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;
```

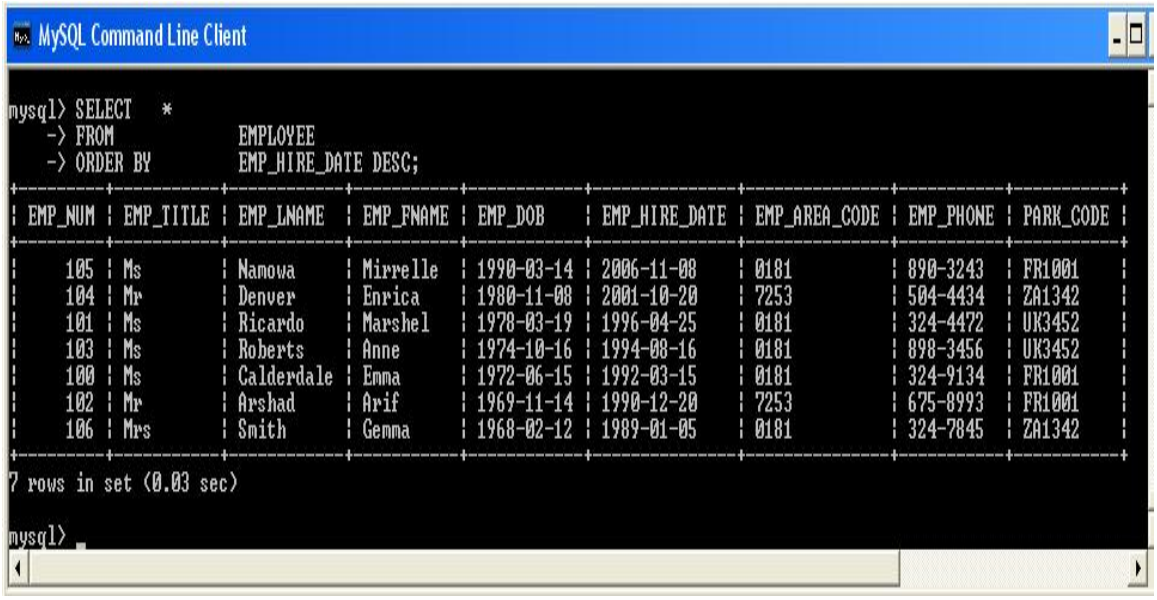


Figure 34: Displaying all employees in descending order of EMP_HIRE_DATE.

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;
```

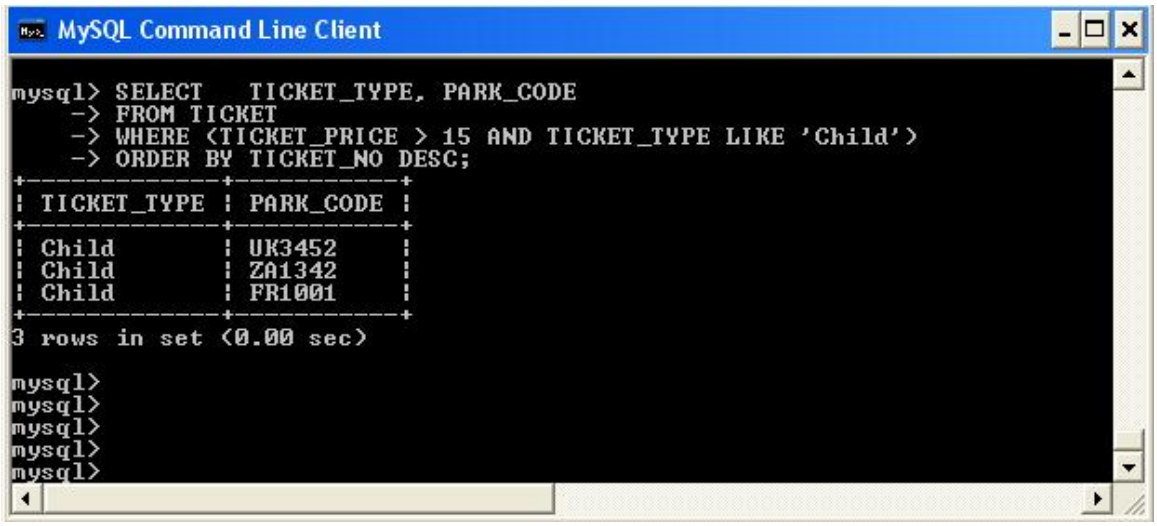


Figure 35: Query results for Task 5.2.

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.

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.

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;
```

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.

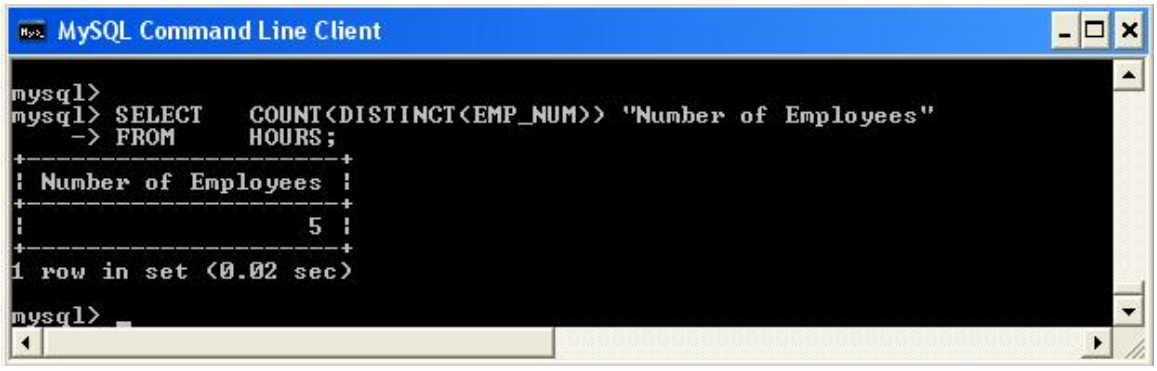


Figure 39: Query output for Task 5.3

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;
```

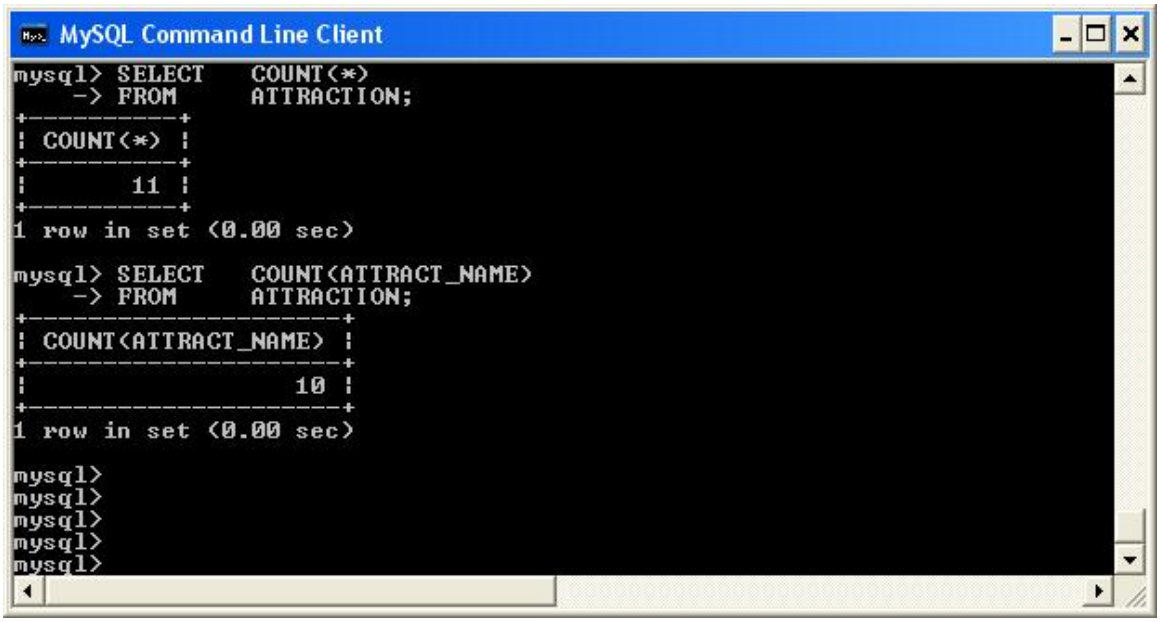


Figure 40: Examples of using the COUNT function

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;

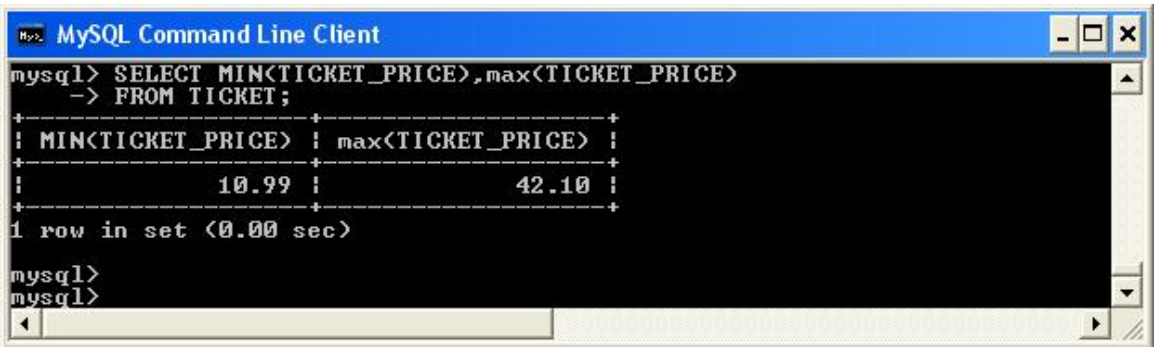


Figure 41: Examples of using the MIN and MAX functions

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;
```

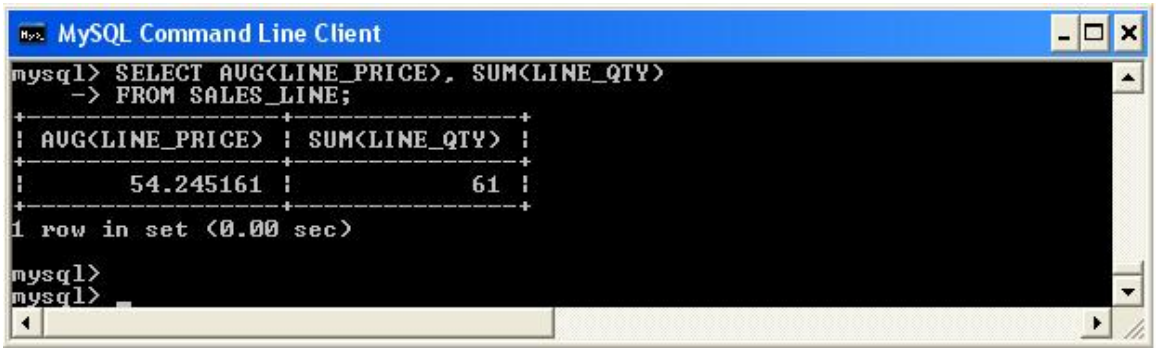


Figure 42: Example showing the AVG and SUM functions

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;
```

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;
```

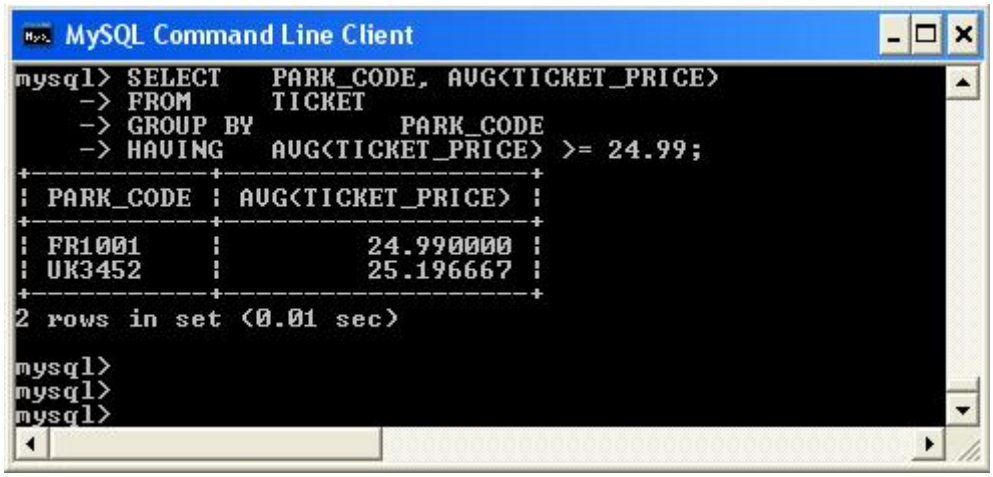


Figure 44: Example of the HAVING clause

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.

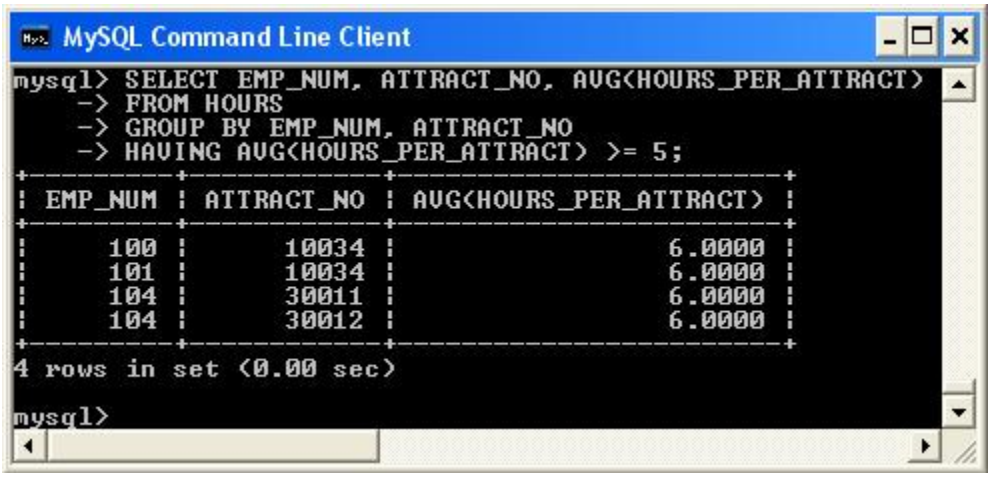


Figure 45: Query output for Task 5.8

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;
```

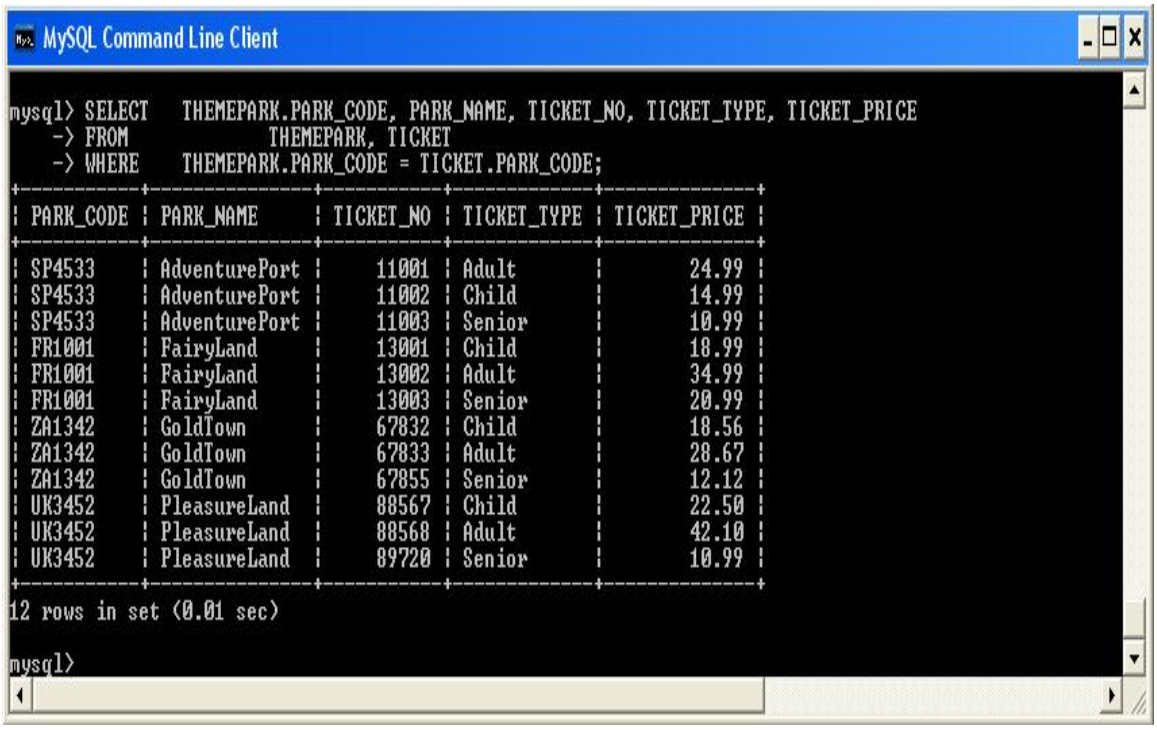


Figure 46: Natural Join between THEMEPARK and TICKET tables

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;
```

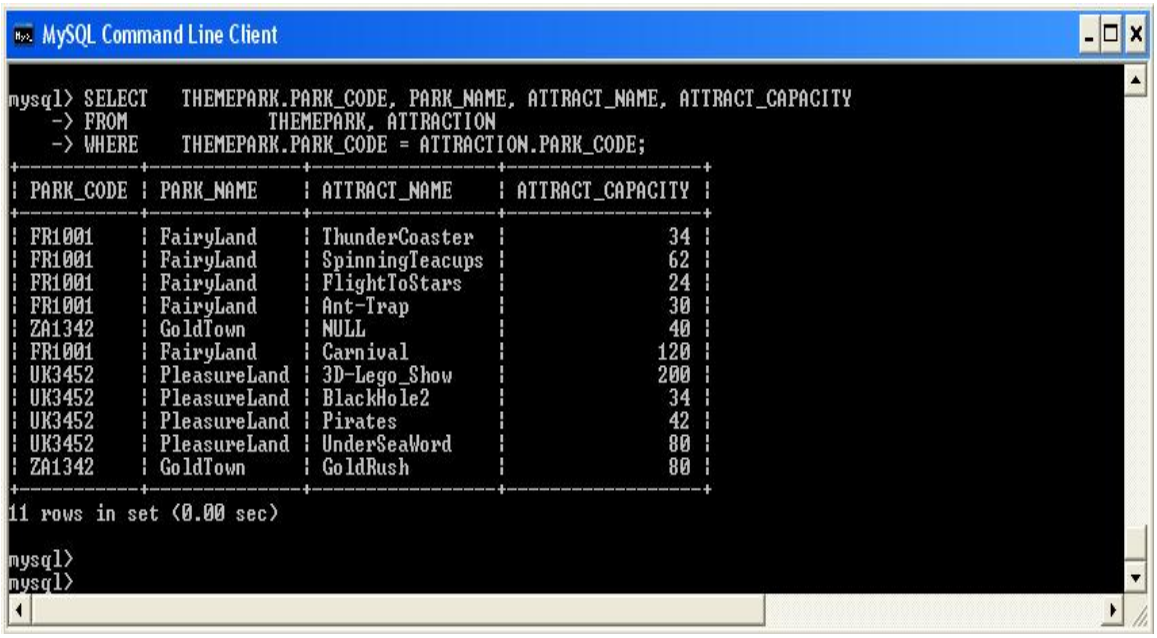


Figure 47: Query output for task 6.1

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.

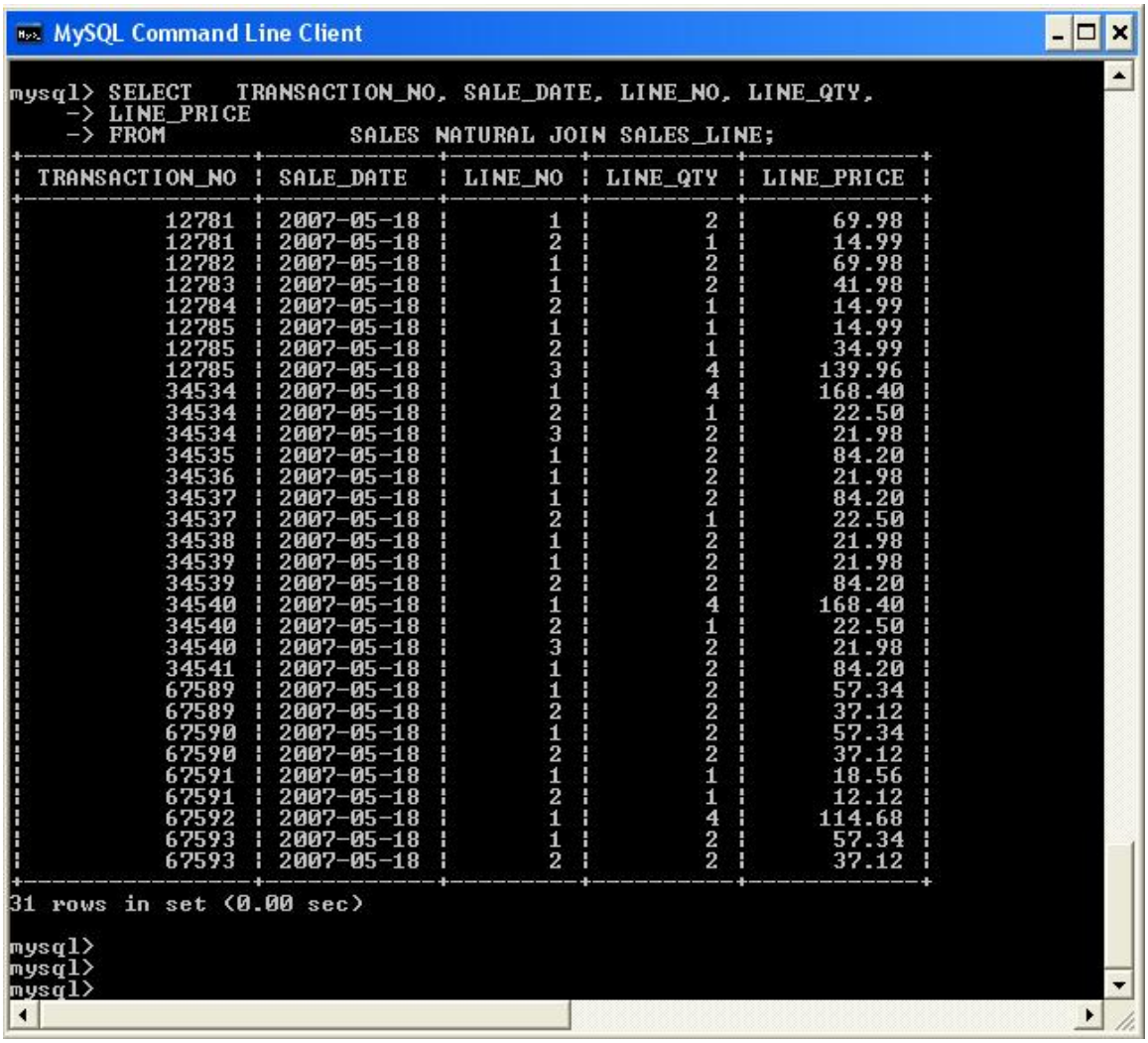


Figure 48: Results of SALES NATURAL JOIN SALES_LINE;

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.

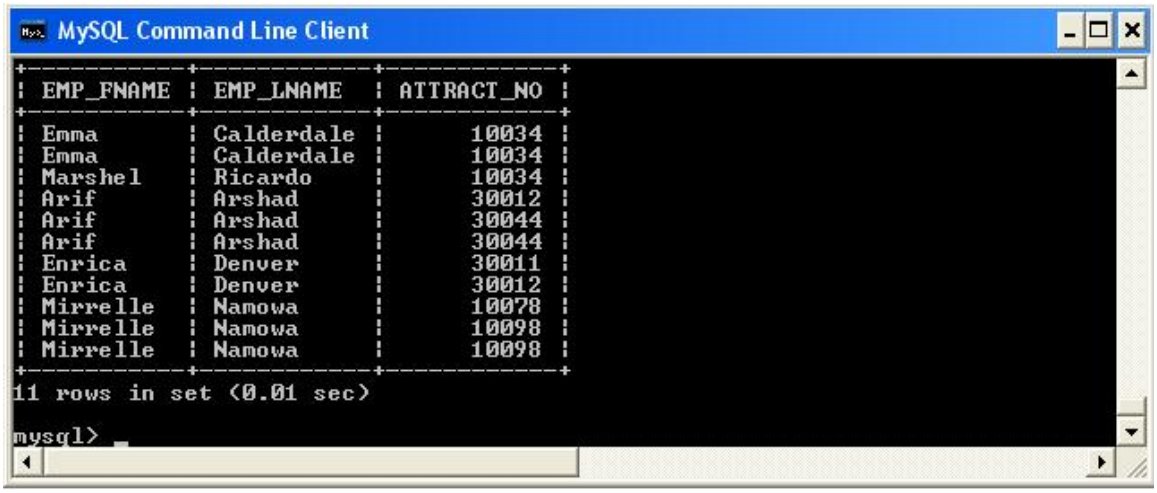


Figure 49: Query results for Task 6.3

6.5 Join USING

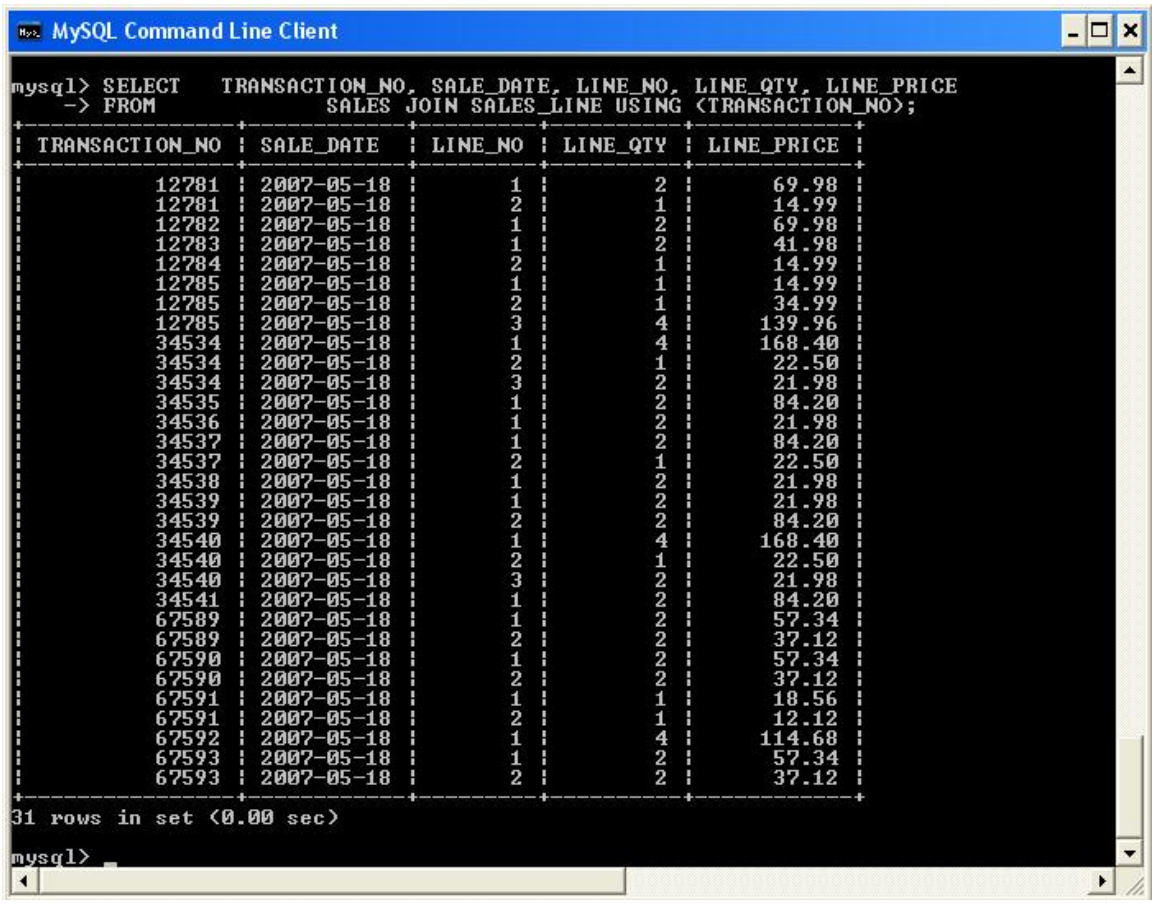
A second way to express a join is through the USING keyword. That query returns only the rows with matching values in the column indicated in the USING clause—and that column must exist in both tables. The syntax is:

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.



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

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.

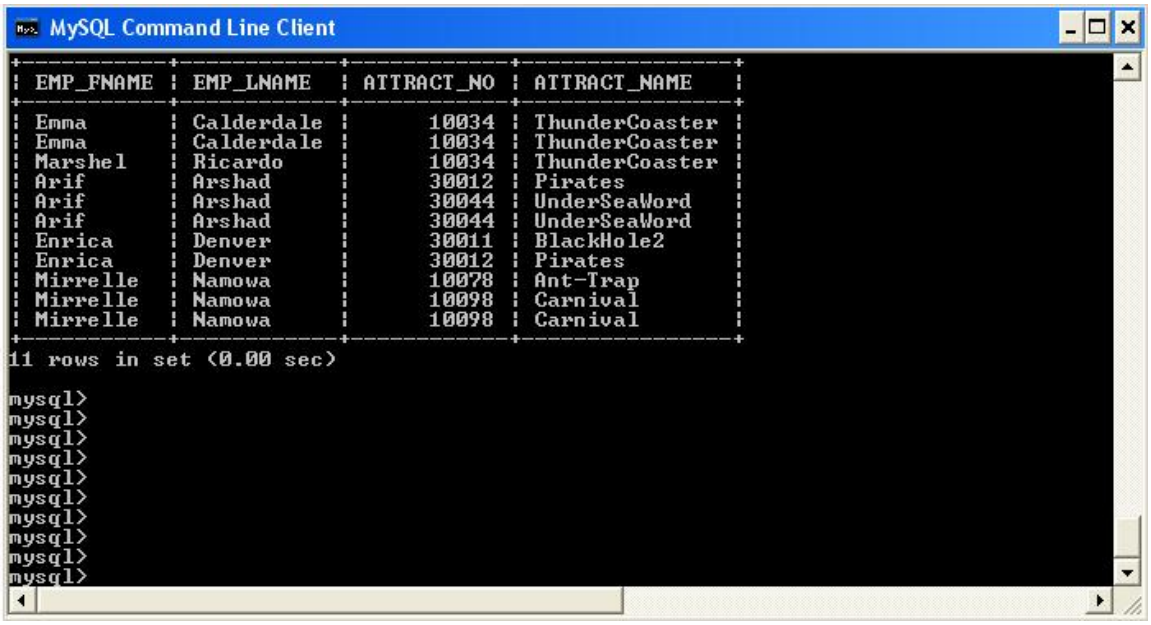


Figure 51: Query results for Task 6.4

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;
```

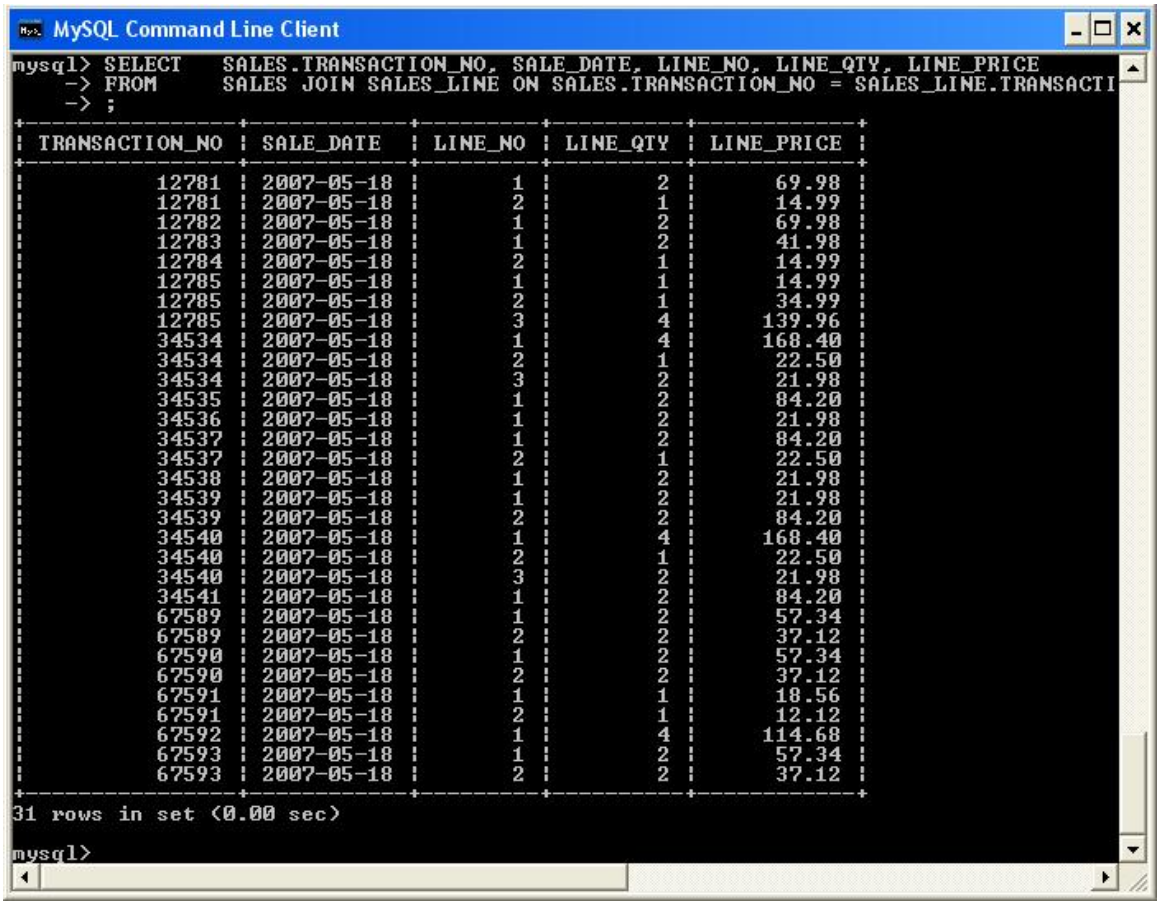


Figure 52: Query results for SALES JOIN SALES_LINE ON

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.

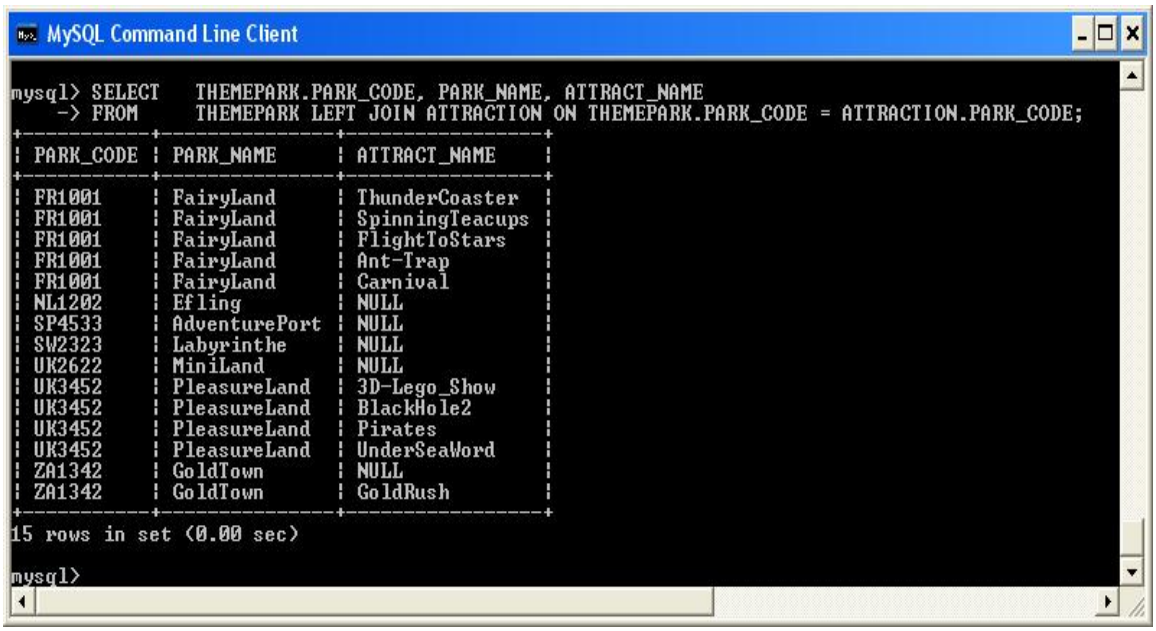


Figure 53: LEFT OUTER JOIN example

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.

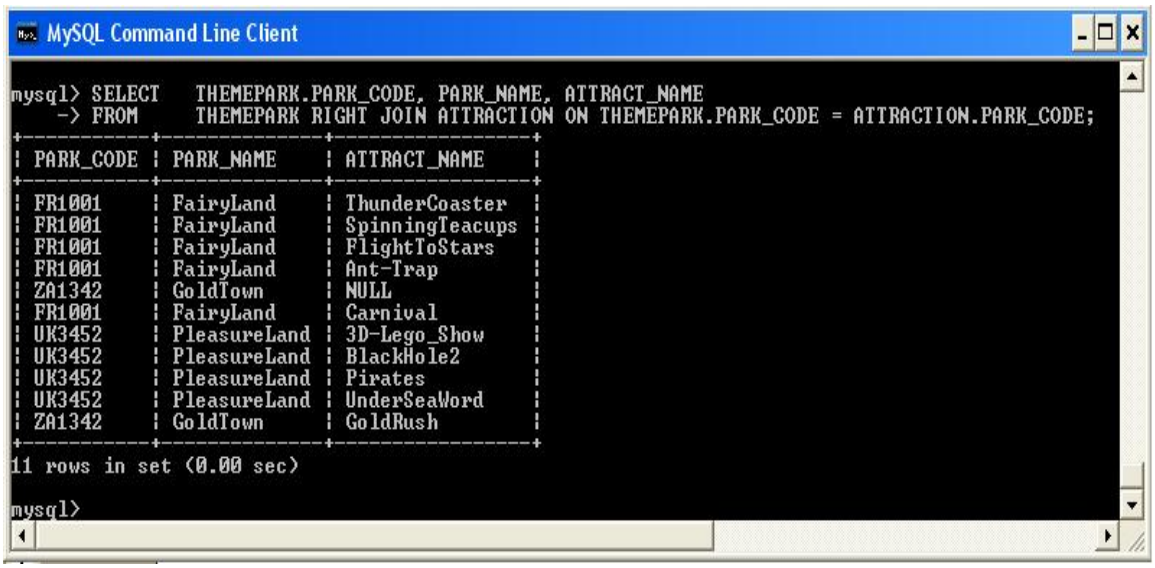


Figure 54: RIGHT OUTER JOIN example

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’).