LAB # 5: SINGLE ROW AND GROUP FUNCTIONS

Objective:

In this lab students will learn how to display the data according to the conditions provided in the query and how to display the sorted data.

SQL Functions

Functions are a very powerful feature of SQL. They 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.

Two Types of SQL Functions

There are two 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. There are different types of single-row functions. This lesson covers the following ones:

- Character
- Number
- Date
- Conversion
- General

Multiple-Row Functions

Functions can manipulate groups of rows to give one result per group of rows. These functions are also known as *group functions*.

Single-Row Functions

Single-row functions:

- Manipulate data items
- Accept arguments and return one value
- Act on each row that is returned
- Return one result per row
- May modify the data type
- Can be nested
- Accept arguments that can be a column or an expression

Syntax:

```
function_name [(arg1, arg2,...)]
```

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.

This lab 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 the DATE data type except the MONTHS_BETWEEN function, which returns a number.)

The following single-row functions are discussed in the next lab titled ―Using Conversion Functions and Conditional Expressions‖:

- **Conversion functions:** Convert a value from one data type to another
- **General functions:**
    - NVL
    - NVL2
    - NULLIF
    - COALESCE
    - CASE

- DECODE

Character Functions

Single-row character functions accept character data as input and can return both character and numeric values. Character functions can be divided into the following:

- Case-conversion functions
- Character-manipulation functions

Case-Conversion 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 the remaining letters to lowercase

Function	Result
LOWER('SQL Course')	sql course
UPPER('SQL Course')	SQL COURSE
INITCAP('SQL Course')	Sql Course

Example:

```
SELECT 'The job id for '||UPPER(last_name)||' is '
       ||LOWER(job_id) AS "EMPLOYEE DETAILS"
FROM   employees;
```

Using Case-Conversion Functions

Example 1:

Display the employee number, name, and department number for employee Higgins:

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  last_name = 'higgins';
```

Output:

```
0 rows selected
```

This example displays the employee number, name, and department number of employee Higgins.

The WHERE clause of the above SQL statement specifies the employee name as higgins. Because all the data in the EMPLOYEES table is stored in proper case, the name higgins does not find a match in the table, and no rows are selected.

Example 1:

```
SELECT employee_id, last_name, department_id
FROM   employees
WHERE  LOWER(last_name) = 'higgins';
```

Output:

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	205	Higgins	110

The WHERE clause of the above SQL statement specifies that the employee name in the EMPLOYEES table is compared to higgins, converting the LAST_NAME column to lowercase for comparison purposes. Because both names are now lowercase, a match is found and one row is selected. The WHERE clause can be rewritten in the following manner to produce the same result:

...WHERE last_name = 'Higgins'

The name in the output appears as it was stored in the database. To display the name in uppercase, use the UPPER function in the SELECT statement.

```
SELECT employee_id, UPPER(last_name), department_id
FROM   employees
WHERE  INITCAP(last_name) = 'Higgins';
```

Character-Manipulation Functions

CONCAT, SUBSTR, LENGTH, INSTR, LPAD, RPAD, and TRIM are the character-manipulation functions that are 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 the numeric position of a named character
- LPAD: Returns an expression left-padded to the length of *n* characters with a character expression
- RPAD: Returns an expression right-padded to the length of *n* characters with a character expression
- TRIM: Trims leading or trailing characters (or both) from a character string (If *trim_character* or *trim_source* is a character literal, you must enclose it within single quotation marks.)

Function	Result
`CONCAT('Hello', 'World')`	`HelloWorld`
`SUBSTR('HelloWorld',1,5)`	`Hello`
`LENGTH('HelloWorld')`	`10`
`INSTR('HelloWorld', 'W')`	`6`
`LPAD(salary,10,'*')`	`*****24000`
`RPAD(salary, 10, '*')`	`24000*****`
`REPLACE ('JACK and JUE','J','BL')`	`BLACK and BLUE`
`TRIM('H' FROM 'HelloWorld')`	`elloWorld`

Using the Character-Manipulation Functions

Example:

```
SELECT employee_id, CONCAT(first_name, last_name) NAME,
    job_id, LENGTH (last_name),
    INSTR(last_name, 'a') "Contains 'a'?"
FROM   employees
WHERE  SUBSTR(job_id, 4) = 'REP';
```

Output:

	EMPLOYEE_ID	NAME	JOB_ID	LENGTH(LAST_NAME)	Contains 'a'?
1	174	EllenAbel	SA_REP	4	0
2	176	JonathonTaylor	SA_REP	6	2
3	178	KimberelyGrant	SA_REP	5	3
4	202	PatFay	MK_REP	3	2

This example displays employee first names and last names joined together, the length of the employee last name, and the numeric position of the letter –all in the employee last name for all employees who have the string, REP, contained in the job ID starting at the fourth position of the job ID.

Number Functions

Number functions accept numeric input and return numeric values.

- ROUND: Rounds value to a specified decimal
- TRUNC: Truncates value to a specified decimal
- MOD: Returns remainder of division

Function	Result
`ROUND(45.926, 2)`	`45.93`
`TRUNC(45.926, 2)`	`45.92`
`MOD(1600, 300)`	`100`

Using the ROUND Function

The ROUND function rounds the column, expression, or value to *n* decimal places. If the second argument is 0 or is missing, the value is rounded to zero decimal places. If the second argument is 2, the value is rounded to two decimal places. Conversely, if the second argument is –2, the value is rounded to two decimal places to the left (rounded to the nearest unit of 100).

The ROUND function can also be used with date functions.

Example:

SELECT ROUND(45.923,2), ROUND(45.923,0), ROUND(45.923,-1)

FROM DUAL;

Output:

	ROUND(45.923,2)	ROUND(45.923,0)	ROUND(45.923,-1)
1	45.92	46	50

DUAL Table

The DUAL table is owned by the user SYS and can be accessed by all users. It contains one column, DUMMY, and one row with the value X. The DUAL table is useful when you want to return a value only once (for example, the value of a constant, pseudocolumn, or expression that is not derived from a table with user data). The DUAL table is generally used for completeness of the SELECT clause syntax, because both SELECT and FROM clauses are mandatory, and several calculations do not need to select from the actual tables.

Using the TRUNC Function

The TRUNC function truncates the column, expression, or value to *n* decimal places.

The TRUNC function works with arguments similar to those of the ROUND function. If the second argument is 0 or is missing, the value is truncated to zero decimal places. If the second argument is 2, the value is truncated to two decimal places. Conversely, if the second argument is –2, the value is truncated to two decimal places to the left. If the second argument is –1, the value is truncated to one decimal place to the left.

The TRUNC function can be used with date functions.

Example:

SELECT TRUNC(45.923,2), TRUNC(45.923),

TRUNC(45.923,-1)

FROM DUAL;

Output:

Using the MOD Function

The MOD function finds the remainder of the first argument divided by the second argument.

Example:

```
SELECT last_name, salary, MOD(salary, 5000)
FROM   employees
WHERE  job_id = 'SA_REP';
```

Output:

	LAST_NAME	SALARY	MOD(SALARY,5000)
1	Abel	11000	1000
2	Taylor	8600	3600
3	Grant	7000	2000

The above example calculates the remainder of the salary after dividing it by 5,000 for all employees whose job ID is SA_REP.

Working with Dates

The Oracle database stores dates in an internal numeric format, representing the century, year, month, day, hours, minutes, and seconds.

The default display and input format for any date is DD-MON-RR. Valid Oracle dates are between January 1, 4712 B.C., and December 31, 9999 A.D.

Example:

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date < '01-FEB-88';
```

Output:

In the above example, the HIRE_DATE column output is displayed in the default format DD-MON-RR. However, dates are not stored in the database in this format. All the components of the date and time are stored. So, although a HIRE_DATE such as 17-JUN-87 is displayed as day, month, and year, there is also *time* and *century* information associated with the date. The complete data might be June 17, 1987, 5:10:43 PM.

Using the SYSDATE Function

SYSDATE is a date function that returns the current database server date and time. You can use SYSDATE just as you would use any other column name. For example, you can display the current date by selecting SYSDATE from a table. It is customary to select SYSDATE from a dummy table called DUAL.

Example:

```
SELECT sysdate
FROM   dual;
```

Output:

	SYSDATE
1	31-MAY-07

Arithmetic with Dates

Because the database stores dates as numbers, you can perform calculations using arithmetic operators such as addition and subtraction. You can add and subtract number constants as well as dates.

Example:

```
SELECT last_name, (SYSDATE-hire_date)/7 AS WEEKS
FROM   employees
WHERE  department_id = 90;
```

Output:

	LAST_NAME	WEEKS
1	King	1041.168239087301587301587301587301587302
2	Kochhar	923.025381944444444444444444444444444444
3	De Haan	750.168239087301587301587301587301587302

The above example displays the last name and the number of weeks employed for all employees in department 90. It subtracts the date on which the employee was hired from the current date (SYSDATE) and divides the result by 7 to calculate the number of weeks that a worker has been employed.

Date-Manipulation Functions

Date functions operate on Oracle dates. All date functions return a value of the 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*

The above list is a subset of the available date functions. ROUND and TRUNC number functions can also be used to manipulate the date values as shown below:

- ROUND(*date*[,'*fmt*']): Returns *date* rounded to the unit that is 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 that is specified by the format model *fmt*. If the format model *fmt* is omitted, *date* is truncated to the nearest day.

Function	Result
MONTHS_BETWEEN ('01-SEP-95','11-JAN-94')	19.6774194
ADD_MONTHS ('31-JAN-96',1)	'29-FEB-96'
NEXT_DAY ('01-SEP-95','FRIDAY')	'08-SEP-95'
LAST_DAY ('01-FEB-95')	'28-FEB-95'

In above example, the ADD_MONTHS function adds one month to the supplied date value, ‒31-JAN-96‖ and returns ‒29-FEB-96.‖ The function recognizes the year 1996 as the leap year and hence returns the last day of the February month. If you change the input date value to ‒31-JAN-95,‖ the function returns ‒28-FEB-95.‖

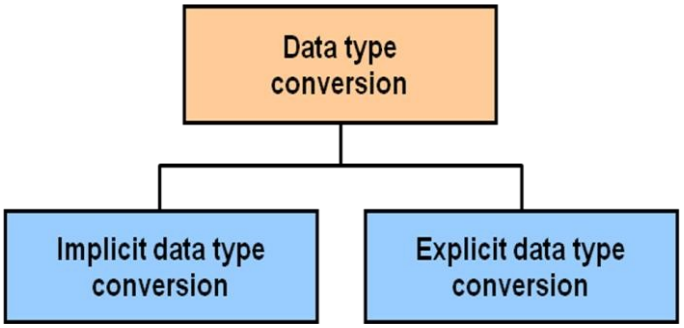
Using ROUND and TRUNC Functions with Dates

The ROUND and TRUNC functions can be used for number and date values. When used with dates, these functions round or truncate to the specified format model. Therefore, you can round dates to the nearest year or month. If the format model is month, dates 1-15 result in the first day of the current month. Dates 16 to 31 result in the first day of the next month. If the format model is year, months 1-6 result in January 1 of the current year. Months 7-12 result in January 1 of the next year.

Assume SYSDATE = '25-JUL-03':

Function	Result
ROUND(SYSDATE,'MONTH')	01-AUG-03
ROUND(SYSDATE ,'YEAR')	01-JAN-04
TRUNC(SYSDATE ,'MONTH')	01-JUL-03
TRUNC(SYSDATE ,'YEAR')	01-JAN-03

Conversion Functions



Implicit Data Type Conversion

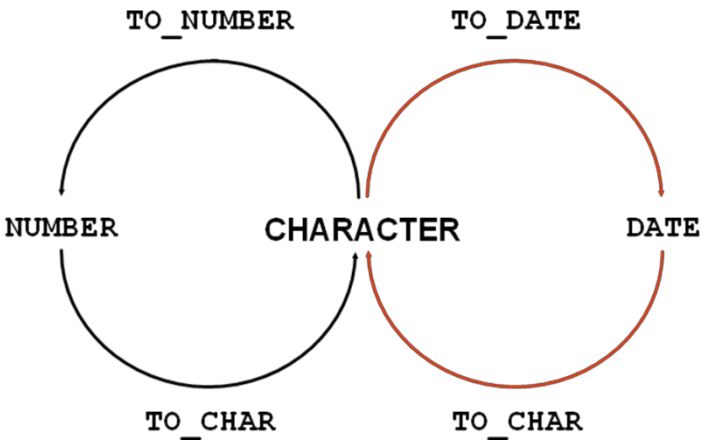
In expressions, the Oracle server can automatically convert the following:

From	To
VARCHAR2 or CHAR	NUMBER
VARCHAR2 or CHAR	DATE

For example, the expression hire_date > '01-JAN-90' results in the implicit conversion from the string '01-JAN-90' to a date. Therefore, a VARCHAR2 or CHAR value can be implicitly converted to a number or date data type in an expression.

Explicit Data Type Conversion

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



Using the TO_CHAR Function with Dates

TO_CHAR converts a datetime data type to a value of VARCHAR2 data type in the format specified by the *format_model*. A format model is a character literal that describes the format of datetime stored in a character string. For example, the datetime format model for the string '11-Nov-1999' is 'DD-Mon-YYYY'. You can use the TO_CHAR function to convert a date from its default format to the one that you specify.

```
TO_CHAR(date, 'format_model')
```

Guidelines

- The format model must be enclosed with single quotation marks and is case-sensitive.
- The format model can include any valid date format element. But be sure to separate the date value from the format model with a comma.
- The names of days and months in the output are automatically padded with blanks.
- To remove padded blanks or to suppress leading zeros, use the fill mode *fm* element.

Elements of the Date Format Model

Element	Result
YYYY	Full year in numbers
YEAR	Year spelled out (in English)
MM	Two-digit value for the month
MONTH	Full name of the month
MON	Three-letter abbreviation of the month
DY	Three-letter abbreviation of the day of the week
DAY	Full name of the day of the week
DD	Numeric day of the month

SELECT employee_id, TO_CHAR(hire_date, 'MM/YY') Month_Hired

FROM employees

WHERE last_name = 'Higgins';

Elements of the Date Format Model

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

Element	Description
AM or PM	Meridian indicator
A.M. or P.M.	Meridian indicator with periods
HH or HH12 or HH24	Hour of day, or hour (1–12), or hour (0–23)
MI	Minute (0–59)
SS	Second (0–59)
SSSSS	Seconds past midnight (0–86399)

Using the TO_CHAR Function with Dates

Example:

The SQL statement in this example displays the last names and hire dates for all the employees. The hire date appears as 17 June 1987.

SELECT last_name,

TO_CHAR(hire_date, 'fmDD Month YYYY')

AS HIREDATE

FROM employees;

Output:

	LAST_NAME	HIREDATE
1	King	17 June 1987
2	Kochhar	21 September 1989
3	De Haan	13 January 1993
4	Hunold	3 January 1990
5	Ernst	21 May 1991
6	Lorentz	7 February 1999
7	Mourgos	16 November 1999
8	Rajs	17 October 1995
9	Davies	29 January 1997
10	Matos	15 March 1998
...		
19	Higgins	7 June 1994
20	Gietz	7 June 1994

Using the TO_CHAR Function with Numbers

When working with number values, such as character strings, you should convert those numbers to the character data type using the TO_CHAR function, which translates a value of NUMBER data type to VARCHAR2 data type. This technique is especially useful with concatenation.

Number Format Elements

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

Element	Description	Example	Result
9	Numeric position (number of 9s determine display width)	999999	1234
0	Display leading zeros	099999	001234
$	Floating dollar sign	$999999	$1234
L	Floating local currency symbol	L999999	FF1234
D	Returns the decimal character in the specified position. The default is a period (.).	99D99	99.99
.	Decimal point in position specified	999999.99	1234.00
G	Returns the group separator in the specified position. You can specify multiple group separators in a number format model.	9,999	9G999
,	Comma in position specified	999,999	1,234
MI	Minus signs to right (negative values)	999999MI	1234-
PR	Parenthesize negative numbers	999999PR	<1234>
EEEE	Scientific notation (format must specify four Es)	99.999EEEE	1.234E+03
U	Returns in the specified position the "Euro" (or other) dual currency	U9999	€1234
V	Multiply by 10 *n* times (*n* = number of 9s after V)	9999V99	123400
S	Returns the negative or positive value	S9999	-1234 or +1234
B	Display zero values as blank, not 0	B9999.99	1234.00

Syntax:

```
TO_CHAR(number, 'format_model')
```

Example:

```
SELECT TO_CHAR(salary, '$99,999.00') SALARY
FROM   employees
WHERE  last_name = 'Ernst';
```

Output:

	SALARY
1	$6,000.00

- The Oracle server displays a string of number signs (#) in place of a whole number whose digits exceed the number of digits provided in the format model.
- The Oracle server rounds the stored decimal value to the number of decimal places provided in the format model.

Using 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 that you select is based on the previously demonstrated format elements.

The fx modifier specifies the exact match for the character argument and date format model of a TO_DATE function:

- Punctuation and quoted text in the character argument must exactly match (except for case) the corresponding parts of the format model.
- The character argument cannot have extra blanks. Without fx, the Oracle server ignores extra blanks.
- Numeric data in the character argument must have the same number of digits as the corresponding element in the format model. Without fx, the numbers in the character argument can omit leading zeros.

Syntax:

```
TO_NUMBER(char[, 'format_model'])
```

Syntax:

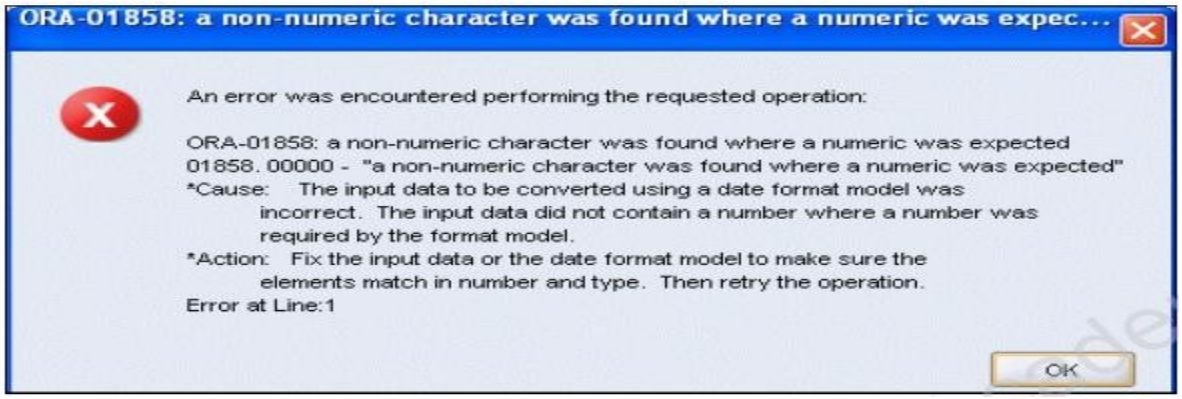
```
TO_DATE(char[, 'format_model'])
```

Example:

Display the name and hire date for all employees who started on May 24, 1999. There are two spaces after the month *May* and before the number *24* in the following example. Because the fx modifier is used, an exact match is required and the spaces after the word *May* are not recognized:

```
SELECT last_name, hire_date
FROM   employees
WHERE  hire_date = TO_DATE('May  24, 1999', 'fxMonth DD, YYYY');
```

The resulting error output looks like this:



Using the TO_CHAR and TO_DATE Function with RR Date Format

Example:

To find employees who were hired before 1990, the RR format can be used. Because the current year is greater than 1999, the RR format interprets the year portion of the date from 1950 to 1999.

```
SELECT last_name, TO_CHAR(hire_date, 'DD-Mon-YYYY')
FROM  employees
WHERE hire_date < TO_DATE('01-Jan-90','DD-Mon-RR');
```

Output:

	LAST_NAME	TO_CHAR(HIRE_DATE,'DD-MON-YYYY')
1	King	17-Jun-1987
2	Kochhar	21-Sep-1989
3	Whalen	17-Sep-1987

Nesting Functions

Single-row functions can be nested to any depth. Nested functions are evaluated from the innermost level to the outermost level.

Example:

```
SELECT last_name,
  UPPER(CONCAT(SUBSTR (LAST_NAME, 1, 8), '_US'))
FROM   employees
WHERE  department_id = 60;
```

This example displays the last names of employees in department 60. The evaluation of the SQL statement involves three steps:

1. The inner function retrieves the first eight characters of the last name.

Result1 = SUBSTR (LAST_NAME, 1, 8)

2. The outer function concatenates the result with _US.

Result2 = CONCAT(Result1, '_US')

3. The outermost function converts the results to uppercase.

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

Output:

	LAST_NAME	UPPER(CONCAT(SUBSTR(LAST_NAME,1,8),'_US'))
1	Hunold	HUNOLD_US
2	Ernst	ERNST_US
3	Lorentz	LORENTZ_US

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;
```

General Functions

The following functions work with any data type and pertain to using nulls:

- NVL (expr1, expr2)
- NVL2 (expr1, expr2, expr3)
- NULLIF (expr1, expr2)
- COALESCE (expr1, expr2, ..., exprn)

NVL Function

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

Syntax

NVL (*expr1*, *expr2*)

In the syntax:

- *expr1* is the source value or expression that may contain a null
- *expr2* is the target value for converting the null

You can use the NVL function to convert any data type, but the return value is always the same as the data type of *expr1*.

NVL Conversions for Various Data Types

Data Type	Conversion Example
NUMBER	NVL(*number_column*,9)
DATE	NVL(*date_column*, '01-JAN-95')
CHAR or VARCHAR2	NVL(*character_column*, 'Unavailable')

Using the NVL Function

Example:

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

```
SELECT last_name, salary, NVL(commission_pct, 0),
  (salary*12) + (salary*12*NVL(commission_pct, 0)) AN_SAL
FROM employees;
```

Output:

	LAST_NAME	SALARY	NVL(COMMISSION_PCT,0)	AN_SAL
1	King	24000	0	288000
2	Kochhar	17000	0	204000
3	De Haan	17000	0	204000
4	Hunold	9000	0	108000
5	Ernst	6000	0	72000
6	Lorentz	4200	0	50400
7	Mourgos	5800	0	69600
8	Rajs	3500	0	42000
9	Davies	3100	0	37200
10	Matos	2600	0	31200
11	Vargas	2500	0	30000
12	Zlotkey	10500	0.2	151200

. . .

Using the NVL2 Function

The NVL2 function examines the first expression. If the first expression is not null, then the NVL2 function returns the second expression. If the first expression is null, then the third expression is returned.

Syntax

NVL2(*expr1*, *expr2*, *expr3*)

In the syntax:

- *expr1* is the source value or expression that may contain a null
- *expr2* is the value that is returned if *expr1* is not null
- *expr3* is the value that is returned if *expr1* is null

Example:

In this example, the COMMISSION_PCT column is examined. If a value is detected, the text literal value of SAL+COMM is returned. If the COMMISSION_PCT column contains a null value, the text literal value of SAL is returned.

```
SELECT last_name,  salary, commission_pct,
    NVL2(commission_pct,
        'SAL+COMM', 'SAL') income
```

FROM employees WHERE department_id IN (50, 80);

Output:

	LAST_NAME	SALARY	COMMISSION_PCT	INCOME
1	Mourgos	5800	(null)	SAL
2	Rajs	3500	(null)	SAL
3	Davies	3100	(null)	SAL
4	Matos	2600	(null)	SAL
5	Vargas	2500	(null)	SAL
6	Zlotkey	10500	0.2	SAL+COMM
7	Abel	11000	0.3	SAL+COMM
8	Taylor	8600	0.2	SAL+COMM

Using the NULLIF Function

The NULLIF function compares two expressions. If they are equal, the function returns a null. If they are not equal, the function returns the first expression. However, you cannot specify the literal NULL for the first expression.

Syntax

NULLIF (*expr1*, *expr2*)

In the syntax:

- *NULLIF compares expr1 and expr2. If they are equal, then the function returns null. If they are not, then the function returns expr1. However, you cannot specify the literal NULL for expr1.*

Example:

In this example shown, the length of the first name in the EMPLOYEES table is compared to the length of the last name in the EMPLOYEES table. When the lengths of the names are equal, a null value is displayed. When the lengths of the names are not equal, the length of the first name is displayed.

```
SELECT first_name, LENGTH(first_name) "expr1", last_name,
    LENGTH(last_name)  "expr2",
    NULLIF(LENGTH(first_name), LENGTH(last_name)) result
```

FROM employees;

Output:

	FIRST_NAME	expr1	LAST_NAME	expr2	RESULT
1	Ellen	5	Abel	4	5
2	Curtis	6	Davies	6	(null)
3	Lex	3	De Haan	7	3
4	Bruce	5	Ernst	5	(null)
5	Pat	3	Fay	3	(null)
6	William	7	Gietz	5	7
7	Kimberely	9	Grant	5	9

. . .

19	Jennifer	8	Whalen	6	8
20	Eleni	5	Zlotkey	7	5

Using the COALESCE Function

The COALESCE function returns the first non-null expression in the list.

Syntax

COALESCE (*expr1*, *expr2, ... exprn*)

In the syntax:

- *expr1* returns this expression if it is not null
- *expr2* returns this expression if the first expression is null and this expression is not null
- *exprn* returns this expression if the preceding expressions are null

Note that all expressions must be of the same data type.

Example:

In this example shown, if the manager_id value is not null, it is displayed. If the manager_id value is null, then the commission_pct is displayed. If the manager_id and commission_pct values are null, then –No commission and no manager‖ is displayed.

```
SELECT last_name, employee_id,
COALESCE(TO_CHAR(commission_pct),TO_CHAR(manager_id),
        'No commission and no manager')
FROM employees;
```

Output:

	LAST_NAME	EMPLOYEE_ID	COALESCE(TO_CHAR(COM
1	King	100	No commission and no manager
2	Kochhar	101	100
3	De Haan	102	100
4	Hunold	103	102
5	Ernst	104	103
6	Lorentz	107	103
7	Mourgos	124	100
8	Rajs	141	124

...

12	Zlotkey	149	.2
13	Abel	174	.3
14	Taylor	176	.2
15	Grant	178	.15
16	Whalen	200	101

...

Conditional Expressions

The two methods that are used to implement conditional processing (IF-THEN-ELSE logic) in a SQL statement are the CASE expression and the DECODE function.

CASE Expression

CASE expressions allow you to use the IF-THEN-ELSE logic in SQL statements without having to invoke procedures.

In a simple CASE expression, the Oracle server searches for the first WHEN ... THEN pair for which expr is equal to comparison_expr and returns return_expr. If none of the WHEN ... THEN pairs meet this condition, and if an ELSE clause exists, then the Oracle server returns else_expr. Otherwise, the Oracle server returns a null. You cannot specify the literal NULL for all the return_exprs and the else_expr.

All of the expressions (expr, comparison_expr, and return_expr) must be of the same data type, which can be CHAR, VARCHAR2, NCHAR, or NVARCHAR2.

Syntax:

Using the CASE Expression

Example:

```
SELECT last_name, job_id, salary,
    CASE job_id WHEN 'IT_PROG'  THEN  1.10*salary
            WHEN 'ST_CLERK' THEN  1.15*salary
            WHEN 'SA_REP'   THEN  1.20*salary
    ELSE     salary END     "REVISED_SALARY"
FROM   employees;
```

Output:

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...				
5	Ernst	IT_PROG	6000	6600
6	Lorentz	IT_PROG	4200	4620
7	Mourgos	ST_MAN	5800	5800
8	Rajs	ST_CLERK	3500	4025
9	Davies	ST_CLERK	3100	3565
...				
13	Abel	SA_REP	11000	13200
14	Taylor	SA_REP	8600	10320
...				

In the SQL statement in the above, the value of JOB_ID is decoded. If JOB_ID is IT_PROG, the salary increase is 10%; if JOB_ID is ST_CLERK, the salary increase is 15%; if JOB_ID is SA_REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be written with the DECODE function.

Example:

The following code is an example of the searched CASE expression. In a searched CASE expression, the search occurs from left to right until an occurrence of the listed condition is found, and then it returns the return expression. If no condition is found to be true, and if an ELSE clause exists, the return expression in the ELSE clause is returned; otherwise, a NULL is returned.

```
SELECT last_name,salary,
(CASE WHEN salary<5000 THEN 'Low'
    WHEN salary<10000 THEN 'Medium'
    WHEN salary<20000 THEN 'Good'
    ELSE 'Excellent'
```

END) qualified_salary

FROM employees;

DECODE Function

The DECODE function decodes an expression in a way similar to the IF-THEN-ELSE logic that is used in various languages. The DECODE function decodes *expression* after comparing it to each *search* value. If the expression is the same as *search*, *result* is returned.

If the default value is omitted, a null value is returned where a search value does not match any of the result values**.**

Syntax:

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

Using the DECODE Function

Example:

```
SELECT last_name, job_id, salary,
    DECODE(job_id, 'IT_PROG',  1.10*salary,
                   'ST_CLERK', 1.15*salary,
                   'SA_REP',   1.20*salary,
           salary)
    REVISED_SALARY
FROM   employees;
```

Output:

	LAST_NAME	JOB_ID	SALARY	REVISED_SALARY
...				
6	Lorentz	IT_PROG	4200	4620
7	Mourgos	ST_MAN	5800	5800
8	Rajs	ST_CLERK	3500	4025
...				
13	Abel	SA_REP	11000	13200
14	Taylor	SA_REP	8600	10320
...				

In the SQL statement in the above example, the value of JOB_ID is tested. If JOB_ID is IT_PROG, the salary increase is 10%; if JOB_ID is ST_CLERK, the salary increase is 15%; if JOB_ID is SA_REP, the salary increase is 20%. For all other job roles, there is no increase in salary.

The same statement can be expressed in pseudocode as an IF-THEN-ELSE statement:

IF job_id = 'IT_PROG' THEN salary = salary*1.10

IF job_id = 'ST_CLERK' THEN salary = salary*1.15

IF job_id = 'SA_REP' THEN salary = salary*1.20

ELSE salary = salary

Example:

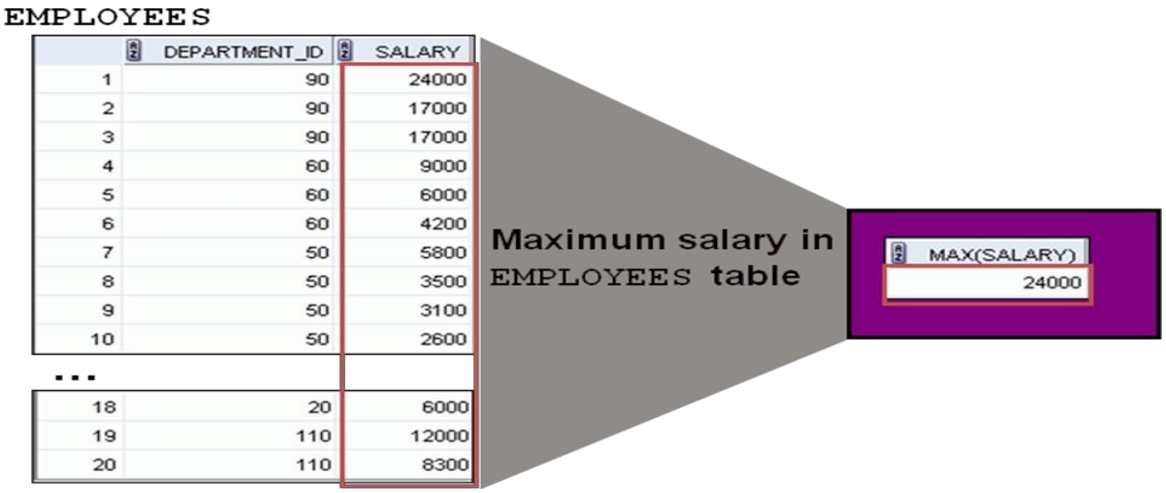
This slide shows another example using the DECODE function. In this example, you determine the tax rate for each employee in department 80 based on the monthly salary. The tax rates are as follows:

Monthly Salary Range	***Tax Rate***
$0.00–1,999.99	00%
$2,000.00–3,999.99	09%
$4,000.00–5,999.99	20%
$6,000.00–7,999.99	30%
$8,000.00–9,999.99	40%
$10,000.00–11,999.99	42%
$12,200.00–13,999.99	44%
$14,000.00 or greater	45%

```
SELECT last_name, salary,
    DECODE (TRUNC(salary/2000, 0),
                0, 0.00,
                1, 0.09,
                2, 0.20,
                3, 0.30,
                4, 0.40,
                5, 0.42,
                6, 0.44,
                  0.45) TAX_RATE
FROM   employees
WHERE  department_id = 80;
```

What Are Group Functions?

Unlike single-row functions, group functions operate on sets of rows to give one result per group. These sets may comprise the entire table or the table split into groups.



Types of Group Functions

Each of the functions accepts an argument. Following are the types of group functions.

- AVG
- COUNT
- MAX
- MIN
- STDDEV

- SUM
- VARIANCE

Group Functions: Syntax

The group function is placed after the SELECT keyword. You may have multiple group functions separated by commas.

```
SELECT      group_function(column), ...
FROM        table
[WHERE      condition]
[ORDER BY   column];
```

Guidelines for using the group functions:

- DISTINCT makes the function consider only nonduplicate values; ALL makes it consider every value, including duplicates. The default is ALL and therefore does not need to be specified.
- The data types for the functions with an expr argument may be CHAR, VARCHAR2, NUMBER, or DATE.
- All group functions ignore null values. To substitute a value for null values, use the NVL, NVL2, or COALESCE functions.

Using the AVG and SUM Functions

You can use the AVG, SUM, MIN, and MAX functions against the columns that can store numeric data. The following example displays the average, highest, lowest, and sum of monthly salaries for all sales representatives.

Example:

```
SELECT AVG(salary), MAX(salary),
       MIN(salary), SUM(salary)
FROM   employees
WHERE  job_id LIKE '%REP%';
```

Output:

Using the MIN and MAX Functions

You can use the MAX and MIN functions for numeric, character, and date data types.

Example:

```
SELECT MIN(hire_date), MAX(hire_date)
FROM employees;
```

Output:

	MIN(HIRE_DATE)	MAX(HIRE_DATE)
1	17-JUN-87	29-JAN-00

The above example displays the most junior and most senior employees.

Example:

The following example displays the employee last name that is first and the employee last name that is last in an alphabetic list of all employees:

```
SELECT MIN(last_name), MAX(last_name)
FROM   employees;
```

Using the COUNT Function

The COUNT function has three formats:

- COUNT(*)
- COUNT(*expr*)
- COUNT(DISTINCT *expr*)

COUNT(*) returns the number of rows in a table that satisfy the criteria of the SELECT statement, including duplicate rows and rows containing null values in any of the columns. If a WHERE clause is included in the SELECT statement, COUNT(*) returns the number of rows that satisfy the condition in the WHERE clause.

In contrast, COUNT(*expr)* returns the number of non-null values that are in the column identified by *expr*.

COUNT(DISTINCT *expr*) returns the number of unique, non-null values that are in the column identified by *expr*.

Example 1:

The following example displays the number of employees in department 50.

SELECT COUNT(*)

FROM employees

WHERE department_id = 50;

Output:

	COUNT(*)
1	5

Example 2:

The following example displays the number of employees in department 80 who can earn a commission.

SELECT COUNT(commission_pct)

FROM employees

WHERE department_id = 80;

Output:

	COUNT(COMMISSION_PCT)
1	3

Using the DISTINCT Keyword

Use the DISTINCT keyword to suppress the counting of any duplicate values in a column.

Example:

The following example displays the number of distinct department values that are in the EMPLOYEES table.

SELECT COUNT(DISTINCT department_id)

FROM employees;

Output:

Group Functions and Null Values

All group functions ignore null values in the column.

However, the NVL function forces group functions to include null values.

Examples:

1. The average is calculated based on *only* those rows in the table in which a valid value is stored in the COMMISSION_PCT column. The average is calculated as the total commission that is paid to all employees divided by the number of employees receiving a commission (four).

2. The average is calculated based on *all* rows in the table, regardless of whether null values are stored in the COMMISSION_PCT column. The average is calculated as the total commission that is paid to all employees divided by the total number of employees in the company (20).

SELECT AVG(commission_pct)

FROM employees;

Output:

	AVG(COMMISSION_PCT)
1	0.2125

Example:

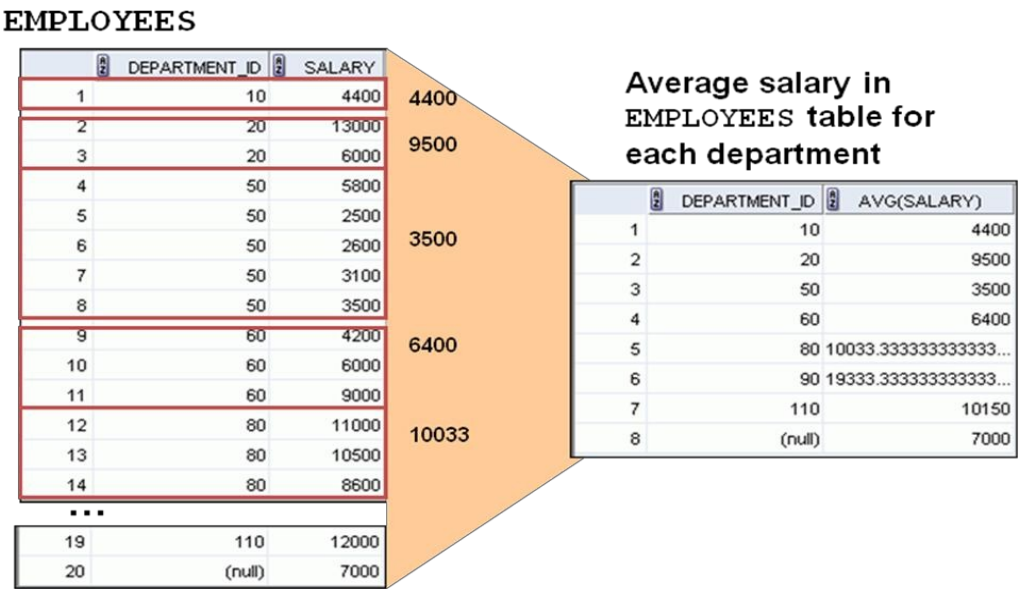
SELECT AVG(NVL(commission_pct, 0))

FROM employees;

Output:

	AVG(NVL(COMMISSION_PCT,0))
1	0.0425

Creating Groups of Data



GROUP BY Clause Syntax

You can use the GROUP BY clause to divide the rows in a table into groups. You can then use the group functions to return summary information for each group.

Syntax:

```
SELECT     column, group_function(column)
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[ORDER BY  column];
```

In the syntax:

group_by_expression specifies columns whose values determine the basis for grouping rows

Guidelines

- If you include a group function in a SELECT clause, you cannot select individual results as well, *unless* the individual column appears in the GROUP BY clause. You receive an error message if you fail to include the column list in the GROUP BY clause.
- Using a WHERE clause, you can exclude rows before dividing them into groups.
- You must include the *columns* in the GROUP BY clause.
- You cannot use a column alias in the GROUP BY clause.

Using the GROUP BY Clause

When using the GROUP BY clause, make sure that all columns in the SELECT list that are not group functions are included in the GROUP BY clause.

Example:

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id;
```

Output:

	DEPARTMENT_ID	AVG(SALARY)
1	(null)	7000
2	90	19333.33333333333...
3	20	9500
4	110	10150
5	50	3500
6	80	10033.33333333333...
7	60	6400
8	10	4400

The above example displays the department number and the average salary for each department. Here is how this SELECT statement, containing a GROUP BY clause, is evaluated:

- The SELECT clause specifies the columns to be retrieved, as follows:

  -Department number column in the EMPLOYEES table

  -The average of all salaries in the group that you specified in the GROUP BY clause
- The FROM clause specifies the tables that the database must access: the EMPLOYEES table
- The WHERE clause specifies the rows to be retrieved. Because there is no WHERE clause, all rows are retrieved by default.
- The GROUP BY clause specifies how the rows should be grouped. The rows are grouped by department number, so the AVG function that is applied to the salary column calculates the average salary for each department.

The GROUP BY column does not have to be in the SELECT clause. For example, the SELECT statement in the slide displays the average salaries for each department without displaying the respective department numbers. Without the department numbers, however, the results do not look meaningful.

You can also use the group function in the ORDER BY clause:

```
SELECT   department_id, AVG(salary)
FROM     employees
GROUP BY department_id
ORDER BY AVG(salary);
```

Using the Group By Clause on Multiple Columns

You can return summary results for groups and subgroups by listing multiple GROUP BY columns. The GROUP BY clause groups rows but does not guarantee the order of the result set. To order the groupings, use the ORDER BY clause.

Example:

```
SELECT   department_id, job_id, SUM(salary)
FROM     employees
WHERE          department_id > 40
GROUP BY department_id, job_id
ORDER BY department_id;
```

Output:

	DEPARTMENT_ID	JOB_ID	SUM(SALARY)
1	50	ST_CLERK	11700
2	50	ST_MAN	5800
3	60	IT_PROG	19200
4	80	SA_MAN	10500
5	80	SA_REP	19600
6	90	AD_PRES	24000
7	90	AD_VP	34000
8	110	AC_ACCOUNT	8300
9	110	AC_MGR	12000

In the above example, the SELECT statement that contains a GROUP BY clause is evaluated as follows:

- The SELECT clause specifies the column to be retrieved:
  - Department ID in the EMPLOYEES table
  - Job ID in the EMPLOYEES table

- The sum of all salaries in the group that you specified in the GROUP BY clause

- The FROM clause specifies the tables that the database must access: the EMPLOYEES table.
- The WHERE clause reduces the result set to those rows where department ID is greater than 40.
- The GROUP BY clause specifies how you must group the resulting rows:
  - First, the rows are grouped by the department ID.
  - Second, the rows are grouped by job ID in the department ID groups.
- The ORDER BY clause sorts the results by department ID.

Illegal Queries Using Group Functions

Whenever you use a mixture of individual items (DEPARTMENT_ID) and group functions (COUNT) in the same SELECT statement, you must include a GROUP BY clause that specifies the individual items (in this case, DEPARTMENT_ID). If the GROUP BY clause is missing, then the error message –not a single-group group function‖ appears and an asterisk (*) points to the offending column. You can correct the error in the first example in the slide by adding the GROUP BY clause:

Examples:

```
SELECT department_id, COUNT(last_name)
FROM   employees;
```

ORA-00937: not a single-group group function
00937. 00000 - "not a single-group group function"

A GROUP BY clause must be added to count the last names for each department_id.

SELECT list.

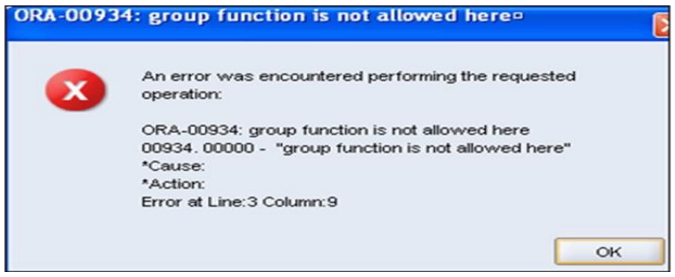
Illegal Queries Using Group Functions (continued)

The WHERE clause cannot be used to restrict groups. The SELECT statement in the following example results in an error because it uses the WHERE clause to restrict the display of the average salaries of those departments that have an average salary greater than $8,000.

Example:

```
SELECT   department_id, AVG(salary)
FROM     employees
WHERE    AVG(salary) > 8000
GROUP BY department_id;
```

Output:



Cannot use the WHERE clause to restrict groups

However, you can correct the error in the example by using the HAVING clause to restrict groups:

Restricting Group Results

You use the HAVING clause to restrict groups in the same way that you use the WHERE clause to restrict the rows that you select.

Syntax:

```
SELECT     column, group_function
FROM       table
[WHERE     condition]
[GROUP BY  group_by_expression]
[HAVING    group_condition]
[ORDER BY  column];
```

In the syntax, *group_condition* restricts the groups of rows returned to those groups for which the specified condition is true.

Using the HAVING Clause

The following example displays the department numbers and maximum salaries for those departments with a maximum salary greater than $10,000.

Example:

```
SELECT   department_id, MAX(salary)
FROM     employees
GROUP BY department_id
HAVING   MAX(salary)>10000 ;
```

Output:

	DEPARTMENT_ID	MAX(SALARY)
1	90	24000
2	20	13000
3	110	12000
4	80	11000

Example:

```
SELECT   job_id, SUM(salary) PAYROLL
FROM     employees
WHERE    job_id NOT LIKE '%REP%'
GROUP BY job_id
HAVING   SUM(salary) > 13000
ORDER BY SUM(salary);
```

Output:

	JOB_ID	PAYROLL
1	IT_PROG	19200
2	AD_PRES	24000
3	AD_VP	34000

The above example displays the job ID and total monthly salary for each job that has a total payroll exceeding $13,000. The example excludes sales representatives and sorts the list by the total monthly salary.

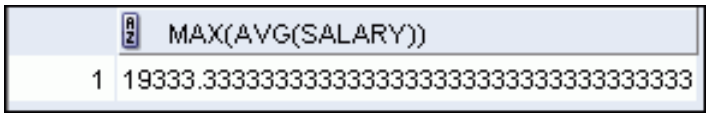
Nesting Group Functions

Group functions can be nested to a depth of two functions. The following example calculates the average salary for each department_id and then displays the maximum average salary.

Example:

```
SELECT
MAX(AVG(salary))
FROM     employees
GROUP BY department_id;
```

Output:



	MAX(AVG(SALARY))
1	19333.333333333333333333333333333333333

Lab task:

The HR department needs your assistance in creating some queries.

1. The HR department needs a report to display the employee number, last name, salary, and salary increased by 15.5% (expressed as a whole number) for each employee. Label the column New Salary.
2. The HR department wants to find the duration of employment for each employee. For each employee, display the last name and calculate the number of months between today and the date on which the employee was hired. Label the column as MONTHS_WORKED. Order your results by the number of months employed. Round the number of months up to the closest whole number.
3. Create a query to display the last name and the number of weeks employed for all employees in department 90. Label the number of weeks column as TENURE. Truncate the number of weeks value to 0 decimal places. Show the records in descending order of the employee‘s tenure

4. Using the DECODE function, write a query that displays the grade of all employees based on the value of the column JOB_ID, using the following data:

Job	***Grade***
AD_PRES	A
ST_MAN	B
IT_PROG	C
SA_REP	D
ST_CLERK	E
None of the above	0

5. Write a query to display the number of people with the same job

6. Determine the number of managers without listing them. Label the column as Number of Managers. *Hint: Use the MANAGER_ID column to determine the number of managers*

7. Create a report to display the manager number and the salary of the lowest-paid employee for that manager. Exclude anyone whose manager is not known. Exclude any groups where the minimum salary is $6,000 or less. Sort the output in descending order of salary

8. Create a query to display the total number of employees and, of that total, the number of employees hired in 1995, 1996, 1997, and 1998. Create appropriate column headings