL expressions are evaluated. As with general arithmetic operators the rules of precedence can also be overridden with the use of parenthesis. The conditional operators below are listed in order of evaluation.

1) Arithmetic operators
2) Concatenation operator
3) Comparison conditions
4) IS NULL, IS NOT NULL, LIKE, [NOT] IN
5) [NOT] BETWEEN
6) NOT logical condition
7) AND logical condition
8) OR logical condition

Example 1

Rules of precedence example where the AND condition is evaluated first

SELECT * FROM employees WHERE
empname = 'Greg' OR
empname = 'Mathew' AND
salary = 10000

empno	empname	salary	manager
AAAA	Greg		AAAC
AAAB	Mathew	10000	AAAC

Example 2

Rules of precedence example where the AND condition is evaluated first

SELECT * FROM employees WHERE
empname = 'Greg' AND
empname = 'Mathew' OR
salary = 10000

empno	empname	salary	manager
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	

3.9 ORDER BY Clause

The **ORDER BY** clause is the last statement in the **SELECT** statement. It is used to sort the results of an SQL expression is a descending or ascending (default) order. The reserved words **ASC** (ascending) and **DESC** (descending) are used to determine the sort order.

The value to sort by can be of the following,

- An expression.
- A column alias.
- A column name.
- A column number.

When sorting in the default sort order of ascending data is displayed in the following order,

- Nulls are displayed last using the default order of ascending and first if descending is used.
- Numeric values are displayed from lowest to highest.
- Character vales are ordered by ASCII value from lowest to highest.

The **ORDER BY** clause can also be used with the following techniques,

- Sorted data using multiple columns.
- Sorting data that is not defined in the **SELECT** clause.

Example 1

The below example displays all rows and sort by **empname**

SELECT * FROM employees ORDER BY empname			
empno	**empname**	**salary**	**manager**
AAAD	Barry	10000	
AAA_E	Ben	5000	AAAD
AAAA	Greg		AAAC
AAAC	John	10000	AAAD
AAAB	Mathew	10000	AAAC

The above example uses the default order of ascending order

Example 2

The below example sorts in descending order

SELECT * FROM employees ORDER BY empname DESC			
empno	**empname**	**salary**	**manager**
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAA	Greg		AAAC
AAA_E	Ben	5000	AAAD
AAAD	Barry	10000	

Example 3

The below example sorts in ascending order using a column alias

SELECT emp, empname Emp, salart, manager FROM employees ORDER BY Emp			
empno	**Emp**	**salary**	**manager**
AAAD	Barry	10000	
AAA_E	Ben	5000	AAAD
AAAA	Greg		AAAC
AAAC	John	10000	AAAD
AAAB	Mathew	10000	AAAC

Example 4

The below example displays all rows and is sorted using a column number

SELECT * FROM employees ORDER BY 2			
empno	**empname**	**salary**	**manager**
AAAD	Barry	10000	
AAA_E	Ben	5000	AAAD
AAAA	Greg		AAAC
AAAC	John	10000	AAAD
AAAB	Mathew	10000	AAAC

Example 5

The below example displays all rows and is sorted using two column names

SELECT * FROM employees ORDER BY salary, manager			
empno	**empname**	**salary**	**manager**
AAA_E	Ben	5000	AAAD
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAAA	Greg		AAAC

3.10 Summary

- The WHERE clause is used to limit data returned by an SQL expression.
- Certain rules for literals must be followed using the WHERE clause.
- Conditional operators are used to determine the condition under which data is selected.
- Logical operators are used to combine multiple conditions of an SQL expression.
- The rules of precedence are used to determine the order in which SQL expressions are evaluated.
- The ORDER BY clause us used to sort the results of an SQL expression is a particular order.

3.11 Exercises

- Select all rows from the employees table where salary is between 10000 and 20000 and order the output on the empname column in descending order.
- Select all rows from the employees table where salary equals 10000 and empname equals Greg.
- Select all rows from employees table where the salary column is null and the manager columns is null.
- Select all rows from the employees table using the following conditions,
    - empname equals Greg

  OR
    - empname equals Mathew AND salary = 10000

4. Single Row Functions

4.1 Objectives

After this chapter you should be able to perform the following,

- Use various types of character, number and date type functions in SQL expressions.
- Use various types of conversion functions.

4.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

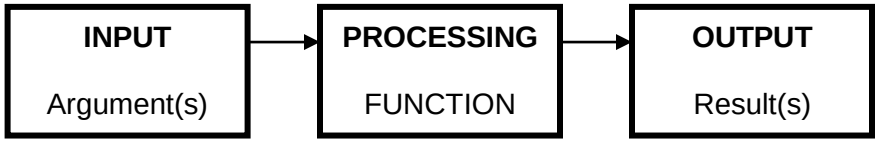
TABLE : employees			
empno	**empname**	**Salary**	**manager**
AAAA	Greg		AAAC
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	John paul	5000	AAAD

4.3 Single Row Functions

Single row functions are used to perform the following,

- Data calculation and manipulation.
- Data formatting.
- Data type conversions.

Functions are called within the block of the SQL statement. Most of the times the function is passed arguments that are used in the functions processing. The function then accepts the arguments and uses them with its own set of instructions to produce a set of results.



Functions will accept any of the following type of arguments,

- Constant Value.
- Pre-defined Variable.
- Column name.
- Expression.

Functions have the following powerful capabilities,

- Functions can be nested within functions.
- Functions can be called in the **SELECT**, **WHERE** and **ORDER BY** clauses.

Single row functions are used on individual rows. They return one set of results per row. Singe row functions are divided into different classes,

- Character functions — Accept character type arguments and return both characters and numbers.
- Number functions — Accept numeric type arguments and return numeric values.
- Date functions — Accept date type arguments and return date type value. However, the function **MONTHS_BETWEEN** returns a Numeric value.
- Date type conversion — Converts one data type to another.
- Other functions — **NVL**, **NVL2**, **NULLIF**, **COALSECE**, **CASE**, **DECODE**.
- Conditional functions — Implements if-then-else logic in SQL statements.

4.4 Character Functions

Character functions are used to manipulate and examine characters within a string. Character functions can return both character and numeric values. There are two types of character manipulation functions,

4.4.1 Case Manipulation Functions

Case manipulation functions are used to change the case of characters in a string. The three case manipulation functions are **LOWER**, **UPPER** and **INITCAP**.

4.4.1.1 LOWER

Converts a string to lowercase. The usage of the **LOWER** function requires one mandatory argument as per below,

LOWER(column/expression)

Example 1

This example of the **LOWER** function converts all character in the empname column to lowercase

SELECT empno, LOWER(empname), salary, manager from employees			
empno	**LOWER (empname)**	**Salary**	**manager**
AAAA	Greg		AAAC
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	john paul	5000	AAAD

4.4.1.2 UPPER

Converts a string to uppercase. The usage of the **UPPER** function requires one mandatory argument as per below,

UPPER(column/expression)

Example 1

This example of the **UPPER** function converts all character in the **empname** column to lowercase

SELECT empno, UPPER(empname), salary, manager from employees			
empno	**UPPER (empname)**	**Salary**	**manager**
AAAA	GREG		AAAC
AAAB	MATHEW	10000	AAAC
AAAC	JOHN	10000	AAAD
AAAD	BARRY	10000	
AAA_E	JOHN PAUL	5000	AAAD

4.4.1.3 INITCAP

Converts the first character of each word in a string to uppercase and the rest to lowercase. The usage of the **INITCAP** function requires one mandatory argument as per below,

INITCAP(column/expression)

Example 1

This example of the **INITCAP** function performs a conversion on the **empname** column

SELECT empno, INITCAP(empname), salary, manager from employees			
empno	**INITCAP (empname)**	**Salary**	**Manager**
AAAA	Greg		AAAC
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	John Paul	5000	AAAD

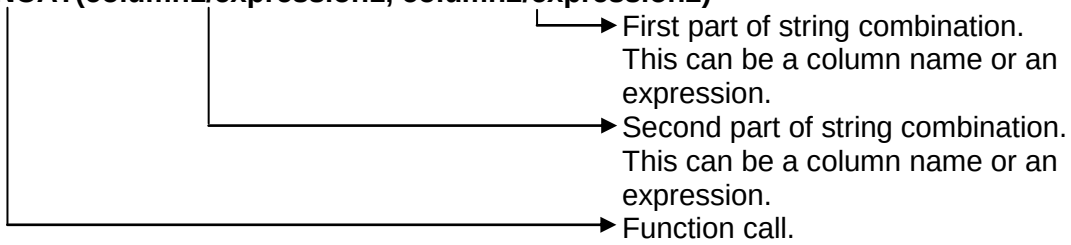
4.4.2 Character Manipulation Functions

There are various types of character manipulation functions. Character manipulation functions are used to format and manipulate the characters of a string. This section explains some of the commonly used character manipulation functions.

4.4.2.1 CONCAT

The **CONCAT** function combines two string into one like the "||" operator. Unlike the "||" operator the **CONCAT** function only allows the combination of two strings.

CONCAT(column1/expression1, column2/expression2)



Example 1

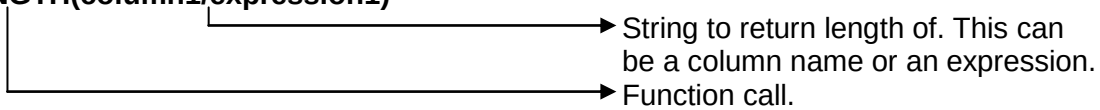
This example of the **CONCAT** function combines the **empno** and **empname** columns

SELECT empno, CONCAT(empno, empname), salary, manager from employees			
empno	**CONCAT (empno, empname)**	**Salary**	**manager**
AAAA	AAAA Greg		AAAC
AAAB	AAAB Mathew	10000	AAAC
AAAC	AAAC John	10000	AAAD
AAAD	AAAD Barry	10000	
AAA_E	AAA_E John Paul	5000	AAAD

4.4.2.2 LENGTH

The **LENGTH** function returns the length of a string.

LENGTH(column1/expression1)



Example 1

This example of the **LENGTH** function return the length of the **empname** column

SELECT empno, LENGTH(empname), salary, manager from employees			
empno	**LENGTH (empname)**	**Salary**	**manager**
AAAA	4		AAAC
AAAB	6	10000	AAAC
AAAC	4	10000	AAAD
AAAD	5	10000	
AAA_E	9	5000	AAAD

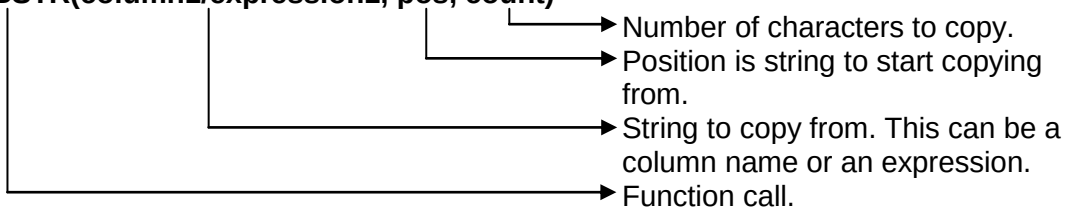
4.4.2.3 SUBSTR

The **SUBSTR** function copies a sub-string from a string based on certain parameters passed into the function.

Rules for using the **SUBSTR** function,

- If **pos** is negative then the copy begins at the end of the string.
- If **count** is omitted then all characters from the start of the copy to the end of the string are copied.

SUBSTR(column1/expression1, pos, count)



Example 1

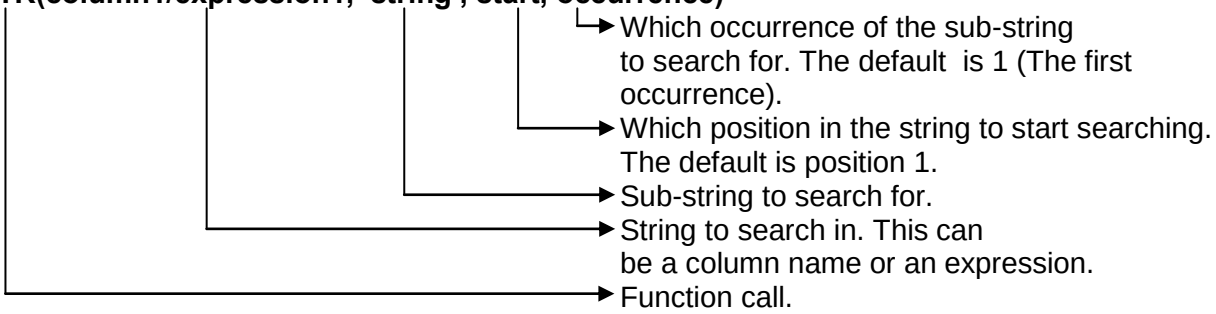
This example of the **SUBSTR** function copies one character from the **empno** column

SELECT SUBSTR(empno,4,1), empname, salary, manager from employees			
empno	**SUBSTR (empno,4,1)**	**Salary**	**manager**
AAAA	A		AAAC
AAAB	B	10000	AAAC
AAAC	C	10000	AAAD
AAAD	D	10000	
AAAA_E	_	5000	AAAD

4.4.2.4 INSTR

The **INSTR** function returns the starting position of a sequence of characters.

INSTR(column1/expression1, 'string', start, occurrence)



Example 1

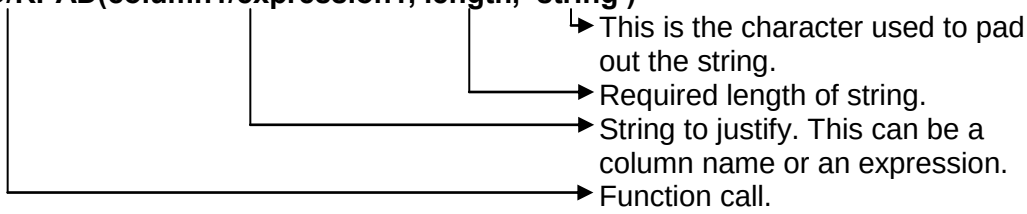
This example of the **INSTR** function return the length of the **empname** column

SELECT empno, INSTR(empname,'a',1,1), salary, manager from employees			
empno	**INSTR (empname,'a',1,1)**	**salary**	**manager**
AAAA	0		AAAC
AAAB	2	10000	AAAC
AAAC	0	10000	AAAD
AAAD	2	10000	
AAA_E	7	5000	AAAD

4.2.2.5 LPAD/RPAD

The **LPAD** and **RPAD** functions justify the characters in the string to left (**RPAD**) or to the right (**LPAD**). The justification works by padding out the string with a certain sequence of characters.

LPAD/RPAD(column1/expression1, length, 'string')



Example 1

This example of the **LPAD** function will right justify the **empname** column using the asterisk '*' character

SELECT empno, LPAD(empname, 15, '*'), salary, manager from employees			
empno	**LPAD (empname, 15, '*')**	**Salary**	**manager**
AAAA	**********Greg		AAAC
AAAB	*********Mathew	10000	AAAC
AAAC	**********John	10000	AAAD
AAAD	**********Barry	10000	
AAA_E	******John Paul	5000	AAAD

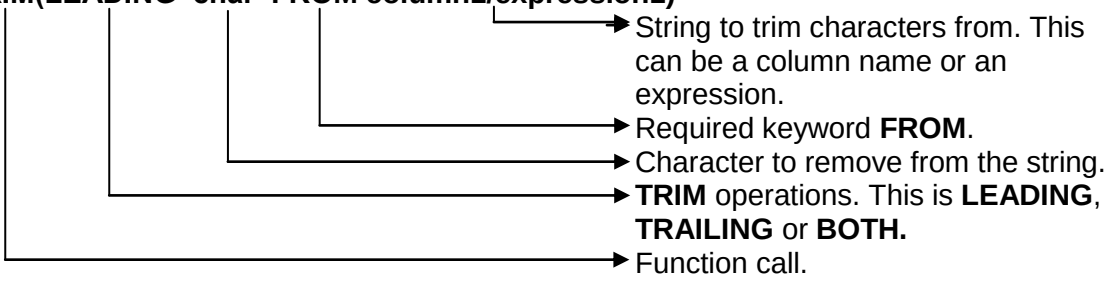
4.2.2.6 TRIM

The **TRIM** function removes leading or trailing characters from a string. Characters can be removed from the start, finish or both ends of the string by using one of the following reserved words as a parameter in the TRIM function,

- **LEADING** Remove characters from the start of the string.
- **TRAILING** Remove characters from the end of the string.
- **BOTH** Remove characters from the start and end of the string.

If none of the above reserved words are used the default operation is **BOTH**.

TRIM(LEADING 'char' FROM column1/expression1)



Example 1

This example uses the **TRIM** function to remove the '**G**' character from the **empname** column.

SELECT empno, TRIM(LEADING 'G' FROM empname), salary, manager from employees			
empno	**TRIM (LEADING 'G' FROM empname)**	**salary**	**manager**
AAAA	reg		AAAC
AAAB	Mathew	10000	AAAC
AAAC	John	10000	AAAD
AAAD	Barry	10000	
AAA_E	John Paul	5000	AAAD

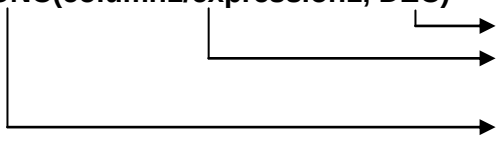
4.5 Number Functions

Number functions are used to perform arithmetic operations on numeric values. There are various types of number functions. Number functions take numeric values as arguments in order to return a numeric value. This section explains some of the most commonly used number functions.

4.5.1 TRUNC

The **TRUNC** function truncates a number to a certain amount of decimal places.

TRUNC(column1/expression1, DEC)



Number of decimal places to truncate the number to.
This is the numeric value to truncate. This can be a column name or an expression.
Function call.

The argument passed in as **DEC** effects the outcome of the number in the following manner,

- If **DEC** is not specified then the default is zero.
- If **DEC** is negative the decimal place is shifted to the left.

Example 1

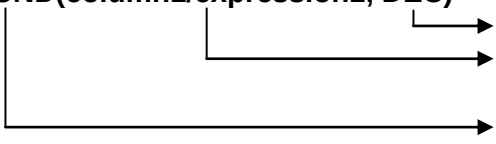
This example of the **TRUNC** function truncates **salary/3** to 2 decimal places

SELECT empno, empname, TRUNC(salary/3, 2), manager from employees			
empno	**empname**	**TRUNC (salary/3, 2)**	**manager**
AAAA	Greg		AAAC
AAAB	Mathew	3333.33	AAAC
AAAC	John	3333.33	AAAD
AAAD	Barry	3333.33	
AAA_E	John Paul	1666.66	AAAD

4.5.2 ROUND

The **ROUND** function rounds a number to a certain amount of decimal places.

ROUND(column1/expression1, DEC)



Number of decimal places to round the number to.
This is the numeric value to round. This can be a column name or an expression.
Function call.

The argument passed in as **DEC** effects the outcome of the number in the following manner,

- If **DEC** is not specified then the default is zero.
- If **DEC** is negative the decimal place is shifted to the left.

Example 1

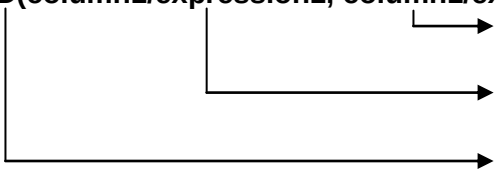
This example of the **ROUND** function rounds salary/3 to 2 decimal places

SELECT empno, empname, ROUND(salary/3, 2), manager from employees			
empno	**empname**	**ROUND (salary/3, 2)**	**manager**
AAAA	Greg		AAAC
AAAB	Mathew	3333.33	AAAC
AAAC	John	3333.33	AAAD
AAAD	Barry	3333.33	
AAA_E	John Paul	1666.67	AAAD

4.5.3 MOD

The **MOD** function divides two numbers and returns the remainder. The **MOD** function can be used to check if a number is odd or even.

MOD(column1/expression1, column1/expression1)



Number to divide. This can be a column name or an expression.
Number to divide by. This can be a column name or an expression.
Function call.

Example 1

This example of the **MOD** function divides salary by 3000 and returns the remainder

SELECT empno, empname, MOD(salary, 11), manager from employees			
empno	**empname**	**MOD (salary, 3000)**	**manager**
AAAA	Greg		AAAC
AAAB	Mathew	1000	AAAC
AAAC	John	1000	AAAD
AAAD	Barry	1000	
AAA_E	John Paul	2000	AAAD

4.6 DATE Functions

Date functions are used to perform arithmetic operations on date datatypes. ORACLE stores the date datatypes with the following characteristics,

- By default ORACLE stores dates using the below internal format,
  - Century 20
  - Year 03
  - Month 09
  - Day 30
  - Hour 7
  - Minute 11
  - Second 23
- All dates must be in the range of “Jan 1, 4712BC” to “Dec 31, 9999AD”
- ORACLE uses the default date display format of DD-MMM-RR. This means that when a date is displayed the century component of the date is dropped.
- DATE datatypes are stored as numbers. This means calculations can be performed on dates using standard arithmetic operators as per below,
  - DATE – DATE Calculate the number of days between two dates.
  - DATE + number Add x number of days to a date.
  - DATE + number/24 Add x number of hours to a date by dividing number of hours by 24.

4.6.1 SYSDATE

The **SYSDATE** function returns the current database date and time. The **SYSDATE** can be used in the **SELECT** clause and in expressions where the argument calls for a date datatype.

Example 1

The below example displays the current database date using the default date format of DD-MMM-RR

SELECT SYSDATE FROM DUAL
SYSDATE
30-SEP-03

Example 2

The below example displays the number of days since 01/01/01

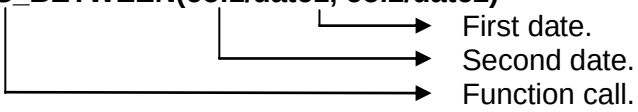
SELECT SYSDATE - TO_DATE('01-01-01', 'MM-DDD-RR') FROM DUAL
SYSDATE
1000

The use of the **TO_DATE** function and date formatting(‘MM-DDD-RR’) shall be discusses later in this chapter

4.6.2 MONTHS_BETWEEN

The **MONTHS_BETWEEN** function returns a numeric value which indicates the duration in months between two dates.

MONTHS_BETWEEN(col1/date1, col2/date2)



The result of the MONTHS_BETWEEN function has two segments. The first segment of the result which is to the left of the decimal point is the number of days between the two dates. The second segment of the result which is to the right of the decimal point indicates a portion of the reaming month.

Example 1

The below example displays the number of months between the two dates

SELECT MONTHS_BETWEEN('10-NOV-2000','10-OCT-2000') FROM DUAL
MONTHS_BETWEEN('10-NOV-2000','10-OCT-2000')
1

Example 2

The below example displays the number of months between the two dates

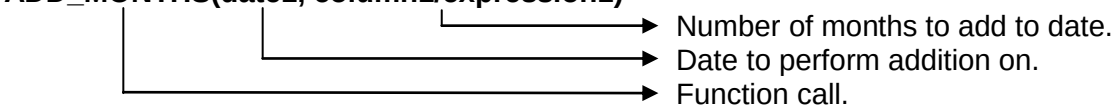
SELECT MONTHS_BETWEEN('10-NOV-2000','11-OCT-2000') FROM DUAL
MONTHS_BETWEEN('10-NOV-2000','11-OCT-2000')
.967741935

In the above example there is not quite one month between the two dates.

4.6.3 ADD_MONTHS

The **ADD_MONTHS** function returns a date value which is the result of an addition between a date and numeric value.

ADD_MONTHS(date1, column1/expression1)



Example 1

The below example displays a date two months in advance of 10-NOV-2000

SELECT ADD_MONTHS('10-NOV-2000',2) FROM DUAL
ADD_MONTHS('10-NOV-2000',2)
11-JAN-2001

Example 2

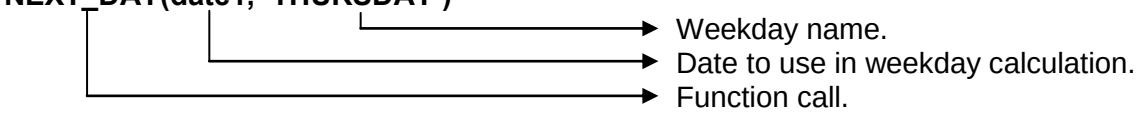
The below example displays a date one month prior to 10-NOV-2000

SELECT ADD_MONTHS('10-NOV-2000',-1) FROM DUAL
ADD_MONTHS('10-NOV-2000',-1)
10-OCT-2000

4.6.4 NEXT_DAY

The **NEXT_DAY** function returns the date for the next weekday occurrence.

NEXT_DAY(date1, 'THURSDAY')



Example 1

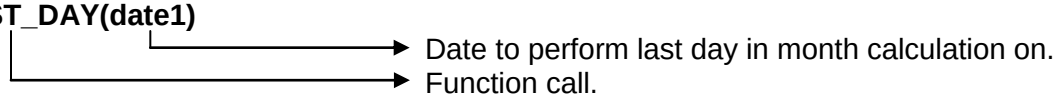
The below example displays the date of the next Thursday after 10-NOV-2000

SELECT NEXT_DAY('10-NOV-2000','THURSDAY') FROM DUAL
NEXT_DAY('10-NOV-2000','THURSDAY')
16-NOV-2000

4.6.5 LAST_DAY

The **LAST_DAY** function returns a date value indicating the last date in the month depending on the date value passed into the function

LAST_DAY(date1)



Example 1

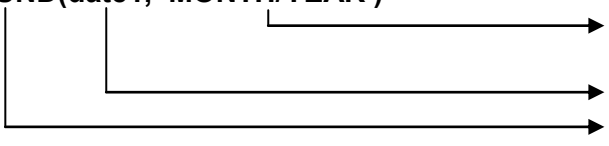
The below example displays the date of the next Thursday after 10-NOV-2000

SELECT LAST_DAY('10-NOV-2000') FROM DUAL
LAST_DAY('10-NOV-2000')
30-NOV-2000

4.6.6 ROUND and TRUNC with dates

The **ROUND** and **TRUNC** functions can be used for rounding and truncating dates to the nearest month or year.

ROUND(date1, 'MONTH/YEAR')



Example 1

The below example rounds the date to the closest month of the date passed to the **ROUND** function

SELECT ROUND('20-NOV-2000','MONTH') FROM DUAL
ROUND('20-NOV-2000','MONTH')
01-DEC-2000

Example 2

The below example truncates the date to the year of the date passed to the **TRUNC** function

SELECT TRUNC('20-NOV-2000','YEAR') FROM DUAL
TRUNC('20-NOV-2000','YEAR')
01-JAN-2000

4.7 Conversion Functions

When the Oracle server is processing SQL statements it expects to find certain data types in certain locations. If the data type found is not of the expected data type then the Oracle server attempts a conversion to the correct data type. This is called implicit data type conversion. The other type of data type conversion is called explicit data type conversion. This type of conversion is

If any data type conversion is unsuccessful the execution of the SQL statement results in an error. Otherwise, the SQL statement is carried out and the results are displayed as per normal.

4.7.1 Implicit Data Type Conversion

The data types used during implicit data type conversion are as follows,

During assignment

From	To
DATE, NUMBER	VARCHAR2
VARCHAR2, CHAR	DATE
VARCHAR2, CHAR	NUMBER

During expression evaluation

From	To
VARCHAR2, CHAR	DATE
VARCHAR2, CHAR	NUMBER

Example 1

The below example uses an implicit data type conversion

SELECT ROUND('20-NOV-2000','MONTH') FROM DUAL
ROUND('20-NOV-2000','MONTH')
01-DEC-2000

The string data type '20-NOV-2000' is converted to a date data type

4.7.2 Explicit Data Type Conversion

There are several conversion functions used for explicit data type conversions. Below are the most common three,

- TO_CHAR Convert data type to a CHAR.
- TO_NUMBER Convert data type to a NUMBER.
- TO_DATE Convert data type to a DATE.

These three conversion functions are explained in detail later in this chapter.

Example 1

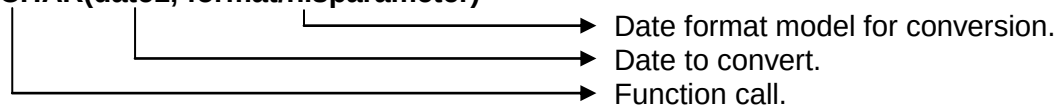
The below example uses an explicit data type conversion

SELECT TRUNC(TO_DATE('20-NOV-2000','DD-MM-YYYY'), 'YEAR') FROM DUAL
TRUNC(TO_DATE('20-NOV-2000','DD-MM-YYYY'), 'YEAR')
01-JAN-2000

4.7.3 TO_CHAR (With Dates)

The **TO_CHAR** function can be used to convert date data types into character data types. The main purpose for this type of conversion is for formatting and display purposes.

TO_CHAR(date1, format/nlsparameter)



Rules used during conversion,

- The date format parameter can be replaced with the **nlsparameter** (national language support). Each **nlsparamter** has a predefined date format.
- If the date format model or **nlsparameter** is omitted the default **nlsparameters** for the session are used.
- The date format model must be enclosed in single quotes.
- Literals in the time format model need to be enclosed in double quotes.
- The value inside the date format model is case sensitive.
- Day and month names in the output are automatically padded with spaces.

Date format model

Element	Description	Type
BC or AD	BC or AD indicator	Date
B.C or A.D	B.C or A.D indicator	Date
DAY	Name of day padded with spaces to a length of nine	Date
D or DD or DDD	Day of Weak, Month or Year.	Date
DY	Three letter abbreviation for the day of week	Date
fm	Fill mode. Remove leading spaces or suppress zeros	Date
I, IY, IYY,IYYY	One, two, three or four digit year	Date
RR	Century based on current year (0-49=20, 50-99=19)	Date
J	Julian day. Number of days since 31/12/4713 B.C	Date
MM	Month. Two digits	Date
MONTH	Name of month padded with spaces to a length of nine	Date
MON	Three character abbreviation for the month	Date
Q	Single digit representing quarter of the year	Date
RM	Month represented in Roman Numerals	Date
SCC or CC	Century. B.C indicated by a prefix of -	Date
sp	Spell out date	Date
SYEAR or YEAR	Year spelled out. B.C indicated by a prefix of -	Date
th	Use suffix of “th” on date	Date
WW or W	Week number of year or month	Date
Y,YYY	Year with comma included	Date
YYYY or SYYYY	Year. B.C indicated by a prefix of -	Date
Y or YY or YYY	Last one, two or three digits of year	Date

The RR date format is used to return the century based on the current two digit year. If the two digit year is between 0-49 then the 20^{th} century is returned. If the two digit year is between 50-99 then the 19^{th} century is returned.

Date format model (continued)

Element	Description	Type
AM or PM	Indicates meridian	Time
A.M. or P.M.	Indicates meridian with decimal point	Time
HH, HH12 or HH24	Hour of the day, or hour (1-12), or hour (0-23)	Time
MI	Minutes on the hour (0-59)	Time
SS	Seconds of the minute (0-59)	Time
SSSSS	Seconds of the day (0-86399)	Time
/ . , :	Punctuation used to separate time elements	Time
“of the”	Literal string used in time format model	Time

Example 1

The below example formats the system date into the following,

- MM Two digit month
- / Date Separator
- YY Two digit year

SELECT TO_CHAR(SYSDATE, ‘MM/YY’) FROM DUAL
TO_CHAR(SYSDATE, ‘MM/YY’)
12/00

Only the number of month and year are displayed.

Example 2

The below example formats the system date into the following,

- HH24 Hour of the day in 24 hour time format
- : Time Separator
- MI Minutes of the hour

SELECT TO_CHAR(SYSDATE, ‘HH24:MI’) FROM DUAL
TO_CHAR(SYSDATE, ‘HH24:MI’)
17:34

Only the time component of the date is displayed because only the time elements from the date format model are specified.

Example 3

The below example formats the system date into the following,

- DD Two digit day of month
- “th” Literal value “th”
- “of” Literal value “of”
- MONTH Name of month
- YYYY Four digit Year

SELECT TO_CHAR(SYSDATE, ‘DDth “of” MONTH YYYY’) FROM DUAL
TO_CHAR(SYSDATE, ‘DDth “of” MONTH YYYY’)
17th of Jan 2003

This example displays a full date including the literal words “th” and “of”

Example 4

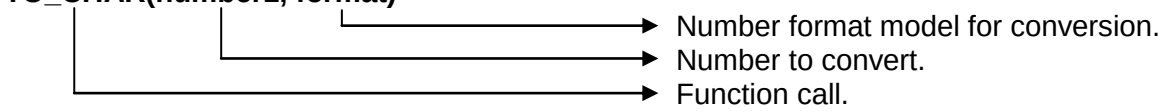
The below example uses the RR date format to return a certain century.

SELECT TO_CHAR('01-JAN-1995', 'DD-Mon-YYRR') FROM DUAL
TO_DATE(’01-JAN-95’, ‘DD-Mon-YYRR’)
01-Jan-95

4.7.4 TO_CHAR (With Numbers)

The **TO_CHAR** function is also used to convert number data types into character data types. The main purpose for this type of conversion is for formatting and displaying.

TO_CHAR(number1, format)



Rules used during conversion,

- If the decimal value will not fit in the format model then it is rounded up to match the format model.
- If a number is to big to fit in the format model then a sequence of "#" characters are displayed instead.

Number format model

Element	Description
9	Indicates a position where a number can be displayed
0	Displays zero if no digit is present
$	Display a dollar sign
L	Display a local currency symbol
U	Displays the dual currency symbol. E.g, Euro
.	Display a decimal point
,	Display a comma
MI	Display a minus sign to the right of the number
PR	Display parenthesize around number for negative values
EEEE	Display number in scientific notation
V	Multiply number by 10 times
B	Suppress zeros
X	Displays number in hexadecimal format

Example 1

The below example displays the value 12345 using the formats from the number model,

- $ Display dollar sign
- 9 Display number if required
- , Display comma at thousands separator
- . Display decimal point to 2 decimal places
- 0 Display zero if no digit is present

SELECT TO_CHAR(12345, '$999,999.00') FROM DUAL
TO_CHAR(12345, '$999,999.00')
$12,345.00

Example 2

The below example displays the value 12345 using the formats from the number model,

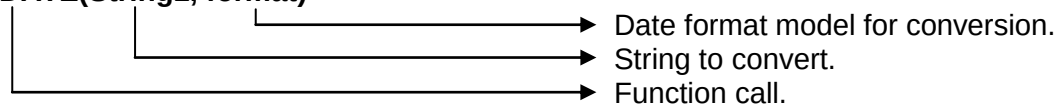
- MI Display a minus sign to the right of the number
- 9 Display number if required
- V Multiply number by 100

SELECT TO_CHAR(-12345, 'MI999999V9') FROM DUAL
TO_CHAR(-12345, 'MI9999999V99')
1234500-

4.7.5 TO_DATE

The **TO_DATE** function can be used to convert a string to a date. The format model used with this function is identical to that shown previously in this chapter.

TO_DATE(String1, format)



When using the **TO_DATE** function it is possible to introduce a strict method of conversion validation by using what is known as the format exact or fx" modifier. The "fx" modified tells the **TO_DATE** conversion function to perform exact character matching between the string and the format model.

Rules used with the "fx" modifier,

- The literal value in the string must match the corresponding format model.
- Additional blanks in the string are inhibited.
- Numeric literals must be the same length ad the format model. This means padding out the length of the string with zeros to match the length of the format model.

Rules used without the "fx" modifier,

- Any blanks in the string are ignored.
- Leading zeros can be omitted from numeric literals.

The Oracle Server will return an error if any of the above conditions are not meet.

Example 1

The below example converts the literal date string '01-NOV-03' to a date datatype

SELECT TO_DATE('01-NOV-03','DD-Mon-YYYY') FROM DUAL
TO_DATE('01-NOV-2003','DD-Mon-YYYY')
01-NOV-03

Example 2

The below example attempts to convert the literal string ' 1-NOV-03' to a data datatype.

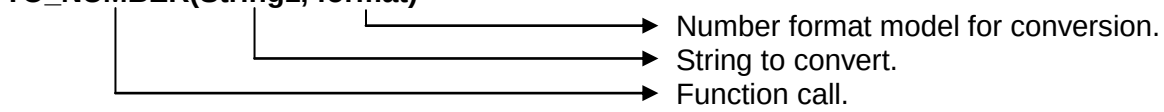
SELECT TO_DATE(' 1-NOV-2003','fxDD-Mon-YYYY') FROM DUAL
ERROR at line 1: ORA-01858: a non-numeric character was found where a numeric was expected

The datatype conversion fails because the literal date string contains a blank character at position 1.

4.7.6 TO_NUMBER

The **TO_NUMBER** function can be used to convert a string to a number. The format model used with this function is identical to that shown previously in this chapter.

TO_NUMBER(String1, format)



Example 1

The below example converts the literal number string '255' into its hexadecimal representation

SELECT TO_NUMBER('255', 'XX') FROM DUAL
TO_NUMBER('255', 'XX')
FF

Example 2

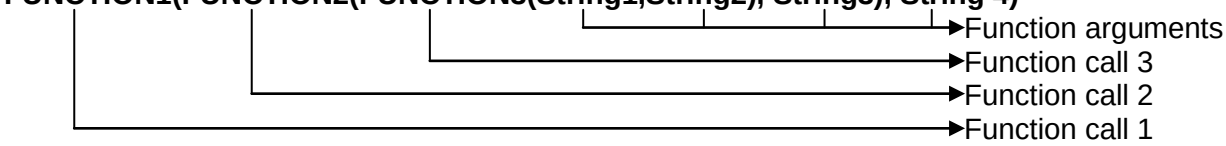
The below example converts the literal number string '$12,345.00' into a number datatype**.**

SELECT TO_NUMBER('$12,345', '$999,999.00') FROM DUAL
TO_NUMBER('$12,345.00', ''$999,999.00')
12345

4.8 Nesting Functions

Calling a function from within a function is knows as nesting functions. You can nest functions to any level you like but the function execution sequence always occurs from the deepest level upwards.

FUNCTION1(FUNCTION2(FUNCTION3(String1,String2), String3), String 4)



The nested functions above will execute in order of Funtion3, Function2 and last of all Function1. Notice that the results of the nested functions are being used as arguments for the next function call, this is the main objective of nesting functions.

Example 1

The below example converts the date datatype into a string and then converts it to uppercase.

SELECT UPPER(TO_CHAR('01-Nov-2003','DD-MON-YYYY)) FROM DUAL
UPPER(TO_CHAR('01-Nov-2003','DD-MON-YYYY))
01-NOV-03

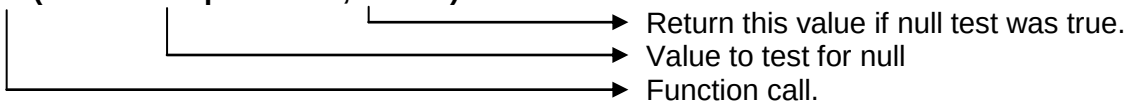
4.9 Generic Functions

This chapter has only looked at only a few of the hundreds of available SQL functions. This section will now discuss some of the more widely used generic functions.

4.9.1 NVL

The **NVL** function returns a value if a particular column or expression is null.

NVL(column1/expression1, value1)



Rules for **NVL**,

- If **coumn1/expression1** is null **NVL** returns **value1**
- If **column1/expression1** is not null **NVL** returns **column1/expression1**

The datatype of the **column1/expression1** can be of any datatype. However, the data that is returned in from **NVL** will always be of the same datatype as **column1/expression1**.
If the datatypes of **column1/expression1** and **value1** are different the **NVL** function will perform an implicit dattype conversion on **value1** to synchronize the datatypes.

Example 1

The below example returns zero because the salary column is null

SELECT NVL(salary, 0) FROM employees WHERE emp = 'AAAA'
NVL(salary, 0)
0

Example 2

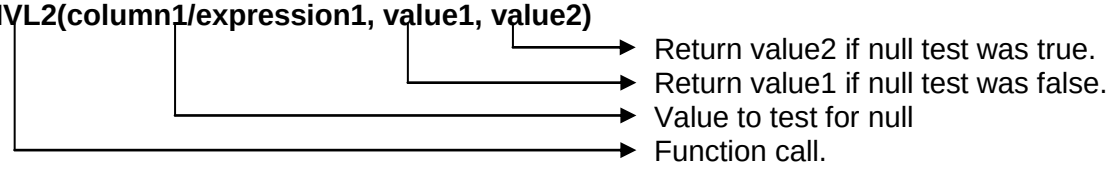
The below example returns 10000 because the salary column is not null

SELECT emp, NVL(salary, 0) FROM employees	
Emp	**NVL(salary, 0)**
AAAA	0
AAAB	10000
AAAC	10000
AAAD	10000
AAA_E	5000

4.9.2 NVL2

The **NVL2** function returns one of two values depending if a particular column or expression is null.

NVL2(column1/expression1, value1, value2)



Rules for **NVL2**,

- If **coumn1/expression1** is null **NVL2** returns **value1**
- If **column1/expression1** is not null **NVL2** returns **value2**

The datatype of the **column1/expression1** can be of any datatype. However, **value1** and **value2** must always be of the same datatype. If the datatypes of **value1** and **value2** are different the **NVL2** function will perform an implicit dattype conversion on **value2** to the same datatype as **value1**. The datatype returned by **NVL2** will be of the same datatype as **value1.** If **value1** was a string then a VARCHAR2 datatype will be returned.

Example 1

The below example test for a null value and returns either **sala** or **salb**

SELECT NVL2(salary, 'sala','salb') FROM employees WHERE emp = 'AAAA'
NVL2(salary, 'sala','salb')
Salb

Because the value tested is null **salb** is returned

Example 2

The below example tests for a null value and returns wither **sala** or **salb**

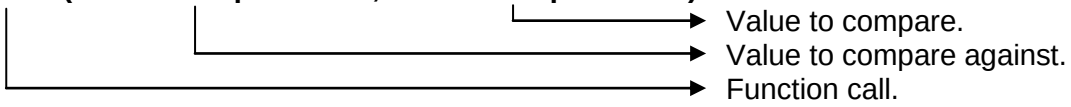
SELECT emp, NVL2(salary, 'sala','salb') FROM employees	
emp	**NVL2(salary, 'sala','salb')**
AAAA	salb
AAAB	sala
AAAC	sala
AAAD	sala
AAA_E	sala

According to the above results only employee **AAAA** has a null value in **salary**

4.9.3 NULLIF

The **NULLIF** function compare two values and returns a value(or null) depending on the results of the comparison.

NULLIF(column1/expression1, column2/expression2)



Rules for **NULLIF**,

- If **coumn1/expression1** is equal to **column2/expression2** then **NULLIF** returns null.
- If **coumn1/expression1** is not equal to **column2/expression2** then **NULLIF** returns **coumn1/expression1**.

Example 1

The below example compares salary against 10000.

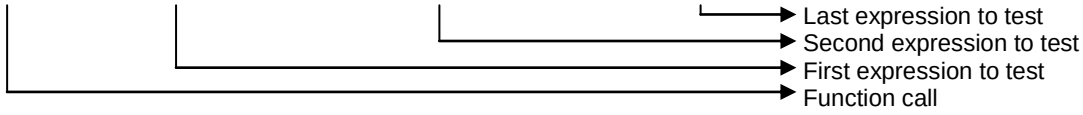
SELECT emp, NULLIF(salary, 10000) FROM employees	
emp	**emp NULLIF(salary, 10000)**
AAAA	
AAAB	
AAAC	
AAAD	
AAA_E	5000

The above results indicate that only employee **AAA_E** has a **salary** other than 10000. However, this is misleading because employee **AAAA** has a salary other than 10000.

4.9.4 COALESCE

The **COALESCE** function evaluates every expression in the argument list from left to right. The **COALESCE** function will return the first expression which does not evaluate to null.

COALESCE(column1/expression1, column2/expression2, column3/expression3)



Rules for **COALESCE**,

- **COALESCE** will return **column1/expression1** if it is not null.
- **COALESCE** will return **column2/expression2** if it is not null and **column1/expression1** is null.
- **COALESCE** will return **column3/expression3** if it is not null and **column1/expression1** and **column2/expression2** are null.

Example 1

The below example return the fist expression which evaluates to a non null value

SELECT COALESCE(salary,manager,emp) FROM employees WHERE emp = 'AAAA'
COALESCE(salary,manager,emp)
AAAC

4.10 Conditional Functions

Oracle SQL provides functionality for If-Then-Else logic which can be embedded into SQL statements. This logic can be used with the **CASE** and **DECODE** functions as per below.

4.10.1 CASE

The **CASE** function provides If-Then-Else logic which is embedded into a **SELECT** statement in the following format,

CASE column1/expression1 ⟶ This is the value to test
WHEN result1 RETURN value1 ⟶ Return **value1** if **column1/expression1** = **result1**
WHEN result2 RETURN value2 ⟶ Return **value2** if **column1/expression1** = **result2**
WHEN result3 RETURN value3 ⟶ Return **value3** if **column1/expression1** = **result3**
ELSE
RETURN value4 ⟶ Return **value4** if **nothing** = **column1/expression1**
END

Rules for the **CASE** function,

- The **CASE** function returns the first **value**# in the list where the **result#** equals **column1/expression1**.
- If none of **result#** equals **column1/expression1** then **value4** is returned.
- If the **CASE** function does not include an **ELSE** section and none of **value#** equals **column/expression1** then **CASE** returns null.

Example 1

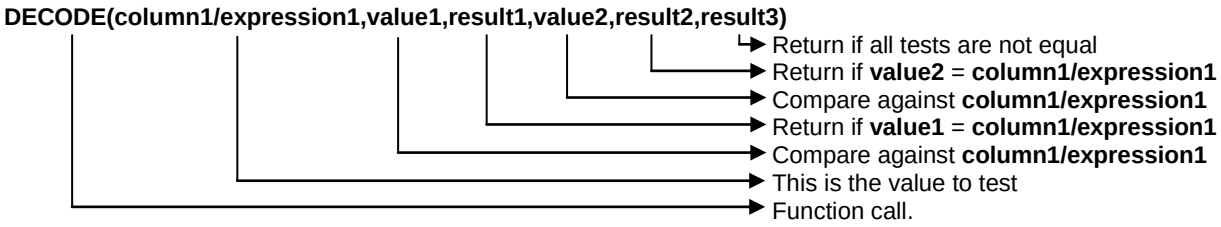
The below example displays a calculated value in **SALARY_BONUS** depending on the value in **SALARY**.

```
SELECT emp, salary
CASE salary
   WHEN 10000 RETURN salary + 1000
   WHEN 5000 RETURN salary + 2000
ELSE
   RETURN salary + 3000
END "SALARY_BONUS"
FROM employees
```

emp	salary	SALARY_BONUS
AAAA		3000
AAAB	10000	11000
AAAC	10000	11000
AAAD	10000	11000
AAA_E	5000	7000

4.10.2 DECODE

The **DECODE** function provides If-Then-Else logic which is embedded into a **SELECT** statement in the following format,



Rules for the **DECODE** function,

- **result3** will be retuned if **column1/expression1** does not equal **value1**,**value2**
- If **result3** is not included and none of the equality tests are true the **DECODE** returns null.
- **DECODE** will compare **column1/expression1** against **value1**,**value2** and return **result#**.
- The **value#** parameters are compared against **column1/expression1** from left to right.
- **DECODE** will return the first **value#** when a match is encountered.

Example 1

The below example displays a calculated value in **SALARY_BONUS** depending on the value in **SALARY**.

```
SELECT emp, salary
DECODE(salary, 10000, 11000, 5000, 7000, 3000) "SALARY_BONUS"
FROM employees
```

emp	salary	SALARY_BONUS
AAAA		3000
AAAB	10000	11000
AAAC	10000	11000
AAAD	10000	11000
AAA_E	5000	7000

Example 2

The below example displays **EMP_CATEGORY** based on the fourth character of the **emp** column

```
SELECT emp, salary
DECODE(substr(emp,4,1), 'A', 'Class A', 'B', 'Class B', 'C', 'Class C', 'D', 'Class D', 'Unknown')
"EMP_CATEGORY"
FROM employees
```

emp	salary	EMP_CATEGORY
AAAA		Class A
AAAB	10000	Class B
AAAC	10000	Class C
AAAD	10000	Class D
AAA_E	5000	Unknown

4.11 Summary

- Single row functions are used for data calculation, manipulation, formatting and data type conversion.
- Single row functions are categorized into the following groups,
  - Character functions
  - Number functions
  - Date functions
  - Datetype conversion functions
  - Generic functions
- Character functions are used for character/string manipulation.
- Number functions are used for arithmetic operations on numeric values.
- Date functions are used for date arithmetic operations of date values.
- Datatype conversion functions are used to convert one datatype to another.
- Conditional functions are used when if-then-else logic is required in am SQL expression.

4.12 Exercises

A. Display all the records of the **employees** table with the **empname** column displayed in uppercase.
B. Display all the records of the **employees** table with the **emp** and **manager** columns concatenated together.
C. Display all the records of the **employees** table with the **salary** column divided by three and rounded to two decimal places.
D. Display the string '17th November 2003 23:59.00' as a date datatype formatted to17-NOV-2003'.
E. What is the total amount of nested functions permitted?
F. Display the date '17-NOV-2003' as a char datatype formatted to '17th November 2003'.
G. Display the number 12882.4598 as a char datatype formatted to '$12,882.46'.
H. Display all the records of the **employees** table with 20% added to the **salary** field. If the **salary** field is null then display the string 'No Salary'.
I. Use the **COALESCE** function to display the salary and manager fields in the one column.
J. Use the **DECODE** function to display the following,

Field	Condition	Display Value
emp	none	emp
salary	Salary = 1000	"10 Gs"
	Salary = 5000	"5 Gs"
	else	"Zero"

K. Use the **CASE** function to display the same result from question I.
L. What is an explicit datatype conversion?
M. What is an implicit datatype conversion?

5. Handling Multiple Tables

5.1 Objectives

After this chapter you should be able to perform the following,

- Retrieve data from multiple tables using the **SELECT** statement.
- Understand and use all type of joins.

5.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

TABLE : customer			
name	**postcode**	**gender**	**age**
Smith	2232	M	35
Jones	2232	F	21
Simons	2199	M	50
Thirley	2245	F	56
Ravenwood	2245	M	28
Andrews	2345	M	43

TABLE : suburb			
name	**postcode**	**population**	**statecode**
Sutherland	2232	20001	NSW
Begsville	2245	1903	NSW
Nadenham	2199	12042	QLD
Grethem	3000	45322	VIC

TABLE : state			
statecode	**state**	**capital**	**sales**
NSW	New South Wales	Sydney	100009
VIC	Victoria	Melbourne	299009
WA	Western Australia	Perth	6700

TABLE : sales_cat		
cat	**from_sale**	**to_sale**
Low	0	100000
Medium	100001	200000
High	200000	999999

TABLE : employees			
code	**name**	**position**	**manager_code**
1000	Jeff	Sales Rep	2000
1001	Martin	Sales Rep	2000
2000	Darren	Sales Manager	3000
3000	Greg	General Manger	

5.3 Using Multiple Tables

Oracle allows you to combine the data from multiple tables in the SELECT statement. The join between tables needs to be established by linking one or more columns from one table to another.

The diagram below illustrates a join of data between two tables giving the combined results in a single row,

TABLE : customer	
name	**Postcode**
Smith	2232
Jones	2232
Simons	2199
Thirley	2245
Ravenwood	2245
Andrews	2345

TABLE : suburb	
postcode	**population**
2232	20001
3000	1903
2199	12042
2245	45322

customer.name	customer.postcode	suburb.postcode	suburb.population
Ravenwood	2245	2245	45322
Thirley	2245	2245	45322
Simons	2199	2199	12042
Jones	2232	2232	20001
Smith	2232	2232	20001

When data form multiple tables is required a join must be established. A join links columns from one table to another for purpose of returning related information from both tables. The above results show that the join between the two columns is based on the postcode field. Note that the information for Andrews is not displayed because no join between the two tables could be established based on the postcode. To join the two tables we need to specify a join condition in the **WHERE** clause of the **SELECT** statement.

For example, **WHERE customer.postcode = suburb.postocde**

You should always ensure that the join between your tables is present and correct. Any incorrect or omitted joins will result in what is known as a Cartesian product. A Cartesian product will join every row from every table specified in the **SELECT** clause resulting is an excessive and useless amount of rows being displayed. A Cartesian product on the above example would link every row from the customer table(6 rows) to the suburb table(4 rows) resulting in 24 rows being displayed. To avoid the Cartesian product ensure that the join condition in your **WHERE** clause is present and correct.

Note :

Unless there is a specific requirement to generate a large number for rows from a Cartesian product you should always ensure that your join condition is correct. However, there may be particular circumstances when there is a requirement for generating a Cartesian product, for example, generating a large number of rows for test data.

5.4 Join Syntax And Rules

When joining data from multiple tables there is various rules and syntax regulations to follow as per below,

Syntax Example 1

```
SELECT customer.name, customer.postcode, suburb.postcode, suburb.population
FROM customer, suburb
WHERE customer.postcode = suburb.postcode
```

Syntax Example 2

```
SELECT c.name, c.postcode, s.postcode, s.population
FROM customer c, suburb s
WHERE c.postcode = s.postcode AND
s.population  > 25000
```

This is the join condition.
These are the tables to retrieve data from (These tables need to be joined by a common column).
These are the columns to display.

Join Rules

SELECT

- The **SELECT** clause identifies which columns from which table will be displayed.
- Column should also be prefixed to make the SQL expression more legible.
- As per normal **SELECT** statement execution, the **SELECT** clause includes all the columns to be displayed. If the tables used in the join contain columns with identical names you must prefix the column with the table name in the format of <**table_name**>.<**column_name**> as per syntax example 1.
- Prefixing tables improves performance by providing the Oracle server with the information about which table to find the column in.

FROM

- The **FROM** clause identifies which tables will be used in the join.
- Tables can be allocated an alias in the **FROM** clause as per syntax example 2.
- Table aliases must be no longer 30 characters.
- Once a table alias is defined it must be used to represent that table within the entire SQL expressions.
- The table alias is only valid during the SQL expression in which it was defined.

WHERE

- To avoid Cartesian products be sure to include the correct number of joins in the **WHERE** clause.
- The minimum number of joins to include in the **WHERE** clause is the number of tables in the **SELECT** clause minus one.
- As per normal the **WHERE** clause can include the **AND** operator. This places further constrictions on the data selected.

5.5 Join Types

The join types available in Oracle 9i are compliant with the ANSI SQL:1999 standard. There are several types of joins available as per below,

Oracle join name	Description
Equijoin	Columns in tables are joined by the equality operator
Non-equijoin	Columns in tables are joined by an operator other than the equality operator
Outer Join	Columns that do not match the join are also displayed
Self Join	Columns in a table are joined to other columns in the same table

5.5.1 Equijoin

An equijoin will join two tables based on one common column from each table. In order to make a join from one table to another the value in the common columns must be identical.

SELECT tab1.col1, tab2.col1, tab2.col2 → Select columns to display
FROM tab1, tab2 → Choose tables to retrieve data from
WHERE tab1.col1 = tab2.col1 → Establish the equijoin (=)

Example 1

The below example joins the suburb and state tables based on the statecode.

```
SELECT su.suburb,su.statecode,st.sales
FROM suburb su, state st
WHERE su.statecode = st.statecode
```

su.suburb	su.statecode	st.sales
Sutherland	NSW	100009
Begsville	NSW	100009
Grethem	VIC	299009

SELECT c.suburb,c.statecode,s.sales

- The **SELECT** clause in Example 1 is selecting two columns from the **suburb** table and one column from the **state** table.

FROM suburb su, state st

- The tables used are **suburb** and **state** which have been allocated an alias of **su** and **st**.

WHERE su.statecode = st.statecode

- The join condition in the **WHERE** clause links the two tables on the common column **statecode**.
- Every join that is made from the **suburb** to the **state** table has a row displayed.
- No data is displayed for **state** values of "**WA**" or "**QLD**" because no join between the common columns could be established.

Example 2

The below example is identical to Example 1 but has another join and another selection constraint.

```
SELECT cu.name, su.suburb,su.statecode,st.sales
FROM customer cu, suburb su, state st
WHERE  cu.postcode = su.postcode AND
su.statecode = st.statecode AND
cu.name = 'Smith'
```

cu.name	su.suburb	st.sales
Smith	NSW	100009

This example actually used three tables so we need two join conditions.

5.5.2 Non-equijoin

A non-equijoin is the opposite to an equijoin. In order to make a join from one table to another the value in the two joining columns must not be linked with the equality(=) operator.

SELECT tab1.col1, tab1.col2, tab2.col1 → Select columns to display
FROM tab1, tab2 → Choose tables to retrieve data from
WHERE tab1.col2 between tab2.col2 and tab2.col3 → Establish the non-equijoin (<>)

Example 1

The below example joins the state and sales_cat tables based on the sales column. The sales column in the state table needs to be between a certain range in the sales_cat table to establish a join between the two tables.

SELECT st.statecode, sc.cat FROM state st, sales_cat sc WHERE st.sales BETWEEN sc.from_sales AND sc.to_sales	
st.statecode	**sc.cat**
NSW	100009
VIC	299009
WA	6700

Example 2

This example is the same login as Example 1 but uses different operators for the join condition.

SELECT st.statecode, sc.cat FROM state st, sales_cat sc WHERE st.sales >= sc.from_sales AND st.sales <= sc.to_sales	
st.statecode	**sc.cat**
NSW	Medium
VIC	High
WA	Low

Guidelines for using Non-equijoins

- When using the **BETWEEN** operator make sure there are no from/to ranges that overlap each other. This is for data integrity purposes.
- When using the **BETWEEN** operator follow the below guidelines,
  - The value should not be less that the lowest from value.
  - The value should not be higher than the highest to value.

  This is for data integrity purposes.
- When using the **BETWEEN** operator specify the range in the order of lowest then highest.
- Other operators such as >=, >, <, <= are available.

5.5.3 Outer Joins

Data is only returned from a join query if it meets the criteria of the join. An outer join is used to return data that does not satisfy the join condition in the **WHERE** clause. An outer join is established with the use of the plus (+) sign.

SELECT tab1.col1, tab2.col1, tab2.col2 → Select columns to display
FROM tab1, tab2 → Choose tables to retrieve data from
WHERE tab1.col1(+) = tab2.col1 → Establish the outer join (+)

Example 1

The below example joins the suburb and state tables based on the statecode.

SELECT su.suburb,st.statecode **FROM suburb su, state st** **WHERE su.statecode(+) = st.statecode**	
su.suburb	**su.statecode**
Sutherland	NSW
Begsville	NSW
Nadenham	
Grethem	VIC

The (+) sign is placed on the state table because there is no corresponding **QLD** value in the state table. This forces the left half of the join condition to be displayed without the right portion.

Example 2

The below example joins the suburb and state tables based on the statecode.

SELECT su.suburb,st.statecode **FROM suburb su, state st** **WHERE su.statecode(+) = st.statecode(+)**	
su.suburb	**su.statecode**
Sutherland	NSW
Begsville	NSW
Nadenham	
Grethem	VIC
	WA

The (+) sign is placed on both tables in order to display all columns which are missing the data required for the link.

Guidelines for using outer joins

- The plus (+) sign is placed on the side of the join condition that is missing the data.
- The plus (+) sign can be placed on only one side of the join condition.
- The outer join cannot include the **IN** operator.
- Outer joins conditions cannot be linked to other conditions in the **WHERE** clause by the **OR** operator.

5.5.4 Self Joins

Self joins are used to link one table to itself within the **WHERE** clause.

SELECT tab1.col1, tab2.col1 → Select columns to display
FROM table tab1, table tab2 → Choose the table to retrieve data from
WHERE tab1.col1 = tab2.col1 → Establish the self join

Note that in the syntax provided above the table aliases **tab1** and **tab2** would refer to the same table.

Example 1

The below example joins the sales_rep table to itself to obtain a list of employees and their managers.

SELECT T1.code, t1.name, t2.name
FROM employees t1, employees t2
WHERE t1.manager = t2.code

t1.code	t1.name	t2.name
1000	Jeff	Darren
1001	Martin	Darren
2000	Darren	Greg

5.5.5 Cross Joins

The cross join adheres to the **1999: SQL Syntax**. It is the equivalent of performing a join that produces a Cartesian product.

SELECT col1, col2 → Select columns to display
FROM tab1 → Choose the first table in the cross join
CROSS JOIN tab2 → Choose the second table in the cross join

Example 1

The below example creates a cross join (Cartesian product) between the suburb and state tables.

SELECT name, statecode, state
FROM suburb
CROSS JOIN state

name	statecode	State
Sutherland	NSW	New South Wales
Begsville	NSW	New South Wales
Nadenham	NSW	New South Wales
Grethem	NSW	New South Wales
Sutherland	VIC	Victoria
Begsville	VIC	Victoria
Nadenham	VIC	Victoria
Grethem	VIC	Victoria
Sutherland	WA	Western Australia
Begsville	WA	Western Australia
Nadenham	WA	Western Australia
Grethem	WA	Western Australia

5.5.6 Natural Joins

The natural join adheres to the **1999: SQL Syntax**. It is the equivalent of performing a normal equijoin. No columns names need to be specified with the natural join, all column names that are identical between the two tables are automatically joined.

SELECT col1, col2 → Select columns to display
FROM tab1 → Choose the first table in the cross join
CROSS JOIN tab2 → Choose the second table in the cross join

Join Rules

- All column names that are identical between the join (columns that form the join) must be of the same datatype or the query will result in an error.
- The **USING** clause can be used in the natural join under the following conditions,
  - When the datatypes between two identical columns are different.
  - To join the two tables based on only one column.
- If the **USING** clause is used then column names cannot be used with table aliases.

Example 1

The below example creates a natural join between the suburb and state tables.

```
SELECT name, statecode, state
FROM suburb
NATURAL JOIN state
```

name	statecode	State
Sutherland	NSW	New South Wales
Begsville	NSW	New South Wales
Nadenham	QLD	Queensland
Grethem	VIC	Victoria

Example 2

The below example creates a natural join between the suburb and state tables and includes further constraints by using a **WHERE** clause.

```
SELECT name, statecode, state
FROM suburb
NATURAL JOIN state
WHERE statecode = "NSW"
```

name	statecode	State
Sutherland	NSW	New South Wales
Begsville	NSW	New South Wales

Example 3

This example creates a join on the customer and suburb tables based only on the postcode columns by including the **USING** clause.

```
SELECT gender, postcode, population
FROM customer
JOIN suburb
USING postcode
```

gender	statecode	State
M	NSW	New South Wales
F	NSW	New South Wales
M	QLD	Queensland
F	NSW	New South Wales
M	QLD	Queensland

5.5.7 The ON Clause

The **ON** clause adheres to the **1999: SQL Syntax**. It is used to join tables based on columns that have different names. The **ON** clause is useful for joining tables to themselves (Self Join).

SELECT tab1.col1, tab2.col1 → Select columns to display
FROM table tab1 JOIN table tab2 → Choose the table to retrieve data from
ON (tab1.col1 = tab2.col1) → Establish the self join

Join Rules

- If the **ON** clause is not used then the SQL expression will join the tables based on identical column names between the tables. This is the equivalent of a natural join.
- The **ON** clause can be used in conjunction with the **WHERE** clause to add further constrains.
- Further constrains can be included in the **WHERE** condition by the use of the **AND** clause.

Example 1

The below example creates a self join on the employees table.

```
SELECT t1.code, t1.name, t2.name
FROM employees t1, employees t2
ON (t1.manager = t2.code)
```

t1.code	t1.name	t2.name
1000	Jeff	Darren
1001	Martin	Darren
2000	Darren	Greg

Example 2

The below example creates a natural join between the suburb and state tables and also includes further data constraints by using the **WHERE** clause.

```
SELECT t1.name, t2.statecode, t2.state
FROM suburb t1, state t2
ON (t1.statecode = t2.statecode)
WHERE statecode = "NSW"
```

name	Statecode	State
Sutherland	NSW	New South Wales
Begsville	NSW	New South Wales

5.5.8 Three Way Joins

The **SQL: 1999 compliant syntax** permits the join of three tables using a three way join. This is the equivalent of a three-way equijoin.

SELECT tab1.col1, tab2.col1, tab3.col1 → Select columns to display
FROM table tab1 → Choose the first table
JOIN table tab2 → Choose the second table
ON (tab1.col1 = tab2.col1) → Join the first table to the second table
JOIN table tab3 → Choose the third table
ON (tab2.col1 = tab3.col1) → Join the second table to the third table

Three Way Join Rules

- The sequence of the joins must be performed from left to right as per below,
  Correct

  Incorrect

- The first join condition only has scope to which tables have been declared so far in the SQL expression. This means the join condition only has scope of the first two tables.
- The second join condition has scope to all tables that have declared so far. This means it has scope of all tables.

Example 1

The below example creates a three-way join on the customer, suburb and state tables.

```
SELECT cu.name, su.statecode,st.sales
FROM customer cu
JOIN suburb su
ON (cu.postcode = su.postcode)
JOIN state st
ON (su.statecode = st.statecode)
```

cu.name	su.statecode	st.sales
Smith	N	100009
Jones	N	100009
Thirley	N	100009
Ravenwood	N	100009

The same example in provided in section **5.5.1 Equijoin** under **Example 2** as a three way equijoin.

5.5.9 Left Outer Join

The left outer join adheres to the **1999: SQL Syntax**. It is the equivalent of the outer join with the plus (+) sign on the left hand side of the equal sign.

SELECT tab1.col1, tab2.col1, tab2.col2 → Select columns to display
FROM table tab1 → Choose the left table
LEFT OUTER JOIN table2 → Choose the right table
ON (tab1.col1 = tab2.col1) → Establish the left outer join condition

Example 1

The below example joins the **customer** and **suburb** tables based on the **postcode**. All rows from the **customer** table are returned even if there is no match with the **suburb** table.

```
SELECT cu.name, su.postcode
FROM customer cu
LEFT OUTER JOIN suburb su
ON (cu.postcode = su.postcode)
```

cu.name	su.postcode
Smith	2232
Jones	2232
Simons	2199
Thirley	2245
Ravenwood	2245
Andrews	

The above query is the equivalent of placing the plus (+) sign next to the **cu.postcode** column.

5.5.10 Right Outer Join

The right outer join adheres to the **1999: SQL Syntax**. It is the equivalent of the outer join with the plus (+) sign on the right hand side of the equal sign.

SELECT tab1.col1, tab2.col1, tab2.col2 → Select columns to display
FROM table tab1 → Choose the left table
RIGHT OUTER JOIN table2 → Choose the right table
ON (tab1.col1 = tab2.col1) → Establish the right outer join condition

Example 1

The below example joins the **customer** and **suburb** tables based on the **postcode**. All rows from the **suburb** table are returned even if there is no match with the **customer** table.

```
SELECT cu.name, su.postcode
FROM customer cu
RIGHT OUTER JOIN suburb su
ON (cu.postcode = su.postcode)
```

cu.name	su.postcode
Smith	2232
Jones	2232
Simons	2199
Thirley	2245
Ravenwood	2245
	3000

The above query is the equivalent of placing the plus (+) sign next to the **su.postcode** column.

5.5.11 Full Outer Join

The full outer join adheres to the **1999: SQL Syntax**. It is the equivalent of the outer join with the plus (+) sign on both signs of the equal sign.

SELECT tab1.col1, tab2.col1, tab2.col2 → Select columns to display
FROM table tab1 → Choose the left table
FULL OUTER JOIN table2 → Choose the right table
ON (tab1.col1 = tab2.col1) → Establish the full outer join condition

Example 1

The below example joins the **customer** and **suburb** tables based on the **postcode**. All rows from both tables are retuned even if there is no match across the tables.

```
SELECT cu.name, su.postcode
FROM customer cu
RIGHT OUTER JOIN suburb su
ON (cu.postcode = su.postcode)
```

cu.name	su.postcode
Smith	2232
Jones	2232
Simons	2199
Thirley	2245
Ravenwood	2245
Andrews	
	3000

The above query is the equivalent of placing the plus (+) sign next to both the **cu.postcode** and **su.postcode** columns.

5.6 Summary

- Oracle allows you to combine the data from multiple tables using the following join types,

Join Classification	
SQL 1999: Compliant Syntax	**Oracle Syntax**
Cross Join	Cartesian Product
Natural Join	Equijoin
Join USING	Non-Equijoin
Left Outer Join	(+)Outer Join
Right Outer Join	Outer Join(+)
Full Outer Join	(+)Outer Join(+)
Join ON	Self Join

- Cartesian products are created when an invalid join condition is established or if the join condition is non existent.
- Equijoins are used to join tables based on equal values between common columns.
- Non-Equijoins are used to join tables based on non-equal values between common columns.
- Outer Joins are used to display rows from a table that do not meet the join condition.
- Self Joins are used to join a table to itself.

5.7 Exercises

A. Build a join SQL expression as per the below requirements,

Join Type	Table names	Table Alias	Columns to Join	Columns to Display
Equijoin	customer, suburb	cu, su	postcode	Cu.name, su.postcode

B. Build a join SQL expression as per the below requirements,

Join Type	Table names	Table Alias	Columns to Join	Columns to Display
Equijoin	sustomer, suburb	cu, su	postcode	cu.name, su.postcode
Equijoin	Suburb, State	Su, st	statecode	st.statecode

C. Build a join SQL expression as per the below requirements,

Join Type	Table names	Table Alias	Columns to Join	Columns to Display
Self join	employees	emp	code, manager_ code	code, name, manager_code

D. Build a join SQL expression as per the below requirements,

Join Type	Table names	Table Alias	Columns to Join	Columns to Display
Outer Join	customer, suburb	cu, su	(+)postcode	cu.name, cu.age, su.postcode
Equijoin	suburb, state	su,st	statecode	st.state

E. Build a join SQL expression as per the below requirements,

Join Type	Table names	Table Alias	Columns to Join	Columns to Display
Outer Join	customer, suburb	cu, su	(+)postcode	cu.name, cu.age, su.postcode
Equijoin	suburb, state	su,st	statecode	state
Non-equijoin	state, sales_cat	st,sl	st.sales (sl.from_sale - sl.to_sale)	sl.cat

F. Build a join SQL expression as per the below requirements,

Join Type	Table names	Table Alias	Columns to Join	Columns to Display
Left Outer Join	customer, suburb	cu, su	(+)postcode	cu.name, cu.age, su.postcode

6. Group Functions

6.1 Objectives

After this chapter you should be able to perform the following,

- Understand the use of group functions
- Understand and use all the common available group functions.
- Understand and user the GROUP BY clause.
- Understand and user the HAVING clause.

6.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

TABLE : customer			
name	**postcode**	**gender**	**age**
Smith	2232	M	35
Jones	2232	F	21
Simons	2199	M	50
Thirley	2245	F	56
Ravenwood	2245	M	28
Andrews	2345	M	43

TABLE : employees			
code	**name**	**position**	**manager_code**
1000	Jeff	Sales Rep	2000
1001	Martin	Sales Rep	2000
2000	Darren	Sales Manager	3000
3000	Greg	General Manger	

6.3 Using Group Functions

Oracle provides the functionality to categorize the data retuned from SQL expressions into particular groups. Data can be grouped multiple ways as per below,

- Single table placed into a group.
- Subset of a single table placed into a group.
- Multiple tables placed into a group.
- Subset of multiple tables placed into a group.

Data is grouped on a specific selection criteria which is defined in an SQL expression. Once data is grouped into the necessary requirements, certain group functions are available to perform operations/calculations on that group.

Example

The below example is a diagram of how a single table can create a group of users and then find an average value of that group.

TABLE : customer	
name	**age**
Smith	35
Jones	21
Simons	50
Thirley	56
Ravenwood	28
Andrews	43

Group	
name	**age**
Smith	35
Simons	50

Group Result	
Average Age	42.5

6.4 Group Function Syntax and Rules

When using group functions there is various rules and syntax regulations to follow as per below,

Syntax Example 1

```
SELECT group function(column1)
FROM customer, suburb
WHERE condition
GROUP BY column1
```

- The **GROUP BY** clause is required when categorizing data into groups.
- WHERE clause condition **.**
- Table(s) to retrieve data from.
- **SELECT** clause identifies the columns and group functions to use.

Group Rules

- The keyword **DISTINCT** can be used to exclude duplicate records from groups. By default **ALL** duplicate records will be included in the group.
- Null values are ignored when passed into group functions.
- Sorting is performed automatically when the **GROUP BY** clause is used. The default sort order is ascending. Use the syntax **ORDER BY DESC** to override the default.
- Only CHAR, VARCHAR2, NUMBER and DATE data types can be used when passing an expression to a group function.

6.5 Common Group Functions

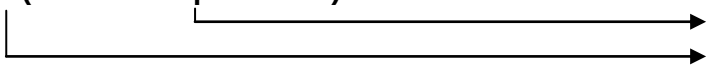
Oracle provides various group functions that are used within SQL expressions. Below is a list of some of the most commonly used group functions.

Group Functions		
Function	**Description**	**Argument Data Type**
AVG	Provides the average value of a group of values	Numeric
SUM	Provide the sum of a group of values	Numeric
MIN	Provides the smallest value in a group of values	Any
MAX	Provides the largest value in a group of values	Any
COUNT	Provides the number of rows in a group of values	Any

6.5.1 AVG

The **AVG** function returns the average value of a group of values.

AVG(column1/expression1)



Example 1

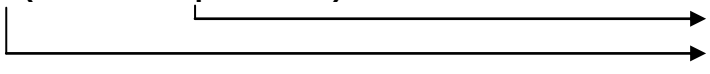
The below example calculates the average **age** for all records in the **customers** table. Records are grouped based on the **postcode**.

SELECT postcode, AVG(age) **FROM customers** **GROUP BY postcode**	
postcode	**AVG(age)**
2199	50
2232	28
2245	42
2345	43

6.5.2 SUM

The **SUM** function returns the addition of a group of values.

SUM(column1/expression1)



Example 1

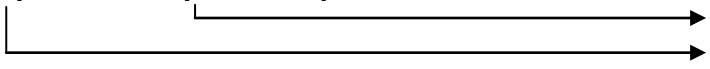
The below example calculates adds the **age** columns together for all records in the **customers** table. Records are grouped based on the **postcode**.

SELECT postcode, AVG(age) **FROM customers** **GROUP BY postcode**	
postcode	**AVG(age)**
2199	50
2232	50
2245	84
2345	43

6.5.3 MIN

The **MIN** function returns the lowest value in a group of values.

MIN(column1/expression1)



Example 1

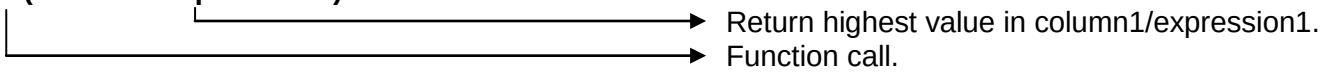
The below example returns the lowest **age** for all records in the **customers** table. Records are grouped based on the **postcode**.

SELECT postcode, MIN(age) FROM customers GROUP BY postcode	
Postcode	**AVG(age)**
2199	50
2232	21
2245	28
2345	43

6.5.4 MAX

The **MAX** function returns the highest value in a group of values.

MAX(column1/expression1)



Example 1

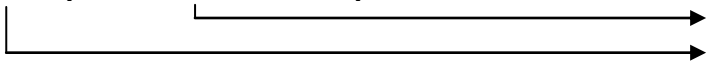
The below example returns the highest **age** for all records in the **customers** table. Records are grouped based on the **postcode**.

SELECT postcode, MAX(age) FROM customers GROUP BY postcode	
postcode	**AVG(age)**
2199	50
2232	35
2245	56
2345	43

6.5.5 COUNT

The **COUNT** group function will return the number of rows in a table.

COUNT(DISTINCT */column1)



Example 1

The below example returns the number of rows in the **employees** table.

```
SELECT COUNT(*)
FROM employees
```

COUNT(*)
4

Example 2

The below example returns the number of rows in the **employees** table which meet the **SELECT** criteria.

```
SELECT COUNT(*)
FROM employees
WHERE position = 'Sales Rep'
```

COUNT(*)
2

Example 3

The below example returns the number of rows in the **employees** table based on the **manager_code** column.

```
SELECT COUNT(manager_code)
FROM employees
```

COUNT(*)
3

Only three rows are counted because any row that has a null in the **manager_code** column is ignored.

Example4

The below example returns the number of rows in the **employees** table based on the **manager_code** column.

```
SELECT COUNT(DISTINCT manager_code)
FROM employees
```

COUNT(*)
2

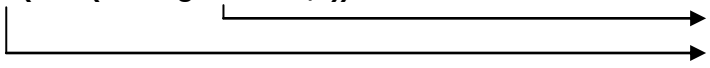
Only two rows are counted due to the following,

- Any row that has a null in the **manager_code** column is ignored.
- The **DISTINCT** keyword ignores duplicates in the **manager_code** column.

6.6 NVL and Group Functions

When performing group functions all null values are ignored. For example, if an attempt was made to calculate the average value on a particular column then any row that had a null in that column would be excluded from the average calculation. In order to include null values in a group function we simply nest the NVL function within the group function as per below,

AVG(NVL(manager_code,0))



Example 1

The below example calculates the average on the **manager_code** columns and excludes nulls from the calculation.

SELECT AVG(manager_code) **FROM employees**
COUNT(*)
2333.33

Example 2

However, if we nest the **NVL** function to replace nulls with zeros we receive a different result.

SELECT AVG(NVL(manager_code,0)) **FROM employees**
COUNT(*)
1750

Group Nesting Rules

- The above examples make reference to nesting group functions. Oracle will only accept a limit of two levels for group functions.

6.7 The GROUP BY Clause

Previously in this chapter we introduced the use of the **GROUP BY** clause. The **GROUP BY** clause will group table information together based on a value within a particular column.

SELECT column1, group function(column2)
FROM table1
GROUP BY column1

- The **GROUP BY** clause specifies which columns the data should be grouped into.
- Table(s) to retrieve data from.
- **SELECT** clause identifies the columns to display and what group function to use.

The syntax shown above indicates that the data will be grouped into the distinct values found in column1 and the group function(column2) applied to each of those groups.

Example 1

The below example groups the table rows based on **postcode** and calculates the average **age** for that group of customers.

```
SELECT postcode, AVG(age)
FROM customer
GROUP BY postcode
```

postcode	AVG(age)
2199	50
2232	28
2245	84
2345	43

Example 2

The below example groups the table rows based on **postcode** and calculates the average **age** for that group of customers. Note that this example does not display the columns used in the **GROUP BY** clause.

```
SELECT AVG(age)
FROM customer
GROUP BY postcode
```

AVG(age)
50
50
84
43

Example 3

The below example groups the table rows based on **postcode** and **gender**. Once the groups have been established the average **age** calculation is performed for each group.

```
SELECT postcode, gender, AVG(age)
FROM customer
GROUP BY postcode, gender
```

postcode	gender	AVG(age)
2232	M	35
2232	F	21
2199	M	50
2245	F	56
2245	M	28
2345	M	43

Because each group only contains a single row the average calculation is ineffective.

GROUP BY clause rules

- When the **GROUP BY** clause is used all columns not using a group function must be specified in the **GROUP BY** clause. If the non-function column is not specified then the error “not a single-group group function” will be displayed.
- Rows can be excluded from the group by the use of the WHERE clause.
- The **GROUP BY** clause cannot contain column aliases.
- The **GROUP BY** clause sort rows in ascending order, this can be overwritten with the **ORDER BY** clause.
- Any column specified in the **GROUP BY** clause are not required to be included in the **SELECT** clause.
- When the **GROUP BY** clause is used columns are grouped in the order of left to right.
- Group functions cannot have constraints in the **WHERE** clause. Group functions must be placed in the **HAVING** clause. The having clause will be discussed further in this chapter.

6.8 The HAVING Clause

Sometimes certain groups of data needs to be excluded from the SQL query. Unfortunately SQL syntax does not permit the exclusion of groups of data using the **WHERE** clause. For the purpose of data exclusion we need to use the **HAVING** clause instead of the **WHERE** clause.

SELECT group function(column1)
FROM customer, suburb
HAVING condition
GROUP BY column1

- The **GROUP BY** clause is required when categorizing data into groups.
- The **HAVING** clause is required when restricting group data **.**
- Table(s) to retrieve data from.
- **SELECT** clause identifies the columns and group functions to use.

Example 1

The below example groups the table rows based on **postcode** and **gender**. Once the groups have been established the average **age** calculation is performed for each group. Provided that the group condition **AVG(age)** is less than 40 the group data will be displayed.

```
SELECT postcode, gender, AVG(age)
FROM customer
GROUP BY postcode, gender
HAVING AVG(age) < 40
```

postcode	gender	AVG(age)
2232	M	35
2232	F	21
2245	M	28

Example 2

This example illustrates how to mix group functions between the **SELECT** and **HAVING** clause. It is important to remember in the below example the **MAX** function is used against the group values and not the result of the **AVG(age)** group function.

```
SELECT postcode, AVG(age)
FROM customer
GROUP BY postcode, gender
HAVING MAX(age) > 30
```

postcode	AVG(age)
2199	50
2232	28
2245	42
2345	43

6.9 Summary

- Group functions are used for grouping data together in order to perform a particular operation on that group.
- Oracle provides various group functions such as **AVG**, **SUM**, **MIN**, **MAX** and **COUNT**.
- By default null values are excluded from group functions. The **NVL** function is used to replace nulls with values within group data.
- The **GROUP BY** clause will group table information together based on a value within a particular column.
- The **HAVING** clause is used to place constantans on the data being selected.

6.10 Exercises

- Calculate the average age of all customers.
- Display the following information from the **customers** table on one row.

  - Minimum customer age

  - Maximum customer age

  - Average customer age

  - Sum of all customer ages
- Write an SQL query to determine the number of rows in the **employees** table based on the distinct values in the **manager_code** column (Do not include nulls in the evaluation).
- Write an SQL query to determine the number of rows in the **employees** table based on the distinct values in the **manager_code** column (Include nulls in the evaluation by replacing them with the value 9999).
- Display the following information from the customers table based on the following,
    - Group customer into **gender**
    - Only customers who have are older

7. Subqueries

7.1 Objectives

After this chapter you should be able to perform the following,

- Understand the general use and purpose of subqueries.
- Understand and write all type of subqueries.

7.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

TABLE : customer			
name	**postcode**	**gender**	**age**
Smith	2232	M	35
Jones	2232	F	21
Simons	2199	M	50
Thirley	2245	F	56
Ravenwood	2245	M	28
Andrews	2345	M	43

TABLE : employees			
code	**name**	**position**	**manager_code**
1000	Jeff	Sales Rep	2000
1001	Martin	Sales Rep	2000
2000	Darren	Sales Manager	3000
3000	Greg	General Manger	

7.3 Purpose of a Subquery

A subquery is a **SELECT** statement which is embedded in another **SELECT** statement. The subquery can appear in the **SELECT**, **WHERE** or **FROM** clause. Subqueries are used to identify rows in a table which match an unknown condition. For example, suppose we look at the below customer table and need to identify which customers are older than Smith.

TABLE : customer			
name	**postcode**	**gender**	**age**
Smith	2232	M	35
Jones	2232	F	21 **<**
Simons	2199	M	50 **>**
Thirley	2245	F	56 **>**
Ravenwood	2245	M	28 **<**
Andrews	2345	M	43 **>**

In order to find all customers who are older than Smith we need to perform two queries,

- Query1, Identify the age of customer Smith
- Query2, Identify all customers who are older than the result of Query1.

These two queries are actually written as one SQL expression. Query1 will actually be a subquery of Query2 as per the below Syntax.

Example 1

The below subquery display all customers who are older than Smith.

```
SELECT name, age
FROM customers where
age >
   (SELECT age
   FROM customers
   WHERE name =  'Smith')
```

name	**age**
Simons	50
Thirley	56
Andrews	43

In the above example the subquery returns a value of 35. This means that the main query will only display customers who have an age greater than 35.

7.4 Subquery Syntax and Rules

When using subquery functions there are various rules and syntax regulations to follow as per below,

Syntax Example 1

```
SELECT column1, column2
FROM table1
WHERE sub_query_link1 >
(SELECT sub_query_link2
FROM table2
WHERE sub_query_condition1)
```

- Determine the subquery value to pass to the main query.
- **FROM** clause identifies the table to use in the subquery.
- Subquery **SELECT** clause to determine the column to use in the subquery.
- **WHERE** clause comparison condition to join main query and subuery.
- **FROM** clause identifies the table to use in the main query .
- Main query **SELECT** clause identifies the columns to display.

Subquery Rules

- The Subquery can be placed in the **FROM**, **WHERE** or **HAVING** clause.
- The inner query (subquery) executes only once.
- The inner query (subquery) executes before the outer query (main query).
- The result of the inner query is used by the outer query.
- The inner and outer query can retrieve results from different tables.
- Subqueries must be enclosed in parenthesis.
- Subqueries must be placed on the right hand side of the comparison condition.
- The ORDER BY clause in a subquerie is not required unless Top-N analysis is being performed.
- Single-row subqueries should only return one row when single-row comparison conditions are used. If more than one row is returned Oracle will return an error message.
- When a subquery returns null the main query will not display anything.

7.5 Subquery Types

The link between the main query and the subquery is controlled by the comparison operator. The comparison operator determines if the subquery is a single-row subquery or a multiple-row subquery.

7.5.1 Single Row Subqueries

Single-row subqueries which use single-row operators(<,>,<=,>=,=,<>) return only one row from the inner subquery.

SELECT column1, column2 → Columns to display are defined in the main query
FROM table1 → Tables to lookup in the main query
WHERE sub_query_link1 > → Comparison condition uses single-row operator
(SELECT sub_query_link2 → Subquery columns are not displayes
FROM table2 → Tables to lookup in the subquery
WHERE sub_query_condition1) → The **WHERE** condition should only return one row

Example 1

The below subquery display all customers who are older than **Ravenwood**.

```
SELECT name, age
FROM customers
WHERE age >
   (SELECT age
   FROM customers
   WHERE name =  'Ravenwood')
```

name	age
Smith	35
Simons	50
Thirley	56
Andrews	43

In the above example the subquery returns a value of **28**. This means that the main query will only display customers who have an age greater than **28**.

Example 2

The below subquery display all customers who are older than **Ravenwood** and younger than **Andrews**. Notice that the main query contains two subqueries.

```
SELECT name, age
FROM customers
WHERE age >
   (SELECT age
   FROM customers
   WHERE name =  'Ravenwood')
AND age <
   (SELECT age
   FROM customers
   WHERE name =  'Andrews')
```

Name	age
Smith	35

In the above example the two inner subqueries return the lower and upper age limits used in the main query. The lower age limit from **Smith** and the upper age limit from **Andrews**.

7.5.1.1 Single Row Subqueries and Group Functions

Usually inner queries (subqueries) return a single row based on column(s) specified in their **SELECT** clause and a comparison operation that returns a single row. Another method of returning a single row is to use group functions in the inner **SELECT** clause.

SELECT column1, column2 ⟶ Columns to display are defined in the main query
FROM table1 ⟶ Tables to lookup in the main query
WHERE sub_query_link1 > ⟶ Comparison condition uses single-row operator
(SELECT group_function(column1) ⟶ Subquery column with group function
FROM table2) ⟶ Tables to lookup in the subquery

Example 1

The below subquery displays all customers who are older than the average age.

```
SELECT name, age
FROM customers
WHERE age >
   (SELECT AVG(age)
   FROM customers)
```

name	age
Simons	50
Thirley	56
Andrews	43

In the above example the subquery returns a value of **38**. This means that the main query will only display customers who have an age greater than **38**.

7.5.2 Multiple Row Subqueries

Multiple-row subqueries which use multiple-row operators(IN, ANY, All) return more than one row from the inner subquery.

The multiple-row operators returns TRUE under the following conditions,

- IN, If the value matches any element retuned from the subquery.
- < ANY, If the value is less than ANY of the elements returned from the subquery.
- > ANY, If the value is more than ANY of the elements returned from the subquery.
- < ALL, If the value is less than ALL of the elements returned from the subquery.
- > ALL, If the value is more than ALL of the elements returned from the subquery.

SELECT column1, column2 → Columns to display are defined in the main query
FROM table1 → Tables to lookup in the main query
WHERE sub_query_link1 IN → Comparison condition uses multiple-row operator
(SELECT sub_query_link2 → Subquery columns are not displayes
FROM table2 → Tables to lookup in the subquery
WHERE sub_query_condition1) → The **WHERE** condition should only return one row

Example 1

The below subquery displays the customers who are the same age as minimum age for each postcode.

```
SELECT name, age, postcode
FROM customers
WHERE age IN
   (SELECT MIN(age)
   FROM customers
   GROUP BY postcode)
```

name	age	postcode
Jones	21	2232
Simons	50	2199
Ravenwood	28	2245
Andrews	43	2345

In the above example the subquery returns the values of **21, 50, 28** and **43**. This means that the main query will only display customers who have an age equal to one of these elements.

Example 2

The below subquery displays the customers who are younger than anyone from postcodes 2240 and above.

```
SELECT name, age, postcode
FROM customers
WHERE age < ANY
   (SELECT age
   FROM customers
   WHERE postcode > 2240)
```

name	age	postcode
Smith	35	2232
Simons	50	2199
Thirley	56	2245
Andrews	43	2345

In the above example the subquery returns the values of **56, 28** and **43**. This means that the main query will only display customers who have an age less than **56** or **28** or **43**.

Note :

The **ANY** operator performs the same operation as its synonym the **SOME** operator.

Example 3

The below subquery displays the customers who are younger than the youngest customer from postcodes 2240 and above.

```
SELECT name, age, postcode
FROM customers
WHERE age < ALL
   (SELECT age
   FROM customers
   WHERE postcode > 2240)
```

name	age	postcode
Jones	21	2232

In the above example the subquery returns the values of **56, 28** and **43**. This means that the main query will only display customers who have an age less than **56** and **28** and **43**.

7.6 Subqueries and the HAVING Clause

Subqueries can also be used in the **HAVING** clause as per below.

SELECT column1, column2 → Columns to display are defined in the main query
FROM table1 → Tables to lookup in the main query
HAVING sub_query_link1 > → Comparison condition uses single row operator
(SELECT column1 → Subquery column with group function
FROM table2 → Tables to lookup in the subquery
WHERE sub_query_condition1) → Subquery constraint

Example 1

The below subquery displays all customers who are older than the average age of all customers.

```
SELECT name, age
FROM customers
HAVING age >
   (SELECT AVG(age)
   FROM customers)
```

Name	age
Simons	50
Thirley	56
Andrews	43

In the above example the subquery returns a value of **38**. This means that the main query will only display customers who have an age greater than **38**.

7.7 Null Values in a Subquery

It is important to remember that subqueries can return null values. For this reason certain precautions must be taken when it is likely that a subquery may return a null value.

Example 1

The below example attempts to find all employees who do not manage any employees.

```
SELECT code, name
FROM employees
WHERE code NOT IN
   (SELECT manager_code
   FROM employees)
```

name	age
No rows selected	

The values of **2000**, **3000** and **null** were returned from the subquery. This means that the main query is looking up every employee to see if they are equal to **2000**, **3000** or **null**. Since no employee code is null there will be no results generated from this query.

To fix this query we can modify the SQL expression as per below,

```
SELECT code, name
FROM employees
WHERE code NOT IN
   (SELECT manager_code
   FROM employees
   WHERE manager_code IS NOT NULL)
```

code	name
1000	Jeff
1001	Martin

7.8 Summary

- Subqueries have the below characteristics,
  - Appear as a **SELECT** statement which is embedded in another **SELECT** statement.
  - Can be included in the **SELECT**, **WHERE** or **FROM** clause.
  - Used to identify rows in a table which match an unknown condition.
- Single-row subqueries which use single-row operators(<,>,<=,>=,=,<>) return only one row from the inner subquery.
- Group functions can be used in inner querie to return single values for single-row subqueries.
- Multiple-row subqueries which use multiple-row operators(IN, ANY, All) return more than one row from the inner subquery.
- Precaution should be taken with Null values being returned from subqueries by the use of the **IS NOT NULL** phrase.

7.9 Exercises

- Write a single-row subquery to display all customer who are older than Ravenwood.
- Write a single-row subquery to display all customer who are older than Ravenwood and younger than Simons.
- Write a multiple-row subquery to display all users that are older than any customer whos name starts with am “S”.
- Write a multiple-row subquery to display all users that are older than all customer whos name starts with am “S”.
- Write a multiple-row subquery to display all customers who are within the age bracket of Ravenwood and Simons.

8. Data Manipulation

8.1 Objectives

After this chapter you should be able to perform the following,

- Understand and use the below DML statements,
  - INSERT
  - UPDATE
  - DELETE
- Control the state of SQL transactions and their data.

8.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

TABLE : employees			
name	**postcode**	**Gender**	**age**
Smith	2232	M	35
Jones	2232	F	21
Simons	2199	M	50
Thirley	2245	F	56
Ravenwood	2245	M	28
Andrews	2345	M	43

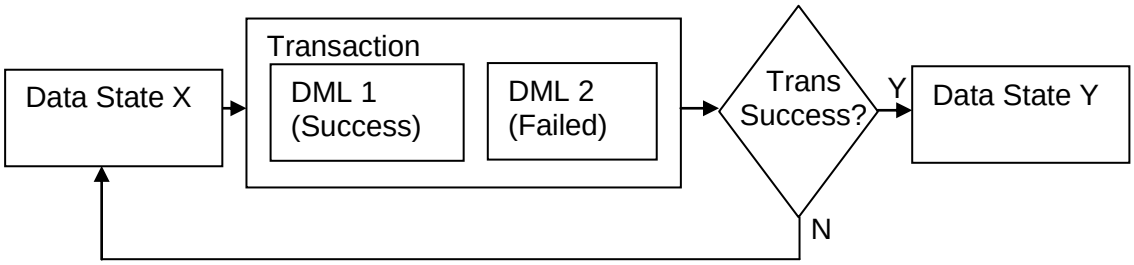
TABLE : vendorpo				
name	**todaysdate**	**purchdate**	**user**	**purchid**

TABLE : vendorpohist				
name	**todaysdate**	**purchdate**	**user**	**purchid**
Grey	10-SEP-2002	10-SEP-2003	PAULS	A01
Rooster	10-NOV-2003	10-NOV-2003	KAYS	A02

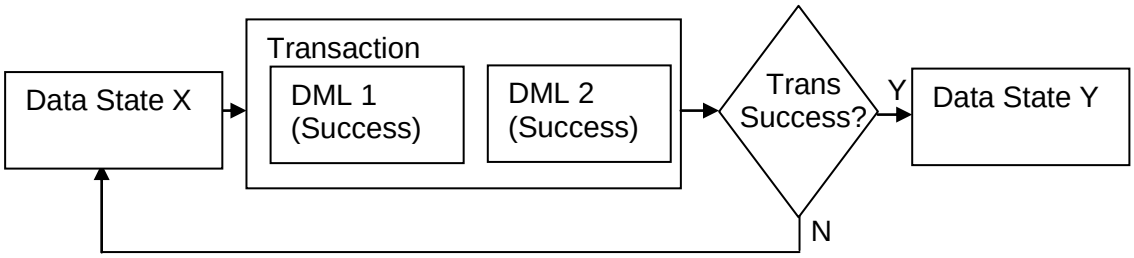
8.3 Data Manipulation Language (DML)

Data manipulation language (DML) operations are performed when data in the database is inserted, deleted or updated. A single or series of DML operations belongs to what is known as a transaction. If a transaction contains a series of DML operations then all of those operations must be successful in order for the transaction to complete successfully. If any one of those DML operations within the transaction fails then all the other DML operations that make up that transaction are rolled back.

Failed Transaction



Successful Transaction



8.4 INSERT Statement

The **INSERT** statement will insert rows into a table. The addition of rows into a table can be performed using the following methods,

- Inserting literal strings using the **VALUES** clause.
- Requesting user input during the **INSERT** operation.
- Copying rows from another table during the **INSERT** operation.

8.4.1 INSERT VALUES Clause

The **INSERT VALUES** clause will insert rows into a table with information provided in the SQL statement as per the below syntax,

INSERT INTO table1 ⟶ Name of table to insert rows into
VALUES (value1, value2, …) ⟶ Column values of new row

INSERT VALUSE Clause Rules

- The data entered in the **VALUES** clause needs to be specified in the order of the table columns.
- If the order of the data in the **VALUES** clause is different from the order of the columns in the table then the column order needs to be specified.
- If column names are not used to specify the order of the values then all fields must be accounted for in the correct order within the **VALUES** clause.
- Character and date values must be enclosed with single quotes.

Example 1

The below **INSERT** statement inserts a single row into the customer table.

```
INSERT INTO employees
VALUES ('Peters','2345','M','32')
```

1 row created

Example 2

The below **INSERT** statement inserts a single row into the customer table.

```
INSERT INTO employees(age,postcode,name,gender)
VALUES ('70','2999', 'Adams','F')
```

1 row created

The above example inserts the column values in the order specified (left to right).

Result

TABLE : customer

name	postcode	Gender	age
Smith	2232	M	35
Jones	2232	F	21
Simons	2199	M	50
Thirley	2245	F	56
Ravenwood	2245	M	28
Andrews	2345	M	43
Adams	2999	F	70
Peters	2345	M	32

8.4.2 INSERT and Nulls

When inserting data into columns a particular value must be provided which matches the column data type. If the data provided is not of the correct data type then the Oracle server will attempt an implicit data type conversion on the data before inserting it. However, if no data is specified then null values will be inserted instead.

The below **INSERT** command provides an example of how null values are implicitly or explicitly used before the **INSERT** operation takes place.

Example 1

This example uses null values implicitly for certain columns.

```
INSERT INTO employees (name,postcode)
VALUES ('Knowles','2000')
```

1 row created

Example 2

This example uses null values explicitly for certain columns.

```
INSERT INTO employees
VALUES ('Grant','2000',null, null)
```

1 row created

Result

TABLE : employees

name	postcode	Gender	age
Smith	2232	M	35
Jones	2232	F	21
Simons	2199	M	50
Thirley	2245	F	56
Ravenwood	2245	M	28
Andrews	2345	M	43
Adams	2999	F	70
Peters	2345	M	32
Knowles	2000		
Grant	2000		

8.4.3 INSERT Rules

Certain rules need to be followed when inserting literal values as per below,

- If a column has a constraint of **NOT NULL** then a value must be specified during an **INSERT** operation.
- If a column has a constraint of **UNIQUE** then the value being inserted into that column must not already exist in the table.
- If a column has a constraint of **FOREIGN KEY** then the value being inserted must exist in the key relationship.
- If a column has a constraint of **CHECK** then the value being inserted met not violate the **CHECK** constraint.
- **INSERT** values must be of the required data type.
- Use empty quotes ('') or zero (0) in the **VALUES** clause to avoid inserting nulls.
- Make sure values are not to wide to fit in the target column.

8.4.4 INSERT and Functions

Functions can be used in the **INSERT** operation to provide certain types of information. The call to the function replaces a literal value which is located within the **VALUES** clause.

INSERT INTO table1 ⟶ Name of table to insert rows into
VALUES (value1, function_call, …) ⟶ Column values of new row (Including function call)

Example 1

The below **INSERT** statement inserts a single row into the customer table.

```
INSERT INTO employees
VALUES ('Thompson','2898','M',STR(ROUND(32.7)))
```

1 row created

TABLE : employees

name	postcode	Gender	age
Smith	2232	M	35
Jones	2232	F	21
Simons	2199	M	50
Thirley	2245	F	56
Ravenwood	2245	M	28
Andrews	2345	M	43
Adams	2999	F	70
Peters	2345	M	32
Knowles	2000		
Grant	2000		
Thompson	2898	M	33

The function call in the above example is actually a nested function which rounds of a literal value and then performs an explicit string conversion.

Example 2

The below **INSERT** statement inserts a single row into the **vendorpo** table.

```
INSERT INTO vendorpo
VALUES ('ABC Bearings',SYSDATE,TO_DATE(11-01-04,'DD/MM/YY'),USER,'A12')
```

1 row created

TABLE : vendorpo

name	todaysdate	purchdate	user	purchid
ABC Bearings	11-FEB-2004	11-JAN-2004	JOHNS	A12

This example uses three system functions as per below,

- SYSDATE, Returns the current system date
- USER, Returns the current user logged in
- TO_DATE, Performs an explicit conversion on a string data-type to a date data-type

8.4.5 INSERT and Substitution Values

INSERT statements can be written so that when they are executed the user is prompted for input. The user input provided will then be assigned to variables declared in the **INSERT** statement. Variables are identified in the **INSERT** statement by the use of the **&** operator as per below,

INSERT INTO table1 ⟶ Name of table to insert rows into
VALUES (value1, '&variable1') ⟶ Values and variables to use during the insert

Example 1

The below **INSERT** statement inserts a single row into the **vendorpo** table but first prompts the user for the values that are to be inserted.

```
INSERT INTO vendorpo
VALUES ('&name',SYSDATE,TO_DATE(11-01-04,'DD/MM/YY'),USER,'A13')
```

```
Name : London
1 row created
```

TABLE : vendorpo

name	todaysdate	purchdate	user	purchid
London	11-FEB-2004	11-JAN-2004	JOHNS	A13

Substitution Value Rules

- The **&** operator is used to identify and mark the position of a substitution value in the SQL statement.
- If a substitution value is located within single quotes ('') then the **INSERT** operation expects a string or date data type.
- If a substitution value is not located within single quotes ('') then the **INSERT** operation expects a number data type.

The role of substation values has its main impact when combined with the use of scripting. If the above **INSERT** statement was saved as a script then it could simply be called numerous times and prompt the user for different values during every execution cycle.

To execute a script from the SQL command line simply type the script filename preceded by the @ symbol.

Example 2

The below example runs a script from the SQL command line,

SQL>@C:\TEMP\Script.sql

8.4.6 Inserting Rows From Another Table

The **INSERT** command can also be used to insert rows that are located in another table. This is achieved by embedding a **SELECT** statement within the **INSERT** statement.

INSERT INTO table1(column1, column2, column3) → **Choose the target table**
SELECT column4, column5, column6 → **Choose the columns to insert**
FROM table2 → **Choose the source table**
WHERE condition1 → **Restrict rows to copy**

Example 1

The below **INSERT** statement copies rows from **vendorpohist** into **vendorpo**.

INSERT INTO vendorpo **SELECT * FROM vendorpohist**
2 rows created

TABLE : vendorpo				
Name	**Todaysdate**	**purchdate**	**user**	**purchid**
London	11-FEB-2004	11-JAN-2004	JOHNS	A13
Grey	10-SEP-2002	10-SEP-2003	PAULS	A01
Rooster	10-NOV-2003	10-NOV-2003	KAYS	A02

Note :

- This query could also have been written by specifying all the required table columns as per below.
- If a column is omitted then a NULL is put in its place.

INSERT INTO vendorpo(name,todaysdate,purchdate,user,purchid)
SELECT name,todaysdate,purchdate,user,purchid FROM vendrpohist

Copying Rules and Guidelines

- The **VALUES** clause is not used when copying from other tables.
- The number of columns specified in the **INSERT** clause must total the number of columns used in the corresponding **SELECT** clause.
- The data types of columns specified in the **INSERT** clause must match the columns used in the corresponding **SELECT** clause.
- Literal values can replace columns names in the **SELECT** clause.

8.4.7 INSERT with Subqueries

When an **INSERT** statement is executed the results are displayed as the number of rows inserted. To query the data directly after it has been inserted include a subquery in the **INSERT** operation as per below.

INSERT INTO → **INSERT statement**
(SELECT column1, column2, column3 → **Choose columns to query**
FROM table2 → **Choose table to query**
WHERE condition1) → **Restrict rows to query**
VALUES (value1, value2, value3) → **Declare values to insert**

Example 1

The below **INSERT** statement will insert the row and then display all the rows as per the subquery.

```
INSERT INTO
(SELECT name, todaysdate, purchdate, user, purchid
FROM vendorpohist)
VALUES ('Toms','10-MAR-2001','10-MAR-2001','JOHNS','A04')
```

name	todaysdate	purchdate	user	purchid
London	11-FEB-2004	11-JAN-2004	JOHNS	A13
Grey	10-SEP-2002	10-SEP-2003	PAULS	A01
Rooster	10-NOV-2003	10-NOV-2003	KAYS	A02
Toms	10-MAR-2001	10-MAR-2001	JOHNS	A04

8.5 UPDATE Statement

The **UPDATE** statement will update existing rows in a table. Much like the **INSERT** statement the **UPDATE** statement can be performed using the following methods,

- Updating columns using literal values.
- Copying rows from another table during the **UPDATE** operation.
- Requesting user input during the **UPDATE** operation.

8.5.1 UPDATE with Literal Values

The **UPDATE** statement in its simplest form as per below will amend rows with literal values which are supplied within the **UPDATE** statement.

UPDATE table1 → **Choose the table to update**
SET col1 = value1, col2= value2 → **Choose the field(s) to update**
WHERE condition1 → **Restrict rows to copy**

If the **WHERE** clause is omitted from the **UPDATE** then all rows in the table are updated according to the **SET** clause.

Example 1

The below **UPDATE** statement amends rows in the **vendorpo** table.

```
UPDATE vendorpo
SET purchdate = '01-JAN-2004'
```

3 rows updated

TABLE : vendorpo

name	todaysdate	purchdate	user	purchid
London	11-FEB-2004	01-JAN-2004	JOHNS	A13
Grey	10-SEP-2002	01-JAN-2004	PAULS	A01
Rooster	10-NOV-2003	01-JAN-2004	KAYS	A02
Toms	10-MAR-2001	01-JAN-2004	JOHNS	A04

Because the **WHERE** clause is omitted all rows in the table are updated.

Example 2

The below **UPDATE** statement amends rows in the **vendorpo** table.

```
UPDATE vendorpo
SET purchdate = '01-JAN-2003', puchdate = '01-JAN-2003'
WHERE name = 'Grey'
```

1 row updated

TABLE : vendorpo

Name	todaysdate	purchdate	user	purchid
London	11-FEB-2004	01-JAN-2004	JOHNS	A13
Grey	01-JAN-2003	01-JAN-2003	PAULS	A01
Rooster	10-NOV-2003	01-JAN-2004	KAYS	A02
Toms	10-MAR-2001	01-JAN-2004	JOHNS	A04

Because the **WHERE** clause is included only specific rows in the table are updated.

8.5.2 Updating Rows From Another Table

The **UPDATE** command can also be used to update rows with values that are located in another table. This is achieved by embedding a **SELECT** statement within the **UPDATE** statement.

UPDATE table1 → **Choose the target table**
SET column1 = → **Choose the columns to update**
(SELECT column2 from table2) → **Choose the source data**
WHERE condition1 → **Restrict rows to update**

Example 1

The below **UPDATE** statement amends rows in **vendorpo** from data in **vendorpohist**.

```
UPDATE vendorpo
SET vendorpo.purchdate =
(SELECT MAX(vendorpohist.purchdate) from vendorpohist)
WHERE vendorpo.name = vendorpohist.name
```

1 row updated

TABLE : vendorpo

Name	todaysdate	purchdate	user	purchid
London	11-FEB-2004	01-JAN-2004	JOHNS	A13
Grey	01-JAN-2003	01-JAN-2004	PAULS	A01
Rooster	10-NOV-2003	01-JAN-2004	KAYS	A02
Toms	10-MAR-2001	01-JAN-2004	JOHNS	A04

8.6 DELETE Statement

The **DELETE** statement will delete existing rows in a table. Much like the **INSERT** statement the **DELETE** statement can be performed using the following methods,

- Updating columns using literal values.
- Copying rows from another table during the **DELETE** operation.
- Requesting user input during the **DELETE** operation.

Before performing a **DELETE** operation the values being deleted are first checked to determine if they are belong to any foreign/primary key relationship. If the data targeted for deletion has this type of integrity constraint then the **DELETE** operation returns with a "child record found" error.

8.6.1 DELETE with Literal Values

The **DELETE** statement in its simplest form as per below will delete rows with literal values which are supplied within the **DELETE** statement.

DELETE FROM table1 ⟶ **Choose the table to delete from**
WHERE condition1 ⟶ **Restrict rows to delete**

If the **WHERE** clause is omitted from the **DELETE** statement then all rows in the table are removed.

Example 1

The below **UPDATE** statement amends rows in **vendorpo** from data in **vendorpohist**.

DELETE FROM vendorpo **WHERE name = 'Rooster'**
1 row delete

TABLE : vendorpo				
Name	**todaysdate**	**purchdate**	**user**	**purchid**
London	11-FEB-2004	01-JAN-2004	JOHNS	A13
Grey	01-JAN-2003	01-JAN-2004	PAULS	A01
Toms	10-MAR-2001	01-JAN-2004	JOHNS	A04
Rooster row deleted				

8.6.2 Deleting Rows Based On Another Table

The **DELETE** command can also be used to remove rows based on values that are located in another table. This is achieved by embedding a **SELECT** statement within the **DELETE** statement.

DELETE FROM table1 → **Choose the table to delete from**
WHERE value1 = → **Restrict rows to remove**
(SELECT column2 from table2) → **Choose what to base the delete on**

Example 1

The below **DELETE** statement deletes rows in **vendorpo** based on data in **vendorpohist**.

```
DELETE FROM vendorpo
WHERE vendorpo.name =
(SELECT vendorpohist.name FROM vendorpohist.name
WHERE purchid = 'A04')
```

1 row updated

TABLE : vendorpo

Name	todaysdate	purchdate	user	purchid
London	11-FEB-2004	01-JAN-2004	JOHNS	A13
Grey	01-JAN-2004	01-JAN-2004	JOHNS	A02
Toms row deleted				

8.7 Default Values

When inserting or changing values the keyword **DEFAULT** can be used instead of a value. This instructs the DML statement to use the predefined **DEFAULT** value during the DML operation. The **DEFAULT** value is declared when the table is created.

INSERT INTO table1 ⟶ Name of table to insert rows into
VALUES (value1, value2, DEFAULT) ⟶ Values and default values to use during the DML

Example 1

The below **INSERT** statement insert a row with the **users** columns default value of **"SYS"**.

INSERT INTO vendorpo **VALUES (Davids, '01-MAR-2004','01-MAR-2004',DEFAULT, 'A00')**
1 row updated

TABLE : vendorpo				
name	**todaysdate**	**purchdate**	**user**	**Purchid**
London	11-FEB-2004	01-JAN-2004	JOHNS	A13
Grey	01-JAN-2004	01-JAN-2004	JOHNS	A02
Davids	01-MAR-2004	01-MAR-2004	SYS	A00

Example 2

The below **UPDATE** statement amends rows in the **vendorpo** table with the **users** columns default value of **"SYS"**.

UPDATE vendorpo **SET user = DEFAULT** **WHERE name = 'London'**
1 row updated

TABLE : vendorpo				
Name	**todaysdate**	**purchdate**	**user**	**purchid**
London	11-FEB-2004	01-JAN-2004	SYS	A13
Grey	01-JAN-2004	01-JAN-2004	JOHNS	A02
Davids	01-MAR-2004	01-MAR-2004	SYS	A00

Rules for default values

- If no default value has been predefined then NULL is used.
- The **DEFAULT** keyword can be used with **INSERT** and **UPDATE** DML operations.

8.8 MERGE Statement

The **MERGE** operation will perform one of two operations. It will insert data if the data does not already exist or it will update data if it does exist. The **MERGE** operation determines if data exists by a join condition specified in the **ON** clause.

MERGE INTO table1 alias1 → **Choose the target table**
USING table2 alias2 → **Choose the source table**
ON join_condition → **Determine the target and source join**
WHEN MATCHED THEN → **Perform update if exists**
UPDATE statements → **Normal UPDATE statement**
WHEN NOT MATCHED THEN → **Perform insert if doe not exist**
INSERT statements → **Normal INSERT statement**

Example 1

The below **MERGE** statement inserts and updates rows in the **vendorpo** table based on the data in the **vendorpohist** table. The join condition is based on the **name** column.

```
MERGE INTO vendorpo t1
USING vendorpohist t2
ON t1.name = t2.name
WHEN MATCHED THEN
   UPDATE SET
   t1.todaysdate = t2.todaysdate
   t1.purchdate = t2.purchdate
   t1.user = t2.user
   t1.purchid = t2.purchid
WHEN NOT MATCHED THEN
   INSERT VALUES(t2.name,t2.todaysdaye,t2,purchdate,t2,user,t2.purchid)
```

1 row updated

TABLE : vendorpo

Name	todaysdate	purchdate	user	purchid
London	11-FEB-2004	01-JAN-2004	SYS	A13
Davids	01-MAR-2004	01-MAR-2004	SYS	A00
Grey	01-JAN-2004	01-JAN-2004	JOHNS	A02
Rooster	10-NOV-2003	10-NOV-2003	KAYS	A02

The above results indicate that **'Grey'** was updated and **'Rooster'** was inserted.

8.9 Database Transactions

Database transactions fall into the three below categories.

Database Transaction Types	
Type	**Description**
DCL	Data Control Language
DDL	Data Definition Language
DML	Data Manipulation Language

A database transaction commences when a DML statement is executed. The database transaction will remain active until one of the following conditions,

- A **COMMIT** or **ROLLBACK** statement s issued.
- The iSQL session is terminated.
- A DCL or DDL statement is issued.
- The database is shutdown.

Once the transaction is closed the next transaction commences when the next DML statement is executed.

8.9.1 DCL Transactions

A Data Control Language transaction is one such as **GRANT** or **REVOKE**. These transactions are committed automatically so only one statement exists in the transactions. Because these transactions commit automatically it is not possible to rollback.

8.9.2 DDL Transactions

A Data Definition Language transaction is one such as **CREATE** or **ALTER**. These transactions are committed automatically so only one statement exists in the transactions. Because these transactions commit automatically it is not possible to rollback.

8.9.3 DML Transactions

A Data Manipulation Language transaction is one such as **INSERT, UPDATE** or **DELETE**. These transactions are described in section 8.4, 8.5 AND 8.6.

8.9.4 Transaction Lifecycle

The lifecycle of a transaction can be controlled through three commands. These three commands instruct the Oracle server to perform certain activity with the contents of a transaction as listed below.

Transaction Control Commands	
Command	**Description**
COMMIT	Applies the contents of the transaction to the database. Once the **COMMIT** statement is issued it cannot be undone
ROLLBACK	Drops the contents of the transaction. Once the **ROLLBACK** statement is issued it cannot be undone
SAVEPOINT x	Marks a pointer in the tranaction list (Where **x** is the name of the savepoint). The **ROLLBACK** statement can be issued with instruction to rollback to the savepoint.
ROLLBACK TO SAVEPOINT x	Drops the contents of the transaction back to the specified savepoint. Once the **ROLLBACK TO SAVEPOINT** statement is issued it cannot be undone

One important point to remember is the available set of transaction control commands only execute in the scope of the current SQL session. The commands will have no impact to others users performing DML operations.

All savepoints are given a marker name. The marker name instructs the ROLLBACK command where to rollback to. If the marker name is used a second time it will overwrite the position of the first marker.

Example1

The above commands would typically be used as per the below transaction lifecycle,

Transaction Groups	**DML Operations**	**Transaction Control Operations**
Transaction 1	*Start of transaction*	
	INSERT…	
	DELETE…	
		ROLLBACK
Transaction 2	*Start of transaction*	
	UPDATE…	
		SAVEPOINT X
	INSERT…	
		ROLLBACK TO X
		COMMIT

In Transaction 1 the **ROLLBACK** statement will drop all the changes made by the **INSERT** and **DELETE** operations leaving the data in a state as it was from the start of transaction 1.

In Transaction 2 there is a **SAVEPOINT X** command which bookmarks a position in the set of DML operations. At a later date the **ROLLBACK TO X** command is issued which instructs the oracle server to rollback to the state of the data as it was when the **SAVEPOINT X** command was issued.

8.9.5 Data State

The visibility of data to users through the course of the transaction lifecycle changes depending on the occurrence of any **COMMIT** or **ROLLBACK** commands.

Example1

The below example demonstrates the visibility of the data during the course of a simple transaction.

Transaction Groups	DML Operations	Data State
	INSERT…	Y
	DELETE…	Y
	ROLLBACK…	X
Transaction 2	UPDATE…	Y
	INSERT…	Y
	COMMIT	Z

Data State X

- Data is visible to all users.
- Changes are rolled back to original data state.
- The data rows being changed are unlocked.

Data State Y

- Data is visible only to user changing the data.
- Changes are not permanent in the database and can be rolled back.
- The data rows being changed are locked.

Data State Z

- Data is visible to all users.
- Changes are made permanent in the database and cannot be rolled back.
- All **SAVEPOINT** references are no longer valid.
- Locks on the data rows are removed.

8.9.6 Implicit Transaction Handling

A transaction is completed automatically under the following conditions,

Implicit Transaction Handling	
Event	**Action**
The iSQL session is terminated	Automatic ROLLBACK
A DCL or DDL statement is issued	Automatic COMMIT
The database is shutdown	Automatic ROLLBACK
System failure	Automatic ROLLBACK
Automatic COMMIT	Automatic COMMIT

Note :

When the Automatic COMMIT database parameter is set to true if forces a **COMMIT** after every DML transaction.

8.9.7 Statement Level Rollback

The Oracle server inserts implicit savepoints between users transactions. This is performed so that if any statement within a transaction should fail abnormally the data state is rolled back to the point of the last implicit savepoint.

Example 1

The below example demonstrates how the Oracle server inserts and uses implicit savepoints to achieve statement level rollback.

Transaction Groups	DML Operations	Transaction Control Operations
Transaction 1	*Start of transaction*	
	INSERT…	
		Implicit SAVEPOINT
	DELETE…	
		Implicit SAVEPOINT
	UPDATE…	
		Implicit SAVEPOINT
	INSERT…	***Error occurs***

8.9.8 Read Consistency

Read consistency is an automatic process controlled by the Oracle server which ensures all users have constant read access to data at all times. This is achieved by making data available as it was before the last **COMMIT**.

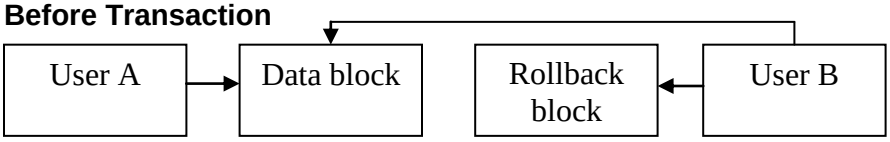
Read consistency ensures the following,

- All changes within one users transaction do not impact with the changes in another users transaction.
- Data is available for read while other transactions are writing it.
- Data is available for write while other transactions are reading it.

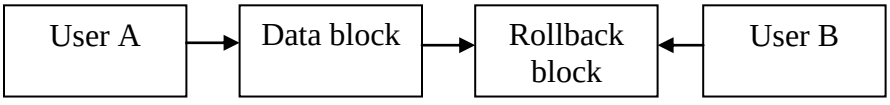
Example 1

The below demonstrates how read consistency is managed by the Oracle Server,

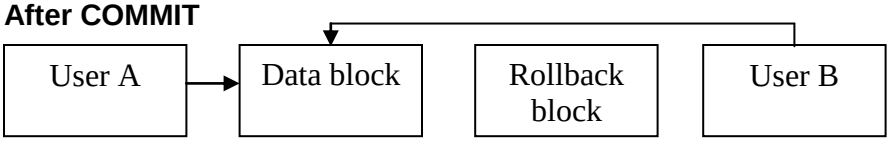
Before Transaction



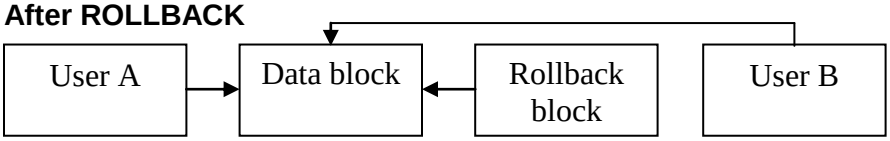
During Transaction



After COMMIT



After ROLLBACK



8.9.9 Locking

Locking is an automatic process controlled by the Oracle server which is used to ensure that concurrent DML operations do not interfere with each other and produce erroneous results. The locking is carried out at the row level, this means that when specific rows in one table are being updated they are not available for update by any other user. Once the transaction performing the update is completed the locks are released.

8.9.9.1 Implicit Locking

Implicit locking occurs automatically and is managed by the Oracles server. These type off locks occur for all SQL statements except the **SELECT** statement. Once the transaction is completed all locks are release from the database object and the data it contains. There are two levels of implicit locking modes as explained below. Both of these modes occur during DML operations.

Shared mode

The shared mode lock occurs at the object level. For example, when rows in a table are being updated the table is locked in shared mode. This means that DDL operations are inhibited on the table.

Exclusive mode

The exclusive mode lock occurs at the data level. This permits other user to acquire shared mode locks on the same objects but places exclusive locks on data within the object. Once an exclusive lock has been established against data no other user can acquire an exclusive lock on the same data.

8.9.9.2 Explicit Locking

Users can also lock data manually. This is known as explicit locking.

8.10 Summary

- Data manipulation language (DML) operations are performed when data in the database is inserted, deleted or updated.
- A single or series of DML operations belongs to what is known as a transaction.
- The **INSERT** statement will insert rows into a table.
- The **UPDATE** statement will update existing rows in a table.
- The **DELETE** statement will delete existing rows in a table
- The keyword **DEFAULT** can be used instead of a value. This will use the predefined **DEFAULT** value during a DML operation.
- The **MERGE** will insert data if it does not already exist otherwise it will update existing data.
- A DCL (Data Control Language) transaction is one such as **GRANT** or **REVOKE**.
- A Data Definition Language transaction is one such as **CREATE** or **ALTER**.
- A Data Manipulation Language transaction is one such as **INSERT, UPDATE** or **DELETE**.
- A database transaction commences when a DML statement is executed.
- Data visibility varies through the course of a transaction lifecycle depending on the occurrence of any **COMMIT** or **ROLLBACK** commands.
- Read consistency ensures all users have constant read access to data at all times.
- Locking ensures that concurrent DML operations do not interfere with each other and produce erroneous results.

8.11 Exercises

- Insert the following values into the employees table,

Field	Values
Name	Adams
Postcode	2345
Gender	M
Age	32

- Insert the following values into the employees table and the round the Age to the closest whole number

Field	Values
Name	Charles
Postcode	2345
Gender	M
Age	49.6

- Change the postcode to 2360 for all record in the employees table which have a postcode of 2345. Make sure that the update operation prompts for the new postcode.
- Delete all record in the employees table who have a postcode of 2345.
- Confirm the successful use of the COMMIT and ROLLBACK commands by performing the following,
  Session 1, Update all employee postcodes to 1000 (Do not COMMIT)
  Session 1, Query all employee postcode (What are the results?)
  Session 2, Query all employee postcode (What are the results?)
  Session 1, COMMIT the update operation
  Session 1, Query all employee postcode (What are the results?)
  Session 2, Query all employee postcode (What are the results?)
- Confirm the use of implicit locking by performing the following,
  Session 1, Update all employee postcodes to 2000 (Do not COMMIT)
  Session 2, Update all employee postcodes to 3000 (What are the results?)
  Session 1, ROLLBACK the update operation

9. Table Management

9.1 Objectives

After this chapter you should be able to perform the following,

- Perform the below table operations,
  - CREATE
  - DROP
  - RENAME
  - TRUNCATE
  - ALTER
- Understand the datatypes that are available when declaring columns in a table.

9.2 Test Data

The below table information is used as test data for the examples shown throughout this chapter.

9.3 Object Types

The database provides a series of structures to store data and assist with data storage as per below,

Object Types	
Type	**Description**
Table	Provides basic storage for data constructed of rows and columns
View	Logical representation of data from table(s)
Sequence	Sequential number generator
Index	Provides indexed representation of specific table columns
Synonym	Used for declaring alias names for database objects

The above object types are merely an example of the available object types. Oracle provide a lot more object types for various types of storage and functionality.

Object Rules

- Must start with a letter.
- Must be between 1 and 30 characters long.
- Can only contain the following characters,
  - A-Z
  - a-z
  - 0-9
  - _, $ and #
- Duplicate object names are prohibited.
- Object names cannot be the same as reserved words. For example, you cannot create a table called “SELECT”.
- Object names are not case sensitive.

9.4 CREATE TABLE

The CREATE TABLE data definition language statement is used to create tables. Unless specified otherwise this table is created within the users default tablespace and by default is ready to store data.

CREATE TABLE → Command to create the table
schema.table_name → Specify which schema to use and the table name
column_name → Every column in a table must have a name
datatype → Every column in a table must be of a certain datatype
size → Every column in a table must be of a certain size
DEFAULT value → Value is used if data is omitted during insert

Example 1

The below **CREATE TABLE** statement creates a table called POSTBOX,

```
CREATE TABLE POSTBOX(
PO_NUM NUMBER(2),
SUBURB STRING(20),
NAME STRING(20))
```

Table Created

Example 2

The tables structure can be verified/displayed by using the describe command as per the below example,

DESC POSTBOX

NAME	NULL?	TYPE
PO_NUM		NUMBER(2)
SUBURB		STRING(20)
STRING		STRING(20)

Note:

- The **CREATE TABLE** statement is DDL so an automatic **COMMIT** occurs.
- Only users who have the **CREATE TABLE** privilege can create tables.
- To avoid null values is table columns a default value for a column can be specified when the table is created. In the event of a null value in a table column during an **INSERT** operation this value will be inserted instead. The **DEFAULT** value can be of a literal value, SQL operation or expression.

9.5 Table Scope

All tables in the database belong to a schema(user). The schema is considered as the owner of the table. Tables within one schema are not visible to another schema unless the relevant permission is granted. Once the relevant permission is granted from one schema to another the table will be become visible. To reference a table that exists in another schema the table must be prefixed with the owning schema name.

Example 1

The below example uses the owning schema name to reference a table.

SELECT * FROM → Standard SELECT clause
schema.table_name → Specify which schema to use and the table name

9.6 Tables used in Oracle

9.6.1 User Tables

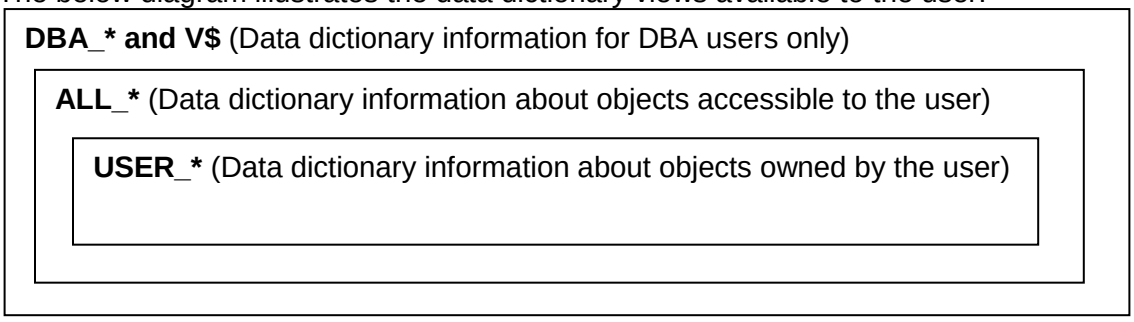
All tables created by the user are owned by that user. These tables are used to store normal user data.

9.6.2 Data Dictionary Tables

Data dictionary tables store information about the database. These tables are managed by the Oracle server and owned by the DBA user SYS. The values in these tables should not be changed by any user. The data within the tables is not legible so various data dictionary views have been built upon the tables to provide an easy to understand interface between the user and the data dictionary.

Data Dictionary Scope

The below diagram illustrates the data dictionary views available to the user.



Example 1

The below example lists all the table accessible to the user,

SELECT table_name from ALL_TABLES
Table list . . ……

Example 2

The below example lists all objects owned by the user,

SELECT * from USER_CATALOG
Object list . . ……

9.7 Data Type

The Oracle server permits various data types for storing all form of information in tables. The below table illustrates the data type offered by the Oracle server.

Data Types	
Type	**Description**
VARCHAR2(num)	Variable length alphanumeric
CHAR(num)	Fixed length alphanumeric
NUMBER(num,dec)	Variable length numeric. Format is num.dec (E.g, 4.1 = 9999.9)
DATE	Date and time values between 1st Jan 4712 B.C to 31st Dec 9999 A.D
LONG	Variable length alphanumeric (2Gb max)
CLOB	Alphanumeric data (4Gb max)
RAW	Raw binary data (2K bits max)
LONG RAW	Raw binary data (2Gb max)
BLOB	Raw binary data (4Gb max)
BFILE	Binary file storage (4Gb max)
ROWID	64 bit encoded row identification of a single row in the database

Data Type Restrictions

Long data types place a few restrictions on table operations as per below,

- When creating a table using a sub-query the content of any **LONG** column will not be copied.
- When using **SELECT** statements a **LONG** column cannot be used in the **GROUP** BY or **ORDER BY** clause.
- Only one **LONG** data type is permitted per table.
- Constraints cannot be defined on a **LONG** data type.

Most table restrictions relate to the use of LONG data types. If there are any table requirements which conflict with the above constrictions you may need to consider using a **CLOB** data type rather than a **LONG**.

9.8 DATETIME Data Types

Oracle 9i introduces several new Date and Time data type features as per below.

Data Types	
Type	**Description**
TIMESTAMP	Date and time permitting fractions of second to be stored
INTERVAL YEAR TO MONTH	Duration of time in months
INTERVAL DAY TO SECOND	Duration of time in seconds

9.8.1 Timestamp

The TIMESTAMP data type provides the same functionality as the3 DATE data type plus more. Besides the usual Day, Month, Year, Hour, Minute and Second components it also allows fractions of a second. The precision of the fractional component is determined when the data type is declared as per below.