**Using String Patterns and Ranges**

In this video, we will learn about some advanced techniques in retrieving data from a relational database table. At the end of this lesson, you will be able to describe how to simplify a SELECT statement by using string patterns, ranges or sets of values. The main purpose of a database management system is not just to store the data, but also facilitate retrieval of the data. In its simplest form, a SELECT statement is select star from table name. Based on a simplified library database model and the table Book, SELECT star from Book gives a result set of four rows. All the data rows for all columns in the table Book are displayed or you can retrieve a subset of columns for example, just two columns from the table book such as Book\_ID and Title. Or you can restrict the result set by using the WHERE clause. For example, you can select the title of the book whose Book\_ID is B1. But what if we don't know exactly what value to specify in the WHERE clause? The WHERE clause always requires a predicate, which is a condition that evaluates to true, false or unknown. But what if we don't know exactly what value the predicate is? For example, what if we can't remember the name of the author, but we remember that their first name starts with R? In a relational database, we can use string patterns to search data rows that match this condition. Let's look at some examples of using string patterns. If we can't remember the name of the author, but we remember that their name starts with R, we use the WHERE clause with the like predicate. The like predicate is used in a WHERE clause to search for a pattern in a column. The percent sign is used to define missing letters. The percent sign can be placed before the pattern, after the pattern, or both before and after the pattern. In this example, we use the percent sign after the pattern, which is the letter R. The percent sign is called a wildcard character. A wildcard character is used to substitute other characters. So, if we can't remember the name of the author, but we can remember that their first name starts with the letter R, we add the like predicate to the WHERE clause. For example, select first name from author, where firstname like 'R%'. This will return all rows in the author table whose author's first name starts with the letter R. And here is the result set. Two rows a return for authors Raul and Rav. What if we wanted to retrieve the list of books whose number of pages is more than 290, but less than 300. We could write the SELECT statement like this, specifying the WHERE clause as, where pages is greater than or equal to 290, and pages is less than or equal to 300. But in a relational database, we can use a range of numbers to specify the same condition. Instead of using the comparison operators greater than or equal to, we use the comparison operator 'between and.' 'Between and' compares two values. The values in the range are inclusive. In this case,we rewrite the query to specify the WHERE clause as where pages between 290 and 300. The result set is the same, but the SELECT statement is easier and quicker to write. In some cases, there are data values that cannot be grouped under ranges. For example, if we want to know which countries the authors are from. If we wanted to retrieve authors from Australia or Brazil, we could write the SELECT statement with the WHERE clause repeating the two country values. However, what if we want to retrieve authors from Canada, India, and China? The WHERE clause would become very long repeatedly listing the required country conditions. Instead, we can use the IN operator. The IN operator allows us to specify a set of values in a WHERE clause. This operator takes a list of expressions to compare against. In this case the countries Australia or Brazil. Now you can describe how to simplify a SELECT statement by using string patterns, ranges, or sets of values.

**Sorting Result Sets**

In this video, we will learn about some advanced techniques in retrieving data from a relational database table and sorting how the result set displays. At the end of this lesson, you will be able to describe how to sort the result set by either ascending or descending order and explain how to indicate which column to use for the sorting order. The main purpose of a database management system is not just to store the data, but also facilitate retrieval of the data. In its simplest form, a select statement is select \* from table name. Based on our simplified library database model, in the table book, select \* from book gives a result set of four rows. All the data rows for all columns in the table book are displayed. We can choose to list the book titles only as shown in this example, select title from book. However, the order does not seem to be in any order. Displaying the results set in alphabetical order would make the result set more convenient. To do this, we use the "order by" clause. To display the result set in alphabetical order, we add the order by clause to the select statement. The order by clause is used in a query to sort the result set by a specified column. In this example, we have used order by on the column title to sort the result set. By default, the result set is sorted in ascending order. In this example, the result set is sorted in alphabetical order by book title. To sort in descending order, use the key word" desc." The result set is now sorted according to the column specified, which is title, and is sorted in descending order. Notice the order of the first three rows. The first three words of the title are the same, so the sorting starts from the point where the characters differ. Another way of specifying the sort column is to indicate the column sequence number. In this example, select title, pages from book order by two, indicates the column sequence number in the query for the sorting order. Instead of specifying the column name pages, the number two is used. In the select statement, the second column specified in the column list is pages, so the sort order is based on the values in the pages column. In this case, the pages column indicates the number of pages in the book. As you can see, the result set is in ascending order by number of pages. Now you can describe how to sort the result set by either ascending or descending order, and explain how to indicate which column to use for the sorting order.

**Grouping Result Sets**

Hello and welcome to Grouping Select Statement Result Sets. In this video, we will learn about some advanced techniques in retrieving data from a relational database table, and sorting, and grouping how the results set displays. At the end of this lesson, you will be able to explain how to eliminate duplicates from a result set and describe how to further restrict a result set. At times, a select statement result set can contain duplicate values. Based on our simplified library database model, in the author table example, the country column lists the two-letter country code of the author's country. If we select just the country column, we get a list of all of the countries. For example, select country from author order by 1. The order by clause sorts the result set. This result set lists the countries the authors belong to, sorted alphabetically by country. In this case, the result set displays 20 rows, one row for each of the 20 authors. But some of the authors come from the same country, so the result set contains duplicates. However, all we need is a list of countries the authors come from. So in this case, duplicates do not make sense. To eliminate duplicates, we use the keyword distinct. Using the keyword "distinct" reduces the result set to just six rows. But what if we wanted to also know how many authors come from the same country? So now we know that the 20 authors come from six different countries. But we might want to also know how many authors come from the same country. To display the result set listing the country and number of authors that come from that country, we add the "group by" clause to the select statement. The "group by" clause groups a result into subsets that has matching values for one or more columns. In this example, countries are grouped and then counted using the count function. Notice the column heading for the second column and the result set. The numeric value "2" displays as a column name because the column name is not directly available in the table. The second column in the result set was calculated by the count function. Instead of using the column named "2," we can assign a column name to the result set. We do this using the "as" keyword. In this example, we change the derived column name "2" to column name "Count" using the "as count" keyword. This helps clarify the meaning of the result set. Now that we have the count of authors from different countries, we can further restrict the number of rows by passing some conditions. For example, we can check if there are more than four authors from the same country. To set a condition to a "group by" clause, we use the keyword "having". The "having" clause is used in combination with the "group by" clause. It is very important to note that the "where" clause is for the entire result set, but the "having" clause works only with the "group by" clause. To check if there are more than four authors from the same country, we add the following to the select statement, having count country greater than four. Only countries that have five or more authors from that country are listed in the result set. In this example, those countries are China with six authors and India, also with six authors. Now you can explain how to eliminate duplicates from a result set and describe how to further restrict a result set.

**Built-in Database Functions**

While it is very much possible to first fetch data from a database and then perform operations on it from your applications and notebooks, most databases come with Built-in Functions. These functions can be included in SQL statements, allowing you to perform operations on data right within the database itself. Using database functions can significantly reduce the amount of data that needs to be retrieved from the database. That is, reduces network traffic and use of bandwidth. When working with large data sets, it may be faster to use built in functions, rather than first retrieving the data into your application and then executing functions on the retrieved data. Note that it is also possible to create your own functions, that is User-Defined Functions in the database; but that is a more advanced topic. For the examples in this lesson, let's consider this PETRESCUE table in a database for a pet rescue organization. It records rescue transaction details and includes the columns: ID, animal, quantity, cost, and rescue date. For the purposes of this lesson, we have populated it with several rows of data, as shown here. What are aggregate or column functions? An aggregate function takes a collection of like values, such as all of the values in a column, as input, and returns a single value or null. Examples of aggregate functions include: sum, minimum, maximum, average, etc. Let's look at some examples based on the PETRESCUE table. The SUM function is used to add up all the values in a column. To use the function, you write the column name within parenthesis, after the function name. For example, to add up all the values in the COST column, select SUM (COST) from PETRESCUE. When you use an aggregate function, the column in the result set by default is given a number. It is possible to explicitly name the resulting column. For example, let's say we want to call the output column in the previous query, as SUM\_OF\_COST. select SUM(COST) as SUM\_OF\_COST from PETRESCUE. Note the use of 'as' in this example. MINimum, as the name implies, is used to get the lowest value. Similarly, MAXimum is used to get the highest value. For example, to get the maximum quantity of any animal rescue in a single transaction, select MAX(QUANTITY) from PETRESCUE. Aggregate functions can also be applied on a subset of data instead of an entire column. For example, to get the minimum quantity of the ID column for dogs. select MIN(ID) from PETRESCUE where animal equals dog. The Average (AVG) function is used to return the average or the mean value. For example, to specify the average value of cost, as: select AVG(COST) from PETRESCUE. Note that we can perform mathematical operations between columns, and then apply aggregate functions on them. For example, to calculate the average cost per dog: select AVG(COST divided by QUANTITY) from PETRESCUE where ANIMAL equals Dog. In this case, the cost is for multiple units; so we first divide the cost by the quantity of the rescue. Now let's look at the Scalar and String functions. Scalar functions perform operations on individual values. For example, to round up or down every value in the cost column to the nearest integer, select ROUND (COST) from PETRESCUE. There is a class of scalar functions called string functions, that can be used for operations on strings. That is char and varchar values. For example, to retrieve the length of each value in animal column, select LENGTH (ANIMAL) from PETRESCUE. Uppercase and lowercase functions can be used to return uppercase or lowercase values of strings. For example, to retrieve animal values in uppercase: select UPPERCASE (ANIMAL) from PETRESCUE. Scalar functions can be used in the where clause. For example, to get lowercase values of the animal column for cat, select star from PETRESCUE where LOWERCASE(ANIMAL) equals cat. This type of statement is useful for matching values in the where clause, if you're not sure whether the values are stored in upper, lower or mixed case in the table. You can also have one function operate on the output of another function. For example, to get unique cases for the animal column in uppercase: select DISTINCT (UPPERCASE(ANIMAL)) from PETRESCUE. In this video, we looked at some built-in SQL aggregate functions, such as sum, minimum, maximum, and average We also looked at scalar and string functions, such as round, lowercase, and uppercase.

**Date and Time Built-in Functions**

In this video, we'll go over date and time SQL functions built into the database. Most databases contain special data types for dates and times. Db2 contains date, time, and timestamp types. In Db2, date has eight digits: for year, month, and day. Time has six digits: hours, minutes, and seconds. Timestamp has 20 digits: year, month, day, hour, minute, seconds, and microseconds where double X represents month and six Zs or Zs represents microseconds. Functions exist to extract the day, month, day of month, day of week, day of year, week, hour, minute, and second. Let us look at some examples of queries for date and time functions. The day function can be used to extract the day portion from a date. For example, to get the day portion for each rescue date involving cat, select DAY (RESCUEDATE) from PETRESCUE where ANIMAL equals cat. Date and time functions can be used in the where clause. For example, to get the number of rescues during the month of May, that is, from Month 5, select COUNT star from PETRESCUE where MONTH (RESCUEDATE) equals 05. You can also perform date or time arithmetic. For example, to find out what date it is three days after each rescue date, maybe you want to know this because the rescue needs to be processed within three days. select (RESCUEDATE plus three DAYS) from PETRESCUE. Special registers current time and current date are also available. For example, to find how many days have passed since each rescue date till now- (CURRENT\_DATE minus RESCUEDATE) from PETRESCUE. The result will be in years, months, days. In this video, we looked at different types of built in SQL functions for working with dates and times.

**Sub-Queries and Nested Selects**

In this video you'll learn how to write sub-queries or nested selects statements. Sub-queries or sub selects are like regular queries but placed within parentheses and nested inside another query. This allows you to form more powerful queries than would have been otherwise possible. An example of a nested query is shown. In this example, the sub-query is inside the where clause of another query. Consider the employees table from the previous video. The first few rows of data are shown here. The table contains several columns, including an employee ID, first name, last name, salary, etc. We will now go over some examples involving this table. Let's consider a scenario which may necessitate the use of sub-queries. Let's say, we want to retrieve the list of employees who earn more than the average salary. To do so, you could try this code. Select \* from employees, where salary > AVG(salary). However, running this query will result in an error like the one shown. Indicating an invalid use of the aggregate function. One of the limitations of built in aggregate functions, like the average function, is that they cannot always be evaluated in the WHERE clause. So to evaluate a function like average in the WHERE clause, we can make use of a sub-select expression like the one shown here. Select EMP\_ID, F\_NAME, L\_NAME, SALARY from employees where SALARY < ( select AVG (SALARY) from employees). Notice that the average function is evaluated in the first part of the sub-query. Allowing us to circumvent the limitation of evaluating it directly in the WHERE clause. The sub-select doesn't just have to go in the WHERE clause. It can also go in other parts of the query, such as in the list of columns to be selected. Such sub-queries are called column expressions. Now, let's look at a scenario where we might want to use a column expression. Say we wanted to compare the salary of each employee with the average salary. We could try a query like select EMP\_ID, SALARY, AVG(SALARY) AS AVG\_SALARY from employees. Running this query will result in an error indicating that no group by clause is specified. We can circumvent this error by using the average function in a sub-query placed in the list of the columns. For example, select EMP\_ID, SALARY, (select AVG(SALARY) from employees ) AS AVG\_SALARY from employees. Another option is to make the sub-query be part of the FROM clause. Sub-queries like these are sometimes called derived tables or table expressions. Because the outer query uses the results of the sub-query as a data source. Let's look at an example to create a table expression that contains nonsensitive employee information. Select \* from (select EMP\_ID, F\_NAME, L\_NAME, DEP\_ID from employees) AS EMP4ALL. The derived table in a sub-query does not include sensitive fields like date of birth or salary. This example is a trivial one, and we could just as easily have included the columns in the outer query. However, such derived tables can prove to be powerful in more complex situations such as when working with multiple tables and doing joins. In this video, you have seen how sub-queries and nested queries can be used to form richer queries and how they can overcome some of the limitations of aggregate functions. You also learned to use sub-queries in the WHERE clause, in the list of columns and in the FROM clause.

**Working with Multiple Tables**

In this video, you will learn how to write queries that access more than one table. There are several ways to access multiple tables in the same query. Namely, using Sub-queries, Implicit JOIN, and JOIN operators, such as INNER JOIN and OUTER JOIN. In this video, we'll examine the first two options. The third option is covered in more detail in other videos. Let's consider the employees and departments tables from a previous video. The employees table contains several columns for categories, such as employee ID, first name, last name, and salary to name a few. The Departments table contains a department ID, department name, Manager ID, and location ID. Some sample data from these tables is shown here. We will utilize these tables for the examples in this video. In a previous video, we learned how to use sub-queries. Now, let's use sub-queries to work with multiple tables. If we want to retrieve only the employee records from the employees table for which a department ID exists in the departments table, we can use a sub-query as follows. Select star from employees, where department\_ID IN, select department\_ID\_department from departments. Here the outer query accesses the employees table and the sub-query on the departments table is used for filtering the result set of the outer query. Let's say we want to retrieve only the list of employees from a specific location. We do not have any location information in the employees table, but the departments table has a column called location ID. Therefore, we can use a sub-query from the Departments table as input to the employee table query as follows. Select star from employees, where department\_ID IN, select department\_ID\_department from departments, where location ID equals L0002. Now, let's retrieve the department ID and department name for employees who earn more than $70,000. To do so, we will need a sub-query on the employees table to satisfy the salary criteria, and then feed it as input to an outer query on the departments table in order to get the matching department info. Select department\_ID\_department department\_name from departments, where department\_ID\_department IN, select department\_ID from employees where salary is greater than 70,000. We can also access multiple tables by specifying them in the FROM clause of the query. Consider the example, select star from employees, departments. Here we specify two tables in the FROM clause. This results in a table join, but note we are not explicitly using the join operator. The resulting join in this example is called a full join or Cartesian join, because every row in the first table is joined with every row in the second table. If you examine the results set, you will see more rows than in both tables individually. We can use additional operands to limit the result set. Let's look at an example where we limit the result set to only rows with matching department IDs. Select star from employees, departments, where employees department\_ID equals departments, department\_ID\_department. Notice that in the WHERE clause, we prefix the name of the column with the name of the table. This is to fully qualify the column name, since it's possible that different tables could have some column names that are exactly the same. Since the table names can sometimes be long, we can use shorter aliases for table names as shown here. Select star from employees E, departments D, where E department\_ID equals D department\_ID\_department. Here, we define the alias E for employees table and D for departments table, and then use these aliases in the WHERE clause. If we wanted to see the department name for each employee, we would enter the code as follows: Select Employee\_ID, Department\_Name from employees E, departments D, where E department\_ID equals D department\_ID\_department. Similar to before, the column names in the select clause can also be prefixed by aliases as shown in the query. Select E. Employee\_ID, D. Department\_ID\_department, from employees E. Departments D where E.Department\_ID equals D. Department\_ID\_department. In this lesson, we have shown you how to work with multiple tables using sub-queries and implicit joins.