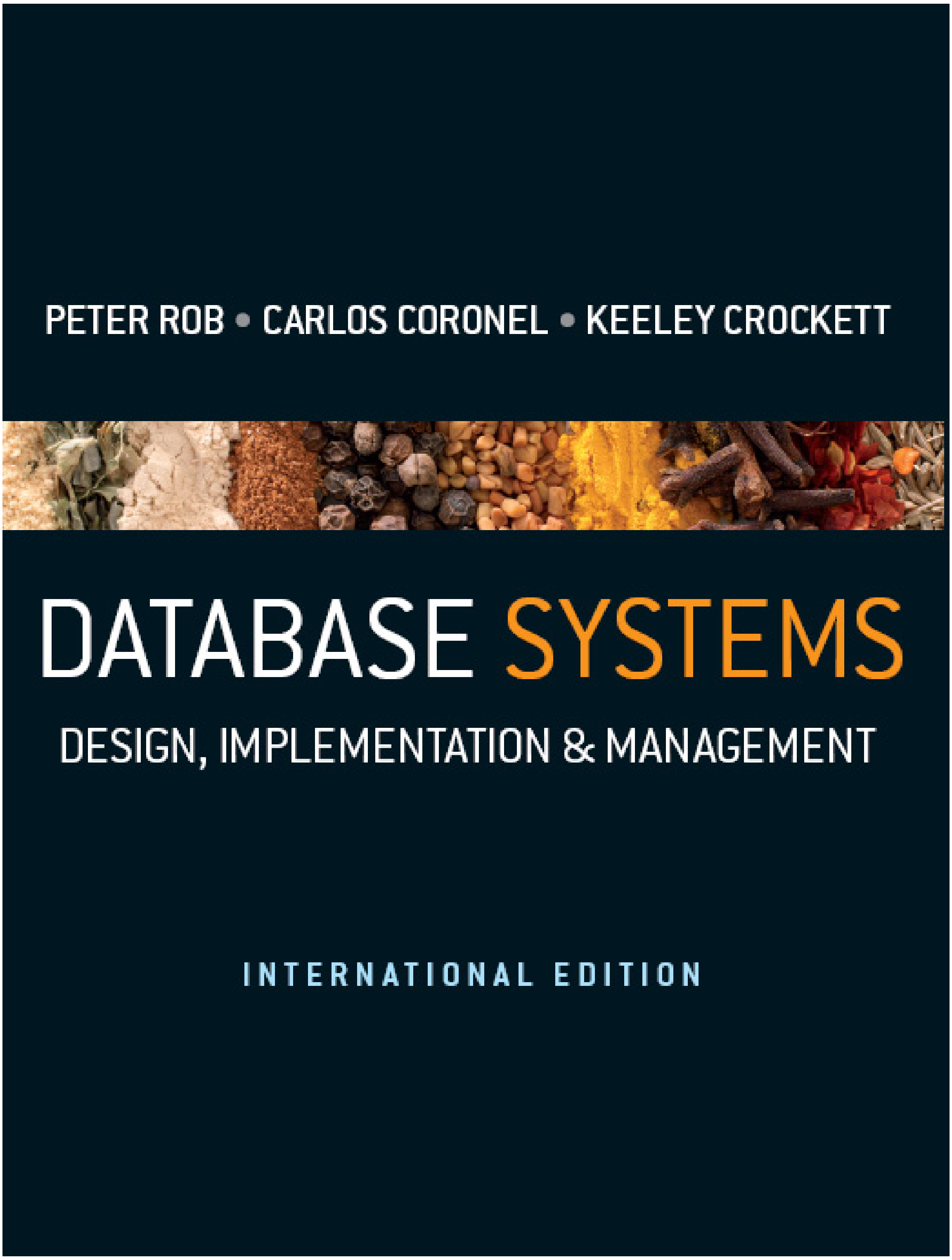
COURSE TECHNOLOGY
CENGAGE Learning™



PETER ROB • CARLOS CORONEL • KEELEY CROCKETT

DATABASE SYSTEMS

DESIGN, IMPLEMENTATION & MANAGEMENT

INTERNATIONAL EDITION

MySQL Lab Guide

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

Table of Contents

Introduction to the MySQL Lab Guide

This lab guide is designed to provide examples and exercises in the fundamentals of SQL within the MySQL 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 9 sessions. Each one comprises of examples, tasks and exercises about a particular concept in SQL and how it is implemented in MySQL.

On completion of this 9 week lab guide you will be able to:

- Create a simple relational database in MySQL.
- Insert, update and delete data the tables.
- Create queries using basic and advanced SELECT statements
- Perform join operations on relational tables
- 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 drag, how to use drag and drop, how you save a file, and so on.

MySQL, is one of the most popular Open Source SQL database management systems. The lab guide has been designed on MySQL version 5.0.45 running on Windows XP

Professional. The MySQL Web site (http://www.mysql.com/) provides the latest information about MySQL database management system.

It is important to note that MySQL is an open source database and is continually under development. Each version and sub-version may implement SQL syntax differently and changes are being made constantly. There are also problems with upward compatibility between different versions. For example some SQL operations that work in versions 3.0 and 4.0 do not work in version 5.0. Furthermore, different variants of a version are released in response to bugs that have been found by database developers who are using the latest versions in their work. If an SQL command does not work as expected or shown in this guide, please consult the MySQL web site for more information.

Lab 1: Starting MySQL

The learning objectives of this lab are to

- Learn how to start MySQL
- Learn how to use the MySQL command line client window
- Obtain help in MySQL

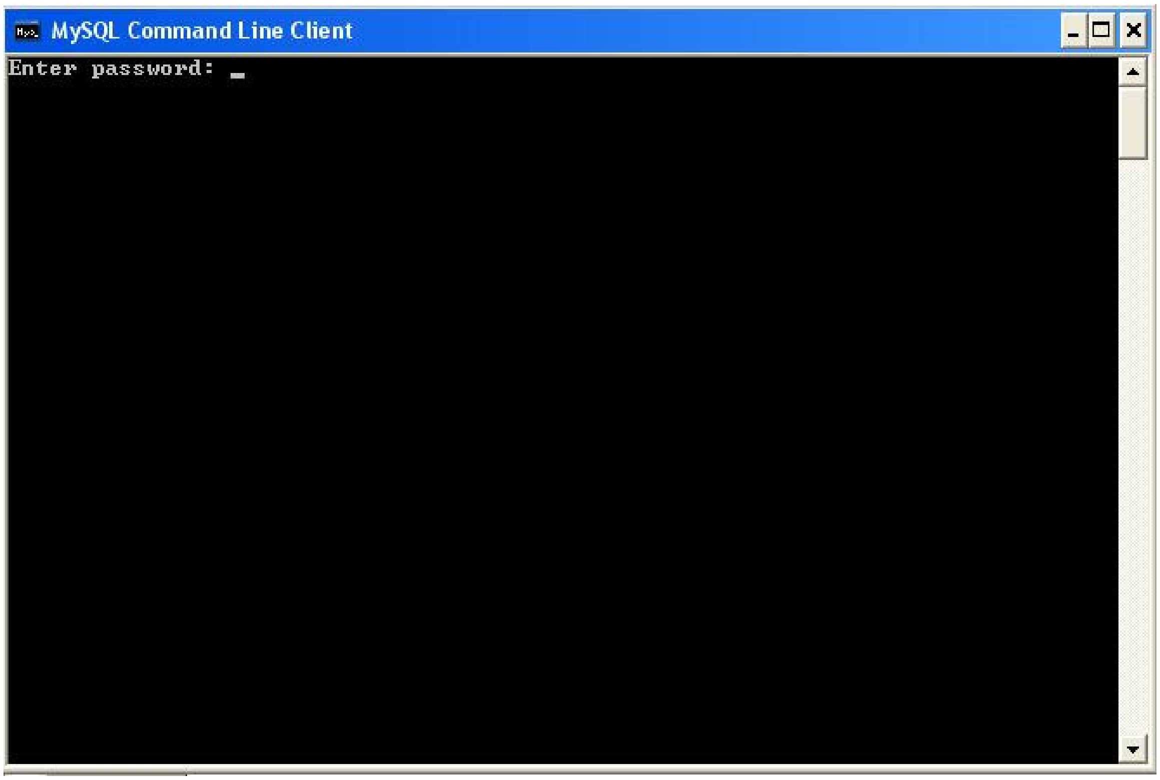
1.1 Starting MySQL

Before starting this guide, you must obtain a user ID and a password created by your database administrator in order to log on to the MySQL RDBMS. How you connect to the MySQL database depends on how the MySQL software was installed on your server and on the access paths and methods defined and managed by the database administrator. You may therefore need to follow specific instructions provided by your instructor, College or University. This section will describe how to start MySQL from a Windows XP installation of MySQL 5.0.45.

To start MySQL you would:

1. Select the Start button
2. Select All Programs and then MySQL
3. Select MySQL Server 5.0
4. Click on the MySQL Command line client

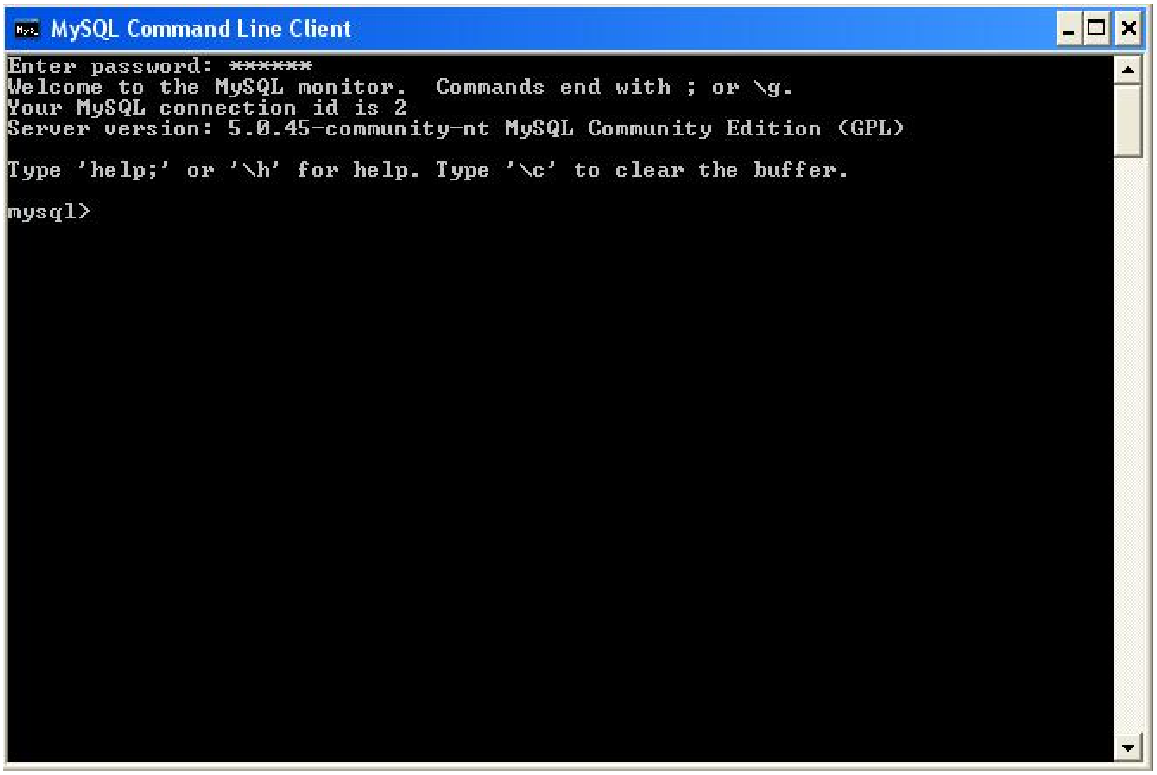
The MySQL command line client window should then open as shown in Figure 1.



```
Enter password: _
```

Figure 1: MySQL command line client window

Enter your password to log on to MySQL as shown in Figure 2.

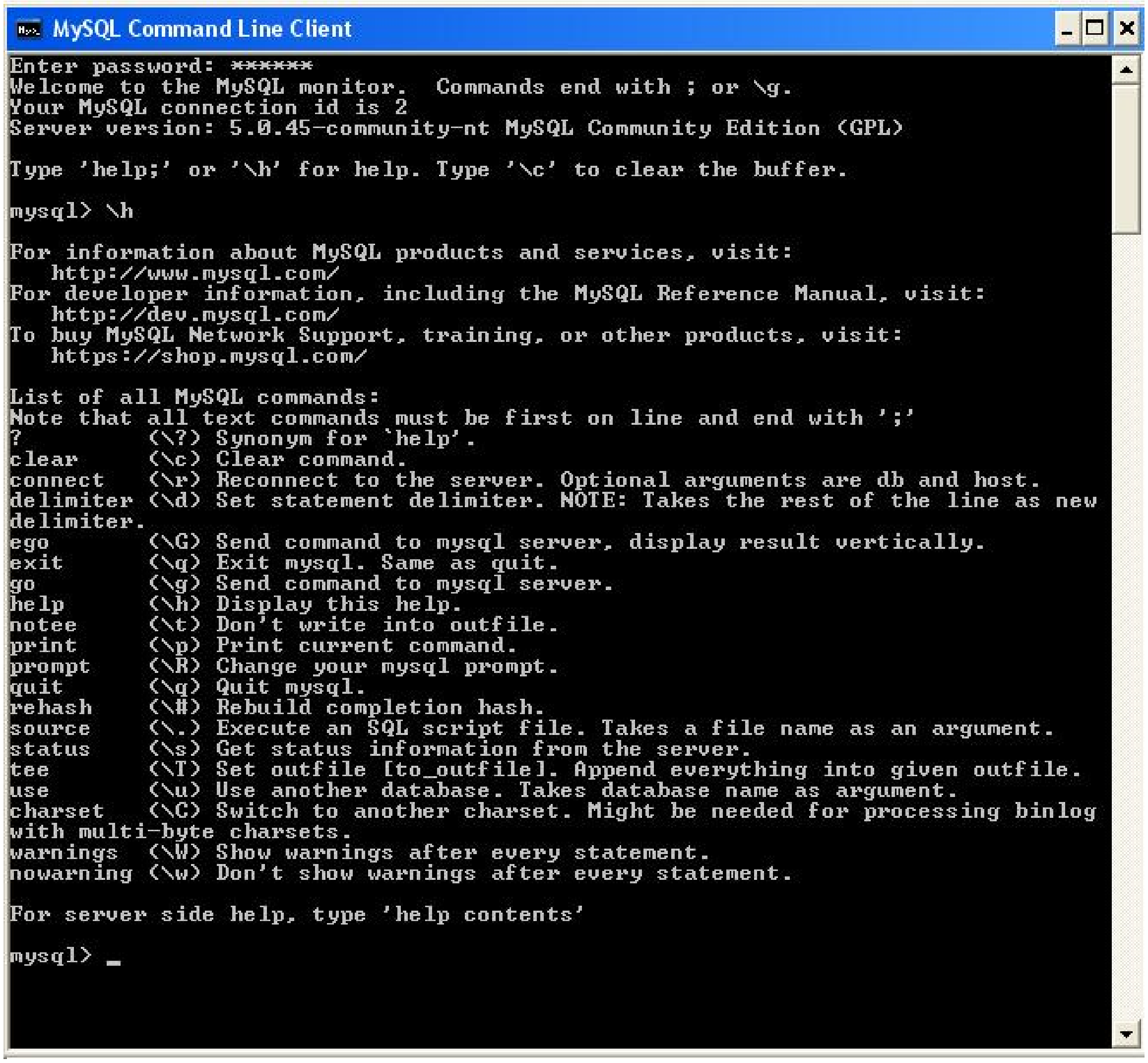


```
Enter password: *******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.45-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Figure 2: Logging on to MySQL

Once you have successfully logged on you will see the opening screen as shown in Figure 2. To work in MySQL requires you to type in commands. For example typing in the following will show you a list of help commands shown in Figure 3:

mysql> \h



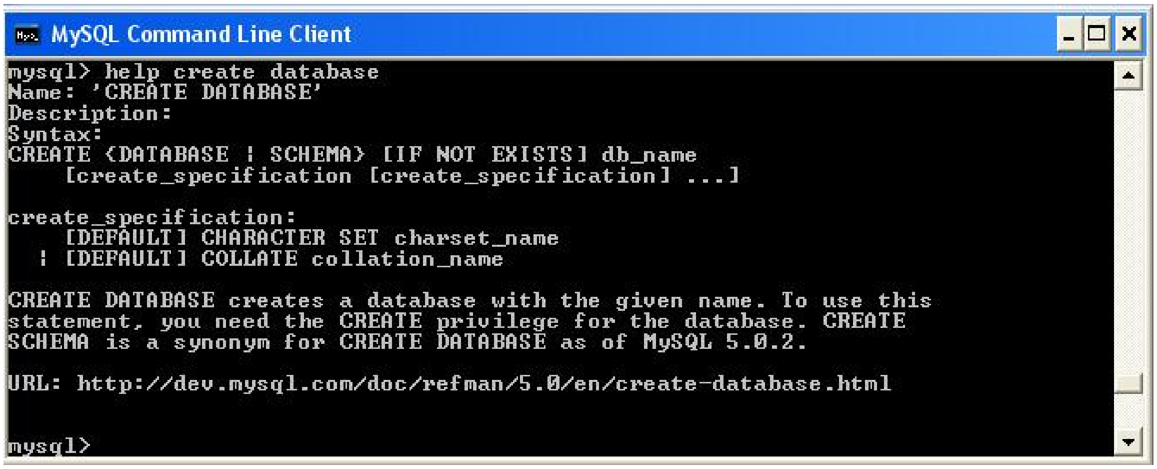
```
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.0.45-community-nt MySQL Community Edition (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> \h

For information about MySQL products and services, visit:
   http://www.mysql.com/
For developer information, including the MySQL Reference Manual, visit:
   http://dev.mysql.com/
To buy MySQL Network Support, training, or other products, visit:
   https://shop.mysql.com/

List of all MySQL commands:
Note that all text commands must be first on line and end with ';'
?         (\?) Synonym for `help'.
clear     (\c) Clear command.
connect   (\r) Reconnect to the server. Optional arguments are db and host.
delimiter (\d) Set statement delimiter. NOTE: Takes the rest of the line as new
delimiter.
ego       (\G) Send command to mysql server, display result vertically.
exit      (\q) Exit mysql. Same as quit.
go        (\g) Send command to mysql server.
help      (\h) Display this help.
notee     (\t) Don't write into outfile.
print     (\p) Print current command.
prompt    (\R) Change your mysql prompt.
quit      (\q) Quit mysql.
rehash    (\#) Rebuild completion hash.
source    (\.) Execute an SQL script file. Takes a file name as an argument.
status    (\s) Get status information from the server.
tee       (\T) Set outfile [to_outfile]. Append everything into given outfile.
use       (\u) Use another database. Takes database name as argument.
charset   (\C) Switch to another charset. Might be needed for processing binlog
with multi-byte charsets.
warnings  (\W) Show warnings after every statement.
nowarning (\w) Don't show warnings after every statement.

For server side help, type 'help contents'

mysql> _
```

Figure 3: Help commands in MySQL

Figure 3 shows some additional sources of help available from three different websites. It also displays a list of commands and the shortcuts for running these commands. If you want help about a specific command you can type the word help followed by the name of the command. For example to display information about how to create a database you would type:

mysql> help create database

Figure 4 shows the results of executing this command.



```
mysql> help create database
Name: 'CREATE DATABASE'
Description:
Syntax:
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_specification [create_specification] ...]

create_specification:
    [DEFAULT] CHARACTER SET charset_name
  | [DEFAULT] COLLATE collation_name

CREATE DATABASE creates a database with the given name. To use this
statement, you need the CREATE privilege for the database. CREATE
SCHEMA is a synonym for CREATE DATABASE as of MySQL 5.0.2.

URL: http://dev.mysql.com/doc/refman/5.0/en/create-database.html


mysql>
```

Figure 4: Example Help command

A full list of help topics available through the command line can be found by first typing:

mysql> help contents

However to get more detailed help you would use the MySQL reference manual. If you are using MySQL from a Windows XP installation, then you can access the manual via the programs menu as shown in Figure 5.

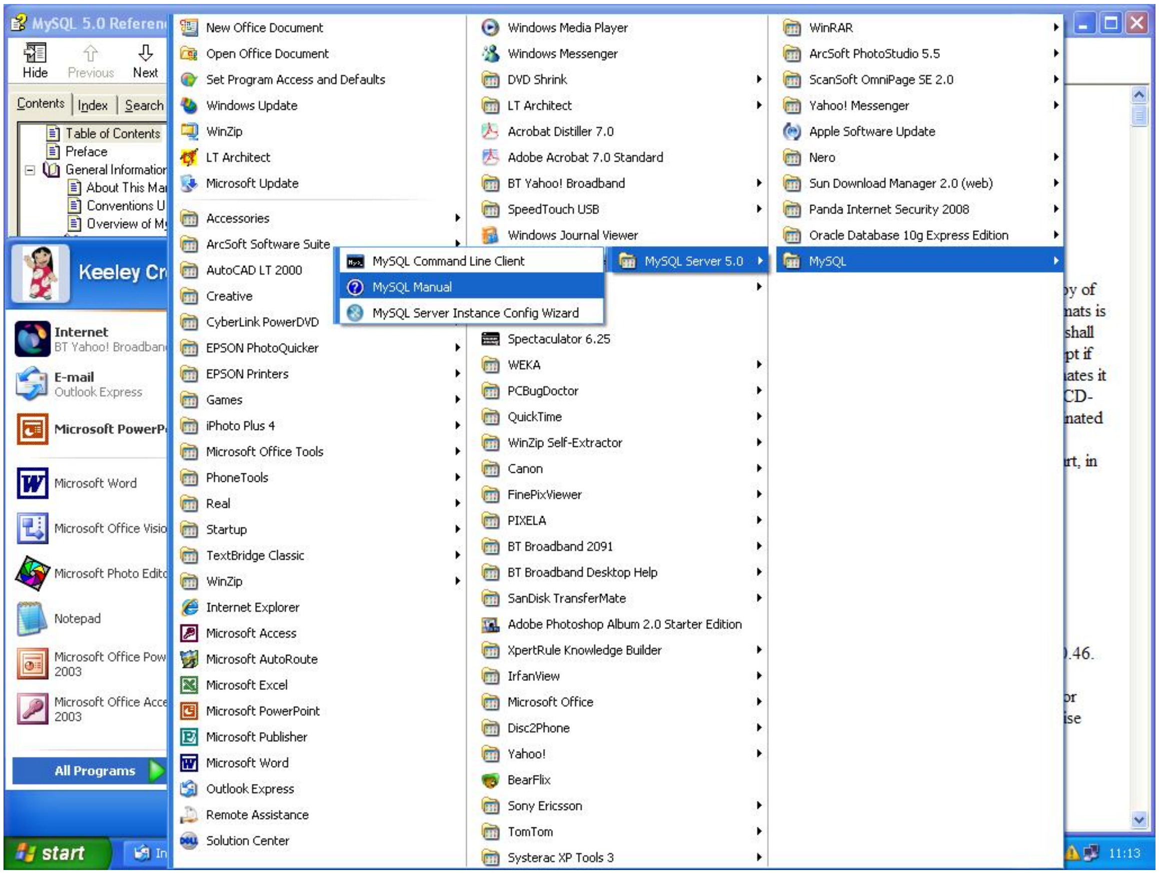
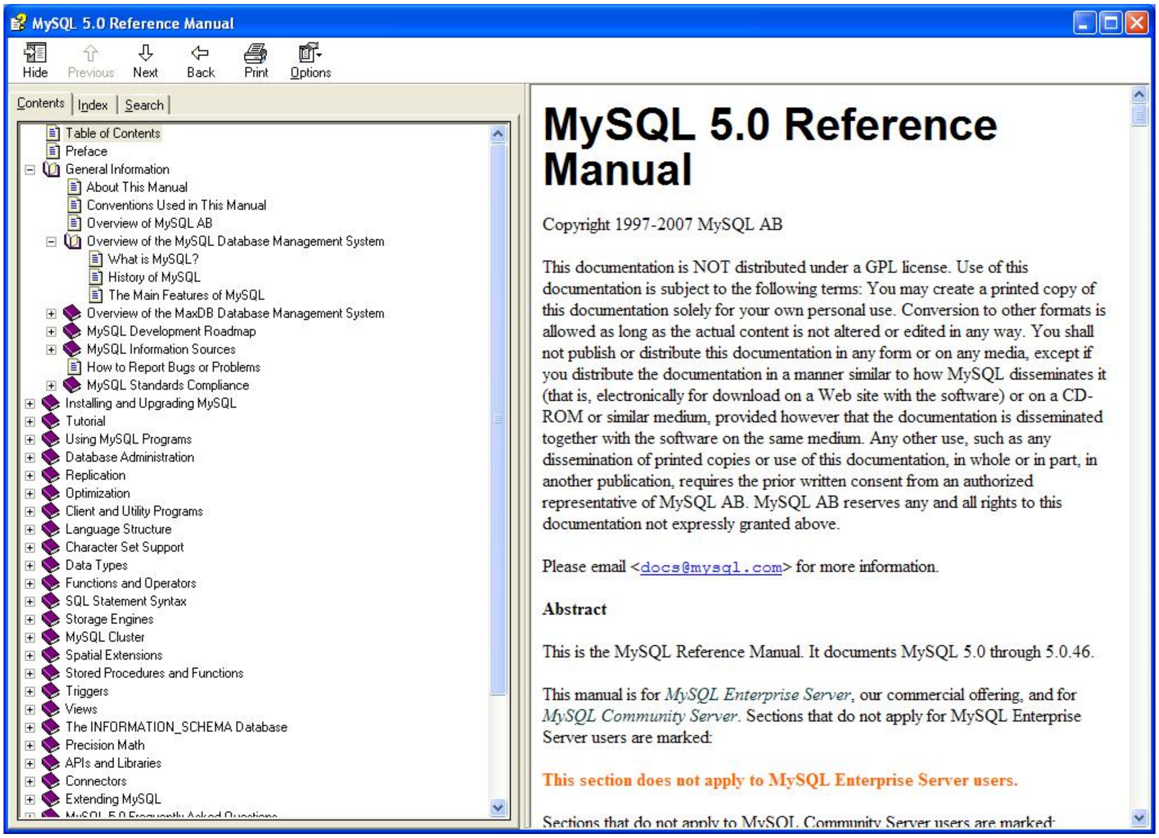


Figure 5: Accessing the MySQL Reference Manual

Figure 6 shows the table of contents for the reference manual.



MySQL 5.0 Reference Manual

Copyright 1997-2007 MySQL AB

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms: You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how MySQL disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of MySQL AB. MySQL AB reserves any and all rights to this documentation not expressly granted above.

Please email <docs@mysql.com> for more information.

Abstract

This is the MySQL Reference Manual. It documents MySQL 5.0 through 5.0.46.

This manual is for *MySQL Enterprise Server*, our commercial offering, and for *MySQL Community Server*. Sections that do not apply for MySQL Enterprise Server users are marked:

This section does not apply to MySQL Enterprise Server users.

Sections that do not apply to MySQL Community Server users are marked:

Figure 6: Contents of the MySQL Reference Manual

1.2 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 7 in the form of an Entity Relationship Diagram (ERD).

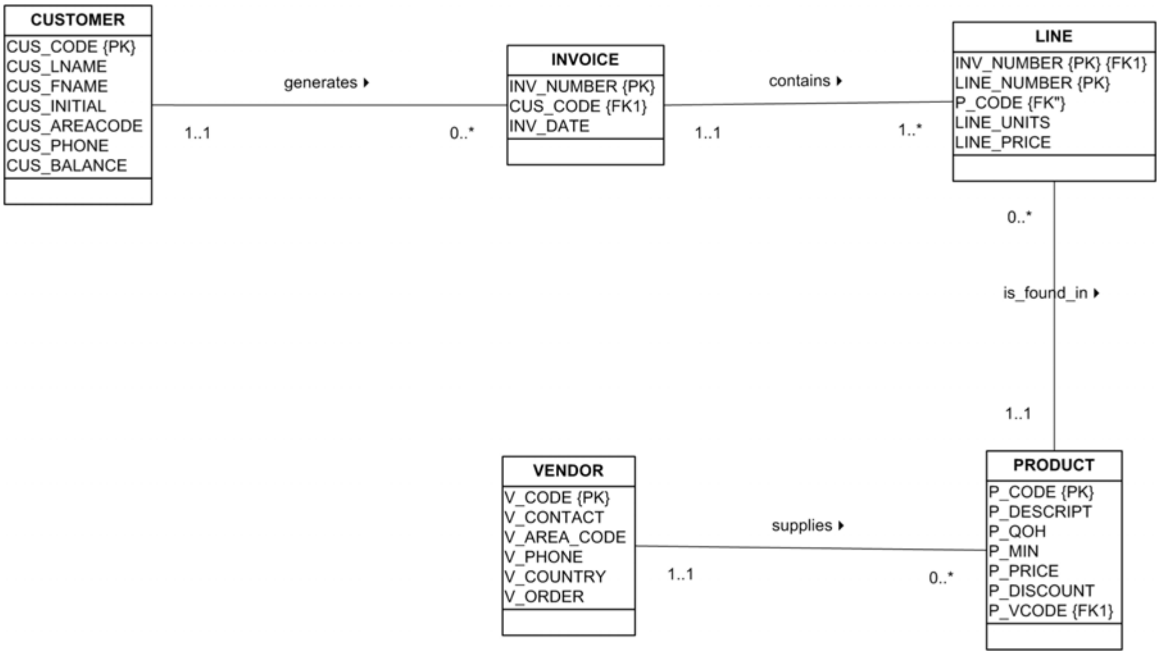
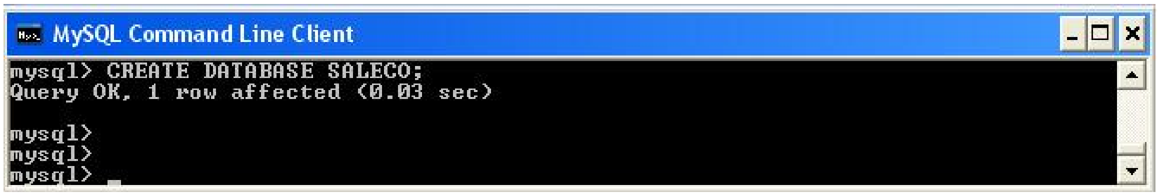


Figure 7 The SaleCo Database ERD

Before creating any tables, MySQL requires you to create a database by executing the **CREATE DATABASE** command. To create a database called SaleCo you would type the following:

mysql> CREATE DATABASE SALECO;

Notice that you need a semi-colon to end the command. Figure 8 shows the successful creation of this database.



```
mysql> CREATE DATABASE SALECO;
Query OK, 1 row affected (0.03 sec)

mysql>
mysql>
mysql> _
```

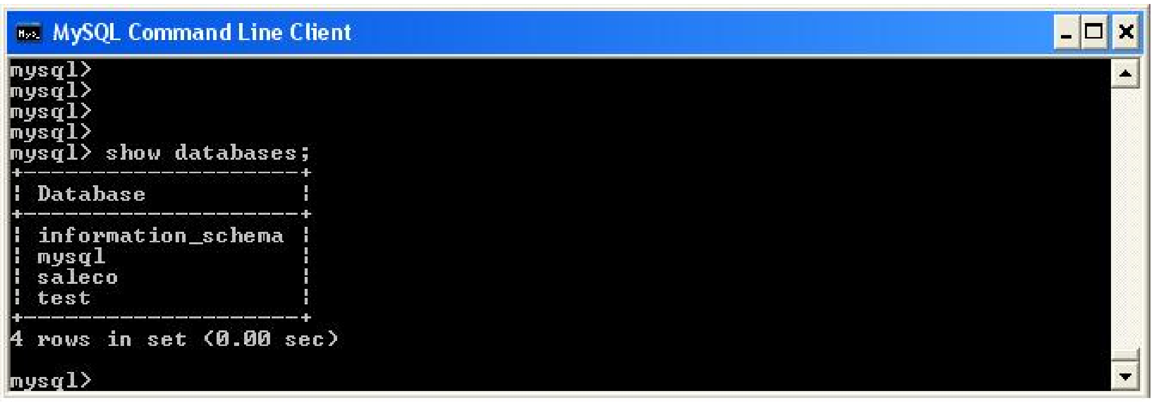
Figure 8 Creating the SaleCo Database

Task 1.1 Create the SALECO database as shown in Figure 8.

To check to see if your database has been created you need to use the SHOW DATABASES command which lists the databases on the MySQL server host. You will only be able to see those databases for which you have some kind of privilege.

Task 1.2 Execute the following MySQL command to show the databases that you currently have access to (Figure 9 is a guide only to what you should see). Check that you can see the SALECO database that you have just created.

mysql> SHOW DATABASES;



```
mysql>
mysql>
mysql>
mysql>
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| saleco             |
| test               |
+--------------------+
4 rows in set (0.00 sec)

mysql>
```

Figure 9 Executing the SHOW DATABASES command

To work with any specific database you first have to select it. When you first login to MySQL, the default database is always selected, so you need to execute the **USE** command followed by the name of the database that you want to use.

Task 1.3 Execute the following MySQL command to begin using the SALECO database.

mysql> USE SALECO;

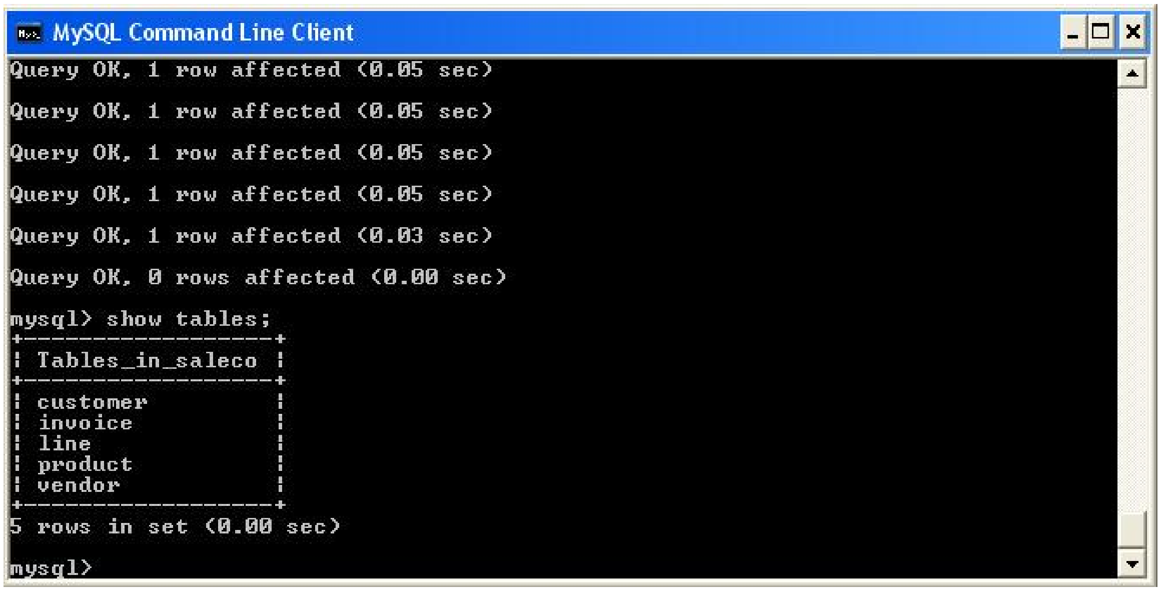
MySQL will then inform you that the database has changed.

Task 1.4 To create the SaleCo database from a MySQL script file you would enter the following command:

```
mysql> SOURCE  C:\MYSQL\SALECO2.SQL
```

Note that in order for this command to work correctly you should have copied the script files accompanying this Lab guide into the directory C:\MYSQL\. If your files are located in a different directory then change the path accordingly.

The command **SOURCE** 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 10. When the script has completed executing, use the **SHOW TABLES** command as shown in Figure 10, to check if all five tables have been created.



```
Query OK, 1 row affected (0.05 sec)

Query OK, 1 row affected (0.05 sec)

Query OK, 1 row affected (0.05 sec)

Query OK, 1 row affected (0.05 sec)

Query OK, 1 row affected (0.03 sec)

Query OK, 0 rows affected (0.00 sec)

mysql> show tables;
+-------------------+
| Tables_in_saleco |
+-------------------+
| customer          |
| invoice           |
| line              |
| product           |
| vendor            |
+-------------------+
5 rows in set (0.00 sec)

mysql>
```

Figure 10. Creating the SaleCo database

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.

Lab 2: Building a database: Table by Table

The learning objectives of this lab are to

- Create table structures using MySQL data types
- Apply SQL constraints to MySQL 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 11. This will involve you creating the table structures in MySQL 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 Structured Query Language, and Appendix D, Converting an ER Model into a Database Structure.

2.2 The Theme Park Database

Figure 11 shows the ERD for the Theme Park database which will be used throughout this lab guide.

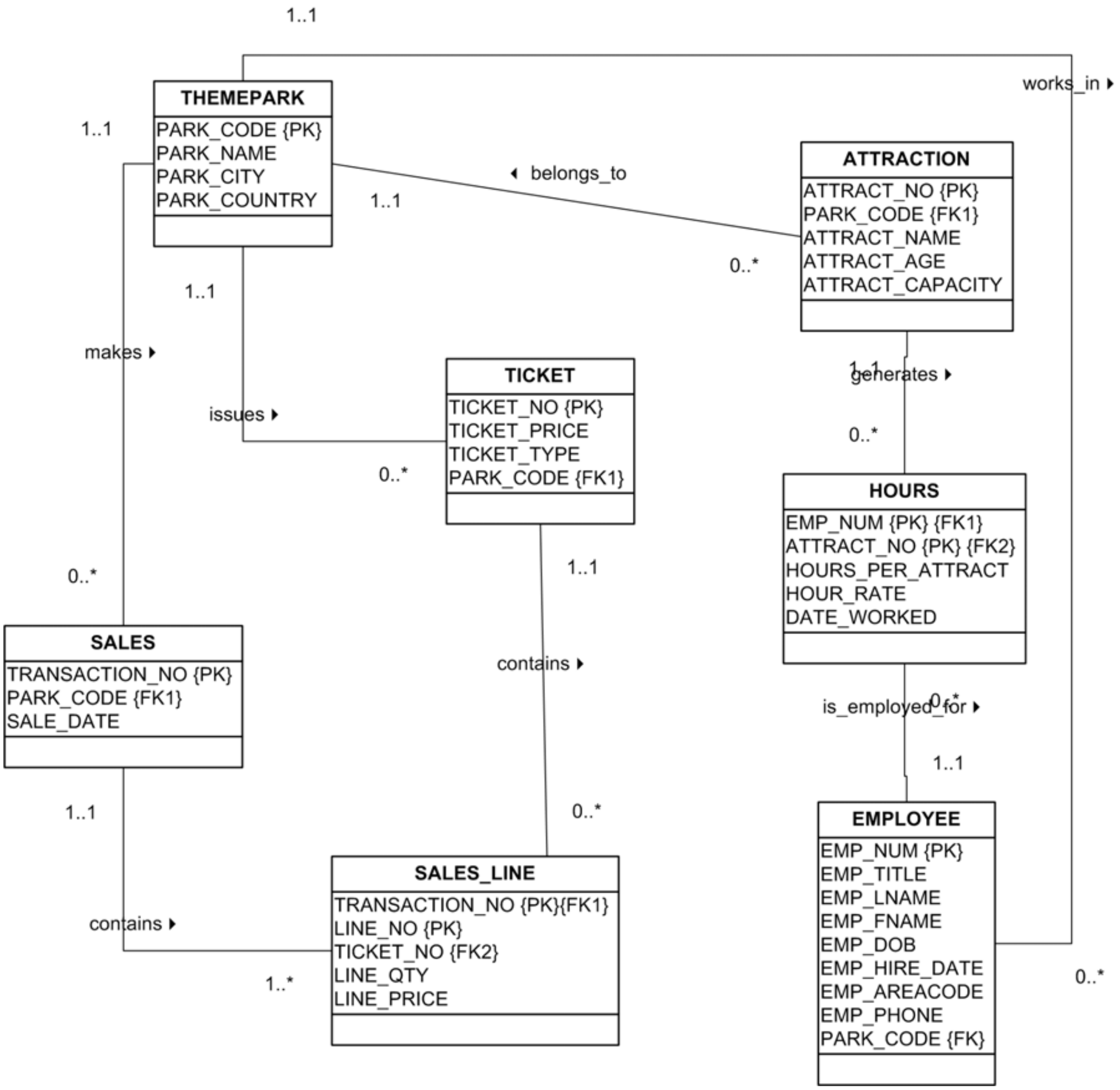


Figure 11 The Theme park Database ERD

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	VARCHAR(10)	XXXXXXXX	NA	Y	PK	
	PARK_NAME	Park Name	VARCHAR(35)	XXXXXXXX	NA	Y		
	PARK_CITY	City	VARCHAR(50)		NA	Y		
	PARK_COUNTRY	Country	CHAR(2)	XX	NA	Y		
EMPLOYEE	EMP_NUM	Employee number	NUMERIC(4)	##	0000 – 9999	Y	PK	
	EMP_TITLE	Employee title	VARCHAR(4)	XXXX	NA	N		
	EMP_LNAME	Last name	VARCHAR(15)	XXXXXXXX	NA	Y		
	EMP_FNAME	First Name	VARCHAR(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	VARCHAR(4)	XXXX	NA	Y		
	EMP_PHONE	Phone	VARCHAR (12)	XXXXXXXX	NA	Y		
	PARK_CODE	Park code	VARCHAR(10)	XXXXXXXX	NA	Y	FK	THEMEPARK

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

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

2.3 Data Types in MySQL

In order to build tables in MySQL you will need to specify the data type for each column. Table 2.2 shows some of the most common data types. If you have previously used an ORACLE DBMS, you will notice that the syntax is different.

Table 2.2 Common MySQL data types[1]

Data Type	Example	Description
CHAR(size)	fieldName CHAR(10)	Stores up to 255 characters. If the content is smaller than the field size, the content will have trailing spaces appended.
VARCHAR(size)	fieldName VARCHAR(100)	Stores up to 255 characters, and a minimum of 4 characters. No trailing spaces are appended to the end of this datatype.

[1] This table was adapted from the web site http://www.developerfusion.co.uk/. A comprehensive and complete list of types can be taken from the MySQL Reference Manual.

		MySQL keeps track of a delimiter to keep track of the end of the field.
TINYTEXT	fieldName TINYTEXT	Stores up to 255 characters. Equivalent to VARCHAR(255).
TEXT	fieldName TEXT	Stores up to 65,535 characters. An Index can be created on the first 255 characters of a field with this data type.
MEDIUMTEXT	fieldName MEDIUMTEXT	Stores up to 16,777,215 characters. An Index can be created on the first 255 characters of a field with this data type.
LONGTEXT	fieldName LONGTEXT	Stores up to 4,294,967,295 characters. An Index can be created on the first 255 characters of a field with this data type. Note: The maximum size of a string in MySQL is currently 16 million bytes, so this data types is not useful at the moment.
ENUM	fieldName ENUM('Yes', 'No')	Stores up to 65,535 enumerated types. The DEFAULT modifier may be used to specify the default value for this field.
INT	fieldName INT	Stores a signed or unsigned integer number. Unsigned integers have a range of 0 to 4,294,967,295, and signed integers have a range of -2,147,438,648 to 2,147,438,647. By default, the INT data type is signed. To create an unsigned integer, use the UNSIGNED attribute. fieldName INT UNSIGNED The ZEROFILL attribute may be used to left-pad any of the integer with zero's. fieldName INT ZEROFILL The AUTO_INCREMENT attribute may be used with any of the Integer data types. The following example could be used to create a primary key using the AUTO_INCREMEMNT attribute. fieldName INT UNSIGNED AUTO_INCREMENT PRIMARY KEY
TINYINT	fieldName TINYINT	Stores a signed or unsigned byte. Unsigned bytes have a range of 0 to 255, and signed bytes have a range of -128 to 127. By default, the TINYINT data type is signed.
MEDIUMINT	fieldName MEDIUMINT	Stores a signed or unsigned medium sized integer. Unsigned fields of this type have a range of 0 to 1,677,215, and signed fields of this type have a range of -8,388,608 to 8,388,607. By default, the MEDIUMINT data type is signed.
BIGINT	fieldName BIGINT	Stores a signed or unsigned big integer. Unsigned fields of this type have a range of 0 to 18,446,744,073,709,551,615, and signed fields of this type have a range of -9,223,372,036,854,775,808 to 9,223,327,036,854,775,807. By default, the BIGINT data type is signed.

<table>
<tr><td>FLOAT</td><td>fieldName FLOAT</td><td>Used for single precision floating point numbers.</td></tr>
<tr><td>DOUBLE</td><td>fieldName DOUBLE</td><td>Used for double precision floating point numbers.</td></tr>
<tr><td>DATE</td><td>fieldName DATE</td><td>Stores dates in the format YYYY-MM-DD.</td></tr>
<tr><td>TIMESTAMP(size)</td><td>fieldName DATETIME</td><td>Stores dates and times in the format YYYY-MM-DD HH:MM:SS.</td></tr>
<tr><td>DATETIME</td><td>fieldName TIMESTAMP(14)</td><td>Automatically keeps track of the time the record was last ammended. The following table shows the formats depending on the size of TIMESTAMP
<table>
<tr><th>Size</th><th>Format</th></tr>
<tr><td>2</td><td>YY</td></tr>
<tr><td>4</td><td>YYMM</td></tr>
<tr><td>6</td><td>YYMMDD</td></tr>
<tr><td>8</td><td>YYYYMMDD</td></tr>
<tr><td>10</td><td>YYYYMMDDHH</td></tr>
<tr><td>12</td><td>YYYYMMDDHHMM</td></tr>
<tr><td>14</td><td>YYYYMMDDHHMMSS</td></tr>
</table></td></tr>
<tr><td>TIME</td><td>fieldName TIME</td><td>Stores times in the format HH:MM:SS.</td></tr>
<tr><td>YEAR(size)</td><td>fieldName YEAR(4)</td><td>Stores the year as either a 2 digit number, or a 4 digit number, depending on the size provided.</td></tr>
</table>

2.4 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 MySQL to say “Query OK”. 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 and the MySQL data types in Table 2.2.

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 a table. The end user may, of course, enter a value other than the default value. In MYSQL the default value must be a constant; it cannot be a function or an expression. This means, for example, that you cannot set the default for a date column to be the value of a function such as the system date like you can do in an ORACLE DBMS.

Note

You will have learnt in Chapter 8 that referential integrity is usually implemented through the use of foreign keys. For a long time, the open-source MySQL RDBMS did not support the use of foreign keys. However, given the importance of maintaining referential integrity within the database this feature was introduced in later versions through the InnoDB table engine. The InnoDB engine provides MySQL with an ACID (Atomicity, Consistency, Isolation, Durability) compliant storage engine that has facilities such as commit and rollback. Full information about the InnoDB engine can be found in the MySQL Reference manual 5.0.

- The FOREIGN KEY CONSTRAINT is used to enforce referential integrity. In order to set up a foreign key relationship between two MySQL tables, three conditions must be met:

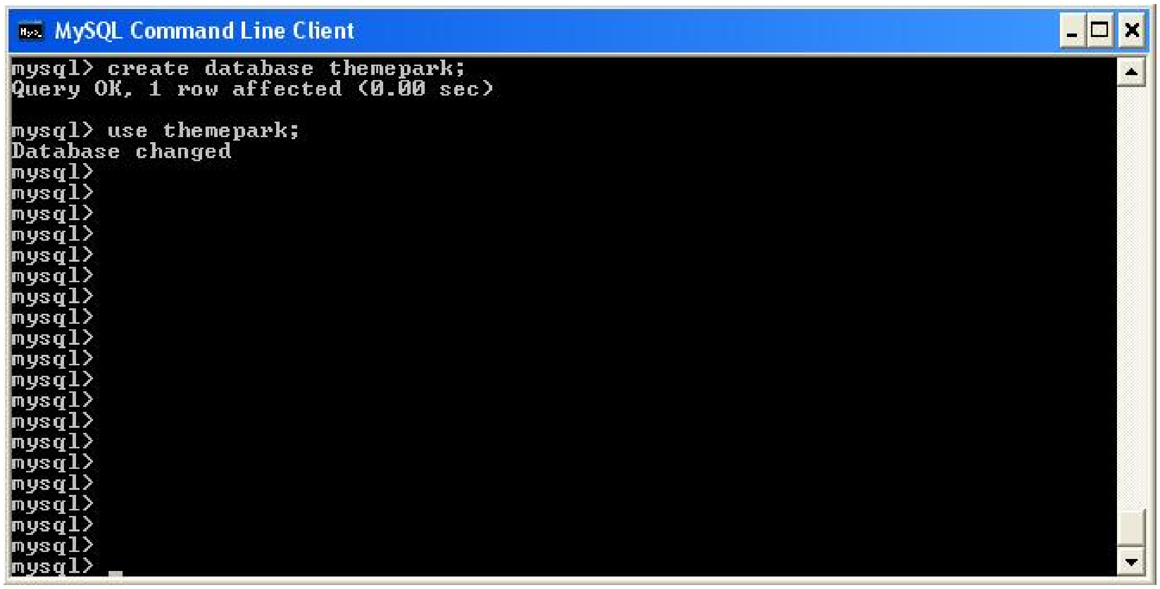
  1. Both tables must be of the InnoDB table type - see the note box.
  2. The fields used in the foreign key relationship must be indexed.
  3. The fields used in the foreign key relationship must be similar in data type.

Note

MySQL 5.0 does not support the use of CHECK constraints which is used to validate data when an attribute value is entered.

2.4.1 Creating the THEMEPARK Database.

Task 2.1 At the MySQL prompt; create a database called Theme Park as shown in Lab 1. Then select the database for use as shown in Figure 12.



```
mysql> create database themepark;
Query OK, 1 row affected (0.00 sec)

mysql> use themepark;
Database changed
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql> _
```

Figure 12 Creating and using the Theme Park Database.

2.4.2 Creating the THEMEPARK TABLE

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

```
CREATE TABLE      THEMEPARK (
PARK_CODE       VARCHAR(10) PRIMARY KEY,
PARK_NAME       VARCHAR(35) NOT NULL,
PARK_CITY       VARCHAR(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       VARCHAR(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 does not accept nulls.

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

2.4.3 Creating the EMPLOYEE TABLE

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

```
CREATE TABLE EMPLOYEE (
EMP_NUM         NUMERIC(4) PRIMARY KEY,
EMP_TITLE       VARCHAR(4),
EMP_LNAME       VARCHAR(15) NOT NULL,
EMP_FNAME       VARCHAR(15) NOT NULL,
EMP_DOB         DATE NOT NULL,
EMP_HIRE_DATE   DATE,
EMP_AREA_CODE   VARCHAR(4) NOT NULL,
EMP_PHONE       VARCHAR(12) NOT NULL,
PARK_CODE       VARCHAR(10),
INDEX           (PARK_CODE),
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. In order to use foreign key constraints in MySQL, notice that the PARK_CODE column is first indexed. 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 and that you cannot have an invalid entry in the foreign key column.

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

2.4.4 Creating the TICKET TABLE

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

```
CREATE TABLE TICKET (
TICKET_NO       NUMERIC(10) PRIMARY KEY,
TICKET_PRICE      NUMERIC(4,2) DEFAULT 00.00 NOT NULL,

TICKET_TYPE       VARCHAR(10),

PARK_CODE         VARCHAR(10),

INDEX             (PARK_CODE),

CONSTRAINT        FK_TICKET_PARK FOREIGN KEY(PARK_CODE)
REFERENCES THEMEPARK(PARK_CODE));
```

As you create the TICKET table, notice that both PRIMARY and FOREIGN KEY constraints have been applied. Remember to store this CREATE TABLE structure in your themepark.sq script.

2.4.5 Creating the ATTRACTION TABLE

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

```
CREATE      TABLE ATTRACTION (

ATTRACT_NO               NUMERIC(10) PRIMARY KEY,

ATTRACT_NAME             VARCHAR(35),

ATTRACT_AGE              NUMERIC(3) DEFAULT 0 NOT NULL,

ATTRACT_CAPACITY         NUMERIC(3) NOT NULL,

PARK_CODE                VARCHAR(10),

INDEX                    (PARK_CODE),
CONSTRAINT          FK_ATTRACT_PARK FOREIGN KEY(PARK_CODE)
REFERENCES THEMEPARK(PARK_CODE));
```

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

2.4.6 Creating the HOURS TABLE

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

```
CREATE TABLE HOURS (
EMP_NUM                     NUMERIC(4),
ATTRACT_NO                  NUMERIC(10),
HOURS_PER_ATTRACT           NUMERIC(2) NOT NULL,
HOUR_RATE                   NUMERIC(4,2) NOT NULL,
DATE_WORKED                 DATE NOT NULL,
INDEX                       (EMP_NUM),
INDEX                       (ATTRACT_NO),
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 table.

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

2.4.7 Creating the SALES TABLE

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

```
CREATE TABLE SALES (
TRANSACTION_NO          NUMERIC PRIMARY KEY,
PARK_CODE               VARCHAR(10),
SALE_DATE               DATE NOT NULL,
INDEX                   (PARK_CODE),
CONSTRAINT          FK_SALES_PARK FOREIGN KEY(PARK_CODE)
REFERENCES THEMEPARK(PARK_CODE));
```

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

2.4.8 Creating the SALESLINE TABLE

Task 2.8 Enter the following SQL command to create the SALES_LINE table.

```
CREATE TABLE SALES_LINE (

TRANSACTION_NO          NUMERIC,

LINE_NO                 NUMERIC(2,0) NOT NULL,

TICKET_NO               NUMERIC(10)  NOT NULL,

LINE_QTY                NUMERIC(4) DEFAULT 0 NOT NULL,

LINE_PRICE              NUMERIC(9,2) DEFAULT 0.00 NOT NULL,

INDEX                   (TRANSACTION_NO),

INDEX                   (TICKET_NO),

CONSTRAINT      PK_SALES_LINE     PRIMARY KEY
(TRANSACTION_NO,LINE_NO),

CONSTRAINT      FK_SALES_LINE_SALES  FOREIGN KEY
(TRANSACTION_NO) REFERENCES SALES(TRANSACTION_NO) 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.

2.5. 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 EMP_LNAME stored in the EMPLOYEE table, the following command creates an index named EMP_LNAME_INDEX:

```
CREATE INDEX EMP_LNAME_INDEX ON EMPLOYEE(EMP_LNAME(8));
```

In MySQL, indexes can only be created using only the leading part of column values. So in the example an index is created using the first 8 characters of the EMP_LNAM column.

Task 2.9 Create the EMP_LNAME_INDEX shown above. 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;

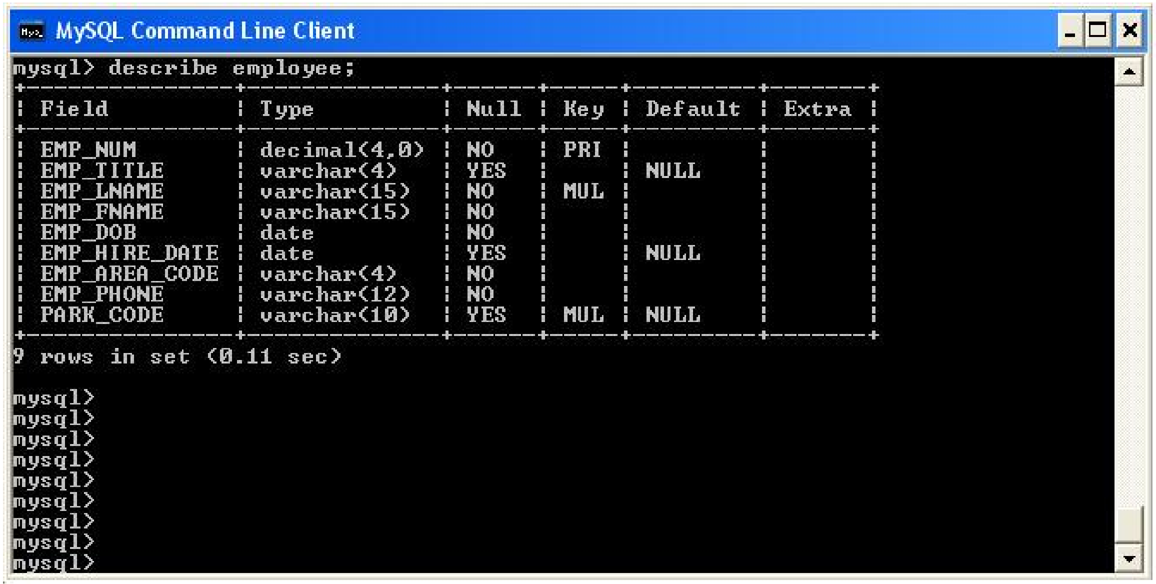
DROP TABLE THEMEPARK;

Task 2.10. Add the DROP commands to the start of your script file and then run the themepark.sql script.

2.6 Display a table's structure

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

DESCRIBE EMPLOYEE as shown in Figure 13.



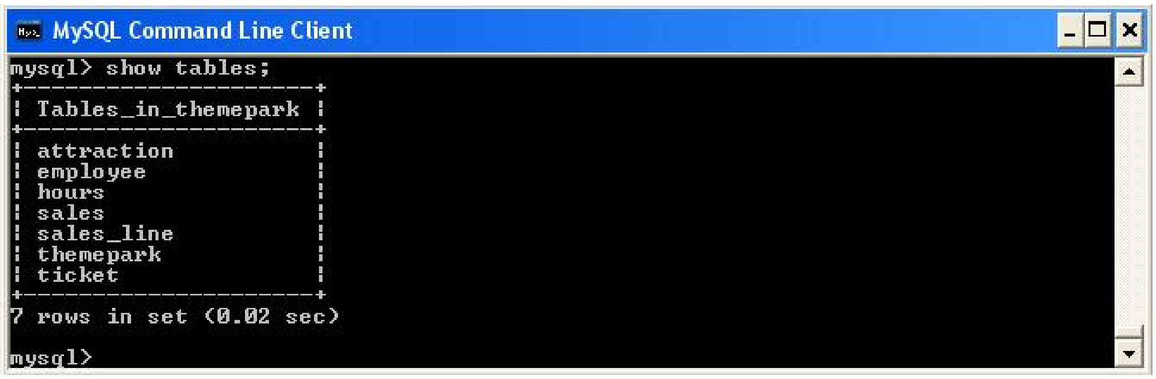
```
mysql> describe employee;
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| EMP_NUM       | decimal(4,0) | NO   | PRI |         |       |
| EMP_TITLE     | varchar(4)   | YES  |     | NULL    |       |
| EMP_LNAME     | varchar(15)  | NO   | MUL |         |       |
| EMP_FNAME     | varchar(15)  | NO   |     |         |       |
| EMP_DOB       | date         | NO   |     |         |       |
| EMP_HIRE_DATE | date         | YES  |     | NULL    |       |
| EMP_AREA_CODE | varchar(4)   | NO   |     |         |       |
| EMP_PHONE     | varchar(12)  | NO   |     |         |       |
| PARK_CODE     | varchar(10)  | YES  | MUL | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
9 rows in set (0.11 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
```

Figure 13 Describing the structure of the THEMEPARK Table

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

2.7 Listing all tables

Task 2.11 Use the SHOW TABLES command as shown in Figure 14, to list all tables that have been created within the THEMEPARK database.



```
mysql> show tables;
+----------------------+
| Tables_in_themepark |
+----------------------+
| attraction           |
| employee             |
| hours                |
| sales                |
| sales_line           |
| themepark            |
| ticket               |
+----------------------+
7 rows in set (0.02 sec)

mysql>
```

Figure 14 Displaying all tables

2.8 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. 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.

Supposing you wanted to modify the column ATTRACT_CAPACITY in the ATTRACTION table by changing the date characteristics from NUMERIC(3) to NUMERIC(4). You would execute the following command:

```
ALTER TABLE ATTRACTION

MODIFY ATTRACT_CAPACITY NUMERIC(4);
```

Note

Some DBMSs impose limitations on when it's possible to change attribute characteristics. 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 MySQL lab. The tables that you have created will be used in the rest of this lab guide to explore the use of SQL in MySQL 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*).

Note

In MySQL there are a number of versions of the INSERT statement. As well as the basic INSERT which inserts rows into a table, the INSERT ... VALUES and INSERT ... SET forms of the statement insert rows based on explicitly specified values. For example, the INSERT ... SELECT form inserts rows selected from another table or tables. You can read more about this in the MySQL Reference manual 5.0.

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 ];
```

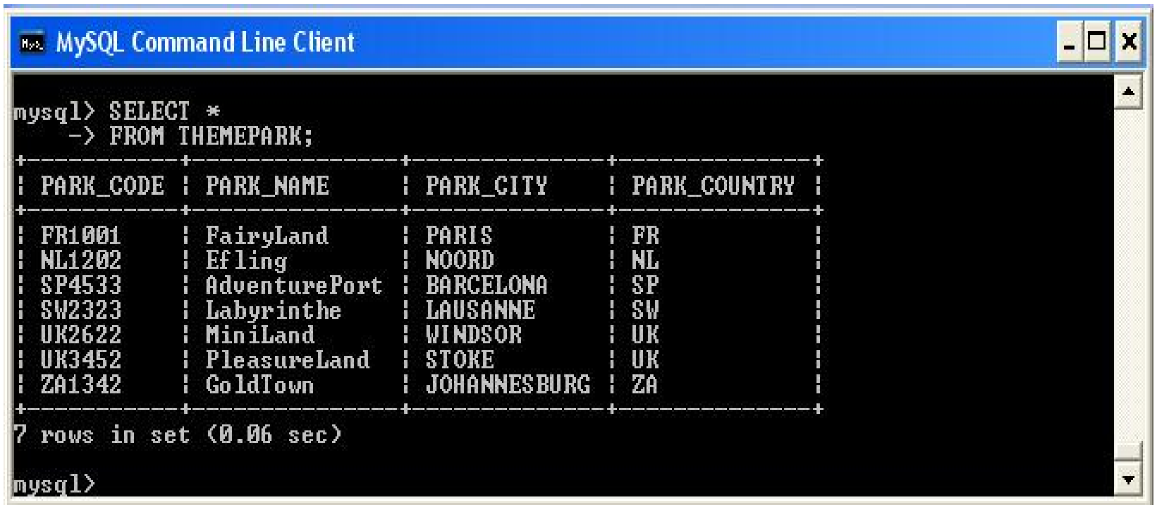
Notice that the command must finish with a semi-colon, and will be executed when the Enter key is pressed at the end of the command.

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 15.



```
mysql> 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 in set (0.06 sec)

mysql>
```

Figure 15: Displaying all columns from the THEMEPARK 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 16 and 17. In these two examples you are selecting specific columns from a single table.

Example 1

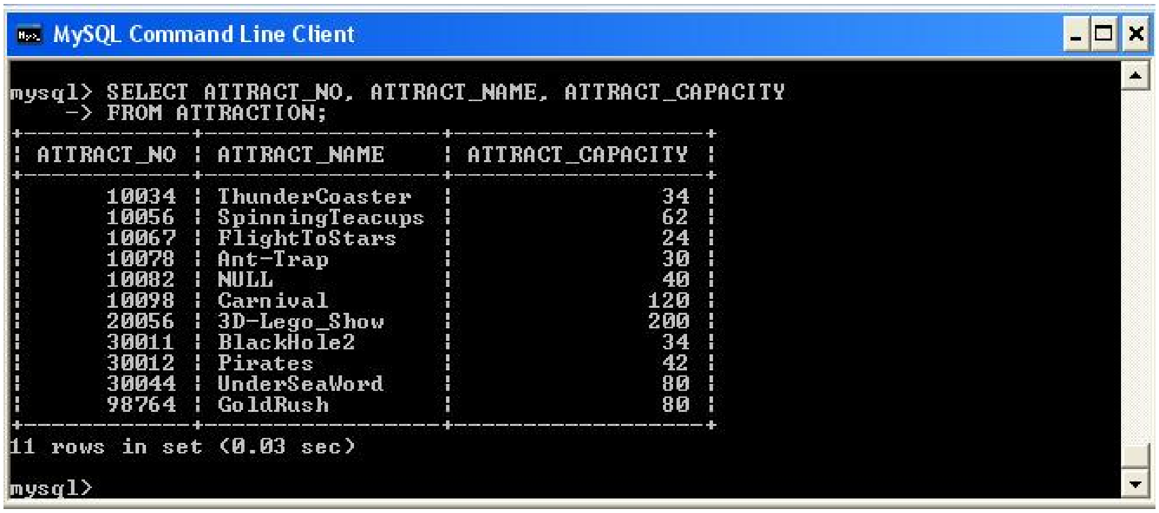
SELECT ATTRACT_NO, ATTRACT_NAME, ATTRACT_CAPACITY

FROM ATTRACTION;

Example 2

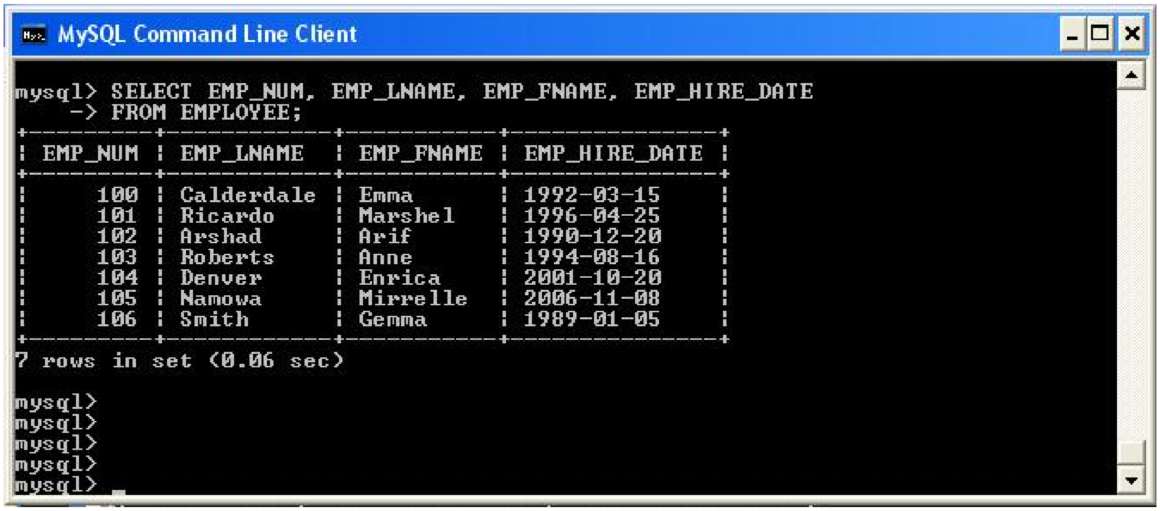
SELECT EMP_NUM, EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE

FROM EMPLOYEE;



```
mysql> SELECT ATTRACT_NO, ATTRACT_NAME, ATTRACT_CAPACITY
    -> FROM ATTRACTION;
+------------+-----------------+------------------+
| ATTRACT_NO | ATTRACT_NAME    | ATTRACT_CAPACITY |
+------------+-----------------+------------------+
|      10034 | ThunderCoaster  |               34 |
|      10056 | SpinningTeacups |               62 |
|      10067 | FlightToStars   |               24 |
|      10078 | Ant-Trap        |               30 |
|      10082 | NULL            |               40 |
|      10098 | Carnival        |              120 |
|      20056 | 3D-Lego_Show    |              200 |
|      30011 | BlackHole2      |               34 |
|      30012 | Pirates         |               42 |
|      30044 | UnderSeaWord    |               80 |
|      98764 | GoldRush        |               80 |
+------------+-----------------+------------------+
11 rows in set (0.03 sec)

mysql>
```

Figure 16: Output for Example 1



```
mysql> SELECT EMP_NUM, EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE
    -> FROM EMPLOYEE;
+---------+------------+-----------+---------------+
| EMP_NUM | EMP_LNAME  | EMP_FNAME | EMP_HIRE_DATE |
+---------+------------+-----------+---------------+
|     100 | Calderdale | Emma      | 1992-03-15    |
|     101 | Ricardo    | Marshel   | 1996-04-25    |
|     102 | Arshad     | Arif      | 1990-12-20    |
|     103 | Roberts    | Anne      | 1994-08-16    |
|     104 | Denver     | Enrica    | 2001-10-20    |
|     105 | Namowa     | Mirrelle  | 2006-11-08    |
|     106 | Smith      | Gemma     | 1989-01-05    |
+---------+------------+-----------+---------------+
7 rows in set (0.06 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
```

Figure 17: 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*

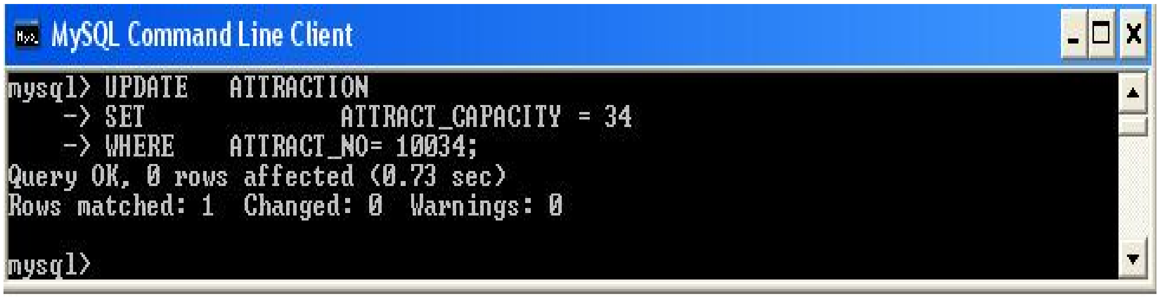
SET *columnname* = *expression* [, *columnname* = *expression*]

[WHERE *conditionlist*];

For example, if you want 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 18.



```
mysql> UPDATE   ATTRACTION
    -> SET              ATTRACT_CAPACITY = 34
    -> WHERE    ATTRACT_NO= 10034;
Query OK, 0 rows affected (0.73 sec)
Rows matched: 1  Changed: 0  Warnings: 0

mysql>
```

Figure 18: 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 a 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

Supposing you decided 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. In order to use the ROLLBACK (and COMMIT) commands in MySQL, you first need to change the value for the **AUTOCOMMIT** to 0 by typing the following command:

```
mysql> SET AUTOCOMMIT = 0;
```

This command needs only to be executed once in a session.

Task 3.7 To restore the data to their “pre-change” condition set the value, type the following commands;

```
mysql> SET AUTOCOMMIT = 0;

mysql> ROLLBACK;
```

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 that 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 EMP_COPY 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_PHONEs 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_CO	PARK_PHONE

			DE	
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:

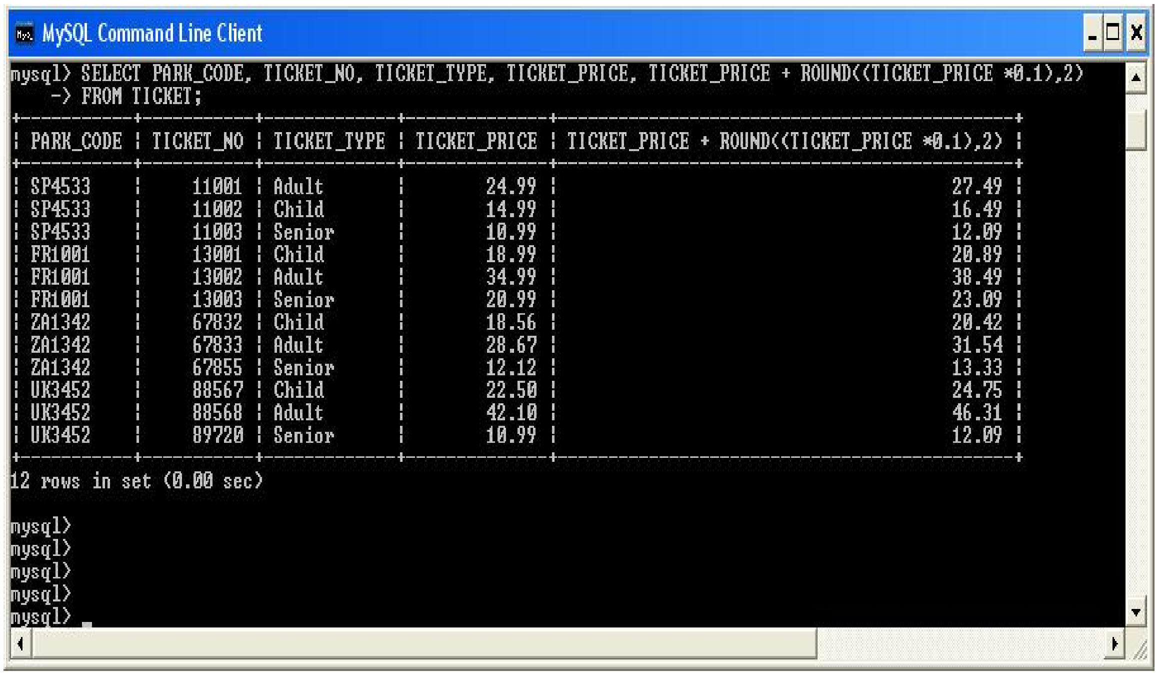
1. Perform operations within parentheses
2. Perform power operations
3. Perform multiplications and divisions
4. Perform additions and subtractions

Task 4.1 Suppose the owners of all the theme parks wanted to compare the current ticket prices, with an increase in the 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 19. The ROUND function is used to ensure the result is displayed to two decimal places.



```
mysql> SELECT PARK_CODE, TICKET_NO, TICKET_TYPE, TICKET_PRICE, TICKET_PRICE + ROUND((TICKET_PRICE *0.1),2)
    -> FROM TICKET;
+-----------+-----------+-------------+--------------+---------------------------------------------+
| PARK_CODE | TICKET_NO | TICKET_TYPE | 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       |        22.50 |                                       24.75 |
| UK3452    |     88568 | Adult       |        42.10 |                                       46.31 |
| UK3452    |     89720 | Senior      |        10.99 |                                       12.09 |
+-----------+-----------+-------------+--------------+---------------------------------------------+
12 rows in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
```

Figure 19: Output showing 10% increase in ticket prices

You will see in Figure 19 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.

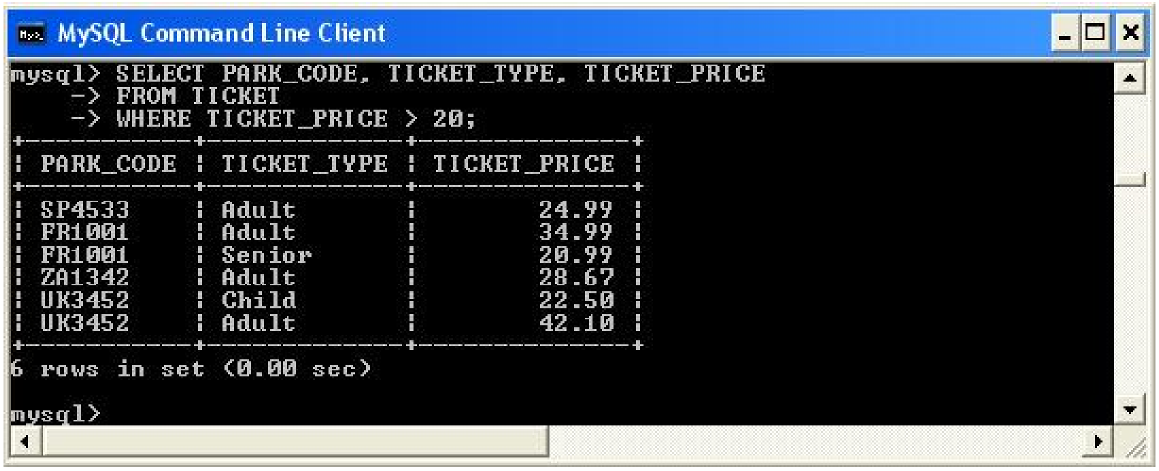
We will now explore some of these conditional operators using examples.

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 20.



```
mysql> SELECT PARK_CODE, TICKET_TYPE, TICKET_PRICE
    -> FROM TICKET
    -> WHERE TICKET_PRICE > 20;
+-----------+-------------+--------------+
| PARK_CODE | TICKET_TYPE | TICKET_PRICE |
+-----------+-------------+--------------+
| SP4533    | Adult       |        24.99 |
| FR1001    | Adult       |        34.99 |
| FR1001    | Senior      |        20.99 |
| ZA1342    | Adult       |        28.67 |
| UK3452    | Child       |        22.50 |
| UK3452    | Adult       |        42.10 |
+-----------+-------------+--------------+
6 rows in set (0.00 sec)

mysql>
```

Figure 20: 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 20?

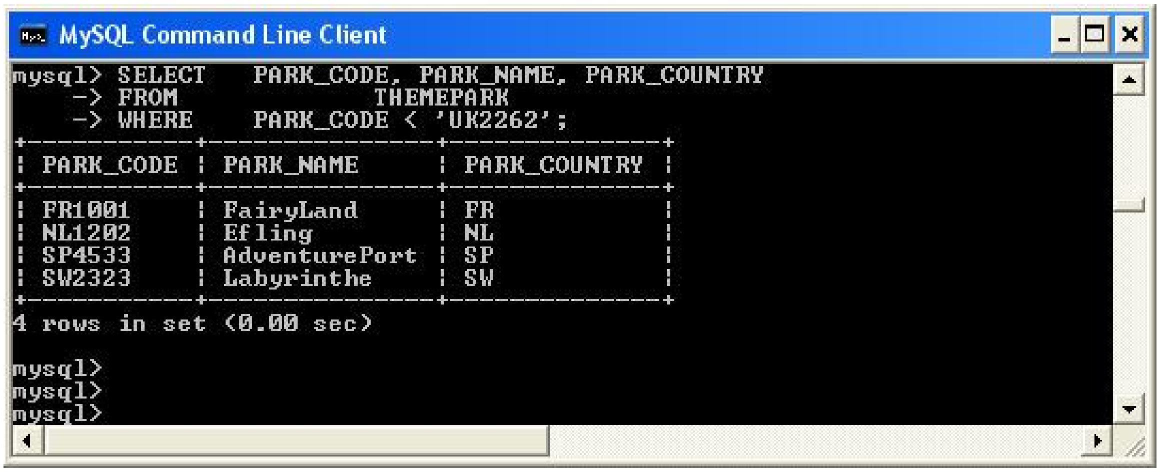
Task 4.3 Modify the query you have just executed to display tickets that are less than €30.00.

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*.) Therefore, the output will be generated as shown in Figure 21.

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



```
mysql> SELECT    PARK_CODE, PARK_NAME, PARK_COUNTRY
    -> FROM              THEMEPARK
    -> WHERE     PARK_CODE < 'UK2262';
+-----------+---------------+--------------+
| PARK_CODE | PARK_NAME     | PARK_COUNTRY |
+-----------+---------------+--------------+
| FR1001    | FairyLand     | FR           |
| NL1202    | Efling        | NL           |
| SP4533    | AdventurePort | SP           |
| SW2323    | Labyrinthe    | SW           |
+-----------+---------------+--------------+
4 rows in set (0.00 sec)

mysql>
mysql>
mysql>
```

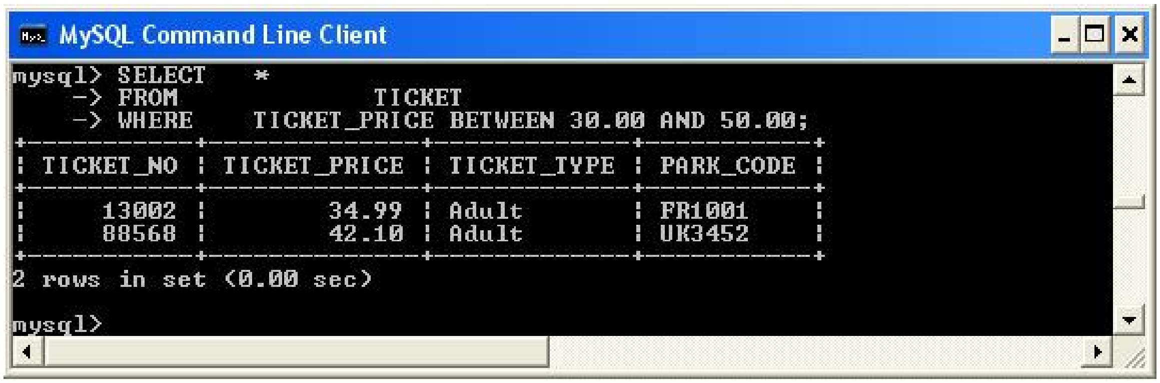
Figure 21: Example of character comparison

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;
```

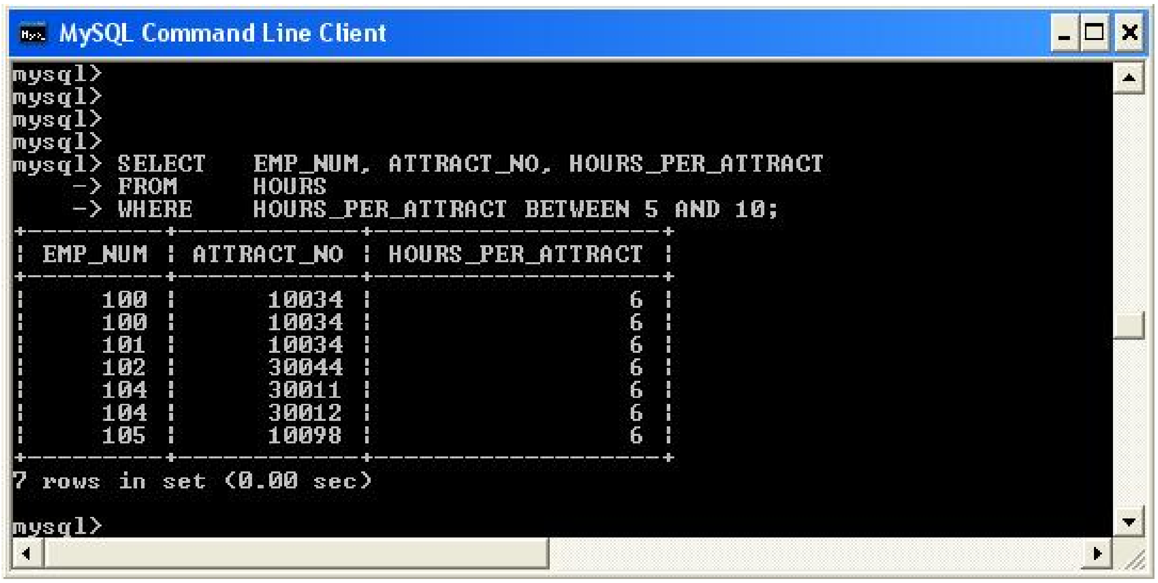
Figure 22 shows the output you should see for this query.



```
mysql> SELECT   *
    -> FROM             TICKET
    -> WHERE    TICKET_PRICE BETWEEN 30.00 AND 50.00;
+-----------+--------------+-------------+-----------+
| TICKET_NO | TICKET_PRICE | TICKET_TYPE | PARK_CODE |
+-----------+--------------+-------------+-----------+
|     13002 |        34.99 | Adult       | FR1001    |
|     88568 |        42.10 | Adult       | UK3452    |
+-----------+--------------+-------------+-----------+
2 rows in set (0.00 sec)

mysql>
```

Figure 22: 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 23.



```
mysql>
mysql>
mysql>
mysql>
mysql> SELECT   EMP_NUM, ATTRACT_NO, HOURS_PER_ATTRACT
    -> FROM     HOURS
    -> WHERE    HOURS_PER_ATTRACT BETWEEN 5 AND 10;
+---------+------------+-------------------+
| 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 in set (0.00 sec)

mysql>
```

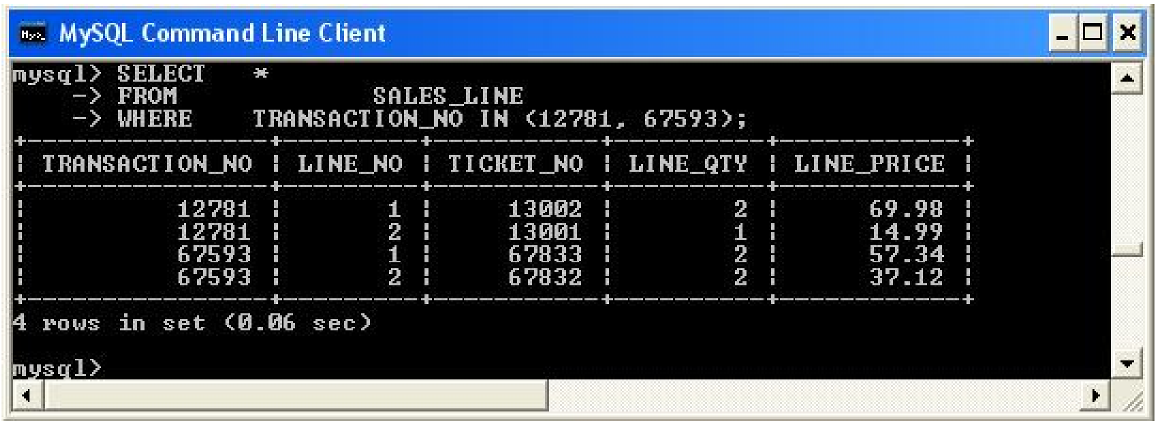
Figure 23: Output for Task 4.5

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);
```

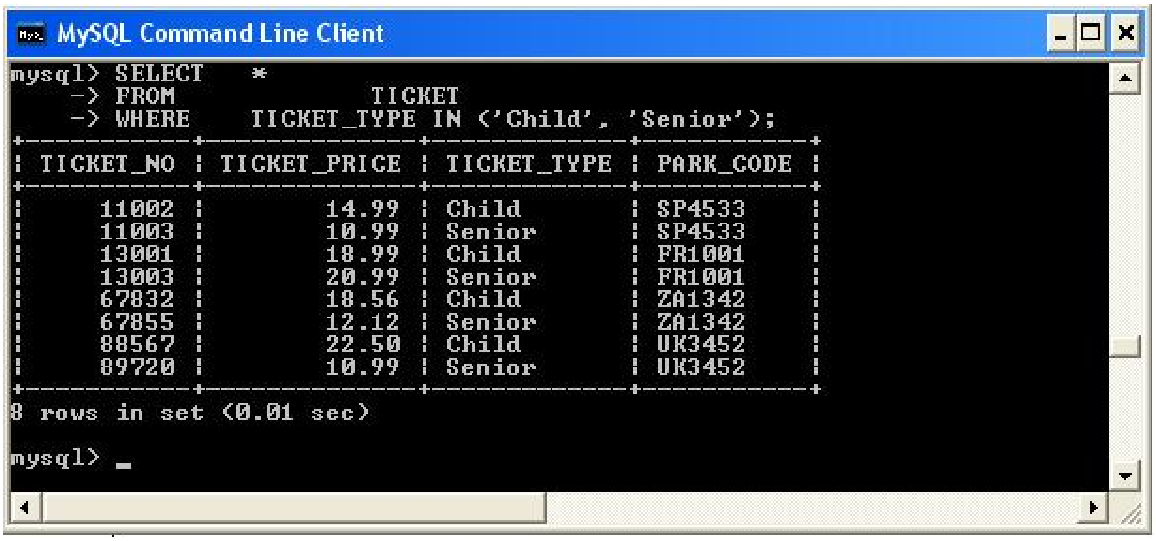
The result of this query is shown in Figure 24.



```
mysql> SELECT   *
    -> FROM             SALES_LINE
    -> 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 |
+----------------+---------+-----------+----------+------------+
4 rows in set (0.06 sec)

mysql>
```

Figure 24 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 25.



```
mysql> SELECT   *
    -> FROM             TICKET
    -> WHERE    TICKET_TYPE IN ('Child', 'Senior');
+-----------+--------------+-------------+-----------+
| TICKET_NO | TICKET_PRICE | TICKET_TYPE | 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 |        22.50 | Child       | UK3452    |
|     89720 |        10.99 | Senior      | UK3452    |
+-----------+--------------+-------------+-----------+
8 rows in set (0.01 sec)

mysql> _
```

Figure 25. Output for Task 4.6

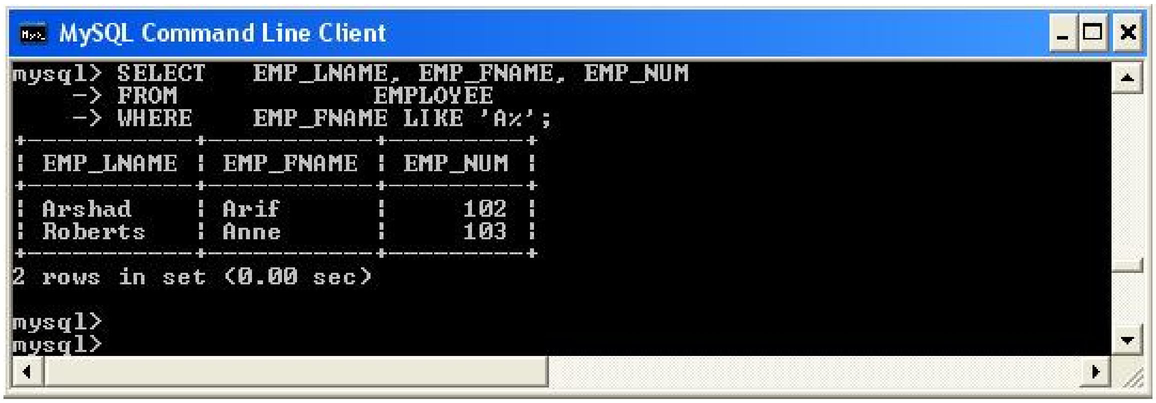
LIKE

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%';
```

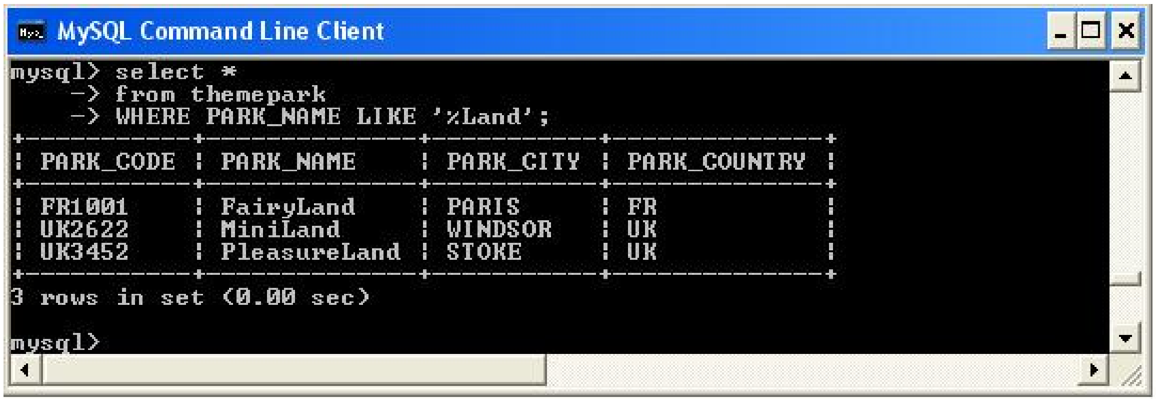
Figure 26 shows the output you should see for this query.



```
mysql> SELECT    EMP_LNAME, EMP_FNAME, EMP_NUM
    -> FROM              EMPLOYEE
    -> WHERE     EMP_FNAME LIKE 'A%';
+-----------+-----------+---------+
| EMP_LNAME | EMP_FNAME | EMP_NUM |
+-----------+-----------+---------+
| Arshad    | Arif      |     102 |
| Roberts   | Anne      |     103 |
+-----------+-----------+---------+
2 rows in set (0.00 sec)

mysql>
mysql>
```

Figure 26 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 27.



```
mysql> select *
    -> from themepark
    -> 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           |
+-----------+--------------+------------+--------------+
3 rows in set (0.00 sec)

mysql>
```

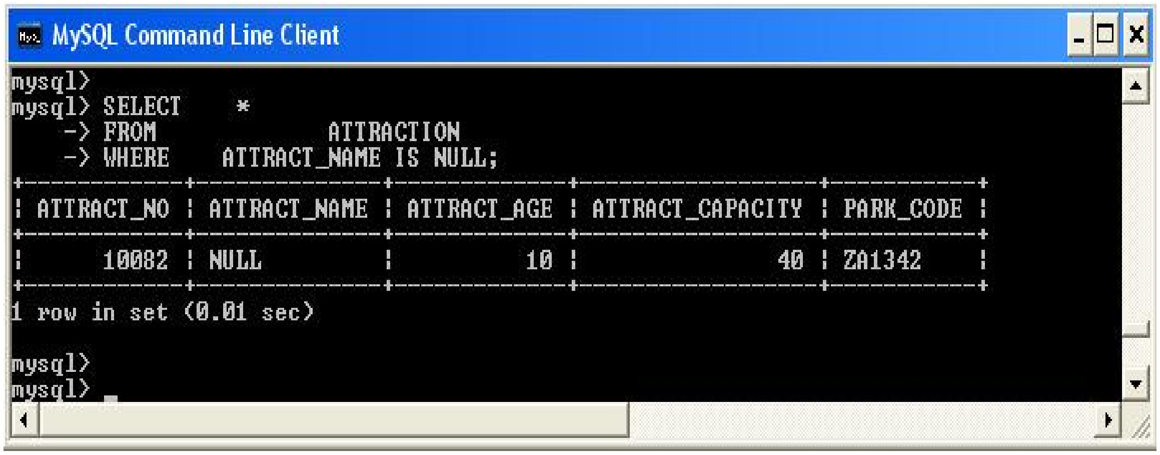
Figure 27 Solution to Task 4.8

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 28.



```
mysql>
mysql> SELECT    *
    -> FROM          ATTRACTION
    -> WHERE    ATTRACT_NAME IS NULL;
+------------+--------------+-------------+------------------+-----------+
| ATTRACT_NO | ATTRACT_NAME | ATTRACT_AGE | ATTRACT_CAPACITY | PARK_CODE |
+------------+--------------+-------------+------------------+-----------+
|      10082 | NULL         |          10 |               40 | ZA1342    |
+------------+--------------+-------------+------------------+-----------+
1 row in set (0.01 sec)

mysql>
mysql> _
```

Figure 28 Listing all Attractions with no name

Logical Operators

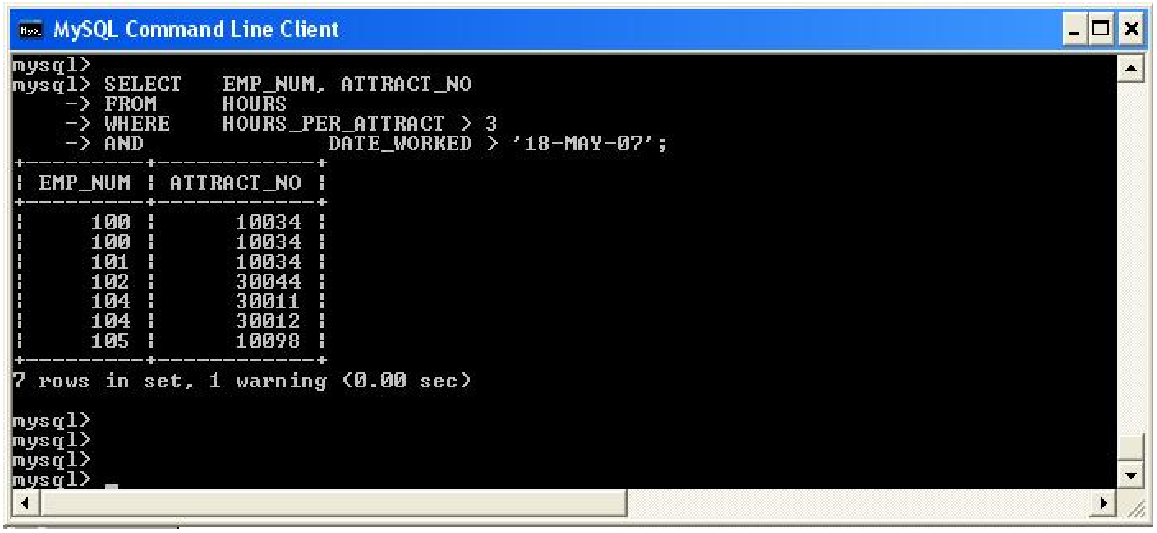
SQL allows you to have multiple conditions in a query through the use of logical operators: AND, OR and NOT. NOT has the highest precedence, followed by AND, and then followed by OR. However, you are strongly recommended to use parentheses to clarify the intended meaning of the query.

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 29.

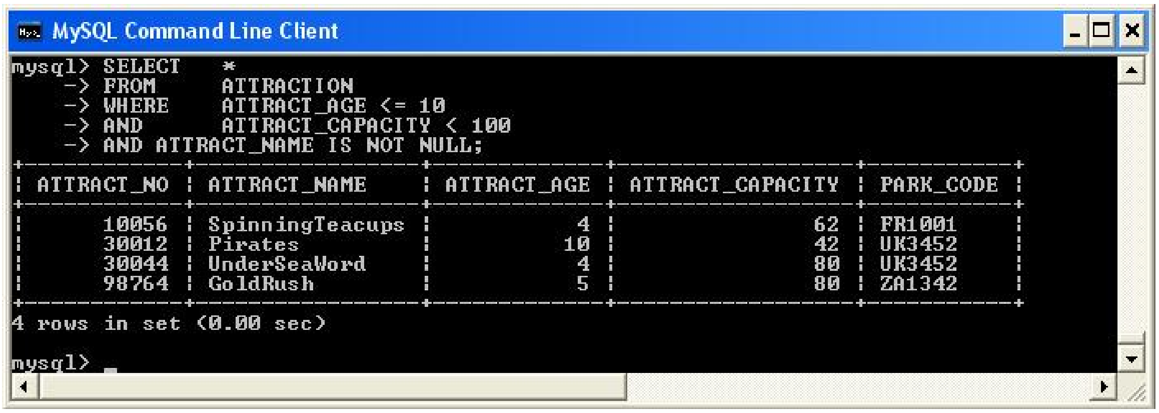


```
mysql>
mysql> SELECT   EMP_NUM, ATTRACT_NO
    -> FROM     HOURS
    -> WHERE    HOURS_PER_ATTRACT > 3
    -> AND              DATE_WORKED > '18-MAY-07';
+---------+------------+
| EMP_NUM | ATTRACT_NO |
+---------+------------+
|     100 |      10034 |
|     100 |      10034 |
|     101 |      10034 |
|     102 |      30044 |
|     104 |      30011 |
|     104 |      30012 |
|     105 |      10098 |
+---------+------------+
7 rows in set, 1 warning (0.00 sec)

mysql>
mysql>
mysql>
mysql> _
```

Figure 29 Query results using the AND operator

Task 4.9 Enter the query above and check you results with those shown in Figure 29.

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 30.



```
mysql> SELECT   *
    -> FROM     ATTRACTION
    -> WHERE    ATTRACT_AGE <= 10
    -> AND      ATTRACT_CAPACITY < 100
    -> AND ATTRACT_NAME IS NOT NULL;
+------------+-----------------+-------------+------------------+-----------+
| 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    |
+------------+-----------------+-------------+------------------+-----------+
4 rows in set (0.00 sec)

mysql> _
```

Figure 30: Query results for Task 4.10

OR

If you wanted to list the names and countries of all Theme parks where of invoice numbers where PARK_COUNTRY = 'FR' OR PARK_COUNTRY = 'UK' you would write the following query.

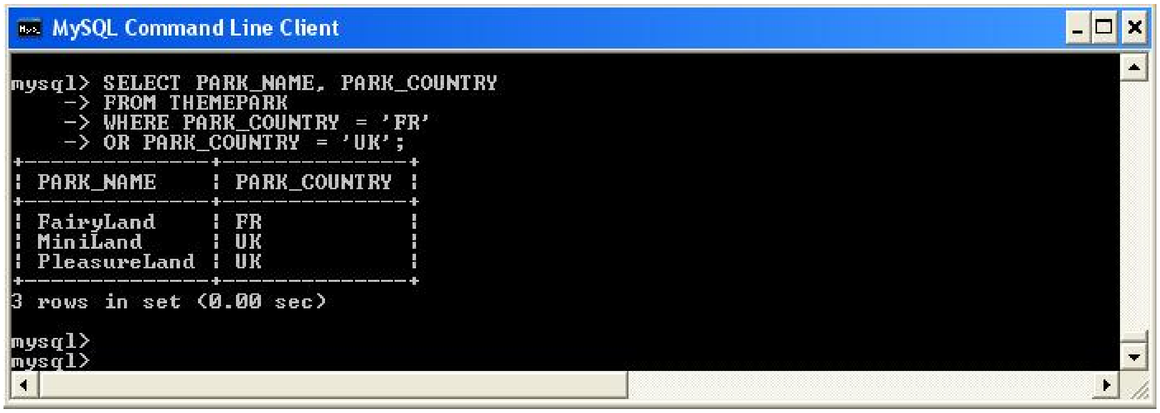
SELECT PARK_NAME, PARK_COUNTRY

FROM THEMEPARK

WHERE PARK_COUNTRY = 'FR'

OR PARK_COUNTRY = 'UK';

The output is shown in Figure 31.



```
mysql> SELECT PARK_NAME, PARK_COUNTRY
    -> FROM THEMEPARK
    -> WHERE PARK_COUNTRY = 'FR'
    -> OR PARK_COUNTRY = 'UK';
+--------------+--------------+
| PARK_NAME    | PARK_COUNTRY |
+--------------+--------------+
| FairyLand    | FR           |
| MiniLand     | UK           |
| PleasureLand | UK           |
+--------------+--------------+
3 rows in set (0.00 sec)

mysql>
mysql>
```

Figure 31: 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.

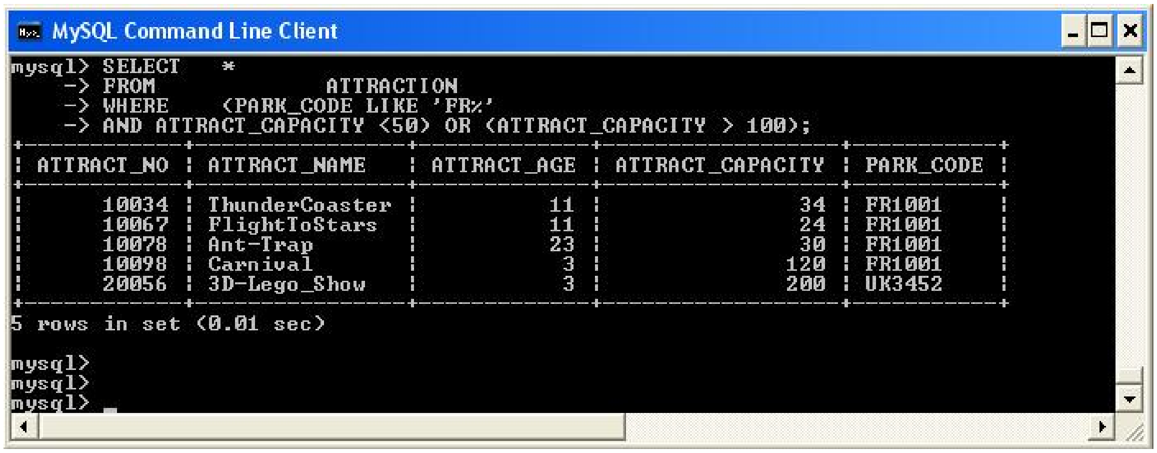
Task 4.11 Test the following query and check your output with that shown in Figure 32. Can you work out what this query is doing?

SELECT *

FROM ATTRACTION

WHERE (PARK_CODE LIKE ‘FR%’

AND ATTRACT_CAPACITY <50) OR (ATTRACT_CAPACITY > 100);



```
mysql> SELECT   *
    -> FROM            ATTRACTION
    -> WHERE    (PARK_CODE LIKE 'FR%'
    -> 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    |
+------------+----------------+-------------+------------------+-----------+
5 rows in set (0.01 sec)

mysql>
mysql>
mysql> _
```

Figure 32: AND and OR example

NOT

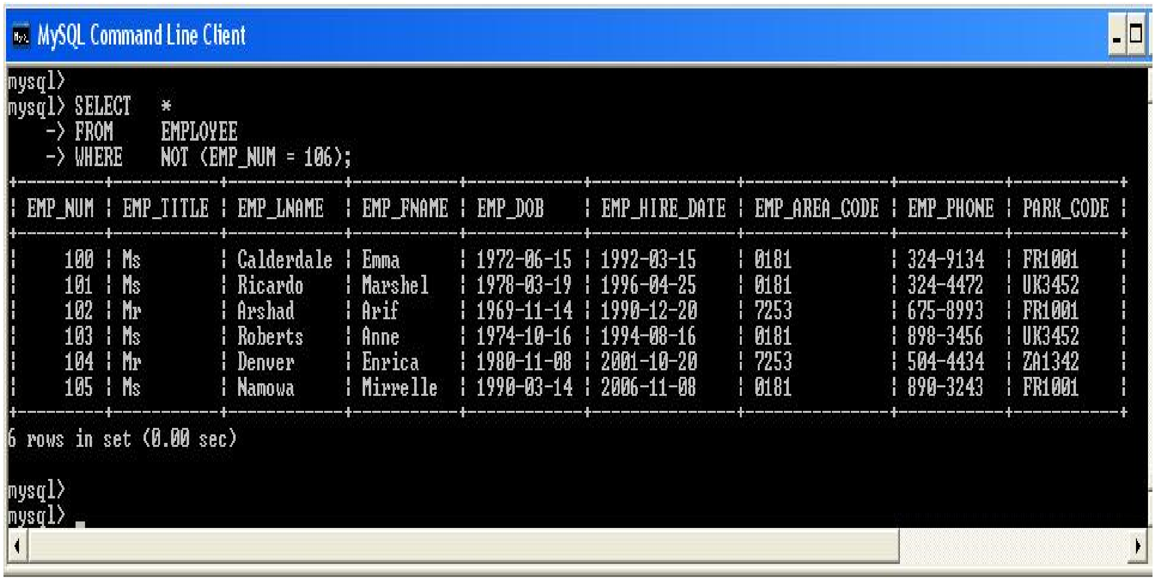
The logical operator **NOT** is used to negate the result of a conditional expression. If you want to see a listing of all rows for which EMP_NUM is not 106, the query would look like:

SELECT *

FROM EMPLOYEE

WHERE NOT (EMP_NUM = 106);

The results of this query are shown in Figure 33. Note that the condition is enclosed in parentheses; that practice is optional, but it is highly recommended for clarity.



```
mysql>
mysql> SELECT   *
    -> FROM     EMPLOYEE
    -> WHERE    NOT (EMP_NUM = 106);
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
| EMP_NUM | EMP_TITLE | EMP_LNAME  | EMP_FNAME | EMP_DOB    | EMP_HIRE_DATE | EMP_AREA_CODE | EMP_PHONE | PARK_CODE |
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
|     100 | Ms        | Calderdale | Emma      | 1972-06-15 | 1992-03-15    | 0181          | 324-9134  | FR1001    |
|     101 | Ms        | Ricardo    | Marshel   | 1978-03-19 | 1996-04-25    | 0181          | 324-4472  | UK3452    |
|     102 | Mr        | Arshad     | Arif      | 1969-11-14 | 1990-12-20    | 7253          | 675-8993  | FR1001    |
|     103 | Ms        | Roberts    | Anne      | 1974-10-16 | 1994-08-16    | 0181          | 898-3456  | UK3452    |
|     104 | Mr        | Denver     | Enrica    | 1980-11-08 | 2001-10-20    | 7253          | 504-4434  | ZA1342    |
|     105 | Ms        | Namowa     | Mirrelle  | 1990-03-14 | 2006-11-08    | 0181          | 890-3243  | FR1001    |
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
6 rows in set (0.00 sec)

mysql>
mysql> _
```

Figure 33: 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 18th 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 than 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

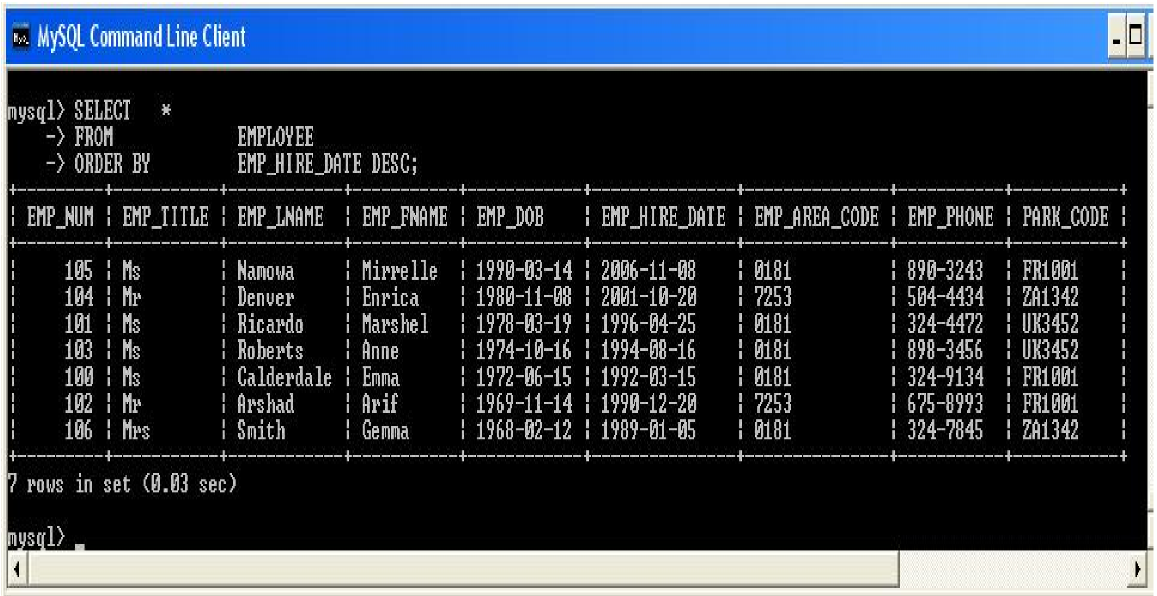
The learning objectives of this lab are to

- 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 34.

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



```
mysql> SELECT   *
    -> FROM          EMPLOYEE
    -> ORDER BY      EMP_HIRE_DATE DESC;
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
| EMP_NUM | EMP_TITLE | EMP_LNAME  | EMP_FNAME | EMP_DOB    | EMP_HIRE_DATE | EMP_AREA_CODE | EMP_PHONE | PARK_CODE |
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
|     105 | Ms        | Namowa     | Mirrelle  | 1990-03-14 | 2006-11-08    | 0181          | 890-3243  | FR1001    |
|     104 | Mr        | Denver     | Enrica    | 1980-11-08 | 2001-10-20    | 7253          | 504-4434  | ZA1342    |
|     101 | Ms        | Ricardo    | Marshel   | 1978-03-19 | 1996-04-25    | 0181          | 324-4472  | UK3452    |
|     103 | Ms        | Roberts    | Anne      | 1974-10-16 | 1994-08-16    | 0181          | 898-3456  | UK3452    |
|     100 | Ms        | Calderdale | Emma      | 1972-06-15 | 1992-03-15    | 0181          | 324-9134  | FR1001    |
|     102 | Mr        | Arshad     | Arif      | 1969-11-14 | 1990-12-20    | 7253          | 675-8993  | FR1001    |
|     106 | Mrs       | Smith      | Gemma     | 1968-02-12 | 1989-01-05    | 0181          | 324-7845  | ZA1342    |
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
7 rows in set (0.03 sec)

mysql> _
```

Figure 34: 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.

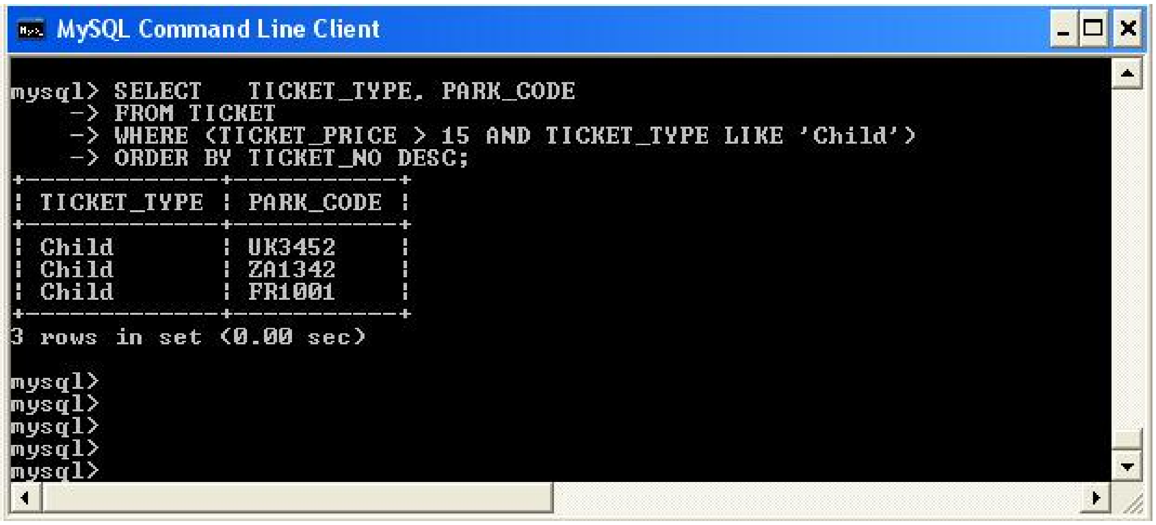
Task 5.1 Enter the following query which contains an example of a cascading order sequence, by ordering the rows in the employee table by the employee's last then first names.

```
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 35. 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;
```



```
mysql> SELECT   TICKET_TYPE, PARK_CODE
    -> FROM TICKET
    -> WHERE (TICKET_PRICE > 15 AND TICKET_TYPE LIKE 'Child')
    -> ORDER BY TICKET_NO DESC;
+-------------+-----------+
| TICKET_TYPE | PARK_CODE |
+-------------+-----------+
| Child       | UK3452    |
| Child       | ZA1342    |
| Child       | FR1001    |
+-------------+-----------+
3 rows in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
```

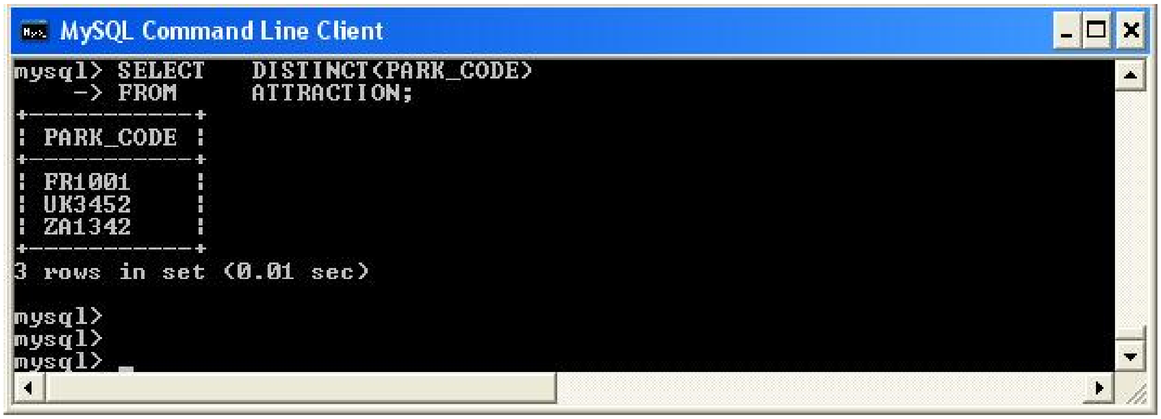
Figure 35: Query results for Task 5.2.

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 36 shows that the query only displays the rows that are different.



```
mysql> SELECT    DISTINCT(PARK_CODE)
    -> FROM      ATTRACTION;
+-----------+
| PARK_CODE |
+-----------+
| FR1001    |
| UK3452    |
| ZA1342    |
+-----------+
3 rows in set (0.01 sec)

mysql>
mysql>
mysql> _
```

Figure 36: 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**

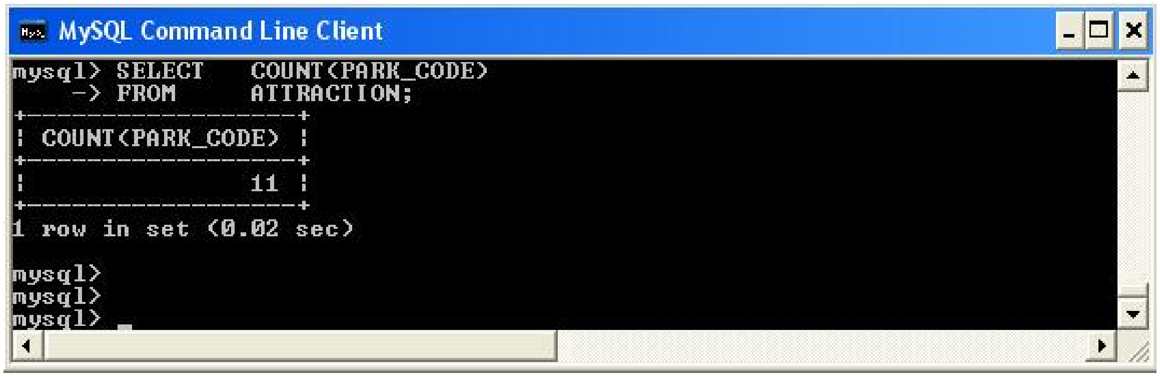
COUNT	The number of rows containing non-null values
MIN	The minimum attribute value encountered in a given column
MAX	The maximum attribute value encountered in a given column
SUM	The sum of all values for a given column
AVG	The arithmetic mean (average) for a specified column

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 37.



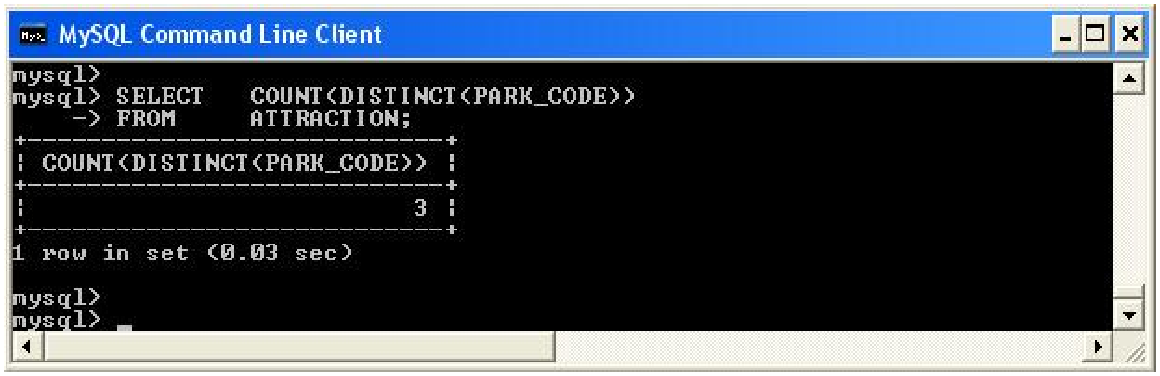
```
mysql> SELECT    COUNT(PARK_CODE)
    -> FROM      ATTRACTION;
+------------------+
| COUNT(PARK_CODE) |
+------------------+
|               11 |
+------------------+
1 row in set (0.02 sec)

mysql>
mysql>
mysql> _
```

Figure 37: 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 38):

SELECT COUNT(DISTINCT(PARK_CODE))

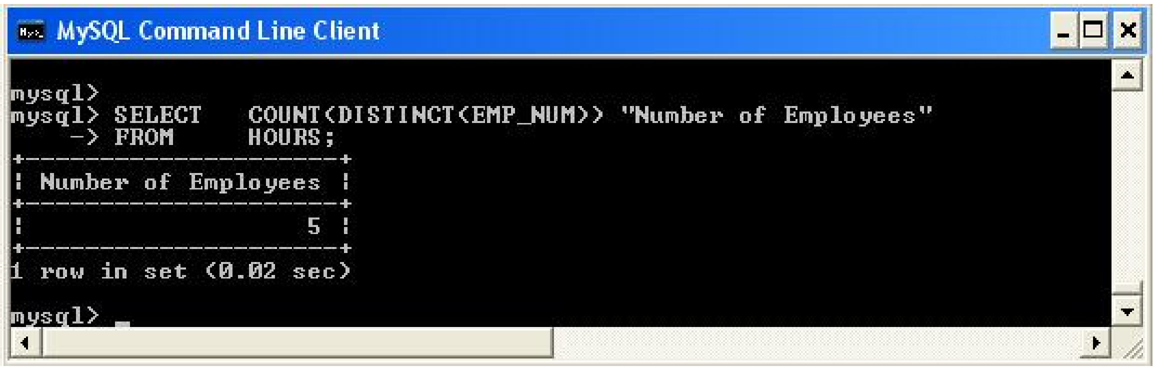
FROM ATTRACTION;



```
mysql>
mysql> SELECT    COUNT(DISTINCT(PARK_CODE))
    -> FROM      ATTRACTION;
+----------------------------+
| COUNT(DISTINCT(PARK_CODE)) |
+----------------------------+
|                          3 |
+----------------------------+
1 row in set (0.03 sec)

mysql>
mysql> _
```

Figure 38: 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 39.



```
mysql>
mysql> SELECT    COUNT(DISTINCT(EMP_NUM)) "Number of Employees"
    -> FROM      HOURS;
+---------------------+
| Number of Employees |
+---------------------+
|                   5 |
+---------------------+
1 row in set (0.02 sec)

mysql> _
```

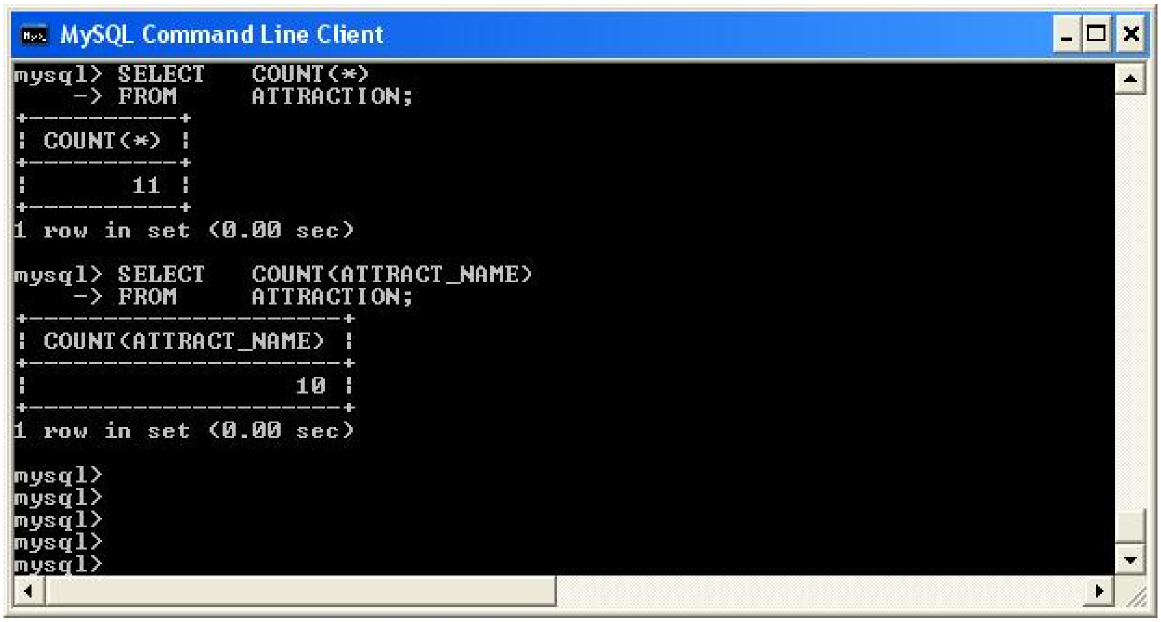
Figure 39: Query output for Task 5.3

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 40. Can you explain why the number of rows returned is different?

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

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



```
mysql> SELECT   COUNT(*)
    -> FROM     ATTRACTION;
+----------+
| COUNT(*) |
+----------+
|       11 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT   COUNT(ATTRACT_NAME)
    -> FROM     ATTRACTION;
+---------------------+
| COUNT(ATTRACT_NAME) |
+---------------------+
|                  10 |
+---------------------+
1 row in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
```

Figure 40: Examples of using the COUNT function

MAX and MIN

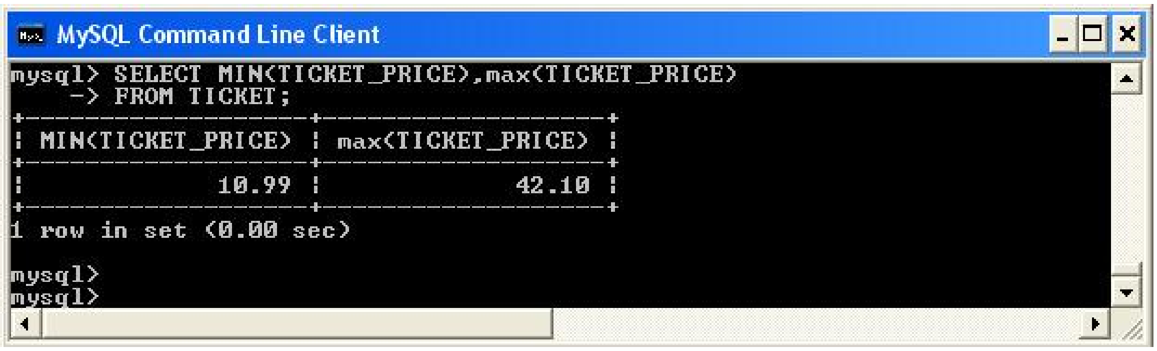
The MAX and MIN functions are used to find answers to problems such as

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 41.

SELECT MIN(TICKET_PRICE),max(TICKET_PRICE)

FROM TICKET;



```
mysql> SELECT MIN(TICKET_PRICE),max(TICKET_PRICE)
    -> FROM TICKET;
+-------------------+-------------------+
| MIN(TICKET_PRICE) | max(TICKET_PRICE) |
+-------------------+-------------------+
|             10.99 |             42.10 |
+-------------------+-------------------+
1 row in set (0.00 sec)

mysql>
mysql>
```

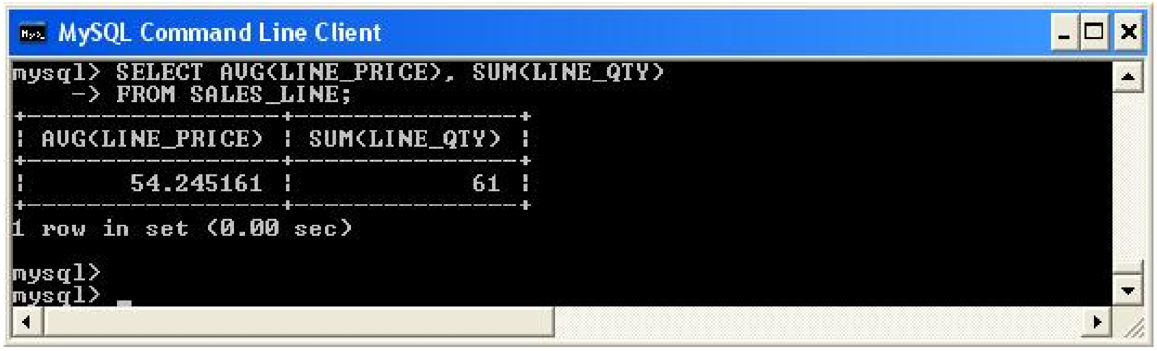
Figure 41: Examples of using the MIN and MAX functions

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 42 shows the output for this query.

SELECT AVG(LINE_PRICE), SUM(LINE_QTY)

FROM SALES_LINE;



```
mysql> SELECT AVG(LINE_PRICE), SUM(LINE_QTY)
    -> FROM SALES_LINE;
+-----------------+---------------+
| AVG(LINE_PRICE) | SUM(LINE_QTY) |
+-----------------+---------------+
|       54.245161 |            61 |
+-----------------+---------------+
1 row in set (0.00 sec)

mysql>
mysql> _
```

Figure 42: Example showing the AVG and SUM functions

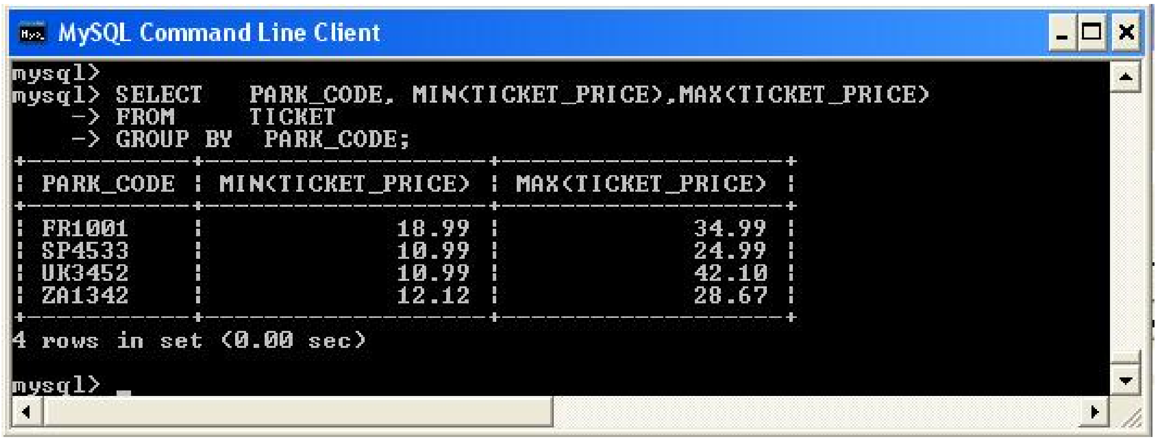
Task 5.6 Write a query that displays the average hourly rate that has been paid to all employees. Hint use the HOURS table. Your query should return €7.03.

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 43. 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;
```



```
mysql>
mysql> SELECT   PARK_CODE, MIN(TICKET_PRICE),MAX(TICKET_PRICE)
    -> FROM     TICKET
    -> GROUP BY  PARK_CODE;
+-----------+-------------------+-------------------+
| PARK_CODE | MIN(TICKET_PRICE) | MAX(TICKET_PRICE) |
+-----------+-------------------+-------------------+
| FR1001    |             18.99 |             34.99 |
| SP4533    |             10.99 |             24.99 |
| UK3452    |             10.99 |             42.10 |
| ZA1342    |             12.12 |             28.67 |
+-----------+-------------------+-------------------+
4 rows in set (0.00 sec)

mysql> _
```

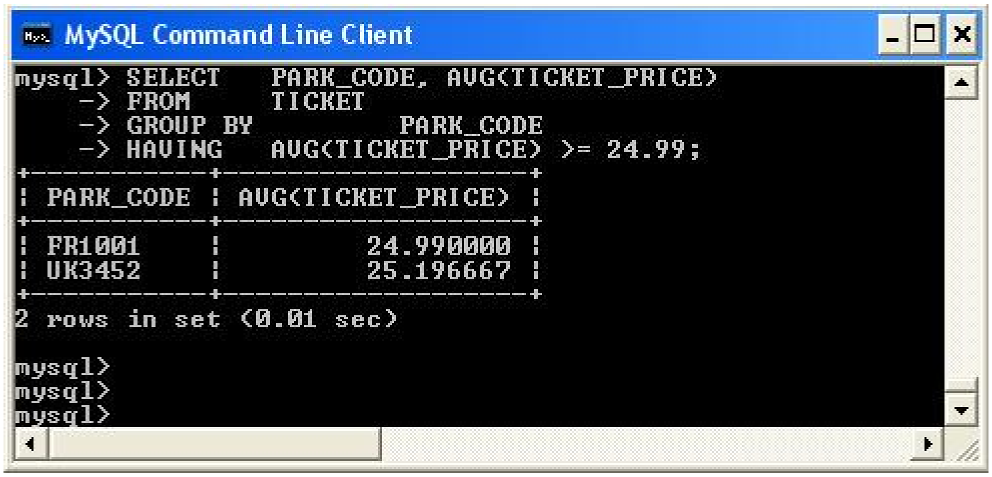
Figure 43: Displaying minimum and maximum ticket prices for each PARK_CODE

Task 5.7 Enter the query above and check the results against the output shown in Figure 43. 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 44.

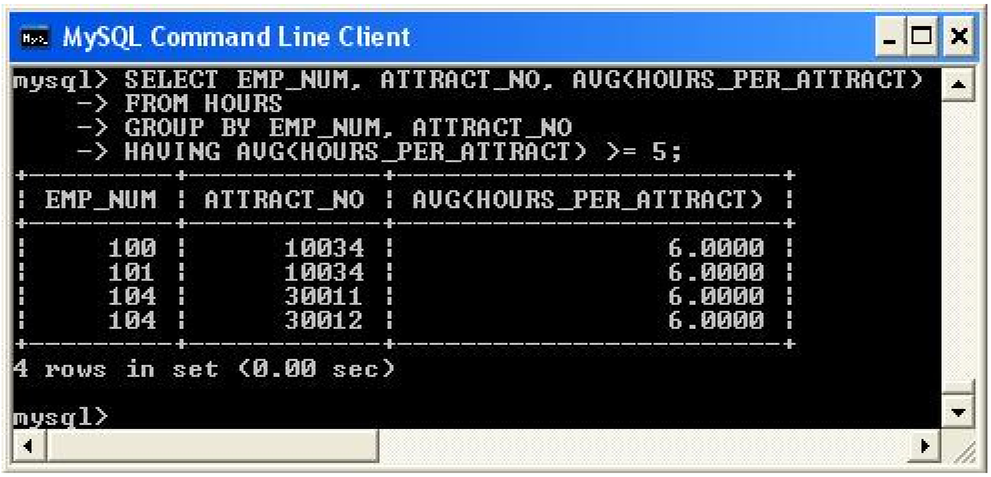
```
SELECT      PARK_CODE, AVG(TICKET_PRICE)
FROM        TICKET
GROUP BY    PARK_CODE
HAVING      AVG(TICKET_PRICE) >= 24.99;
```



```
mysql> SELECT   PARK_CODE, AVG(TICKET_PRICE)
    -> FROM     TICKET
    -> GROUP BY         PARK_CODE
    -> HAVING   AVG(TICKET_PRICE) >= 24.99;
+-----------+-------------------+
| PARK_CODE | AVG(TICKET_PRICE) |
+-----------+-------------------+
| FR1001    |         24.990000 |
| UK3452    |         25.196667 |
+-----------+-------------------+
2 rows in set (0.01 sec)

mysql>
mysql>
mysql>
```

Figure 44: 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 45.



```
mysql> SELECT EMP_NUM, ATTRACT_NO, AVG(HOURS_PER_ATTRACT)
    -> FROM HOURS
    -> GROUP BY EMP_NUM, ATTRACT_NO
    -> HAVING AVG(HOURS_PER_ATTRACT) >= 5;
+---------+------------+------------------------+
| EMP_NUM | ATTRACT_NO | AVG(HOURS_PER_ATTRACT) |
+---------+------------+------------------------+
|     100 |      10034 |                 6.0000 |
|     101 |      10034 |                 6.0000 |
|     104 |      30011 |                 6.0000 |
|     104 |      30012 |                 6.0000 |
+---------+------------+------------------------+
4 rows in set (0.00 sec)

mysql>
```

Figure 45: 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

Note

In MySQL, the CROSS JOIN command is a syntactically equivalent to INNER JOIN (they can replace each other). In standard SQL, they are not equivalent. INNER JOIN is used with an ON clause, CROSS JOIN is used otherwise. For more information, see the MySQL Reference Manual 5.0

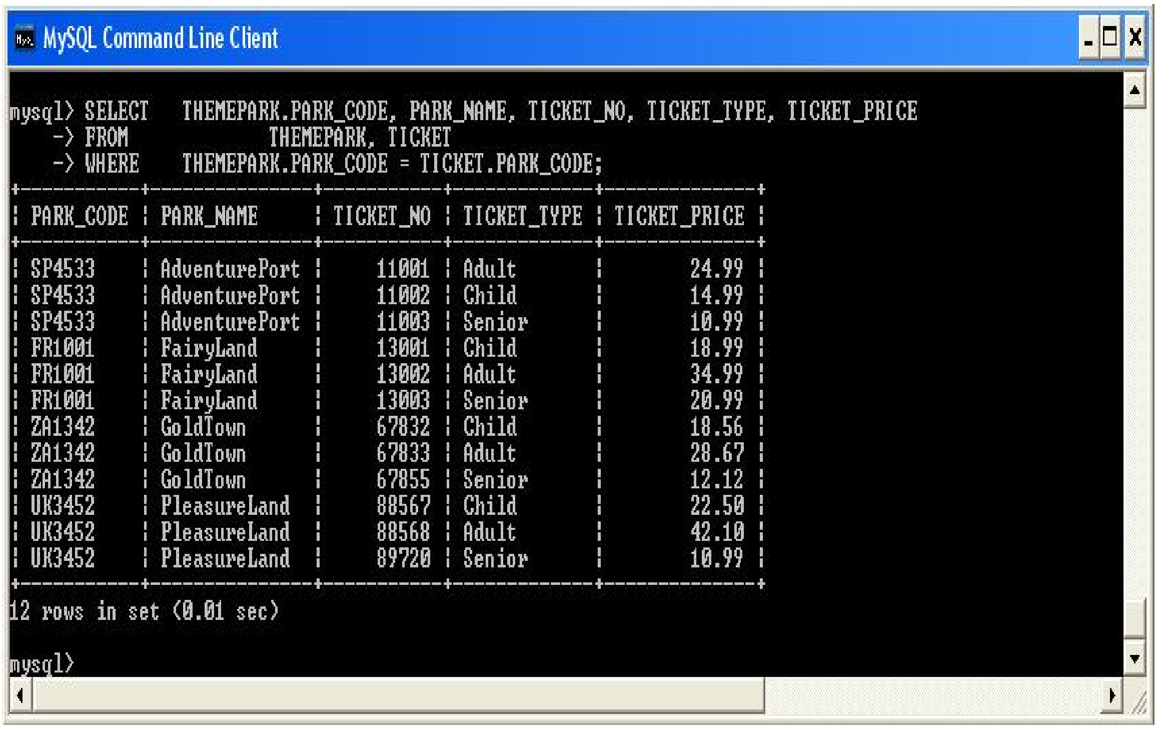
6.1 Introduction to Joins

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 46.

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

FROM        THEMEPARK, TICKET

WHERE       THEMEPARK.PARK_CODE = TICKET.PARK_CODE;
```



```
mysql> SELECT   THEMEPARK.PARK_CODE, PARK_NAME, TICKET_NO, TICKET_TYPE, TICKET_PRICE
    -> FROM          THEMEPARK, TICKET
    -> WHERE    THEMEPARK.PARK_CODE = TICKET.PARK_CODE;
+-----------+--------------+-----------+-------------+--------------+
| PARK_CODE | PARK_NAME    | TICKET_NO | TICKET_TYPE | TICKET_PRICE |
+-----------+--------------+-----------+-------------+--------------+
| SP4533    | AdventurePort |     11001 | Adult       |        24.99 |
| SP4533    | AdventurePort |     11002 | Child       |        14.99 |
| SP4533    | AdventurePort |     11003 | Senior      |        10.99 |
| FR1001    | FairyLand    |     13001 | Child       |        18.99 |
| FR1001    | FairyLand    |     13002 | Adult       |        34.99 |
| FR1001    | FairyLand    |     13003 | Senior      |        20.99 |
| ZA1342    | GoldTown     |     67832 | Child       |        18.56 |
| ZA1342    | GoldTown     |     67833 | Adult       |        28.67 |
| ZA1342    | GoldTown     |     67855 | Senior      |        12.12 |
| UK3452    | PleasureLand |     88567 | Child       |        22.50 |
| UK3452    | PleasureLand |     88568 | Adult       |        42.10 |
| UK3452    | PleasureLand |     89720 | Senior      |        10.99 |
+-----------+--------------+-----------+-------------+--------------+
12 rows in set (0.01 sec)

mysql>
```

Figure 46: 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.

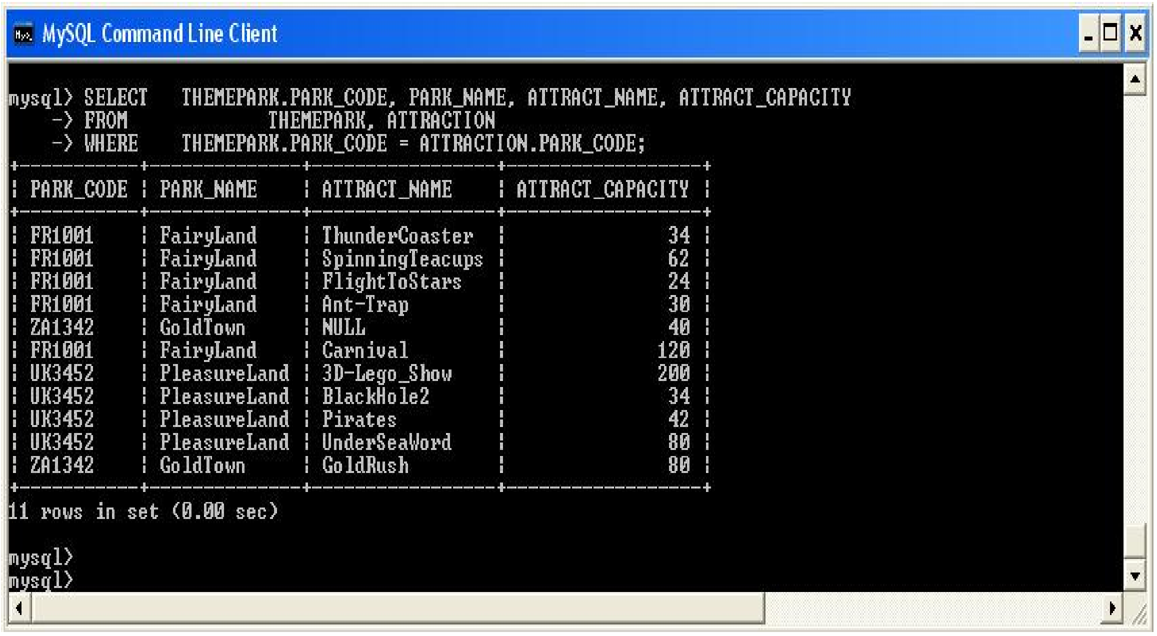
- The join condition in the WHERE clause tells the SELECT statement which rows will be returned. In this case, the SELECT statement returns all rows for which the PARK_CODE values in the PRODUCT and VENDOR tables are equal.

- 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 your results with those shown in Figure 47. 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;
```



```
mysql> SELECT   THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME, ATTRACT_CAPACITY
    -> FROM             THEMEPARK, ATTRACTION
    -> 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 |
| ZA1342    | GoldTown     | NULL            |               40 |
| FR1001    | FairyLand    | Carnival        |              120 |
| UK3452    | PleasureLand | 3D-Lego_Show    |              200 |
| UK3452    | PleasureLand | BlackHole2      |               34 |
| UK3452    | PleasureLand | Pirates         |               42 |
| UK3452    | PleasureLand | UnderSeaWord    |               80 |
| ZA1342    | GoldTown     | GoldRush        |               80 |
+-----------+--------------+-----------------+------------------+
11 rows in set (0.00 sec)

mysql>
mysql>
```

Figure 47: 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 46). 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*

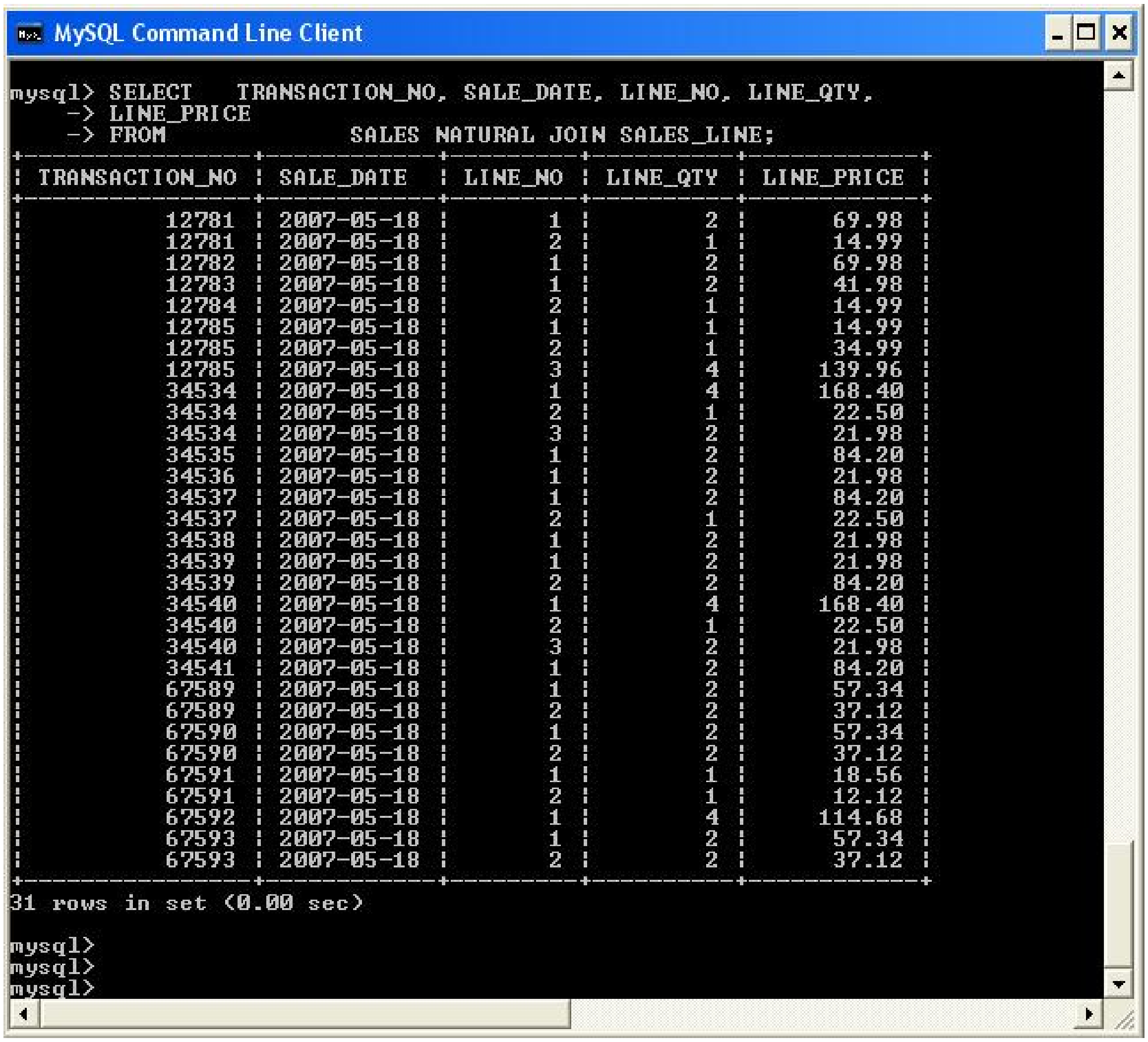
The natural join will perform the following tasks:

- 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 48.



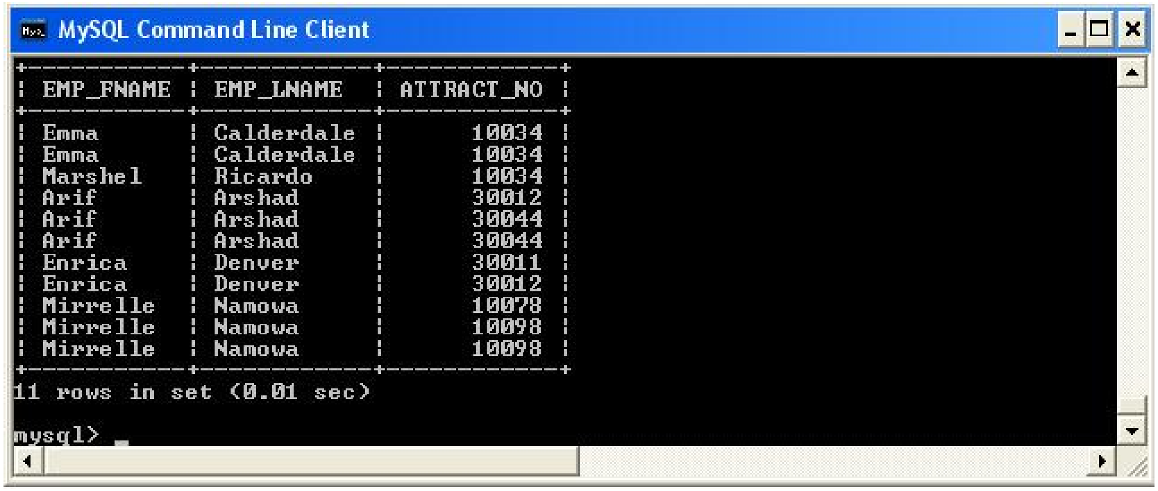
```
mysql> SELECT   TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY,
    -> LINE_PRICE
    -> FROM             SALES NATURAL JOIN SALES_LINE;
+----------------+------------+---------+----------+------------+
| TRANSACTION_NO | SALE_DATE  | LINE_NO | LINE_QTY | LINE_PRICE |
+----------------+------------+---------+----------+------------+
|          12781 | 2007-05-18 |       1 |        2 |      69.98 |
|          12781 | 2007-05-18 |       2 |        1 |      14.99 |
|          12782 | 2007-05-18 |       1 |        2 |      69.98 |
|          12783 | 2007-05-18 |       1 |        2 |      41.98 |
|          12784 | 2007-05-18 |       2 |        1 |      14.99 |
|          12785 | 2007-05-18 |       1 |        1 |      14.99 |
|          12785 | 2007-05-18 |       2 |        1 |      34.99 |
|          12785 | 2007-05-18 |       3 |        4 |     139.96 |
|          34534 | 2007-05-18 |       1 |        4 |     168.40 |
|          34534 | 2007-05-18 |       2 |        1 |      22.50 |
|          34534 | 2007-05-18 |       3 |        2 |      21.98 |
|          34535 | 2007-05-18 |       1 |        2 |      84.20 |
|          34536 | 2007-05-18 |       1 |        2 |      21.98 |
|          34537 | 2007-05-18 |       1 |        2 |      84.20 |
|          34537 | 2007-05-18 |       2 |        1 |      22.50 |
|          34538 | 2007-05-18 |       1 |        2 |      21.98 |
|          34539 | 2007-05-18 |       1 |        2 |      21.98 |
|          34539 | 2007-05-18 |       2 |        2 |      84.20 |
|          34540 | 2007-05-18 |       1 |        4 |     168.40 |
|          34540 | 2007-05-18 |       2 |        1 |      22.50 |
|          34540 | 2007-05-18 |       3 |        2 |      21.98 |
|          34541 | 2007-05-18 |       1 |        2 |      84.20 |
|          67589 | 2007-05-18 |       1 |        2 |      57.34 |
|          67589 | 2007-05-18 |       2 |        2 |      37.12 |
|          67590 | 2007-05-18 |       1 |        2 |      57.34 |
|          67590 | 2007-05-18 |       2 |        2 |      37.12 |
|          67591 | 2007-05-18 |       1 |        1 |      18.56 |
|          67591 | 2007-05-18 |       2 |        1 |      12.12 |
|          67592 | 2007-05-18 |       1 |        4 |     114.68 |
|          67593 | 2007-05-18 |       1 |        2 |      57.34 |
|          67593 | 2007-05-18 |       2 |        2 |      37.12 |
+----------------+------------+---------+----------+------------+
31 rows in set (0.00 sec)

mysql>
mysql>
mysql>
```

Figure 48: Results of SALES NATURAL JOIN SALES_LINE;

One important difference between the natural join and the “old-style” join syntax as illustrated in Figure 46, 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 employees 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 49.



```
+------------+------------+-------------+
| EMP_FNAME  | EMP_LNAME  | ATTRACT_NO  |
+------------+------------+-------------+
| Emma       | Calderdale |       10034 |
| Emma       | Calderdale |       10034 |
| Marshel    | Ricardo    |       10034 |
| Arif       | Arshad     |       30012 |
| Arif       | Arshad     |       30044 |
| Arif       | Arshad     |       30044 |
| Enrica     | Denver     |       30011 |
| Enrica     | Denver     |       30012 |
| Mirrelle   | Namowa     |       10078 |
| Mirrelle   | Namowa     |       10098 |
| Mirrelle   | Namowa     |       10098 |
+------------+------------+-------------+
11 rows in set (0.01 sec)

mysql> _
```

Figure 49: Query results for Task 6.3

6.5 Join USING

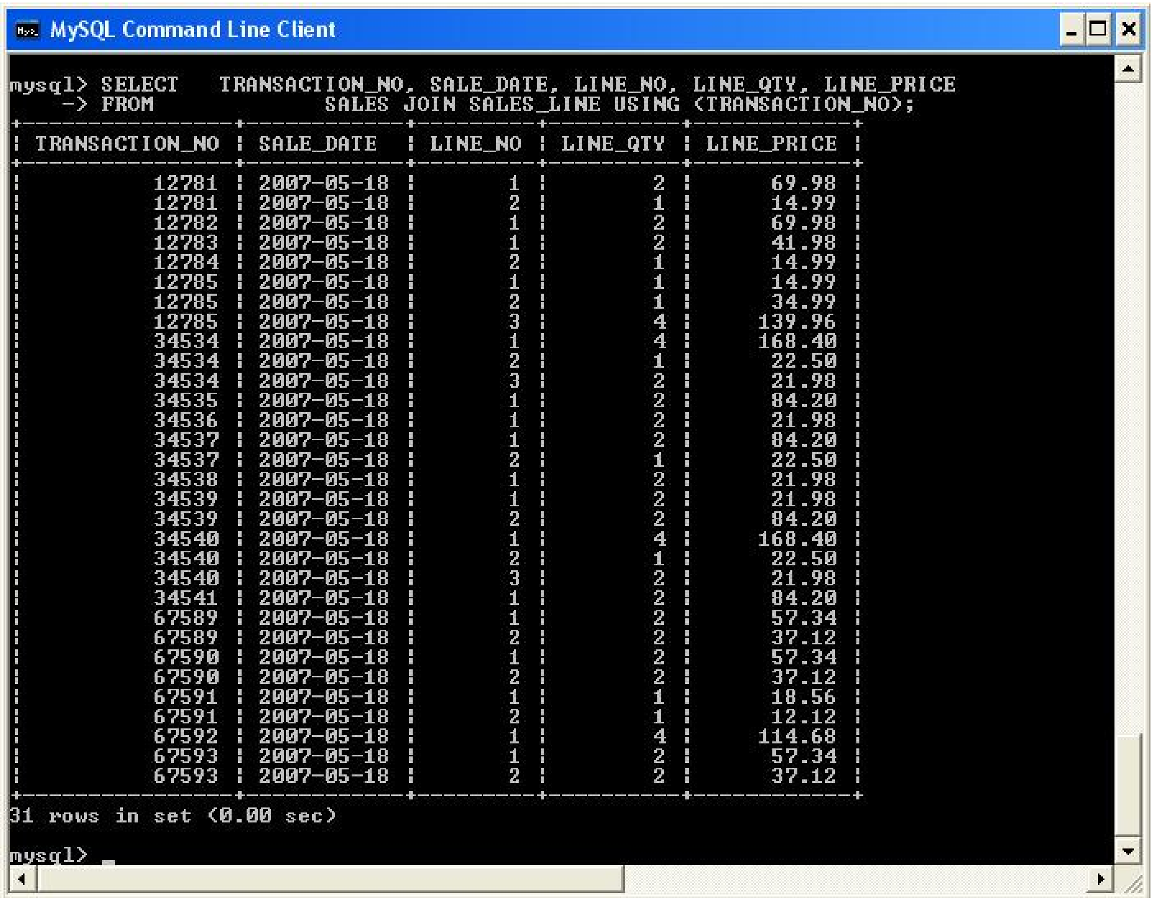
A second way to express a join is through the USING keyword. That query returns only the rows with matching values in the column indicated in the USING clause—and that column must exist in both tables. The syntax is:

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 50.

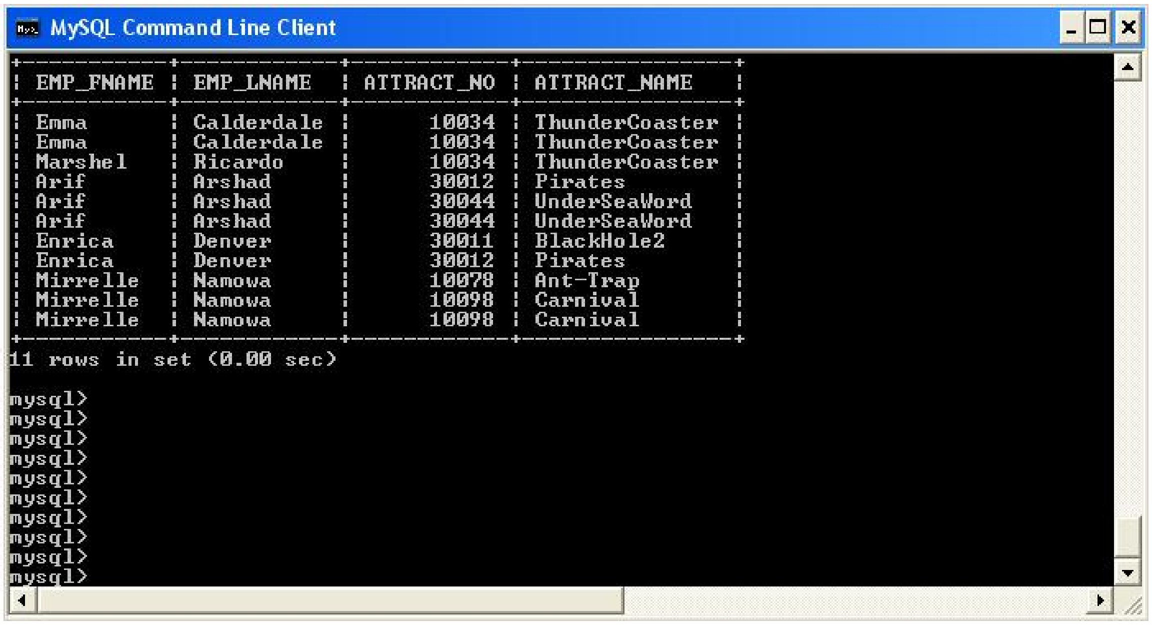


```
mysql> SELECT   TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY, LINE_PRICE
    -> FROM             SALES JOIN SALES_LINE USING (TRANSACTION_NO);
+----------------+------------+---------+----------+------------+
| TRANSACTION_NO | SALE_DATE  | LINE_NO | LINE_QTY | LINE_PRICE |
+----------------+------------+---------+----------+------------+
|          12781 | 2007-05-18 |       1 |        2 |      69.98 |
|          12781 | 2007-05-18 |       2 |        1 |      14.99 |
|          12782 | 2007-05-18 |       1 |        2 |      69.98 |
|          12783 | 2007-05-18 |       1 |        2 |      41.98 |
|          12784 | 2007-05-18 |       2 |        1 |      14.99 |
|          12785 | 2007-05-18 |       1 |        1 |      14.99 |
|          12785 | 2007-05-18 |       2 |        1 |      34.99 |
|          12785 | 2007-05-18 |       3 |        4 |     139.96 |
|          34534 | 2007-05-18 |       1 |        4 |     168.40 |
|          34534 | 2007-05-18 |       2 |        1 |      22.50 |
|          34534 | 2007-05-18 |       3 |        2 |      21.98 |
|          34535 | 2007-05-18 |       1 |        2 |      84.20 |
|          34536 | 2007-05-18 |       1 |        2 |      21.98 |
|          34537 | 2007-05-18 |       1 |        2 |      84.20 |
|          34537 | 2007-05-18 |       2 |        1 |      22.50 |
|          34538 | 2007-05-18 |       1 |        2 |      21.98 |
|          34539 | 2007-05-18 |       1 |        2 |      21.98 |
|          34539 | 2007-05-18 |       2 |        2 |      84.20 |
|          34540 | 2007-05-18 |       1 |        4 |     168.40 |
|          34540 | 2007-05-18 |       2 |        1 |      22.50 |
|          34540 | 2007-05-18 |       3 |        2 |      21.98 |
|          34541 | 2007-05-18 |       1 |        2 |      84.20 |
|          67589 | 2007-05-18 |       1 |        2 |      57.34 |
|          67589 | 2007-05-18 |       2 |        2 |      37.12 |
|          67590 | 2007-05-18 |       1 |        2 |      57.34 |
|          67590 | 2007-05-18 |       2 |        2 |      37.12 |
|          67591 | 2007-05-18 |       1 |        1 |      18.56 |
|          67591 | 2007-05-18 |       2 |        1 |      12.12 |
|          67592 | 2007-05-18 |       1 |        4 |     114.68 |
|          67593 | 2007-05-18 |       1 |        2 |      57.34 |
|          67593 | 2007-05-18 |       2 |        2 |      37.12 |
+----------------+------------+---------+----------+------------+
31 rows in set (0.00 sec)

mysql> _
```

Figure 50: 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.

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 51.



```
+------------+------------+------------+----------------+
| EMP_FNAME  | EMP_LNAME  | ATTRACT_NO | ATTRACT_NAME   |
+------------+------------+------------+----------------+
| Emma       | Calderdale |      10034 | ThunderCoaster |
| Emma       | Calderdale |      10034 | ThunderCoaster |
| Marshel    | Ricardo    |      10034 | ThunderCoaster |
| Arif       | Arshad     |      30012 | Pirates        |
| Arif       | Arshad     |      30044 | UnderSeaWord   |
| Arif       | Arshad     |      30044 | UnderSeaWord   |
| Enrica     | Denver     |      30011 | BlackHole2     |
| Enrica     | Denver     |      30012 | Pirates        |
| Mirrelle   | Namowa     |      10078 | Ant-Trap       |
| Mirrelle   | Namowa     |      10098 | Carnival       |
| Mirrelle   | Namowa     |      10098 | Carnival       |
+------------+------------+------------+----------------+
11 rows in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
```

Figure 51: Query results for Task 6.4

6.6 Join ON

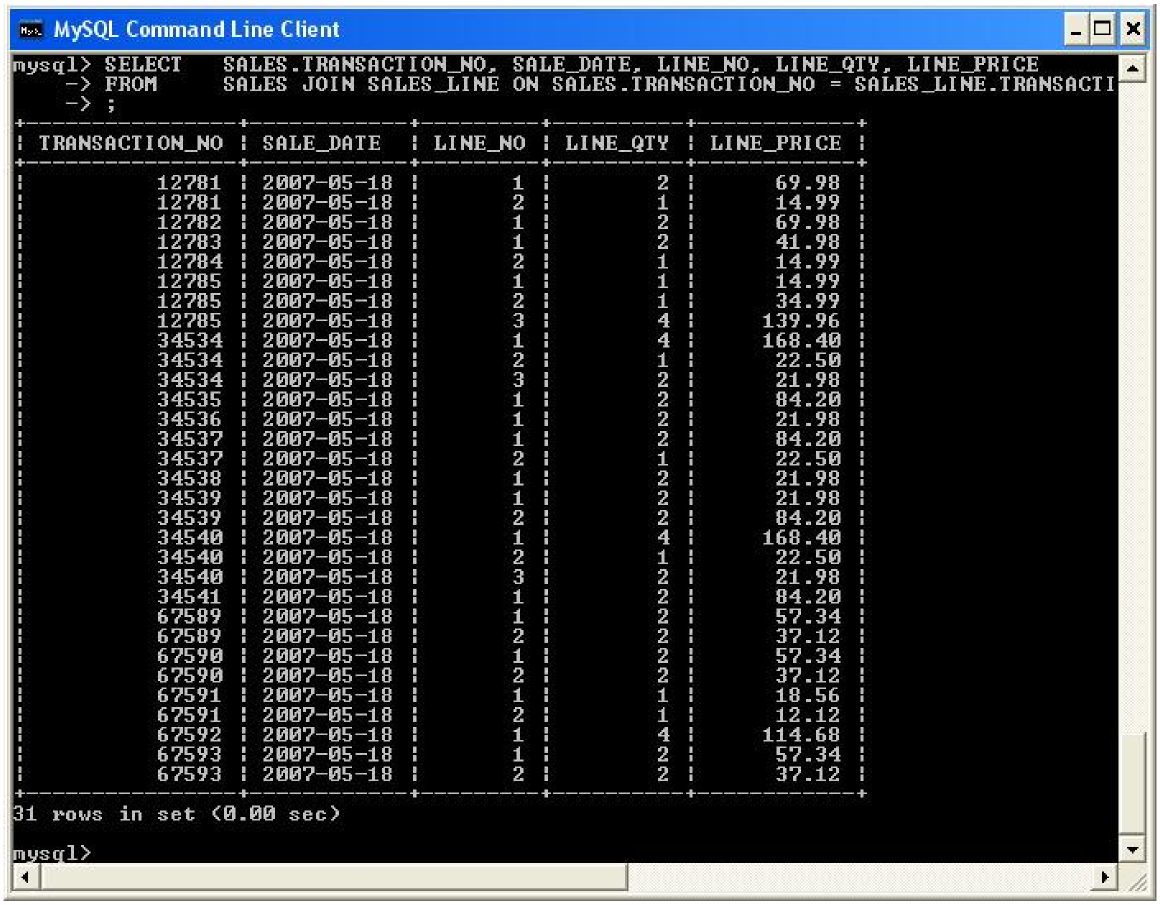
The previous two join styles used common attribute names in the joining tables. Another way to express a join when the tables have no common attribute names is to use the JOIN ON operand. That query will return only the rows that meet the indicated join condition. The join condition will typically include 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 52.

```
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;
```



```
mysql> SELECT    SALES.TRANSACTION_NO, SALE_DATE, LINE_NO, LINE_QTY, LINE_PRICE
    -> FROM      SALES JOIN SALES_LINE ON SALES.TRANSACTION_NO = SALES_LINE.TRANSACTI
    -> ;
+----------------+------------+---------+----------+------------+
| TRANSACTION_NO | SALE_DATE  | LINE_NO | LINE_QTY | LINE_PRICE |
+----------------+------------+---------+----------+------------+
|          12781 | 2007-05-18 |       1 |        2 |      69.98 |
|          12781 | 2007-05-18 |       2 |        1 |      14.99 |
|          12782 | 2007-05-18 |       1 |        2 |      69.98 |
|          12783 | 2007-05-18 |       1 |        2 |      41.98 |
|          12784 | 2007-05-18 |       2 |        1 |      14.99 |
|          12785 | 2007-05-18 |       1 |        1 |      14.99 |
|          12785 | 2007-05-18 |       2 |        1 |      34.99 |
|          12785 | 2007-05-18 |       3 |        4 |     139.96 |
|          34534 | 2007-05-18 |       1 |        4 |     168.40 |
|          34534 | 2007-05-18 |       2 |        1 |      22.50 |
|          34534 | 2007-05-18 |       3 |        2 |      21.98 |
|          34535 | 2007-05-18 |       1 |        2 |      84.20 |
|          34536 | 2007-05-18 |       1 |        2 |      21.98 |
|          34537 | 2007-05-18 |       1 |        2 |      84.20 |
|          34537 | 2007-05-18 |       2 |        1 |      22.50 |
|          34538 | 2007-05-18 |       1 |        2 |      21.98 |
|          34539 | 2007-05-18 |       1 |        2 |      21.98 |
|          34539 | 2007-05-18 |       2 |        2 |      84.20 |
|          34540 | 2007-05-18 |       1 |        4 |     168.40 |
|          34540 | 2007-05-18 |       2 |        1 |      22.50 |
|          34540 | 2007-05-18 |       3 |        2 |      21.98 |
|          34541 | 2007-05-18 |       1 |        2 |      84.20 |
|          67589 | 2007-05-18 |       1 |        2 |      57.34 |
|          67589 | 2007-05-18 |       2 |        2 |      37.12 |
|          67590 | 2007-05-18 |       1 |        2 |      57.34 |
|          67590 | 2007-05-18 |       2 |        2 |      37.12 |
|          67591 | 2007-05-18 |       1 |        1 |      18.56 |
|          67591 | 2007-05-18 |       2 |        1 |      12.12 |
|          67592 | 2007-05-18 |       1 |        4 |     114.68 |
|          67593 | 2007-05-18 |       1 |        2 |      57.34 |
|          67593 | 2007-05-18 |       2 |        2 |      37.12 |
+----------------+------------+---------+----------+------------+
31 rows in set (0.00 sec)

mysql>
```

Figure 52: 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:

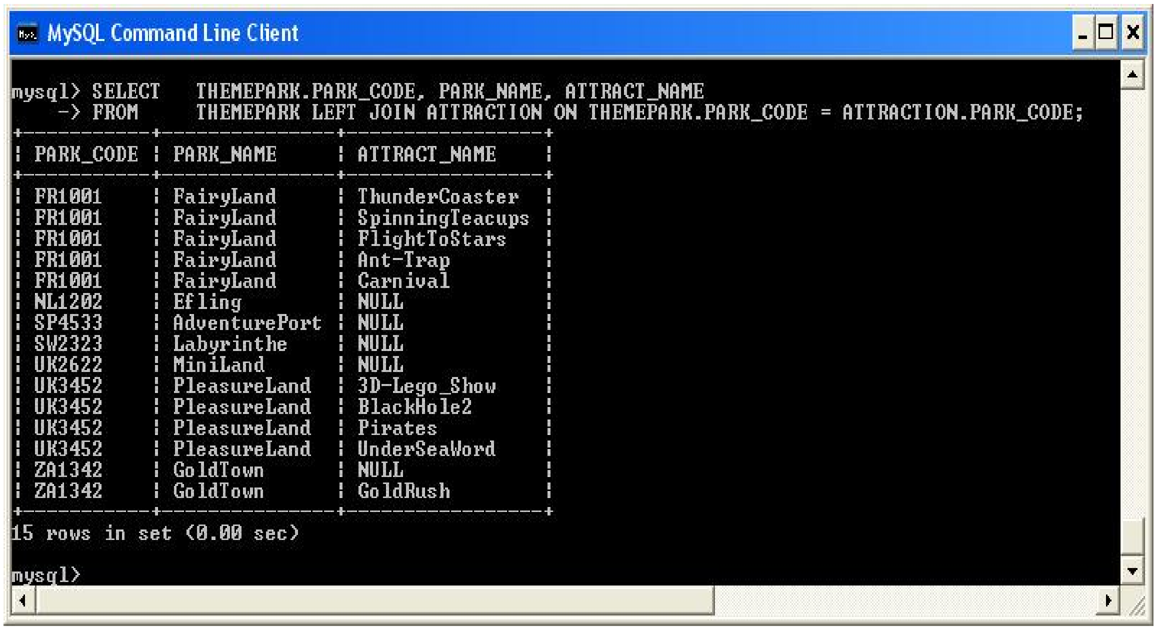
SELECT column-list

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 53.



```
mysql> SELECT   THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME
    -> 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        | NULL            |
| SP4533    | AdventurePort | NULL            |
| SW2323    | Labyrinthe    | NULL            |
| UK2622    | MiniLand      | NULL            |
| UK3452    | PleasureLand  | 3D-Lego_Show    |
| UK3452    | PleasureLand  | BlackHole2      |
| UK3452    | PleasureLand  | Pirates         |
| UK3452    | PleasureLand  | UnderSeaWord    |
| ZA1342    | GoldTown      | NULL            |
| ZA1342    | GoldTown      | GoldRush        |
+-----------+---------------+-----------------+
15 rows in set (0.00 sec)

mysql>
```

Figure 53: LEFT OUTER JOIN example

Task 6.5 Enter the query above and check your results with those shown in Figure 53.

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:

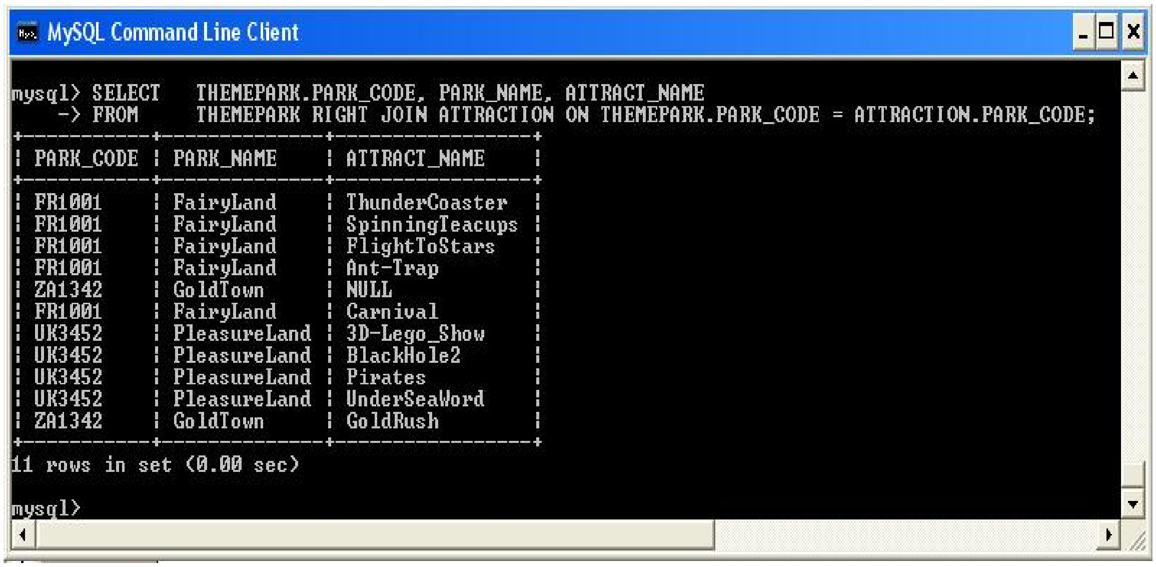
SELECT column-list

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 54.



```
mysql> SELECT   THEMEPARK.PARK_CODE, PARK_NAME, ATTRACT_NAME
    -> FROM     THEMEPARK RIGHT 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        |
| ZA1342    | GoldTown     | NULL            |
| FR1001    | FairyLand    | Carnival        |
| UK3452    | PleasureLand | 3D-Lego_Show    |
| UK3452    | PleasureLand | BlackHole2      |
| UK3452    | PleasureLand | Pirates         |
| UK3452    | PleasureLand | UnderSeaWord    |
| ZA1342    | GoldTown     | GoldRush        |
+-----------+--------------+-----------------+
11 rows in set (0.00 sec)

mysql>
```

Figure 54: RIGHT OUTER JOIN example

Task 6.6 Enter the query above and check your results with those shown in Figure 54.

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. (Hint: Remember the format of MySQL dates is '2007-05-18').

Lab 7: SQL Functions

The learning objectives of this lab are to

- Learn about selected MySQL 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 that are implemented in MySQL in detail. 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

In MySQL there are a number of useful date and time functions. However, first it is important to briefly look at the main date and time types are available to MySQL. These are shown in the table below:

Table 7.1 MySQL Date and Time data types

DATETIME	YYYY-MM-DD HH:MM:SS
DATE	YYYY-MM-DD
TIMESTAMP	YYYYMMDDHHSSMM
TIME	HH:MM:SS

YEAR	YYYY

As you can see from Table 7.1, the DATE type is stored in a special internal format that includes just the year, month and day whilst the DATETIME data type also stores the hours, minutes, and seconds. If you try to enter a date in a format other than the Year-Month-Day format then it might work, but it won't be storing them as you expect!

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

```
SELECT      DISTINCT(SALE_DATE )
FROM        SALES;
```

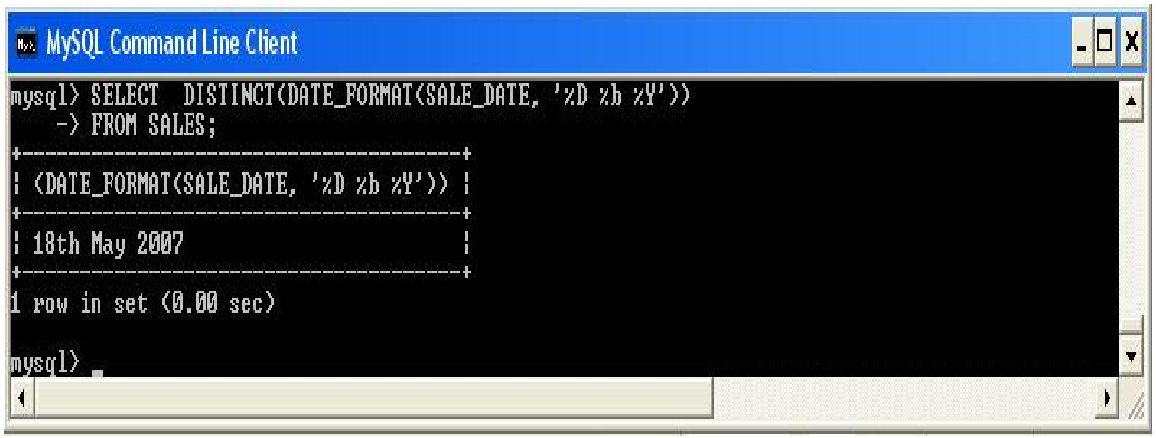
It is possible to change the format of the date using the DATE_FORMAT() function. The syntax of this function is

```
DATE_FORMAT(date,format)
```

The function formats the date value according to the format string.

For example, the following query formats the date as 18th May 2007 using ‘ date specifiers’ as shown in Figure 55.

```
SELECT  DISTINCT(DATE_FORMAT(SALE_DATE, ‘%D %b %Y’))

FROM SALES;
```



```
mysql> SELECT  DISTINCT(DATE_FORMAT(SALE_DATE, '%D %b %Y'))
    -> FROM SALES;
+--------------------------------------+
| (DATE_FORMAT(SALE_DATE, '%D %b %Y')) |
+--------------------------------------+
| 18th May 2007                        |
+--------------------------------------+
1 row in set (0.00 sec)

mysql> _
```

Figure 55 Formatting Dates in MySQL

Table 7.2 taken directly from the MySQL Manual 5.0 shows a complete list of specifiers that can be used in the `format` string.

Specifier	Description
`%a`	Abbreviated weekday name (`Sun`..`Sat`)
`%b`	Abbreviated month name (`Jan`..`Dec`)
`%c`	Month, numeric (`0`..`12`)
`%D`	Day of the month with English suffix (`0th`, `1st`, `2nd`, `3rd`, …)
`%d`	Day of the month, numeric (`00`..`31`)
`%e`	Day of the month, numeric (`0`..`31`)
`%f`	Microseconds (`000000`..`999999`)
`%H`	Hour (`00`..`23`)
`%h`	Hour (`01`..`12`)
`%I`	Hour (`01`..`12`)
`%i`	Minutes, numeric (`00`..`59`)
`%j`	Day of year (`001`..`366`)
`%k`	Hour (`0`..`23`)
`%l`	Hour (`1`..`12`)
`%M`	Month name (`January`..`December`)
`%m`	Month, numeric (`00`..`12`)
`%p`	`AM` or `PM`
`%r`	Time, 12-hour (`hh:mm:ss` followed by `AM` or `PM`)
`%S`	Seconds (`00`..`59`)
`%s`	Seconds (`00`..`59`)
`%T`	Time, 24-hour (`hh:mm:ss`)
`%U`	Week (`00`..`53`), where Sunday is the first day of the week
`%u`	Week (`00`..`53`), where Monday is the first day of the week
`%V`	Week (`01`..`53`), where Sunday is the first day of the week; used with `%X`

`%v`	Week (`01`..`53`), where Monday is the first day of the week; used with `%x`
`%W`	Weekday name (`Sunday`..`Saturday`)
`%w`	Day of the week (`0`=Sunday..`6`=Saturday)
`%X`	Year for the week where Sunday is the first day of the week, numeric, four digits; used with `%V`
`%x`	Year for the week, where Monday is the first day of the week, numeric, four digits; used with `%v`
`%Y`	Year, numeric, four digits
`%y`	Year, numeric (two digits)
`%%`	A literal ‘`%`’ character
`%`*`x`*	*`x`*, for any ‘*`x`*’ not listed above

Task 7.2 Using the date specifiers in Table 7.2, modify the query shown in Figure 55 to display the date in the format ’Fri – 18 – 5 – 07’.

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

CURRENT DATE and CURRENT TIME

The CURRENT_DATE function returns today’s date while the CURRENT_TIME function returns the current time.

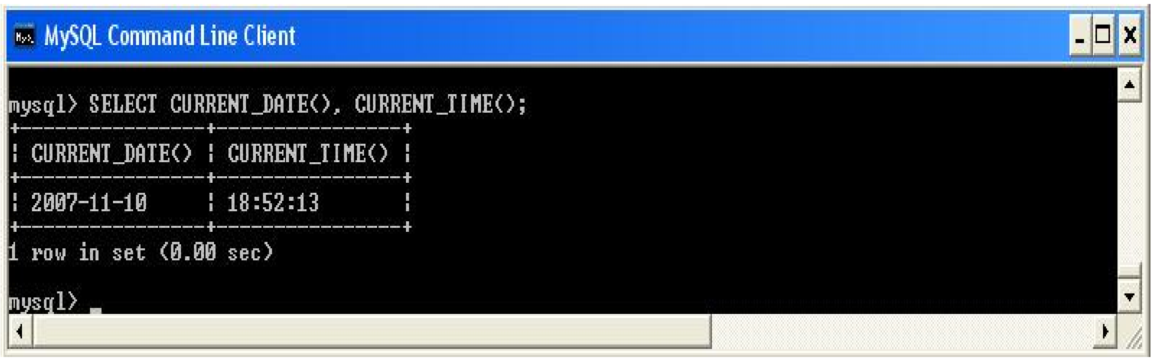
Task 7.3 Enter the following query to display today’s date and time. Notice that in MySQL the functions are called using the SELECT statement but no FROM clause is needed.

```
mysql> SELECT CURRENT_DATE(), CURRENT_TIME();
```

Note

CURRENT_TIME and CURRENT_DATE are synonyms for CURTIME() and CURDATE respectively.

The output for this query is shown in Figure 56.



```
mysql> SELECT CURRENT_DATE(), CURRENT_TIME();
+----------------+----------------+
| CURRENT_DATE() | CURRENT_TIME() |
+----------------+----------------+
| 2007-11-10     | 18:52:13       |
+----------------+----------------+
1 row in set (0.00 sec)

mysql> _
```

Figure 56 Displaying the current date and time.

MONTH, DAYOFMONTH and YEAR

MySQL provides functions for extracting the month, day or year from any given date. The syntax of each function is as follows:

DAYOFMONTH(date) returns the day of the month for date, in the range 0 to 31.

MONTH(date) returns the month for date, in the range 0 to 12.

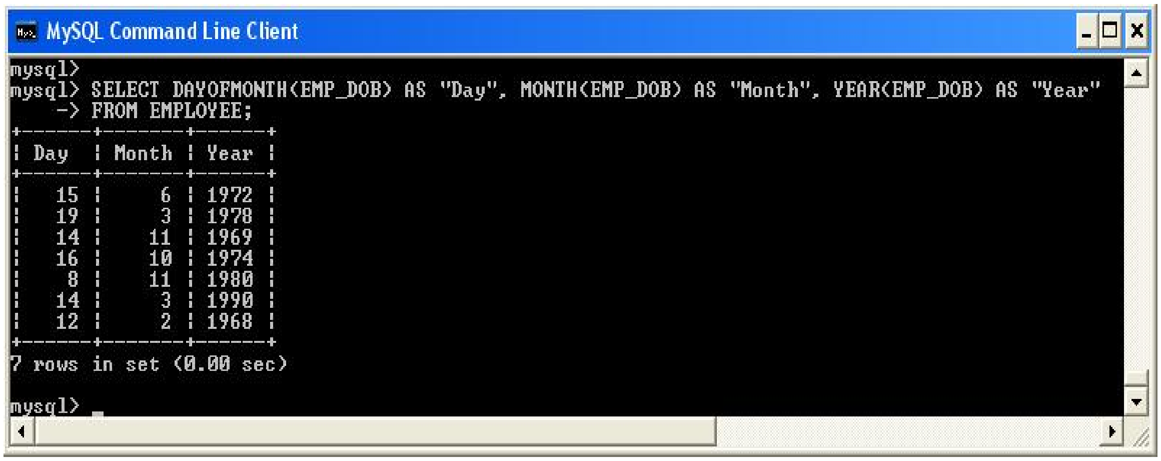
YEAR(date) returns the year for date, in the range 1000 to 9999, or 0 for the “zero” date.

The following query shows how these three functions can be used to display different parts of an employee’s date of birth. The output of this query is shown in Figure 57.

SELECT DAYOFMONTH(EMP_DOB) AS “Day”, MONTH(EMP_DOB) AS “Month”,

YEAR(EMP_DOB) AS “Year”

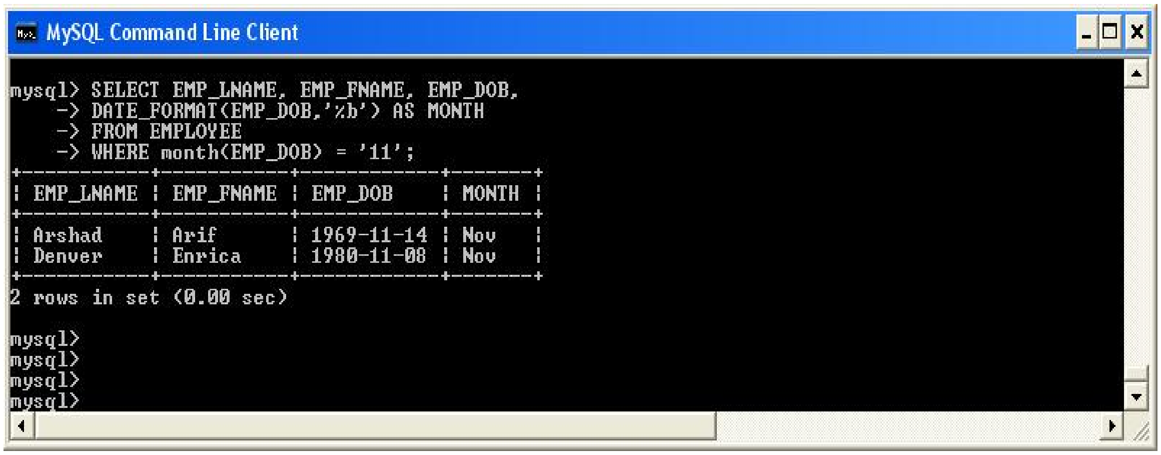
FROM EMPLOYEE;



```
mysql>
mysql> SELECT DAYOFMONTH(EMP_DOB) AS "Day", MONTH(EMP_DOB) AS "Month", YEAR(EMP_DOB) AS "Year"
    -> FROM EMPLOYEE;
+------+-------+------+
| Day  | Month | Year |
+------+-------+------+
|   15 |     6 | 1972 |
|   19 |     3 | 1978 |
|   14 |    11 | 1969 |
|   16 |    10 | 1974 |
|    8 |    11 | 1980 |
|   14 |     3 | 1990 |
|   12 |     2 | 1968 |
+------+-------+------+
7 rows in set (0.00 sec)

mysql> _
```

Figure 57 Using the MONTH, DAYOFMONTH and YEAR functions.

Task 7.3 Write a query that displays all employees who were born in November. Your output should match that shown in Figure 58.



```
mysql> SELECT EMP_LNAME, EMP_FNAME, EMP_DOB,
    -> DATE_FORMAT(EMP_DOB,'%b') AS MONTH
    -> FROM EMPLOYEE
    -> WHERE month(EMP_DOB) = '11';
+-----------+-----------+------------+-------+
| EMP_LNAME | EMP_FNAME | EMP_DOB    | MONTH |
+-----------+-----------+------------+-------+
| Arshad    | Arif      | 1969-11-14 | Nov   |
| Denver    | Enrica    | 1980-11-08 | Nov   |
+-----------+-----------+------------+-------+
2 rows in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
```

Figure 58 Output for Task 7.3.

DATEDIFF

The DATEDIFF function subtracts two dates and returns a value in days from one date to the other. The following example calculates the number of days between the 1st January 2008 and the 25th December 2008.

```
SELECT  DATEDIFF(‘2008-12-25’,‘2008-01-01’);
```

Task 7.4 Enter the query above and see how many days it is until the 25th December. Then modify the query to see how many days it is from today’s date until 25th December 2009.

DATE_ADD and DATE_SUB

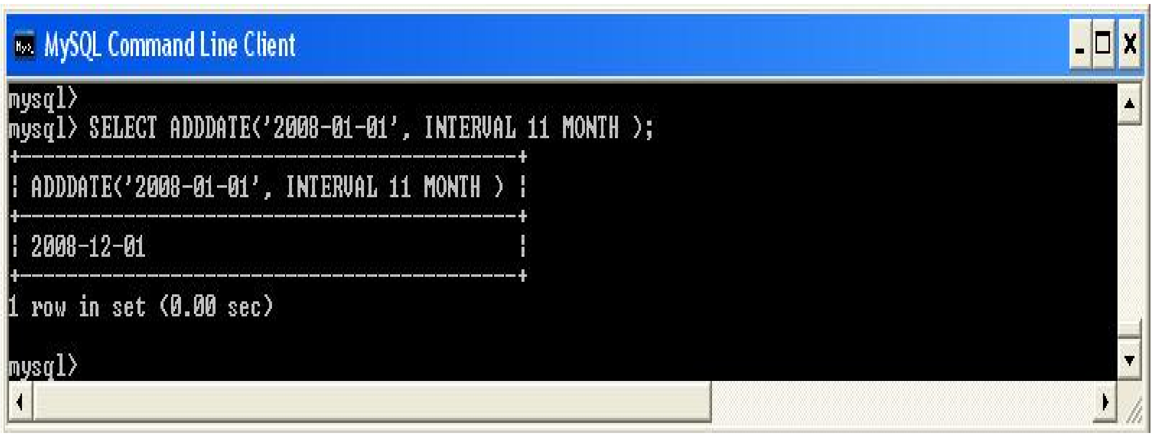
The DATE_ADD and DATE_SUB functions both perform date arithmetic and allow you to either add or subtract two dates from one another. The syntax of these functions is:

```
DATE_ADD(date,INTERVAL expr unit)

DATE_SUB(date,INTERVAL expr unit)
```

Where expr is an expression specifying the interval value to be added or subtracted from the starting date and unit is a keyword indicating the units in which the expression should be interpreted.

For example, the following query adds 11 months to the date 1st January 2008 to display a new date of 1st December 2008. The output for this query is shown in Figure 59.

SELECT ADDDATE(‘2008-01-01’, INTERVAL 11 MONTH);



```
mysql>
mysql> SELECT ADDDATE('2008-01-01', INTERVAL 11 MONTH );
+-------------------------------------------+
| ADDDATE('2008-01-01', INTERVAL 11 MONTH ) |
+-------------------------------------------+
| 2008-12-01                                |
+-------------------------------------------+
1 row in set (0.00 sec)

mysql>
```

Figure 59 Adding months to a date

A full list of the different interval types can be found in the MySQL Reference Manual 5.0.

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 hiredate). Check that the output is correct.

SELECT EMP_LNAME, EMP_FNAME, EMP_HIRE_DATE,

ADDDATE(EMP_HIRE_DATE, INTERVAL 12 MONTH)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 20 days of a month:

```
SELECT *
FROM SALES
WHERE SALE_DATE >= LAST_DAY(SALE_DATE)-20;
```

7.2 Numeric Functions

In this section, you will learn about MySQL 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 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
TRUNCATE	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 2.

SELECT LINE_PRICE, ROUND(LINE_PRICE,1) AS “LINE_PRICE1”,

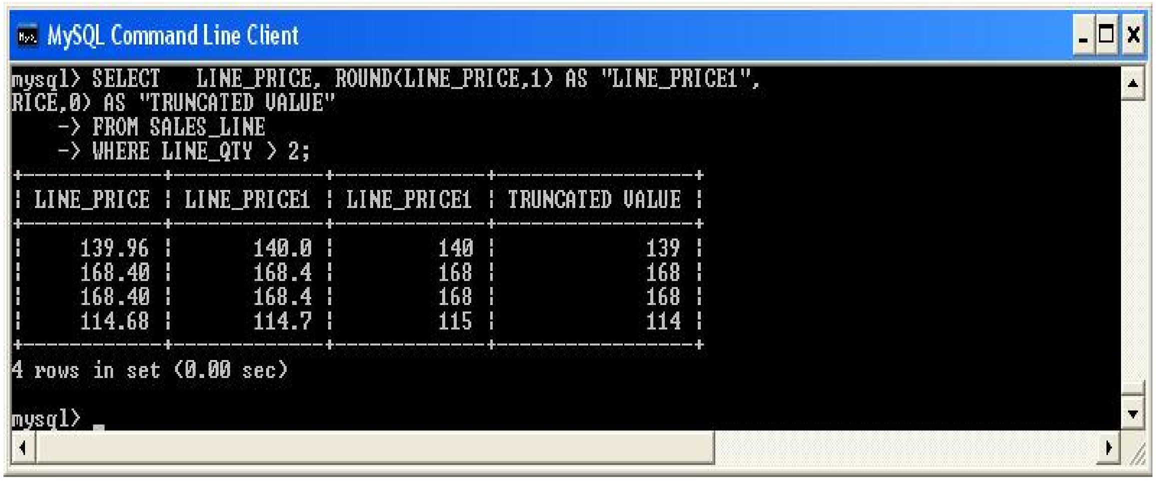
ROUND(LINE_PRICE,0) AS “LINE_PRICE1”,

TRUNCATE(LINE_PRICE,0) AS “TRUNCATED VALUE”

FROM SALES_LINE

WHERE LINE_QTY > 2;

The output for this query can be seen in Figure 60.



```
mysql> SELECT   LINE_PRICE, ROUND(LINE_PRICE,1) AS "LINE_PRICE1",
RICE,0) AS "TRUNCATED VALUE"
    -> FROM SALES_LINE
    -> WHERE LINE_QTY > 2;
+------------+-------------+-------------+-----------------+
| LINE_PRICE | LINE_PRICE1 | LINE_PRICE1 | TRUNCATED VALUE |
+------------+-------------+-------------+-----------------+
|     139.96 |       140.0 |         140 |             139 |
|     168.40 |       168.4 |         168 |             168 |
|     168.40 |       168.4 |         168 |             168 |
|     114.68 |       114.7 |         115 |             114 |
+------------+-------------+-------------+-----------------+
4 rows in set (0.00 sec)

mysql> _
```

Figure 60 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-used functions in programming.

Table 5 shows a subset of the most useful string manipulation functions in MySQL.

Table 5 Selected MySQL string functions.

Function	Description
CONCAT	Concatenates data from two different character columns and returns a single column. Syntax: CONCAT(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)

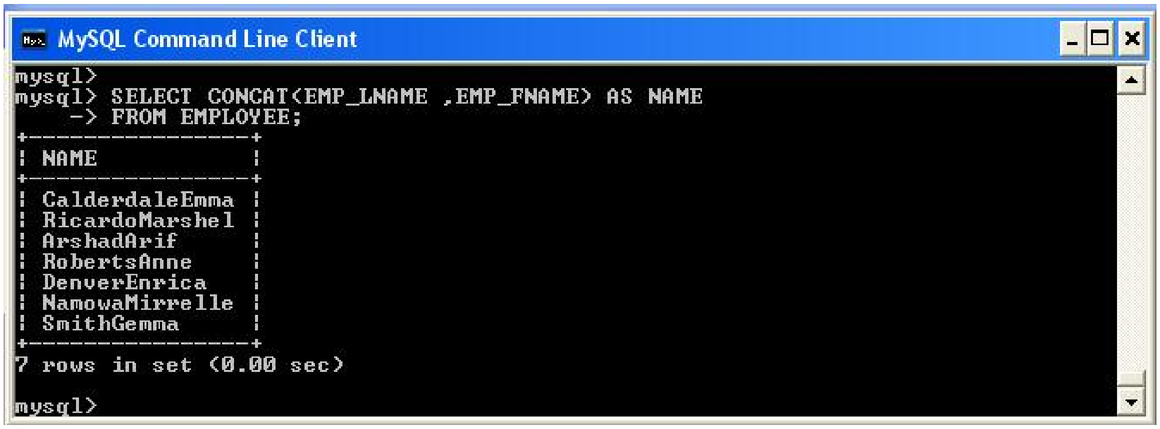
We will now look at examples of some of these string functions.

CONCAT

The following query illustrates the CONCAT function. It lists all employee first and last names concatenated together. The output for this query can be seen in Figure 61.

SELECT CONCAT(EMP_LNAME ,EMP_FNAME) AS NAME

FROM EMPLOYEE;



```
mysql>
mysql> SELECT CONCAT(EMP_LNAME ,EMP_FNAME) AS NAME
    -> FROM EMPLOYEE;
+------------------+
| NAME             |
+------------------+
| CalderdaleEmma   |
| RicardoMarshel   |
| ArshadArif       |
| RobertsAnne      |
| DenverEnrica     |
| NamowaMirrelle   |
| SmithGemma       |
+------------------+
7 rows in set (0.00 sec)

mysql>
```

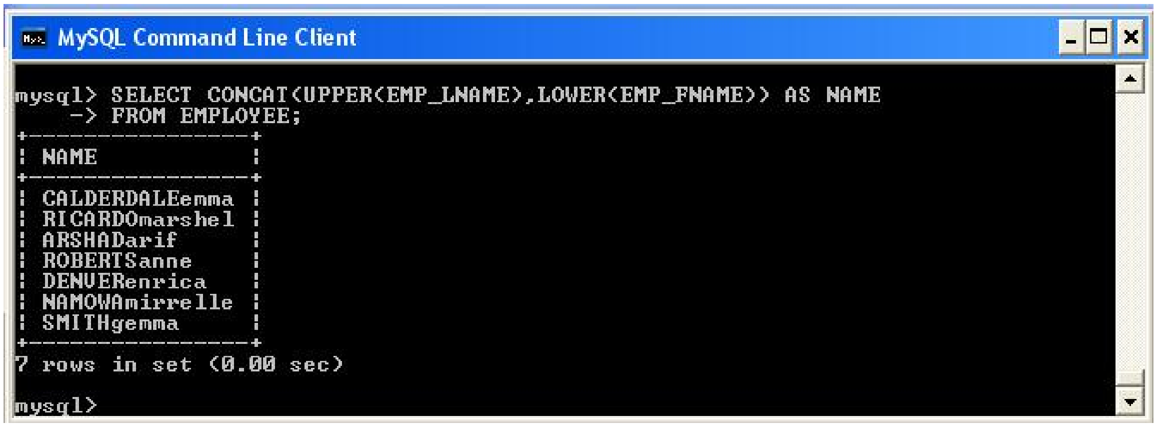
Figure 61 Concatenation of employee's first and last names

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 62.

SELECT CONCAT(UPPER(EMP_LNAME),LOWER(EMP_FNAME)) AS NAME

FROM EMPLOYEE;



```
mysql> SELECT CONCAT(UPPER(EMP_LNAME),LOWER(EMP_FNAME)) AS NAME
    -> FROM EMPLOYEE;
+------------------+
| NAME             |
+------------------+
| CALDERDALEemma   |
| RICARDOmarshel   |
| ARSHADarif       |
| ROBERTSanne      |
| DENVERenrica     |
| NAMOWAmirrelle   |
| SMITHgemma       |
+------------------+
7 rows in set (0.00 sec)

mysql>
```

Figure 62 Displaying upper and lower case employee names.

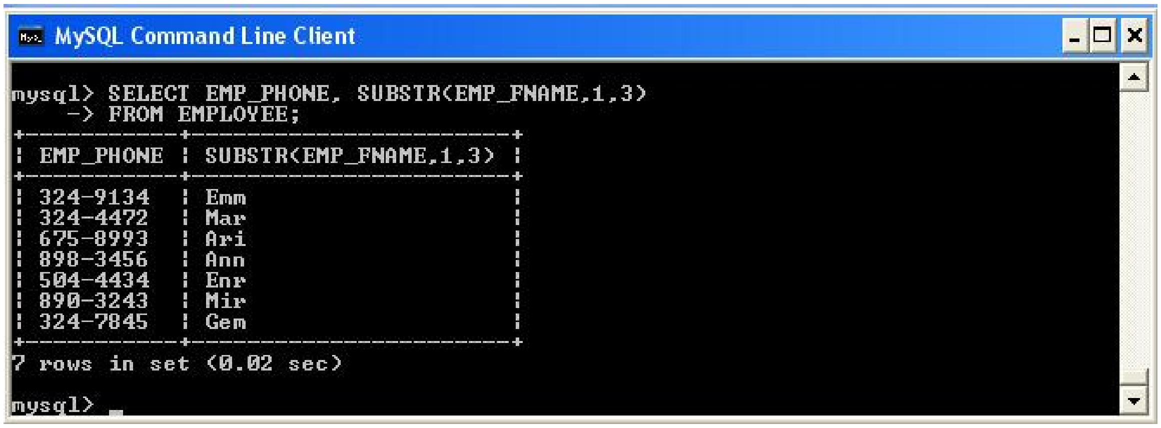
SUBSTR

The following example lists the first three characters of all the employees' first name.

The output of this query is shown in Figure 63.

SELECT EMP_PHONE, SUBSTR(EMP_FNAME,1,3)

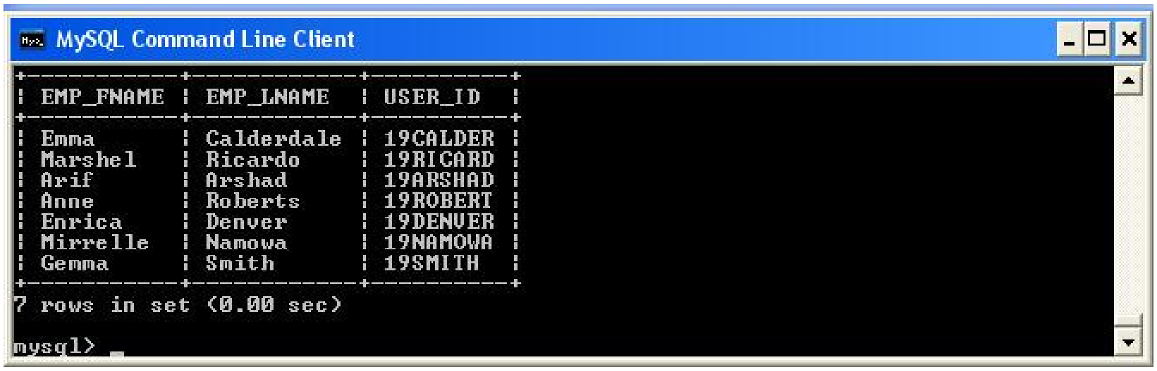
FROM EMPLOYEE;



```
mysql> SELECT EMP_PHONE, SUBSTR(EMP_FNAME,1,3)
    -> FROM EMPLOYEE;
+-----------+-----------------------+
| EMP_PHONE | SUBSTR(EMP_FNAME,1,3) |
+-----------+-----------------------+
| 324-9134  | Emm                   |
| 324-4472  | Mar                   |
| 675-8993  | Ari                   |
| 898-3456  | Ann                   |
| 504-4434  | Enr                   |
| 890-3243  | Mir                   |
| 324-7845  | Gem                   |
+-----------+-----------------------+
7 rows in set (0.02 sec)

mysql> _
```

Figure 63 Displaying the first 3 characters of the employees first name

Task 7.10 Write a query which generates a list of employee user IDs, using the first day of the month they were born and the first six characters of last name in UPPER case. Your query should return the results shown in Figure 64.



```
+-----------+------------+----------+
| EMP_FNAME | EMP_LNAME  | USER_ID  |
+-----------+------------+----------+
| Emma      | Calderdale | 19CALDER |
| Marshel   | Ricardo    | 19RICARD |
| Arif      | Arshad     | 19ARSHAD |
| Anne      | Roberts    | 19ROBERT |
| Enrica    | Denver     | 19DENVER |
| Mirrelle  | Namowa     | 19NAMOWA |
| Gemma     | Smith      | 19SMITH  |
+-----------+------------+----------+
7 rows in set (0.00 sec)

mysql> _
```

Figure 64 Results for Task 7.10.

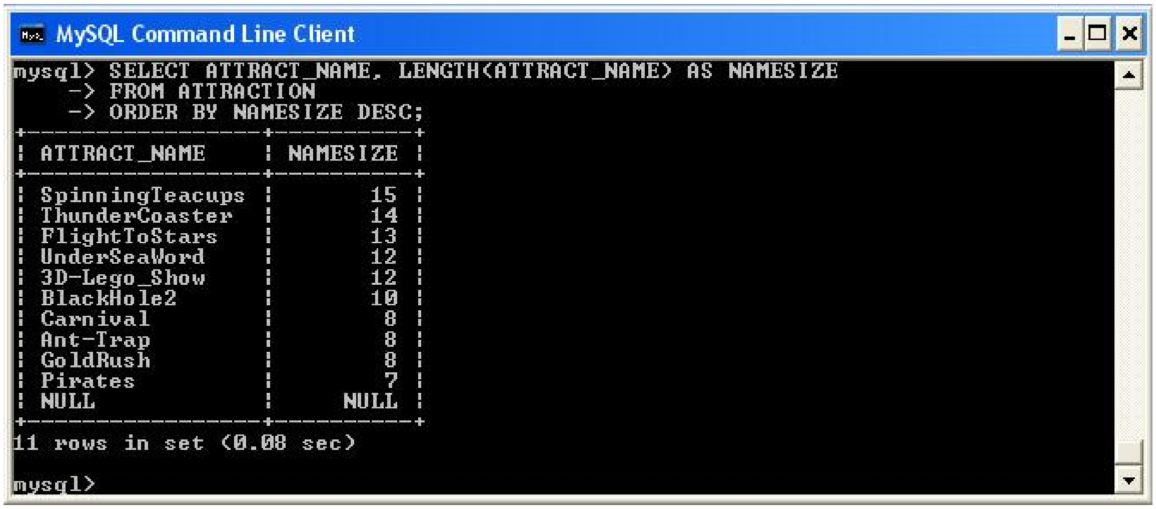
LENGTH

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 65.

SELECT ATTRACT_NAME, LENGTH(ATTRACT_NAME) AS NAMESIZE

FROM ATTRACTION

ORDER BY NAMESIZE DESC;



```
mysql> SELECT ATTRACT_NAME, LENGTH(ATTRACT_NAME) AS NAMESIZE
    -> FROM ATTRACTION
    -> ORDER BY NAMESIZE DESC;
+-----------------+----------+
| ATTRACT_NAME    | NAMESIZE |
+-----------------+----------+
| SpinningTeacups |       15 |
| ThunderCoaster  |       14 |
| FlightToStars   |       13 |
| UnderSeaWord    |       12 |
| 3D-Lego_Show    |       12 |
| BlackHole2      |       10 |
| Carnival        |        8 |
| Ant-Trap        |        8 |
| GoldRush        |        8 |
| Pirates         |        7 |
| NULL            |     NULL |
+-----------------+----------+
11 rows in set (0.08 sec)

mysql>
```

Figure 65 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 MySQL, some conversions occur implicitly. For example, MySQL automatically converts numbers to strings when needed, and vice versa.

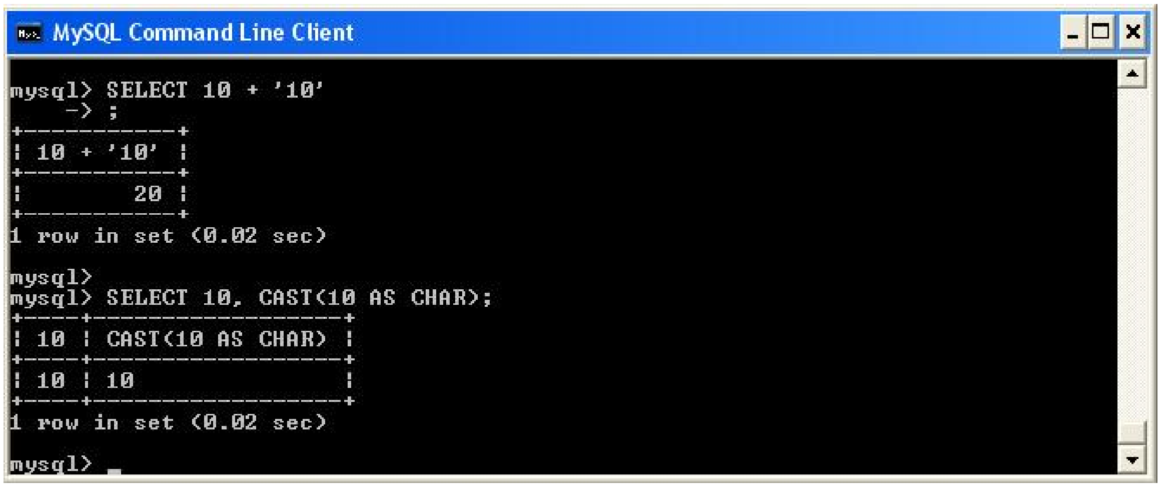
So if you enter the following query:

SELECT 10 + ‘10’

MySQL would give you an answer of 20 as it would automatically convert the string containing ‘10’ into the number 10 (see figure 66).

If you want to explicitly convert a number to a string then you can use either the **CAST** or CONCAT function. However MySQL 5.0 recommends only the CAST function is used. Let’s look at an example. The following query produces the output shown in Figure 66.

SELECT 10, CAST(10 AS CHAR);



```
mysql> SELECT 10 + '10'
    -> ;
+-----------+
| 10 + '10' |
+-----------+
|        20 |
+-----------+
1 row in set (0.02 sec)

mysql>
mysql> SELECT 10, CAST(10 AS CHAR);
+----+------------------+
| 10 | CAST(10 AS CHAR) |
+----+------------------+
| 10 | 10               |
+----+------------------+
1 row in set (0.02 sec)

mysql> _
```

Figure 66 Example of type conversions

Note

The MySQL Reference Manual 5.0 provides a set of rules that allow us to determine how the coversion will occur when using the CONVERT function on different data types.

IFNULL

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

IFNULL(expr1,expr2)

If expr1 is not NULL, IFNULL() returns expr1; otherwise it returns expr2. It is equivalent to Oracle's NVL function. It is useful for avoiding errors caused by incorrect calculation when one of the arguments is null.

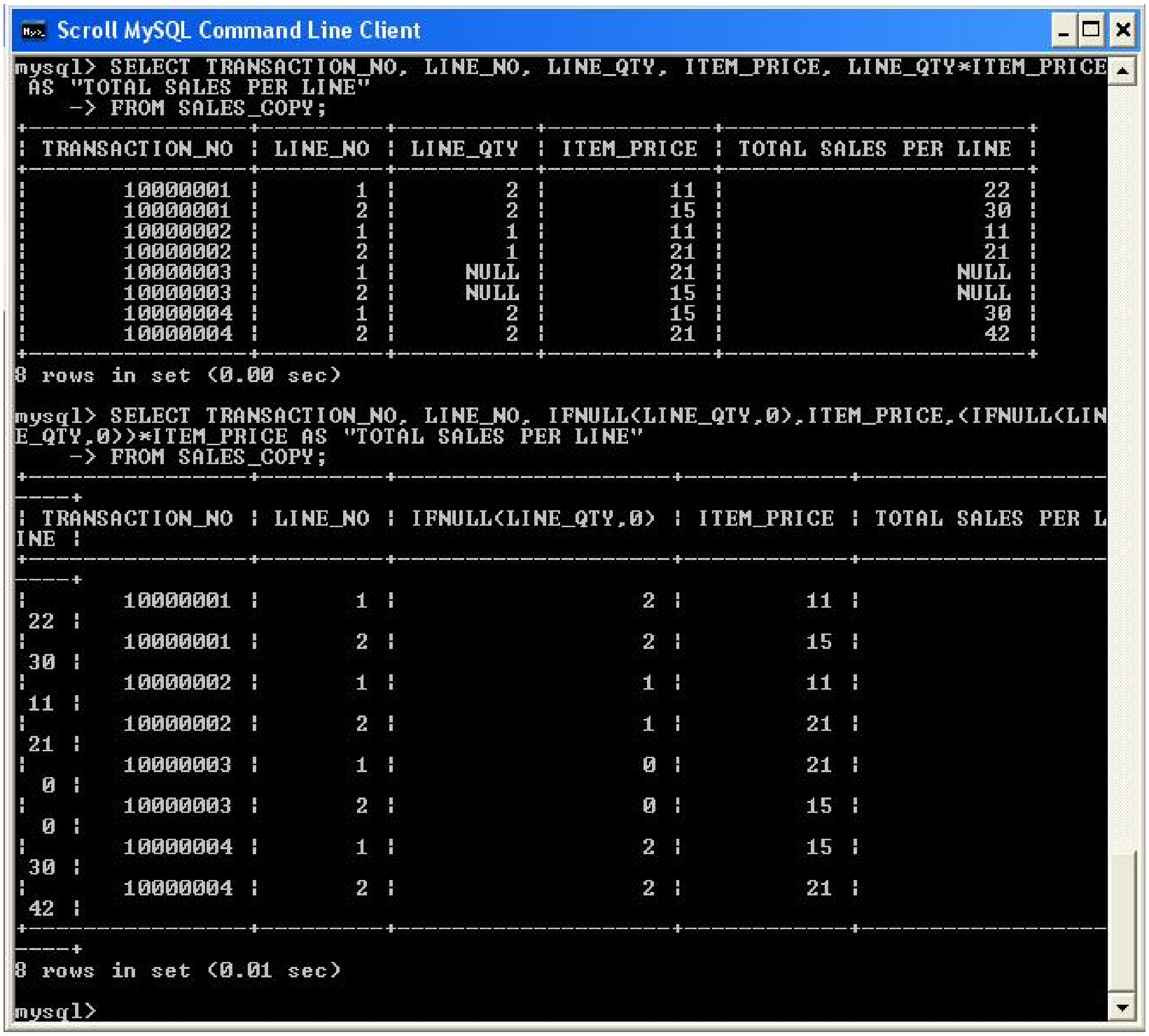
Task 7.11 Load and run the script sales_copy.sql which accompanies this lab guide. DESCRIBE the structure of the SALES_COPY table and examine the lack of constraints on this table. Write a query to view all the rows and notice that in some rows no values have been entered for LINE_QTY or LINE_PRICE. (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 IFNULL function and in two rows the calculation can not 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 IFNULL function and notice that the calculation has been achieved for all rows.

```
SELECT TRANSACTION_NO, LINE_NO,
IFNULL(LINE_QTY,0),ITEM_PRICE,(IFNULL(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 67.



```
mysql> SELECT TRANSACTION_NO, LINE_NO, LINE_QTY, ITEM_PRICE, LINE_QTY*ITEM_PRICE
 AS "TOTAL SALES PER LINE"
    -> FROM SALES_COPY;
+----------------+---------+----------+------------+----------------------+
| TRANSACTION_NO | LINE_NO | LINE_QTY | ITEM_PRICE | 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 |     NULL |         21 |                 NULL |
|       10000003 |       2 |     NULL |         15 |                 NULL |
|       10000004 |       1 |        2 |         15 |                   30 |
|       10000004 |       2 |        2 |         21 |                   42 |
+----------------+---------+----------+------------+----------------------+
8 rows in set (0.00 sec)

mysql> SELECT TRANSACTION_NO, LINE_NO, IFNULL(LINE_QTY,0),ITEM_PRICE,(IFNULL(LIN
E_QTY,0))*ITEM_PRICE AS "TOTAL SALES PER LINE"
    -> FROM SALES_COPY;
+----------------+---------+--------------------+------------+------------------
----+
| TRANSACTION_NO | LINE_NO | IFNULL(LINE_QTY,0) | ITEM_PRICE | TOTAL SALES PER L
INE |
+----------------+---------+--------------------+------------+------------------
----+
|       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 in set (0.01 sec)

mysql>
```

Figure 67 Illustration of the IFNULL function.

CASE

The CASE 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. There are two versions of the CASE function. The syntax of each is shown below.

CASE value WHEN [compare_value] THEN result [WHEN [compare_value] THEN result ...] [ELSE result] END

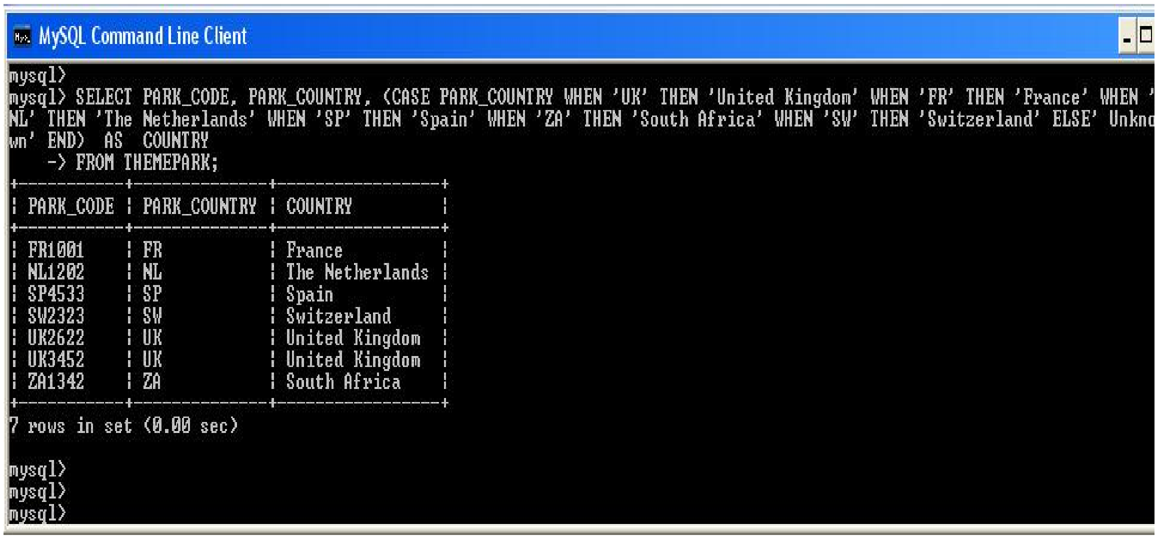
CASE WHEN [condition] THEN result [WHEN [condition] THEN result ...] [ELSE result] END

The first version returns the result where value=compare_value. The second version returns the result for the first condition that is true. If there was no matching result value, the result after ELSE is returned, or NULL if there is no ELSE part.

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 68.

SELECT PARK_CODE, PARK_COUNTRY, (CASE PARK_COUNTRY WHEN 'UK' THEN 'United Kingdom' WHEN 'FR' THEN 'France' WHEN 'NL' THEN 'The Netherlands' WHEN 'SP' THEN 'Spain' WHEN 'ZA' THEN 'South Africa' WHEN 'SW' THEN 'Switzerland' ELSE' Unknown' END) AS COUNTRY

FROM THEMEPARK;



```
mysql>
mysql> SELECT PARK_CODE, PARK_COUNTRY, (CASE PARK_COUNTRY WHEN 'UK' THEN 'United Kingdom' WHEN 'FR' THEN 'France' WHEN '
NL' THEN 'The Netherlands' WHEN 'SP' THEN 'Spain' WHEN 'ZA' THEN 'South Africa' WHEN 'SW' THEN 'Switzerland' ELSE' Unkno
wn' END)  AS  COUNTRY
    -> 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 in set (0.00 sec)

mysql>
mysql>
mysql>
```

Figure 68 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, and the first two characters of first name in lower case. 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.

Lab 8: 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, which 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's 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.

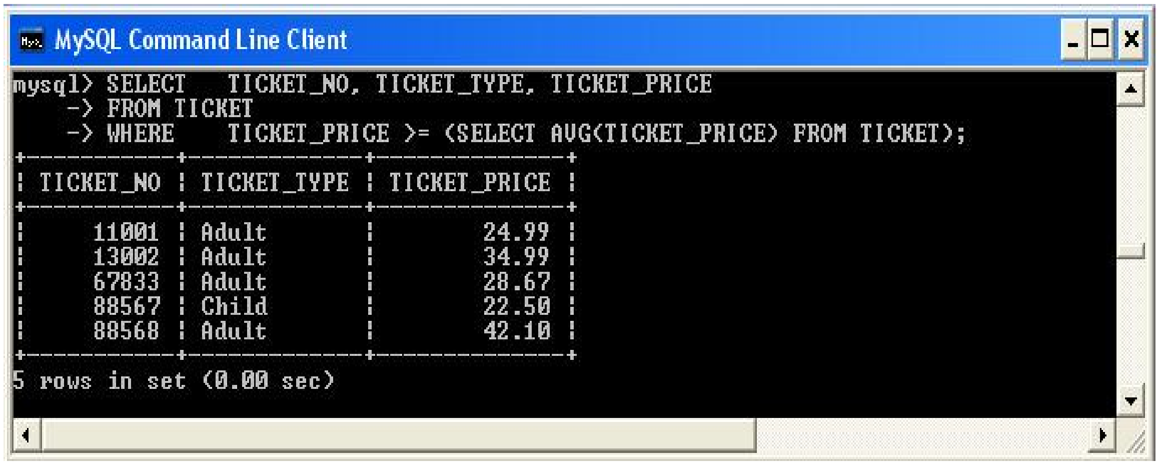
> ***Note***
>
> You can also read more about subqueries in Chapter 9 Advanced SQL.

8.1 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 69.

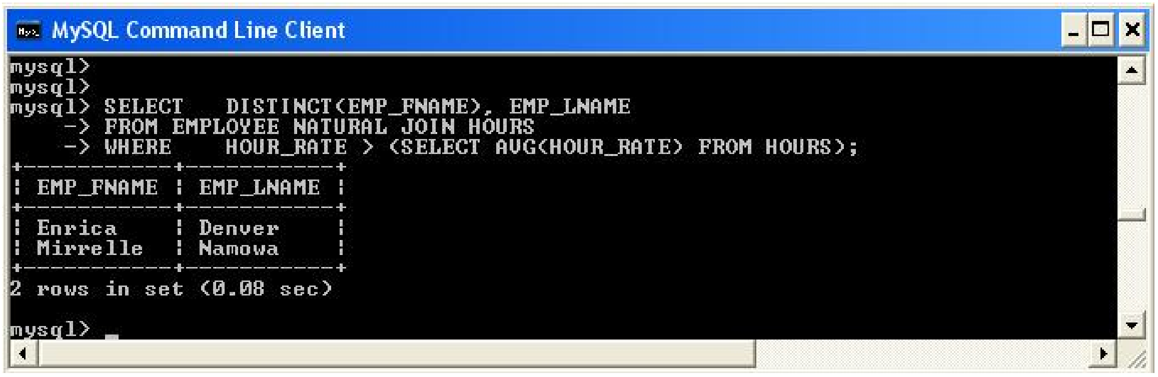


```
mysql> SELECT   TICKET_NO, TICKET_TYPE, TICKET_PRICE
    -> FROM TICKET
    -> WHERE    TICKET_PRICE >= (SELECT AVG(TICKET_PRICE) FROM TICKET);
+-----------+-------------+--------------+
| TICKET_NO | TICKET_TYPE | TICKET_PRICE |
+-----------+-------------+--------------+
|     11001 | Adult       |        24.99 |
|     13002 | Adult       |        34.99 |
|     67833 | Adult       |        28.67 |
|     88567 | Child       |        22.50 |
|     88568 | Adult       |        42.10 |
+-----------+-------------+--------------+
5 rows in set (0.00 sec)
```

Figure 69 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 8.1 Write a query that displays the first name, 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 70.



```
mysql>
mysql>
mysql> SELECT   DISTINCT(EMP_FNAME), EMP_LNAME
    -> FROM EMPLOYEE NATURAL JOIN HOURS
    -> WHERE    HOUR_RATE > (SELECT AVG(HOUR_RATE) FROM HOURS);
+-----------+-----------+
| EMP_FNAME | EMP_LNAME |
+-----------+-----------+
| Enrica    | Denver    |
| Mirrelle  | Namowa    |
+-----------+-----------+
2 rows in set (0.08 sec)

mysql> _
```

Figure 70 Output for task 8.1

8.2 IN Subqueries

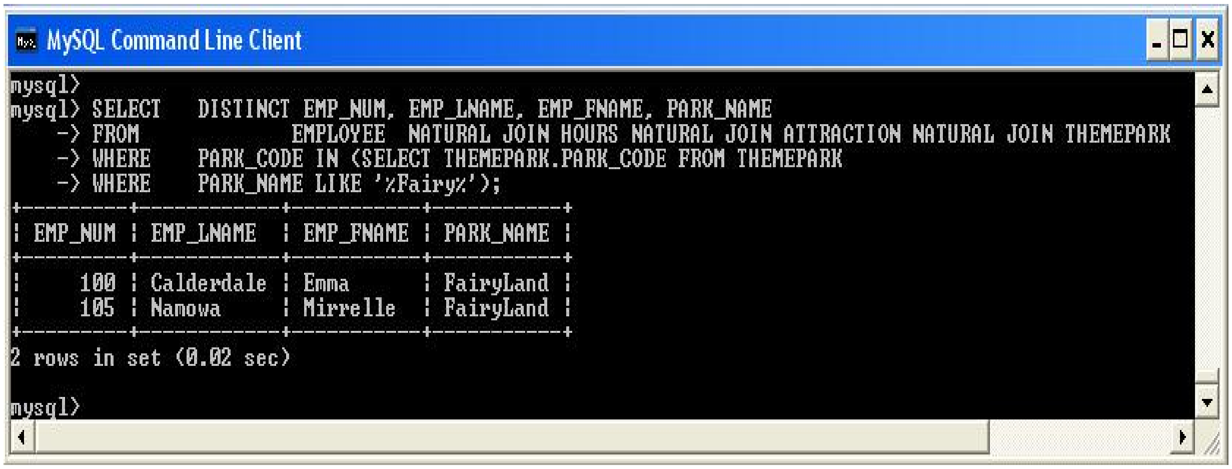
The following query displays all employees who work in a Theme Park that has the word ‘Fairy’ in its name. As there are a number of different Theme Parks that match this criteria you need to compare the PARK_CODE not to one park code (single value), but to a list of park codes. When you want to compare 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 a Theme Park.

SELECT DISTINCT EMP_NUM, EMP_LNAME, EMP_FNAME, PARK_NAME

FROM EMPLOYEE NATURAL JOIN HOURS NATURAL JOIN ATTRACTION NATURAL JOIN THEMEPARK

WHERE PARK_CODE IN (SELECT THEMEPARK.PARK_CODE FROM THEMEPARK WHERE PARK_NAME LIKE ‘%Fairy%’);

The result of that query is shown in Figure 71.



```
mysql>
mysql> SELECT   DISTINCT EMP_NUM, EMP_LNAME, EMP_FNAME, PARK_NAME
    -> FROM             EMPLOYEE  NATURAL JOIN HOURS NATURAL JOIN ATTRACTION NATURAL JOIN THEMEPARK
    -> WHERE    PARK_CODE IN (SELECT THEMEPARK.PARK_CODE FROM THEMEPARK
    -> WHERE    PARK_NAME LIKE '%Fairy%');
+---------+------------+-----------+-----------+
| EMP_NUM | EMP_LNAME  | EMP_FNAME | PARK_NAME |
+---------+------------+-----------+-----------+
|     100 | Calderdale | Emma      | FairyLand |
|     105 | Namowa     | Mirrelle  | FairyLand |
+---------+------------+-----------+-----------+
2 rows in set (0.02 sec)

mysql>
```

Figure 71 Employees who work in a Theme Park LIKE 'Fairy'.

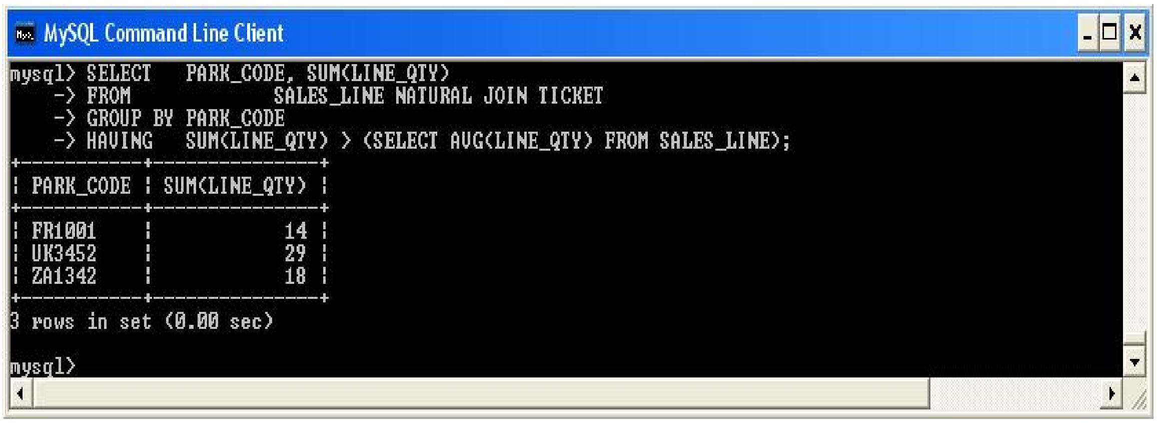
Task 8.2 Enter and execute the above query and compare your output with that shown in Figure 71.

8.3 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);
```

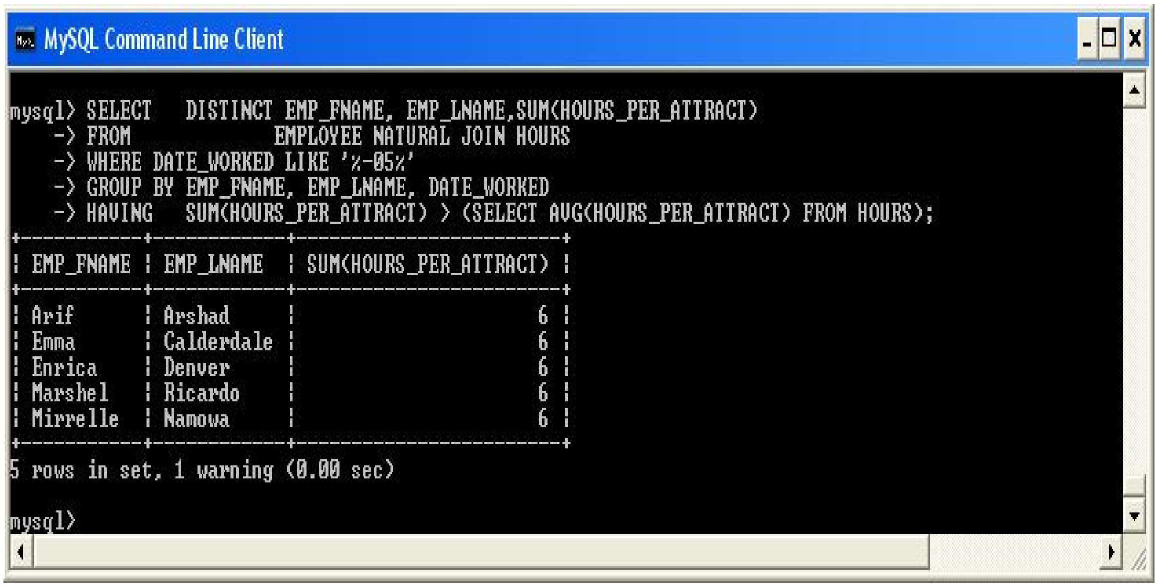
The result of that query is shown in Figure 72.



```
mysql> 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);
+-----------+---------------+
| PARK_CODE | SUM(LINE_QTY) |
+-----------+---------------+
| FR1001    |            14 |
| UK3452    |            29 |
| ZA1342    |            18 |
+-----------+---------------+
3 rows in set (0.00 sec)

mysql>
```

Figure 72 PARK_CODES where tickets are selling above average.

Task 8.3 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 that the average number of hours in total during May 2007. Your output should match that shown in Figure 73.



```
mysql> SELECT   DISTINCT EMP_FNAME, EMP_LNAME,SUM(HOURS_PER_ATTRACT)
    -> FROM             EMPLOYEE NATURAL JOIN HOURS
    -> WHERE DATE_WORKED LIKE '%-05%'
    -> GROUP BY EMP_FNAME, EMP_LNAME, DATE_WORKED
    -> HAVING   SUM(HOURS_PER_ATTRACT) > (SELECT AVG(HOURS_PER_ATTRACT) FROM HOURS);
+-----------+-----------+-----------------------+
| EMP_FNAME | EMP_LNAME | SUM(HOURS_PER_ATTRACT) |
+-----------+-----------+-----------------------+
| Arif      | Arshad    |                      6 |
| Emma      | Calderdale |                     6 |
| Enrica    | Denver    |                      6 |
| Marshel   | Ricardo   |                      6 |
| Mirrelle  | Namowa    |                      6 |
+-----------+-----------+-----------------------+
5 rows in set, 1 warning (0.00 sec)

mysql>
```

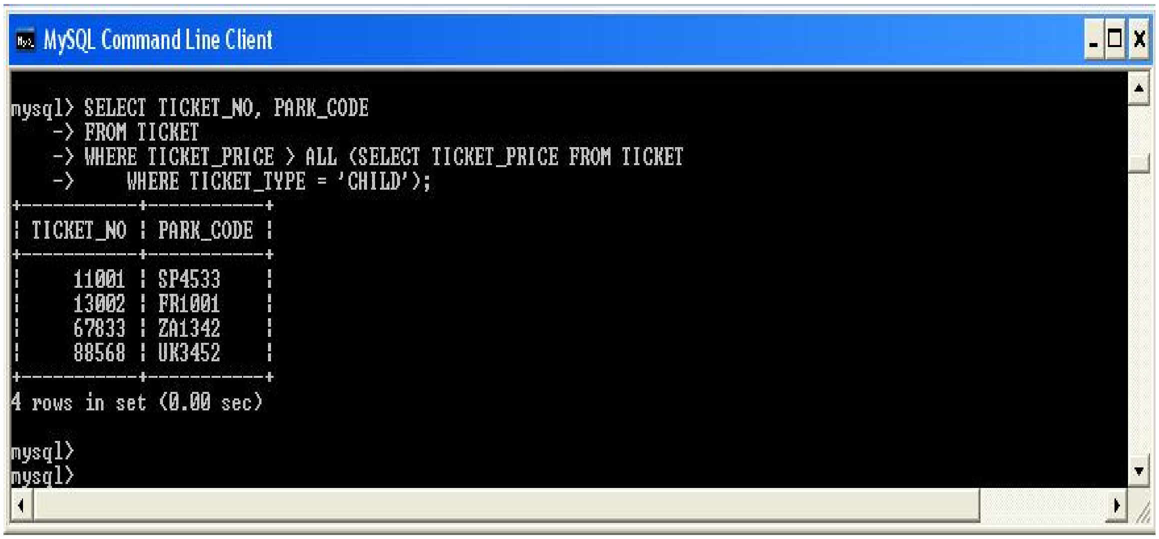
Figure 73 Output for task 8.3

8.4 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 that 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 74.



```
mysql> SELECT TICKET_NO, PARK_CODE
    -> FROM TICKET
    -> WHERE TICKET_PRICE > ALL (SELECT TICKET_PRICE FROM TICKET
    ->     WHERE TICKET_TYPE = 'CHILD');
+-----------+-----------+
| TICKET_NO | PARK_CODE |
+-----------+-----------+
|     11001 | SP4533    |
|     13002 | FR1001    |
|     67833 | ZA1342    |
|     88568 | UK3452    |
+-----------+-----------+
4 rows in set (0.00 sec)

mysql>
mysql>
```

Figure 74 Example of ALL.

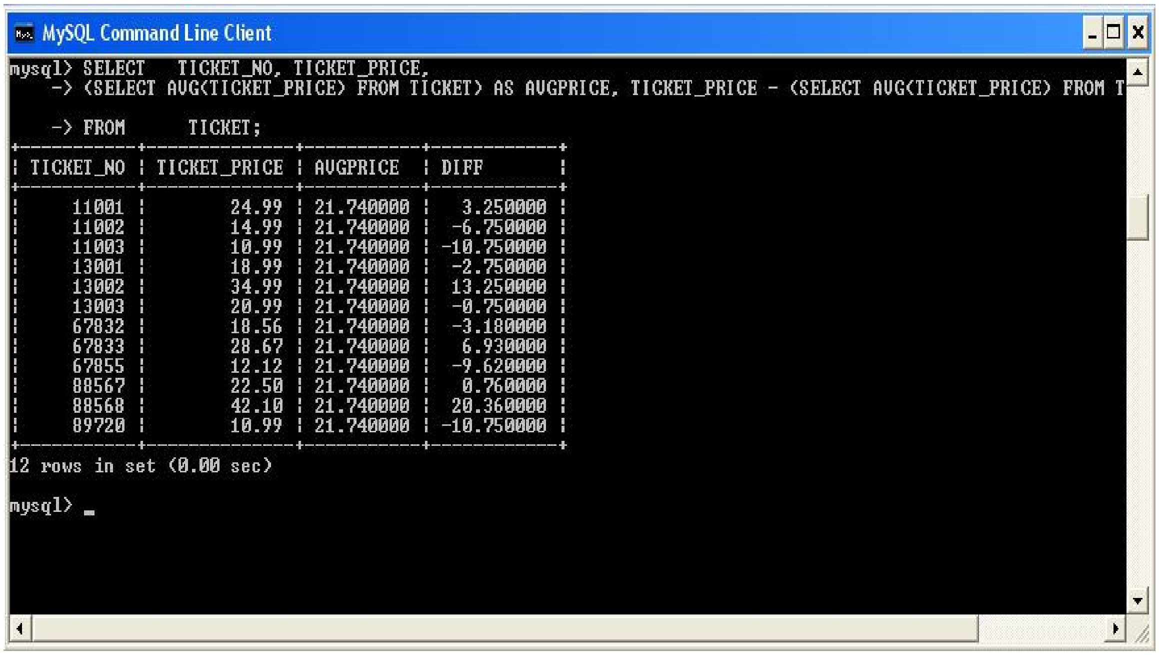
This query is a typical example of a nested query. The use of the ALL operator allows you to compare a single value (TICKET_PRICE) with a list of values returned by the nested query, using a comparison operator other than equals. For a row to appear in the result set, it has to meet the criterion TICKET_PRICE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

8.5 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 tickets' 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 75.



```
mysql> SELECT   TICKET_NO, TICKET_PRICE,
    -> (SELECT AVG(TICKET_PRICE) FROM TICKET) AS AVGPRICE, TICKET_PRICE - (SELECT AVG(TICKET_PRICE) FROM T

    -> FROM     TICKET;
+-----------+--------------+-----------+------------+
| TICKET_NO | TICKET_PRICE | AVGPRICE  | DIFF       |
+-----------+--------------+-----------+------------+
|     11001 |        24.99 | 21.740000 |   3.250000 |
|     11002 |        14.99 | 21.740000 |  -6.750000 |
|     11003 |        10.99 | 21.740000 | -10.750000 |
|     13001 |        18.99 | 21.740000 |  -2.750000 |
|     13002 |        34.99 | 21.740000 |  13.250000 |
|     13003 |        20.99 | 21.740000 |  -0.750000 |
|     67832 |        18.56 | 21.740000 |  -3.180000 |
|     67833 |        28.67 | 21.740000 |   6.930000 |
|     67855 |        12.12 | 21.740000 |  -9.620000 |
|     88567 |        22.50 | 21.740000 |   0.760000 |
|     88568 |        42.10 | 21.740000 |  20.360000 |
|     89720 |        10.99 | 21.740000 | -10.750000 |
+-----------+--------------+-----------+------------+
12 rows in set (0.00 sec)

mysql> _
```

Figure 75 Displaying the difference in ticket prices.

This inline query output returns one single value (the average ticket's price) and that the 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 8.4 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'.

8.6 Correlated Subqueries

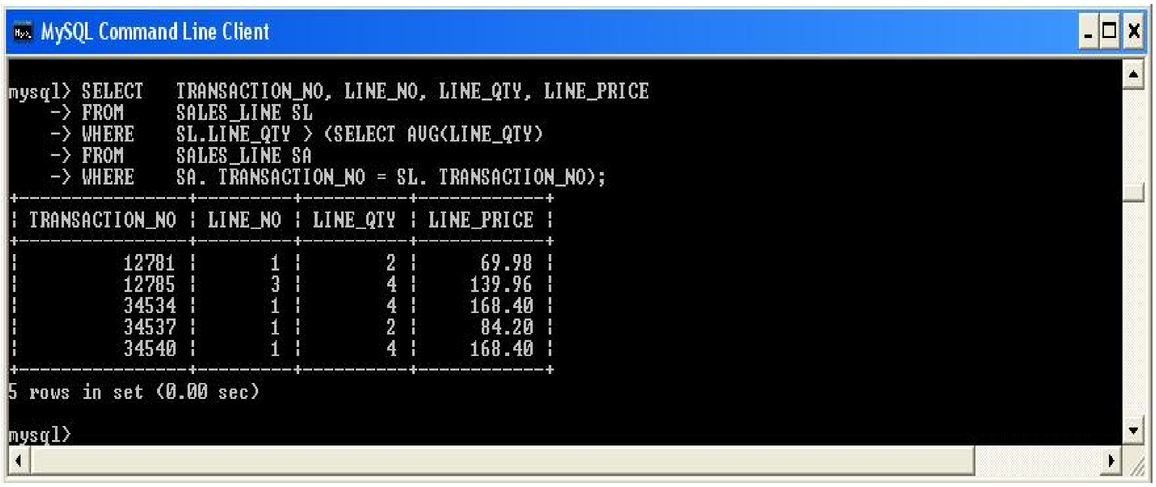
A correlated subquery is a subquery that executes once for each row in the outer query.

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.

That 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 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);
```



```
mysql> 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);
+----------------+---------+----------+------------+
| TRANSACTION_NO | LINE_NO | LINE_QTY | LINE_PRICE |
+----------------+---------+----------+------------+
|          12781 |       1 |        2 |      69.98 |
|          12785 |       3 |        4 |     139.96 |
|          34534 |       1 |        4 |     168.40 |
|          34537 |       1 |        2 |      84.20 |
|          34540 |       1 |        4 |     168.40 |
+----------------+---------+----------+------------+
5 rows in set (0.00 sec)

mysql>
```

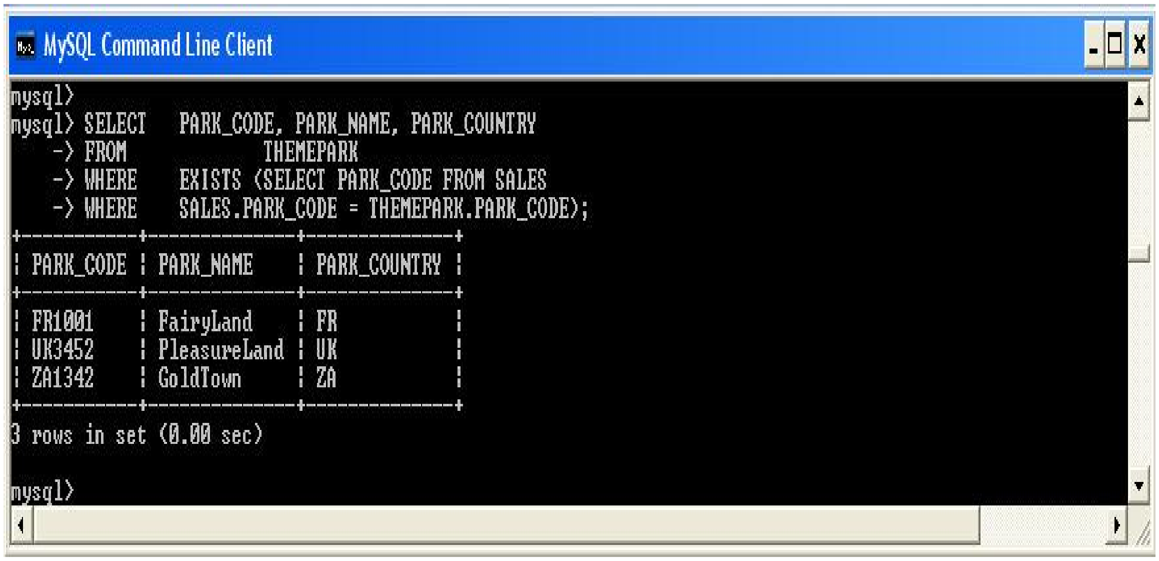
Figure 76 Example of a correlated subquery

As you examine the output shown in figure 76, 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 77.



```
mysql>
mysql> SELECT   PARK_CODE, PARK_NAME, PARK_COUNTRY
    -> FROM             THEMEPARK
    -> WHERE    EXISTS (SELECT PARK_CODE FROM SALES
    -> WHERE    SALES.PARK_CODE = THEMEPARK.PARK_CODE);
+-----------+--------------+--------------+
| PARK_CODE | PARK_NAME    | PARK_COUNTRY |
+-----------+--------------+--------------+
| FR1001    | FairyLand    | FR           |
| UK3452    | PleasureLand | UK           |
| ZA1342    | GoldTown     | ZA           |
+-----------+--------------+--------------+
3 rows in set (0.00 sec)

mysql>
```

Figure 77 Example of correlated subqueries

Task 8.5 Type in and execute the two correlated subqueries in this section and check your output against that shown in figures 76 and 77.

Task 8.6 Modify the second query you entered in task 8.5 to display all the theme parks where there have been no recorded tickets sales recently.

Lab 9: Views

The learning objectives of this lab are to

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

9.1 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*

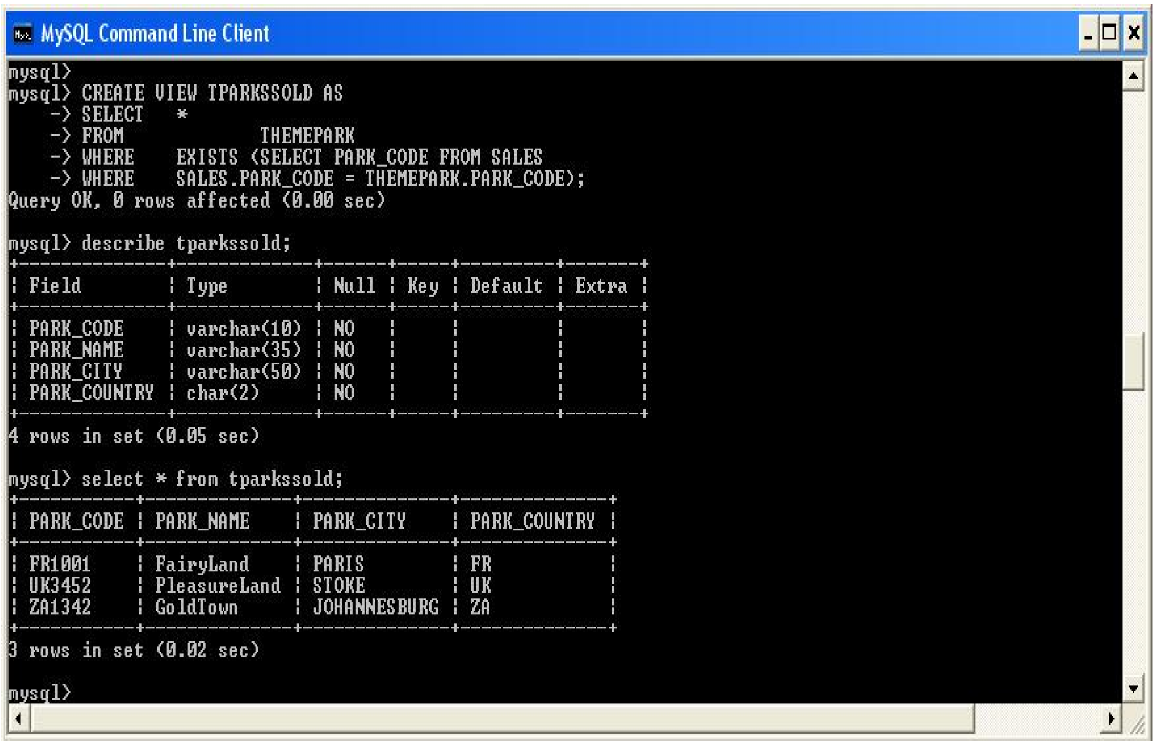
The CREATE VIEW statement is a data definition command that stores the subquery specification—the SELECT statement used to generate the virtual table—in the data dictionary. For example, to create a view of only those Theme Parks were tickets have been sold you would do so as follows:

```
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 78.



```
mysql>
mysql> CREATE VIEW TPARKSSOLD AS
    -> SELECT   *
    -> FROM             THEMEPARK
    -> WHERE    EXISTS (SELECT PARK_CODE FROM SALES
    -> WHERE    SALES.PARK_CODE = THEMEPARK.PARK_CODE);
Query OK, 0 rows affected (0.00 sec)

mysql> describe tparkssold;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| PARK_CODE    | varchar(10) | NO   |     |         |       |
| PARK_NAME    | varchar(35) | NO   |     |         |       |
| PARK_CITY    | varchar(50) | NO   |     |         |       |
| PARK_COUNTRY | char(2)     | NO   |     |         |       |
+--------------+-------------+------+-----+---------+-------+
4 rows in set (0.05 sec)

mysql> select * from tparkssold;
+-----------+--------------+--------------+--------------+
| PARK_CODE | PARK_NAME    | PARK_CITY    | PARK_COUNTRY |
+-----------+--------------+--------------+--------------+
| FR1001    | FairyLand    | PARIS        | FR           |
| UK3452    | PleasureLand | STOKE        | UK           |
| ZA1342    | GoldTown     | JOHANNESBURG | ZA           |
+-----------+--------------+--------------+--------------+
3 rows in set (0.02 sec)

mysql>
```

Figure 78 Creating the TPARKSSOLD view.

Task 9.1 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;

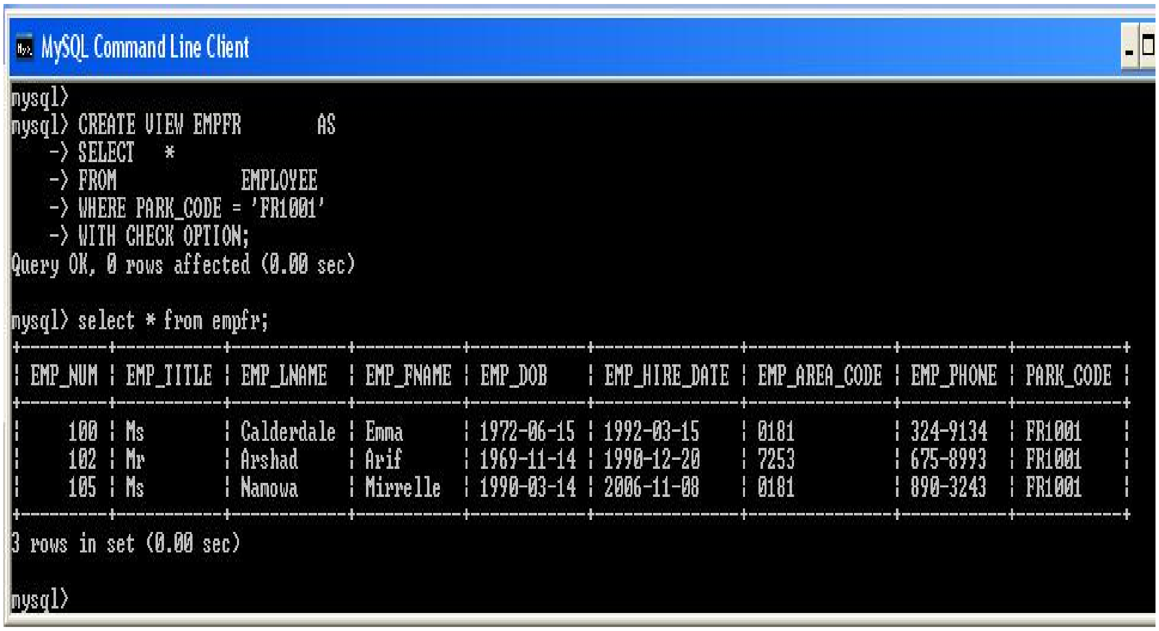
Task 9.2 Create a view called TICKET_SALES which contains details of the min, max and average sales at each Theme Park. The name of the theme park should also be displayed. Hint 1: you will need to join three tables. Hint 2: You will need to give the columns in the query that use the functions an alias. Once you have created your view, write a query to display the contents.

Task 9.3 Add your view TICKET_SALES and the associated DROP command to your themepark.sql scrip you created in lab 2.

9.2 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 79.

```
CREATE VIEW EMPFR       AS

SELECT         *

FROM           EMPLOYEE

WHERE PARK_CODE = 'FR1001'

WITH CHECK OPTION;
```



```
mysql>
mysql> CREATE VIEW EMPFR        AS
    -> SELECT   *
    -> FROM             EMPLOYEE
    -> WHERE PARK_CODE = 'FR1001'
    -> WITH CHECK OPTION;
Query OK, 0 rows affected (0.00 sec)

mysql> select * from empfr;
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
| EMP_NUM | EMP_TITLE | EMP_LNAME  | EMP_FNAME | EMP_DOB    | EMP_HIRE_DATE | EMP_AREA_CODE | EMP_PHONE | PARK_CODE |
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
|     100 | Ms        | Calderdale | Emma      | 1972-06-15 | 1992-03-15    | 0181          | 324-9134  | FR1001    |
|     102 | Mr        | Arshad     | Arif      | 1969-11-14 | 1990-12-20    | 7253          | 675-8993  | FR1001    |
|     105 | Ms        | Namowa     | Mirrelle  | 1990-03-14 | 2006-11-08    | 0181          | 890-3243  | FR1001    |
+---------+-----------+------------+-----------+------------+---------------+---------------+-----------+-----------+
3 rows in set (0.00 sec)

mysql>
```

Figure 79 Creating the EMPFR view

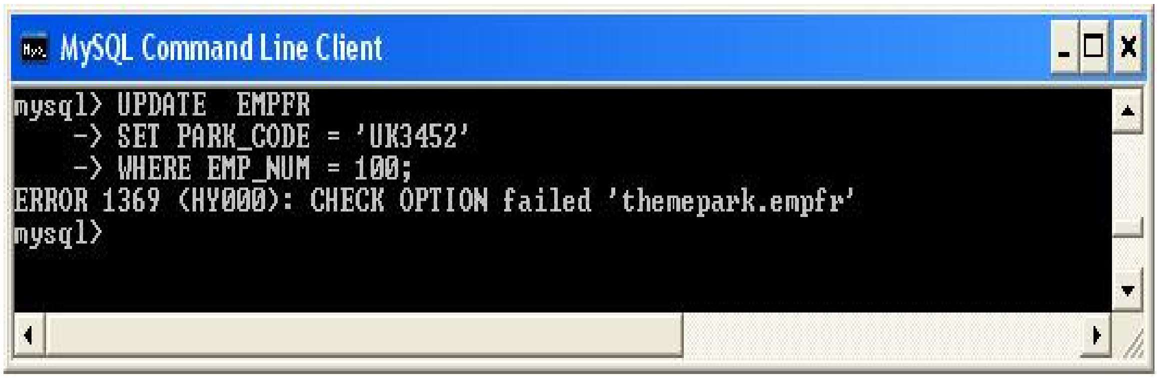
So for example if employee 'Emma Caulderdale' was to leave the park and move to park 'UK3452', we would want to update her information with the following query:

UPDATE EMPFR

SET PARK_CODE = 'UK3452'

WHERE EMP_NUM = 100;

However running this update gives the errors shown in Figure 80. This is because if the update was to occur, the view would no longer be able to see this employee.



```
mysql> UPDATE  EMPFR
    -> SET PARK_CODE = 'UK3452'
    -> WHERE EMP_NUM = 100;
ERROR 1369 (HY000): CHECK OPTION failed 'themepark.empfr'
mysql>
```

Figure 80 Creating the EMPFR view

Task 9.4 Create the view EMPFR and tray and update the Theme Park that employee number 101 works in.

Task 9.5. 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 9.6 Remove the EMPFR view.

9.4 Exercises

E9.1 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.

E9.2 Check that the view works, by displaying its contents.

E9.3 Using your view EMP_DETAILS, write a query that displays all employee first and last names and the park names.

E9.4 Remove the view EMPDETAILS.

CONCLUSION

You have now reached the end of this MySQL 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

Dyer, R. *MySQL in a Nutshell* 2e, OReilly; Rev Ed edition, (2008)

Reese, G. *MySQL Pocket Reference* 2e, O'Reilly, (2007)

WEB SITES

MySQL	**http://www.mysql.com/**
MySQL 5.0 Reference Manual	**http://dev.mysql.com/doc/refman/5.0/en/index.html**
MySQL Development Zone	**http://dev.mysql.com/**

BUGS

To report a bug in MySQL visit the site http://bugs.mysql.com/