CHAPTER 4

Queries: Selecting Data from a Table

About this chapter

The SELECT statement retrieves data from the database. You can use it to retrieve a subset of the rows in one or more tables and to retrieve a subset of the columns in one or more tables.

This chapter focuses on the basics of single-table SELECT statements. Advanced uses of SELECT are described later in this manual.

Contents

Query overview

A query requests data from the database and receives the results. This process is also known as data retrieval. All SQL queries are expressed using the SELECT statement.

☞ This chapter assumes a familiarity with very simple queries. For an introduction to queries, see "Selecting Data from Database Tables" on page 231 of the book *First Guide to SQL Anywhere Studio*.

Queries are made up of clauses

You construct SELECT statements from **clauses**. In the following SELECT syntax, each new line is a separate clause. Only the more common clauses are listed here.

```
SELECT select-list
    [ FROM table-expression ]
    [ ON search-condition ]
    [ WHERE search-condition ]
    [ GROUP BY column-name ]
    [ HAVING search-condition ]
    [ ORDER BY { expression | integer } ]
```

The clauses in the SELECT statement are as follows:

- The SELECT clause specifies the columns you want to retrieve. It is the only required clause in the SELECT statement.
- The FROM clause specifies the tables from which columns are pulled. It is required in all queries that retrieve data from tables. In the current chapter, the *table-expression* is a single table name. SELECT statements without FROM clauses have a different meaning, and we ignore them in this chapter.
- The ON clause specifies how tables in the FROM clause are to be joined. It is used only for multi-table queries and is not discussed in this chapter.
- The WHERE clause specifies the rows in the tables you want to see.
- The GROUP BY clause allows you to collect aggregate data.
- The HAVING clause specifies rows on which aggregate data is to be collected.
- By default, rows are returned from relational databases in an order that has no meaning. You can use the ORDER BY clause to sort the rows in the result set.

Most of the clauses are optional, but if they are included then they must appear in the correct order.

☞ For a complete listing of the syntax of the SELECT statement syntax, see "SELECT statement" on page 542 of the book *Adaptive Server Anywhere Reference Manual*.

This chapter discusses only the following set of queries:

- Queries with only a single table in the FROM clause. For information on multi-table queries, see "Joins: Retrieving Data from Several Tables" on page 129.
- Queries with no GROUP BY, HAVING, or ORDER BY clauses. For information on these, see "Summarizing, Grouping, and Sorting Query Results" on page 109.

Entering queries

In this manual, SELECT statements and other SQL statements are displayed with each clause on a separate row, and with the SQL keywords in upper case. This is not a requirement. You can enter SQL keywords in any case, and you break lines at any point.

Keywords and line breaks

For example, the following SELECT statement finds the first and last names of contacts living in California from the **Contact** table.

```
SELECT first_name, last_name
FROM Contact
WHERE state = 'CA'
```

It is equally valid, though not as readable, to enter this statement as follows:

```
SELECT first_name,
last_name from contact
wHere state
 = 'CA'
```

Case sensitivity of strings and identifiers

Identifiers (that is, table names, column names, and so on) are case insensitive in Adaptive Server Anywhere databases.

Strings are case sensitive by default, so that 'CA', 'ca', 'cA', and 'Ca' are equivalent, but if you create a database as case-sensitive then the case of strings is significant. The sample database is case insensitive.

Qualifying identifiers

You can **qualify** the names of database identifiers if there is ambiguity about which object is being referred to. For example, the sample database contains several tables with a column called **city**, so you may have to qualify references to **city** with the name of the table. In a larger database you may also have to use the name of the owner of the table to identify the table.

```
SELECT dba.contact.city
FROM contact
WHERE state = 'CA'
```

Since the examples in this chapter involve single-table queries, column names in syntax models and examples are usually not qualified with the names of the tables or owners to which they belong.

These elements are left out for readability; it is never wrong to include qualifiers.

The remaining sections in this chapter analyze the syntax of the SELECT statement in more detail.

The SELECT clause: specifying columns

The select list

The **select list** commonly consists of a series of column names separated by commas, or an asterisk as shorthand to represent all columns.

More generally, the select list can include one or more expressions, separated by commas. The general syntax for the select list looks like this:

SELECT *expression* [, *expression*]...

If any table or column name in the list does not conform to the rules for valid identifiers, you must enclose the identifier in double quotes.

The select list expressions can include * (all columns), a list of column names, character strings, column headings, and expressions including arithmetic operators. You can also include aggregate functions, which are discussed in "Summarizing, Grouping, and Sorting Query Results" on page 109.

For a complete listing of what expressions can consist of, see "Expressions" on page 183 of the book *Adaptive Server Anywhere Reference Manual*.

The following sections provide examples of the kinds of expressions you can use in a select list.

Selecting all columns from a table

The asterisk (*) has a special meaning in SELECT statements. It stands for all the column names in all the tables specified by the FROM clause. You can use it to save typing time and errors when you want to see all the columns in a table.

When you use SELECT *, the columns are returned in the order in which they were defined when the table was created

The syntax for selecting all the columns in a table is:

```
SELECT *
FROM table-expression
```

SELECT * finds all the columns currently in a table, so that changes in the structure of a table such as adding, removing, or renaming columns automatically modify the results of SELECT *. Listing the columns individually gives you more precise control over the results.

Example

The following statement retrieves all columns in the **department** table. No WHERE clause is included; and so this statement retrieves every row in the table:

```
SELECT *
FROM department
```

The results look like this:

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703

You get exactly the same results by listing all the column names in the table in order after the SELECT keyword:

```
SELECT dept_id, dept_name, dept_head_id
FROM department
```

Like a column name, "*" can be qualified with a table name, as in the following query:

```
SELECT department.*
FROM department
```

Selecting specific columns from a table

To SELECT only specific columns in a table, use this syntax:

SELECT *column_name* [, *column_name*]...
FROM *table-name*

You must separate each column name from the column name that follows it with a comma, for example:

```
SELECT emp_lname, emp_fname
FROM employee
```

Rearranging the order of columns

The order in which you list the column names determines the order in which the columns are displayed. The two following examples show how to specify column order in a display. Both of them find and display the department names and identification numbers from all five of the rows in the department table, but in a different order.

```
SELECT dept_id, dept_name
FROM department
```

dept_id	dept_name
100	R & D
200	Sales
300	Finance
400	Marketing
500	Shipping

```
SELECT dept_name, dept_id
FROM department
```

dept_name	dept_id
R & D	100
Sales	200
Finance	300
Marketing	400
Shipping	500

Renaming columns using aliases in query results

Query results consist of a set of columns. By default, the heading for each column is the expression supplied in the select list.

When query results are displayed, each column's default heading is the name given to it when it was created. You can specify a different column heading, or **alias**, in one of the following ways:

SELECT *column-name* **AS** *alias*

SELECT *column-name alias*

SELECT *alias* = *column-name*

Providing an alias can produce more readable results. For example, you can change **dept_name** to **Department** in a listing of departments as follows:

```
SELECT dept_name AS Department,
   dept_id AS "Identifying Number"
FROM department
```

Department	Identifying Number
R & D	100
Sales	200
Finance	300
Marketing	400
Shipping	500

Using spaces and keywords in alias

The **Identifying Number** alias for **dept_id** is enclosed in double quotes because it is an identifier. You also use double quotes if you wish to use keywords in aliases. For example, the following query is invalid without the quotation marks:

```
SELECT dept_name AS Department,
   dept_id AS "integer"
FROM department
```

If you wish to ensure compatibility with Adaptive Server Enterprise, you should use quoted aliases of 30 bytes or less.

Character strings in query results

The SELECT statements you have seen so far produce results that consist solely of data from the tables in the FROM clause. Strings of characters can also be displayed in query results by enclosing them in single quotation marks and separate them from other elements in the select list with commas.

To enclose a quotation mark in a string, you precede it with another quotation mark.

For example:

```
SELECT 'The department''s name is' AS " ",
   Department = dept_name
FROM department
```

	Department
The department's name is	R & D
The department's name is	Sales
The department's name is	Finance
The department's name is	Marketing
The department's name is	Shipping

Computing values in the select list

The expressions in the select list can be more complicated than just column names or strings. For example, you can perform computations with data from numeric columns in a select list.

Arithmetic operations

To illustrate the numeric operations you can carry out in the select list, we start with a listing of the names, quantity in stock, and unit price of products in the sample database. The number of zeroes in the **unit_price** column is truncated for readability.

```
SELECT name, quantity, unit_price
FROM product
```

name	quantity	unit_price
Tee Shirt	28	9.00
Tee Shirt	54	14.00
Tee Shirt	75	14.00
Baseball Cap	112	9.00
Baseball Cap	12	10.00
Visor	36	7.00
Visor	28	7.00
Sweatshirt	39	24.00
Sweatshirt	32	24.00
Shorts	80	15.00

Suppose the practice is to replenish the stock of a product when there are ten items left in stock. The following query lists the number of each product that must be sold before re-ordering:

```
SELECT name, quantity - 10
   AS "Sell before reorder"
FROM product
```

name	Sell before reorder
Tee Shirt	18
Tee Shirt	44
Tee Shirt	65
Baseball Cap	102
Baseball Cap	2

name	Sell before reorder
Visor	26
Visor	18
Sweatshirt	29
Sweatshirt	22
Shorts	70

You can also combine the values in columns. The following query lists the total value of each product in stock:

```
SELECT name,
   quantity * unit_price AS "Inventory value"
FROM product
```

name	Inventory value
Tee Shirt	252.00
Tee Shirt	756.00
Tee Shirt	1050.00
Baseball Cap	1008.00
Baseball Cap	120.00
Visor	252.00
Visor	196.00
Sweatshirt	936.00
Sweatshirt	768.00
Shorts	1200.00

Arithmetic operator precedence

When there is more than one arithmetic operator in an expression, multiplication, division, and modulo are calculated first, followed by subtraction and addition. When all arithmetic operators in an expression have the same level of precedence, the order of execution is left to right. Expressions within parentheses take precedence over all other operations.

For example, the following SELECT statement calculates the total value of each product in inventory, and then subtracts five dollars from that value.

```
SELECT name, quantity * unit_price - 5
FROM product
```

To avoid misunderstandings, it is recommended that you use parentheses. The following query has the same meaning and gives the same results as the previous one, but some may find it easier to understand:

```
SELECT name, ( quantity * unit_price ) - 5
FROM product
```

String operations

You can concatenate strings using a string concatenation operator. You can use either || (SQL/92 compliant) or + (supported by Adaptive Server Enterprise) as the concatenation operator.

The following example illustrates the use of the string concatenation operator in the select list:

```
SELECT emp_id, emp_fname || ' ' || emp_lname AS Name
FROM employee
```

emp_id	Name
102	Fran Whitney
105	Matthew Cobb
129	Philip Chin
148	Julie Jordan
...	...

Date and time operations

Although you can use operators on date and time columns, this typically involves the use of functions. For information on SQL functions, see "SQL Functions" on page 267 of the book *Adaptive Server Anywhere Reference Manual*.

Eliminating duplicate query results

The optional DISTINCT keyword eliminates duplicate rows from the results of a SELECT statement.

If you do not specify DISTINCT, you get all rows, including duplicates. Optionally, you can specify all before the select list to get all rows. For compatibility with other implementations of SQL, Adaptive Server syntax allows the use of ALL to explicitly ask for all rows. ALL is the default.

For example, if you search for all the cities in the **contact** table without DISTINCT, you get 60 rows:

```
SELECT city
FROM contact
```

You can eliminate the duplicate entries using DISTINCT. The following query returns only 16 rows.:

```
SELECT DISTINCT city
FROM contact
```

NULL values are not distinct

The DISTINCT keyword treats NULL values as duplicates of each other. In other words, when DISTINCT is included in a SELECT statement, only one NULL is returned in the results, no matter how many NULL values are encountered.

The FROM clause: specifying tables

The FROM clause is required in every SELECT statement involving data from tables or views.

☞ The FROM clause can include JOIN conditions linking two or more tables, and can include joins to other queries (derived tables). For information on these features, see "Joins: Retrieving Data from Several Tables" on page 129.

Qualifying table names

In the FROM clause, the full naming syntax for tables and views is always permitted, such as:

```
SELECT select-list
FROM owner.table_name
```

Qualifying table and view names is necessary only when there might be some confusion about the name.

Using correlation names

You can give table names **correlation names** to save typing. You assign the correlation name in the FROM clause by entering it after the table name, like this:

```
SELECT d.dept_id, d.dept_name
FROM Department d
```

All other references to the **Department** table, for example in a WHERE clause, *must* use the correlation name. Correlation names must conform to the rules for valid identifiers.

The WHERE clause: specifying rows

The WHERE clause in a SELECT statement specifies the search conditions for exactly which rows are retrieved. The general format is:

SELECT *select_list*
FROM *table_list*
WHERE *search-condition*

Search conditions, (also called **qualifications** or **predicates**), in the WHERE clause include the following:

- **Comparison operators** (=, <, >, and so on) For example, you can list all employees earning more than $50,000:

```
SELECT emp_lname
FROM employee
WHERE salary > 50000
```

- **Ranges** (BETWEEN and NOT BETWEEN) For example, you can list all employees earning between $40,000 and $60,000:

```
SELECT emp_lname
FROM employee
WHERE salary BETWEEN 40000 AND 60000
```

- **Lists** (IN, NOT IN) For example, you can list all customers in Ontario, Quebec, or Manitoba:

```
SELECT company_name , state
FROM customer
WHERE state IN( 'ON', 'PQ', 'MB')
```

- **Character matches** (LIKE and NOT LIKE) For example, you can list all customers whose phone numbers start with 415. (The phone number is stored as a string in the database):

```
SELECT company_name , phone
FROM customer
WHERE phone LIKE '415%'
```

- **Unknown values** (IS NULL and IS NOT NULL) For example, you can list all departments with managers:

```
SELECT dept_name
FROM Department
WHERE dept_head_id IS NOT NULL
```

- **Combinations** (AND, OR) For example, you can list all employees earning over $50,000 whose first name begins with the letter A.

```
SELECT emp_fname,  emp_lname
FROM employee
WHERE salary > 50000
```

```
AND emp_fname like 'A%
```

In addition, the WHERE keyword can introduce the following:

- **Transact-SQL join conditions** Joins are discussed in "Joins: Retrieving Data from Several Tables" on page 129.

The following sections describe how to use WHERE clauses. For a complete listing of search conditions, see "Search conditions" on page 194 of the book *Adaptive Server Anywhere Reference Manual*.

Using comparison operators in the WHERE clause

You can use comparison operators in the WHERE clause. The operators follow the syntax:

WHERE *expression comparison-operator expression*

For a listing of comparison operators, see "Comparison conditions" on page 195 of the book *Adaptive Server Anywhere Reference Manual*. For a description of what an expression can be, see "Expressions" on page 183 of the book *Adaptive Server Anywhere Reference Manual*.

Notes on comparisons

- **Sort orders** In comparing character data, < means earlier in the sort order and > means later in the sort order. The sort order is determined by the **collation** chosen when the database is created. You can find out the collation by running the *dbinfo* command-line utility against the database:

  ```
  dbinfo -c "uid=dba;pwd=sql"
  ```

  You can also find the collation from Sybase Central. It is on the Extended Information tab of the database property sheet.

- **Trailing blanks** When you create a database, you indicate whether trailing blanks are to be ignored or not for the purposes of comparison.

  By default, databases are created with trailing blanks not ignored. For example, 'Dirk' is not the same as 'Dirk '. You can create databases with blank padding, so that trailing blanks are ignored. Trailing blanks are ignored by default in Adaptive Server Enterprise databases.

- **Comparing dates** In comparing dates, < means earlier and > means later.

- **Case sensivitivity** When you create a database, you indicate whether string comparisons are case sensitive or not.

  By default, databases are created case insensitive. For example, 'Dirk' is the same as 'DIRK'. You can create databases to be case sensitive, which is the default behavior for Adaptive Server Enterprise databases.

Here are some SELECT statements using comparison operators:

```
SELECT *
FROM product
WHERE quantity < 20
```

```
SELECT E.emp_lname, E.emp_fname
FROM employee E
WHERE emp_lname > 'McBadden'
```

```
SELECT id, phone
FROM contact
WHERE state  != 'CA'
```

The NOT operator

The NOT operator negates an expression. Either of the following two queries will find all Tee shirts and baseball caps that cost $10 or less. However, note the difference in position between the negative logical operator (NOT) and the negative comparison operator (!>).

```
SELECT id, name, quantity
FROM product
WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
AND NOT unit_price > 10
```

```
SELECT id, name, quantity
FROM product
WHERE (name = 'Tee Shirt' OR name = 'BaseBall Cap')
AND unit_price !> 10
```

Using ranges (between and not between) in the WHERE clause

The BETWEEN keyword specifies an inclusive range, in which the lower value and the upper value are searched for as well as the values they bracket.

❖ **To list all the products with prices between $10 and $15, inclusive:**

♦ Enter the following query:

```
SELECT name, unit_price
FROM product
WHERE unit_price BETWEEN 10 AND 15
```

name	unit_price
Tee Shirt	14.00
Tee Shirt	14.00
Baseball Cap	10.00
Shorts	15.00

You can use NOT BETWEEN to find all the rows that are not inside the range.

❖ **To list all the products cheaper than $10 or more expensive than $15:**

♦ Enter the following query:

```
SELECT name, unit_price
FROM product
WHERE unit_price NOT BETWEEN 10 AND 15
```

name	unit_price
Tee Shirt	9.00
Tee Shirt	9.00
Visor	7.00
Visor	7.00
Sweatshirt	24.00
Sweatshirt	24.00

Using lists in the WHERE clause

The IN keyword allows you to select values that match any one of a list of values. The expression can be a constant or a column name, and the list can be a set of constants or, more commonly, a subquery.

For example, without in, if you want a list of the names and states of all the contacts who live in Ontario, Manitoba, or Quebec, you can type this query:

```
SELECT company_name , state
FROM customer
WHERE state = 'ON' OR state = 'MB' OR state = 'PQ'
```

However, you get the same results if you use IN. The items following the IN keyword must be separated by commas and enclosed in parentheses. Put single quotes around character, date, or time values. For example:

```
SELECT company_name , state
FROM customer
WHERE state IN( 'ON', 'MB', 'PQ')
```

Perhaps the most important use for the IN keyword is in nested queries, also called **subqueries**.

Matching character strings in the WHERE clause

The LIKE keyword indicates that the following character string is a matching pattern. LIKE is used with character, binary, or date and time data.

The syntax for like is:

{ WHERE | HAVING } *expression* **[NOT] LIKE** *match-expression*

The expression to be matched is compared to a match-expression that can include these special symbols:

Symbols	Meaning
%	Matches any string of 0 or more characters
_	Matches any one character
[specifier]	The specifier in the brackets may take the following forms: ♦ **Range** A range is of the form *rangespec1-rangespec2*, where *rangespec1* indicates the start of a range of characters, the hyphen indicates a range, and *rangespec2* indicates the end of a range of characters ♦ **Set** A set can be comprised of any discrete set of values, in any order. For example, [a2bR]. Note that the range [a-f], and the sets [abcdef] and [fcbdae] return the same set of values.
[^specifier]	The caret symbol (^) preceding a specifier indicates non-inclusion. [^a-f] means not in the range a-f; [^a2bR] means not a, 2, b, or R.

You can match the column data to constants, variables, or other columns that contain the wildcard characters shown in the table. When using constants, you should enclose the match strings and character strings in single quotes.

Examples

All the following examples use LIKE with the **last_name** column in the **Contact** table. Queries are of the form:

```
SELECT last_name
FROM contact
WHERE last_name LIKE match-expression
```

The first example would be entered as

```
SELECT last_name
FROM contact
WHERE last_name LIKE 'Mc%'
```

Match expression	Description	Returns
'Mc%'	Search for every name that begins with the letters **Mc**	McEvoy
'%er'	Search for every name that ends with **er**	Brier, Miller, Weaver, Rayner
'%en%'	Search for every name containing the letters **en**.	Pettengill, Lencki, Cohen
'_ish'	Search for every four-letter name ending in **ish**.	Fish
'Br[iy][ae]r'	Search for Brier, Bryer, Briar, or Bryar.	Brier
'[M-Z]owell'	Search for all names ending with **owell** that begin with a single letter in the range M to Z.	Powell
'M[^c]%'	Search for all names beginning with M' that do not have c as the second letter	Moore, Mulley, Miller, Masalsky

Wildcards require LIKE

Wildcard characters used without LIKE are interpreted as **literals** rather than as a pattern: they represent exactly their own values. The following query attempts to find any phone numbers that consist of the four characters 415% only. It does not find phone numbers that start with 415.

```
SELECT phone
FROM Contact
WHERE phone = '415%'
```

Using LIKE with date and time values

You can use LIKE on date and time fields as well as on character data. When you use LIKE with date and time values, the dates are converted to the standard DATETIME format, and then to VARCHAR.

One feature of using LIKE when searching for DATETIME values is that, since date and time entries may contain a variety of date parts, an equality test has to be written carefully in order to succeed.

For example, if you insert the value 9:20 and the current date into a column named **arrival_time**, the clause:

```
WHERE arrival_time = '9:20'
```

fails to find the value, because the entry holds the date as well as the time. However, the clause below would find the 9:20 value:

```
WHERE arrival_time LIKE '%9:20%'
```

Using NOT LIKE

With NOT LIKE, you can use the same wildcard characters that you can use with LIKE. To find all the phone numbers in the **Contact** table that do not have 415 as the area code, you can use either of these queries:

```
SELECT phone
FROM Contact
WHERE phone NOT LIKE '415%'
```

```
SELECT phone
FROM Contact
WHERE NOT phone LIKE '415%'
```

Character strings and quotation marks

When you enter or search for character and date data, you must enclose it in single quotation marks, as in the following example.

```
SELECT first_name, last_name
FROM contact
WHERE first_name = 'John'
```

If the **quoted_identifier** database option is set to OFF (it is ON by default), you can also use double quotes around character or date data.

❖ **To set the quoted_identifier option off for the current user ID:**

♦ Type the following command:

```
SET OPTION quoted_identifier = 'OFF'
```

The **quoted_identifier** option is provided for compatibility with Adaptive Server Enterprise. By default, the Adaptive Server Enterprise option is **quoted_identifier** OFF and the Adaptive Server Anywhere option is **quoted_identifier** ON.

Quotation marks in strings

There are two ways to specify literal quotations within a character entry. The first method is to use two consecutive quotation marks. For example, if you have begun a character entry with a single quotation mark and want to include a single quotation mark as part of the entry, use two single quotation marks:

```
'I don''t understand.'
```

With double quotation marks (quoted_identifier OFF):

```
"He said, ""It is not really confusing."""
```

The second method, applicable only with **quoted_identifier** OFF, is to enclose a quotation in the other kind of quotation mark. In other words, surround an entry containing double quotation marks with single quotation marks, or vice versa. Here are some examples:

```
'George said, "There must be a better way."'
```

```
"Isn't there a better way?"
```

```
'George asked, "Isn''t there a better way?"'
```

Unknown Values: NULL

A NULL in a column means that the user or application has made no entry in that column. A data value for the column is unknown or not available

NULL does not mean the same as zero (numerical values) or blank (character values). Rather, NULL values allow you to distinguish between a deliberate entry of zero for numeric columns or blank for character columns and a non-entry, which is NULL for both numeric and character columns.

Entering NULL

NULL can be entered in a column where NULL values are permitted, as specified in the create table statement, in two ways:

- **Default** If no data is entered, and the column has no other default setting, NULL is entered.
- **Explicit entry** You can explicitly enter the value NULL by typing the word NULL (without quotation marks).

  If the word NULL is typed in a character column with quotation marks, it is treated as data, not as a null value.

For example, the **dept_head_id** column of the department table allows nulls. You can enter two rows for departments with no manager as follows:

```
INSERT INTO department (dept_id, dept_name)
VALUES (201, 'Eastern Sales')

INSERT INTO department
VALUES (202, 'Western Sales', null)
```

When NULLs are retrieved

When NULLS are retrieved, displays of query results in Interactive SQL show (NULL) in the appropriate position:

```
SELECT *
FROM department
```

dept_id	dept_name	dept_head_id
100	R & D	501
200	Sales	904
300	Finance	1293
400	Marketing	1576
500	Shipping	703
201	Eastern Sales	(NULL)
202	Western Sales	(NULL)

Testing a column for NULL

You can use IS NULL in search conditions to compare column values to NULL and to select them or perform a particular action based on the results of the comparison. Only columns that return a value of TRUE are selected or result in the specified action; those that return FALSE or UNKNOWN do not.

The following example selects only rows for which **unit_price** is less than $15 or is NULL:

```
SELECT quantity , unit_price
FROM product
WHERE unit_price < 15
OR unit_price IS NULL
```

The result of comparing NULL is UNKNOWN, since it is not possible to determine whether NULL is equal (or not equal) to a given value or to another NULL. The following cases return TRUE when expression is any column, variable or literal, or combination of these, which evaluates as NULL:

- *expression* IS NULL
- *expression* = NULL
- *expression* = *x* where *x* is a variable or parameter containing NULL. This exception facilitates writing stored procedures with null default parameters.
- *expression* != *n* where n is a literal not containing NULL and expression evaluates to NULL.

The negative versions of these expressions return TRUE when the expression does not evaluate to NULL:

- *expression* IS NOT NULL
- *expression* != NULL
- *expression* != *x*

Note that the far right side of these exceptions is a literal null, or a variable or parameter containing NULL. If the far right side of the comparison is an expression (such as @nullvar + 1), the entire expression evaluates to NULL.

There are some conditions that never return true, so that queries using these conditions do not return result sets. For example, the following comparison can never be determined to be true, since NULL means having an unknown value:

```
WHERE column1 > NULL
```

This logic also applies when you use two column names in a WHERE clause, that is, when you join two tables. A clause containing the condition

```
WHERE column1 = column2
```

does not return rows where the columns contain NULL.

You can also find NULL or non-NULL with this pattern:

```
WHERE column_name IS [NOT] NULL
```

For example:

```
WHERE advance < $5000 OR advance IS NULL
```

Properties of NULL

The following list expands on the properties of NULL.

- **The difference between FALSE and UNKNOWN** Although neither FALSE nor UNKNOWN returns values, there is an important logical difference between FALSE and UNKNOWN, because the opposite of false ("not false") is true. For example,

  ```
  1 = 2
  ```

  evaluates to false and its opposite,

  ```
  1 != 2
  ```

  evaluates to true. But "not unknown" is still unknown. If null values are included in a comparison, you cannot negate the expression to get the opposite set of rows or the opposite truth value.

- **Substituting a value for NULLs** Use the isnull built-in function to substitute a particular value for nulls. The substitution is made only for display purposes; actual column values are not affected. The syntax is:

  isnull (*expression*, *value*)

  For example, use the following statement to select all the rows from test, and display all the null values in column t1 with the value unknown.

  ```
  SELECT ISNULL(t1, 'unknown')
  FROM test
  ```

- **Expressions that evaluate to NULL** An expression with an arithmetic or bitwise operator evaluates to NULL if any of the operands are null. For example:

  ```
  1 + column1
  ```

  evaluates to NULL if column1 is NULL.

- **Concatenating strings and NULL** If you concatenate a string and NULL, the expression evaluates to the string. For example:

```
SELECT 'abc' || NULL || 'def'
```

returns the string **abcdef**.

Connecting conditions with logical operators

The logical operators AND, OR, and NOT are used to connect search conditions in WHERE clauses.

Using AND

The AND operator joins two or more conditions and returns results only when all of the conditions are true. For example, the following query finds only the rows in which the contact's last name is Purcell and the contact's first name is Beth. It does not find the row for Beth Glassmann.

```
SELECT *
FROM contact
WHERE first_name = 'Beth'
   AND last_name = 'Purcell'
```

Using OR

The OR operator also connects two or more conditions, but it returns results when *any* of the conditions is true. The following query searches for rows containing variants of Elizabeth in the **first_name** column.

```
SELECT *
FROM contact
WHERE first_name = 'Beth'
   OR first_name = 'Liz'
```

Using NOT

The NOT operator negates the expression that follows it. The following query lists all the contacts who do not live in California:

```
SELECT *
FROM contact
WHERE NOT state = 'CA'
```

When more than one logical operator is used in a statement, AND operators are normally evaluated before OR operators. You can change the order of execution with parentheses. For example:

```
SELECT *
FROM contact
WHERE ( city = 'Lexington'
        OR city = 'Burlington' )
   AND state = 'MA'
```