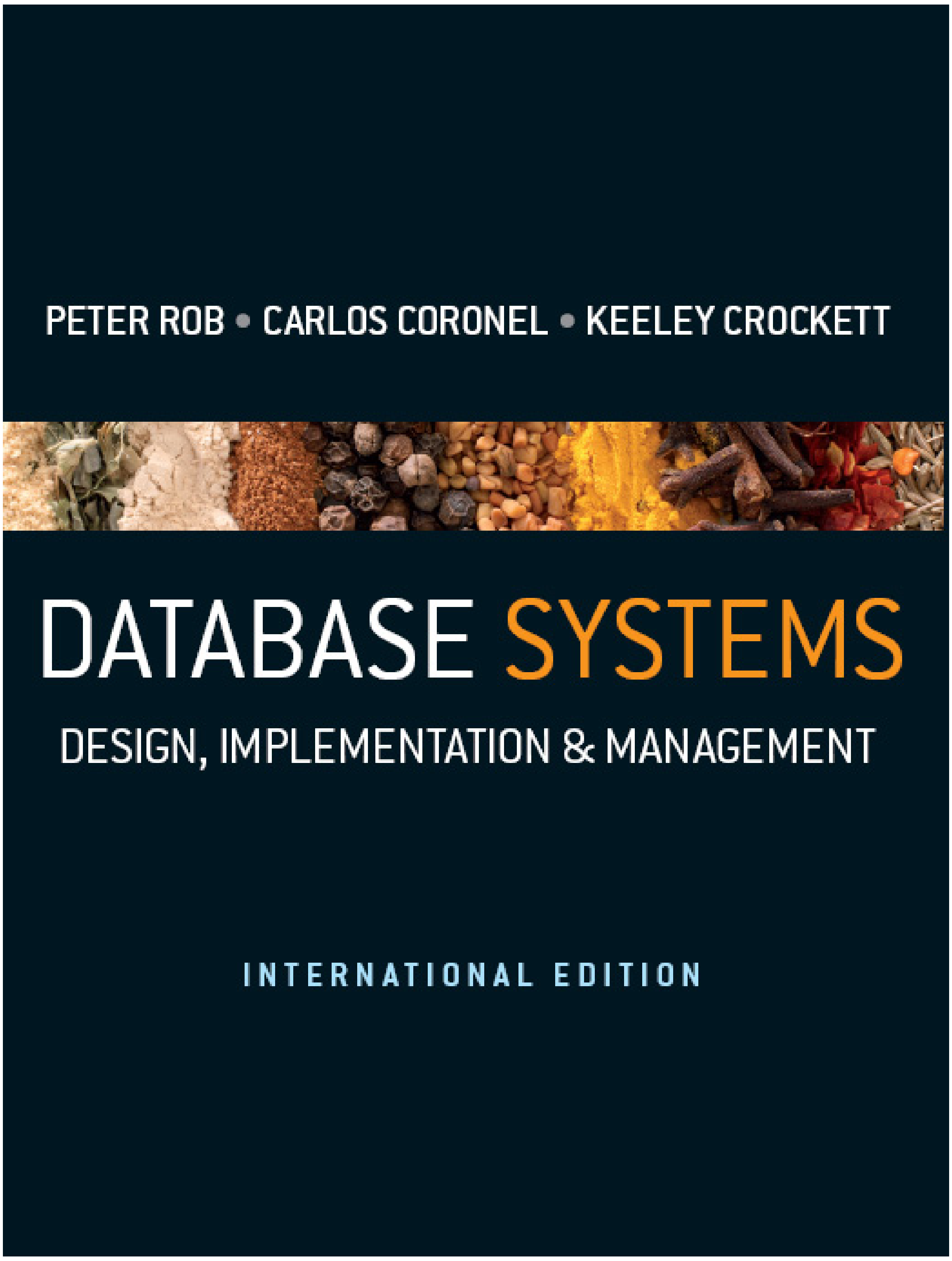
COURSE TECHNOLOGY
CENGAGE Learning™


PETER ROB • CARLOS CORONEL • KEELEY CROCKETT
DATABASE SYSTEMS
DESIGN, IMPLEMENTATION & MANAGEMENT
INTERNATIONAL EDITION

ORACLE 10g Lab Guide

A supplement to: *Database Systems: Design, Implementation and Management (International Edition)*
Rob, Coronel & Crockett (ISBN: 9781844807321)

Table of Contents

Introduction to the ORACLE 10g Lab Guide

This lab guide is designed to provide examples and exercises in the fundamentals of SQL within the ORACLE 10g environment. The objective is not to develop full blown applications but to illustrate the concepts of SQL using simple examples. The lab guide has been divided up into 10 sessions. Each one comprises of examples, tasks and exercises about a particular concept in SQL and how it is implemented in ORACLE 10g. On completion of this 10 week lab guide you will be able to:

- Create a simple relational database in ORACLE 10g
- Insert, update and delete data the tables
- Create queries using basic and advanced SELECT statements
- Perform join operations on relational tables
- Apply set operators
- Use aggregate functions in SQL
- Write subqueries
- Create views of the database

This lab guide assumes that you know how to perform basic operations in the Microsoft Windows environment. Therefore, you should know what a folder is, how to maximize or minimize a folder, how to create a folder, how to select a file, how you maximize and minimize windows, what clicking and double-clicking indicate, how you create a folder, how you drag, how to use drag and drop, how you save a file, and so on.

The lab guide has been designed on ORACLE 10g version 10.2.0.1.0 running on Windows XP Professional. Before starting this guide, you must log on to your ORACLE RDBMS, using a user ID and a password created by your database administrator. How you connect to the ORACLE database depends on how the ORACLE software was installed on your server and on the access paths and methods defined and managed by the database administrator. Follow the instructions provided by your instructor, College or University.

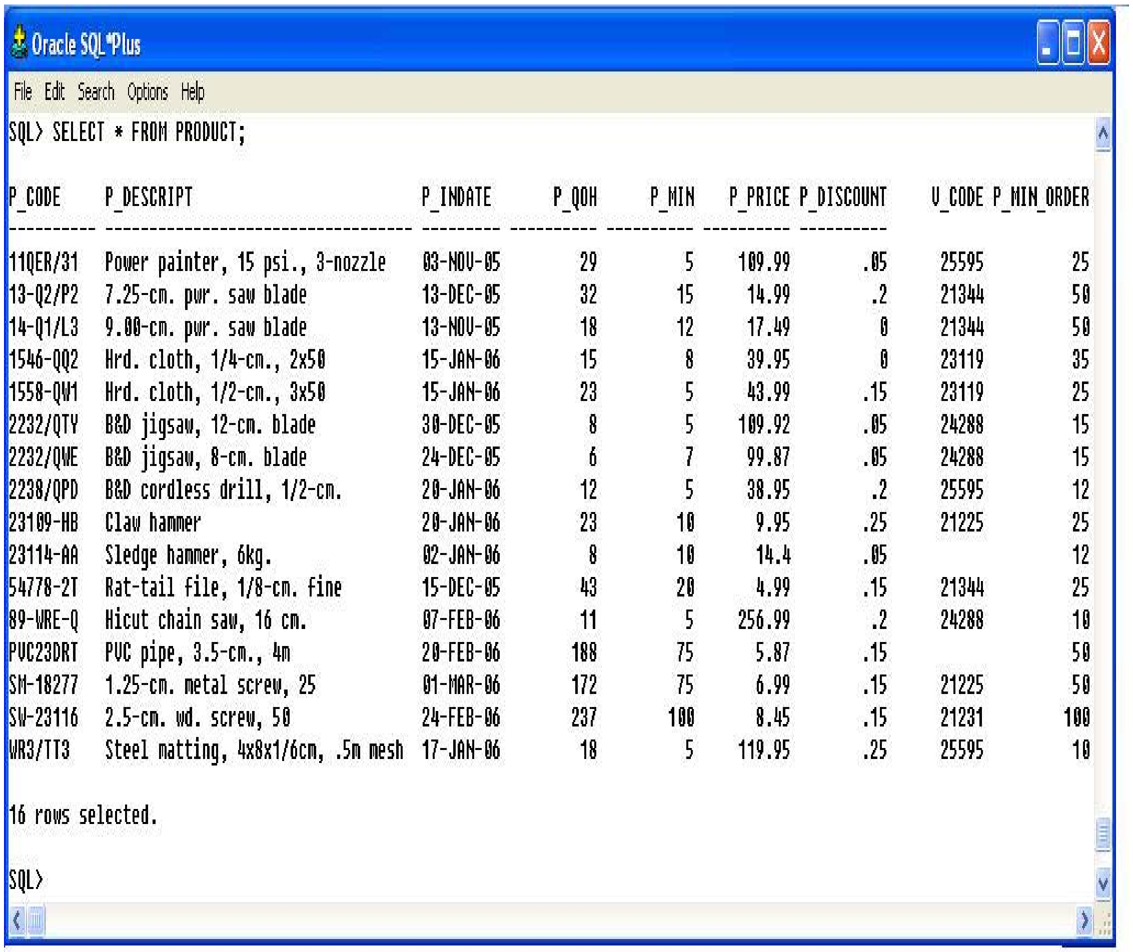
Lab 1: The ORACLE 10g DBMS interfaces

The learning objectives of this lab are to:

- Learn how to use two standard ORACLE 10g interfaces to SQL
- Learn the basic command line SQL editing commands
- Load and run database scripts in the two interfaces

1.1 Introduction

The ORACLE 10g DBMS has a number of interfaces for executing SQL queries. The most basic interface, known as the ORACLE SQL *Plus interface, is used to directly execute SQL commands such as those you will have learnt about in Chapter 8, Introduction to Structured Query Language. An example of the ORACLE SQL *Plus interface can be seen in Figure 1.



```
SQL> SELECT * FROM PRODUCT;

P_CODE     P_DESCRIPT                         P_INDATE       P_QOH      P_MIN    P_PRICE P_DISCOUNT     V_CODE P_MIN_ORDER
---------- ---------------------------------- --------- ---------- ---------- ---------- ----------
11QER/31   Power painter, 15 psi., 3-nozzle   03-NOV-05         29          5     109.99        .05      25595          25
13-Q2/P2   7.25-cm. pwr. saw blade            13-DEC-05         32         15      14.99         .2      21344          50
14-Q1/L3   9.00-cm. pwr. saw blade            13-NOV-05         18         12      17.49          0      21344          50
1546-QQ2   Hrd. cloth, 1/4-cm., 2x50          15-JAN-06         15          8      39.95          0      23119          35
1558-QW1   Hrd. cloth, 1/2-cm., 3x50          15-JAN-06         23          5      43.99        .15      23119          25
2232/QTY   B&D jigsaw, 12-cm. blade           30-DEC-05          8          5     109.92        .05      24288          15
2232/QWE   B&D jigsaw, 8-cm. blade            24-DEC-05          6          7      99.87        .05      24288          15
2238/QPD   B&D cordless drill, 1/2-cm.        20-JAN-06         12          5      38.95         .2      25595          12
23109-HB   Claw hammer                        20-JAN-06         23         10       9.95        .25      21225          25
23114-AA   Sledge hammer, 6kg.                02-JAN-06          8         10       14.4        .05                     12
54778-2T   Rat-tail file, 1/8-cm. fine        15-DEC-05         43         20       4.99        .15      21344          25
89-WRE-Q   Hicut chain saw, 16 cm.            07-FEB-06         11          5     256.99         .2      24288          10
PVC23DRT   PVC pipe, 3.5-cm., 4m              20-FEB-06        188         75       5.87        .15                     50
SM-18277   1.25-cm. metal screw, 25           01-MAR-06        172         75       6.99        .15      21225          50
SW-23116   2.5-cm. wd. screw, 50              24-FEB-06        237        100       8.45        .15      21231         100
WR3/TT3    Steel matting, 4x8x1/6cm, .5m mesh 17-JAN-06         18          5     119.95        .25      25595          10

16 rows selected.

SQL>
```

Figure 1: The ORACLE SQL *Plus interface

In Figure 1, the following SQL query has been entered at the command line:

SELECT P_CODE, P_DESCRIPT, P_INDATE, P_SALECODE

FROM PRODUCT;

Notice that a semi-colon (**;**) is needed at the end of the SQL query. This ends the SQL statement and when the enter key is pressed the query is executed. The results are displayed immediately below the query.

ORACLE 10g also has a web based interface known as *i*SQL *Plus. This interface has its own command language in addition to being able to execute any SQL statement. The main benefit of this interface is that it allows online editing of SQL statements to take place easily. You can also do some simple formatting of the query output. Figure 2 shows the ORACLE 10g *i*SQL *Plus interface.

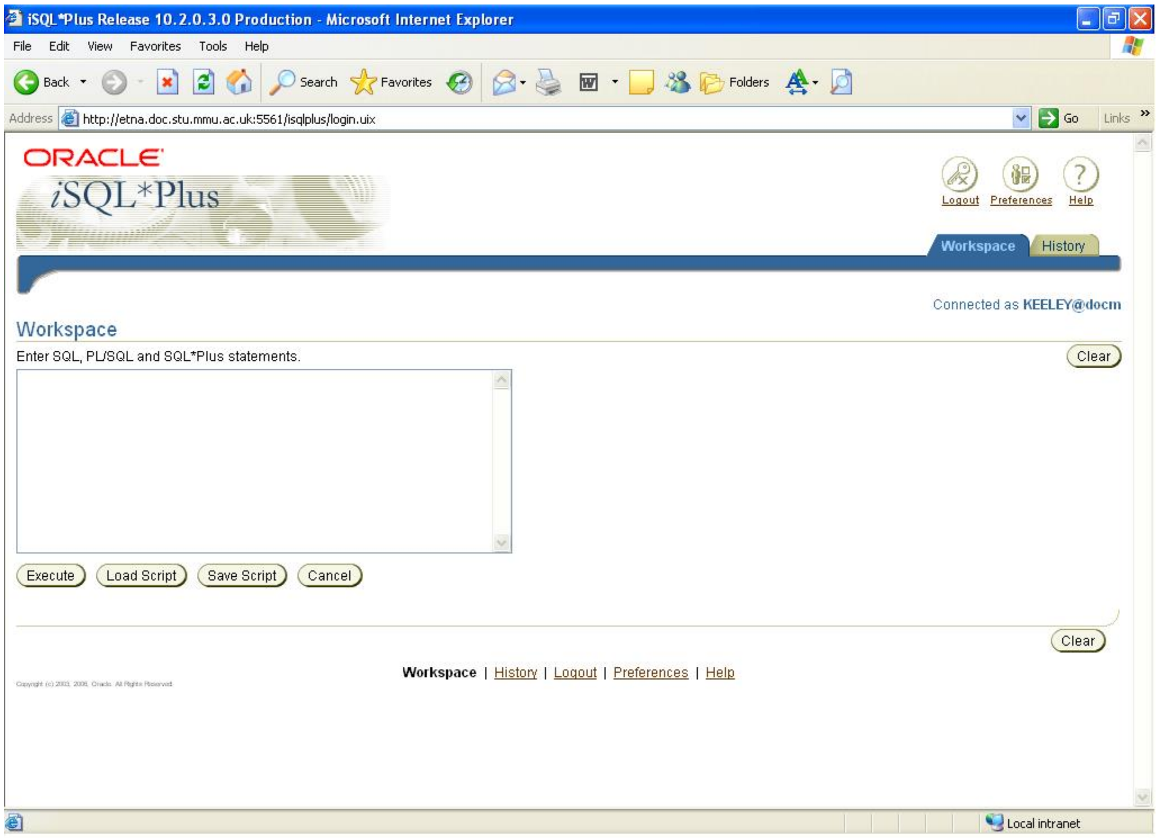

iSQL*Plus Release 10.2.0.3.0 Production - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Folders
Address http://etna.doc.stu.mmu.ac.uk:5561/isqlplus/login.uix
Go Links
ORACLE iSQL*Plus
Logout Preferences Help
Workspace History
Connected as KEELEY@docm
Workspace
Enter SQL, PL/SQL and SQL*Plus statements.
Clear
Execute Load Script Save Script Cancel
Clear
Workspace | History | Logout | Preferences | Help
Local intranet

Figure 2: The ORACLE *i*SQL *Plus interface

Which interface you use to do these lab exercises depends on how the ORACLE software was installed on your server and on the access paths and methods defined and managed by your database administrator. Follow the instructions provided by your instructor, College or University in order to start up and log into the ORACLE database before commencing any of the tasks, examples and exercises in this lab guide.

1.3 Creating Databases from script files

In this section you will learn how to create a small database called SaleCo from a script file. The SQL script file SaleCo.sql for creating the tables and loading the data in the database are located in the Student CD-ROM companion. The database design for the SaleCo database is shown in Figure 3 in the form of an Entity Relationship Diagram (ERD).

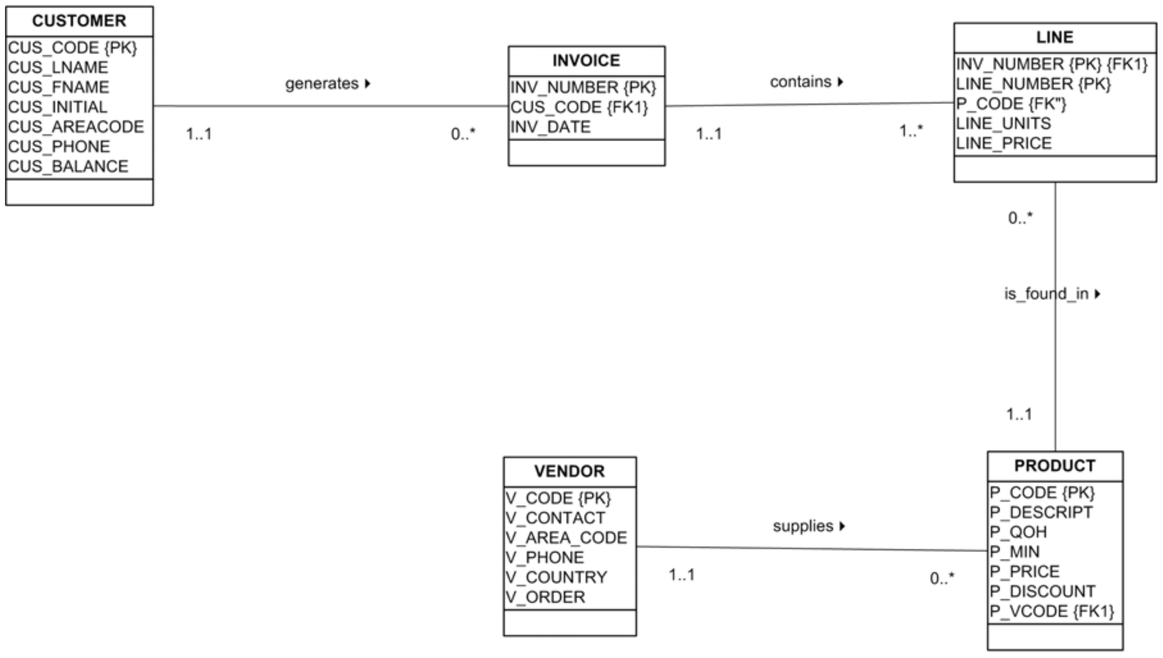


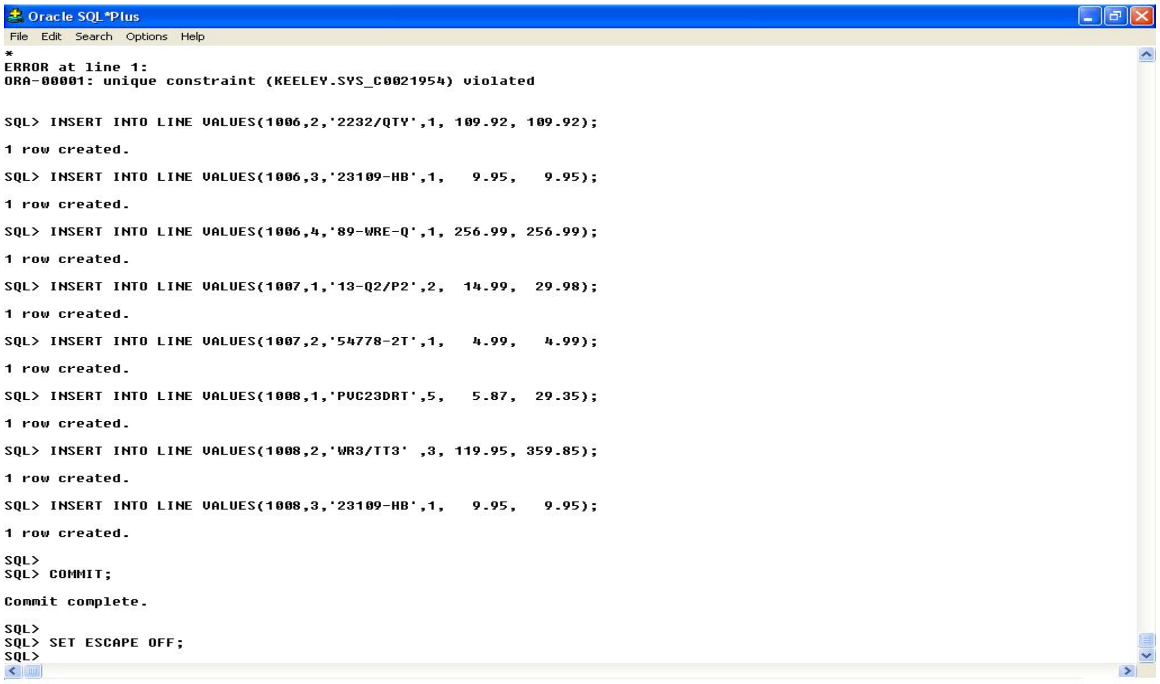
Figure 3: The SaleCo Database ERD

Task 1.1 For this task you should ensure that the script SaleCo.sql has been copied into your own working directory.

If you are using the command line ORACLE SQL *Plus interface and your own working directory is H:\ORACLE, then in order to create the ORACLE tables you would enter the following command:

SQL>@h:\Oracle\SaleCo

This will load and execute the script to create the SaleCo database. Notice that prompts will indicate that tables are being created and data added as shown in Figure 4.



```
*
ERROR at line 1:
ORA-00001: unique constraint (KEELEY.SYS_C0021954) violated

SQL> INSERT INTO LINE VALUES(1006,2,'2232/QTY',1, 109.92, 109.92);

1 row created.

SQL> INSERT INTO LINE VALUES(1006,3,'23109-HB',1,   9.95,   9.95);

1 row created.

SQL> INSERT INTO LINE VALUES(1006,4,'89-WRE-Q',1, 256.99, 256.99);

1 row created.

SQL> INSERT INTO LINE VALUES(1007,1,'13-Q2/P2',2,  14.99,  29.98);

1 row created.

SQL> INSERT INTO LINE VALUES(1007,2,'54778-2T',1,   4.99,   4.99);

1 row created.

SQL> INSERT INTO LINE VALUES(1008,1,'PVC23DRT',5,   5.87,  29.35);

1 row created.

SQL> INSERT INTO LINE VALUES(1008,2,'WR3/TT3' ,3, 119.95, 359.85);

1 row created.

SQL> INSERT INTO LINE VALUES(1008,3,'23109-HB',1,   9.95,   9.95);

1 row created.

SQL>
SQL> COMMIT;

Commit complete.

SQL>
SQL> SET ESCAPE OFF;
SQL>
```

Figure 4 Creating the SaleCo database using the command line ORACLE SQL *Plus interface

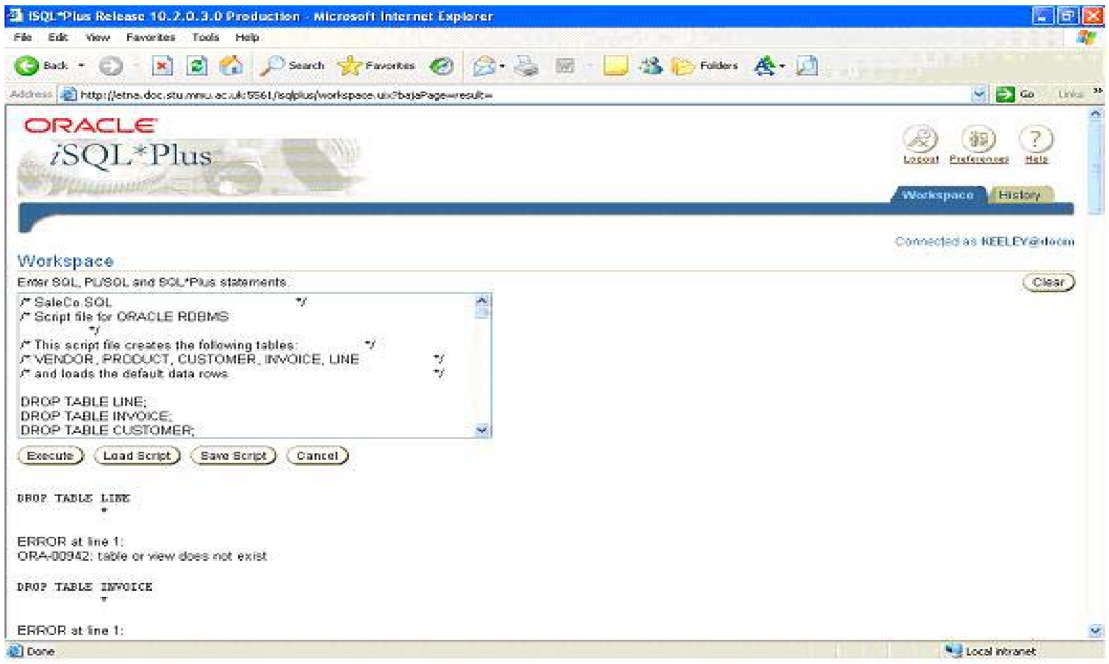
If you are using the ORACLE *i*SQL *Plus interface:

1. Use the **LOAD SCRIPT** button
2. Select **FILE BROWSE** to locate the SaleCo.sql file in your working directory

3. Then click the **LOAD** button

4. When the script has been loaded click the **EXECUTE** button

This script will display several messages stating that the different tables required have been created and that test has been inserted into them. The results can be seen in Figure 5.


iSQL*Plus Release 10.2.0.3.0 Production - Microsoft Internet Explorer
File Edit View Favorites Tools Help
http://etna.doc.stu.mmu.ac.uk:5561/isqlplus/workspace.uix?bajaPage=result=
ORACLE
iSQL*Plus
Logout Preferences Help
Workspace History
Connected as KEELEY@docm
Workspace
Enter SQL, PL/SQL and SQL*Plus statements.
Clear
Execute Load Script Save Script Cancel
Done
Local intranet

```
/* SaleCo.SQL                                    */
/* Script file for ORACLE RDBMS
              */
/* This script file creates the following tables:          */
/* VENDOR, PRODUCT, CUSTOMER, INVOICE, LINE              */
/* and loads the default data rows                       */

DROP TABLE LINE;
DROP TABLE INVOICE;
DROP TABLE CUSTOMER;
```

```
DROP TABLE LINE
           *

ERROR at line 1:
ORA-00942: table or view does not exist

DROP TABLE INVOICE
           *

ERROR at line 1:
```

Figure 5: Creating the SaleCo database using the ORACLE iSQL *Plus interface

Note

When you run the script for the first time you will see some error messages on the screen. These error messages are caused by the script attempting to DROP the database tables before they have been created. Including SQL DROP commands in a script that is being used for development is a good idea to ensure that if changes are made to the database structure, all tables are then recreated to reflect this change. If you run the script again you will see that the error messages no longer appear.

Note

Chapter 8, Introduction to Structured Query Language and Chapter 9, Advanced SQL should be studied alongside this lab guide. You can study Appendix A, Designing Databases with Visio Professional: A Tutorial, if you want to create the database design shown in Figure 3.

1.4 Command Line SQL Editing Commands

Throughout this guide we will be using the command line ORACLE SQL *Plus interface. A number of SQL commands exist in order to perform simple editing of SQL statements that are entered. SQL editing commands are entered one line at a time and are not stored in the SQL buffer. A list of SQL commands that you should become familiar with are shown below:

Command	Description
A [APPEND] *text*	Adds text to the end of the current line
C [HANGE] / *old* / *new*	Changes *old* text to *new* text in the current line
C [HANGE] / *text* /	Deletes *text* from the current line
CL [EAR] BUFF [ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
DEL *n*	Deletes line *n*
DEL *m n*	Deletes lines *m* to *n* inclusive
I [NPUT]	Inserts an indefinite number of lines
I [INPUT] text	Inserts a line consisting of text
L [IST]	Lists all lines in the SQL buffer
L [IST] *n*	Lists one line specified by n
L [IST] *m n*	Lists a range of lines *m* to *n* inclusive
R [UN]	Displays and runs the current SQL statement in the buffer
n	Specified the line to make the current line
n text	Replaces line n with *text*
0 text	Inserts a line before line 1
SAVE <filename>	Save stores the current contents of the SQL buffer in a file
START	The start command is used to run a script

Note

Many of the SQL Commands can be abbreviated to just their first letter, for example LIST can be abbreviated to L.

Note

It is important to note that these commands are not available in *iSQL*Plus. If you are using the *iSQL*Plus interface you will not be able to complete the following tasks and exercises in the rest of Lab 1 and you should proceed to Lab 2.

Task 1.2 Enter in the following SQL statement at the SQL command prompt:

```
SQL> SELECT CUS_CODE, CUS_LNAME, CUS_FNAME, CUS_PHONE, CUS_BALANCE FROM
CUSTOMER
WHERE CUS_BALANCE > 0;
```

Task 1.3 Listing commands in the buffer.

Enter the list command at the SQL prompt as shown below:

```
SQL>list
SQL> SELECT CUS_CODE, CUS_LNAME, CUS_FNAME, CUS_AREACODE, CUS_BALANCE
  2  FROM CUSTOMER
  3* WHERE CUS_BALANCE > 0
```

Notice that the semicolon you entered at the end of the SELECT command is not listed. This semicolon is necessary to indicate the end of the command when you enter it, but it is not part of the SQL command and SQL*Plus does not store it in the SQL buffer.

Task 1.4 Correcting an error in command line.

Suppose you try to select the CUS_AREACODE column but mistakenly enter it as CU_AREACODE. Enter the following command, purposely misspelling CUS_AREACODE in the first line as shown below:

```
SQL> SELECT CUS_CODE, CUS_LNAME, CU_AREACODE, CUS_BALANCE

FROM CUSTOMER

WHERE CUS_AREACODE =0181;
```

You see this message on your screen:

```
SELECT CUS_CODE, CUS_LNAME, CU_AREACODE, CUS_BALANCE
                   *
ERROR at line 1:

ORA-00904: invalid column name
```

Examine the error message; it indicates an invalid column name in line 1 of the query. The asterisk shows the point of error – the miss-typed column CUS_AREACODE.

Instead of re-entering the entire command, you can correct the mistake by editing the command in the buffer. The line containing the error is now the current line. Use the CHANGE command to correct the mistake. This command has three parts, separated by slashes or any other non-alphanumeric character:

- the word CHANGE or the letter C
- the sequence of characters you want to change
- the replacement sequence of characters

The CHANGE command finds the first occurrence in the current line of the character sequence to be changed and changes it to the new sequence. You do not need to use the CHANGE command to re-enter an entire line.

To change CU_AREACODE to CUS_AREACODE, change the line with the CHANGE command as shown below:

```
SQL> CHANGE /CU_AREACODE/CUS_AREACODE
```

The corrected line appears on your screen:

```
1* SELECT CUS_CODE, CUS_LNAME, CUS_AREACODE, CUS_BALANCE
```

Now that you have corrected the error, you can use the RUN command to run the command again and the correct result is displayed as follows:

```
CUS_CODE    CUS_LNAME    CUS_AREACODE    CUS_BALANCE

10010       Ramas        0181            0
10012       Smith        0181            345.86
10013       Olowski      0181            536.75
10015       O'Brian      0181            0
10016       Brown        0181            221.19
10017       Williams     0181            768.93
10019       Smith        0181            0
```

Task 1.5 Adding a new line.

To insert a new line after the current line you would use the INPUT command. To insert a line before line 1, enter a zero (0) and follow the zero with text. SQL*Plus inserts the line at the beginning of the buffer and all lines are renumbered starting at 1. Suppose you want to add a fourth line to the SQL query you have just modified in task 1.4. Since line 3 is already the current line, enter INPUT and press Return. SQL*Plus then prompts you for the new line:

```
SQL> INPUT
4
```

Enter the new line and then press Return.

```
SQL> 4 ORDER BY CUS_BALANCE
```

SQL*Plus prompts you again for a new line numbered ‘5’. Press Return again to indicate that you will not enter any more lines, and then use RUN to verify and re-run the query.

Task 1.6 Appending text to a line.

To add text to the end of a line in the buffer, use the APPEND command.

Use the LIST command (or the line number) to list the line you want to change.

Enter APPEND followed by the text you want to add. If the text you want to add begins with a blank, separate the word APPEND from the first character of the text by two blanks: one to separate APPEND from the text; and one to go into the buffer with the text.

For example, to append a space and the clause DESC to line 4 of the current query, you should first list the line you want to amend:

```
SQL> LIST 4
4* ORDER BY CUS_BALANCE
```

Then, enter the following command (be sure to type two spaces between APPEND and DESC):

```
SQL> APPEND  DESC
4* ORDER BY CUS_BALANCE DESC
```

Type RUN to verify the query and obtain the results shown below:

```
CUS_CODE    CUS_LNAME    CUS_AREACODE    CUS_BALANCE

10017       Williams     0181            768.93
10013       Olowski      0181            536.75
10012       Smith        0181            345.86
10016       Brown        0181            221.19
10019       Smith        0181            0
10010       Ramas        0181            0
10015       O'Brian      0181            0
```

Task 1.7 Deleting Lines.

Use the DEL command to delete lines in the SQL buffer. Enter DEL, specifying the line numbers you want to delete. Suppose you want to delete the current line to the last line inclusive. Use the DEL command as shown:

```
SQL>  DEL * LAST
```

DEL makes the following line of the buffer (if any) the current line.

Task 1.8 Saving and starting scripts.

To save the current script in the buffer use the save command as shown below:

```
SQL> SAVE SALES.sql
```

The START command retrieves a script and runs the commands it contains. Use START to run a script containing SQL commands and SQL*Plus commands. Type the START command and then the name of the file like this:

START *file_name*

ORACLE 10g, SQL*Plus assumes the file has a .SQL extension by default. To retrieve and run the command stored in SALES.SQL, enter the following:

```
SQL> START SALES
```

SQL*Plus will then run the commands in the file SALES and displays the results of the commands on your screen, formatting the query results according to the SQL*Plus commands in the file. You can also use the “at” sign (@) command to run a script like this:

```
SQL> @SALES
```

Both the @ and @@ commands list and run the commands in the specified script in the same manner as START. To see the commands as SQL*Plus "enters" them, you can **SET ECHO ON**. The ECHO system variable controls the listing of the commands in scripts run with the START, @ and @@ commands. Setting the ECHO variable OFF suppresses the listing. START, @ and @@ leave the last SQL command or PL/SQL block of the script in the buffer.

1.5 Exercises

Answer the following questions.

E1.1 Fill in the blanks.

When appending text to a line:

1) Use the [   ] command to display the line you want to change.

2) Enter [   ] followed by the text you want to add.

E1.2 You type in the following SQL query below:

```
SQL> SELECT EMP_AME, DATE_HIRED,  FROM EMPLOYEE WHERE DATE_HIRED = '01-
MAY-05';
```

You see this message on your screen:

```
SELECT    EMP_AME, DATE_HIRED
                         *
ERROR at line 1:
ORA-00904: invalid column name
```

Which SQL command would you use to correct this error?

E1.3 Is the following statement correct?

```
SQL>  DEL * LAST
```

E1.4 Assuming you have typed the following query into the buffer as shown below:

```
SQL> SELECT EMP_NO, EMP_LNAME, DOB, DATE_HIRED
2  FROM EMPLOYEE
3  WHERE SALARY>16000;
```

You then type the following SQL command:

```
SQL> L 1
```

Is the following output correct?

```
SQL> SELECT EMP_NO, EMP_LNAME, DOB, DATE_HIRED
```

E1.5 Fill in the blanks.

1. To insert a new line after the current line use the [ ] command.
2. To insert a line before line 1 enter a [ ] and follow with the text.
3. SQL*Plus then inserts the line at the beginning of the [ ] and all lines are renumbered starting at 1.

E1.6 Are the following statements True or False?

1. The @ command can be used to load and run SQL scripts in command line.
2. The LIST command shows all lines in the SQL buffer.

E1.7 Fill in blanks.

1. The CHANGE command finds the [ ] occurrence in the [ ] line of the character sequence to be changed and changes it to the new sequence.

Lab 2: Creating a database from a script file

The learning objectives of this lab are to:

- Create table structures using ORACLE data types
- Apply SQL constraints to ORACLE tables
- Create a simple index

2.1 Introduction

In this section you will learn how to create a small database called Theme Park from the ERD shown in Figure 4. This will involve you creating the table structures in ORACLE using the CREATE TABLE command. In order to do this, appropriate data types will need to be selected from the data dictionary for each table structure, along with any constraints that have been imposed (e.g. primary and foreign key). Converting any ER model to a set of tables in a database requires following specific rules that govern the conversion. The application of those rules requires an understanding of the effects of updates and deletions on the tables in the database. You can read more about these rules in Chapter 8, Introduction to Structure red Query Language, and Appendix D, Converting an ER Model into a Database Structure.

2.2 The Theme Park Database

Figure 6 shows the ERD for the Theme Park database, which will be used throughout this lab guide.

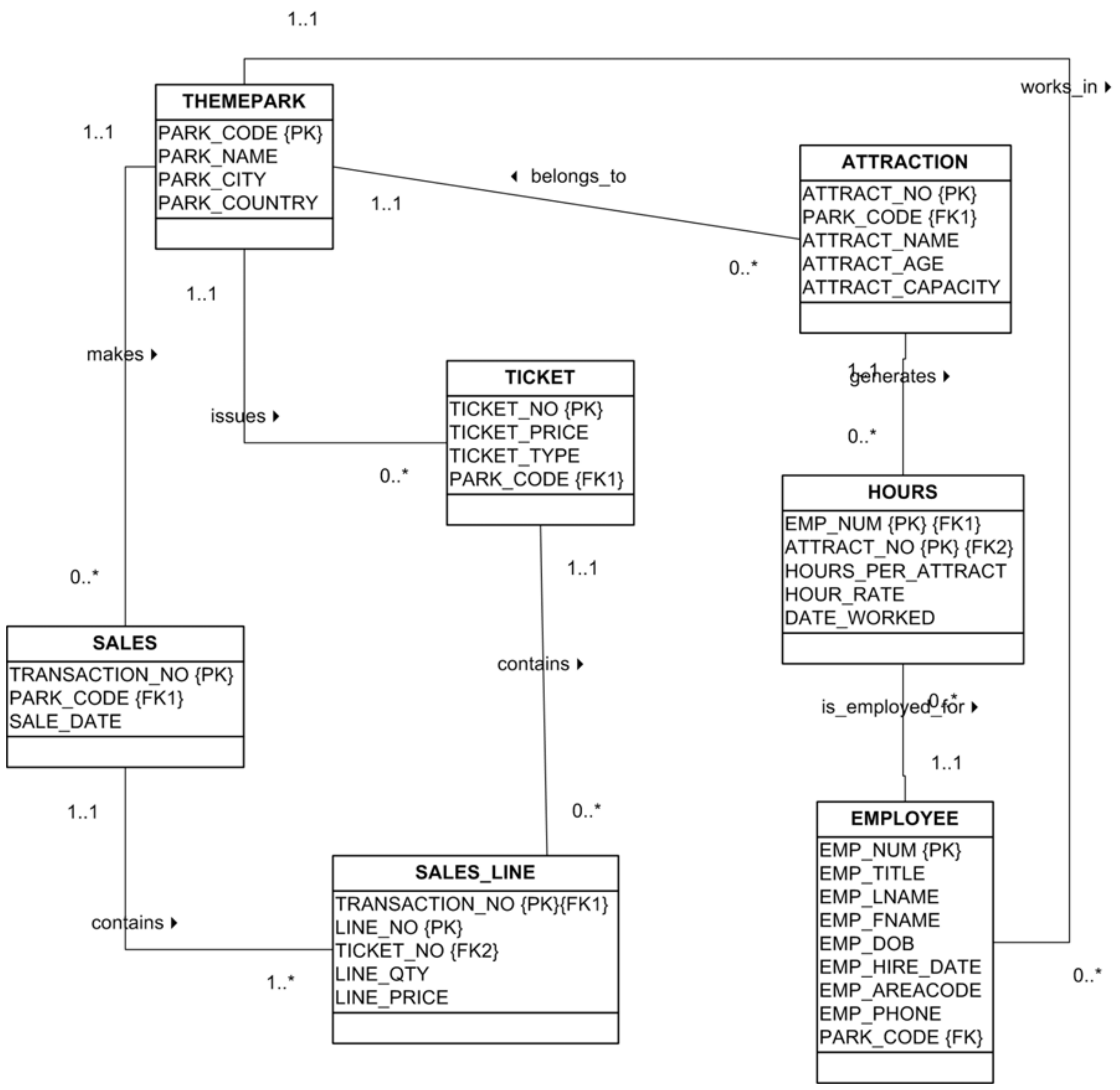


Figure 6: The Theme Park Database ERD

Table 2.1 Shows the Data Dictionary for the Theme Park database, which will be used to create each table structure.

Table 2.1 Data Dictionary for the Theme Park Database

Table Name	Attribute Name	Contents	Data Type	Format	Range	Required	PK or FK	FK Referenced Table
THEMEPARK	PARK_CODE	Park code	VARCHAR2(10)	XXXXXXXX	NA	Y	PK	
	PARK_NAME	Park Name	VARCHAR2(35)	XXXXXXXX	NA	Y		
	PARK_CITY	City	VARCHAR2(50)		NA	Y		
	PARK_COUNTRY	Country	CHAR(2)	XX	NA	Y		
EMPLOYEE	EMP_NUM	Employee number	NUMBER(4)	##	0000 – 9999	Y	PK	
	EMP_TITLE	Employee title	VARCHAR2(4)	XXXX	NA	N		
	EMP_LNAME	Last name	VARCHAR2(15)	XXXXXXXX	NA	Y		
	EMP_FNAME	First Name	VARCHAR2(15)	XXXXXXXX	NA	Y		
	EMP_DOB	Date of Birth	DATE	DD-MON-YY	NA	Y		
	EMP_HIRE_DATE	Hire date	DATE	DD-MON-YY	NA	Y		
	EMP_AREACODE	Area code	VARCHAR2(4)	XXXX	NA	Y		
	EMP_PHONE	Phone	VARCHAR2(12)	XXXXXXXX	NA	Y		
	PARK_CODE	Park code	VARCHAR2(10)	XXXXXXXX	NA	Y	FK	THEMEPARK

TICKET	TICKET_NO	Ticket number	NUMBER(10)	##########	NA	Y		
	TICKET_PRICE	Price	NUMBER(4,2)	####.##	0.00 – 0000.00			
	TICKET_TYPE	Type of ticket	VARCHAR2(10)	XXXXXXXXXX	Adult, Child,Senior,Other			
	PARK_CODE	Park code	VARCHAR2(10)	XXXXXXXX	NA	Y	FK	THEMEPARK
ATTRACTION	ATTRACT_NO	Attraction number	NUMBER(10)	##########	N/A	Y	PK	
	PARK_CODE	Park code	VARCHAR2(10)	XXXXXXXX	NA	Y	FK	THEMEPARK
	ATTRACT_NAME	Name	VARCHAR2(35)	XXXXXXX	N/A	N		
	ATTRACT_AGE	Age	NUMBER(3)	###	Default 0	Y		
	ATTRACT_CAPACITY	Capacity	NUMBER(3)	###	N/A	Y		
HOURS	EMP_NUM	Employee number	NUMBER(4)	##	0000 – 9999	Y	PK / FK	EMPLOYEE
	ATTRACT_NO	Attraction number	NUMBER(10)	##########	N/A	Y	PK / FK	ATTRACTION
	HOURS_PER_ATTRACT	Number of hours	NUMBER(2)	##	N/A	Y		
	HOUR_RATE	Hourly Rate	NUMBER(4,2)	####.##	N/A	Y		
	DATE_WORKED	Date worked	DATE	DD-MON-YY	N/A	Y		

SALES	TRANSACTION_NO	Transaction No	NUMBER	###########	N/A	Y	PK	
	PARK_CODE	Park code	VARCHAR2(10)	XXXXXXXX	NA	Y	FK	THEMEPARK
	SALE_DATE	Date of Sale	DATE	DD-MON-YY	SYSDATE	Y		
SALESLINE	TRANSACTION_NO	Transaction No	NUMBER	###########	N/A	Y	PK / FK	SALES
	LINE_NO	Line number	NUMBER(2)	##	N/A	Y		
	TICKET_NO	Ticket number	NUMBER(10)	##########	NA	Y	FK	TICKET
	LINE_QTY	Quantity	NUMBER(4)	####	N/A	Y		
	LINE_PRICE	Price of line	NUMBER(9,2)	#########.##	N/A	Y		

2.3 Creating the Table Structures

Use the following SQL commands to create the table structures for the Theme Park database. Enter each one separately to ensure that you have no errors. Successful table creation will prompt ORACLE to say “Table Created”. It is useful to store each correct table structure in a script file, in case the entire database needs to be recreated again at a later date. You can use a simple text editor such as notepad in order to do this. Save the file as themepark.sql. Note that the table-creating SQL commands used in this example are based on the data dictionary shown in Table 2.1.

As you examine each of the SQL table-creating command sequences in the following tasks, note the following features:

- The NOT NULL specifications for the attributes ensure that a data entry will be made. When it is crucial to have the data available, the NOT NULL specification will not allow the end user to leave the attribute empty (with no data entry at all).
- The UNIQUE specification creates a unique index in the respective attribute. Use it to avoid duplicated values in a column.
- The primary key attributes contain both a NOT NULL and a UNIQUE specification. Those specifications enforce the entity integrity requirements. If the NOT NULL and UNIQUE specifications are not supported, use PRIMARY KEY without the specifications.
- The entire table definition is enclosed in parentheses. A comma is used to separate each table element (attributes, primary key and foreign key) definition.
- The DEFAULT constraint is used to assign a value to an attribute when a new row is added to the table. The end user may, of course, enter a value other than the default value.
- The CHECK constraint is used to validate data when an attribute value is entered. The CHECK constraint does precisely what its name suggests: it checks to see that a specified condition exists. If the CHECK constraint is met for the specified attribute (that is, the condition is true), the data are accepted for that attribute. If the condition is found to be false, an error message is generated and the data are not accepted.

2.3.1 Creating the THEMEPARK TABLE

Task 2.1 Enter the following SQL command to create the THEMEPARK table.

```
CREATE TABLE      THEMEPARK (
PARK_CODE         VARCHAR2(10) PRIMARY KEY,
PARK_NAME         VARCHAR2(35) NOT NULL,
PARK_CITY         VARCHAR2(50) NOT NULL,
PARK_COUNTRY      CHAR(2) NOT NULL);
```

Notice that when you create the THEMEPARK table structure you set the stage for the enforcement of entity integrity rules by using:

```
PARK_CODE         VARCHAR2(10) PRIMARY KEY,
```

As you create this structure, also notice that the NOT NULL constraint is used to ensure that the columns PARK_NAME, PARK_CITY and PARK_COUNTRY do not accept nulls.

Remember to store this CREATE TABLE structure in your themepark.sq script.

2.3.2 Creating the EMPLOYEE TABLE

Task 2.2 Enter the following SQL command to create the EMPLOYEE table.

```
CREATE TABLE EMPLOYEE (
EMP_NUM           NUMBER(4) PRIMARY KEY,
EMP_TITLE         VARCHAR2(4),
EMP_LNAME         VARCHAR2(15) NOT NULL,
EMP_FNAME         VARCHAR2(15) NOT NULL,
EMP_DOB           DATE NOT NULL,
EMP_HIRE_DATE   DATE DEFAULT SYSDATE,
EMP_AREA_CODE   VARCHAR2(4) NOT NULL,
EMP_PHONE       VARCHAR2(12) NOT NULL,
CONSTRAINT      FK_EMP_PARK FOREIGN KEY(PARK_CODE) REFERENCES
THEMEPARK(PARK_CODE));
```

As you look at the CREATE TABLE sequence, note that referential integrity has been enforced by specifying a constraint called FKP_EMP_PARK. This foreign key constraint definition ensures that you cannot delete a Theme Park from the THEMEPARK table if at least one employee row references that Theme Park. The ORACLE RDBMS will automatically enforce referential integrity for foreign keys. This ensures that you cannot have an invalid entry in the foreign key column. In this example the employee's hire date is set to the SYSDATE using the DEFAULT constraint, so when a new employee record is created the SYSDATE function always returns today's date.

Remember to store this CREATE TABLE structure in your themepark.sq script.

2.3.3 Creating the TICKET TABLE

Task 2.3 Enter the following SQL command to create the TICKET table.

```
CREATE TABLE TICKET (
TICKET_NO       NUMBER(10) PRIMARY KEY,
TICKET_PRICE    NUMBER(4,2) DEFAULT 00.00 NOT NULL,
TICKET_TYPE     VARCHAR2(10),
PARK_CODE          VARCHAR2(10),
CONSTRAINT         CK_TICKET_TYPE CHECK (TICKET_TYPE
IN('Adult','Child','Senior','Other')),
CONSTRAINT         FK_TICKET_PARK FOREIGN KEY(PARK_CODE)
REFERENCES THEMEPARK(PARK_CODE))
```

As you create the TICKET table, notice that a number of constraints have been applied. For example, a CHECK constraint called CK_TICKET_TYPE is used to validate that the ticket type is 'Adult', 'Child', 'Senior' or 'Other'.

Remember to store this CREATE TABLE structure in your themepark.sq script.

2.3.4 Creating the ATTRACTION TABLE

Task 2.4 Enter the following SQL command to create the ATTRACTION table.

```
CREATE TABLE ATTRACTION (
ATTRACT_NO         NUMBER(10) PRIMARY KEY,
ATTRACT_NAME       VARCHAR2(35),
ATTRACT_AGE        NUMBER(3) DEFAULT 0 NOT NULL,
ATTRACT_CAPACITY NUMBER(3) NOT NULL,
PARK_CODE          VARCHAR2(10),
CONSTRAINT         FK_ATTRACT_PARK FOREIGN KEY(PARK_CODE)
REFERENCES  THEMEPARK(PARK_CODE));
```

Remember to store this CREATE TABLE structure in your themepark.sq script.

2.3.5 Creating the HOURS TABLE

Task 2.5 Enter the following SQL command to create the HOURS table.

```
CREATE TABLE HOURS (
EMP_NUM              NUMBER(4),
ATTRACT_NO           NUMBER(10),
HOURS_PER_ATTRACT    NUMBER(2) NOT NULL,
HOUR_RATE            NUMBER(4,2) NOT NULL,
DATE_WORKED          DATE NOT NULL,
CONSTRAINT           PK_HOURS PRIMARY KEY(EMP_NUM,
ATTRACT_NO, DATE_WORKED),
CONSTRAINT FK_HOURS_EMP  FOREIGN KEY   (EMP_NUM) REFERENCES
EMPLOYEE(EMP_NUM),
CONSTRAINT FK_HOURS_ATTRACT FOREIGN KEY (ATTRACT_NO)
REFERENCES ATTRACTION(ATTRACT_NO));
```

As you create the HOURS table, notice that the HOURS table contains FOREIGN KEYS to both the ATTRACTION and the EMPLOYEE's table.

Remember to store this CREATE TABLE structure in your themepark.sq script.

2.3.6 Creating the SALES TABLE

Task 2.6 Enter the following SQL command to create the SALES table.

```
CREATE TABLE SALES (
TRANSACTION_NO          NUMBER PRIMARY KEY,
PARK_CODE               VARCHAR2(10),
SALE_DATE               DATE DEFAULT SYSDATE NOT NULL,
CONSTRAINT       FK_SALES_PARK FOREIGN KEY(PARK_CODE)
REFERENCES THEMEPARK(PARK_CODE) ON DELETE CASCADE,
CONSTRAINT       SALE_CK1 CHECK (SALE_DATE > TO_DATE('01-JAN-
2005','DD-MON-YYYY')));
```

As you create the SALES table, look closely at the SALE_CK1 CHECK constraint. Can you describe the purpose of this constraint?

Remember to store this CREATE TABLE structure in your themepark.sq script.

2.3.7 Creating the SALESLINE TABLE

Task 2.7 Enter the following SQL command to create the SALES_LINE table.

```
CREATE TABLE SALES_LINE (
TRANSACTION_NO NUMBER,
LINE_NO          NUMBER(2,0) NOT NULL,
TICKET_NO        NUMBER(10)  NOT NULL,
LINE_QTY         NUMBER(4) DEFAULT 0 NOT NULL,
LINE_PRICE       NUMBER(9,2) DEFAULT 0.00 NOT NULL,
CONSTRAINT PK_SALES_LINE   PRIMARY KEY
(TRANSACTION_NO,LINE_NO),
CONSTRAINT FK_SALES_LINE_SALES  FOREIGN KEY (TRANSACTION_NO)
REFERENCES SALES ON DELETE CASCADE,
CONSTRAINT FK_SALES_LINE_TICKET FOREIGN KEY (TICKET_NO)
REFERENCES TICKET(TICKET_NO));
```

As you create the SALES_LINE table, examine the constraint called FK_SALES_LINE_SALES. What is the purpose of ON DELETE CASCADE?

Remember to store this CREATE TABLE structure in your themepark.sq script.

2.4. Creating Indexes

You learned in Chapter 3, The Relational Database Model, that indexes can be used to improve the efficiency of searches and to avoid duplicate column values. Using the **CREATE INDEX** command, SQL indexes can be created on the basis of any selected attribute. For example, based on the attribute SALE_DATE stored in the SALES table, the following command creates an index named SALE_DATE_INDEX:

```
CREATE INDEX SALE_DATE_INDEX ON SALES(SALE_DATE);
```

Task 2.8 Create the SALE_DATE_INDEX shown above and add the CREATE INDEX SQL command to your script file themepark.sql.

The **DROP TABLE** command permanently deletes a table (and thus its data) from the database schema. When you write a script file to create a database schema, it is useful to add DROP TABLE commands at the start of the file. If you need to amend the table structures in any way, just one script can then be run to re-create all the database structures. Primary and foreign key constraints control the order in which you drop the tables – generally you drop in the reverse order of creation. The DROP commands for the Theme Park database are:

```
DROP TABLE SALES_LINE;
DROP TABLE SALES;
DROP TABLE HOURS;
DROP TABLE ATTRACTION;
DROP TABLE TICKET;
DROP TABLE EMPLOYEE;
DROP TABLE THEMEPARK;
```

Task 2.9. Add the DROP commands to the start of your script file and then run the themepark.sql script.

2.4 Display a table's structure

The command DESCRIBE is used to display the structure of an individual table. To see the structure of the THEMEPARK table you would enter the command

DESCRIBE THEMEPARK.

Task 2.10 Use the DESCRIBE command to view the structure of the other database tables that you have created in this lab.

2.5 Listing all tables

To list all tables that have been created by yourself you need to access ORACLE's data dictionary in order to view the metadata. For example, to find out the names of all of the relational tables that you have created, you must type the following query:

```
SELECT TABLE_NAME

FROM USER_TABLES;
```

Task 2.11 Execute a query which displays all the tables you own and check to see if the tables you have created in this lab are present.

Note

To find out more about data dictionaries in ORACLE DBMS visit the Oracle Technological Network at: http://www.oracle.com/technology/index.html.

2.6 Altering the table structure

All changes in the table structure are made by using the **ALTER TABLE** command, followed by a keyword that produces the specific change you want to make. Three options are available: ADD, MODIFY, and DROP. Entering ADD enables you to add a column, and MODIFY enables you to change column characteristics. Most RDBMSs do not allow you to delete a column (unless the column does not contain any values), because such an action may delete crucial data that are used by other tables. The basic syntax to add or modify columns is:

ALTER TABLE *tablename*

{ADD | MODIFY}(*columnname datatype* [{ADD | MODIFY} *columnname datatype*]);

The ALTER TABLE command can also be used to add table constraints. In those cases, the syntax would be:

ALTER TABLE *tablename*

ADD *constraint* [ADD *constraint*] ;

where *constraint* refers to a constraint definition, e.g. primary or foreign key.

However, when removing a constraint you need to specify the name given to the constraint. That is one reason why you should always name your constraints in your CREATE TABLE or ALTER TABLE statement.

If you wanted to modify the column ATTRACT_CAPACITY in the ATTRACTION table by changing the date characteristics from NUMBER(3) to NUMBER(4), you would execute the following command:

ALTER TABLE ATTRACTION

```
MODIFY (ATTRACT_CAPACITY NUMBER(4));
```

Note

Some DBMSs impose limitations on when it's possible to change attribute characteristics. For example, ORACLE lets you increase (but not decrease) the size of a column. The reason for this restriction is that an attribute modification will affect the integrity of the data in the database. In fact, some attribute changes can be done only when there are no data in any rows for the affected attribute.

You can learn more about altering a table's structure in Chapter 8, Introduction to Structured Query Language.

You have now reached the end of the first ORACLE lab. The tables that you have created will be used in the rest of this lab guide to explore the use of SQL in ORACLE in more detail.

Lab 3: Data Manipulation Commands

The learning objectives for this lab are:

- To know how to insert, update and delete data from within a table
- To learn how to retrieve data from a table using the SELECT statement

3.1 Adding Table Rows

SQL requires the use of the **INSERT** command to enter data into a table. The INSERT command's basic syntax looks like this:

INSERT INTO *tablename* VALUES (*value1, value2, ... , valuen*).

The order in which you insert data is important. For example, because the TICKET uses its PARK_CODE to reference the THEMEPARK table's PARK_CODE, an integrity violation will occur if those THEMEPARK table PARK_CODE values don't yet exist. Therefore, you need to enter the THEMEPARK rows before the TICKET rows. Complete the following tasks to insert data into the THEMEPARK and TICKET tables:

Task 3.1 Enter the first two rows of data into the THEMEPARK table using the following SQL insert commands;

```
INSERT INTO THEMEPARK VALUES ('FR1001','FairyLand','PARIS','FR');

INSERT INTO THEMEPARK VALUES ('UK3452','PleasureLand','STOKE','UK');
```

Task 3.2 Enter the following corresponding rows of data into the TICKET table using the following SQL insert commands.

```
INSERT INTO TICKET VALUES (13001,18.99,'Child','FR1001');

INSERT INTO TICKET VALUES (13002,34.99,'Adult','FR1001');

INSERT INTO TICKET VALUES (13003,20.99,'Senior','FR1001');

INSERT INTO TICKET VALUES (88567,22.50,'Child','UK3452');

INSERT INTO TICKET VALUES (88568,42.10,'Adult','UK3452');

INSERT INTO TICKET VALUES (89720,10.99,'Senior','UK3452');
```

Any changes made to the table contents are not physically saved on disk until you close the database, close the program you are using, or use the **COMMIT** command. The COMMIT command will permanently save *any* changes – such as rows added, attributes modified, and rows deleted – made to any table in the database. Therefore, if you intend to make your changes to the THEMEPARK and TICKET tables permanent, it is a good idea to save those changes by using COMMIT;

Task 3.3 COMMIT the changes to the THEMEPARK and TICKET tables to the database.

Task 3.4 Run the script file **themeparkdata.sql** to insert the rest of the data into the Theme Park database. This script file is available on the CD-ROM companion. Ensure you COMMIT the changes to the database.

3.2 Retrieving data from a table using the SELECT Statement

In Chapter 8, Introduction to Structured Query Language, you studied the SELECT command. The SELECT command has many optional clauses, but in its simplest can be written as:

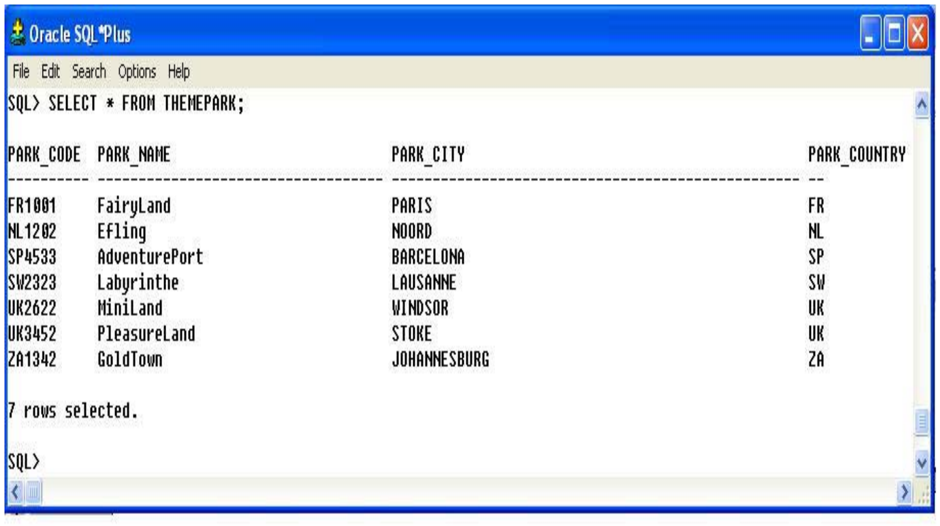
```
SELECT     columnlist
FROM       tablelist
[WHERE     conditionlist ];
```

Notice that the command must finish with a semicolon and will be executed when the Enter key is pressed at the end of the command.

The simplest query involves viewing all columns in one table. To display the details of all Theme Parks in the Theme Park database, type the following:

```
SELECT *
FROM THEMEPARK;
```

You should see the output displayed in Figure 7.



```
SQL> SELECT * FROM THEMEPARK;

PARK_CODE  PARK_NAME                          PARK_CITY                                          PARK_COUNTRY
---------- ---------------------------------- -------------------------------------------------- --
FR1001     FairyLand                          PARIS                                              FR
NL1202     Efling                             NOORD                                              NL
SP4533     AdventurePort                      BARCELONA                                          SP
SW2323     Labyrinthe                         LAUSANNE                                           SW
UK2622     MiniLand                           WINDSOR                                            UK
UK3452     PleasureLand                       STOKE                                              UK
ZA1342     GoldTown                           JOHANNESBURG                                       ZA

7 rows selected.

SQL>
```

Figure 7: Displaying all columns from the CUSTOMER Table

The SELECT command and the FROM clause are necessary for any SQL query, and must always be included so that the DBMS knows which columns we want to display and which table they come from.

Task 3.5. Type in the following examples of the SELECT statement and check your results with those provided in Figures 8 and 9. In these two examples you are selecting specific columns from a single table.

Example 1

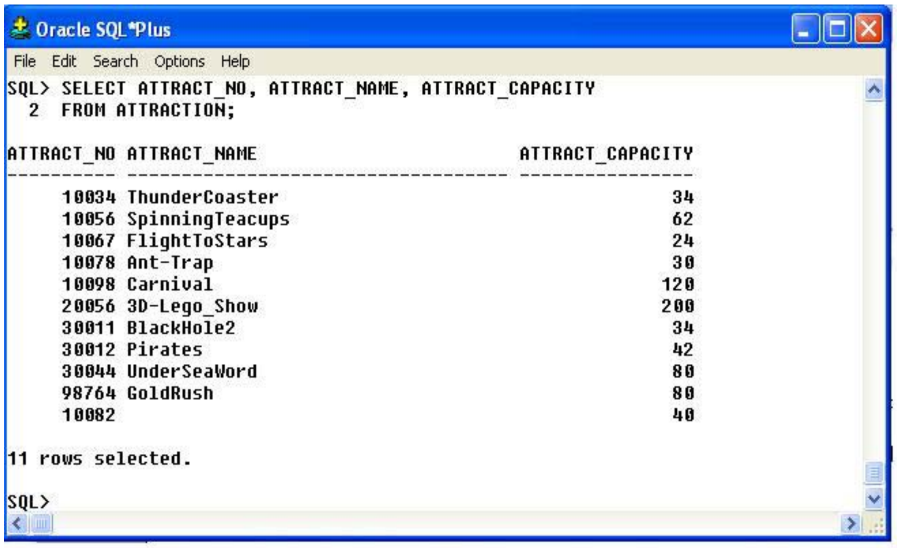
SELECT ATTRACT_NO, ATTRACT_NAME, ATTRACT_CAPACITY

FROM ATTRACTION;

Example 2

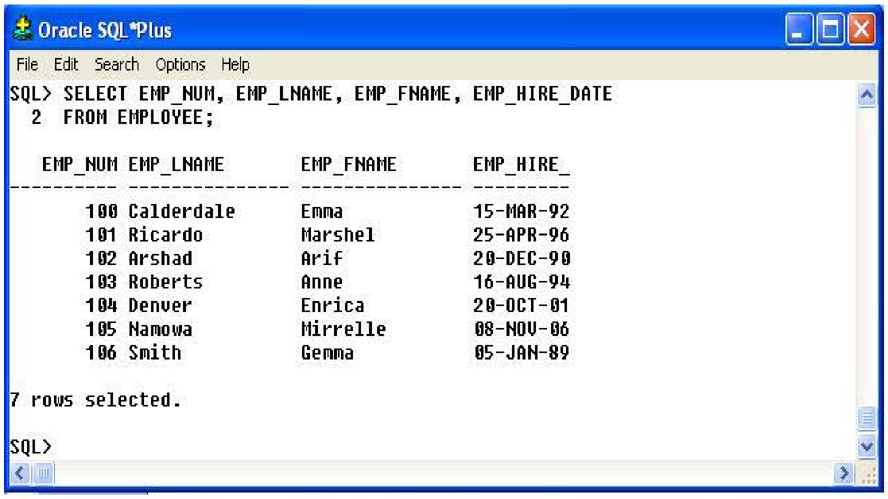
SELECT EMP_NUM, EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE

FROM EMPLOYEE;



```
SQL> SELECT ATTRACT_NO, ATTRACT_NAME, ATTRACT_CAPACITY
  2  FROM ATTRACTION;

ATTRACT_NO ATTRACT_NAME                         ATTRACT_CAPACITY
---------- ------------------------------------ ----------------
     10034 ThunderCoaster                                     34
     10056 SpinningTeacups                                    62
     10067 FlightToStars                                      24
     10078 Ant-Trap                                           30
     10098 Carnival                                          120
     20056 3D-Lego_Show                                      200
     30011 BlackHole2                                         34
     30012 Pirates                                            42
     30044 UnderSeaWord                                       80
     98764 GoldRush                                           80
     10082                                                    40

11 rows selected.

SQL>
```

Figure 8: Output for Example 1



```
SQL> SELECT EMP_NUM, EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE
  2  FROM EMPLOYEE;

   EMP_NUM EMP_LNAME       EMP_FNAME       EMP_HIRE_
---------- --------------- --------------- ---------
       100 Calderdale      Emma            15-MAR-92
       101 Ricardo         Marshel         25-APR-96
       102 Arshad          Arif            20-DEC-90
       103 Roberts         Anne            16-AUG-94
       104 Denver          Enrica          20-OCT-01
       105 Namowa          Mirrelle        08-NOV-06
       106 Smith           Gemma           05-JAN-89

7 rows selected.

SQL>
```

Figure 9: Output for Example 2

3.3 Updating table rows

The **UPDATE** command is used to modify data in a table. The syntax for this command is:

UPDATE *tablename*

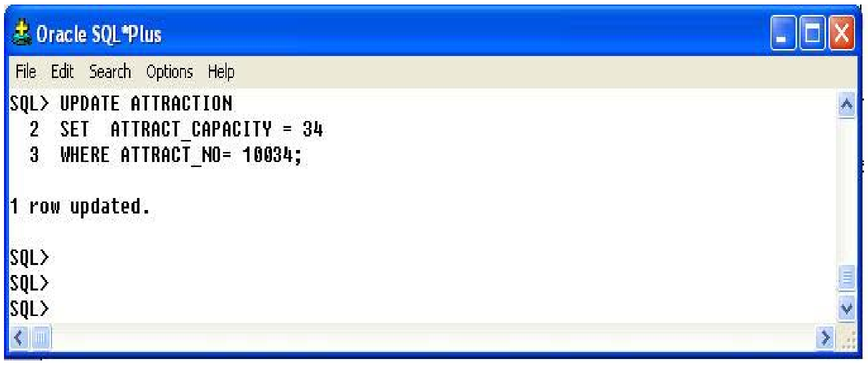
SET *columnname* = *expression* [, *columnname* = *expression*]

[WHERE *conditionlist*];

For example, if you wanted to change the attraction capacity of the attraction number 10034 from 34 to 38, the primary key, ATTRACT_NO would be used to locate the correct (second) row. You would type:

```
UPDATE     ATTRACTION
SET        ATTRACT_CAPACITY = 34
WHERE      ATTRACT_NO= 10034;
```

The output is shown in Figure 10.



```
SQL> UPDATE ATTRACTION
  2  SET  ATTRACT_CAPACITY = 34
  3  WHERE ATTRACT_NO= 10034;

1 row updated.

SQL>
SQL>
SQL>
```

Figure 10: Updating the attraction capacity

Note

If more than one attribute is to be updated in the row, separate each attribute with commas.

Remember, the UPDATE command is a set-oriented operator. Therefore, if you don't specify a WHERE condition, the UPDATE command will apply the changes to *all* rows in the specified table.

Task 3.6 Enter the following SQL UPDATE command to update the age a person can go on a specific ride in the Theme Park.

```
UPDATE      ATTRACTION
SET         ATTRACT_AGE = 14;
```

Confirm the update by using this command to check the ATTRACTION table's listing:

```
SELECT      *       FROM        ATTRACTION;
```

Notice that all the values of ATTRACT_AGE have the same value.

3.4 Restoring table contents

Suppose you decide you have made a mistake in updating the attraction age to be the same for all attractions within the Theme Park. Assuming you have not yet used the COMMIT command to store the changes permanently in the database, you can restore the database to its previous condition with the **ROLLBACK** command. ROLLBACK undoes any changes and brings the data back to the values that existed before the changes were made.

Task 3.7 To restore the data to their "pre-change" condition, type:

ROLLBACK;

and press Enter. Use the SELECT statement again to see that the ROLLBACK did, in fact, restore the data to their original values.

Note

For more information about ROLLBACK, See section 8.3.5, Restoring Table Contents in Chapter 8, Introduction to Structured Query Language.

3.5 Deleting table rows

It is easy to delete a table row using the **DELETE** statement. The syntax is:

```
DELETE FROM tablename
[WHERE conditionlist ];
```

For example, if you want to delete a specific Theme Park from the THEMEPARK table you could use the PARK_CODE as shown in the following SQL command:

```
DELETE      FROM          THEMEPARK
WHERE       PARK_CODE = ′ SW2323′;
```

In this example, the primary key value lets SQL find the exact record to be deleted. However, deletions are not limited to a primary key match; any attribute may be used.

If you do not specify a WHERE condition, *all* rows from the specified table will be deleted!

Note

If you make a mistake while working through this lab, use the themepark.sql script to re-create the database schema and insert the sample data.

3.6 Inserting Table rows with a subquery

Subqueries are often used to add multiple rows to a table, using another table as the source of the data. The syntax for the INSERT statement is:

INSERT INTO *tablename*   SELECT *columnlist* FROM *tablename*;

In that case, the INSERT statement uses a SELECT subquery. A **subquery**, also known as a nested query or an inner query, is a query that is embedded (or nested) inside another query. The inner query is always executed first by the RDBMS. Given the previous SQL statement, the INSERT portion represents the outer query and the SELECT portion represents the inner query, or subquery.

Task 3.8 Use the following steps to populate your EMPLOYEE table.

- Run the script emp_copy.sql which is available on the accompanying CD-ROM. This script creates a table called EMP_COPY, which we will populate using data from the EMPLOYEE table in the THEMEPARK database.
- Add the rows to EMP_COPY table by copying all rows from EMPLOYEE.

  ```
  INSERT INTO EMP_COPY SELECT * FROM EMPLOYEE;
  ```
- Permanently save the changes: COMMIT;

If you followed those steps correctly, you now have the EMPLOYEE table populated with the data that will be used in the remaining sections of this lab guide.

3.7 Exercises

E3.1 Load and run the script park_copy.sql which creates the PARK_COPY table.

E3.2 Describe the PARK_COPY and THEMEPARK tables and notice that they are different.

E3.3 Write a subquery to populate the fields PARK_CODE, PARK_NAME and PARK_COUNTRY in the PARK_COPY using data from the THEMEPARK table. Display the contents of the PARK_COPY table.

E3.4 Update the AREA_CODE and PARK_PHONE fields in the PARK_COPY table with the following values.

PARK_CODE	**PARK_AREA_CODE**	**PARK_PHONE**
FR1001	5678	223-556
UK3452	0181	678-789
ZA1342	8789	797-121

E3.5 Add the following new Theme Parks to the PARK_COPY TABLE.

PARK_CODE	**PARK_NAME**	**PARK_COUNTRY**	**PARK_AREA_CODE**	**PARK_PHONE**
AU1001	**SkiWorld**	**AU**	**1212**	**440-232**
GR5001	**RoboLand**	**GR**	**4565**	**123-123**

E3.6 Delete the Theme Park called RoboLand.

Lab 4: Basic SELECT statements

The learning objectives of this lab are to:

- Use arithmetic operators in SQL statements
- Select rows from a table with conditional restrictions
- Apply logical operators to have multiple conditions

4.1 Using Arithmetic operators in SQL Statements

SQL commands are often used in conjunction with arithmetic operators. As you perform mathematical operations on attributes, remember the rules of precedence. As the name suggests, the **rules of precedence** are the rules that establish the order in which computations are completed. For example, note the order of the following computational sequence:

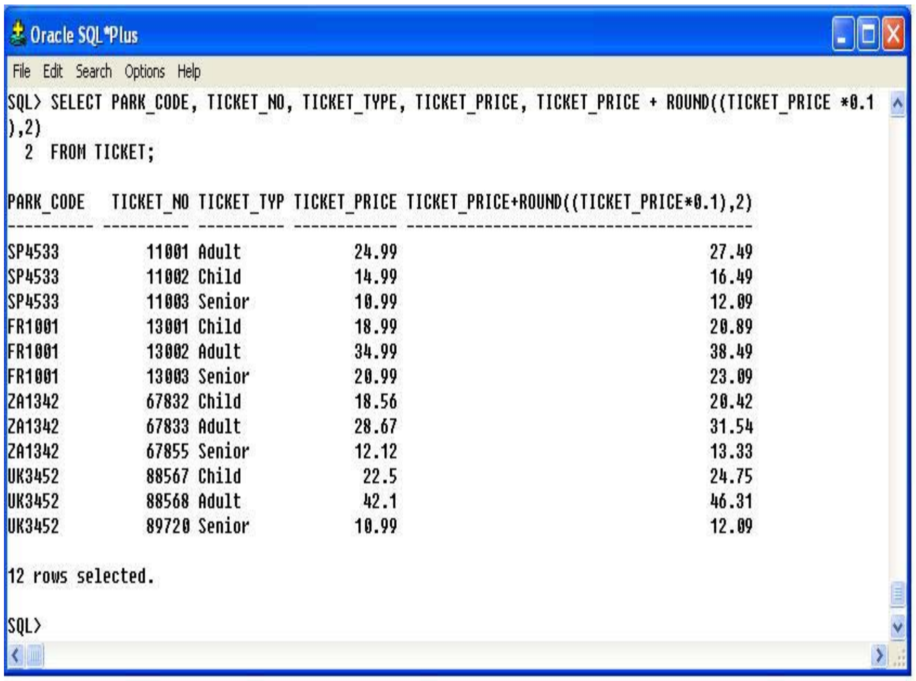
1. Perform operations within parentheses
2. Perform power operations
3. Perform multiplications and divisions
4. Perform additions and subtractions

Task 4.1 Suppose the owners of all the Theme Parks wanted to compare the current ticket prices, with an increase in price of each ticket by 10%. To generate this query type:

```
SELECT PARK_CODE, TICKET_NO, TICKET_TYPE, TICKET_PRICE,
TICKET_PRICE + ROUND((TICKET_PRICE *0.1),2)
```

FROM TICKET;

The output for this query is shown in Figure 11. The ROUND function is used to ensure the result is displayed to two decimal places.



```
SQL> SELECT PARK_CODE, TICKET_NO, TICKET_TYPE, TICKET_PRICE, TICKET_PRICE + ROUND((TICKET_PRICE *0.1
),2)
  2  FROM TICKET;

PARK_CODE   TICKET_NO TICKET_TYP TICKET_PRICE TICKET_PRICE+ROUND((TICKET_PRICE*0.1),2)
---------- ---------- ---------- ------------ ----------------------------------------
SP4533          11001 Adult             24.99                                    27.49
SP4533          11002 Child             14.99                                    16.49
SP4533          11003 Senior            10.99                                    12.09
FR1001          13001 Child             18.99                                    20.89
FR1001          13002 Adult             34.99                                    38.49
FR1001          13003 Senior            20.99                                    23.09
ZA1342          67832 Child             18.56                                    20.42
ZA1342          67833 Adult             28.67                                    31.54
ZA1342          67855 Senior            12.12                                    13.33
UK3452          88567 Child              22.5                                    24.75
UK3452          88568 Adult              42.1                                    46.31
UK3452          89720 Senior            10.99                                    12.09

12 rows selected.

SQL>
```

Figure 11: Output showing 10% increase in ticket prices

You will see in Figure 11, that the last column is named after the arithmetic expression in the query. To rename the column heading, a column alias needs to be used. Modify the query as follows and note that the name of the heading has changed to PRICE_INCREASE when you execute the following query.

SELECT PARK_CODE, TICKET_NO, TICKET_TYPE, TICKET_PRICE,

TICKET_PRICE + ROUND((TICKET_PRICE *0.1),2) PRICE_INCREASE

FROM TICKET;

Note

When dealing with column names that require spaces, the optional keyword AS can be used. For example:

SELECT PARK_CODE, TICKET_NO, TICKET_TYPE, TICKET_PRICE,

TICKET_PRICE + ROUND((TICKET_PRICE *0.1),2) AS

“PRICE INCREASE”

FROM TICKET;

4.2 Selecting rows with conditional restrictions

Numerous conditional restrictions can be placed on the selected table contents in the WHERE clause of the SELECT statement. For example, the comparison operators shown in Table 1 can be used to restrict output.

Table 1 Comparison Operators

SYMBOL	**MEANING**
=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<> or !=	Not equal to
BETWEEN	Used to check if an attribute is within a range
IN	Used to check if an attribute value matches any value within a list

LIKE	Used to check if an attribute value matches a given string pattern
IS NULL / IS NOT NULL	Used to check if an attribute is NULL / is not NULL

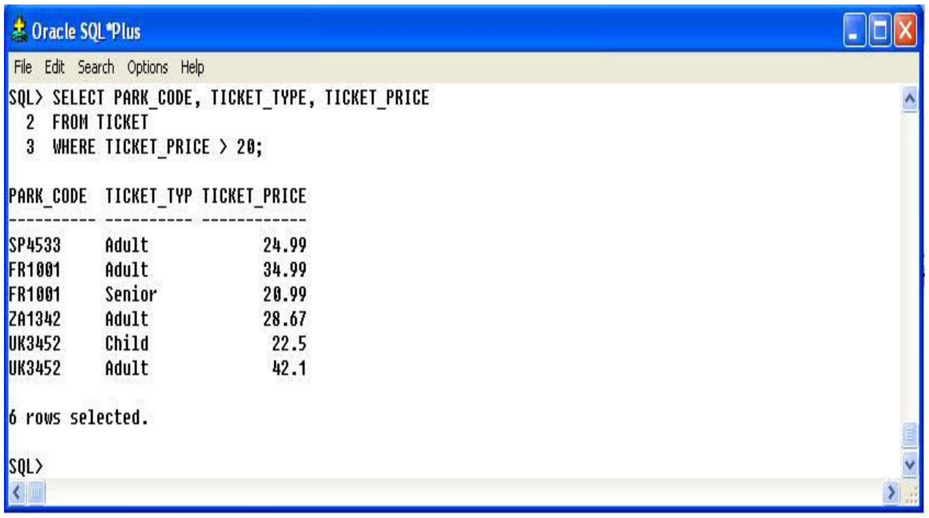
We will now explore some of these conditional operators using examples.

Greater than

The following example uses the greater than operator to display the Theme Park code, ticket price and ticket type of all tickets where the ticket price is greater than €20.00.

```
SELECT PARK_CODE, TICKET_TYPE, TICKET_PRICE

FROM TICKET

WHERE TICKET_PRICE > 20;
```

The output is shown in Figure 12.



```
SQL> SELECT PARK_CODE, TICKET_TYPE, TICKET_PRICE
  2  FROM TICKET
  3  WHERE TICKET_PRICE > 20;

PARK_CODE  TICKET_TYP TICKET_PRICE
---------- ---------- ------------
SP4533     Adult             24.99
FR1001     Adult             34.99
FR1001     Senior            20.99
ZA1342     Adult             28.67
UK3452     Child              22.5
UK3452     Adult              42.1

6 rows selected.

SQL>
```

Figure 12: Tickets costing greater than €20.00

Task 4.2 Type in and execute the query and test out the greater than operator. Do you get the same results has shown in Figure 12?

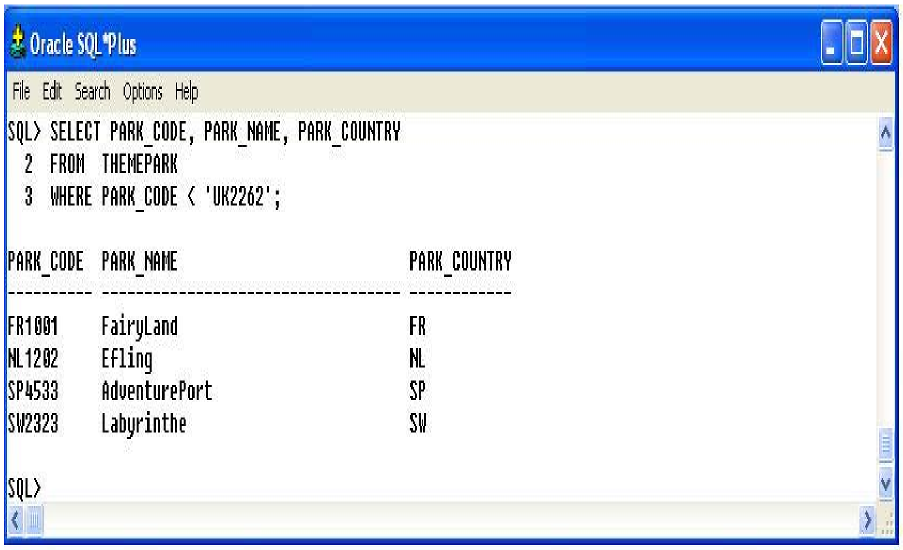
Task 4.3 Modify the query you have just executed to display tickets that are less than €30.00.

Character comparisons

Comparison operators may even be used to place restrictions on character-based attributes.

Task 4.4 Execute the following query which produces a list of all rows in which the PARK_CODE is alphabetically less than UK2262. (Because the ASCII code value for the letter *B* is greater than the value of the letter *A*, it follows that *A* is less than *B*.) The output will be generated as shown in Figure 13.

```
SELECT      PARK_CODE, PARK_NAME, PARK_COUNTRY
FROM        THEMEPARK
WHERE       PARK_CODE < 'UK2262';
```



```
SQL> SELECT PARK_CODE, PARK_NAME, PARK_COUNTRY
  2  FROM  THEMEPARK
  3  WHERE PARK_CODE < 'UK2262';

PARK_CODE  PARK_NAME                         PARK_COUNTRY
---------- --------------------------------- ------------
FR1001     FairyLand                         FR
NL1202     Efling                            NL
SP4533     AdventurePort                     SP
SW2323     Labyrinthe                        SW

SQL>
```

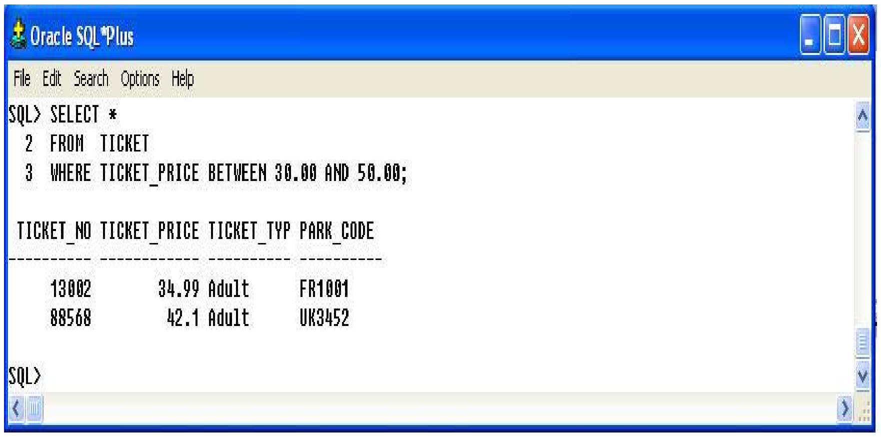
Figure 13: Example of character comparison

BETWEEN

The operator BETWEEN may be used to check whether an attribute value is within a range of values. For example, if you want to see a listing for all tickets whose prices are between €30 and €50, use the following command sequence:

```
SELECT      *
FROM        TICKET
WHERE       TICKET_PRICE BETWEEN 30.00 AND 50.00;
```

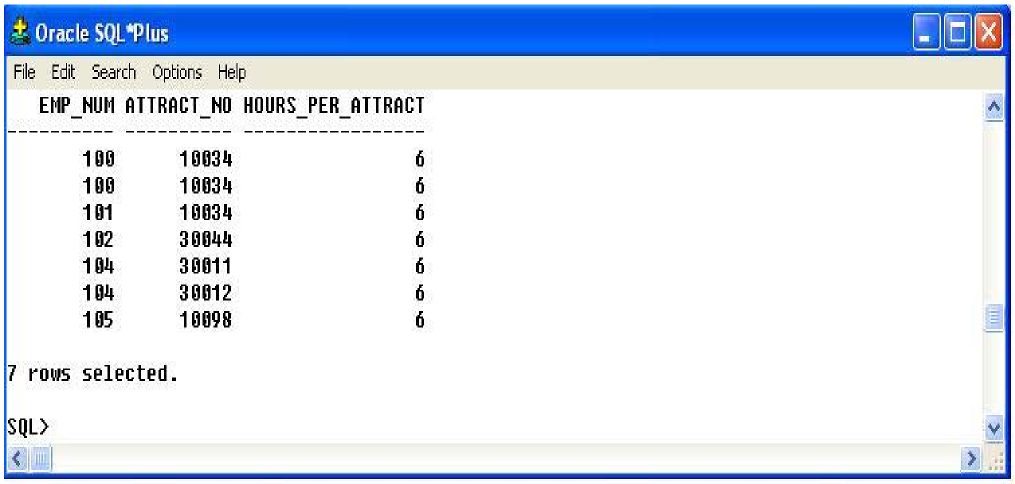
Figure 14 shows the output you should see for this query.



```
SQL> SELECT *
  2  FROM  TICKET
  3  WHERE TICKET_PRICE BETWEEN 30.00 AND 50.00;

 TICKET_NO TICKET_PRICE TICKET_TYP PARK_CODE
---------- ------------ ---------- ----------
     13002        34.99 Adult      FR1001
     88568         42.1 Adult      UK3452

SQL>
```

Figure 14: Displaying ticket prices BETWEEN two values

Task 4.5 Write a query which displays the employee number, attraction no., the hours worked per attraction and the date worked where the hours worked per attraction is between 5 and 10. Hint: You will need to select data from the HOURS table. The output for the query is shown in Figure 15.



```
   EMP_NUM ATTRACT_NO HOURS_PER_ATTRACT
---------- ---------- -----------------
       100      10034                 6
       100      10034                 6
       101      10034                 6
       102      30044                 6
       104      30011                 6
       104      30012                 6
       105      10098                 6

7 rows selected.

SQL>
```

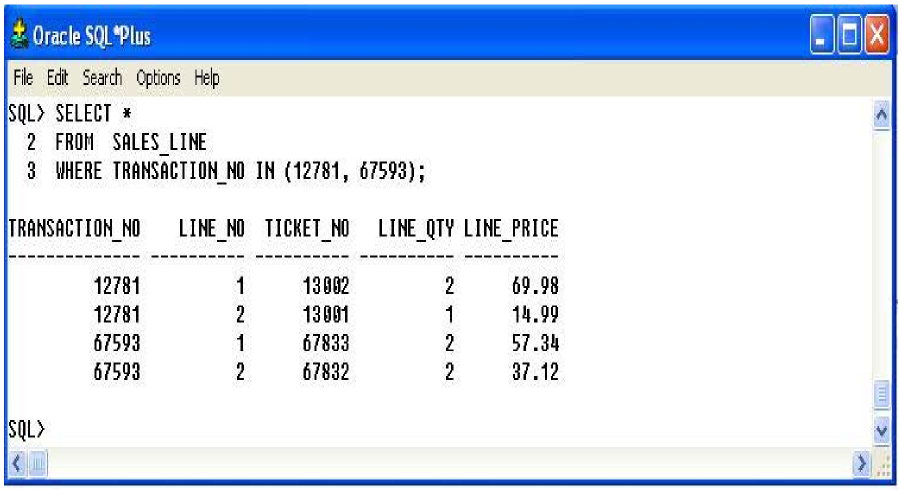
Figure 15: Output for Task 4.5

IN

The IN operator is used to test for values which are in a list. The following query finds only the rows in the SALES_LINE table that match up to a specific sales transaction. i.e. TRANSACTION_NO is either 12781 or 67593.

```
SELECT     *
FROM       SALES_LINE
WHERE      TRANSACTION_NO IN (12781, 67593);
```

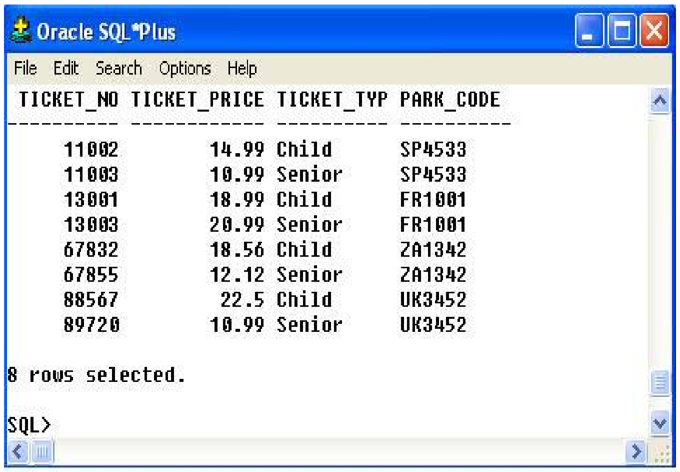
The result of this query is shown in Figure 16.



```
SQL> SELECT *
  2  FROM  SALES_LINE
  3  WHERE TRANSACTION_NO IN (12781, 67593);

TRANSACTION_NO    LINE_NO  TICKET_NO   LINE_QTY LINE_PRICE
-------------- ---------- ---------- ---------- ----------
         12781          1      13002          2      69.98
         12781          2      13001          1      14.99
         67593          1      67833          2      57.34
         67593          2      67832          2      37.12

SQL>
```

Figure 16: Selecting rows using the IN command

Task 4.6 Write a query to display all tickets that are of type Senior or Child. Hint: Use the TICKET table. The output you should see is shown in Figure 17.



```
 TICKET_NO TICKET_PRICE TICKET_TYP PARK_CODE
---------- ------------ ---------- ----------
     11002        14.99 Child      SP4533
     11003        10.99 Senior     SP4533
     13001        18.99 Child      FR1001
     13003        20.99 Senior     FR1001
     67832        18.56 Child      ZA1342
     67855        12.12 Senior     ZA1342
     88567         22.5 Child      UK3452
     89720        10.99 Senior     UK3452

8 rows selected.

SQL>
```

Figure 17: Output for Task 4.5

LIKE

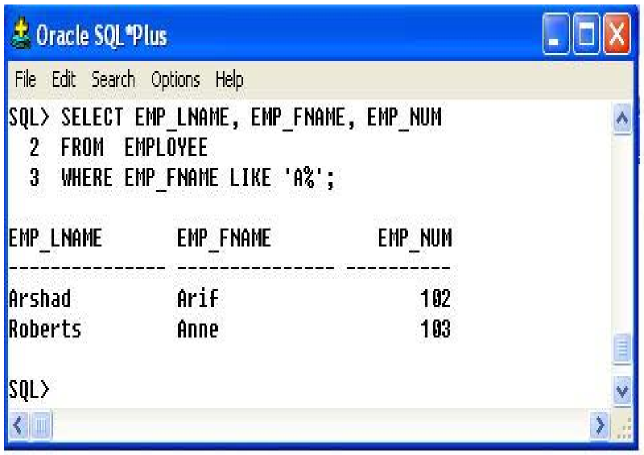
The LIKE operator is used to find patterns within string attributes. Standard SQL allows you to use the percent sign (%) and underscore (_) wildcard characters to make matches when the entire string is not known. % means any and all *following* characters are eligible while _ means any *one* character may be substituted for the underscore.

Task 4.7 Enter the following query which finds all EMPLOYEE rows whose first names begin with the letter A.

```
SELECT    EMP_LNAME, EMP_FNAME, EMP_NUM
FROM      EMPLOYEE
```

WHERE EMP_FNAME LIKE ′A%′;

Figure 18 shows the output you should see for this query.

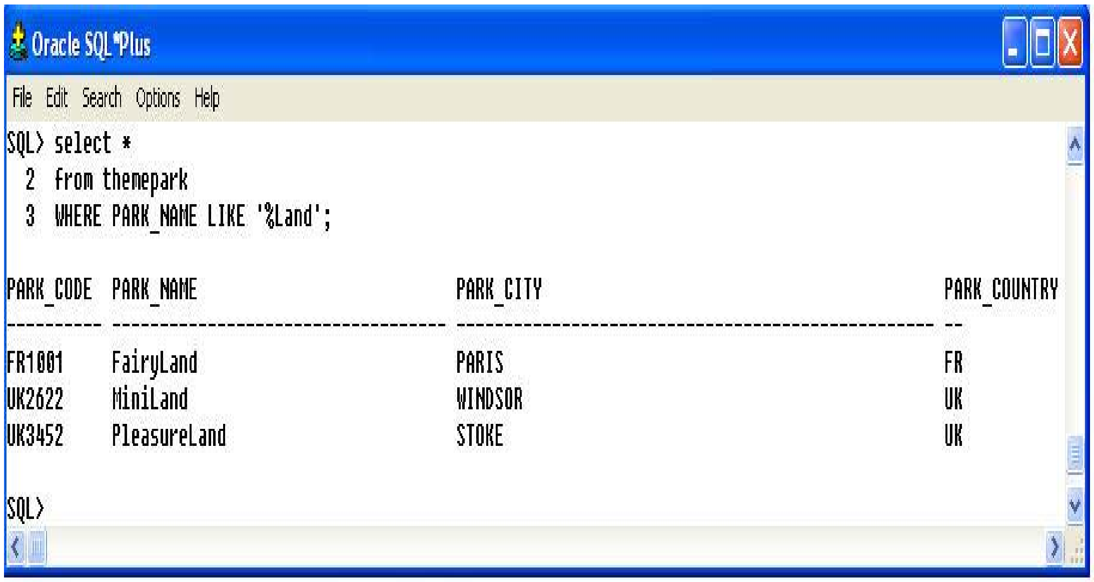


```
SQL> SELECT EMP_LNAME, EMP_FNAME, EMP_NUM
  2  FROM  EMPLOYEE
  3  WHERE EMP_FNAME LIKE 'A%';

EMP_LNAME       EMP_FNAME          EMP_NUM
--------------- --------------- ----------
Arshad          Arif                   102
Roberts         Anne                   103

SQL>
```

Figure 18: Query using the LIKE command

Task 4.8 Write a query which finds all Theme Parks that have a name ending in "Land".

The output you should see is shown in Figure 19.



```
SQL> select *
  2  from themepark
  3  WHERE PARK_NAME LIKE '%Land';

PARK_CODE  PARK_NAME                           PARK_CITY                                          PARK_COUNTRY
---------- ----------------------------------- -------------------------------------------------- --
FR1001     FairyLand                           PARIS                                              FR
UK2622     MiniLand                            WINDSOR                                            UK
UK3452     PleasureLand                        STOKE                                              UK

SQL>
```

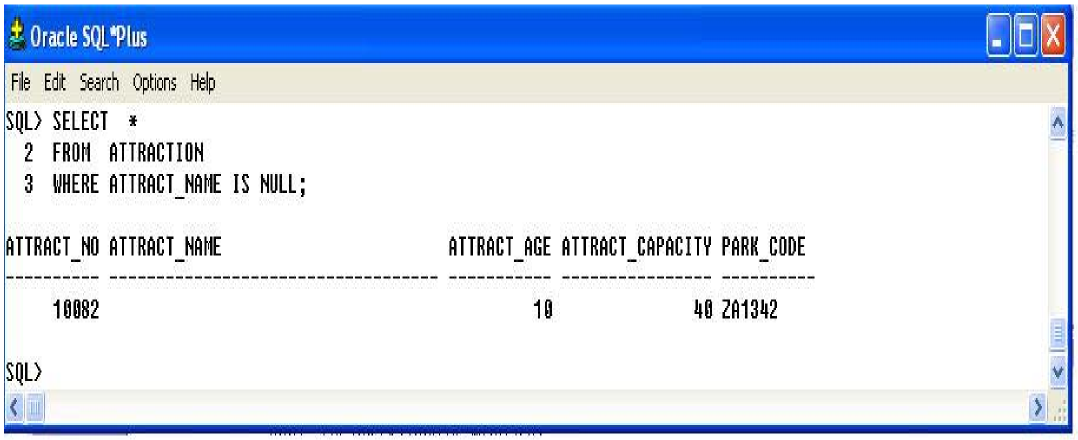
Figure 19: Solution to Task 4.8

NULL and IS NULL

IS NULL is used to check for a null attribute value. In the following example the query lists all attractions that do not have an attraction name assigned (ATTRACT_NAME is null). The query could be written as:

```
SELECT     *
FROM       ATTRACTION
WHERE      ATTRACT_NAME IS NULL;
```

The output for this query is shown in Figure 20.



```
SQL> SELECT  *
  2  FROM  ATTRACTION
  3  WHERE ATTRACT_NAME IS NULL;

ATTRACT_NO ATTRACT_NAME                             ATTRACT_AGE ATTRACT_CAPACITY PARK_CODE
---------- ---------------------------------------- ----------- ---------------- ----------
     10082                                                   10               40 ZA1342

SQL>
```

Figure 20: Listing all Attractions with no name

Logical Operators

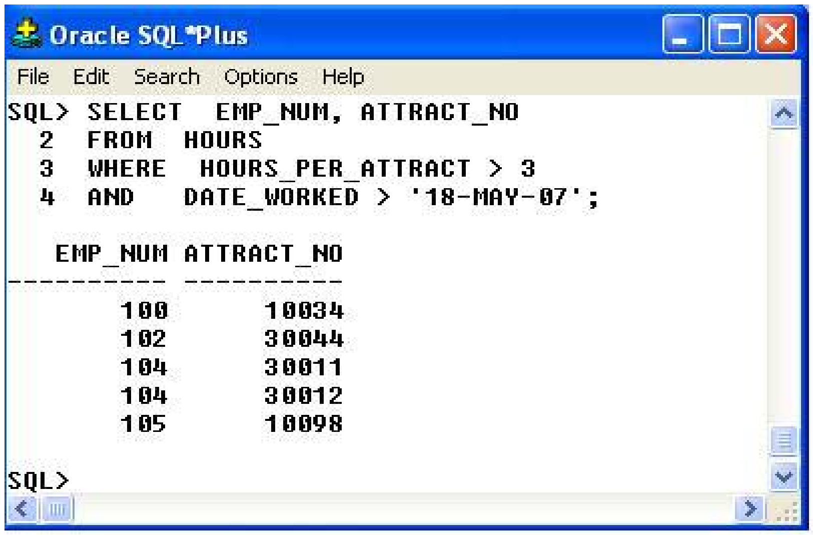
SQL allows you to have multiple conditions in a query through the use of logical operators: AND, OR and NOT. ORACLE SQL precedence rules give NOT as the highest precedence, followed by AND, and then followed by OR. However, you are strongly recommended to use parentheses to clarify the intended meaning of the query.

AND

This logical AND connective is used to set up a query where there are two conditions which must be met for the query to return the required row(s). The following query displays the employee number (EMP_NUM) and the attraction number (ATTRACT_NUM) for which the numbers of hours worked (HOURS_PER_ATTRACT) by the employee is greater than 3 and the date worked (DATE_WORKED) is after 18th May 2007.

```
SELECT      EMP_NUM, ATTRACT_NO
FROM        HOURS
WHERE       HOURS_PER_ATTRACT > 3
AND         DATE_WORKED > '18-MAY-07';
```

This query will produce the output shown in Figure 21.

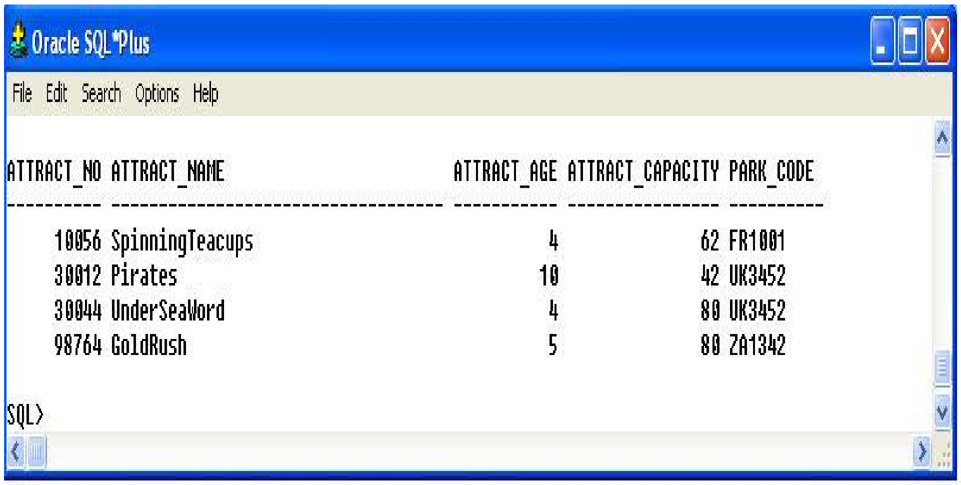


```
SQL> SELECT  EMP_NUM, ATTRACT_NO
  2  FROM  HOURS
  3  WHERE  HOURS_PER_ATTRACT > 3
  4  AND   DATE_WORKED > '18-MAY-07';

   EMP_NUM ATTRACT_NO
---------- ----------
       100      10034
       102      30044
       104      30011
       104      30012
       105      10098

SQL>
```

Figure 21: Query results using the AND operator

Task 4.9 Enter the query above and check you results with those shown in Figure 21.

Task 4.10 Write a query which displays the details of all attractions which are suitable for children aged 10 or under and have a capacity of less than 100. You should not display any information for attractions which currently have no name. Your output should correspond to that shown in Figure 22.



```
ATTRACT_NO ATTRACT_NAME                       ATTRACT_AGE ATTRACT_CAPACITY PARK_CODE
---------- ---------------------------------- ----------- ---------------- ----------
     10056 SpinningTeacups                              4               62 FR1001
     30012 Pirates                                     10               42 UK3452
     30044 UnderSeaWord                                 4               80 UK3452
     98764 GoldRush                                     5               80 ZA1342

SQL>
```

Figure 22: Query results for Task 4.10

OR

If you wanted to list the names and countries of all Theme Parks in the UK or France you would write the following query.

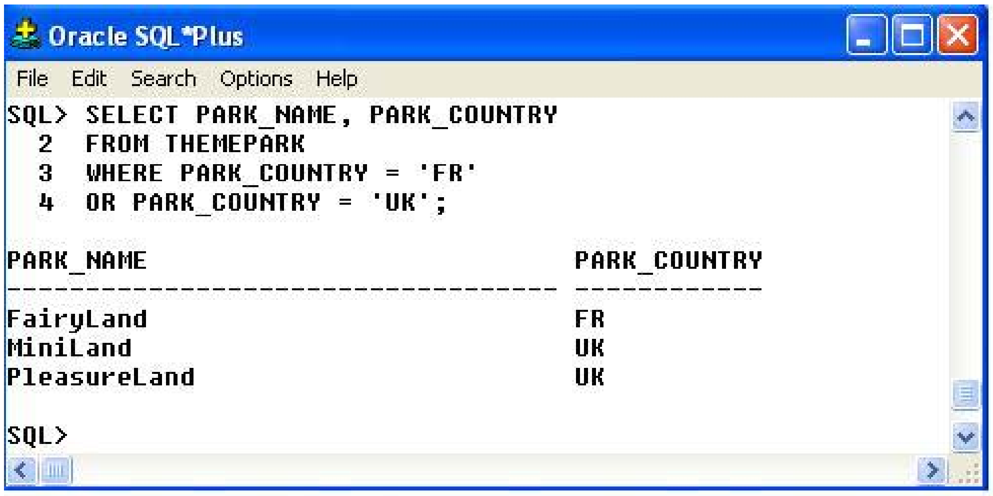
SELECT PARK_NAME, PARK_COUNTRY

FROM THEMEPARK

WHERE PARK_COUNTRY = 'FR'

OR PARK_COUNTRY = 'UK';

The output is shown in Figure 23.



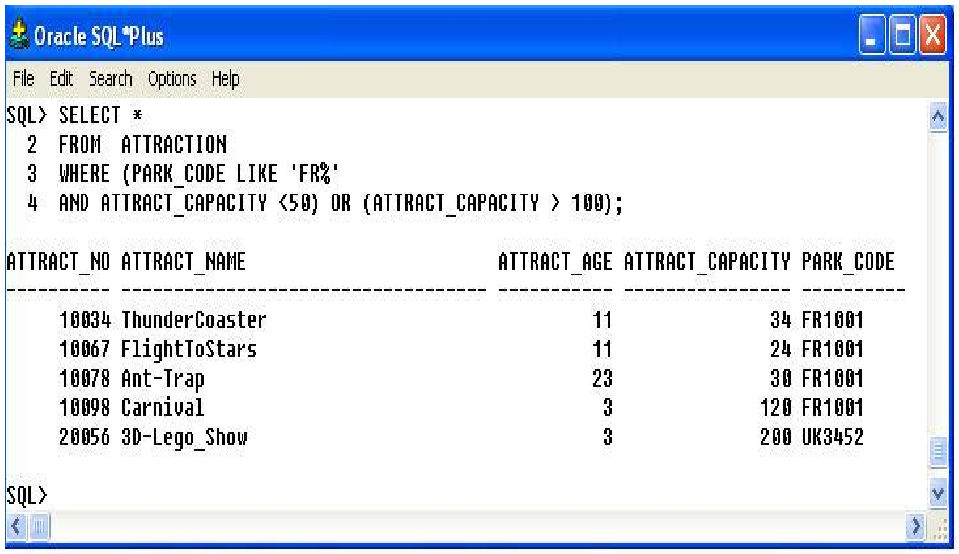
```
SQL> SELECT PARK_NAME, PARK_COUNTRY
  2  FROM THEMEPARK
  3  WHERE PARK_COUNTRY = 'FR'
  4  OR PARK_COUNTRY = 'UK';

PARK_NAME                            PARK_COUNTRY
-----------------------------------  ------------
FairyLand                            FR
MiniLand                             UK
PleasureLand                         UK

SQL>
```

Figure 23: Query results using the OR operator

When using AND and OR in the same query it is advisable to use parentheses to make explicit the precedence.

Task 4.11 Test the following query and check your output with that shown in Figure 24. Can you work out what this query is doing?

```
SELECT      *
FROM        ATTRACTION
WHERE       (PARK_CODE LIKE 'FR%'
AND ATTRACT_CAPACITY <50) OR (ATTRACT_CAPACITY > 100);
```



```
SQL> SELECT *
  2  FROM  ATTRACTION
  3  WHERE (PARK_CODE LIKE 'FR%'
  4  AND ATTRACT_CAPACITY <50) OR (ATTRACT_CAPACITY > 100);

ATTRACT_NO ATTRACT_NAME                        ATTRACT_AGE ATTRACT_CAPACITY PARK_CODE
---------- ----------------------------------- ----------- ---------------- ----------
     10034 ThunderCoaster                               11               34 FR1001
     10067 FlightToStars                                11               24 FR1001
     10078 Ant-Trap                                     23               30 FR1001
     10098 Carnival                                      3              120 FR1001
     20056 3D-Lego_Show                                  3              200 UK3452

SQL>
```

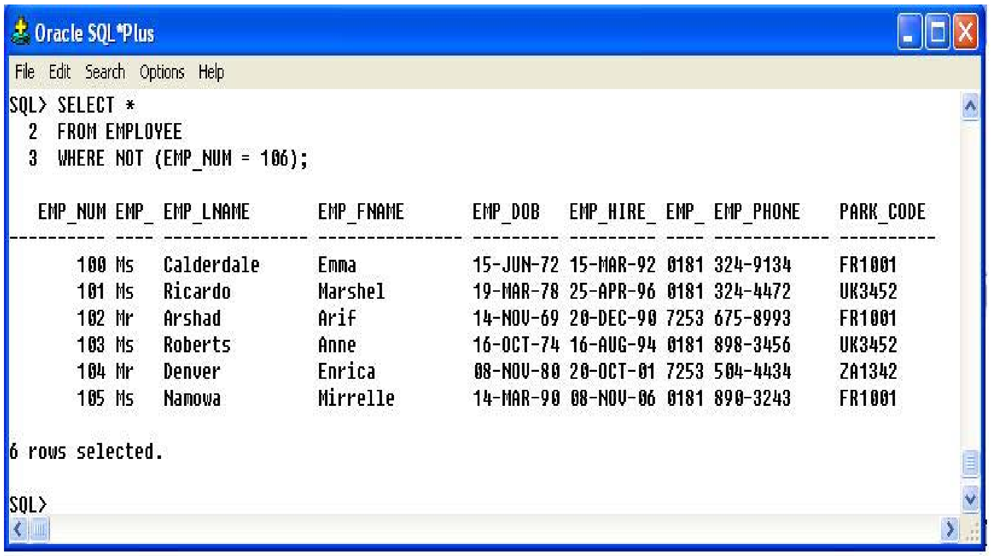
Figure 24: AND and OR example

NOT

The logical operator **NOT** is used to negate the result of a conditional expression. If you want to see a listing of all rows for which EMP_NUM is not 106, the query would look like:

```
SELECT      *
FROM EMPLOYEE
WHERE       NOT (EMP_NUM = 106);
```

The results of this query are shown in Figure 25. Note that the condition is enclosed in parentheses; that practice is optional, but it is highly recommended for clarity.



```
SQL> SELECT *
  2  FROM EMPLOYEE
  3  WHERE NOT (EMP_NUM = 106);

   EMP_NUM EMP_ EMP_LNAME       EMP_FNAME       EMP_DOB   EMP_HIRE_ EMP_ EMP_PHONE    PARK_CODE
---------- ---- --------------- --------------- --------- --------- ---- ------------ ----------
       100 Ms   Calderdale      Emma            15-JUN-72 15-MAR-92 0181 324-9134     FR1001
       101 Ms   Ricardo         Marshel         19-MAR-78 25-APR-96 0181 324-4472     UK3452
       102 Mr   Arshad          Arif            14-NOV-69 20-DEC-90 7253 675-8993     FR1001
       103 Ms   Roberts         Anne            16-OCT-74 16-AUG-94 0181 898-3456     UK3452
       104 Mr   Denver          Enrica          08-NOV-80 20-OCT-01 7253 504-4434     ZA1342
       105 Ms   Namowa          Mirrelle        14-MAR-90 08-NOV-06 0181 890-3243     FR1001

6 rows selected.

SQL>
```

Figure 25: Listing all employees except EMP_NUM=106

Exercises

E4.1 Write a query to display all Theme Parks except those in the UK.

E4.2 Write a query to display all the sales that occurred on the 19th May 2007.

E4.3 Write a query to display the ticket prices between €20 AND €30.

E4.4 Display all attractions that have a capacity of more that 60 at the Theme Park FR1001.

E4.5 Write a query to display the hourly rate for each attraction where an employee had worked, along with the hourly rate increased by 20%. Your query should only display the ATTRACT_NO, HOUR_RATE and the HOUR_RATE with the 20% increase.

Lab 5: Advanced SELECT Statements

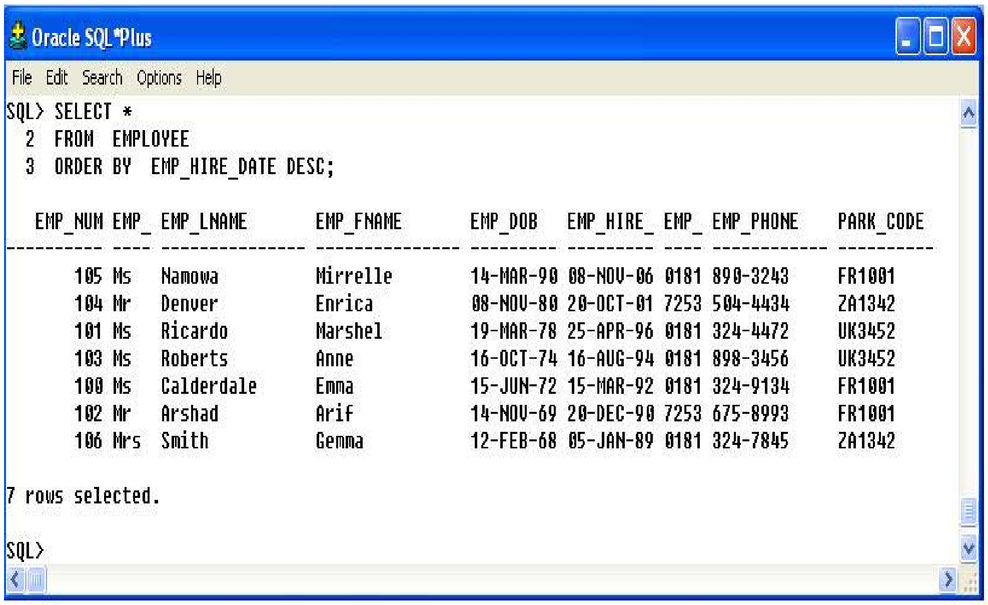
The learning objectives of this lab are to:

- Sort the data in the resulting query
- Apply SQL aggregate functions

5.1 Sorting Data

The **ORDER BY** clause is especially useful when the listing order of the query is important. Although you have the option of declaring the order type – ascending (**ASC**) or descending (**DESC**) – the default order is ascending. For example, if you want to display all employees listed by EMP_HIRE_DATE in descending order you would write the following query. The output is shown in Figure 26.

```
SELECT      *
FROM        EMPLOYEE
ORDER BY    EMP_HIRE_DATE DESC;
```



```
SQL> SELECT *
  2  FROM  EMPLOYEE
  3  ORDER BY  EMP_HIRE_DATE DESC;

   EMP_NUM EMP_ EMP_LNAME       EMP_FNAME       EMP_DOB   EMP_HIRE_ EMP_ EMP_PHONE    PARK_CODE
---------- ---- --------------- --------------- --------- --------- ---- ------------ ----------
       105 Ms   Namowa          Mirrelle        14-MAR-90 08-NOV-06 0181 890-3243     FR1001
       104 Mr   Denver          Enrica          08-NOV-80 20-OCT-01 7253 504-4434     ZA1342
       101 Ms   Ricardo         Marshel         19-MAR-78 25-APR-96 0181 324-4472     UK3452
       103 Ms   Roberts         Anne            16-OCT-74 16-AUG-94 0181 898-3456     UK3452
       100 Ms   Calderdale      Emma            15-JUN-72 15-MAR-92 0181 324-9134     FR1001
       102 Mr   Arshad          Arif            14-NOV-69 20-DEC-90 7253 675-8993     FR1001
       106 Mrs  Smith           Gemma           12-FEB-68 05-JAN-89 0181 324-7845     ZA1342

7 rows selected.

SQL>
```

Figure 26: Displaying all employees in descending order of EMP_HIRE_DATE

The ORDER BY command can also be used to produce a cascading order sequence. This is where the query results are ordered against a sequence of attributes.

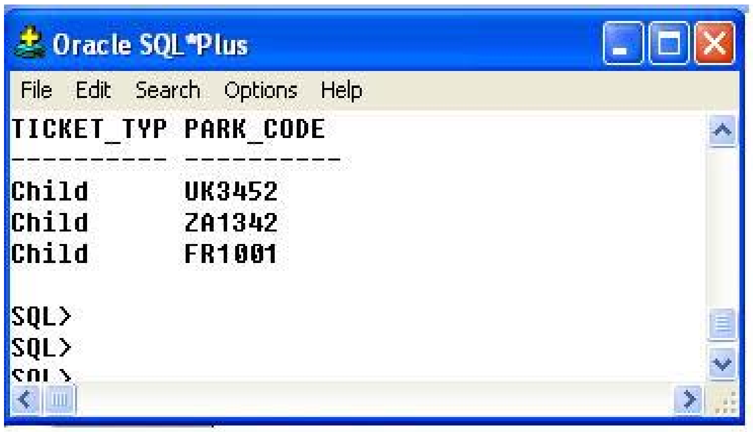
Task 5.1 Enter the following query, which contains an example of a cascading order sequence, by ordering the rows in the employee table by the employee's last then first names.

```
SELECT      *
FROM        EMPLOYEE
ORDER BY    EMP_LNAME, EMP_FNAME;
```

It is worth noting that if the ordering column has nulls, they are listed either first or last (depending on the RDBMS). The ORDER BY clause can be used in conjunction with other SQL commands and is listed last in the SELECT command sequence.

Task 5.2 Enter the following query and check your output against the results shown in Figure 27. Describe in your own words what this query is actually doing.

```
SELECT        TICKET_TYPE, PARK_CODE
FROM TICKET
WHERE (TICKET_PRICE > 15 AND TICKET_TYPE LIKE 'Child')
ORDER BY TICKET_NO DESC;
```



```
TICKET_TYP PARK_CODE
---------- ----------
Child      UK3452
Child      ZA1342
Child      FR1001

SQL>
SQL>
SQL>
```

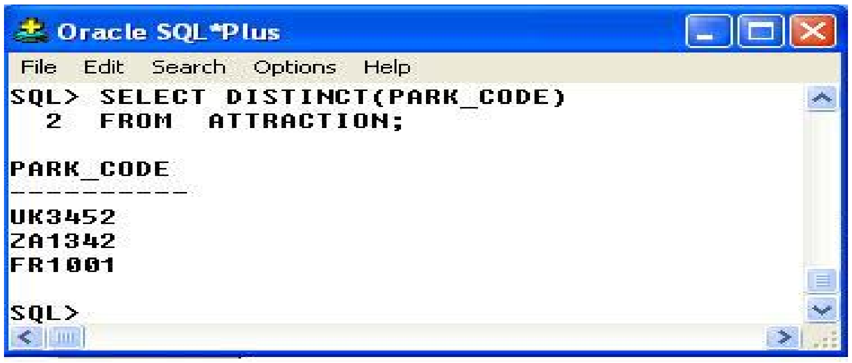
Figure 27: Query results for Task 5.2

5.2 Listing Unique Values

The SQL command DISTINCT is used to produce a list of only those values that are different from one another. For example, to list only the different Theme Parks from within the ATTRACTION table, you would enter the following query.

```
SELECT      DISTINCT(PARK_CODE)
FROM        ATTRACTION;
```

Figure 28 shows that the query only displays the rows that are different.



```
SQL> SELECT DISTINCT(PARK_CODE)
  2  FROM  ATTRACTION;

PARK_CODE
----------
UK3452
ZA1342
FR1001

SQL>
```

Figure 28: Displaying DISTINCT rows

5.3 Aggregate Functions

SQL can perform mathematical summaries through the use of aggregate (or group) functions. Aggregate functions return results based on groups of rows. By default, the entire result is treated as one group. Table 3 shows some of the basic aggregate functions.

Table 3 Basic SQL Aggregate Functions

FUNCTION	**OUTPUT**
COUNT	The number of rows containing non-null values

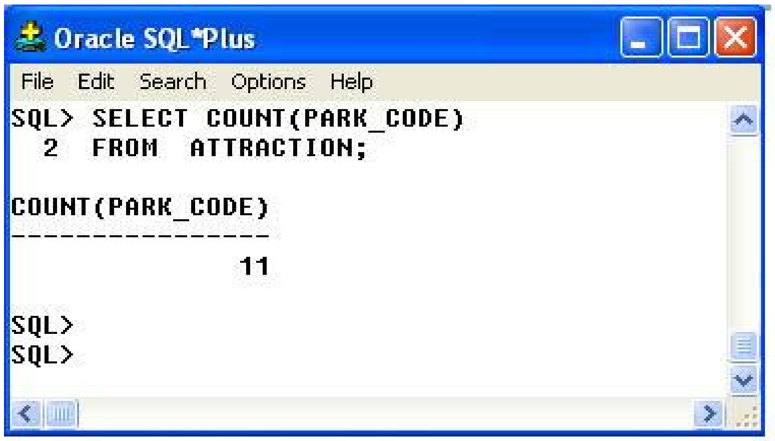
MIN	The minimum attribute value encountered in a given column
MAX	The maximum attribute value encountered in a given column
SUM	The sum of all values for a given column
AVG	The arithmetic mean (average) for a specified column

COUNT

The COUNT function is used to tally the number of non-null values of an attribute. COUNT can be used in conjunction with the DISTINCT clause. If you wanted to find out how many different Theme Parks contained attractions from the ATTRACTION table you would write the following query

```
SELECT      COUNT(PARK_CODE)
FROM        ATTRACTION;
```

The query would return 11 rows as shown in Figure 29.

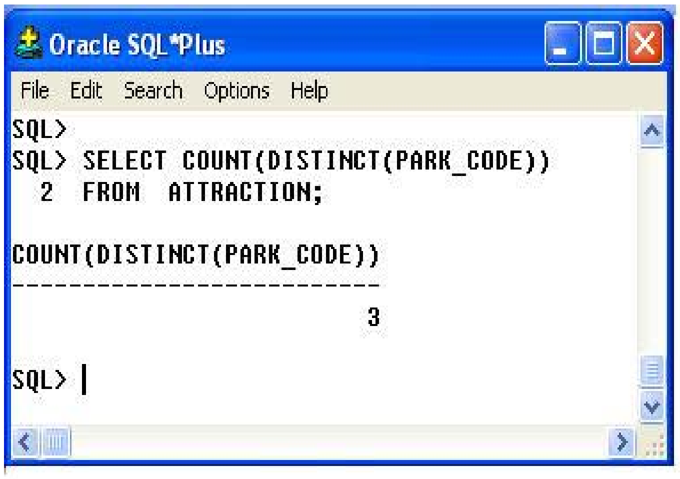


```
SQL> SELECT COUNT(PARK_CODE)
  2  FROM  ATTRACTION;

COUNT(PARK_CODE)
----------------
              11

SQL>
SQL>
```

Figure 29: Counting the number of Theme parks in ATTRACTION

However, if you wanted to know how many different Theme Parks were in the ATTRACTION table, you would modify the query as follows (For the output see Figure 30):

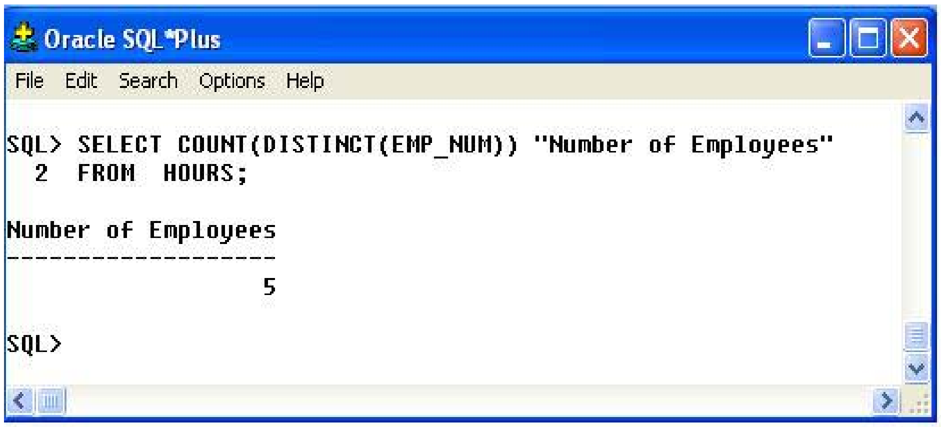
```
SELECT      COUNT(DISTINCT(PARK_CODE))
FROM        ATTRACTION;
```



```
SQL>
SQL> SELECT COUNT(DISTINCT(PARK_CODE))
  2  FROM  ATTRACTION;

COUNT(DISTINCT(PARK_CODE))
--------------------------
                         3

SQL> |
```

Figure 30: Counting the number of DISTINCT Theme parks in ATTRACTION

Task 5.3 Write a query that displays the number of distinct employees in the HOURS table. You should label the column “Number of Employees”. Your output should match that shown in Figure 31.



```
SQL> SELECT COUNT(DISTINCT(EMP_NUM)) "Number of Employees"
  2  FROM  HOURS;

Number of Employees
-------------------
                  5

SQL>
```

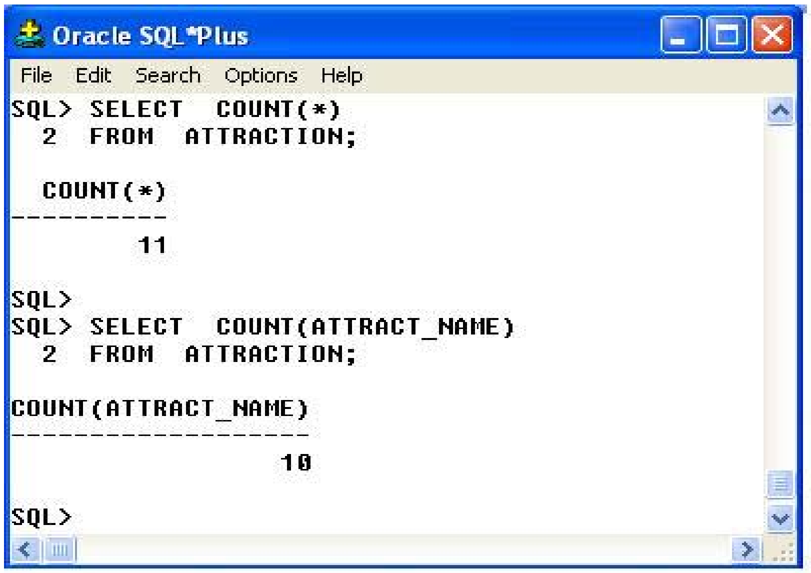
Figure 31: Query output for Task 5.3

COUNT always returns the number of non-null values in the given column. Another use for the COUNT function is to display the number of rows returned by a query, including the rows that contain rows using the syntax COUNT(*).

Task 5.4 Enter the following two queries and examine their output shown in Figure 32. Can you explain why the number of rows returned is different?

```
SELECT      COUNT(*)
FROM        ATTRACTION;
```

```
SELECT      COUNT(ATTRACT_NAME)
FROM        ATTRACTION;
```



```
SQL> SELECT  COUNT(*)
  2  FROM  ATTRACTION;

  COUNT(*)
----------
        11

SQL>
SQL> SELECT  COUNT(ATTRACT_NAME)
  2  FROM  ATTRACTION;

COUNT(ATTRACT_NAME)
-------------------
                 10

SQL>
```

Figure 32: Examples of using the COUNT function

MAX and MIN

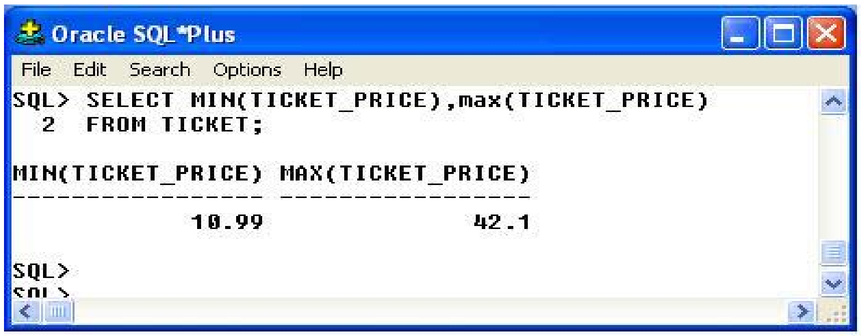
The MAX and MIN functions are used to find answers to problems such as,

"What is the highest and lowest ticket price sold in all Theme Parks?"

Task 5.5 Enter the following query which illustrates the use of the MIN and Max functions. Check the query results with those shown in Figure 33.

SELECT MIN(TICKET_PRICE),max(TICKET_PRICE)

FROM TICKET;



```
SQL> SELECT MIN(TICKET_PRICE),max(TICKET_PRICE)
  2  FROM TICKET;

MIN(TICKET_PRICE) MAX(TICKET_PRICE)
----------------- -----------------
            10.99              42.1

SQL>
SQL>
```

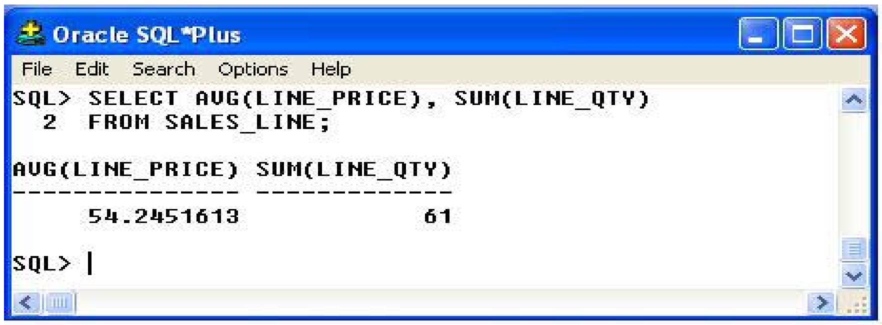
Figure 33: Examples of using the MIN and MAX functions

SUM and AVG

The SUM function computes the total sum for any specified attribute, using whatever condition(s) you have imposed. The AVG function calculates the arithmetic mean (average) for a specified attribute. The following query displays the average amount spent on Theme Park tickets per customer (LINE_PRICE) and the total number of tickets purchase (LINE_QTY). Figure 34 shows the output for this query.

SELECT AVG(LINE_PRICE), SUM(LINE_QTY)

FROM SALES_LINE;



```
SQL> SELECT AVG(LINE_PRICE), SUM(LINE_QTY)
  2  FROM SALES_LINE;

AVG(LINE_PRICE) SUM(LINE_QTY)
--------------- -------------
     54.2451613            61

SQL> |
```

Figure 34: Example showing the AVG and SUM functions

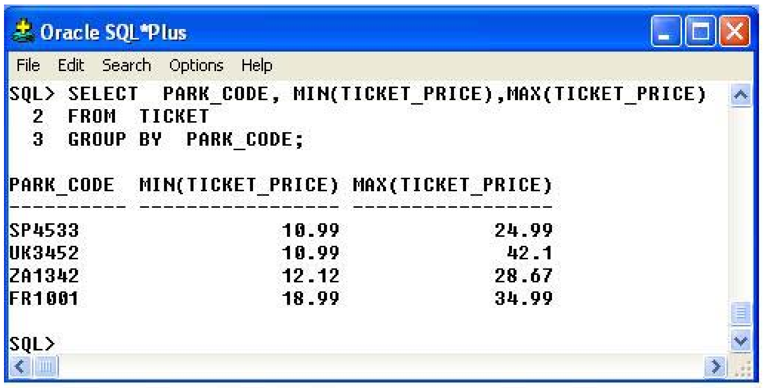
Task 5.6 Write a query that displays the average hourly rate that has been paid to all employees. Hint use the HOURS table. Your query should return €7.03.

Task 5.7 Write a query that displays the average attraction age for all attractions where the PARK_CODE = 'UK3452'. Your query should return 7.25 years.

GROUP BY

The GROUP BY clause is generally used when you have attribute columns combined with aggregate functions in the SELECT statement. It is valid only when used in conjunction with one of the SQL aggregate functions, such as COUNT, MIN, MAX, AVG and SUM. The GROUP BY clause appears after the WHERE statement. When using GROUP BY you should include all the attributes that are in the SELECT statement that do not use an aggregate function. The following query displays the minimum and maximum ticket price of all parks. The output is shown in Figure 35. Notice that the query groups only by the PARK_CODE, as no aggregate function is applied to this attribute in the SELECT statement.

```
SELECT       PARK_CODE, MIN(TICKET_PRICE),MAX(TICKET_PRICE)
FROM         TICKET
GROUP BY     PARK_CODE;
```



```
SQL> SELECT  PARK_CODE, MIN(TICKET_PRICE),MAX(TICKET_PRICE)
  2  FROM  TICKET
  3  GROUP BY  PARK_CODE;

PARK_CODE  MIN(TICKET_PRICE) MAX(TICKET_PRICE)
---------- ----------------- -----------------
SP4533                 10.99             24.99
UK3452                 10.99              42.1
ZA1342                 12.12             28.67
FR1001                 18.99             34.99

SQL>
```

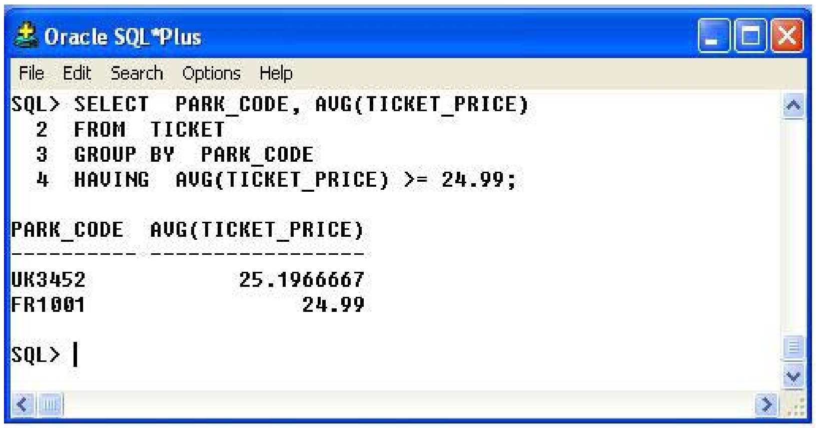
Figure 35: Displaying minimum and maximum ticket prices for each PARK_CODE

Task 5.7 Enter the query above and check the results against the output shown in Figure 35. What happens if you miss out the GROUP BY clause?

HAVING

The HAVING clause is an extension to the GROUP BY clause and is applied to the output of a GROUP BY operation. Supposing you wanted to list the average ticket price at each Theme Park but wanted to limit the listing to Theme Parks whose average ticket price was greater or equal to €24.99. This can be achieved by the following query whose output is shown in Figure 36.

```
SELECT      PARK_CODE, AVG(TICKET_PRICE)
FROM        TICKET
GROUP BY    PARK_CODE
HAVING      AVG(TICKET_PRICE) >= 24.99;
```



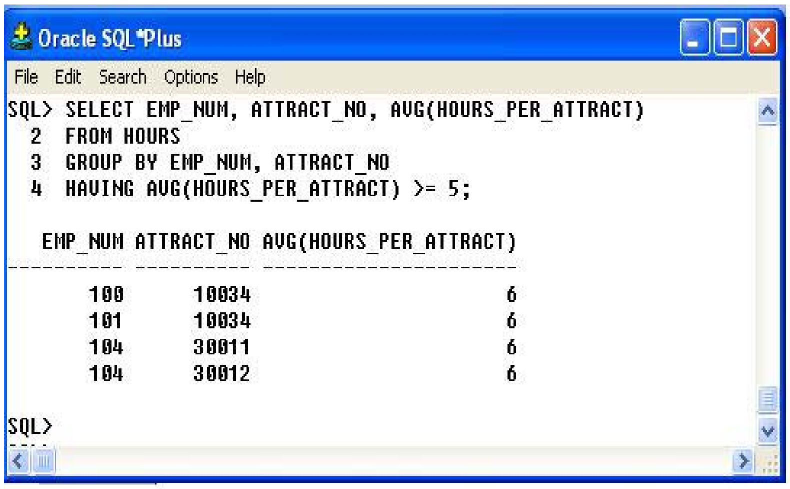
Oracle SQL*Plus

File Edit Search Options Help

```
SQL> SELECT  PARK_CODE, AVG(TICKET_PRICE)
  2  FROM  TICKET
  3  GROUP BY  PARK_CODE
  4  HAVING  AVG(TICKET_PRICE) >= 24.99;

PARK_CODE  AVG(TICKET_PRICE)
---------- -----------------
UK3452            25.1966667
FR1001                 24.99

SQL> |
```

Figure 36: Example of the HAVING clause

Task 5.8 Using the HOURS table, write a query to display the employee number (EMP_NUM), the attraction number (ATTRACT-NO) and the average hours worked per attraction (HOURS_PER_ATTRACT), limiting the result to where the average hours worked per attraction is greater or equal to 5. Check your results against those shown in Figure 37.



Oracle SQL*Plus

File Edit Search Options Help

```
SQL> SELECT EMP_NUM, ATTRACT_NO, AVG(HOURS_PER_ATTRACT)
  2  FROM HOURS
  3  GROUP BY EMP_NUM, ATTRACT_NO
  4  HAVING AVG(HOURS_PER_ATTRACT) >= 5;

   EMP_NUM ATTRACT_NO AVG(HOURS_PER_ATTRACT)
---------- ---------- ----------------------
       100      10034                      6
       101      10034                      6
       104      30011                      6
       104      30012                      6

SQL>
```

Figure 37: Query output for Task 5.8

5.4 Exercises

E5.1 Write a query to display all unique employees that exist in the HOURS table.

E5.2 Display the employee numbers of all employees and the total number of hours they have worked.

E5.3. Show the attraction number and the minimum and maximum hourly rate for each attraction.

E5.4 Write a query to show the transaction numbers and line prices (in the SALES_LINE table) that are greater than €50.

E5.5 Display all information from the SALES table in descending order of the sale date.

Lab 6: JOINING DATABASE TABLES

The learning objectives of this lab are to:

- Learn how to perform the following types of database joins
  - Cross Join
  - Natural Join
  - Outer Joins

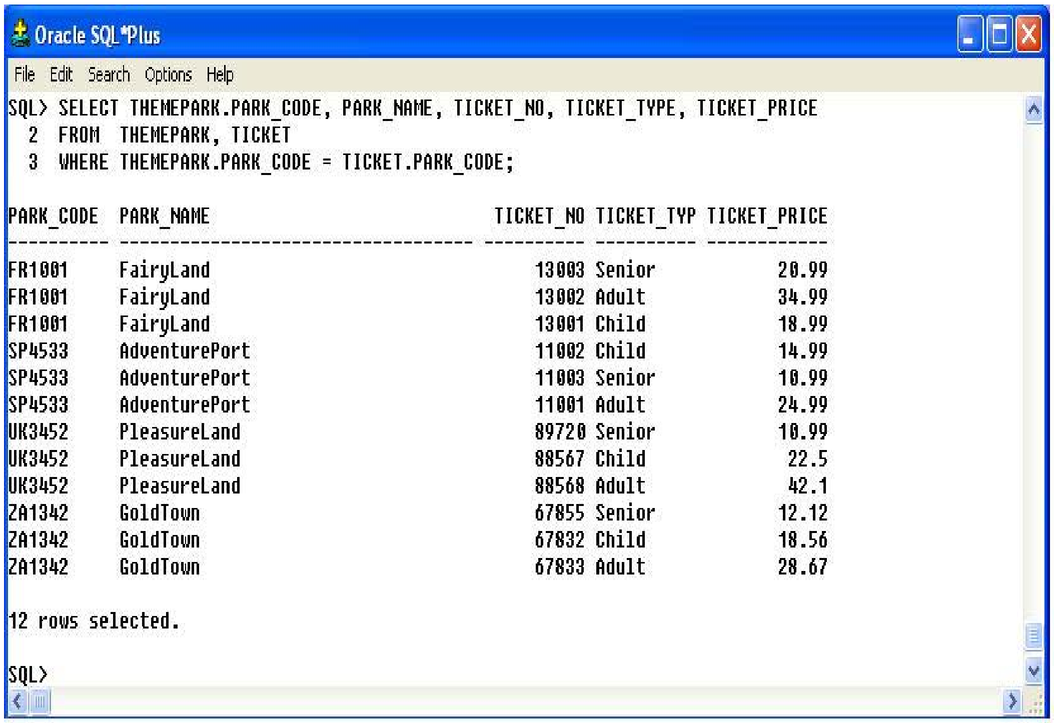
6.1 Introduction to Joins

The relational join operation merges rows from two or more tables and returns the rows with one of the following conditions:

- Have common values in common columns (natural join)
- Meet a given join condition (equality or inequality)
- Have common values in common columns or have no matching values (outer join)

There are a number of different joins that can be performed. The most common is the natural join. To join tables, you simply enumerate the tables in the FROM clause of the SELECT statement. The DBMS will create the Cartesian product of every table in the FROM clause. However, to get the correct result – that is, a natural join – you must select only the rows in which the common attribute values match. That is done with the WHERE clause. Use the WHERE clause to indicate the common attributes that are used to link the tables (sometimes referred to as the *join condition*). For example, suppose you want to join the two tables THEMEPARK and TICKET. Because PARK_CODE is the foreign key in the TICKET table and the primary key in the THEMEPARK table, the link is established on PARK_CODE. It is important to note that when the same attribute name appears in more than one of the joined tables, the source table of the attributes listed in the SELECT command sequence must be defined. To join the THEMEPARK and TICKET tables you would use the following, which produces the output shown in Figure 38.

```
SELECT      THEMEPARK.PARK_CODE, PARK_NAME, TICKET_NO,
            TICKET_TYPE, TICKET_PRICE
FROM        THEMEPARK, TICKET
WHERE       THEMEPARK.PARK_CODE = TICKET.PARK_CODE;
```



```
SQL> SELECT THEMEPARK.PARK_CODE, PARK_NAME, TICKET_NO, TICKET_TYPE, TICKET_PRICE
  2  FROM  THEMEPARK, TICKET
  3  WHERE THEMEPARK.PARK_CODE = TICKET.PARK_CODE;

PARK_CODE  PARK_NAME                         TICKET_NO TICKET_TYP TICKET_PRICE
---------- ---------------------------------- ---------- ---------- ------------
FR1001     FairyLand                              13003 Senior            20.99
FR1001     FairyLand                              13002 Adult             34.99
FR1001     FairyLand                              13001 Child             18.99
SP4533     AdventurePort                          11002 Child             14.99
SP4533     AdventurePort                          11003 Senior            10.99
SP4533     AdventurePort                          11001 Adult             24.99
UK3452     PleasureLand                           89720 Senior            10.99
UK3452     PleasureLand                           88567 Child              22.5
UK3452     PleasureLand                           88568 Adult              42.1
ZA1342     GoldTown                               67855 Senior            12.12
ZA1342     GoldTown                               67832 Child             18.56
ZA1342     GoldTown                               67833 Adult             28.67

12 rows selected.

SQL>
```

Figure 38: Natural Join between THEMEPARK and TICKET tables

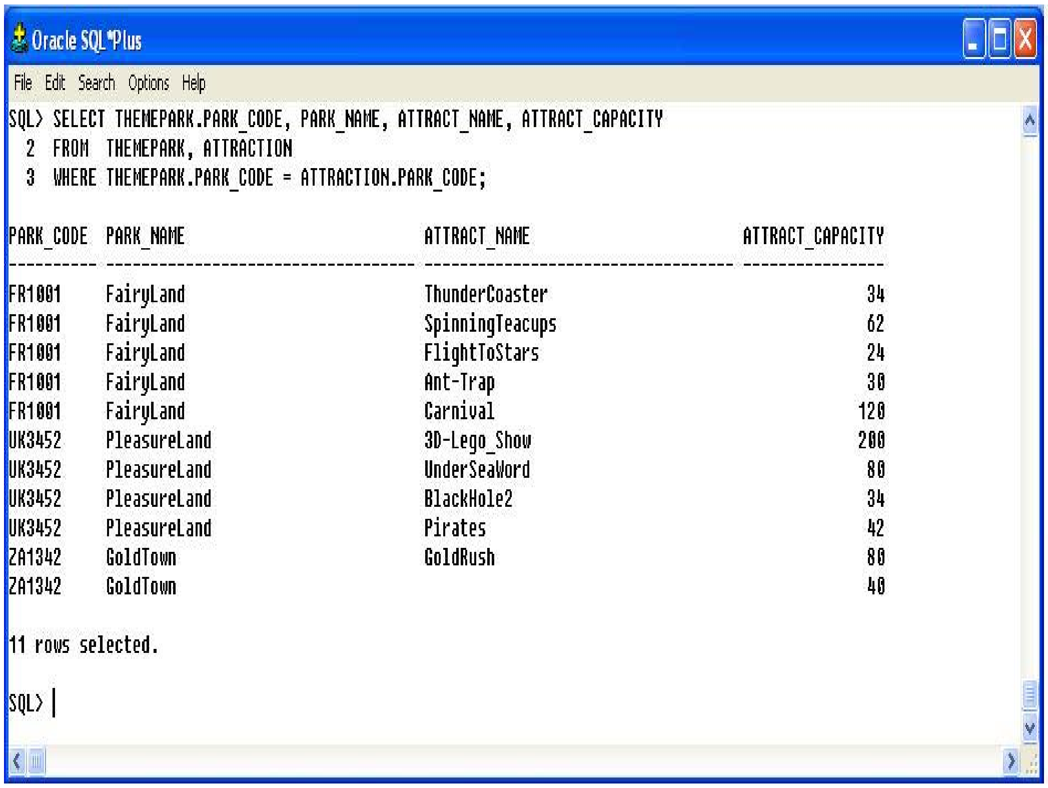
As you examine the preceding query, note the following points:

- The FROM clause indicates which tables are to be joined. If three or more tables are included, the join operation takes place two tables at a time, starting from left to right. For example, if you are joining tables T1, T2, and T3, first table T1 is joined to T2. The results of that join are then joined to table T3.
- The join condition in the WHERE clause tells the SELECT statement which rows will be returned. In this case, the SELECT statement returns all rows for which the PARK_CODE values in the PRODUCT and VENDOR tables are equal.
- The number of join conditions is always equal to the number of tables being joined minus one. For example, if you join three tables (T1, T2, and T3), you will have two join conditions (j1 and j2). All join conditions are connected through an AND logical operator. The first join condition (j1) defines the join criteria for T1 and T2. The second join condition (j2) defines the join criteria for the output of the first join and table T3.
- Generally, the join condition will be an equality comparison of the primary key in one table and the related foreign key in the second table.

Task 6.1 Execute the following query and check you results with those shown in Figure 39. Then modify the SELECT statement and change THEMEPARK.PARK_CODE to just PARK_CODE. What happens?

```
SELECT      THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME,
            ATTRACT_CAPACITY
FROM        THEMEPARK, ATTRACTION
```

```
WHERE       THEMEPARK.PARK_CODE = ATTRACTION.PARK_CODE;
```



```
SQL> SELECT THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME, ATTRACT_CAPACITY
  2  FROM  THEMEPARK, ATTRACTION
  3  WHERE THEMEPARK.PARK_CODE = ATTRACTION.PARK_CODE;

PARK_CODE  PARK_NAME                           ATTRACT_NAME                       ATTRACT_CAPACITY
---------- ----------------------------------- ---------------------------------- ----------------
FR1001     FairyLand                           ThunderCoaster                                   34
FR1001     FairyLand                           SpinningTeacups                                  62
FR1001     FairyLand                           FlightToStars                                    24
FR1001     FairyLand                           Ant-Trap                                         30
FR1001     FairyLand                           Carnival                                        120
UK3452     PleasureLand                        3D-Lego_Show                                    200
UK3452     PleasureLand                        UnderSeaWord                                     80
UK3452     PleasureLand                        BlackHole2                                       34
UK3452     PleasureLand                        Pirates                                          42
ZA1342     GoldTown                            GoldRush                                         80
ZA1342     GoldTown                                                                             40

11 rows selected.

SQL> |
```

Figure 39: Query output for task 6.1

6.2 Joining tables with an alias

An alias may be used to identify the source table from which the data are taken. For example, the aliases P and T can be used to label the THEMEPARK and TICKET tables as shown in the query below (which produces the same output as shown in Figure 39). Any legal table name may be used as an alias.

```
SELECT      P.PARK_CODE, PARK_NAME, TICKET_NO, TICKET_TYPE,
            TICKET_PRICE

FROM        THEMEPARK P, TICKET T
```

```
WHERE         P.PARK_CODE =T.PARK_CODE;
```

6.3 Cross Join

A **cross join** performs a relational product (also known as the Cartesian product) of two tables. The cross join syntax is:

SELECT *column-list* FROM *table1* CROSS JOIN *table2*

For example:

```
SELECT * FROM SALES CROSS JOIN SALES_LINE;
```

performs a cross join of the SALES and SALES_LINE tables. That CROSS JOIN query generates 589 rows. (There were 19 sales rows and 31 SALES_LINE rows, thus giving 19 × 31 = 589 rows.)

Task 6.2 Write a CROSS JOIN query which selects all rows from the EMPLOYEE and HOURS tables. How many rows were returned?

6.4 Natural Join

The natural join returns all rows with matching values in the matching columns and eliminates duplicate columns. That style of query is used when the tables share one or more common attributes with common names. The natural join syntax is:

SELECT *column-list* FROM *table1* NATURAL JOIN *table2*

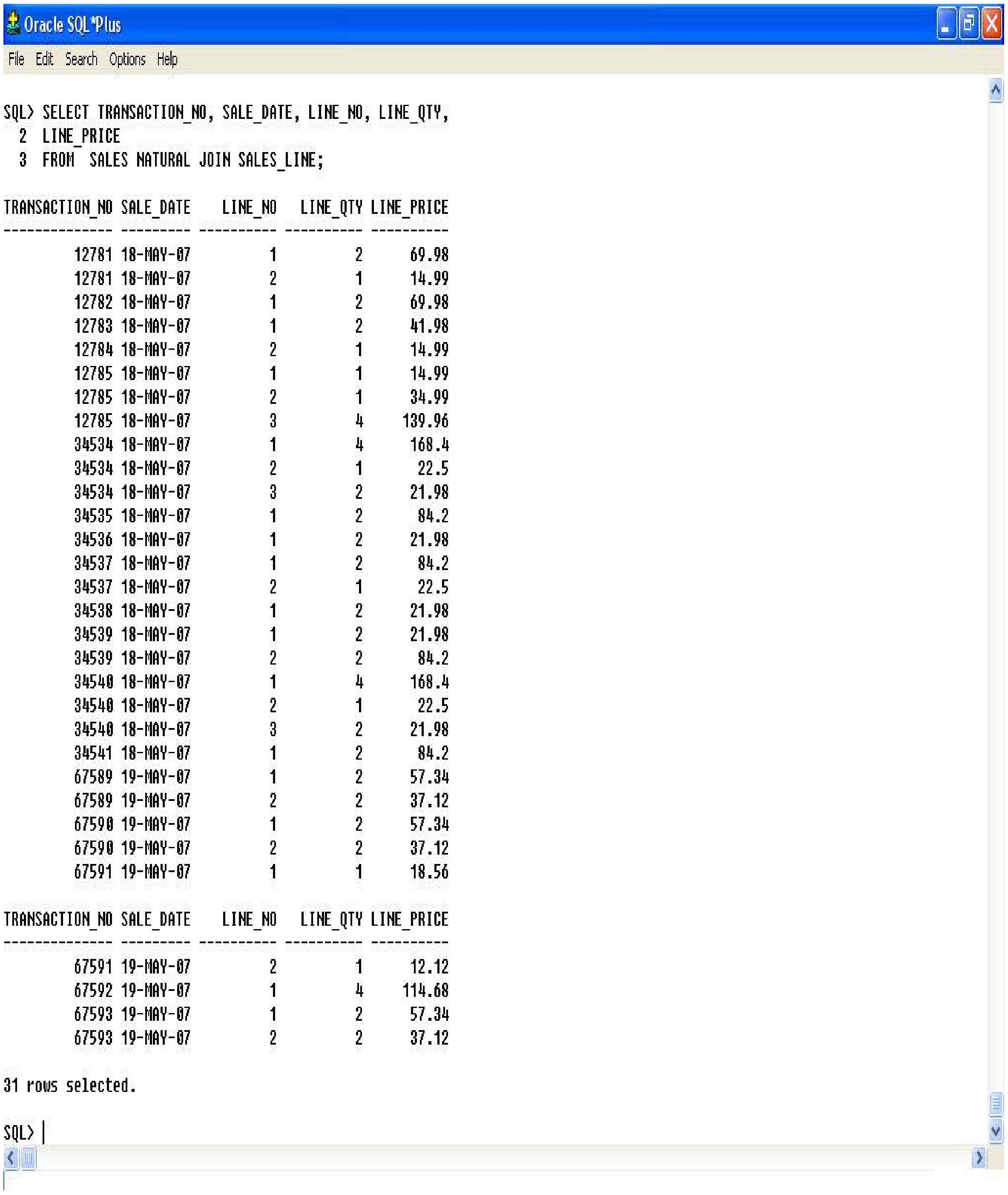
The natural join will perform the following tasks:

- Determine the common attribute(s) by looking for attributes with identical names and compatible data types
- Select only the rows with common values in the common attribute(s)
- If there are no common attributes, return the relational product of the two tables

The following example performs a natural join of the SALES and SALES_LINE tables and returns only selected attributes:

```
SELECT      TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY,
            LINE_PRICE
FROM        SALES NATURAL JOIN SALES_LINE;
```

The results of this query can be seen in Figure 40.

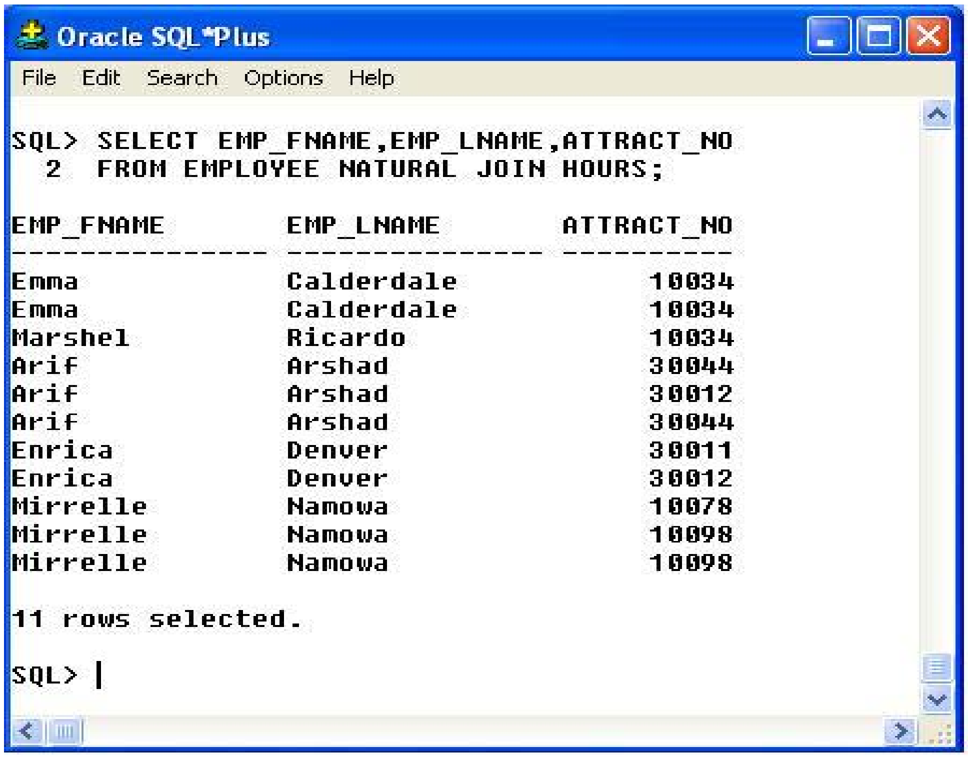


```
SQL> SELECT TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY,
  2  LINE_PRICE
  3  FROM  SALES NATURAL JOIN SALES_LINE;

TRANSACTION_NO SALE_DATE    LINE_NO   LINE_QTY LINE_PRICE
-------------- --------- ---------- ---------- ----------
         12781 18-MAY-07          1          2      69.98
         12781 18-MAY-07          2          1      14.99
         12782 18-MAY-07          1          2      69.98
         12783 18-MAY-07          1          2      41.98
         12784 18-MAY-07          2          1      14.99
         12785 18-MAY-07          1          1      14.99
         12785 18-MAY-07          2          1      34.99
         12785 18-MAY-07          3          4     139.96
         34534 18-MAY-07          1          4      168.4
         34534 18-MAY-07          2          1       22.5
         34534 18-MAY-07          3          2      21.98
         34535 18-MAY-07          1          2       84.2
         34536 18-MAY-07          1          2      21.98
         34537 18-MAY-07          1          2       84.2
         34537 18-MAY-07          2          1       22.5
         34538 18-MAY-07          1          2      21.98
         34539 18-MAY-07          1          2      21.98
         34539 18-MAY-07          2          2       84.2
         34540 18-MAY-07          1          4      168.4
         34540 18-MAY-07          2          1       22.5
         34540 18-MAY-07          3          2      21.98
         34541 18-MAY-07          1          2       84.2
         67589 19-MAY-07          1          2      57.34
         67589 19-MAY-07          2          2      37.12
         67590 19-MAY-07          1          2      57.34
         67590 19-MAY-07          2          2      37.12
         67591 19-MAY-07          1          1      18.56

TRANSACTION_NO SALE_DATE    LINE_NO   LINE_QTY LINE_PRICE
-------------- --------- ---------- ---------- ----------
         67591 19-MAY-07          2          1      12.12
         67592 19-MAY-07          1          4     114.68
         67593 19-MAY-07          1          2      57.34
         67593 19-MAY-07          2          2      37.12

31 rows selected.

SQL> |
```

Figure 40: Results of SALES NATURAL JOIN SALES_LINE

One important difference between the natural join and the "old-style" join syntax as illustrated in Figure 33, Section 6.1, is that the NATURAL JOIN command does not require the use of a table qualifier for the common attributes.

Task 6.3 Write a query that displays the employee's first and last name (EMP_FNAME and EMP_LNAME), the attraction number (ATTRACT_NO) and the date worked. **Hint**: You will have to join the HOURS and the EMPLOYEE tables. Check your results with those shown in Figure 41.



```
SQL> SELECT EMP_FNAME,EMP_LNAME,ATTRACT_NO
  2  FROM EMPLOYEE NATURAL JOIN HOURS;

EMP_FNAME       EMP_LNAME       ATTRACT_NO
--------------- --------------- ----------
Emma            Calderdale           10034
Emma            Calderdale           10034
Marshel         Ricardo              10034
Arif            Arshad               30044
Arif            Arshad               30012
Arif            Arshad               30044
Enrica          Denver               30011
Enrica          Denver               30012
Mirrelle        Namowa               10078
Mirrelle        Namowa               10098
Mirrelle        Namowa               10098

11 rows selected.

SQL> |
```

Figure 41: Query results for Task 6.3

6.5 Join USING

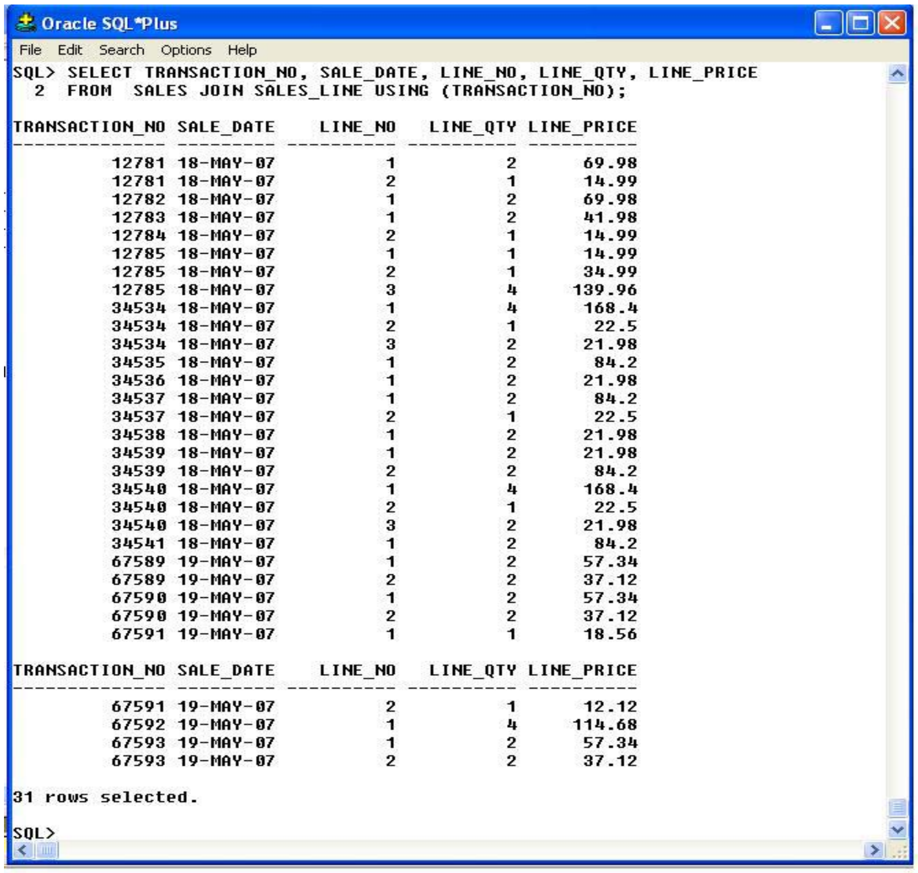
A second way to express a join is through the USING keyword. That query returns only the rows with matching values in the column indicated in the USING clause – and that column must exist in both tables. The syntax is:

SELECT *column-list* FROM *table1* JOIN *table2* USING (*common-column*)

To see the JOIN USING query in action, let's perform a join of the SALES and SALEs_LINE tables by writing:

```
SELECT      TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY,
            LINE_PRICE
FROM        SALES JOIN SALES_LINE USING (TRANSACTION_NO);
```

The SQL statement produces the results shown in Figure 42.

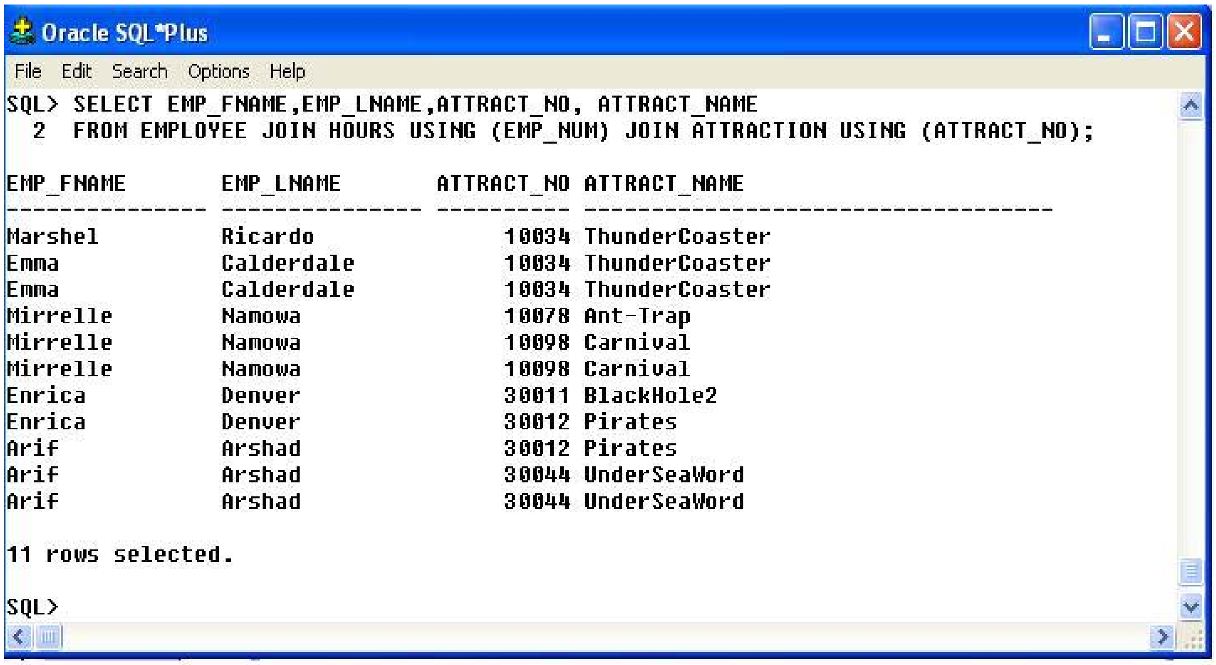


```
SQL> SELECT TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY, LINE_PRICE
  2  FROM  SALES JOIN SALES_LINE USING (TRANSACTION_NO);

TRANSACTION_NO SALE_DATE    LINE_NO   LINE_QTY LINE_PRICE
-------------- --------- ---------- ---------- ----------
         12781 18-MAY-07          1          2      69.98
         12781 18-MAY-07          2          1      14.99
         12782 18-MAY-07          1          2      69.98
         12783 18-MAY-07          1          2      41.98
         12784 18-MAY-07          2          1      14.99
         12785 18-MAY-07          1          1      14.99
         12785 18-MAY-07          2          1      34.99
         12785 18-MAY-07          3          4     139.96
         34534 18-MAY-07          1          4      168.4
         34534 18-MAY-07          2          1       22.5
         34534 18-MAY-07          3          2      21.98
         34535 18-MAY-07          1          2       84.2
         34536 18-MAY-07          1          2      21.98
         34537 18-MAY-07          1          2       84.2
         34537 18-MAY-07          2          1       22.5
         34538 18-MAY-07          1          2      21.98
         34539 18-MAY-07          1          2      21.98
         34539 18-MAY-07          2          2       84.2
         34540 18-MAY-07          1          4      168.4
         34540 18-MAY-07          2          1       22.5
         34540 18-MAY-07          3          2      21.98
         34541 18-MAY-07          1          2       84.2
         67589 19-MAY-07          1          2      57.34
         67589 19-MAY-07          2          2      37.12
         67590 19-MAY-07          1          2      57.34
         67590 19-MAY-07          2          2      37.12
         67591 19-MAY-07          1          1      18.56

TRANSACTION_NO SALE_DATE    LINE_NO   LINE_QTY LINE_PRICE
-------------- --------- ---------- ---------- ----------
         67591 19-MAY-07          2          1      12.12
         67592 19-MAY-07          1          4     114.68
         67593 19-MAY-07          1          2      57.34
         67593 19-MAY-07          2          2      37.12

31 rows selected.

SQL>
```

Figure 42: Query results for SALES JOIN SALES_LINE USING TRANSACTION_NO

As was the case with the NATURAL JOIN command, the JOIN USING operand does not require table qualifiers. As a matter of fact, ORACLE will return an error if you specify the table name in the USING clause.

Task 6.4 Rewrite the query you wrote in **Task 6.3** so that the attraction name (ATTRACT_NAME located in the ATTRACTION table) is also displayed. Express the joins through the USING keyword. Hint: You will need to join three tables. Your output should match that shown in Figure 43.



```
SQL> SELECT EMP_FNAME,EMP_LNAME,ATTRACT_NO, ATTRACT_NAME
  2  FROM EMPLOYEE JOIN HOURS USING (EMP_NUM) JOIN ATTRACTION USING (ATTRACT_NO);

EMP_FNAME       EMP_LNAME       ATTRACT_NO ATTRACT_NAME
--------------- --------------- ---------- -----------------------------------
Marshel         Ricardo              10034 ThunderCoaster
Emma            Calderdale           10034 ThunderCoaster
Emma            Calderdale           10034 ThunderCoaster
Mirrelle        Namowa               10078 Ant-Trap
Mirrelle        Namowa               10098 Carnival
Mirrelle        Namowa               10098 Carnival
Enrica          Denver               30011 BlackHole2
Enrica          Denver               30012 Pirates
Arif            Arshad               30012 Pirates
Arif            Arshad               30044 UnderSeaWord
Arif            Arshad               30044 UnderSeaWord

11 rows selected.

SQL>
```

Figure 43: Query results for Task 6.4

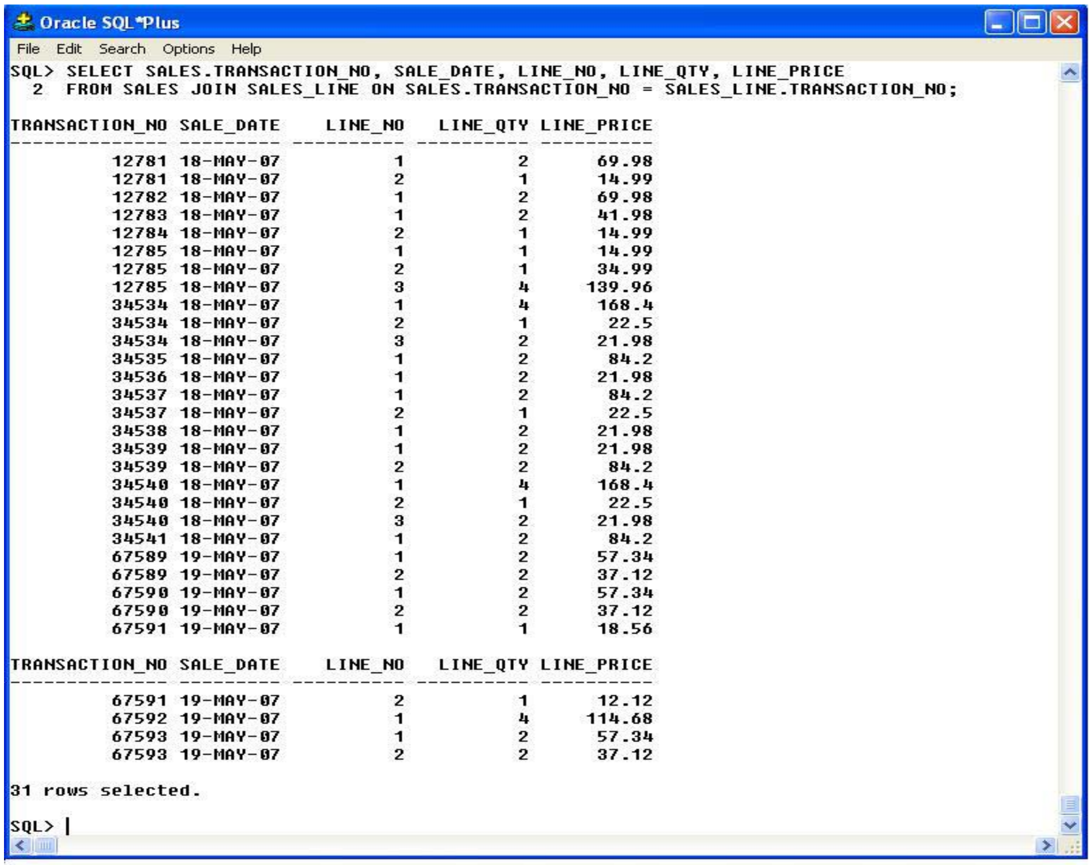
6.6 Join ON

The previous two join styles used common attribute names in the joining tables. Another way to express a join when the tables have no common attribute names is to use the JOIN ON operand. That query will return only the rows that meet the indicated join condition. The join condition will typically include an equality comparison expression of two columns. (The columns may or may not share the same name but, obviously, must have comparable data types.) The syntax is:

SELECT *column-list* FROM *table1* JOIN *table2* ON *join-condition*

The following example performs a join of the SALES and SALES_LINE tables, using the ON clause. The result is shown in Figure 44.

```
SELECT      SALES.TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY,
            LINE_PRICE
FROM        SALES JOIN SALES_LINE ON SALES.TRANSACTION_NO =
            SALES_LINE.TRANSACTION_NO;
```



```
SQL> SELECT SALES.TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY, LINE_PRICE
  2  FROM SALES JOIN SALES_LINE ON SALES.TRANSACTION_NO = SALES_LINE.TRANSACTION_NO;

TRANSACTION_NO SALE_DATE    LINE_NO   LINE_QTY LINE_PRICE
-------------- --------- ---------- ---------- ----------
         12781 18-MAY-07          1          2      69.98
         12781 18-MAY-07          2          1      14.99
         12782 18-MAY-07          1          2      69.98
         12783 18-MAY-07          1          2      41.98
         12784 18-MAY-07          2          1      14.99
         12785 18-MAY-07          1          1      14.99
         12785 18-MAY-07          2          1      34.99
         12785 18-MAY-07          3          4     139.96
         34534 18-MAY-07          1          4      168.4
         34534 18-MAY-07          2          1       22.5
         34534 18-MAY-07          3          2      21.98
         34535 18-MAY-07          1          2       84.2
         34536 18-MAY-07          1          2      21.98
         34537 18-MAY-07          1          2       84.2
         34537 18-MAY-07          2          1       22.5
         34538 18-MAY-07          1          2      21.98
         34539 18-MAY-07          1          2      21.98
         34539 18-MAY-07          2          2       84.2
         34540 18-MAY-07          1          4      168.4
         34540 18-MAY-07          2          1       22.5
         34540 18-MAY-07          3          2      21.98
         34541 18-MAY-07          1          2       84.2
         67589 19-MAY-07          1          2      57.34
         67589 19-MAY-07          2          2      37.12
         67590 19-MAY-07          1          2      57.34
         67590 19-MAY-07          2          2      37.12
         67591 19-MAY-07          1          1      18.56

TRANSACTION_NO SALE_DATE    LINE_NO   LINE_QTY LINE_PRICE
-------------- --------- ---------- ---------- ----------
         67591 19-MAY-07          2          1      12.12
         67592 19-MAY-07          1          4     114.68
         67593 19-MAY-07          1          2      57.34
         67593 19-MAY-07          2          2      37.12

31 rows selected.

SQL> |
```

Figure 44: Query results for SALES JOIN SALES_LINE ON

Note that unlike the NATURAL JOIN and the JOIN USING operands, the JOIN ON clause requires a table qualifier for the common attributes. If you do not specify the table qualifier, you will get a “column ambiguously defined” error message.

6.7 The Outer Join

An outer join returns not only the rows matching the join condition (that is, rows with matching values in the common columns), but also the rows with unmatched values. The ANSI standard defines three types of outer joins: left, right, and full. The left and right designations reflect the order in which the tables are processed by the DBMS. Remember that join operations take place two tables at a time. The first table named in the FROM clause will be the left side, and the second table named will be the right side. If three or more tables are being joined, the result of joining the first two tables becomes the left side; the third table becomes the right side.

LEFT OUTER JOIN

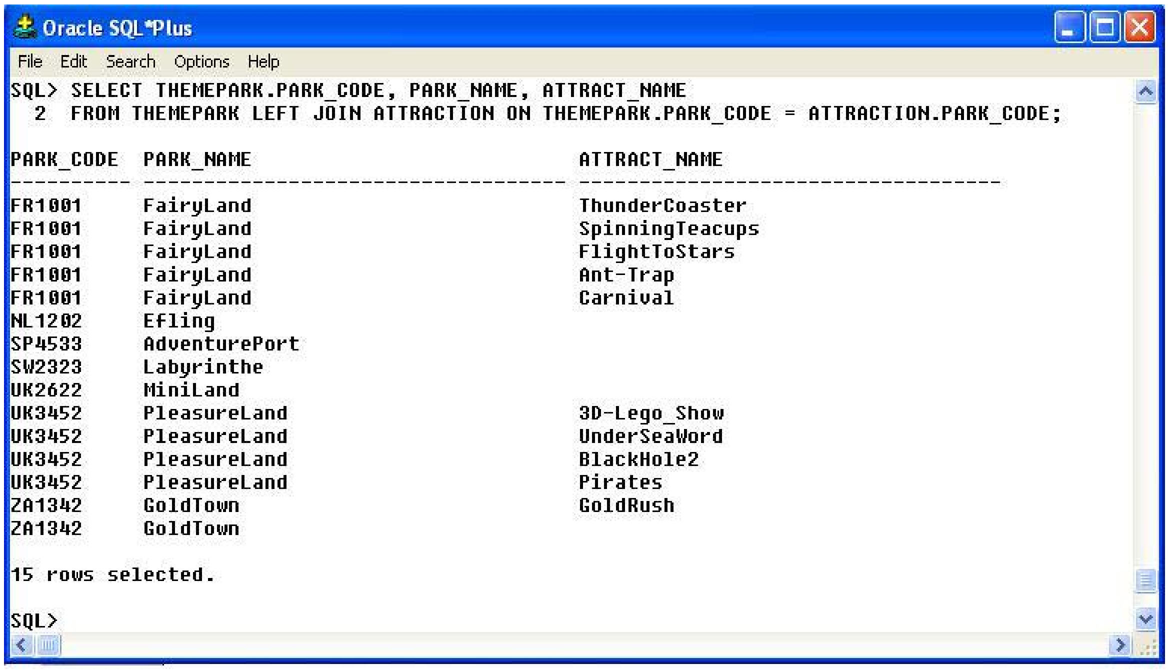
The left outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column), but also the rows in the left side table with unmatched values in the right side table. The syntax is:

```
SELECT      column-list
FROM        table1 LEFT [OUTER] JOIN table2 ON join-condition
```

For example, the following query lists the park code, park name, and attraction name for all attractions and includes those Theme Parks with no currently listed attractions:

```
SELECT      THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME
FROM        THEMEPARK LEFT JOIN ATTRACTION ON
            THEMEPARK.PARK_CODE = ATTRACTION.PARK_CODE;
```

The results of this query are shown in Figure 45.



```
SQL> SELECT THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME
  2  FROM THEMEPARK LEFT JOIN ATTRACTION ON THEMEPARK.PARK_CODE = ATTRACTION.PARK_CODE;

PARK_CODE  PARK_NAME                            ATTRACT_NAME
---------- ------------------------------------ -----------------------------------
FR1001     FairyLand                            ThunderCoaster
FR1001     FairyLand                            SpinningTeacups
FR1001     FairyLand                            FlightToStars
FR1001     FairyLand                            Ant-Trap
FR1001     FairyLand                            Carnival
NL1202     Efling
SP4533     AdventurePort
SW2323     Labyrinthe
UK2622     MiniLand
UK3452     PleasureLand                         3D-Lego_Show
UK3452     PleasureLand                         UnderSeaWord
UK3452     PleasureLand                         BlackHole2
UK3452     PleasureLand                         Pirates
ZA1342     GoldTown                             GoldRush
ZA1342     GoldTown

15 rows selected.

SQL>
```

Figure 45: LEFT OUTER JOIN example

Task 6.5 Enter the query above and check your results with those shown in Figure 45.

RIGHT OUTER JOIN

The right outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column), but also the rows in the right side table with unmatched values in the left side table. The syntax is:

SELECT column-list

FROM *table1* RIGHT [OUTER] JOIN *table2* ON *join-condition*

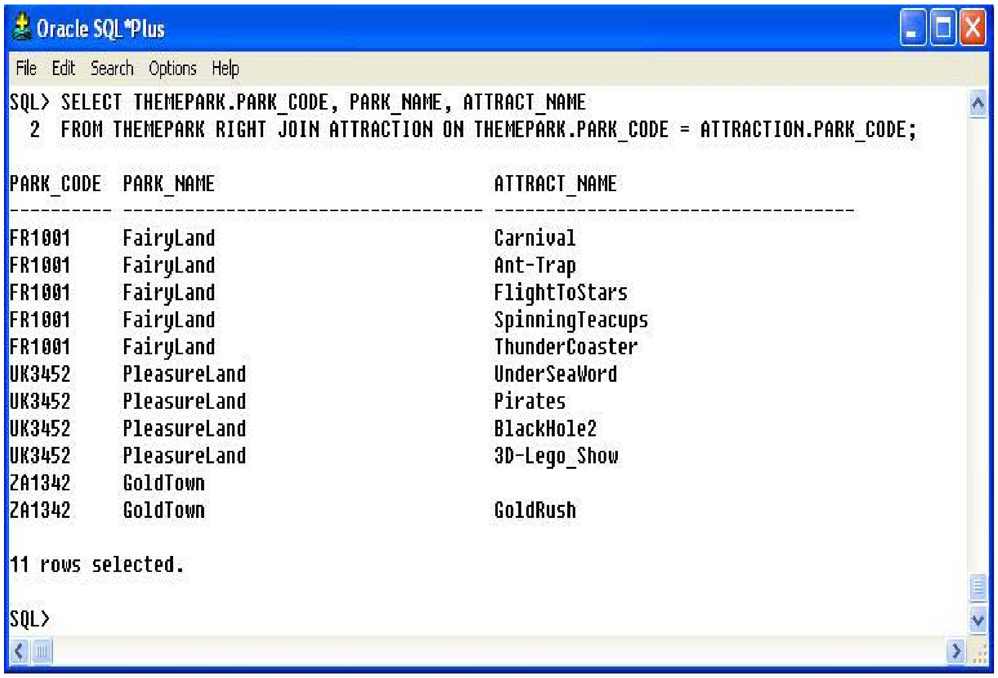
For example, the following query lists the park code, park name, and attraction name for all attractions and also includes those attractions that do not have a matching park code:

SELECT THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME

FROM THEMEPARK RIGHT JOIN ATTRACTION ON

THEMEPARK.PARK_CODE = ATTRACTION.PARK_CODE;

The results of this query are shown in Figure 46.



```
SQL> SELECT THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME
  2  FROM THEMEPARK RIGHT JOIN ATTRACTION ON THEMEPARK.PARK_CODE = ATTRACTION.PARK_CODE;

PARK_CODE  PARK_NAME                           ATTRACT_NAME
---------- ----------------------------------- -----------------------------------
FR1001     FairyLand                           Carnival
FR1001     FairyLand                           Ant-Trap
FR1001     FairyLand                           FlightToStars
FR1001     FairyLand                           SpinningTeacups
FR1001     FairyLand                           ThunderCoaster
UK3452     PleasureLand                        UnderSeaWord
UK3452     PleasureLand                        Pirates
UK3452     PleasureLand                        BlackHole2
UK3452     PleasureLand                        3D-Lego_Show
ZA1342     GoldTown
ZA1342     GoldTown                            GoldRush

11 rows selected.

SQL>
```

Figure 46: RIGHT OUTER JOIN example

Task 6.6 Enter the query above and check your results with those shown in Figure 48.

FULL OUTER JOIN

The full outer join returns not only the rows matching the join condition (that is, rows with matching values in the common column), but also all of the rows with unmatched values in either side table. The syntax is:

SELECT column-list

FROM *table1* FULL [OUTER] JOIN *table2* ON *join-condition*

For example, the following query lists the park code, park name, and attraction name for all attractions. In our Theme Park example the results are the same as those shown in Figure 45 for the LEFT OUTER JOIN. This is because there are no attractions without matching Theme Parks in the ATTRACTIONS table so only the Theme Parks with no attractions are shown.

```
SELECT      THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME
FROM        THEMEPARK FULL JOIN ATTRACTION ON
            THEMEPARK.PARK_CODE = ATTRACTION.PARK_CODE;
```

6.9 Exercises

E6.1 Use the cross join to display all rows in the EMPLOYEE and HOURS tables. How many rows were returned?

E6.2 Write a query to display the attraction number, employee first and last names and the date they worked on the attraction. Order the results by the date worked.

E6.3 Display the park names and total sales for Theme Parks who are located in the country ‘UK’ or ‘FR’.

E6.4 Write a query to display the names of attractions that currently have not had any employees working on them.

E6.5 List the sale date, line quantity and line price of all transactions on the 18th May 2007.

Lab 7: SQL Functions

The learning objectives of this lab are to:

- Learn about selected ORACLE date and time functions
- Be able to perform string manipulations
- Utilise single row numeric functions
- Perform conversions between data types

There are many types of SQL functions, such as arithmetic, trigonometric, string, date, and time functions. Lab 7 will cover a selection of these SQL functions in detail that are implemented by the ORACLE DBMS. Functions always use a numerical, date, or string value. The value may be part of the command itself (a constant or literal) or it may be an attribute located in a table. Therefore, a function may appear anywhere in a SQL statement where a value or an attribute can be used.

7.1 Date and Time Functions

ORACLE supports both date and time using the type DATE. The DATE type is stored in a special internal format that includes not just the month, day, and year, but also the hour, minute, and second. When a DATE value is displayed, ORACLE must first convert that value from the special internal format to a printable string. The conversion is done by the conversion function TO_CHAR. Conversion functions will be covered in more detail in section 7.4. ORACLE's default format for DATE is "DD-MON-YY".

Task 7.1 Enter the following query and examine how the date is displayed.

```
SELECT     SALE_DATE
FROM       SALES;
```

So whenever a DATE value is displayed, ORACLE will call the TO_CHAR conversion function automatically with the default DATE format. It is possible to change the format of the date.

Task 7.2 Enter the following query and notice how the date is format has changed.

```
SELECT     TO_CHAR(SALE_DATE, 'DD/MM/YYYY')
FROM       SALES;
```

You will now explore some of the main ORACLE date / time functions.

TO_CHAR

TO_CHAR returns a character string or a formatted string from a date value. The syntax is: TO_CHAR(date_value, fmt)

WHERE   fmt = format used; this can be:

- MONTH: name of month
- MON: three-letter month name
- MM: two-digit month name
- D: number for day of week
- DD: number day of month
- DAY: name of day of week
- YYYY: four-digit year value
- YY: two-digit year value

The following example lists all employees at the Theme Park who were hired after 2000.

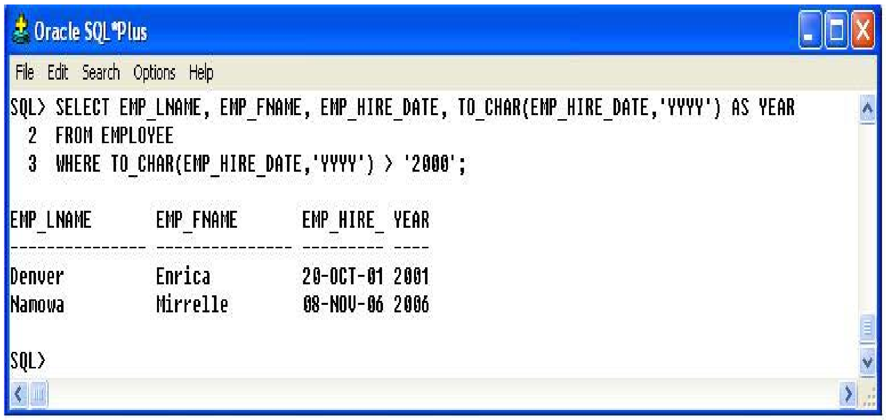
SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE,

TO_CHAR(EMP_HIRE_DATE,'YYYY') AS YEAR

FROM EMPLOYEE

WHERE TO_CHAR(EMP_HIRE_DATE,'YYYY') > '2000';

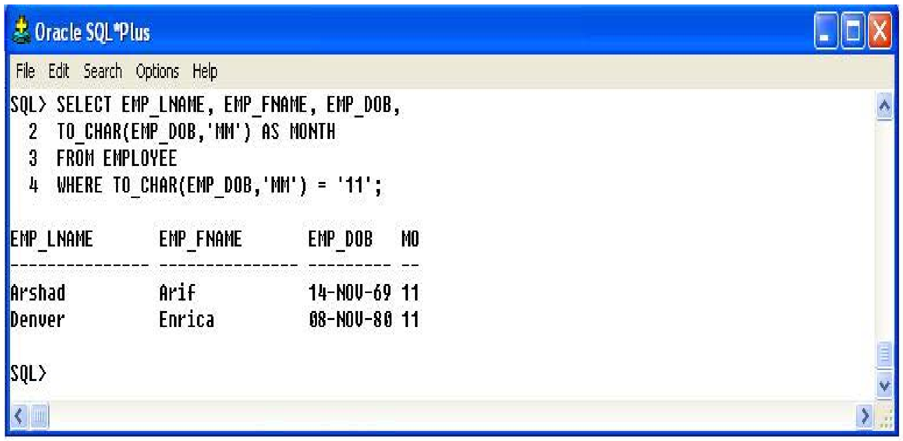
The output for this query is shown in Figure 47.



```
SQL> SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE, TO_CHAR(EMP_HIRE_DATE,'YYYY') AS YEAR
  2  FROM EMPLOYEE
  3  WHERE TO_CHAR(EMP_HIRE_DATE,'YYYY') > '2000';

EMP_LNAME       EMP_FNAME       EMP_HIRE_ YEAR
--------------- --------------- --------- ----
Denver          Enrica          20-OCT-01 2001
Namowa          Mirrelle        08-NOV-06 2006

SQL>
```

Figure 47: Employees hired after the year 2000

Task 7.3 Write a query that displays all employees who were born in November. Your output should match that shown in Figure 49.



```
SQL> SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,
  2  TO_CHAR(EMP_DOB,'MM') AS MONTH
  3  FROM EMPLOYEE
  4  WHERE TO_CHAR(EMP_DOB,'MM') = '11';

EMP_LNAME       EMP_FNAME       EMP_DOB   MO
--------------- --------------- --------- --
Arshad          Arif            14-NOV-69 11
Denver          Enrica          08-NOV-80 11

SQL>
```

Figure 49: Output for Task 7.3

TO_DATE

The TO_DATE function returns a date value using a character string and a date format mask. It can also be used to translate a date between formats. The syntax of the TO_DATE function is:

TO_DATE(char_value, fmt)

WHERE fmt = format used; this can be:

- MONTH: name of month
- MON: three-letter month name
- MM: two-digit month name
- D: number for day of week
- DD: number day of month
- DAY: name of day of week
- YYYY: four-digit year value
- YY: two-digit year value

The following example calculates the number of days between the 1st January 2008 and the 25th December 2008.

SELECT TO_DATE(′2008/12/25′,′YYYY/MM/DD′) −

TO_DATE(‘2008/01/01’,′YYYY/MM/DD′)

FROM DUAL;

In this query the TO_DATE function translates the text string to a valid ORACLE date used in date arithmetic. DUAL is ORACLE’s pseudo table used only for cases where a table is not really needed.

Task 7.4 Enter the query above and see how many days it is until the 25th December.

SYSDATE

The SYSDATE function returns today's date.

Task 7.5 Enter the following query to display today's date. Notice that the ORACLE dummy table DUAL is used.

```
SELECT SYSDATE
FROM DUAL;
```

ADD_MONTHS

The function ADD_MONTHS adds a number of months to a date. The syntax is:

```
ADD_MONTHS(date_value, n)
WHERE n = number of months.
```

Task 7.6 Enter the following query which lists the hire dates of all employees along with the date of their first work appraisal (one year from the hire date).

```
SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE,
ADD_MONTHS(EMP_HIRE_DATE,12) AS "FIRST APPRAISAL"
FROM EMPLOYEE;
```

LAST_DAY

The function LAST_DAY returns the date of the last day of the month given in a date. The syntax is:

LAST_DAY(date_value).

Task 7.7 Enter the following query which lists all sales transactions that were made in the last 12 days of a month:

```
SELECT *
FROM SALES
WHERE SALE_DATE >= LAST_DAY(SALE_DATE)-12;
```

MONTHS_BETWEEN

The MONTHS_BETWEEN function finds the number of months between two dates. The syntax is:

MONTHS_BETWEEN(date_value, date_value).

7.2 Numeric Functions

In this section, you will learn about ORACLE single row numeric functions. Numeric functions take one numeric parameter and return one value. A description of the functions you will explore in this lab can be found in Table 4.

Note

Do not confuse the SQL aggregate functions you saw in the previous chapter with the numeric functions in this section. The first group operates over a set of values (multiple rows – hence, the name *aggregate functions*), while the numeric functions covered here operate over a single row.

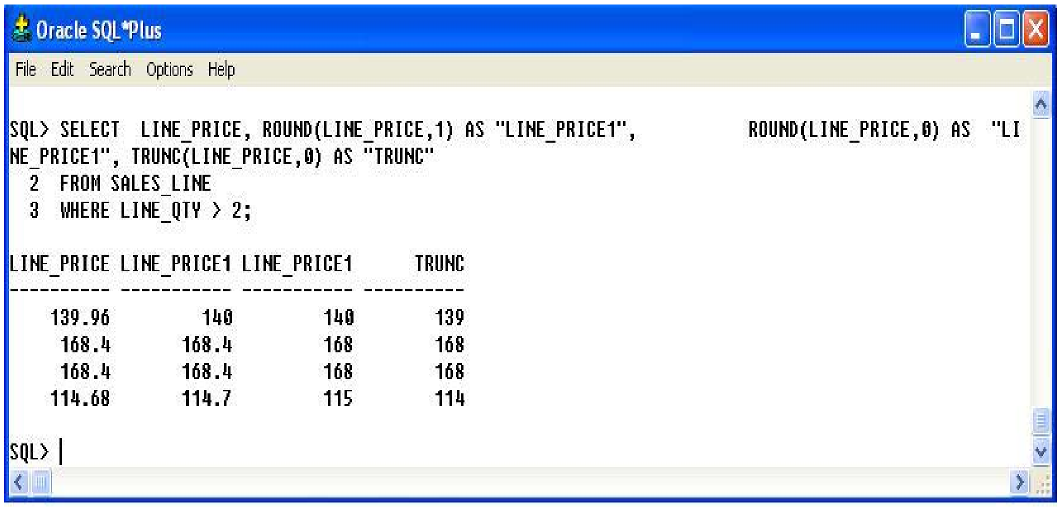
Table 4 Selected ORACLE Numeric Functions

Function	Description
ABS	Returns the absolute value of a number Syntax: ABS(numeric_value)
ROUND	Rounds a value to a specified precision (number of digits) Syntax: ROUND(numeric_value, p) where p = precision
TRUNC	Truncates a value to a specified precision (number of digits) Syntax: TRUNC(numeric_value, p) where p = precision
MOD	Returns the remainder of division. Syntax MOD(m.n) where m is divided by n.

The following example displays the individual LINE_PRICE from the sales line table rounded to one and zero places, and truncated where the quantity of tickets purchased on that line is greater than two.

```
SELECT        LINE_PRICE, ROUND(LINE_PRICE,1) AS “LINE_PRICE1”,
              ROUND(LINE_PRICE,0) AS  “LINE_PRICE1”,
TRUNC(LINE_PRICE,0) AS “TRUNC”
FROM SALES_LINE
WHERE LINE_QTY > 2;
```

The output for this query can be seen in Figure 50.



```
SQL> SELECT  LINE_PRICE, ROUND(LINE_PRICE,1) AS "LINE_PRICE1",          ROUND(LINE_PRICE,0) AS  "LI
NE_PRICE1", TRUNC(LINE_PRICE,0) AS "TRUNC"
  2  FROM SALES_LINE
  3  WHERE LINE_QTY > 2;

LINE_PRICE LINE_PRICE1 LINE_PRICE1      TRUNC
---------- ----------- ----------- ----------
    139.96         140         140        139
     168.4       168.4         168        168
     168.4       168.4         168        168
    114.68       114.7         115        114

SQL> |
```

Figure 50 Example of ROUND and TRUNC

Task 7.8 Enter the following query and execute it. Can you explain the results of this query?

```
SELECT  TRANSACTION_NO, LINE_PRICE, MOD(LINE_PRICE, 10)
FROM SALES_LINE
WHERE LINE_QTY > 2;
```

7.3 String Functions

String manipulation functions are amongst the most commonly used functions in programming. Table 5 shows a subset of the most useful string manipulation functions in ORACLE.

Table 5 Selected ORACLE string functions.

Function	**Description**
CONCAT or ||	Concatenates data from two different character columns and returns a single column. Syntax: strg_value || strg_value
UPPER/LOWER	Returns a string in all capital or all lowercase letters Syntax: UPPER(strg_value) , LOWER(strg_value)
SUBSTR	Returns a substring or part of a given string parameter Syntax: SUBSTR(strg_value, p, l) where p = start position and l = length of characters
LENGTH	Returns the number of characters in a string value Syntax: LENGTH(strg_value)

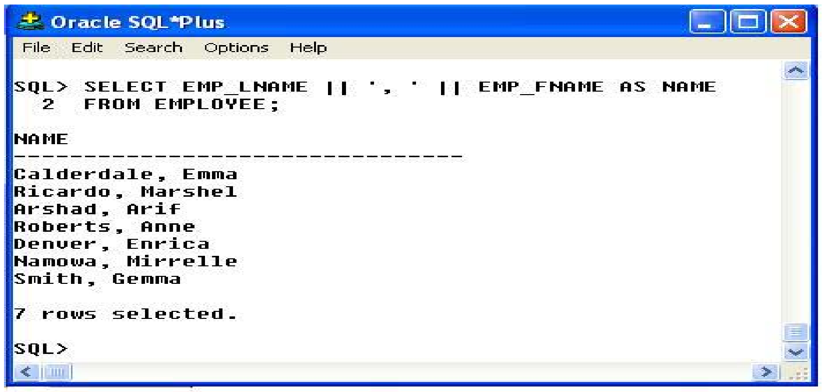
We will now look at examples of some of these string functions.

CONCAT

The following query illustrates the CONCAT function. It lists all employees' first and last names concatenated together. The output for this query can be seen in Figure 51.

SELECT EMP_LNAME || ', ' || EMP_FNAME AS NAME

FROM EMPLOYEE;



```
SQL> SELECT EMP_LNAME || ', ' || EMP_FNAME AS NAME
  2  FROM EMPLOYEE;

NAME
--------------------------------
Calderdale, Emma
Ricardo, Marshel
Arshad, Arif
Roberts, Anne
Denver, Enrica
Namowa, Mirrelle
Smith, Gemma

7 rows selected.

SQL>
```

Figure 51: Concatenation of employee's first and last names

Task 7.9 The query above can also be written in using the function name CONCAT. Enter the following query and compare the output to that shown in Figure 51.

SELECT CONCAT(EMP_LNAME ,EMP_FNAME) AS NAME

FROM EMPLOYEE;

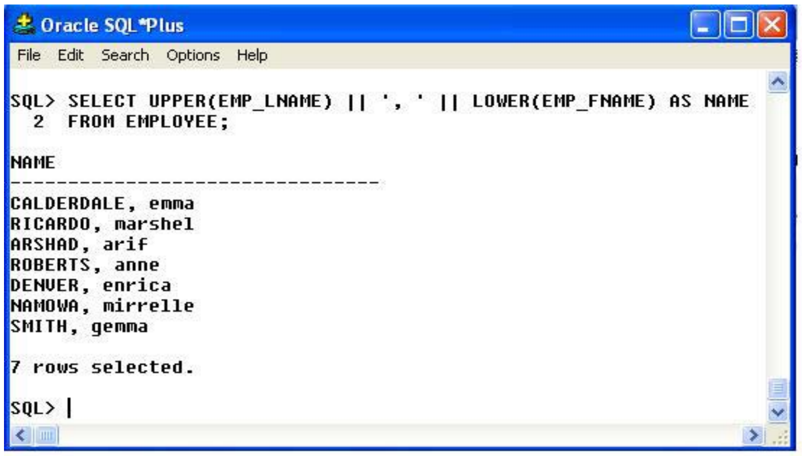
When using the CONCAT function as shown in Task 7.9 you are limited to the concatenation of two strings only.

UPPER/LOWER

The following query lists all employee last names in all capital letters, and all first names in all lowercase letters. The output for the query is shown in Figure 52.

SELECT UPPER(EMP_LNAME) || ′, ′ || LOWER(EMP_FNAME) AS NAME

FROM EMPLOYEE;



```
SQL> SELECT UPPER(EMP_LNAME) || ', ' || LOWER(EMP_FNAME) AS NAME
  2  FROM EMPLOYEE;

NAME
--------------------------------
CALDERDALE, emma
RICARDO, marshel
ARSHAD, arif
ROBERTS, anne
DENVER, enrica
NAMOWA, mirrelle
SMITH, gemma

7 rows selected.

SQL> |
```

Figure 52: Displaying upper and lower case employee names

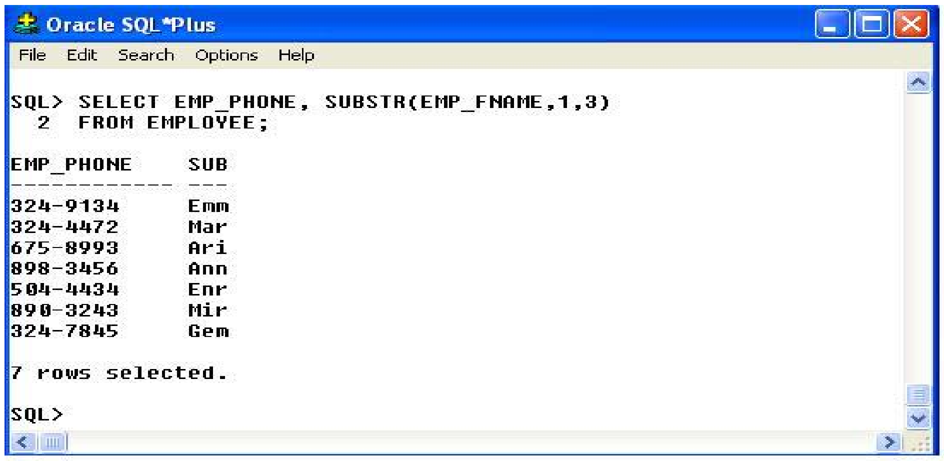
SUBSTR

The following example lists the first three characters of all the employees’ first names.

The output of this query is shown in Figure 53.

SELECT EMP_PHONE, SUBSTR(EMP_FNAME,1,3)

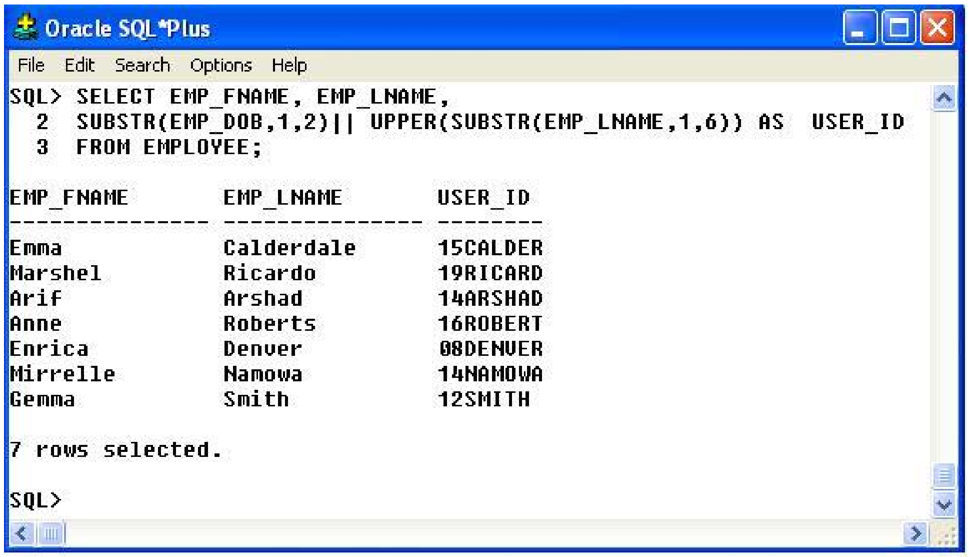
FROM EMPLOYEE;



```
SQL> SELECT EMP_PHONE, SUBSTR(EMP_FNAME,1,3)
  2  FROM EMPLOYEE;

EMP_PHONE    SUB
------------ ---
324-9134     Emm
324-4472     Mar
675-8993     Ari
898-3456     Ann
504-4434     Enr
890-3243     Mir
324-7845     Gem

7 rows selected.

SQL>
```

Figure 53: Displaying the first 3 characters of the employees’ first names

Task 7.10 Write a query which generates a list of employee user IDs, using first the day of the month they were born and then the first six characters of their last name in UPPER case. Your query should return the results shown in Figure 54.



```
SQL> SELECT EMP_FNAME, EMP_LNAME,
  2  SUBSTR(EMP_DOB,1,2)|| UPPER(SUBSTR(EMP_LNAME,1,6)) AS  USER_ID
  3  FROM EMPLOYEE;

EMP_FNAME       EMP_LNAME       USER_ID
--------------- --------------- --------
Emma            Calderdale      15CALDER
Marshel         Ricardo         19RICARD
Arif            Arshad          14ARSHAD
Anne            Roberts         16ROBERT
Enrica          Denver          08DENVER
Mirrelle        Namowa          14NAMOWA
Gemma           Smith           12SMITH

7 rows selected.

SQL>
```

Figure 54: Results for Task 7.10

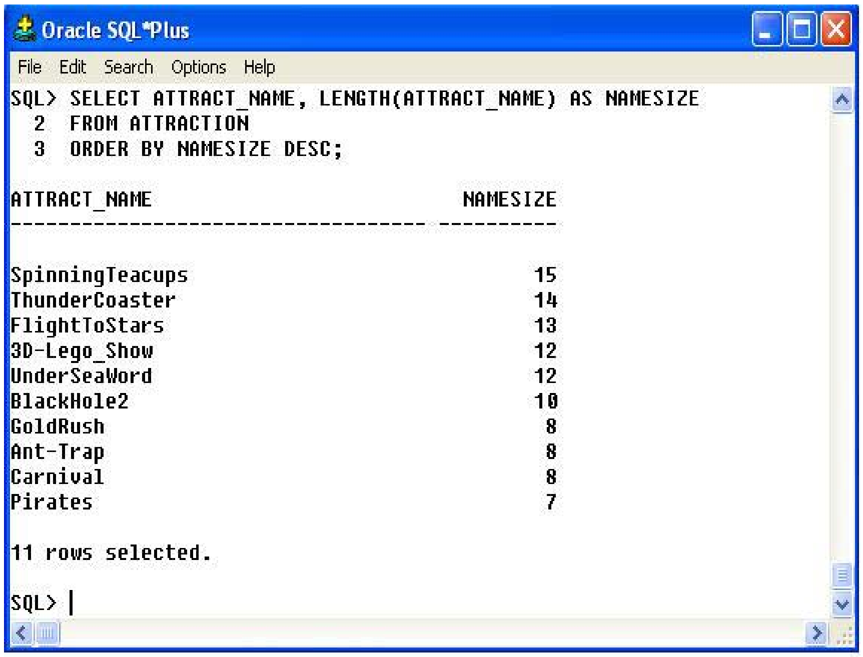
LENGTH

The following example lists all attraction names and the length of their names; ordered descended by attraction name length. The output of this query is shown in Figure 55.

SELECT ATTRACT_NAME, LENGTH(ATTRACT_NAME) AS NAMESIZE

FROM ATTRACTION

ORDER BY NAMESIZE DESC;



```
SQL> SELECT ATTRACT_NAME, LENGTH(ATTRACT_NAME) AS NAMESIZE
  2  FROM ATTRACTION
  3  ORDER BY NAMESIZE DESC;

ATTRACT_NAME                          NAMESIZE
---------------------------------- ----------

SpinningTeacups                             15
ThunderCoaster                              14
FlightToStars                               13
3D-Lego_Show                                12
UnderSeaWord                                12
BlackHole2                                  10
GoldRush                                     8
Ant-Trap                                     8
Carnival                                     8
Pirates                                      7

11 rows selected.

SQL> |
```

Figure 55: Displaying the length of attraction names

7.4 Conversion Functions

Conversion functions allow you to take a value of a given data type and convert it to the equivalent value in another data type. In Section 7.2, you learned about two of the basic conversion functions: TO_CHAR and TO_DATE. Note that the TO_CHAR function takes a date value and returns a character string representing a day, a month, or a year. In the same way, the TO_DATE function takes a character string representing a date and returns an actual date in ORACLE format. In this section you will see how the TO_CHAR function is used to convert numbers to a formatted character string and how the TO_NUMBER function is used to convert text strings to numeric values.

TO_CHAR (numeric)

The TO_CHAR numeric function returns a character string or a formatted string from a numeric value and is very useful for formatting numeric columns in reports. The syntax is:

```
TO_CHAR(numeric_value, fmt)
```

WHERE fmt = format used. This can be:

- 9 = displays a digit
- 0 = displays a leading zero
- , = displays the comma
- . = displays the decimal point
- € = displays the euro sign

Task 7.11 Test the following query which lists all the sales line transactions for each Theme Park using formatted values.

```
SELECT      PARK_NAME, TICKET_TYPE, TO_CHAR(TICKET_PRICE, '99.99')
            AS PRICE,
            TO_CHAR(LINE_QTY, '999.99') AS QUANTITY,
            TO_CHAR(LINE_PRICE, '999.99') AS PRICE
FROM        TICKET NATURAL JOIN SALES_LINE NATURAL JOIN
        THEMEPARK;
```

TO_CHAR (date)

The TO_CHAR date function returns a character string or a formatted character string from a date value. The syntax is:

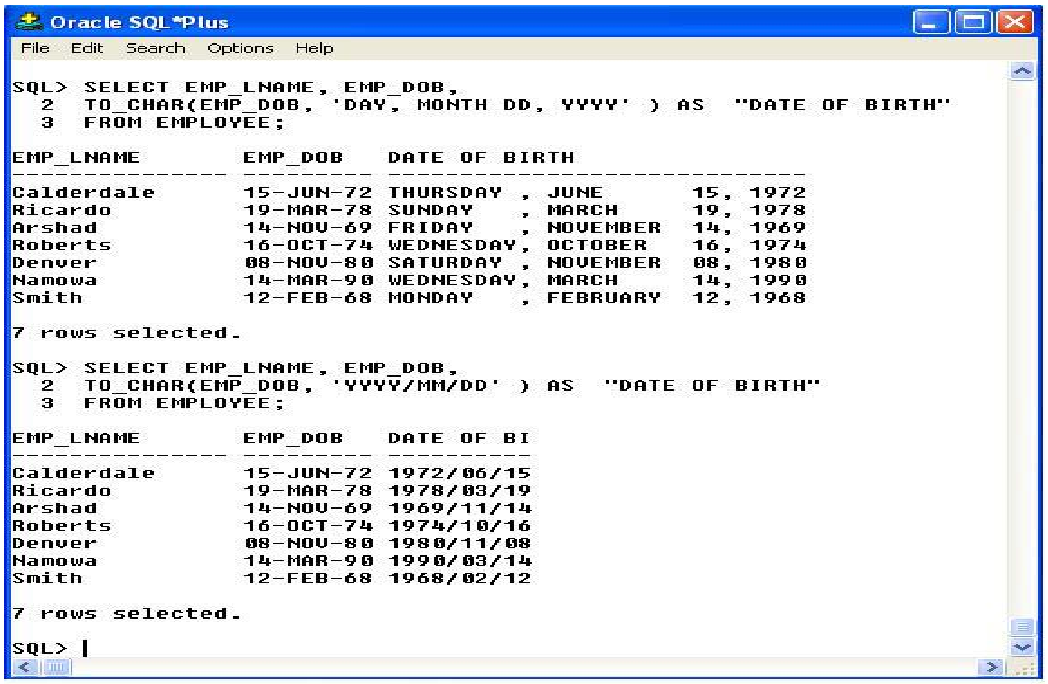
TO_CHAR(date_value, fmt)

WHERE fmt = format used; This can be:

- MONTH: name of month
- MON: three-letter month name
- MM: two-digit month name
- D: number for day of week
- DD: number day of month
- DAY: name of day of week
- YYYY: four-digit year value
- YY: two-digit year value

The following examples list all employee dates of birth, using different date formats. The results of both queries can be seen in Figure 56.

```
SELECT EMP_LNAME, EMP_DOB,
TO_CHAR(EMP_DOB, ′DAY, MONTH DD, YYYY′ ) AS “DATE OF BIRTH”
FROM EMPLOYEE;
SELECT EMP_LNAME, EMP_DOB,
TO_CHAR(EMP_DOB, ′YYYY/MM/DD′ ) AS “DATE OF BIRTH”
FROM EMPLOYEE;
```



```
SQL> SELECT EMP_LNAME, EMP_DOB,
  2  TO_CHAR(EMP_DOB, 'DAY, MONTH DD, YYYY' ) AS  "DATE OF BIRTH"
  3  FROM EMPLOYEE;

EMP_LNAME        EMP_DOB   DATE OF BIRTH
---------------- --------- ------------------------------
Calderdale       15-JUN-72 THURSDAY , JUNE      15, 1972
Ricardo          19-MAR-78 SUNDAY   , MARCH     19, 1978
Arshad           14-NOV-69 FRIDAY   , NOVEMBER  14, 1969
Roberts          16-OCT-74 WEDNESDAY, OCTOBER   16, 1974
Denver           08-NOV-80 SATURDAY , NOVEMBER  08, 1980
Namowa           14-MAR-90 WEDNESDAY, MARCH     14, 1990
Smith            12-FEB-68 MONDAY   , FEBRUARY  12, 1968

7 rows selected.

SQL> SELECT EMP_LNAME, EMP_DOB,
  2  TO_CHAR(EMP_DOB, 'YYYY/MM/DD' ) AS  "DATE OF BIRTH"
  3  FROM EMPLOYEE;

EMP_LNAME        EMP_DOB   DATE OF BI
---------------- --------- ----------
Calderdale       15-JUN-72 1972/06/15
Ricardo          19-MAR-78 1978/03/19
Arshad           14-NOV-69 1969/11/14
Roberts          16-OCT-74 1974/10/16
Denver           08-NOV-80 1980/11/08
Namowa           14-MAR-90 1990/03/14
Smith            12-FEB-68 1968/02/12

7 rows selected.

SQL> |
```

Figure 56: Examples of the TO_CHAR (date) function

TO_NUMBER

The TO_NUMBER function returns a formatted number from a character string, using a given format. It is used to convert text strings to numeric values when importing data to a table from another source in text format. The syntax is:

TO_NUMBER(char_value, fmt)

WHERE fmt = format used; This can be:

- 9 = displays a digit
- 0 = displays a leading zero
- , = displays the comma
- . = displays the decimal point
- € = displays the euro sign

- B = leading blank
- S = leading sign
- MI = trailing minus sign

For example, the query shown below uses the TO_NUMBER function to convert text formatted to ORACLE default numeric values using the format masks given:

```
SELECT TO_NUMBER('-123.99', 'S999.99'),
        TO_NUMBER('99.78-','B999.99MI')
FROM DUAL;
```

Note

If you can not display the Euro symbol ("€") in SQL*Plus, contact your instructor or database administrator.

NVL

The NVL function lets you substitute a value when a null value is encountered in the results of a query. The syntax is:

```
NVL(x, y)
```

WHERE x = attribute or expression and y = value to return if x is null.

If *x* is null, then NVL returns *y*. If *x* is not null, then NVL returns *x*. The data type of the return value is always the same as the data type of *x*. It is useful for avoiding errors caused by incorrect calculation when one of the arguments is null.

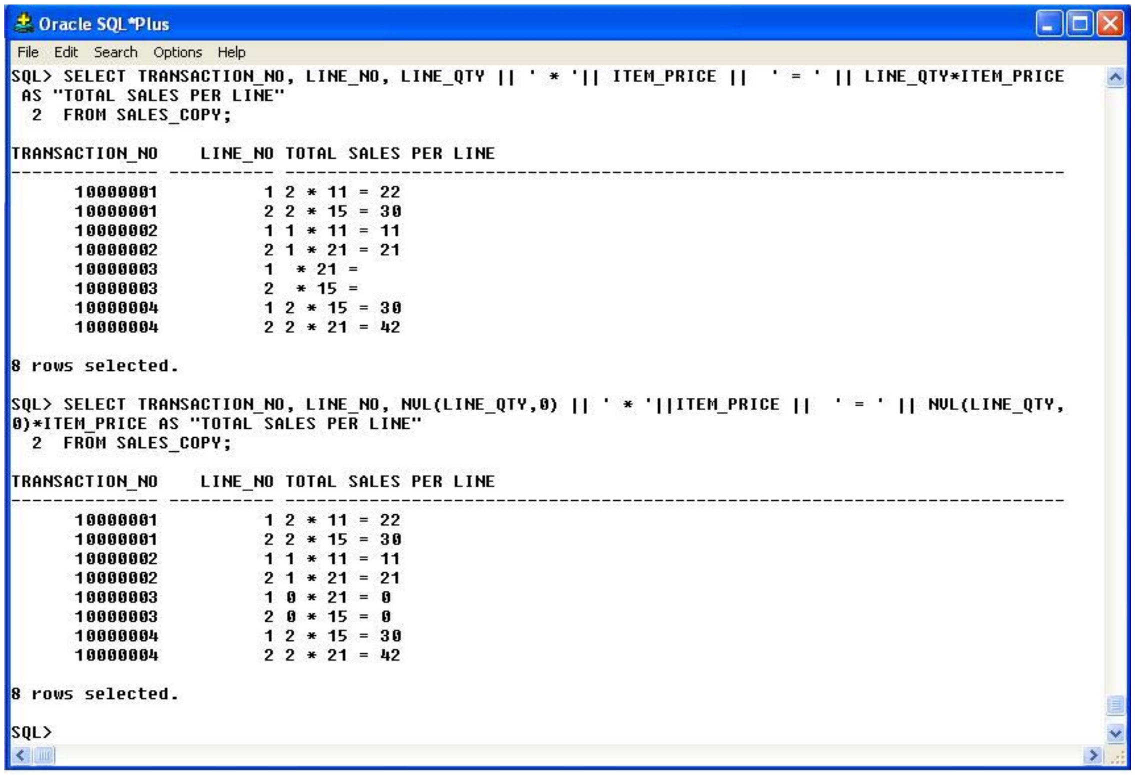
Task 7.12 Load and run the script sales_copy.sql which accompanies this lab guide. DESCRIBE the structure of the SALES_COPY table and examine the lack of constraints on this table. Write a query to view all the rows and notice that in some rows no values have been entered for LINE_QTY or LINE_PRICE. (In these instances these rows have NULL values.) Next, enter the following query which displays to the screen the Total of the LINE_QTY * LINE_PRICE. Notice that this query does not use the NVL function and in two rows the calculation cannot be made.

```
SELECT TRANSACTION_NO, LINE_NO, LINE_QTY || ' * '|| ITEM_PRICE ||  ' = ' ||
LINE_QTY*ITEM_PRICE AS "TOTAL SALES PER LINE"
FROM SALES_COPY;
```

Next, run the following version of the query which uses the NVL function and notice that the calculation has been achieved for all rows.

```
SELECT TRANSACTION_NO, LINE_NO, NVL(LINE_QTY,0) || ' * '||ITEM_PRICE ||
' = ' || NVL(LINE_QTY,0)*ITEM_PRICE AS "TOTAL SALES PER LINE"
FROM SALES_COPY;
```

The results of running both these queries can be seen in Figure 57.



```
SQL> SELECT TRANSACTION_NO, LINE_NO, LINE_QTY || ' * '|| ITEM_PRICE ||  ' = ' || LINE_QTY*ITEM_PRICE
 AS "TOTAL SALES PER LINE"
  2  FROM SALES_COPY;

TRANSACTION_NO    LINE_NO TOTAL SALES PER LINE
-------------- ---------- --------------------------------------------------------------------------------
      10000001          1 2 * 11 = 22
      10000001          2 2 * 15 = 30
      10000002          1 1 * 11 = 11
      10000002          2 1 * 21 = 21
      10000003          1  * 21 =
      10000003          2  * 15 =
      10000004          1 2 * 15 = 30
      10000004          2 2 * 21 = 42

8 rows selected.

SQL> SELECT TRANSACTION_NO, LINE_NO, NVL(LINE_QTY,0) || ' * '||ITEM_PRICE ||  ' = ' || NVL(LINE_QTY,
0)*ITEM_PRICE AS "TOTAL SALES PER LINE"
  2  FROM SALES_COPY;

TRANSACTION_NO    LINE_NO TOTAL SALES PER LINE
-------------- ---------- --------------------------------------------------------------------------------
      10000001          1 2 * 11 = 22
      10000001          2 2 * 15 = 30
      10000002          1 1 * 11 = 11
      10000002          2 1 * 21 = 21
      10000003          1 0 * 21 = 0
      10000003          2 0 * 15 = 0
      10000004          1 2 * 15 = 30
      10000004          2 2 * 21 = 42

8 rows selected.

SQL>
```

Figure 57: Illustration of the NVL function

DECODE

The DECODE function compares an attribute or expression with a series of values and returns an associated value or a default value if no match is found. The syntax is:

DECODE(*e, x*, *y*, *d*)

WHERE *e* = attribute or expression, *x* = value with which to compare *e*

y = value to return in *e* = *x*

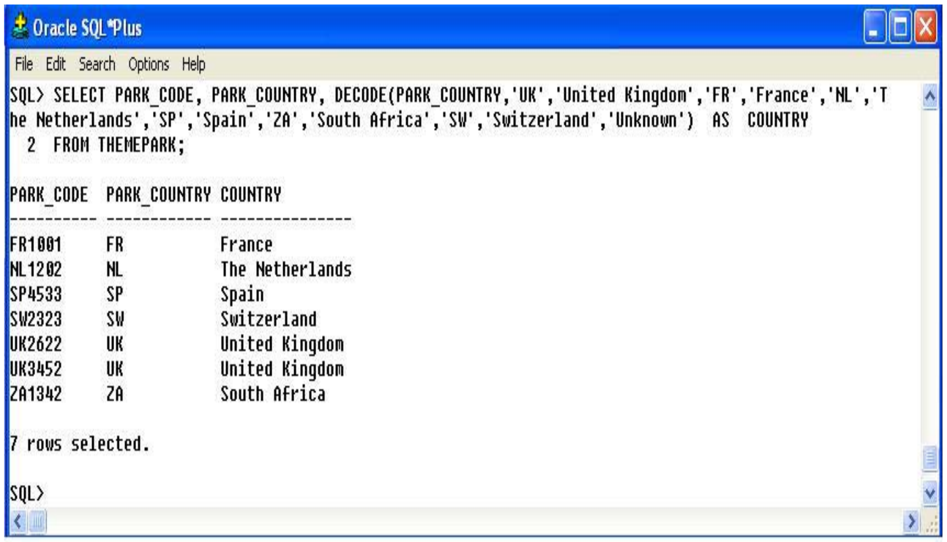
d = default value to return if *e* is not equal to *x*

Let’s now look at the following example, which compares the country code in the PARK_COUNTRY field and decodes it into the name of the country. If there is no match it returns the value ‘Unknown’. The output is shown in Figure 58.

SELECT PARK_CODE, PARK_COUNTRY,

DECODE(PARK_COUNTRY,'UK','United Kingdom','FR','France','NL','The Netherlands','SP','Spain','ZA','South Africa','SW','Switzerland','Unknown') AS COUNTRY

FROM THEMEPARK;



```
SQL> SELECT PARK_CODE, PARK_COUNTRY, DECODE(PARK_COUNTRY,'UK','United Kingdom','FR','France','NL','T
he Netherlands','SP','Spain','ZA','South Africa','SW','Switzerland','Unknown')  AS  COUNTRY
  2  FROM THEMEPARK;

PARK_CODE  PARK_COUNTRY COUNTRY
---------- ------------ ---------------
FR1001     FR           France
NL1202     NL           The Netherlands
SP4533     SP           Spain
SW2323     SW           Switzerland
UK2622     UK           United Kingdom
UK3452     UK           United Kingdom
ZA1342     ZA           South Africa

7 rows selected.

SQL>
```

Figure 58: Displaying the names of countries using the DECODE function

It is worth noting that the above decode statement is equivalent to the following IF-THEN-ELSE statement:

```
IF PARK_COUNTRY = ‘UK’ THEN
     result := 'United Kingdom';
ELSIF PARK_COUNTRY = ‘FR’ THEN
    result := 'FRANCE';
ELSIF PARK_COUNTRY = ‘NL’ THEN
    result := 'The Netherlands';
ELSIF PARK_COUNTRY = ‘SP’ THEN
    result := 'Spain';
ELSIF PARK_COUNTRY = ‘ZA’ THEN
    result := 'South Africa';
ELSIF PARK_COUNTRY = ‘SW’ THEN
    result := 'Switzerland';
ELSE
    result := 'Unknown;
```

END IF;

7.5 Exercises

E7.1 Write a query which lists the names and dates of births of all employees born on the 14th day of the month.

E7.2 Write a query which lists the approximate age of the employees on the company's tenth anniversary date (11/25/2008).

E7.3 Write a query which generates a list of employee user passwords, using the first three digits of their phone number, the first two characters of first name in lower case and the employee number. Label the column USER_PASSWORD;

E7.4 Write a query which displays the last date a ticket was purchased in all Theme Parks. You should also display the Theme Park name. Print the date in the format 12th January 2007.

E7.5 Write a query that displays the length of employment of each employee. For each employee, display the first and last names and calculate the number of months between today and the date that they were hired.

Lab 8: SET Operators

The learning objectives of this lab are to:

- Be able to apply the following set operators in SQL statements
    - UNION: All distinct rows selected by either query
    - UNION ALL: All rows selected by either query, including all duplicates
    - INTERSECT: All distinct rows selected by both queries
    - MINUS: All distinct rows selected by the first query, but not the second

SQL data manipulation commands are set oriented; that is, they operate over entire sets of rows and columns (tables) at once. Using sets, you can combine two or more sets to create new sets (or relations). UNION, INTERSECT and MINUS work properly only if relations are *union-compatible*. In SQL terms, *union-compatible* means that the names of the relation attributes must be the same and their data types must be identical.

8.1 Union

The UNION statement combines rows from two or more queries *without including duplicate rows*. The syntax of the UNION statement is:

query UNION *query*

In other words, the UNION statement combines the output of two SELECT queries.

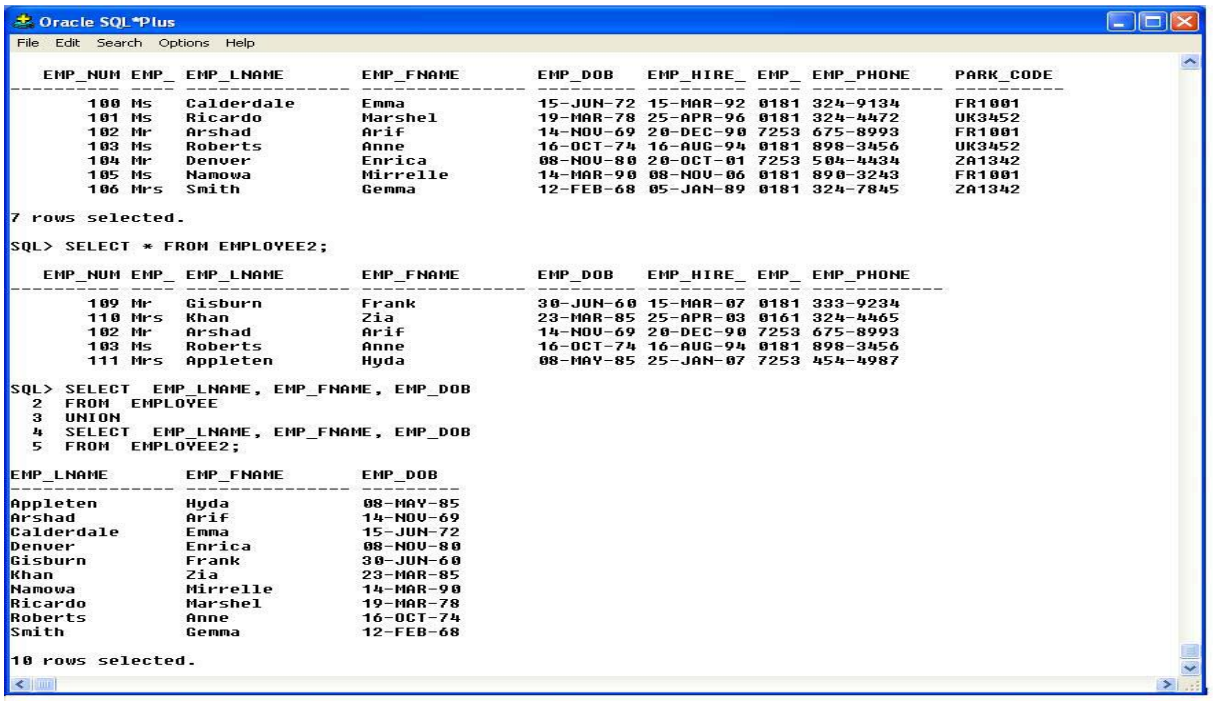
To demonstrate the use of the UNION statement in SQL, you will first need to run an SQL script to create a new employee table called EMPLOYEE2 (The script called employee2.sql accompanies this lab guide). The EMPLOYEE2 table also contains details of employees who work at the Theme Parks and contains some records the same as those in the EMPLOYEE table.

Task 8.1 Run the script called employee2.sql and write a query to display all its contents.

To show the effect of combing our existing EMPLOYEE table with the new EMPLOYEE2 table without the duplicates, the UNION query is written as follows:

```
SELECT      EMP_LNAME, EMP_FNAME, EMP_DOB
FROM        EMPLOYEE
UNION
SELECT      EMP_LNAME, EMP_FNAME, EMP_DOB
FROM        EMPLOYEE2;
```

Figure 59 shows the contents of the EMPLOYEE and EMPLOYEE2 tables and the result of the UNION query.



```
   EMP_NUM EMP_ EMP_LNAME       EMP_FNAME       EMP_DOB   EMP_HIRE_ EMP_ EMP_PHONE    PARK_CODE
---------- ---- --------------- --------------- --------- --------- ---- ------------ ----------
       100 Ms   Calderdale      Emma            15-JUN-72 15-MAR-92 0181 324-9134     FR1001
       101 Ms   Ricardo         Marshel         19-MAR-78 25-APR-96 0181 324-4472     UK3452
       102 Mr   Arshad          Arif            14-NOV-69 20-DEC-90 7253 675-8993     FR1001
       103 Ms   Roberts         Anne            16-OCT-74 16-AUG-94 0181 898-3456     UK3452
       104 Mr   Denver          Enrica          08-NOV-80 20-OCT-01 7253 504-4434     ZA1342
       105 Ms   Namowa          Mirrelle        14-MAR-90 08-NOV-06 0181 890-3243     FR1001
       106 Mrs  Smith           Gemma           12-FEB-68 05-JAN-89 0181 324-7845     ZA1342

7 rows selected.

SQL> SELECT * FROM EMPLOYEE2;

   EMP_NUM EMP_ EMP_LNAME       EMP_FNAME       EMP_DOB   EMP_HIRE_ EMP_ EMP_PHONE
---------- ---- --------------- --------------- --------- --------- ---- ------------
       109 Mr   Gisburn         Frank           30-JUN-60 15-MAR-07 0181 333-9234
       110 Mrs  Khan            Zia             23-MAR-85 25-APR-03 0161 324-4465
       102 Mr   Arshad          Arif            14-NOV-69 20-DEC-90 7253 675-8993
       103 Ms   Roberts         Anne            16-OCT-74 16-AUG-94 0181 898-3456
       111 Mrs  Appleten        Hyda            08-MAY-85 25-JAN-07 7253 454-4987

SQL> SELECT  EMP_LNAME, EMP_FNAME, EMP_DOB
  2  FROM  EMPLOYEE
  3  UNION
  4  SELECT  EMP_LNAME, EMP_FNAME, EMP_DOB
  5  FROM  EMPLOYEE2;

EMP_LNAME       EMP_FNAME       EMP_DOB
--------------- --------------- ---------
Appleten        Hyda            08-MAY-85
Arshad          Arif            14-NOV-69
Calderdale      Emma            15-JUN-72
Denver          Enrica          08-NOV-80
Gisburn         Frank           30-JUN-60
Khan            Zia             23-MAR-85
Namowa          Mirrelle        14-MAR-90
Ricardo         Marshel         19-MAR-78
Roberts         Anne            16-OCT-74
Smith           Gemma           12-FEB-68

10 rows selected.
```

Figure 59: Example of UNION

Task 8.2 Run the UNION query above and check your answers with those shown in Figure 59.

8.2 UNION ALL

If the Theme Park management wants to know how many customers are on *both* the EMPLOYEE and EMPLOYEE2 lists, a UNION ALL query can be used to produce a relation that retains the duplicate rows. Therefore, the following query will keep all rows from both queries (including the duplicate rows) and return 17 rows.

```
SELECT      EMP_LNAME, EMP_FNAME, EMP_DOB

FROM        EMPLOYEE

UNION ALL
```

```
SELECT    EMP_LNAME, EMP_FNAME, EMP_DOB
FROM      EMPLOYEE2;
```

8.3 INTERSECT

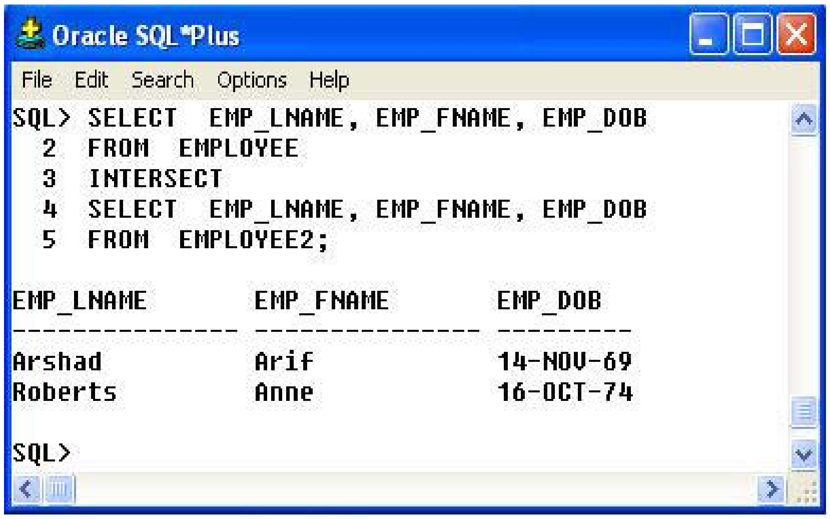
The INTERSECT statement can be used to combine rows from two queries, returning only the rows that appear in both sets. The syntax for the INTERSECT statement is:

query INTERSECT *query*

To generate the list of duplicate employee records, you can use:

```
SELECT    EMP_LNAME, EMP_FNAME, EMP_DOB
FROM      EMPLOYEE
INTERSECT
SELECT    EMP_LNAME, EMP_FNAME, EMP_DOB
FROM      EMPLOYEE2;
```

Task 8.3 Run the INTERSECT query above and check your answers with those shown in Figure 60.



```
SQL> SELECT  EMP_LNAME, EMP_FNAME, EMP_DOB
  2  FROM  EMPLOYEE
  3  INTERSECT
  4  SELECT  EMP_LNAME, EMP_FNAME, EMP_DOB
  5  FROM  EMPLOYEE2;

EMP_LNAME       EMP_FNAME       EMP_DOB
--------------- --------------- ---------
Arshad          Arif            14-NOV-69
Roberts         Anne            16-OCT-74

SQL>
```

Figure 60: Example of INTERSECT

Task 8.4 Write a query that returns the employee numbers for all employees who have worked on any attractions in any Theme Park who are located in area code 0181. (If an employee has worked on an attraction, there must be an hour's record for that employee in the HOURS table.) Note for this question you should use the original EMPLOYEEs' table. Your query should return 3 rows.

8.4 MINUS

The MINUS statement in SQL combines rows from two queries and returns only the rows that appear in the first set but not in the second. The syntax for the MINUS statement is:

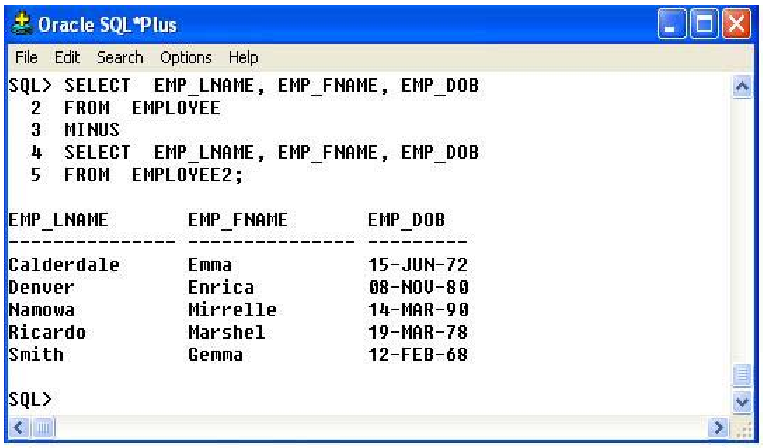
query MINUS *query*

For example, if the Theme Park managers want to know what employees are in the EMPLOYEE table that cannot be found in the EMPLOYEE2 table, they can use the following query:

To generate the list of duplicate employee records, you can use:

```
SELECT    EMP_LNAME, EMP_FNAME, EMP_DOB
FROM      EMPLOYEE
MINUS
SELECT    EMP_LNAME, EMP_FNAME, EMP_DOB
FROM      EMPLOYEE2;
```

Task 8.5 Run the MINUS query above and check your answers with those shown in Figure 61.



```
SQL> SELECT  EMP_LNAME, EMP_FNAME, EMP_DOB
  2  FROM  EMPLOYEE
  3  MINUS
  4  SELECT  EMP_LNAME, EMP_FNAME, EMP_DOB
  5  FROM  EMPLOYEE2;

EMP_LNAME       EMP_FNAME       EMP_DOB
--------------- --------------- ---------
Calderdale      Emma            15-JUN-72
Denver          Enrica          08-NOV-80
Namowa          Mirrelle        14-MAR-90
Ricardo         Marshel         19-MAR-78
Smith           Gemma           12-FEB-68

SQL>
```

Figure 61: Example of MINUS

8.5 Exercises

E8.1 Enter and run the following query. Explain your results:

SELECT EMP_NUM FROM EMPLOYEE2

```
UNION

SELECT EMP_NUM FROM HOURS;
```

E8.2 Enter and run the following query. Explain your results.

```
SELECT EMP_NUM FROM EMPLOYEE2

INTERSECT

SELECT EMP_NUM FROM HOURS;
```

E8.3 Write a query which identifies all employees who worked on attractions on 19^{h} May 2007, except for employee number 100.

Lab 9: Subqueries

The learning objectives of this lab are to:

- Learn how to use subqueries to extract rows from processed data
- Select the most suitable subquery format
- Use correlated subqueries

First, let's outline the basic characteristics of a subquery that were introduced in Chapter 8, Introduction to Structured Query Language.

- A subquery is a query (SELECT statement) inside a query
- A subquery is normally expressed inside parentheses
- The first query in the SQL statement is known as the outer query
- The query inside the SQL statement is known as the inner query
- The inner query is executed first
- The output of an inner query is used as the input for the outer query
- The entire SQL statement is sometimes referred to as a nested query

A subquery can return one value or multiple values. To be precise, the subquery can return:

- *One single value (one column and one row).* This subquery is used anywhere a single value is expected, as in the right side of a comparison expression. Obviously, when you assign a value to an attribute, that value is a single value, not a list of values. Therefore, the subquery must return only one value (one column, one row). If the query returns multiple values, the DBMS will generate an error.

- *A list of values (one column and multiple rows).* This type of subquery is used anywhere a list of values is expected, such as when using the IN clause. This type of subquery is used frequently in combination with the IN operator in a WHERE conditional expression.
- *A virtual table (multicolumn, multirow set of values).* This type of subquery can be used anywhere a table is expected, such as when using the FROM clause.

It is important to note that a subquery can return no values at all; it is a NULL. In such cases, the output of the outer query may result in an error or a null empty set, depending where the subquery is used (in a comparison, an expression, or a table set).

In the following sections, you will learn how to write subqueries within the SELECT statement to retrieve data from the database.

Note

You can also read more about subqueries in Chapter 9, Advanced SQL.

9.1 SELECT Subqueries

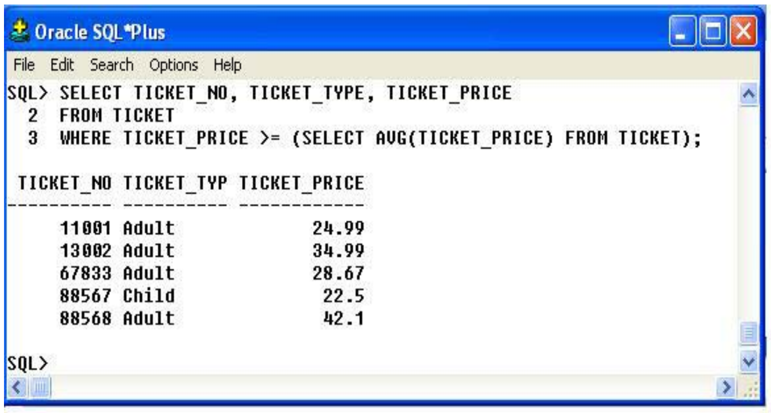
The most common type of subquery uses an inner SELECT subquery on the right side of a WHERE comparison expression. For example, to find the prices of all tickets with a price less than or equal to the average ticket price, you write the following query:

SELECT TICKET_NO, TICKET_TYPE, TICKET_PRICE

FROM TICKET

WHERE TICKET_PRICE >= (SELECT AVG(TICKET_PRICE) FROM TICKET);

The output of the query is shown in Figure 62.

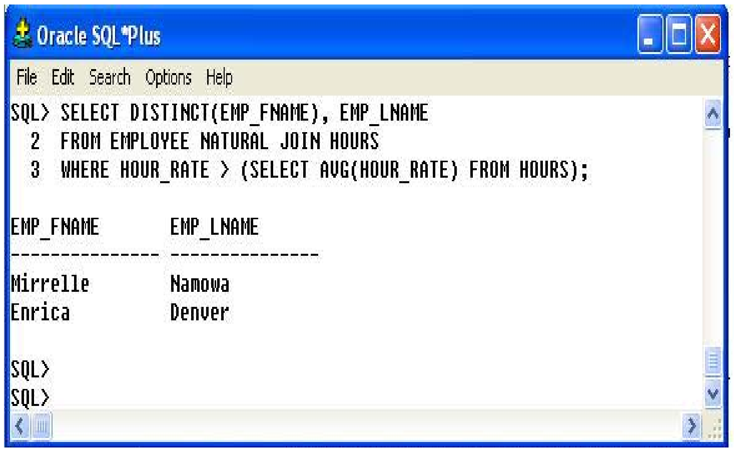


```
SQL> SELECT TICKET_NO, TICKET_TYPE, TICKET_PRICE
  2  FROM TICKET
  3  WHERE TICKET_PRICE >= (SELECT AVG(TICKET_PRICE) FROM TICKET);

 TICKET_NO TICKET_TYP TICKET_PRICE
---------- ---------- ------------
     11001 Adult             24.99
     13002 Adult             34.99
     67833 Adult             28.67
     88567 Child              22.5
     88568 Adult              42.1

SQL>
```

Figure 62: Example of SELECT Subquery

Note that this type of query, when used in a >, <, =, >=, or <= conditional expression, requires a subquery that returns only one single value (one column, one row). The value generated by the subquery must be of a “comparable” data type; if the attribute to the left of the comparison symbol is a character type, the subquery must return a character string. Also, if the query returns more than a single value, the DBMS will generate an error.

Task 9.1 Write a query that displays the first then last name of all employees who earn more than the average hourly rate. Do not display duplicate rows. Your output should match that shown in Figure 63.



```
SQL> SELECT DISTINCT(EMP_FNAME), EMP_LNAME
  2  FROM EMPLOYEE NATURAL JOIN HOURS
  3  WHERE HOUR_RATE > (SELECT AVG(HOUR_RATE) FROM HOURS);

EMP_FNAME       EMP_LNAME
--------------- ---------------
Mirrelle        Namowa
Enrica          Denver

SQL>
SQL>
```

Figure 63: Output for task 9.1

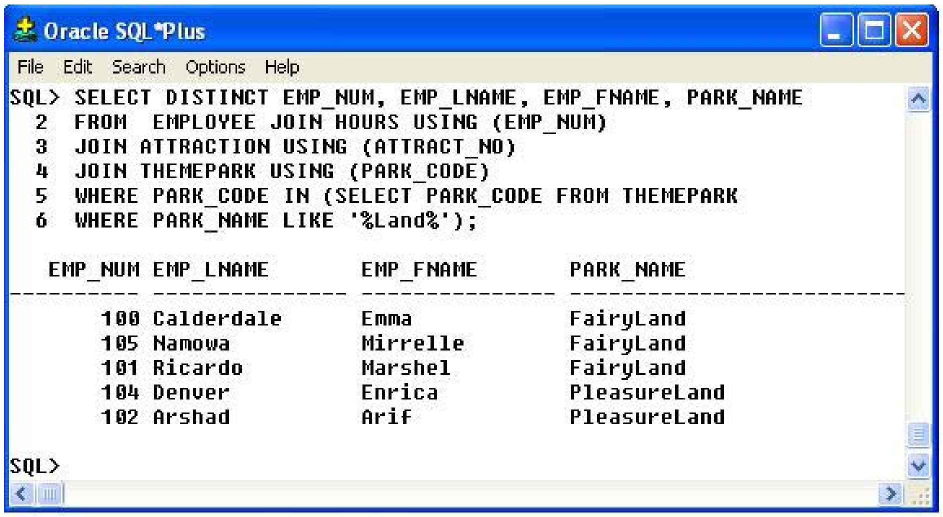
9.2 IN Subqueries

The following query displays all employees who work in a Theme Park that has the word 'Land' in its name. As there are a number of different Theme Parks that match this criterion you need to compare the PARK_CODE not to one park code (single value), but to a list of park codes. When you want to compare a single attribute to a list of values, you use the IN operator. When the PARK_CODE values are not known beforehand, but they can be derived using a query, you must use an IN subquery. The following example lists all employees who have worked in such Theme Park.

```
SELECT     DISTINCT EMP_NUM, EMP_LNAME, EMP_FNAME, PARK_NAME
FROM       EMPLOYEE JOIN HOURS USING (EMP_NUM)
                     JOIN ATTRACTION USING (ATTRACT_NO)
                     JOIN THEMEPARK USING (PARK_CODE)
WHERE      PARK_CODE IN (SELECT PARK_CODE FROM THEMEPARK
```

WHERE PARK_NAME LIKE '%Land%');

The result of that query is shown in Figure 64.



```
SQL> SELECT DISTINCT EMP_NUM, EMP_LNAME, EMP_FNAME, PARK_NAME
  2  FROM  EMPLOYEE JOIN HOURS USING (EMP_NUM)
  3  JOIN ATTRACTION USING (ATTRACT_NO)
  4  JOIN THEMEPARK USING (PARK_CODE)
  5  WHERE PARK_CODE IN (SELECT PARK_CODE FROM THEMEPARK
  6  WHERE PARK_NAME LIKE '%Land%');

   EMP_NUM EMP_LNAME       EMP_FNAME       PARK_NAME
---------- --------------- --------------- ---------------------------
       100 Calderdale      Emma            FairyLand
       105 Namowa          Mirrelle        FairyLand
       101 Ricardo         Marshel         FairyLand
       104 Denver          Enrica          PleasureLand
       102 Arshad          Arif            PleasureLand

SQL>
```

Figure 64: Employees who work in a Theme Park LIKE 'Land'

Task 9.2 Enter and execute the above query and compare your output with that shown in Figure 64.

9.3 HAVING Subqueries

A subquery can also be used with a HAVING clause. Remember that the HAVING clause is used to restrict the output of a GROUP BY query by applying a conditional criteria to the grouped rows. For example, to list all PARK_CODEs where the total quantity of tickets sold is greater than the average quantity sold, you would write the following query:

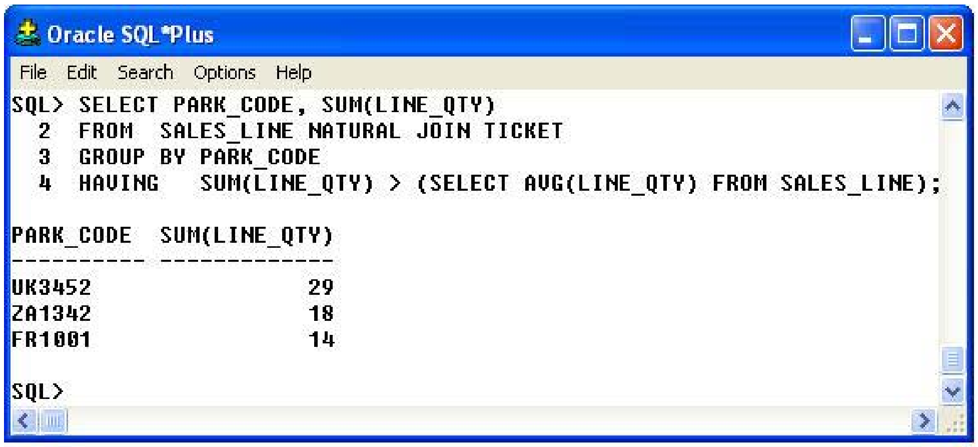
SELECT PARK_CODE, SUM(LINE_QTY)

FROM SALES_LINE NATURAL JOIN TICKET

GROUP BY PARK_CODE

HAVING   SUM(LINE_QTY) > (SELECT AVG(LINE_QTY) FROM SALES_LINE);

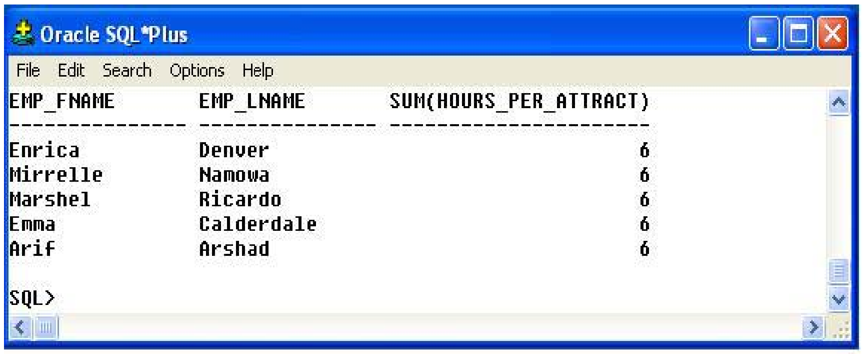
The result of that query is shown in Figure 65.



```
SQL> SELECT PARK_CODE, SUM(LINE_QTY)
  2  FROM  SALES_LINE NATURAL JOIN TICKET
  3  GROUP BY PARK_CODE
  4  HAVING   SUM(LINE_QTY) > (SELECT AVG(LINE_QTY) FROM SALES_LINE);

PARK_CODE  SUM(LINE_QTY)
---------- -------------
UK3452                29
ZA1342                18
FR1001                14

SQL>
```

Figure 65: PARK_CODES where tickets are selling above average

Task 9.3 Using the query above as a guide, write a new query to display the first and last names of all employees who have worked in total less than the average number of hours in total during May 2007. Your output should match that shown in Figure 66.



```
EMP_FNAME       EMP_LNAME       SUM(HOURS_PER_ATTRACT)
--------------- --------------- ----------------------
Enrica          Denver                               6
Mirrelle        Namowa                               6
Marshel         Ricardo                              6
Emma            Calderdale                           6
Arif            Arshad                               6

SQL>
```

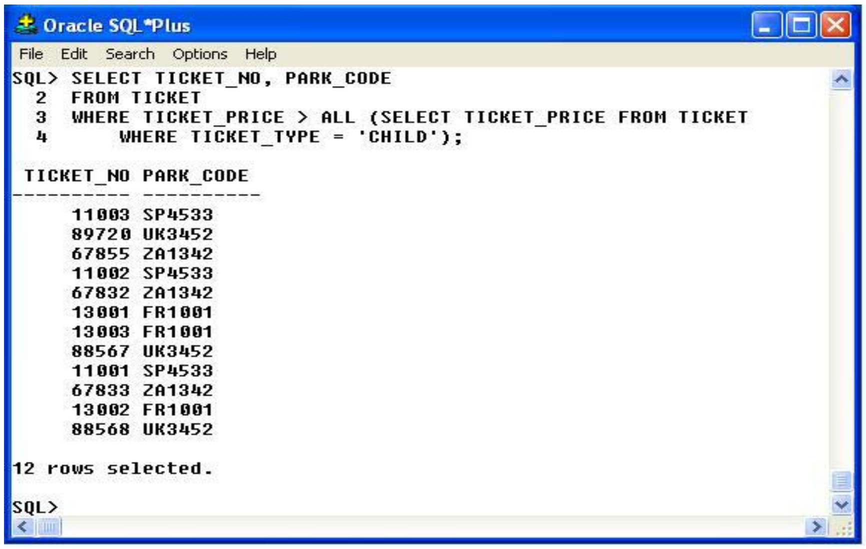
Figure 66: Output for task 9.3

9.4 Multirow Subquery operator ALL

So far, you have learned that you must use an IN subquery when you need to compare a value to a list of values. But the IN subquery uses an equality operator; that is, it selects only those rows that match (are equal to) at least one of the values in the list. What happens if you need to do an inequality comparison (> or <) of one value to a list of values? For example, to find the TICKET_NUMBERS and corresponding PARK_CODES of the tickets hat are priced higher than the highest-priced 'Child' ticket you could write the following query.

```
SELECT TICKET_NO, PARK_CODE

FROM TICKET

WHERE TICKET_PRICE > ALL (SELECT TICKET_PRICE FROM TICKET

    WHERE TICKET_TYPE = 'CHILD');
```

The output of that query is shown in Figure 67.



```
SQL> SELECT TICKET_NO, PARK_CODE
  2  FROM TICKET
  3  WHERE TICKET_PRICE > ALL (SELECT TICKET_PRICE FROM TICKET
  4      WHERE TICKET_TYPE = 'CHILD');

 TICKET_NO PARK_CODE
---------- ----------
     11003 SP4533
     89720 UK3452
     67855 ZA1342
     11002 SP4533
     67832 ZA1342
     13001 FR1001
     13003 FR1001
     88567 UK3452
     11001 SP4533
     67833 ZA1342
     13002 FR1001
     88568 UK3452

12 rows selected.

SQL>
```

Figure 67: Example of ALL

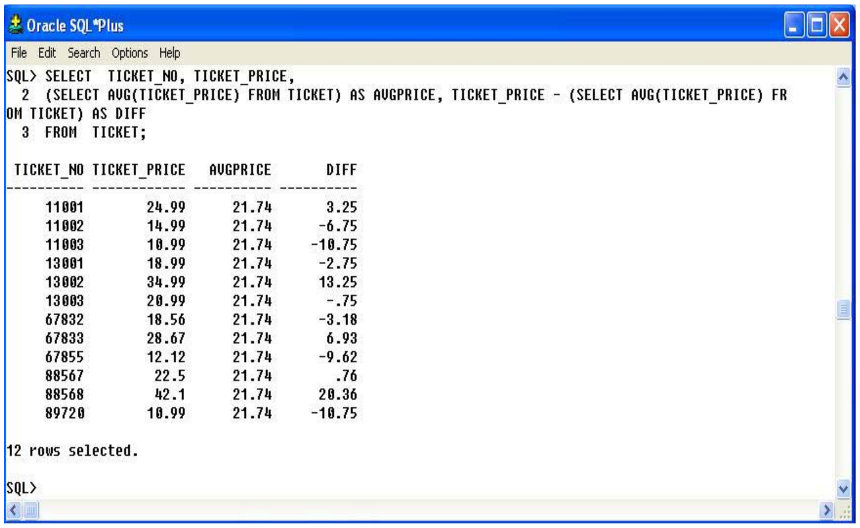
This query is a typical example of a nested query. The use of the ALL operator allows you to compare a single value (TICKET_PRICE) with a list of values returned by the nested query, using a comparison operator other than equals. For a row to appear in the result set, it has to meet the criterion TICKET_PRICE > ALL of the individual values returned by the nested query.

9.5 Attribute list Subqueries

The SELECT statement uses the attribute list to indicate what columns to project in the resulting set. Those columns can be attributes of base tables or computed attributes, or the result of an aggregate function. The attribute list can also include a subquery expression, also known as an inline subquery. A subquery in the attribute list must return one single value; otherwise an error code is raised. For example, a simple inline query can be used to list the difference between each ticket's price and the average ticket price:

```
SELECT      TICKET_NO, TICKET_PRICE,
        (SELECT AVG(TICKET_PRICE) FROM TICKET) AS AVGPRICE,
        TICKET_PRICE - (SELECT AVG(TICKET_PRICE) FROM TICKET) AS DIFF
FROM  TICKET;
```

The output for this query is shown in Figure 68.

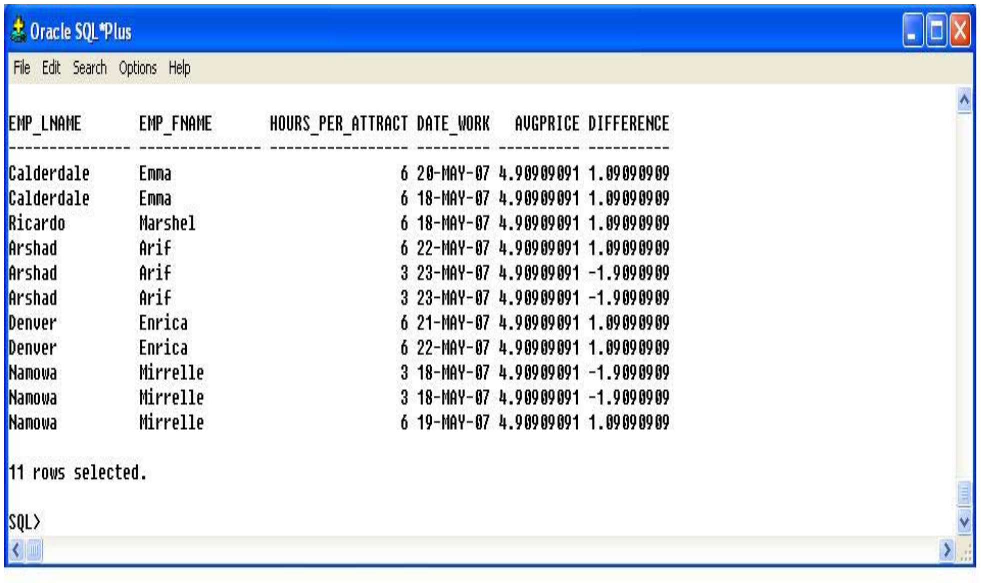


```
SQL> SELECT  TICKET_NO, TICKET_PRICE,
  2  (SELECT AVG(TICKET_PRICE) FROM TICKET) AS AVGPRICE, TICKET_PRICE - (SELECT AVG(TICKET_PRICE) FR
OM TICKET) AS DIFF
  3  FROM  TICKET;

 TICKET_NO TICKET_PRICE   AVGPRICE       DIFF
---------- ------------ ---------- ----------
     11001        24.99      21.74       3.25
     11002        14.99      21.74      -6.75
     11003        10.99      21.74     -10.75
     13001        18.99      21.74      -2.75
     13002        34.99      21.74      13.25
     13003        20.99      21.74       -.75
     67832        18.56      21.74      -3.18
     67833        28.67      21.74       6.93
     67855        12.12      21.74      -9.62
     88567         22.5      21.74        .76
     88568         42.1      21.74      20.36
     89720        10.99      21.74     -10.75

12 rows selected.

SQL>
```

Figure 68: Displaying the difference in ticket prices

This inline query output returns one single value (the average ticket's price) and that value is the same in every row. Note also that the query used the full expression instead of the column aliases when computing the difference. In fact, if you try to use the alias in the difference expression, you will get an error message. The column alias cannot be used in computations in the attribute list when the alias is defined in the same attribute list.

Task 9.4 Write a query to display an employee's first name, last name and date worked which lists the difference between the number of hours an employee has worked on an attraction and the average hours worked on that attraction. Label this column 'DIFFERENCE' and the average hours column 'AVERAGE'. Check your output against that shown in Figure 69.



```
EMP_LNAME       EMP_FNAME       HOURS_PER_ATTRACT DATE_WORK   AVGPRICE DIFFERENCE
--------------- --------------- ----------------- --------- ---------- ----------
Calderdale      Emma                            6 20-MAY-07 4.90909091 1.09090909
Calderdale      Emma                            6 18-MAY-07 4.90909091 1.09090909
Ricardo         Marshel                         6 18-MAY-07 4.90909091 1.09090909
Arshad          Arif                            6 22-MAY-07 4.90909091 1.09090909
Arshad          Arif                            3 23-MAY-07 4.90909091 -1.9090909
Arshad          Arif                            3 23-MAY-07 4.90909091 -1.9090909
Denver          Enrica                          6 21-MAY-07 4.90909091 1.09090909
Denver          Enrica                          6 22-MAY-07 4.90909091 1.09090909
Namowa          Mirrelle                        3 18-MAY-07 4.90909091 -1.9090909
Namowa          Mirrelle                        3 18-MAY-07 4.90909091 -1.9090909
Namowa          Mirrelle                        6 19-MAY-07 4.90909091 1.09090909

11 rows selected.

SQL>
```

Figure 69: Output for task 9.4.

9.6 Correlated Subqueries

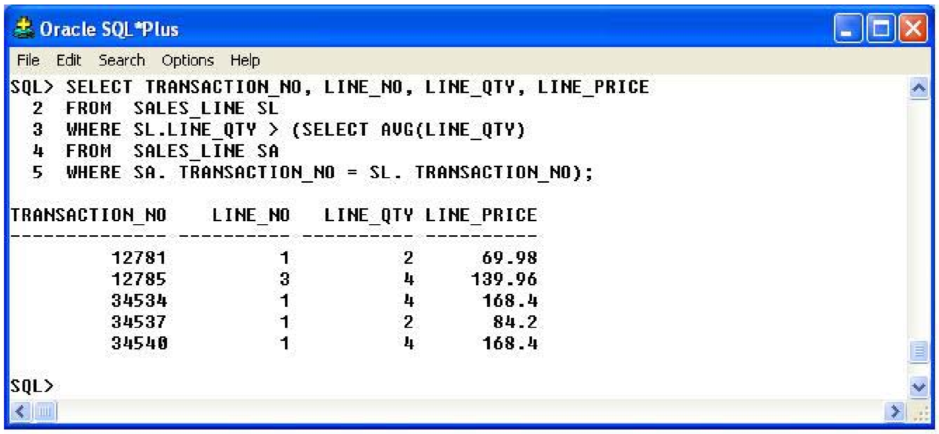
A correlated subquery is a subquery that executes once for each row in the outer query.

The relational DBMS uses the same sequence to produce correlated subquery results:

1. It initiates the outer query
2. For each row of the outer query result set, it executes the inner query by passing the outer row to the inner query

This process is the opposite of the subqueries you have seen so far. The query is called a *correlated* subquery because the inner query is *related* to the outer query, and because the inner query references a column of the outer subquery. For example, suppose you want to know all the ticket sales in which the quantity sold value is greater than the average quantity sold value for *that* ticket (as opposed to the average for *all tickets*). The following correlated query completes the preceding two-step process:

```
SELECT      TRANSACTION_NO, LINE_NO, LINE_QTY, LINE_PRICE

FROM        SALES_LINE SL

WHERE       SL.LINE_QTY > (SELECT AVG(LINE_QTY)

FROM        SALES_LINE SA

WHERE       SA. TRANSACTION_NO = SL. TRANSACTION_NO);
```



```
SQL> SELECT TRANSACTION_NO, LINE_NO, LINE_QTY, LINE_PRICE
  2  FROM  SALES_LINE SL
  3  WHERE SL.LINE_QTY > (SELECT AVG(LINE_QTY)
  4  FROM  SALES_LINE SA
  5  WHERE SA. TRANSACTION_NO = SL. TRANSACTION_NO);

TRANSACTION_NO    LINE_NO   LINE_QTY LINE_PRICE
-------------- ---------- ---------- ----------
         12781          1          2      69.98
         12785          3          4     139.96
         34534          1          4      168.4
         34537          1          2       84.2
         34540          1          4      168.4

SQL>
```

Figure 70: Example of a correlated subquery

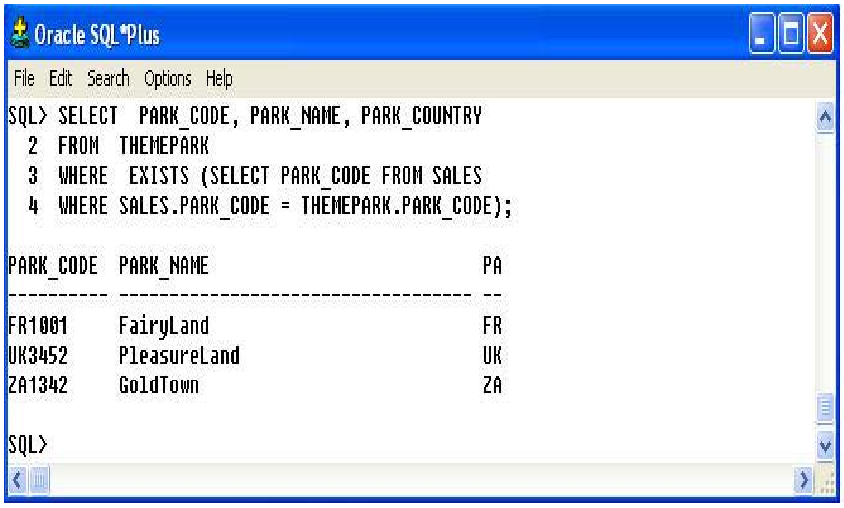
As you examine the output shown in Figure 70, note that the SALES_LINE table is used more than once; so you must use table aliases.

Correlated subqueries can also be used with the EXISTS special operator. For example, suppose you want to know all the names of all Theme Parks where tickets have been recently sold. In that case, you could use a correlated subquery as follows:

```
SELECT      PARK_CODE, PARK_NAME, PARK_COUNTRY

FROM        THEMEPARK

WHERE       EXISTS (SELECT PARK_CODE FROM SALES
```

WHERE SALES.PARK_CODE = THEMEPARK.PARK_CODE);

The output for this query is shown in Figure 71.



```
SQL> SELECT  PARK_CODE, PARK_NAME, PARK_COUNTRY
  2  FROM  THEMEPARK
  3  WHERE  EXISTS (SELECT PARK_CODE FROM SALES
  4  WHERE SALES.PARK_CODE = THEMEPARK.PARK_CODE);

PARK_CODE  PARK_NAME                            PA
---------- ----------------------------------- --
FR1001     FairyLand                            FR
UK3452     PleasureLand                         UK
ZA1342     GoldTown                             ZA

SQL>
```

Figure 71: Example of correlated subqueries

Task 9.5 Type in and execute the two correlated subqueries in this section and check your output against that shown in Figures 70 and 71.

Task 9.6 Modify the second query you entered in task 9.5 to display all the Theme Parks were there have been no recorded tickets sales recently.

Lab 10: Views

The learning objectives of this lab are to:

- Create a simple view
- Manage database constraints in views using the READ ONLY and WITH CHECK OPTION

10.1 Views

A **view** is a virtual table based on a SELECT query. The query can contain columns, computed columns, aliases, and aggregate functions from one or more tables. The tables on which the view is based are called **base tables**. You can create a view by using the **CREATE VIEW** command:

CREATE VIEW *viewname* AS SELECT *query*

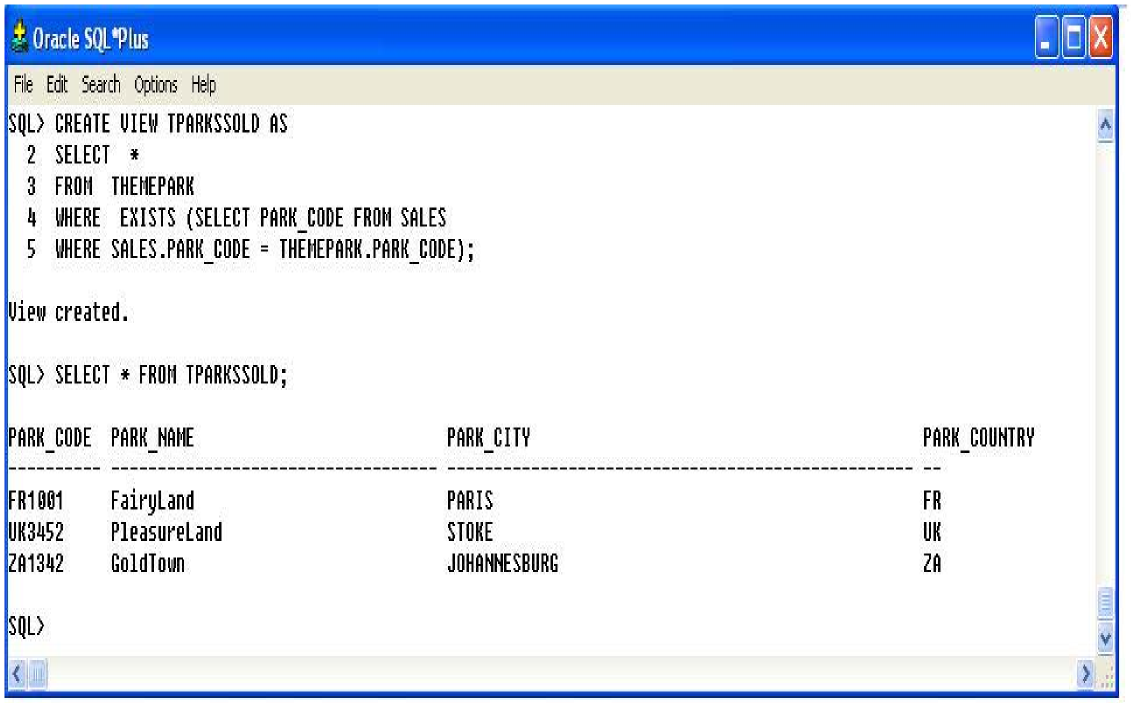
The CREATE VIEW statement is a data definition command that stores the subquery specification – the SELECT statement used to generate the virtual table – in the data dictionary. For example, to create a view of only those Theme Parks were tickets have been sold you would do so as follows:

```
CREATE VIEW TPARKSSOLD AS
SELECT      *
FROM        THEMEPARK
WHERE       EXISTS (SELECT PARK_CODE FROM SALES
WHERE       SALES.PARK_CODE = THEMEPARK.PARK_CODE);
```

To display the contents of this view you would type:

SELECT * FROM TPARKSSOLD;

The created view can be seen in Figure 72.



```
SQL> CREATE VIEW TPARKSSOLD AS
  2  SELECT  *
  3  FROM  THEMEPARK
  4  WHERE  EXISTS (SELECT PARK_CODE FROM SALES
  5  WHERE SALES.PARK_CODE = THEMEPARK.PARK_CODE);

View created.

SQL> SELECT * FROM TPARKSSOLD;

PARK_CODE  PARK_NAME                             PARK_CITY                                          PARK_COUNTRY
---------- ------------------------------------ -------------------------------------------------- --
FR1001     FairyLand                             PARIS                                              FR
UK3452     PleasureLand                          STOKE                                              UK
ZA1342     GoldTown                              JOHANNESBURG                                       ZA

SQL>
```

Figure 72: Creating the TPARKSSOLD view

Task 10.1 Create the TPARKSSOLD view.

As you will have learned in Chapter 8, Introduction to Structured Query Language, relational view has several special characteristics. These are worth repeating here.

- You can use the name of a view anywhere a table name is expected in a SQL statement.

- Views are dynamically updated. That is, the view is re-created on demand each time it is invoked. Therefore, if more tickets are sold in other Theme Parks, then those new ticket sales will automatically appear (or disappear) in the TPARKSSOLD view the next time it is invoked.
- Views provide a level of security in the database because the view can restrict users to only specified columns and specified rows in a table.

To remove the view TPARKSSOLD you could issue the following command
DROP VIEW TPARKSSOLD;

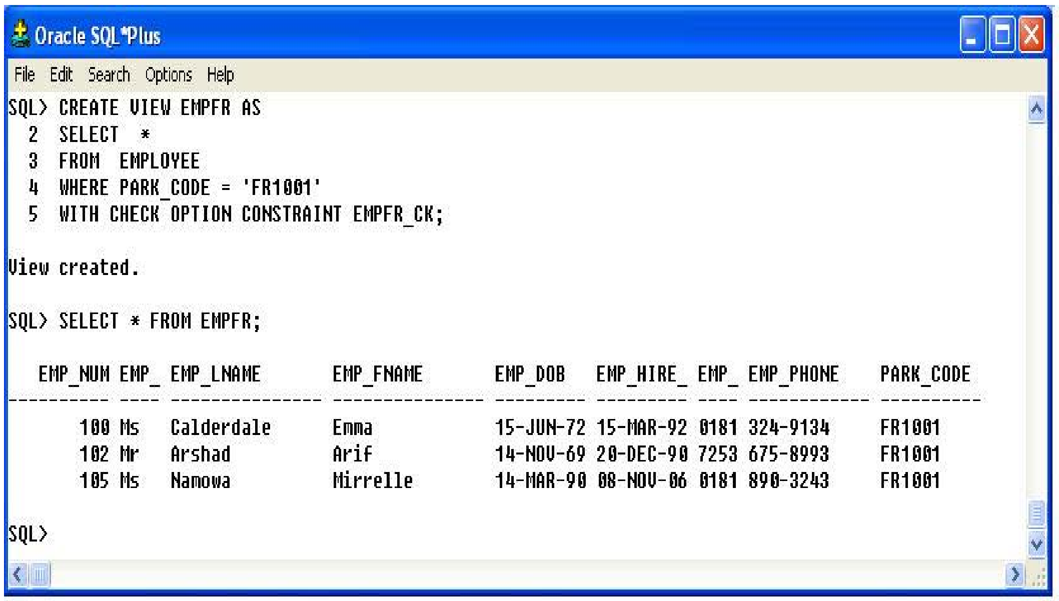
Task 10.2 Create a view called TICKET_SALES which contains details of the min, max and average sales at each Theme Park. The name of the Theme Park should also be displayed. Hint 1: You will need to join three tables. Hint 2: You will need to give the columns in the query that use the functions an alias. Once you have created your view, write a query to display the contents.

Task 10.3 Add your view TICKET_SALES and the associated DROP command to your themepark.sql scrip you created in Lab 2.

10.2 Views – using the WITH CHECK OPTION

It is possible to perform referential integrity constraints through the use of a view so that database constraints can be enforced. The following view DISPLAYS employees who work in Theme Park FR1001 using the WITH CHECK OPTION clause. This clause ensures that INSERTs and UPDATEs cannot be performed on any rows that the view has not selected. The results of creating this view can be seen in Figure 73.

```
CREATE VIEW EMPFR       AS
SELECT         *
FROM           EMPLOYEE
WHERE PARK_CODE = ‘FR1001’
WITH CHECK OPTION CONSTRAINT EMPFR_CK;
```



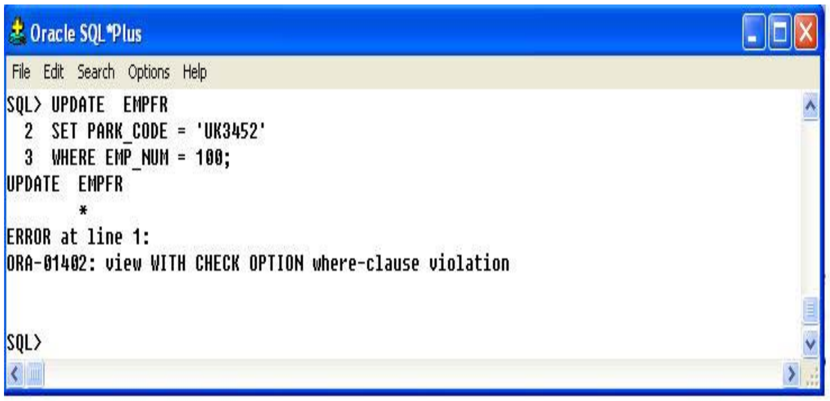
```
SQL> CREATE VIEW EMPFR AS
  2  SELECT  *
  3  FROM  EMPLOYEE
  4  WHERE PARK_CODE = 'FR1001'
  5  WITH CHECK OPTION CONSTRAINT EMPFR_CK;

View created.

SQL> SELECT * FROM EMPFR;

   EMP_NUM EMP_ EMP_LNAME       EMP_FNAME       EMP_DOB   EMP_HIRE_ EMP_ EMP_PHONE    PARK_CODE
---------- ---- --------------- --------------- --------- --------- ---- ------------ ----------
       100 Ms   Calderdale      Emma            15-JUN-72 15-MAR-92 0181 324-9134     FR1001
       102 Mr   Arshad          Arif            14-NOV-69 20-DEC-90 7253 675-8993     FR1001
       105 Ms   Namowa          Mirrelle        14-MAR-90 08-NOV-06 0181 890-3243     FR1001

SQL>
```

Figure 73: Creating the EMPFR view

So for example if employee ‘Emma Caulderdale’ was to leave the park and move to park ‘UK3452’, we would want to update her information with the following query:

```
UPDATE  EMPFR
SET PARK_CODE = 'UK3452'
WHERE EMP_NUM = 100;
```

However running this update gives the errors shown in Figure 74. This is because if the update was to occur, the view would no longer be able to see this employee.



```
SQL> UPDATE  EMPFR
  2  SET PARK_CODE = 'UK3452'
  3  WHERE EMP_NUM = 100;
UPDATE  EMPFR
        *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation


SQL>
```

Figure 74: Creating the EMPFR view

Task 10.4 Create the view EMPFR and tray and update the Theme Park that employee number 101 works in.

Task 10.5. Employee Emma Cauderdale (EMP_NUM =100) has now changed her phone number to 324-9652. Update her information in the EMPFR view. Write a query to show her new phone number has been updated.

Task 10.6 Remove the EMPFR view.

10.3 Views – using the READ ONLY option

By adding the READ ONLY option to a view, you can ensure that no changes at all can be made to the data inside it. This includes all insertions, updates and deletions. The EMPFR view with a READ ONLY option would be created as follows:

```
CREATE VIEW EMPFR      AS
SELECT        *
FROM          EMPLOYEE
WHERE PARK_CODE = ‘FR1001’
WITH READ ONLY;
```

Task 10.7 Create the EMPFR view with the READ ONLY option. Then try and delete EMP_NUM 100 and observe what happens.

10.4 Exercises

E10.1 The Theme Park managers want to create a view called EMP_DETAILS which contains the following information:
EMP_NO, PARK_CODE, PARK_NAME, EMP_LNAME_EMP_FNAME, EMP_HIRE_DATE and EMP_DOB.
The view should only be read only.

E10.2 Check that the view works by displaying its contents.

E10.3 Using your view EMP_DETAILS, write a query that displays all employee first and last names and the park names.

E10.4 Attempt to update the park name ‘Labyrinthe’ to ‘MazeHaze’ to check if the constraints on your view work.

E10.5 Remove the view EMPDETAILS.

CONCLUSION

You have now reached the end of this ORACLE SQL lab guide. Only a few examples are shown in this tutorial. The objective is not to develop full-blown applications, but to show you some examples of the fundamental features of SQL which you can build on with further reading and practice.

FURTHER READING

Loney, K. ORACLE Database 10g: The Complete Reference, Osborne McGraw-Hill, 2005.

Price J. ORACLE Database 10g SQL (ORACLE Press S.), Osborne McGraw-Hill, 2005

WEB SITES

ORACLE Technology Network	**http://otn.oracle.com/**
ORACLE Main Site	**http://www.oracle.com/**