CHAPTER 7

Using Subqueries

About this chapter

When you create a query, you use WHERE and HAVING clauses to restrict the rows that the query will display.

Sometimes, the rows you select depend on information stored in more than one table. A subquery in the WHERE or HAVING clause allows you to select rows from one table according to specifications obtained from another table. Additional ways to do this can be found in "Joins: Retrieving Data from Several Tables" on page 129

Before your start

This chapter assumes some knowledge of queries and the syntax of the select statement. Information about queries is located in "Queries: Selecting Data from a Table" on page 85.

Contents

What is a subquery?

A relational database stores information about different types of objects in different tables. For example, you should store information particular to products in one table, and information that pertains to sales orders in another. The product table contains the information about the various products. The sales order items table contains information about one customers' orders.

In general, only the simplest questions can be answered using only one table. For example, if the company reorders products when there are fewer than 50 of them in stock, then it is possible to answer the question "Which products are nearly out of stock?" with this query:

```
SELECT id, name, description, quantity
FROM product
WHERE quantity < 50
```

However, if "nearly out of stock" depends on how many items of each type the typical customer orders, the number "50" will have to be replaced by a value obtained from the **sales_order_items** table.

Structure of the subquery

A subquery is structured like a regular query, and appears in the main query's WHERE or HAVING clause. In the above example, for instance, you can use a subquery to select the average number of items that a customer orders, and then use that figure in the main query to find products that are nearly out of stock. The following query finds the names and descriptions of the products which number less than double the average number of items of each type that a customer orders.

```
SELECT name, description
FROM product
WHERE quantity <  2 * (
   SELECT avg (quantity)
   FROM sales_order_items
   )
```

SQL subqueries always appear in the WHERE or HAVING clauses of the main query. In the WHERE clause, they help select the rows from the tables listed in the FROM clause that appear in the query results. In the HAVING clause, they help select the row groups, as specified by the main query's GROUP BY clause, that appear in the query results.

Using Subqueries in the WHERE clause

Subqueries in the WHERE clause work as part of the row selection process. You use a subquery in the where clause when the critieria that you use to select rows depend on the results of another table.

Example

Find the products whose in-stock quantities are less than double the average ordered quantity.

```
SELECT name, description
FROM product
WHERE quantity <  2 * (
   SELECT avg (quantity)
   FROM sales_order_items)
```

This query is executed in two steps: first, find the average number of items requested per order; and then find which products in stock number less than double that quantity.

The query in two steps

The number of items requested per item type, customer, and order is stored in the **quantity** column of the **sales_order_items** table. The subquery is

```
SELECT avg (quantity)
FROM sales_order_items
```

It returns the average quantity of items in the **sales_order_items** table, which is the number 25.851413.

The next query returns the ID numbers, names, and descriptions of the items whose in-stock quantities are less than twice the previously-extracted value

```
SELECT name, description
FROM product
WHERE quantity < 2*25.851413
```

Using a subquery combines the two steps into a single operation.

Purpose of a subquery in the WHERE clause

A subquery in the WHERE clause is part of a search condition. The chapter "Queries: Selecting Data from a Table" on page 85 describes simple search conditions that can be used in the WHERE clause.

Subqueries in the HAVING clause

Although subqueries are usually used as search conditions in the WHERE clause, they are occasionally found in the HAVING clause of a query. When a subquery appears in the HAVING clause, it, like any expression in the HAVING clause, is used as part of the row group selection.

Here is a request that lends itself naturally to a query with a subquery in the HAVING clause: "Which products' average in-stock quantity is less than double the average number of each item ordered per customer?"

Example

```
SELECT name, avg (quantity)
FROM product
GROUP BY name
HAVING avg (quantity) > 2* (
   SELECT avg (quantity)
   FROM sales_order_items
   )
```

name	avg(quantity)
Baseball Cap	62.000000
Shorts	80.000000
Tee Shirt	52.333333

The query is executed as follows:

- The subquery calculates the average quantity of items in the **sales_order_items** table.
- The main query then goes through the **product** table, calculating the average quantity product, grouping by product name.
- The HAVING clause then checks if that quantity is more than double the quantity found by the subquery. If so, the main query returns that row group; otherwise, it doesn't.
- The SELECT clause produces one summary row for each group, showing the name of each product and its in-stock quantity.

Outer references in the HAVING clause

You can also use outer references in a HAVING clause, as shown in this request, a slight variation on the one above:

Example

"Find the ID numbers and line ID numbers of those products whose average ordered quantities is less more than half the in-stock quantities of those products."

```
SELECT prod_id, line_id
FROM sales_order_items
GROUP BY prod_id, line_id
HAVING 2* avg (quantity) > (
   SELECT quantity
   FROM product
   WHERE product.id = sales_order_items.prod_id)
```

prod_id	line_id
300	1
401	2
500	1
501	2
600	1

In this example, the subquery must produce the average in-stock quantity of the product corresponding to the row group being tested by the HAVING clause. The subquery selects records for that particular product, using the outer reference **sales_order_items.prod_id**.

A subquery with a comparison returns a single value

This query uses the comparison ">", suggesting that the subquery must return exactly one value. In this case, it does, as the **id** field of the **product** table is a primary key, so there is only one record in the **product** table corresponding to any particular product id.

Subquery tests

The chapter "Queries: Selecting Data from a Table" on page 85 describes simple search conditions that can be used in the HAVING clause. As a subquery is just an expression that appears in the WHERE or HAVING clauses, the search conditions on subqueries may look familiar.

They are:

- **Subquery comparison test** For each record in the table(s) in the main query, compares the value of an expression to a single value produced by the subquery.
- **Quantified comparison test** Compares the value of an expression to each of the set of values produced by a subquery.
- **Subquery set membership test** Checks if the value of an expression matches one of the set of values produced by a subquery.
- **Existence test** Checks if the subquery produces any rows of query results.

Subquery comparison test

The subquery comparison test (=, <>, <. <=, >, >=) is a modified version of the simple comparison test; the only difference between the two is that in the former, the expression following the operator is a subquery. This test is used to compare a value from a row in the main query to a *single* value produced by the subquery.

Example

This query contains an example of a subquery comparison test

```
SELECT name, description, quantity
FROM product
WHERE quantity <  2 * (
   SELECT avg (quantity)
   FROM sales_order_items)
```

Name	description	quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
Visor	Plastic Visor	28
Sweatshirt	Hooded Sweatshirt	39
Sweatshirt	Zipped Sweatshirt	32

The following subquery retrieves a single value – the average quantity of items of each type per customer's order — from the **sales_order_items** table.

```
SELECT avg (quantity)
FROM sales_order_items
```

Then the main query compares the quantity of each in-stock item to that value.

A subquery in a comparison test returns one value

A subquery in a comparison test must return exactly one value. Consider this query, whose subquery extracts two columns from the **sales_order_items** table:

```
SELECT name, description, quantity
FROM product
WHERE quantity <  2 * (
   SELECT avg (quantity), max (quantity)
   FROM sales_order_items)
```

It returns the error

> subquery allowed only one select list item

Similarly, this query returns multiple values from the **quantity** column – one for each row in the **sales_order_items** table.

```
SELECT name, description, quantity
FROM product
WHERE quantity <  2 * (
   SELECT quantity
   FROM sales_order_items)
```

It returns the error

subquery cannot return more than one result

The subquery must appear to the right of a comparison operator

The subquery comparison test allows a subquery only on the right side of the comparison operator. Thus the comparison

main-query-expression < *subquery*

is allowed, but the comparison

subquery < *main-query-expression*

is not.

Quantified comparison tests with ANY and ALL

The quantified comparison test is broken into two categories, the ALL test and the ANY test:

The ANY test

The ANY test is used in conjunction with one of the SQL comparison operators (=, <>, <, <=, >, >=) to compare a single value to the column of data values produced by the subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column. If *any* of the comparisons yields a TRUE result, the ANY test returns TRUE.

A subquery used with ANY must return a single column.

Example

Find the customer and product ID's of those products that were ordered after the first order of product #2005 was shipped.

```
SELECT id, cust_id
FROM sales_order
WHERE order_date > ANY (
   SELECT ship_date
   FROM sales_order_items
   WHERE id=2005)
```

id	cust_id
2006	105
2007	106
2008	107
2009	108
…	…

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* order of product #2005. If an order date is greater than the shipping date for *one* shipment of product #2005, then the id and customer id from the **sales_order** table are displayed. The ANY test is thus analogous to the OR operator: the above query can be read, "Was this sales order placed after the first order of product #2005 was shipped, or after the second order of product #2005 was shipped, or…"

Understanding the ANY operator

The ANY operator can be a bit confusing. It is tempting to read the query as "Return those orders which were placed after any orders of product #2005 were shipped". But this means that the query will return the order ID's and customer ID's for the orders placed after *all* orders of product #2005 were shipped – which is not what the query does!

Instead, try reading the query like this: "Return the order ID's and customer ID's for those orders which were placed after *at least one* order of product #2005 were shipped." Using the keyword SOME may provide a more intuitive way to phrase the query; the following query is equivalent to the previous query.

```
SELECT id, cust_id
FROM sales_order
WHERE order_date > SOME (
   SELECT ship_date
   FROM sales_order_items
   WHERE id=2005)
```

The keyword SOME is equivalent to the keyword ANY.

Notes about the ANY operator

There are two additional important characteristics of the ANY test:

- **Empty subquery results** If the subquery produces an empty column of query results, the ANY test returns FALSE. This makes sense, as if there are no results, then it is not true that at least one result satisfies the comparison test.
- **NULL values in column** If the comparison test is not FALSE for any data value in the column, and is NULL for one or more values, the ANY search returns NULL. This is because in this situation, you cannot conclusively state whether there is a value for the subquery for which the comparison test holds; there may or may not be, depending on the "correct" values for the NULL data.

The ALL test

Like the ANY test, the ALL test is used in conjunction with one of the six SQL comparison operators (=, <>, <, <=, >, >=) to compare a single value to the column of data values produced by the subquery. To perform the test, SQL uses the specified comparison operator to compare the test value to each data value in the column. If *all* of the comparisons yield TRUE results, the ALL test returns TRUE.

Example

Here is a request that is naturally handled with the ALL test: "Find the customer and product ID's of those products that were ordered after all orders of product #2001 were shipped."

```
SELECT id, cust_id
FROM sales_order
WHERE order_date > ALL (
   SELECT ship_date
   FROM sales_order_items
   WHERE id=2001)
```

id	cust_id
2002	102
2003	103
2004	104
2005	101
…	…

In executing this query, the main query tests the order dates for each order against the shipping dates of *every* order of product #2001. If an order date is greater than the shipping date for *every* shipment of product #2001, then the id and customer id from the **sales_order** table are returned. The ALL test is thus analogous to the AND operator: the above query can be read, "Was this sales order placed before the first order of product #2001 was shipped, and before the second order of product #2001 was shipped, and…"

Notes about the ALL operator

There are three additional important characteristics of the ALL test:

- **Empty subquery results** If the subquery produces an empty column of query results, the ALL test returns TRUE. This makes sense, as if there are no results, then it is true that the comparison test holds for every value in the result set.
- **NULL values in column** If the comparison test is not FALSE for any data value in the column, and is NULL for one or more values, the ALL search returns NULL. This is because in this situation, you cannot conclusively state whether the comparison test holds for every value in the subquery result set; it may or may not, depending on the "correct" values for the NULL data.
- **Negating the ALL test** The following expressions are *not* equivalent.

  ```
  NOT a = ALL (subquery)
  ```

  ```
  a <> ALL (subquery)
  ```

  This is explained in detail in "Quantified comparison test" on page 181.

Testing set membership with IN conditions

You can use the subquery set membership test to compare a value from the main query to more than one value in the subquery.

The subquery set membership test compares a single data value for each row in the main query to the single column of data values produced by the subquery. If the data value from the main query matches *one* of the data values in the column, the subquery returns TRUE.

Example

Select the names of the employees who head the Shipping or Finance departments:

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
   SELECT dept_head_id
   FROM department
   WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

emp_fname	emp_lname
David	Scott
Jose	Martinez

The subquery in this example

```
SELECT dept_head_id
FROM department
WHERE (dept_name='Finance' OR dept_name = 'Shipping')
```

extracts from the **department** table the id numbers that correspond to the heads of the Shipping and Finance departments. The main query then returns the names of the employees whose id numbers match one of the two found by the subquery.

Negation of the set membership test

The subquery set membership test can also be used to extract those rows whose column values are not equal to any of those produced by a subquery. To negate a set membership test, insert the word NOT in front of the keyword IN.

Example

The subquery in this query returns the first and last names of the employees that are not heads of the Finance or Shipping departments.

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id NOT IN (
   SELECT dept_head_id
   FROM department
   WHERE (dept_name='Finance' OR dept_name =
'Shipping'))
```

Existence test

Subqueries used in the subquery comparison test and set membership test both return data values from the subquery table. Sometimes, however, you are not concerned with *which* results the subquery returns, but simply with whether the subquery returns *any* results. The existence test (EXISTS) checks whether a subquery produces any rows of query results. If the subquery produces one or more rows of results, the EXISTS test returns TRUE; otherwise, it returns FALSE.

Example

Here is an example of a request that can be expressed using a subquery: "Which customers placed orders after July 13, 1994?"

```
SELECT fname, lname
FROM customer
WHERE EXISTS (
   SELECT *
   FROM sales_order
   WHERE (order_date > '1994-07-13') AND (customer.id =
sales_order.cust_id))
```

fname	lname
Grover	Pendelton
Ling Ling	Andrews
Bubba	Murphy
Almen	de Joie

Explanation of the existence test

Here, for each row in the **customer** table, the subquery checks if that record's customer ID corresponds to one that has placed an order after July 13, 1994. If it does, the query extracts the first and last names of that customer from the main table.

The EXISTS test does not use the results of the subquery; it just checks if the subquery produces any rows. So the following two subqueries both produce the same results:

```
SELECT *
FROM sales_order
WHERE (order_date > '1994-07-13') AND (customer.id =
sales_order.cust_id)
```

```
SELECT ship_date
FROM sales_order
WHERE (order_date > '1994-07-13') AND (customer.id =
sales_order.cust_id)
```

It does not matter which columns from the **sales_order** table appear in the SELECT statement, though by convention, the "SELECT *" notation is used.

Negating the existence test

You can reverse the logic of the EXISTS test using the NOT EXISTS form. In this case, the test returns TRUE if the subquery produces no rows, and FALSE otherwise.

Correlated subqueries

You may have noticed that the subquery contains a reference to the **id** column from the **customer** table. References to columns or expressions in the main table(s) are called **outer references**. Conceptually, SQL processes the above query by going through the **customer** table, and performing the subquery for each customer. If the order date in the **sales_order** table is after July 13, 1994, and the customer ID in the **customer** and **sales_order** tables match, then the first and last names from the customer table are displayed. Since the subquery references the main query, the subquery in this section, unlike those from previous sections, will return an error if you attempt to run it by itself.

Outer references

Within the body of a subquery, it is often necessary to refer to the value of a column in the active row of the main query. Consider the following query:

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
   SELECT avg (quantity)
   FROM sales_order_items
   WHERE product.id = sales_order_items.prod_id)
```

This query extracts the names and descriptions of the products whose in-stock quantities are less than double the average ordered quantity of that product — specifically, the product being tested by the WHERE clause in the main query. The subquery does this by scanning the **sales_order_items** table. But the **product.id** column in the WHERE clause of the subquery refers to a column in the table named in the FROM clause of the *main* query — not the subquery. As SQL moves through each row of the **product** table, it uses the **id** value of the current row when evaluates the WHERE clause of the subquery.

Description of an outer reference

The **product.id** column in this subquery is an example of an **outer reference**. A subquery that uses an outer reference is called a **correlated subquery**. An outer reference is a column name that does not refer to any of the columns in any of the tables in the FROM clause of the subquery. Instead, the column name refers to a column of a table specified in the FROM clause of the main query. As the above example shows, the value of a column in an outer reference comes from the row currently being tested by the main query.

Subqueries and joins

Many queries that make use of subqueries are automatically rewritten by the subquery optimizer as joins.

Example

Consider the request, "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" It can be handled by this query:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id IN (
   SELECT id
   FROM customer
   WHERE lname = 'Clarke' OR fname = 'Suresh')
```

order_date	sales_rep
1994-01-05	1596
1993-01-27	667
1993-11-11	467
1994-02-04	195
1994-02-19	195
1994-04-02	299
1993-11-09	129
1994-01-29	690
1994-05-25	299

The subquery yields a list of customer ID's that correspond to the two customers whose names are listed in the WHERE clause, and the main query finds the order dates and sales representatives corresponding to those two people's orders.

Replacing a subquery with a join

The same question can be answered using joins. Here is an alternative form of the query, using a two-table join:

```
SELECT order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

This form of the query joins the **sales_order** table to the **customer** table to find the orders for each customer, and then returns only those records for Suresh and Mrs. Clarke.

Some joins cannot be written as subqueries

Both of these queries will find the correct order dates, and sales representatives, and neither is more right than the other. Many people will find the subquery form more natural, because the request doesn't ask for any information about customer ID's, and because it might seem odd to join the **sales_order** and **customer** tables together to answer the question. If, however, the request is changed to include some information from the **customer** table, the subquery form will no longer work. For example, the request "When did Mrs. Clarke and Suresh place their orders, and by which representatives, and what are their full names?", it is necessary to include the **customer** table in the main WHERE clause:

```
SELECT fname, lname, order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

fname	lname	order_date	sales_rep
Belinda	Clarke	1994-01-05	1596
Belinda	Clarke	1993-01-27	667
Belinda	Clarke	1993-11-11	467
Belinda	Clarke	1994-02-04	195
Belinda	Clarke	1994-02-19	195
Suresh	Naidu	1994-04-02	299
Suresh	Naidu	1993-11-09	129
Suresh	Naidu	1994-01-29	690
Suresh	Naidu	1994-05-25	299

Some subqueries cannot be written as joins

Similarly, there are cases when a subquery will work, but a join will not. One example is the following query:

```
SELECT name, description, quantity
FROM product
WHERE quantity <  2 * (
   SELECT avg (quantity)
   FROM sales_order_items)
```

name	description	quantity
Tee Shirt	Tank Top	28
Baseball Cap	Wool cap	12
Visor	Cloth Visor	36
…	…	…

In this case, the inner query is a summary query and the outer query is not, so there is no way the two queries can be combined by a simple join.

For more on joins, see. "Queries: Selecting Data from a Table" on page 85

Nested subqueries

As we have seen, subqueries always appear in the HAVING clause or the WHERE clause of a query. A subquery may itself contain a WHERE clause and/or a HAVING clause, and, consequently, a subquery may appear in another subquery. Subqueries inside other subqueries are called **nested subqueries**.

Examples

List the product ID's and line ID's of those items that were shipped when any item in the fees department was ordered.

```
SELECT id, line_id
FROM sales_order_items
WHERE ship_date = ANY (
   SELECT order_date
   FROM sales_order
   WHERE fin_code_id IN (
      SELECT code
      FROM fin_code
      WHERE (description = 'Fees')))
```

id	line_id
2001	1
2001	2
2001	3
2002	1
2002	2
…	…

Explanation of the nested subqueries

- In this example, the innermost subquery produces a column of financial codes whose descriptions are "Fees":

  ```
  SELECT code
  FROM fin_code
  WHERE (description = 'Fees')
  ```

- The next subquery finds the order dates of the items whose codes match one of the codes selected in the innermost subquery:

  ```
  SELECT order_date
  FROM sales_order
  WHERE fin_code_id IN (subquery)
  ```

- Finally, the outermost query finds the product ID's and line ID's of the items that were shipped on one of the dates found in the subquery.

```
SELECT id, line_id
FROM sales_order_items
WHERE ship_date = ANY (subquery)
```

Nested subqueries can also have more than three levels. Though there is no maximum number of levels, queries with three or more levels take considerably longer to run than do smaller queries.

How subqueries work

Understanding which queries are valid and which ones aren't can be complicated when a query contains a subquery. Similarly, figuring out what a multi-level query does can also be very involved, and it helps to understand how SQL processes subqueries. For general information about how queries are processed, see "Summarizing, Grouping, and Sorting Query Results" on page 109

Correlated subqueries

In a simple query, SQL evaluates and processes the query's WHERE clause once for each row of the query. Sometimes, though, the subquery returns only one result, making it unnecessary for SQL to evaluate it more than once for the entire result set.

Uncorrelated subqueries

Consider this query:

```
SELECT name, description
FROM product
WHERE quantity <  2 * (
   SELECT avg (quantity)
   FROM sales_order_items)
```

In this example, the subquery calculates exactly one value: the average quantity from the **sales_order_items** table. In evaluating the query, SQL computes this value once, and compares each value in the **quantity** field of the **product** table to it to determine whether the corresponding row is selected.

Correlated subqueries

When a subquery contains an outer reference, this shortcut can no longer be used. For instance, the subquery in the query

```
SELECT name, description
FROM product
WHERE quantity < 2 * (
   SELECT avg (quantity)
   FROM sales_order_items
   WHERE product.id=sales_order_items.prod_id)
```

returns a value that is dependent upon the active row in the **product** table. Such subqueries are called **correlated subqueries**. In these cases, there the subquery might return a different value for each row of the outer query, making it necessary for SQL to perform more than one evaluation.

Converting subqueries in the WHERE clause to joins

In general, a query that uses joins is executed faster than is a multi-level query. For this reason, whenever possible, the Adaptive Server Anywhere query optimizer will convert a multi-level query to a query that uses joins. The conversion is carried out without any user action. This section describes which subqueries can be converted to joins so that you can understand the performace of queries in your database.

Example

The question "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" can be written as a two-level query:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id IN (
   SELECT id
   FROM customer
   WHERE lname = 'Clarke' OR fname = 'Suresh')
```

An alternate, and equally correct way to write the query uses joins:

```
SELECT fname, lname, order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

The criteria that must be satisfied in order for a multi-level query to be able to be rewritten with joins differ for the various types of operators. Recall that when a subquery appears in the query's WHERE clause, it is of the form

SELECT *select-list*
FROM *table*
WHERE [**NOT**] *expression comparison-operator* (*subquery*) |
[**NOT**] *expression comparison-operator* **ANY** / **SOME** (*subquery*) |
[**NOT**] *expression comparison-operator* **ALL** (*subquery*) |
[**NOT**] *expression* **IN** (*subquery*) |
[**NOT**] **EXISTS** (*subquery*)
GROUP BY *group-by-expression*
HAVING *search-condition*

Whether a subquery can be converted to a join depends on a number of factors, such as the type of operator and the structure of the query.

Comparison operators

A subquery that follows a comparison operator (=, <>, <, <=, >, >=) must satisfy certain conditions if it is to be converted into a join. Subqueries that follow comparison operators in general are valid only if they return exactly one value for each row of the main query. In addition to this criterion, a subquery is converted to a join only if

- It does not contain a GROUP BY clause
- It does not contain the keyword DISTINCT
- It is not a UNION query
- It is not an aggregate query

Example

Suppose the request "When were Suresh's products ordered, and by which sales representative?" were phrased as the subquery

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id = (
   SELECT id
   FROM customer
   WHERE fname = 'Suresh')
```

This query satisfies the criteria, and therefore, it would be converted to a query that uses a join:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

However, the request, "Find the products whose in-stock quantities are less than double the average ordered quantity" cannot be converted to a join, as the subquery contains the aggregate function AVG:

```
SELECT name, description
FROM product
WHERE quantity <  2 * (
   SELECT avg (quantity)
   FROM sales_order_items)
```

Quantified comparison test

A subquery that follows one of the keywords ALL, ANY and SOME is converted into a join only if it satisfies certain criteria.

- The subquery does not contain a GROUP BY clause
- The subquery does not contain the keyword DISTINCT

- The subquery is not a UNION query
- The subquery is not an aggregate query
- If the subquery follows the keywords ANY or SOME, it must not be negated; if it follows the keyword ALL, it must be negated.

The first four of these conditions are relatively straightforward.

Example

The request "When did Mrs. Clarke and Suresh place their orders, and by which sales representatives?" can be handled in subquery form:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id = ANY (
   SELECT id
   FROM customer
   WHERE lname = 'Clarke' OR fname = 'Suresh')
```

Alternately, it can be phrased in join form

```
SELECT fname, lname, order_date, sales_rep
FROM sales_order, customer
WHERE cust_id=customer.id AND (lname = 'Clarke' OR fname
= 'Suresh')
```

However, request, "When did Mrs. Clarke, Suresh, and any employee who is also a customer, place their orders?" would be phrased as a union query, and thus cannot be converted to a join:

```
SELECT order_date, sales_rep
FROM sales_order
WHERE cust_id = ANY (
   SELECT id
   FROM customer
   WHERE lname = 'Clarke' OR fname = 'Suresh'
   UNION
   SELECT id
   FROM employee)
```

Similarly, the request "Find the customer and product ID's of those products that were not ordered after all orders of product #2001 were shipped," is naturally expressed with a subquery

A subquery with the ALL operator that can be converted to a join

```
SELECT id, cust_id
FROM sales_order
WHERE NOT order_date > ALL (
   SELECT ship_date
   FROM sales_order_items
   WHERE id=2001)
```

It would be converted to the join:

```
SELECT sales_order.id, cust_id
FROM sales_order, sales_order_items
WHERE (sales_order_items.id=2001) and (order_date <=
ship_date)
```

However, the request "Find the customer and product ID's of those products that were not shipped after the first shipping dates of all the products" would be phrased as the aggregate query

```
SELECT id, cust_id
FROM sales_order
WHERE NOT order_date > ALL (
   SELECT first (ship_date)
   FROM sales_order_items )
```

Therefore, it would not be converted to a join.

Negating subqueries with the ANY and ALL operators

The fifth criterion is a little more puzzling: queries of the form

SELECT *select-list*
FROM *table*
WHERE [**NOT**] *expression comparison-operator* **ALL** (*subquery*)

are converted to joins, as are queries of the form

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ANY** (*subquery*)

but the queries

SELECT *select-list*
FROM *table*
WHERE *expression comparison-operator* **ALL** (*subquery*)

and

SELECT *select-list*
FROM *table*
WHERE NOT *expression comparison-operator* **ANY** (*subquery*)

are not.

Logical equivalence of ANY and ALL expressions

This is because the first two queries are in fact equivalent, as are the last two. Recall that the any operator analogous to the or operator, but with a variable number of arguments; and that the ALL operator is similarly analogous to the AND operator. Just as the expression

```
NOT ((> A) AND (> B))
```

is equivalent to the expression

```
(<= A) OR (<= B)
```

the expression

```
NOT order_date > ALL (
   SELECT first (ship_date)
   FROM sales_order_items )
```

is equivalent to the expression

```
order_date <= ANY (
   SELECT first (ship_date)
   FROM sales_order_items )
```

Negating the ANY and ALL expressions

In general, the expression

NOT *column-name operator* **ANY** (*subquery*)

is equivalent to the expression

column-name inverse-operator **ALL** (*subquery*)

and the expression

NOT *column-name operator* **ALL** (*subquery*)

is equivalent to the expression

column-name inverse-operator **ANY** (*subquery*)

where *inverse-operator* is obtained by negating *operator*, as shown in the table:

Table of operators and their inverses

The following table lists the inverse of each operator.

Operator	inverse-operator
=	<>
<	=>
>	=<
=<	>
=>	<
<>	=

Set membership test

A query containing a subquery that follows the keyword IN is converted into a join only if:

- The subquery does not contain a GROUP BY clause
- The subquery does not contain the keyword DISTINCT
- The subquery is not a UNION query

- The subquery is not an aggregate query
- The subquery must not be negated

Example

So, the request "Find the names of the employees who are also department heads", which is expressed by the query

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
   SELECT dept_head_id
   FROM department
   WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

would be converted to a joined query, as it satisfies the conditions; however, the request, "Find the names of the employees who are also either department heads or customers" would not be converted to a join if it were expressed by the UNION query

A UNION query following the IN operator can't be converted

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
   SELECT dept_head_id
   FROM department
   WHERE (dept_name='Finance' or dept_name = 'Shipping')
   UNION
   SELECT cust_id
   FROM sales_order)
```

Similarly, the request "Find the names of employees who are not department heads" is formualted as the negated subquery

```
SELECT emp_fname, emp_lname
FROM employee
WHERE NOT emp_id IN (
   SELECT dept_head_id
   FROM department
   WHERE (dept_name='Finance' OR dept_name =
'Shipping'))
```

and would not be converted.

The conditions that must be fulfilled in order for a subquery that follows the must be fulfilled in order for a subquery that follows the must be fulfilled in order for a subquery that follows the IN keyword to be converted to a join are identical to those that must be fulfilled in order for a subquery that follows the ANY keyword to be converted. This is not a coincidence, and the reason for this is that the expression

A query with an IN operator can be converted to one with an ANY operator

WHERE *column-name* **IN** (*subquery*)

is logically equivalent to the expression

WHERE *column-name* **= ANY** (*subquery*)

So the query

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id IN (
   SELECT dept_head_id
   FROM department
   WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

is equivalent to the query

```
SELECT emp_fname, emp_lname
FROM employee
WHERE emp_id = ANY (
   SELECT dept_head_id
   FROM department
   WHERE (dept_name='Finance' or dept_name =
'Shipping'))
```

Conceptually, SQL converts a query with the IN operator to one with an ANY operator, and decides accordingly whether to convert the subquery to a join.

Existence test

A subquery that follows the keyword EXISTS is converted to a join only if it satisfies the following two conditions:

- The subquery is not negated
- The subquery is correlated; that is, it contains an outer reference.

Example

Therefore, the request, "Which customers placed orders after July 13, 1994?", which can be formulated by this query whose non-negated subquery subquery contains the outer reference **customer.id** = **sales_order.cust_id**, could be converted to a join.

```
SELECT fname, lname
FROM customer
WHERE EXISTS (
   SELECT *
   FROM sales_order
   WHERE (order_date > '1994-07-13') AND (customer.id =
sales_order.cust_id))
```

The EXISTS keyword essentially tells SQL to check for NULLs. When inner joins are used, SQL automatically only displays the rows in which there is data from all of the tables in the FROM clause; that is, SQL only returns the rows that do not contain NULLs. So this query returns the same rows as does the one with the subquery:

```
SELECT fname, lname
FROM customer, sales_order
WHERE (sales_order.order_date > '1994-07-13') AND
(customer.id = sales_order.cust_id).
```