SQL Short notes

chapter 1

A (SELECT) statement retrieves information from the database.

In the syntax:

SELECT is a list of one or more columns

\* selects all columns

DISTINCT suppresses duplicates

column|expression selects the named column or the expression

alias gives selected columns different headings

FROM table specifies the table containing the columns

• SQL statements are not case sensitive, unless indicated.

SQL statements can be entered on one or many lines.

• Keywords cannot be split across lines or abbreviated.

Clauses are usually placed on separate lines for readability and ease of editing.

• Indents should be used to make code more readable.

Keywords typically are entered in uppercase; all other words, such as table names and columns, are entered in lowercase.

(Operator Precedence)

Multiplication and division take priority over addition and subtraction.

Operators of the same priority are evaluated from left to right.

Parentheses are used to force prioritized evaluation and to clarify statements.

(Defining a Null Value)

A null is a value that is unavailable, unassigned, unknown, or inapplicable.

A null is not the same as zero or a blank space.

In the COMMISSION\_PCT column in the EMPLOYEES table,

notice that only a sales manager or sales representative can earn a commission.

Other employees are not entitled to earn commissions. A null represents that fact.

(A column alias)

Renames a column heading

Is useful with calculations

Immediately follows the column name -

there can also be the optional AS keyword between the column name and alias

Requires double quotation marks if it contains spaces or special characters or is case sensitive

(Concatenation Operator)

Concatenates columns or character strings to other columns

Is represented by two vertical bars (||)

Creates a resultant column that is a character expression

(Literal Character Strings)

A literal is a character, a number, or a date included in the SELECT list.

Date and character literal values must be enclosed within single quotation marks.

Each character string is output once for each  
row returned.

example

SELECT last\_name||' is a '||job\_id

AS "Employee Details"

(Duplicate Rows (continued))

To eliminate duplicate rows in the result, include the DISTINCT keyword in the SELECT clause

example

SELECT DISTINCT department\_id, job\_id

FROM employees;



**Note:** The SQL\*Plus environment is covered in Appendix C.

**Displaying the Table Structure**

**Describe departments**



**Chapter 2**

**Where clause**

**Limiting the Rows Selected**

**You can restrict the rows returned from the query by using the WHERE clause.**

WHERErestricts the query to rows that meet a condition

*condition*is composed of column names, expressions, constants, and a comparison operator

The WHERE clause can compare values in columns, literal values, arithmetic expressions, or functions. It consists of three elements:

* + - Column name
    - Comparison condition
    - Column name, constant, or list of values

**Character Strings and Dates**

Character strings and dates in the WHERE clause must be enclosed in single quotation marks (''). Number constants, however, should not be enclosed in single quotation marks.

**Comparison Conditions**

**For Example**

... WHERE hire\_date='01-JAN-95'

... WHERE salary>=6000

... WHERE last\_name='Smith'

An alias cannot be used in the WHERE clause.

**Note:** The symbol != and ^= can also represent the *not equal to* condition.

**The BETWEEN Condition**

You can display rows based on a range of values using the BETWEEN range condition. The range that you specify contains a lower limit and an upper limit.

**Note : that BETWEEN … AND … is actually translated by Oracle server to a pair of AND conditions: (a >= lower limit) AND (a <= higher limit).**

**The IN Condition**

To test for values in a specified set of values, use the IN condition. The IN condition is also known as the *membership condition*.

Example

SELECT employee\_id, manager\_id, department\_id

FROM employees

WHERE last\_name IN ('Hartstein', 'Vargas');

If characters or dates are used in the list, they must be enclosed in single quotation marks ('').

**Instructor Note**

Explain that IN ( ... ) is actually translated by Oracle server to a set of OR conditions: a = value1 OR a = value2 OR a = value3.

**Using the LIKE Condition**

* Use the LIKE condition to perform wildcard searches of valid search string values.
* Search conditions can contain either literal characters or numbers:
  + % denotes zero or many characters.
  + \_ denotes one character.



Example

SELECT last\_name, hire\_date

FROM employees

WHERE hire\_date LIKE '%95';

**The ESCAPE Option**

When you need to have an exact match for the actual *%* and *\_* characters, use the ESCAPE option.

Example

SELECT employee\_id, last\_name, job\_id

FROM employees

WHERE job\_id LIKE '%SA\\_%' ESCAPE '\';

**The NULL Conditions**

The IS NULL condition tests for nulls. A null value means the value is unavailable

. Therefore, you cannot test with = because a null cannot be equal or unequal

to any value.

Example. to display last name, job ID, and commission for all employees who are NOT entitled to get a commission, use the following SQL statement:

SELECT last\_name, job\_id, commission\_pct

FROM employees

WHERE commission\_pct IS NULL;

**Logical Conditions**

**A logical condition combines the result of two component conditions t**

* + - AND
    - OR
    - NOT

**Using the AND Operator**

**AND requires both conditions to be true**

SELECT employee\_id, last\_name, job\_id, salary

FROM employees

WHERE salary >=10000

AND job\_id LIKE '%MAN%';

**Using the OR Operator**

**OR requires either condition to be true**

SELECT employee\_id, last\_name, job\_id, salary

FROM employees

WHERE salary >= 10000

OR job\_id LIKE '%MAN%';

**The NOT Operator**

The slide example displays the last name and job ID of all employees whose job ID *is not* IT\_PROG, ST\_CLERK, or SA\_REP.

SELECT last\_name, job\_id

FROM employees

WHERE job\_id

NOT IN ('IT\_PROG', 'ST\_CLERK', 'SA\_REP');

**Note:** The NOT operator can also be used with other SQL operators, such as BETWEEN, LIKE, and NULL.

... WHERE job\_id NOT IN ('AC\_ACCOUNT', 'AD\_VP')

... WHERE salary NOT BETWEEN 10000 AND 15000

... WHERE last\_name NOT LIKE '%A%'

... WHERE commission\_pct IS NOT NULL

**Rules of Precedence**

The rules of precedence determine the order in which expressions are evaluated and calculated.

**SELECT last\_name, job\_id, salary**

**FROM employees**

**WHERE job\_id = 'SA\_REP'**

**OR job\_id = 'AD\_PRES'**

**AND salary > 15000;**

**The ORDER BY Clause**

The order of rows returned in a query result is undefined. The ORDER BY clause can be used to sort the rows.

Sort rows with the ORDER BY clause

* ASC: ascending order, default
* DESC: descending order
* The ORDER BY clause comes last in the SELECT statement.

SELECT last\_name, job\_id, department\_id, hire\_date

FROM employees

ORDER BY hire\_date desc ;

**Sorting by Column Aliases**

You can use a column alias in the ORDER BY clause. The slide example sorts the data by annual salary.

**SELECT employee\_id, last\_name, salary\*12 annsal**

**FROM employees**

**ORDER BY annsal;**

**Sorting by Multiple Columns**

**Example**

Display the last names and salaries of all employees. Order the result by department number, and then in descending order by salary.

SELECT last\_name, salary

FROM employees

ORDER BY department\_id, salary DESC;

**Chapter 3**

**Single row function**

**Objectives**

**After completing this lesson, you should be able to**

**do the following:**

* **Describe various types of functions available  
  in SQL**
* **Use character, number, and date functions in SELECT statements**

**Describe the use of conversion functions**

**SQL Functions**

Functions are a very powerful feature of SQL and can be used to do the following:

* + - Perform calculations on data
    - Modify individual data items
    - Manipulate output for groups of rows
    - Format dates and numbers for display
    - Convert column data types

SQL functions sometimes take arguments and always return a value.

**Note:** Most of the functions described in this lesson are specific to Oracle’s version of SQL.

**SQL Functions (continued)**

There are two distinct types of functions:

* + - Single-row functions
    - Multiple-row functions

**Single-Row Functions**

These functions operate on single rows only and return one result per row.

**Multiple-Row Functions**

Functions can manipulate groups of rows to give one result per group of rows. These functions are known as group functions. This is covered in a later lesson.

**Single row functions:**

* **Manipulate data items**
* **Accept arguments and return one value**
* **Act on each row returned**
* **Return one result per row**
* **May modify the data type**
* **Can be nested**
* **Accept arguments which can be a column or an expression**
* ***function\_name* [(*arg1, arg2,...*)]**

**Single-Row Functions**

Single-row functions are used to manipulate data items. They accept one or more arguments and return one value for each row returned by the query

* + - User-supplied constant
    - Variable value
    - Column name
    - Expression

**Features of single-row functions include:**

* + - Acting on each row returned in the query
    - Returning one result per row
    - Possibly returning a data value of a different type than that referenced
    - Possibly expecting one or more arguments
    - Can be used in SELECT, WHERE, and ORDER BY clauses; can be nested

**In the syntax:**

*function\_name* is the name of the function.

*arg1, arg2* is any argument to be used by the function. This can be represented by a column name or expression.

**Single-Row Functions (continued)**

This lesson covers the following single-row functions:

* + - Character functions: Accept character input and can return both character and number values
    - Number functions: Accept numeric input and return numeric values
    - Date functions: Operate on values of the DATE data type (All date functions return a value of DATE data type except the MONTHS\_BETWEEN function, which returns a number.)
    - Conversion functions: Convert a value from one data type to another
    - General functions:
      * NVL
      * NVL2
      * NULLIF
      * COALSECE
      * CASE
      * DECODE

**Character Functions**

Character functions can be divided into the following

* + - Case-manipulation functions
    - Character-manipulation functions

**Note:** The functions discussed in this lesson are only some of the available functions

**Case Manipulation Functions**

LOWER, UPPER, and INITCAP are the three case-conversion functions.

* + - LOWER: Converts mixed case or uppercase character strings to lowercase
    - UPPER: Converts mixed case or lowercase character strings to uppercase
    - INITCAP: Converts the first letter of each word to uppercase and remaining letters to lowercase

SELECT 'The job id for '||UPPER(last\_name)||' is '

||LOWER(job\_id) AS "EMPLOYEE DETAILS"

FROM employees;

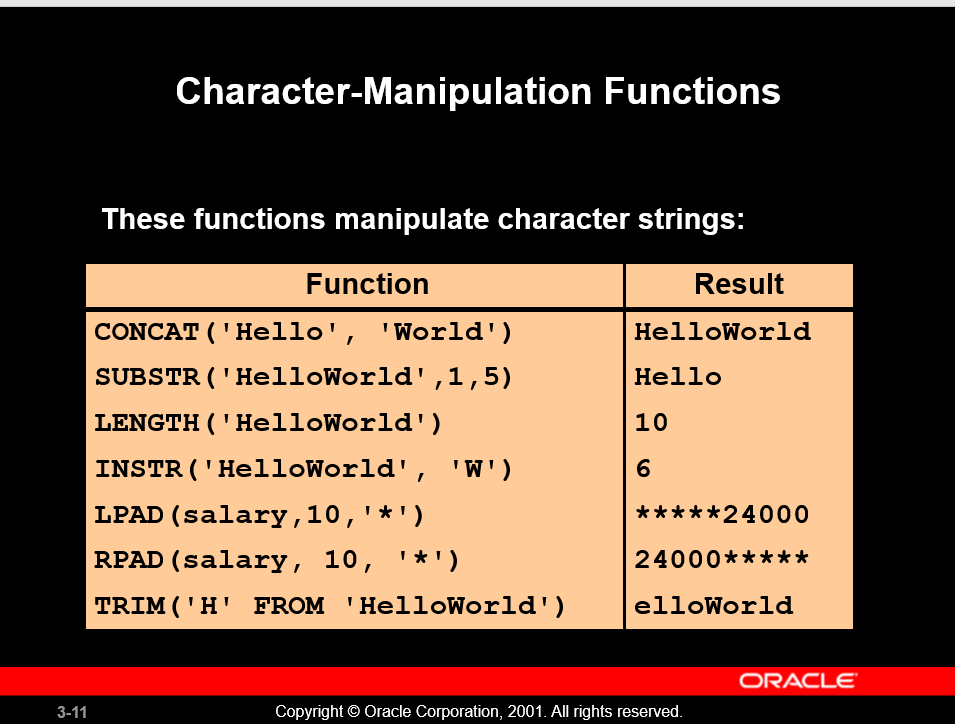
**Slide example**

**SELECT employee\_id, last\_name, department\_id**

**FROM employees**

**WHERE LOWER(last\_name) = 'higgins';**

**Character Manipulation Functions**



CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD, and TRIM are the character manipulation functions covered in this lesson.

* + - CONCAT: Joins values together (You are limited to using two parameters with CONCAT.)
    - SUBSTR: Extracts a string of determined length
    - LENGTH: Shows the length of a string as a numeric value
    - INSTR: Finds numeric position of a named character
    - LPAD: Pads the character value right-justified
    - RPAD: Pads the character value left-justified
    - TRIM: Trims heading or trailing characters (or both) from a character string (If *trim\_character* or *trim\_source* is a character literal, you must enclose it in single quotes.)

**Instructor Note**

Be sure to point out RPAD to the students, because this function is needed in a practice exercise. Also, TRIM, which was a new function in Oracle8*i*, does the job of both the LTRIM and the RTRIM functions.

**Example**

Modify the SQL statement on the slide to display the data for those employees whose last names end with an *n*.

SELECT employee\_id, CONCAT(first\_name, last\_name) NAME,

LENGTH (last\_name), INSTR(last\_name, 'a') "Contains 'a'?"

FROM employees

**WHERE SUBSTR(last\_name, -1, 1) = 'n'**;

**Number Functions**

Number functions accept numeric input and return numeric values.

* **ROUND: Rounds value to specified decimal**

**ROUND(45.926, 2) 45.93**

**ROUND(45.926, -1) 50**

**ROUND(45.926, 0) 0**

* **TRUNC: Truncates value to specified decimal**

**TRUNC (45.926, 2) 45.92**

**TRUNC (45.926, -2) 0**

**TRUNC (45.926) 45**

* **MOD: Returns remainder of division**

**MOD(1600, 300) 100**

**MOD Function**

**The MOD function finds the remainder of value1 divided by value2**

**Working with Dates**

* **Oracle database stores dates in an internal numeric format: century, year, month, day, hours, minutes, seconds.**
* **The default date display format is DD-MON-RR.**
  + **Allows you to store 21st century dates in the 20th century by specifying only the last two digits of the year.**
  + **Allows you to store 20th century dates in the 21st century in the same way.**

**This data is stored internally as follows:**

**CENTURY YEAR MONTH DAY HOUR MINUTE SECOND**

**19 94 06 07 5 10 43**

**WORKING WITH DATES**

**Sysdate is a function that returns**

* **Date**
* **Time**

**The SYSDATE Function**

SYSDATE is a date function that returns the current database server date and time

**Example**

Display the current date using the DUAL table.

SELECT SYSDATE

FROM DUAL

**Arithmetic with Dates**

Since the database stores dates as numbers, you can perform calculations using arithmetic operator 

**SELECT last\_name, (SYSDATE-hire\_date)/7 AS WEEKS**

**FROM employees**

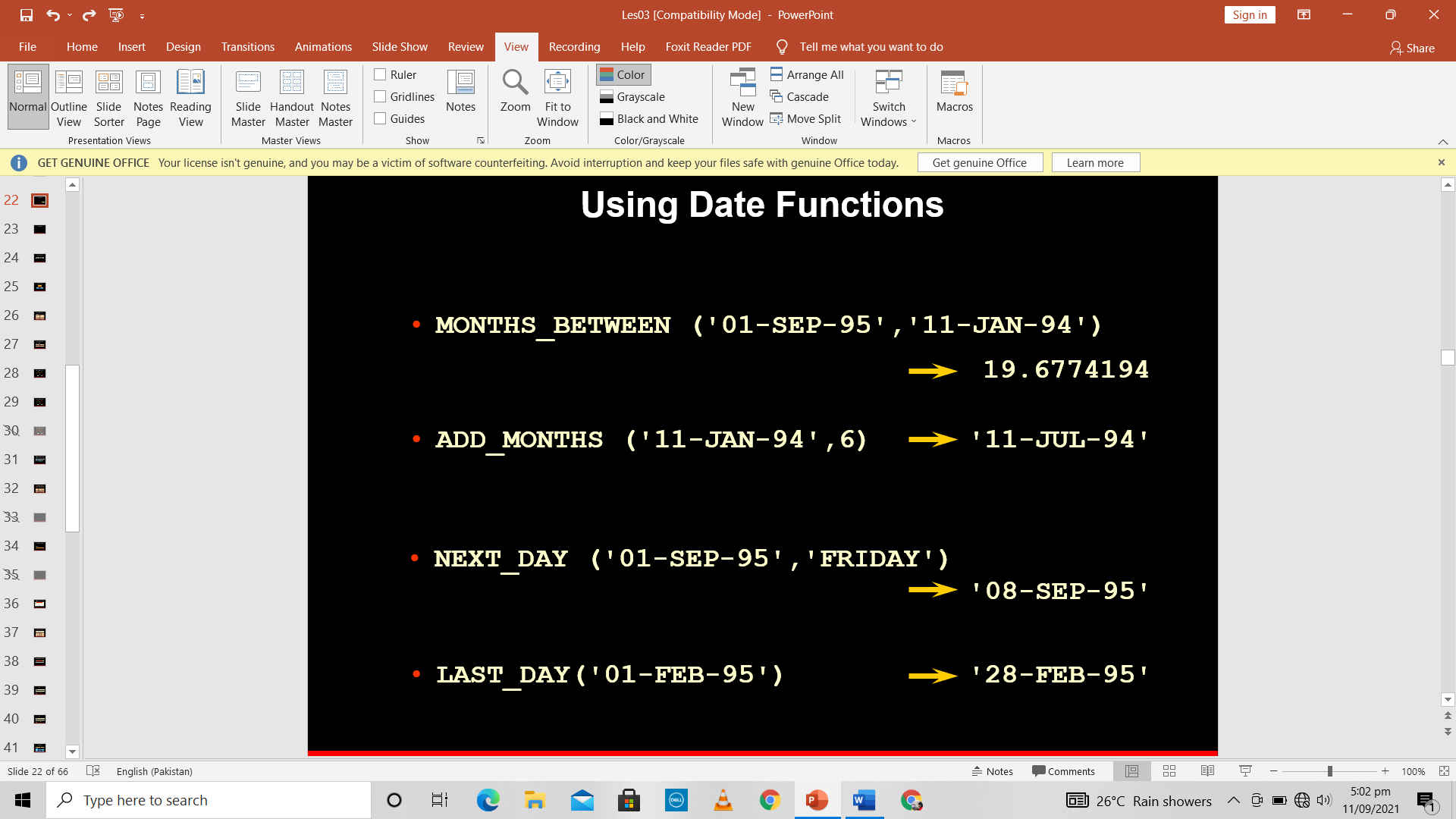
**WHERE department\_id = 90;**

**Date Functions**

Date functions operate on Oracle dates. All date functions return a value of DATE data type except MONTHS\_BETWEEN, which returns a numeric value.

* + - MONTHS\_BETWEEN(*date1, date2*): Finds the number of months between *date1* and *date2*. The result can be positive or negative. If *date1* is later than *date2*, the result is positive; if *date1* is earlier than *date2*, the result is negative. The noninteger part of the result represents a portion of the month.
    - ADD\_MONTHS(*date, n*): Adds *n* number of calendar months to *date*. The value of *n* must be an integer and can be negative.
    - NEXT\_DAY(*date,* '*char*'): Finds the date of the next specified day of the week ('*char*') following *date*. The value of *char* may be a number representing a day or a character string.
    - LAST\_DAY(*date*): Finds the date of the last day of the month that contains *date*.
    - ROUND(*date*[,'*fmt*']): Returns *date* rounded tothe unit specified by the format model *fmt.* If the format model *fmt* is omitted, *date* is rounded to the nearest day.
    - TRUNC(*date*[, '*fmt*']): Returns *date* with the time portion of the day truncated to the unit specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is truncated to the nearest day.

This list is a subset of the available date functions. The format models are covered later in this lesson. Examples of format models are month and year.



For example, display the employee number, hire date, number of months employed, six-month review date, first Friday after hire date, and last day of the hire month for all employees employed for fewer than 36 months.

SELECT employee\_id, hire\_date,

MONTHS\_BETWEEN (SYSDATE, hire\_date) TENURE,

ADD\_MONTHS (hire\_date, 6) REVIEW,

NEXT\_DAY (hire\_date, 'FRIDAY'), LAST\_DAY(hire\_date)

FROM employees

WHERE MONTHS\_BETWEEN (SYSDATE, hire\_date) < 36;

**Conversion function**

Implicit data type conversions work according to the rules explained in the next two slides.

Explicit data type conversions are done by using the conversion functions. Conversion functions convert a value from one data type to another

Note: Although implicit data type conversion is available, it is recommended that you do explicit data type conversion to ensure the reliability of your SQL statements

**Implicit Data Type Conversion**

The assignment succeeds if the Oracle server can convert the data type of the value used in the assignment to that of the assignment target.

**Note:** CHAR to NUMBER conversions succeed only if the character string represents a valid number.

**Instructor Note**

There are several new data types available in the Oracle9*i* release pertaining to time. These include: TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR, INTERVAL DAY. These are discussed later in the course.

You can also refer students to the *Oracle9i SQL Reference,* “Basic Elements of Oracle SQL.”

**Explicit Data Type Conversion**

SQL provides three functions to convert a value from one data type to another:





**TO\_CHAR(*date,* '*format\_model*')**

**The format model:**

* **Must be enclosed in single quotation marks and is case sensitive**
* **Can include any valid date format element**
* **Has an *fm* element to remove padded blanks or suppress leading zeros**
* **Is separated from the date value by a comma**

EXAMPLE SELECT employee\_id, TO\_CHAR(hire\_date, 'MM/YY') Month\_Hired

FROM employees

WHERE last\_name = 'Higgins';

**Sample Format Elements of Valid Date Formats**

**Date Format Elements - Time Formats**

Use the formats listed in the following tables to display time information and literals and to change numerals to spelled numbers

**Other Formats**



**Specifying Suffixes to Influence Number Display**

**Example**

Modify the slide example to display the dates in a format that appears as Seventh of June 1994 12:00:00 AM.

SELECT last\_name,

TO\_CHAR(hire\_date, 'fmDdspth "of" Month YYYY fmHH:MI:SS AM')

HIREDATE

FROM employees;

**The TO\_CHAR Function with Numbers**

**Number Format Elements**

If you are converting a number to the character data type, you can use the following format elements:

**SELECT TO\_CHAR(salary, '$99,999.00') SALARY**

**FROM employees**

**WHERE last\_name = 'Ernst';**

**The TO\_NUMBER and TO\_DATE Functions**

You may want to convert a character string to either a number or a date. To accomplish this task, use the TO\_NUMBER or TO\_DATE functions. The format model you choose is based on the previously demonstrated format elements.

**Nesting Functions (continued)**

The slide example displays the head of the company, who has no manager. The evaluation of the SQL statement involves two steps:

1. Evaluate the inner function to convert a number value to a character string.

* + - * Result1 = TO\_CHAR(manager\_id)

2. Evaluate the outer function to replace the null value with a text string.

* + - * NVL(Result1, 'No Manager')

The entire expression becomes the column heading because no column alias was given.

**Example**

Display the date of the next Friday that is six months from the hire date. The resulting date should appear as Friday, August 13th, 1999. Order the results by hire date.

SELECT TO\_CHAR(NEXT\_DAY(ADD\_MONTHS

(hire\_date, 6), 'FRIDAY'),

'fmDay, Month DDth, YYYY')

"Next 6 Month Review"

FROM employees

ORDER BY hire\_date;

**Instructor Note**

Demo: 3\_nest.sql

Purpose: To illustrate nesting of several single row functions

**General Functions**

These functions work with any data type and pertain to the use of null values in the expression list.



**The NVL Function**

To convert a null value to an actual value, use the NVL function.



To calculate the annual compensation of all employees, you need to multiply the monthly salary by 12 and then add the commission percentage to it.

SELECT last\_name, salary, commission\_pct,

(salary\*12) + (salary\*12\*commission\_pct) AN\_SAL

FROM employees;

Chapter 4

Display data from multiple tables

**Objectives**

**After completing this lesson, you should be able to**

**do the following:**

* **Write SELECT statements to access data from more than one table using equality and nonequality joins**
* **View data that generally does not meet a join condition by using outer joins**
* **Join a table to itself by using a self join**

**Data from Multiple Tables**

Sometimes you need to use data from more than one table.

* + - Employee IDs exist in the EMPLOYEES table.
    - Department IDs exist in both the EMPLOYEES and DEPARTMENTS tables.
    - Location IDs exist in the DEPARTMENTS table.

To produce the report, you need to link the EMPLOYEES and DEPARTMENTS tables and access data from both of them.

**Cartesian Products**

* **A Cartesian product is formed when:**
  + **A join condition is omitted**
  + **A join condition is invalid**
  + **All rows in the first table are joined to all rows in the second table**
* **To avoid a Cartesian product, always include a valid join condition in a WHERE clause.**
* **When a join condition is invalid or omitted completely**
* Cartesian products are useful for some tests when you need to generate a large number of rows to simulate a reasonable amount of data.
* A Cartesian product is generated if a join condition is omitted. The example on the slide displays employee last name and department name from the EMPLOYEES and DEPARTMENTS tables. Because no WHERE clause has been specified, all rows (20 rows) from the EMPLOYEES table are joined with all rows (8 rows) in the DEPARTMENTS table, thereby generating 160 rows in the output.

SELECT last\_name, department\_name dept\_name

FROM employees, departments;

**Types of Joins**

**Oracle Proprietary**

**Joins (8*i* and prior):**

* **Equijoin**
* **Non-equijoin**
* **Outer join**
* **Self join**

Compliant Joins

* **Cross joins**
* **Natural joins**
* **Using clause**
* **Full or two sided outer joins**
* **Arbitrary join conditions for outer joins**

**Use a join to query data from more than one table.**

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

**WHERE *table1.column1* = *table2.column2***

* **Write the join condition in the WHERE clause.**
* **Prefix the column name with the table name when the same column name appears in more than one table.**

**When data from more than one table in the database is required, a *join* condition is used**

, write a simple join condition in the WHERE clause.

In the syntax:

*table1.column* denotes the table and column from which data is retrieved

*table1.column1* = is the condition that joins (or relates) the tables together  
 *table2.column2*

**Guidelines**

* + - When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
    - If the same column name appears in more than one table, the column name must be prefixed with the table name.
    - To join *n* tables together, you need a minimum of n-1 join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.
* **Equijoins**
* To determine an employee’s department name, you compare the value in the DEPARTMENT\_ID column in the EMPLOYEES table with the DEPARTMENT\_ID values in the DEPARTMENTS table. The relationship between the EMPLOYEES and DEPARTMENTS tables is an *equijoin*—that is, values in the DEPARTMENT\_ID column on both tables must be equal. Frequently, this type of join involves primary and foreign key complements.
* **Note:** Equijoins are also called *simple joins* or *inner joins*.

**Instructor Note**

Explain the use of a decision matrix for simplifying writing joins. For example, if you want to display the name and department number of all the employees who are in the same department as Goyal, you can start by making the following decision tree:

***Columns to Display Originating Table Condition***

last\_name employees last\_name='Goyal'

department\_name departments employees.department\_id = departments.department\_id

Now the SQL statement can be easily formulated by looking at the decision matrix. The first column gives the column list in the SELECT statement, the second column gives the tables for the FROM clause, and the third column gives the condition for the WHERE clause.

**Retrieving Records with Equijoins**

SELECT employees.employee\_id, employees.last\_name,

employees.department\_id, departments.department\_id,

departments.location\_id

FROM employees, departments

WHERE employees.department\_id = departments.department\_id;

**Retrieving Records with Equijoins**

In the slide example:

* + - The SELECT clause specifies the column names to retrieve:
      * employee last name, employee number, and department number, which are columns in the EMPLOYEES table
      * department number, department name, and location ID, which are columns in the DEPARTMENTS table
    - The FROM clause specifies the two tables that the database must access:
      * EMPLOYEES table
      * DEPARTMENTS table
    - The WHERE clause specifies how the tables are to be joined:

EMPLOYEES.DEPARTMENT\_ID = DEPARTMENTS.DEPARTMENT\_ID

Because the DEPARTMENT\_ID column is common to both tables, it must be prefixed by the table name to avoid ambiguity.

**Additional Search Conditions**

In addition to the join, you may have criteria for your WHERE clause to restrict the rows under consideration for one or more tables in the join. For example, to display employee Matos' department number and department name, you need an additional condition in the WHERE clause.

SELECT last\_name, employees.department\_id,

department\_name

FROM employees, departments

WHERE employees.department\_id = departments.department\_id

AND last\_name = 'Matos';

**Qualifying Ambiguous Column Names**

You need to qualify the names of the columns in the WHERE clause with the table name to avoid ambiguity. Without the table prefixes, the DEPARTMENT\_ID column could be from either the DEPARTMENTS table or the EMPLOYEES table. It is necessary to add the table prefix to execute your query.

**Notice how table aliases are identified in the FROM clause in the example**

**Guidelines**

* + - Table aliases can be up to 30 characters in length, but shorter is better.
    - If a table alias is used for a particular table name in the FROM clause, then that table alias must be substituted for the table name throughout the SELECT statement.
    - Table aliases should be meaningful.
    - The table alias is valid only for the current SELECT statement.

**Additional Search Conditions**

Sometimes you may need to join more than two tables. For example, to display the last name, the department name, and the city for each employee, you have to join the EMPLOYEES, DEPARTMENTS, and LOCATIONS tables.

SELECT e.last\_name, d.department\_name, l.city

FROM employees e, departments d, locations l

WHERE e.department\_id = d.department\_id

AND d.location\_id = l.location\_id;

**Non-Equijoins**

A non-equijoin is a join condition containing something other than an equality operator.

The relationship between the EMPLOYEES table and the JOB\_GRADES table has an example of a non-equijoin.

**Non-Equijoins (continued)**

The slide example creates a non-equijoin to evaluate an employee’s salary grade. The salary must be *between* any pair of the low and high salary ranges

**Outer Joins**

**Using Outer Joins to Return Records with No Direct Match**

The missing rows can be returned if an *outer join* operator is used in the join condition. The operator is a plus sign enclosed in parentheses (+), and it is *placed on the* “*side*” *of the join that is deficient in information*. This operator has the effect of creating one or more null rows, to which one or more rows from the nondeficient table can be joined.

In the syntax:

*table1.column =* is the condition that joins (or relates) the tables together.

*table2.column* (+) is the outer join symbol, which can be placed on either side of the  
 WHERE clause condition, but not on both sides. (Place the outer  
 join symbol following the name of the column in the table without the matching rows.)

**Syntax**

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

**WHERE *table1.column(+)* = *table2.column;***

Or

**SELECT *table1.column, table2.column***

**FROM *table1, table2***

**WHERE *table1.column* = *table2.column(+);***

**Joining a Table to Itself**

Sometimes you need to join a table to itself. To find the name of each employee’s manager, you need to join the EMPLOYEES table to itself, or perform a self join. For example, to find the name of Whalen’s manager, you need to:

* + - Find Whalen in the EMPLOYEES table by looking at the LAST\_NAME column.
    - Find the manager number for Whalen by looking at the MANAGER\_ID column. Whalen’s manager number is 101.
    - Find the name of the manager with EMPLOYEE\_ID 101 by looking at the LAST\_NAME column. Kochhar’s employee number is 101, so Kochhar is Whalen’s manager.

In this process, you look in the table twice. The first time you look in the table to find Whalen in the LAST\_NAME column and MANAGER\_ID value of 101. The second time you look in the EMPLOYEE\_ID column to find 101 and the LAST\_NAME column to find Kochhar.

Example SELECT worker.last\_name || ' works for '

|| manager.last\_name

FROM employees worker, employees manager

WHERE worker.manager\_id = manager.employee\_id

**Creating Cross Joins**

* **The CROSS JOIN clause produces the cross-product of two tables.**
* **This is the same as a Cartesian product between the two tables.**

**SELECT last\_name, department\_name**

**FROM employees**

**CROSS JOIN departments ;**

**Creating Natural Joins**

* **The NATURAL JOIN clause is based on all columns in the two tables that have the same name.**
* **It selects rows from the two tables that have equal values in all matched columns.**
* **If the columns having the same names have different data types, an error is returned.**

**SELECT department\_id, department\_name,**

**location\_id, city**

**FROM departments**

**NATURAL JOIN locations ;**

**Creating Joins with the USING Clause**

* **If several columns have the same names but the data types do not match, the NATURAL JOIN clause can be modified with the USING clause to specify the columns that should be used for an equijoin.**
* **Use the USING clause to match only one column when more than one column matches.**
* **Do not use a table name or alias in the referenced columns.**
* **The NATURAL JOIN and USING clauses are mutually exclusive.**

**SELECT e.employee\_id, e.last\_name, d.location\_id**

**FROM employees e JOIN departments d**

**USING (department\_id) ;**

**Creating Joins with the ON Clause**

* **The join condition for the natural join is basically an equijoin of all columns with the same name.**
* **To specify arbitrary conditions or specify columns to join, the ON clause is used.**
* **The join condition is separated from other *search* conditions.**
* **The ON clause makes code easy to understand.**

**SELECT e.employee\_id, e.last\_name, e.department\_id,**

**d.department\_id, d.location\_id**

**FROM employees e JOIN departments d**

**ON (e.department\_id = d.department\_id);**

**Creating Three-Way Joins with the ON Clause**

**SELECT employee\_id, city, department\_name**

**FROM employees e**

**JOIN departments d**

**ON d.department\_id = e.department\_id**

**JOIN locations l**

**ON d.location\_id = l.location\_id;**

**INNER Versus OUTER Joins**

* **In SQL: 1999, the join of two tables returning only matched rows is an inner join.**
* **A join between two tables that returns the results of the inner join as well as unmatched rows left (or right) tables is a left (or right) outer join.**
* **A join between two tables that returns the results of an inner join as well as the results of a left and right join is a full outer join.**

 **Summary**

There are multiple ways to join tables.

**Types of Joins**

* + - Equijoins
    - Non-equijoins
    - Outer joins
    - Self joins
    - Cross joins
    - Natural joins
    - Full or outer joins

**Cartesian Products**

A Cartesian product results in all combinations of rows displayed. This is done by either omitting the WHERE clause or specifying the CROSS JOIN clause.

**Table Aliases**

* + - Table aliases speed up database access.
    - Table aliases can help to keep SQL code smaller, by conserving memory.