Data Query Language (DQL)

DQL consists of only one statement: the SELECT statement. However, the SELECT statement is the most complex of all SQL statements because of the sheer number of clauses and options, not to mention the ability to write some queries in two or three different ways. I present the SELECT statement using a series of examples, starting with a very simple query and working on to more complicated ones. These are the same queries that I use in the SQL demonstration videos. I have numbered the steps for easy reference.

1. The SELECT statement has two required clauses: SELECT, which names the operation and lists the desired columns, and the FROM clause that names the source table(s) and/or view(s). In this example, we use the asterisk (*) to select all columns:

   ```
   SELECT * FROM HORSE;
   ```

2. Using SELECT * is not a best practice because your results will change whenever columns are added or removed from the table. It is always better to supply the column names in the form of a comma-separated list.

   ```
   SELECT HORSE_ID, HORSE_NAME, OWNER_ID, BIRTH_DATE
     FROM HORSE;
   ```

3. If you want the rows in the query results (also known as the *result set*) to be in any particular sequence, you can use the ORDER BY clause to specify the desired sequence in the form of a comma-separated list of column names. The default is ascending sequence (ASC), but in this example we want the BIRTH_DATE to be in descending (DESC) sequence.

   ```
   SELECT HORSE_ID, HORSE_NAME, OWNER_ID, BIRTH_DATE
     FROM HORSE
     ORDER BY BIRTH_DATE DESC;
   ```

4. Thus far we have been selecting all the rows from the source table (HORSE). You can use the WHERE clause to specify an expression that filters rows. The following query selects only mares (females).

   ```
   SELECT HORSE_ID, HORSE_NAME, OWNER_ID, BIRTH_DATE, GENDER
      FROM HORSE
      WHERE GENDER = 'M';
   ```

5. The WHERE clause can contain compound expressions using logical operators (AND, OR, NOT, etc.) to connect them. The WHERE clause is evaluated for each row found in the source table(s) with a result of true, false, or unknown (null values evaluate to unknown). Here we add another comparison (predicate) to the previous query to find only mares born after 1/1/2015. Note that mares born on 1/1/2015 will not be included because we used the "greater than" (>) comparison operator. If we want to include them, we could change it to "greater than or equal to" (>=).

   ```
   SELECT HORSE_ID, HORSE_NAME, OWNER_ID, BIRTH_DATE, GENDER
     FROM HORSE
   ```

```
 WHERE GENDER='M' OR BIRTH_DATE>'2010-01-01';
```

6. The NOT logical operator can be used to reverse the result of a predicate (true becomes false and false becomes true). This example finds all horses that are not mares.

```
SELECT HORSE_ID, HORSE_NAME, OWNER_ID, BIRTH_DATE, GENDER
  FROM HORSE
 WHERE NOT GENDER = 'M';
```

7. The previous query can be rewritten (simplified) using the not equal (<>) operator, as shown here.

```
SELECT HORSE_ID, HORSE_NAME, OWNER_ID, BIRTH_DATE, GENDER
  FROM HORSE
 WHERE GENDER <> 'M';
```

8. This query uses both the AND and OR logical operators. However, without parentheses to control the order in which the predicates are compared, you have to remember that AND is processed before OR in order to understand the results.

```
SELECT TRACK_ID, RACE_NUMBER, RACE_DATE, HORSE_ID,
       PLACE, LENGTHS_BEHIND
  FROM RACE_RESULT
 WHERE HORSE_ID LIKE 'KY%'
   AND PLACE <= 2 OR LENGTHS_BEHIND <= 1.5
ORDER BY TRACK_ID, RACE_NUMBER, RACE_DATE, HORSE_ID;
```

9. Here is a modified form of the previous query with parentheses added so that the OR is evaluated before the AND. The result set will be quite different.

```
SELECT TRACK_ID, RACE_NUMBER, RACE_DATE, HORSE_ID,
       PLACE, LENGTHS_BEHIND
  FROM RACE_RESULT
 WHERE HORSE_ID LIKE 'KY%'
   AND (PLACE <= 2 OR LENGTHS_BEHIND <= 1.5)
 ORDER BY TRACK_ID, RACE_NUMBER, RACE_DATE, HORSE_ID;
```

10. Suppose we have been asked to find all the horses that are owned by an owner who lives in the city of Winona. One way to do this is to run one query to find the IDs of the owners who live in Winona and then key those IDs into a query that searches the HORSE table. Here are those queries:

```
SELECT OWNER_ID FROM OWNER WHERE CITY = 'Winona';
```

```
SELECT HORSE_ID, HORSE_NAME, OWNER_ID
  FROM HORSE
 WHERE OWNER_ID = '00011' OR OWNER_ID = '00850'
    OR OWNER_ID = '02001' OR OWNER_ID = '02003';
```

11. We can shorten the previous statement quite a bit by using the IN predicate, which evaluates to true if the value of the named column matches and of the times in the list of values provided. Here is an example.

```
SELECT HORSE_ID, HORSE_NAME, OWNER_ID
  FROM HORSE
 WHERE OWNER_ID IN ('00011', '00850', '02001','02003');
```

12. However, the query in the previous example has a major issue – if an owner moves in or out of Winona, the results will be incorrect from that point forward. The solution is to use a subquery to find the OWNER_ID values. In this case, we can use a *non-correlated* subquery, which runs only once (before the outer query).

```
SELECT HORSE_ID, HORSE_NAME, OWNER_ID
  FROM HORSE
 WHERE OWNER_ID IN
      (SELECT OWNER_ID FROM OWNER WHERE CITY = 'Winona');
```

13. If we want to list horse names along with their owner names, we must join the HORSE and OWNER tables. However, whenever you join tables, you must tell the DBMS how you want rows matched, or it will join each row in one table with every row in the other table, a result known as a *Cartesian product*. The following example has no join matching logic, so it produces a Cartesian product. There are 16 rows in the HORSE table, and every horse must have an OWNER per business rules, so you would expect 16 rows in the result set. Instead we see 144 rows (16 horses times 9 owners), and it looks like every horse is owned by every owner.

```
SELECT HORSE_NAME, OWNER.OWNER_ID, OWNER_LAST_NAME,
       OWNER_FIRST_NAME
  FROM HORSE, OWNER;
```

14. The original way to specify join matching logic (join predicates) in SQL was to include them in the WHERE clause. Here is the previous query with the join predicate added. Note that we have to qualify the OWNER_ID column names with table names (table_name.column_name) because the column name is ambiguous (a column named OWNER_ID appears in both the HORSE and OWNER tables).

```
SELECT HORSE_NAME, OWNER.OWNER_ID, OWNER_LAST_NAME,
       OWNER_FIRST_NAME
  FROM HORSE, OWNER
 WHERE HORSE.OWNER_ID = OWNER.OWNER_ID;
```

15. Table names can be fairly long, and thus you get tired of typing them repeatedly in SQL statements. A simple way around that is to use a table alias, specified immediately after the table name in the FROM clause. Optionally, the keyword 'AS' can be placed between the table name and the alias. The only caveat is that when an alias is specified in the FROM clause, you must use the alias for all table references in the query – you can no longer use the table name

itself. You are free to use whatever alias you wish, but they should be short lest you defeat the purpose of the alias. Many SQL programmers simply use single letters from the alphabet, beginning with 'A'. Here is the previous query with aliases A and B added.

```
SELECT A.HORSE_NAME, A.HORSE_ID, B.OWNER_LAST_NAME,
       B.OWNER_FIRST_NAME
  FROM HORSE A, OWNER B
 WHERE A.OWNER_ID = B.OWNER_ID;
```

16. The JOIN keyword provides a more flexible way to write joins and it has the added benefit of moving join predicates out of the WHERE clause, thereby simplifying it. Here is the previous query rewritten to use the JOIN clause.

```
SELECT A.HORSE_NAME, A.HORSE_ID, B.OWNER_LAST_NAME,
       B.OWNER_FIRST_NAME
  FROM HORSE A
  JOIN OWNER B
    ON A.OWNER_ID = B.OWNER_ID;
```

17. Now let's look at a recursive join (joining rows in a table to other rows in the same table). In this example, we want to list the father (sire) and mother (dam) for each horse in the HORSE table. The SIRE_HORSE_ID contains the value of the HORSE_ID of the horse's father (if known) and the DAM_HORSE_ID contains the value of the HORSE_ID for the horse's mother. Recursive joins (also called self-joins) take a lot of getting used to. You may want to look at the contents of the HORSE table and visually find the row for each SIRE_HORSE_ID and DAM_HORSE_ID value. Also note that with queries involving self-joins, ever column in the source table is ambiguous, so you must always qualify the column names. Here is the query:

```
SELECT H.HORSE_ID, H.HORSE_NAME,
       S.HORSE_NAME AS SIRE, D.HORSE_NAME AS DAM
  FROM HORSE H
  JOIN HORSE S
    ON H.SIRE_HORSE_ID = S.HORSE_ID
  JOIN HORSE D
    ON H.DAM_HORSE_ID  = D.HORSE_ID
 ORDER BY H.HORSE_NAME;
```

18. If you ran the previous query, you would no doubt notice that it returns only 7 rows, and yet there are 16 horses in the HORSE table. The reason for this is that only 7 of the horses have values for the SIRE_HORSE_ID and DAM_HORSE_ID columns. We used an inner join in the previous query, so non-matched rows are discarded. Here is the same query updated to perform left outer joins. The result set will contain all 16 rows regardless of which rows have matching sire and/or dam rows.

```
SELECT H.HORSE_ID, H.HORSE_NAME AS HORSE,
       S.HORSE_NAME AS SIRE, D.HORSE_NAME AS DAM
  FROM HORSE H
  LEFT JOIN HORSE S
         ON H.SIRE_HORSE_ID = S.HORSE_ID
  LEFT JOIN HORSE D
         ON H.DAM_HORSE_ID = D.HORSE_ID;
```

19. Let's change gears and look at some SQL functions. As in mathematics, an SQL *function* returns a single value each time it is executed. For example, you can use the UPPER function on character data to shift all alphabetic letters to upper case. Each DBMS product comes with a set of functions, some of which are commonly used in all SQL implementations, but many of which are proprietary. In this example, the function is used in the SELECT clause, so it is executed once per row.

```
SELECT UPPER(HORSE_NAME)
  FROM HORSE;
```

20. Some functions aggregate rows together. By default, these functions will aggregate all the rows returned by the query into a single row in the result set. We'll look at other options shortly. In the following query, we use the SUM, AVG, MIN, and MAX functions on the PURSE column to return the total, average, minimum and maximum winnings for all horses.

```
SELECT SUM(PURSE),
       AVG(PURSE),
       MIN(PURSE),
       MAX(PURSE)
  FROM RACE_RESULT;
```

21. You may have noticed that when you use expressions such as functions to form columns in the result set, the DBMS assigns a column name for the result based on its own internal rules. You can always provide your own column name by assigning an alias right after the expression that forms the column, and you can optionally include the keyword AS for clarity. Here is the previous query with column aliases added.

```
SELECT SUM(PURSE) AS TOTAL_PURSE,
       AVG(PURSE) AS AVERAGE_PURSE,
       MIN(PURSE) AS SMALLEST_PURSE,
       MAX(PURSE) AS LARGEST_PURSE
  FROM RACE_RESULT;
```

22. The COUNT function provides a count of the rows in a result set. It has two forms. COUNT(*) counts rows regardless of data values in the rows, which COUNT(column_name) counts only non-null values in the named column. Here is an example using COUNT(*) that counts the number of rows containing a value of PURSE that exceeds 10000.

```
SELECT COUNT(*) AS NUM_PURSES_OVER_10000
  FROM RACE_RESULT
 WHERE PURSE > 10000;
```

23. You can also use a subquery in the WHERE clause to derive a value that is used in a predicate. In this example, we want to find the number of purses that are above the average of all purses.

```
SELECT COUNT(*) AS NUM_PURSES_ABOVE_AVG
  FROM RACE_RESULT
 WHERE PURSE > (SELECT AVG(PURSE) FROM RACE_RESULT);
```

24. Suppose we want to find the total earnings (purse) for each horse represented in the RACE_RESULT table. The GROUP BY clause directs the DBMS to form groups of rows for each value of the column(s) named in the clause, returning one row in the result set for each group. If we were to write the query without the GROUP BY clause, it would look like this:

```
SELECT HORSE_ID, SUM(PURSE), COUNT(*) AS NUM_RACES
  FROM RACE_RESULT;
```

MySQL will actually run the above query, returning what I have always thought is an odd result showing the first value of HORSE_ID it finds, followed by the grand total of the purse and the count of the number of rows in the RACE_RESULT table. All of the other relational DBMSs with which I am familiar (MS Access, Oracle, DB2, Sql Server, Sybase, and Vertica) return an error when you try to run a statement like this one. The rule to follow is whenever the SELECT clause contains an aggregate function, then every column in the result set must either be formed using an aggregate function or be included in the GROUP BY clause list.

Here is the preceding query with the GROUP BY clause added:

```
SELECT HORSE_ID, SUM(PURSE), COUNT(*) AS NUM_RACES
  FROM RACE_RESULT
 GROUP BY HORSE_ID;
```

25. Now suppose you want to filter out any horses that ran in only one race. You cannot do this simply using the WHERE clause because it is applied before grouping, and what you need is to filter on the count of the number of races for each horse. The HAVING clause does exactly what we need here – it has syntax like the WHERE clause, but it is applied to each group after grouping has taken place. Here is the modified query:

```
SELECT HORSE_ID, SUM(PURSE) AS TOTAL_PURSE, COUNT(*) AS NUM_RACES
  FROM RACE_RESULT
 GROUP BY HORSE_ID
 HAVING COUNT(*) > 1;
```

26. Finally, we need a query that finds the average purse earned per race by horses that ran more than one race. We could use the AVG function for this, but just to show how flexible column expressions in the SELECT clause can be, I am calculating it by dividing the sum of the PURSE column by the COUNT of the number of races. I nested that calculation inside a ROUND function that rounds the result to two decimal places. As you can see, SQL SELECT statements are very powerful.

```
SELECT HORSE_ID, ROUND(SUM(PURSE) / COUNT(*), 2) AS AVG_PURSE
  FROM RACE_RESULT
```

```
GROUP BY HORSE_ID
HAVING COUNT(*) > 1;
```