All the queries you've seen so far in this book have contained just one SELECT statement. In this chapter, you will

- Learn how to place a SELECT statement within an outer SELECT, UPDATE, or DELETE statement. The inner SELECT statement is known as a subquery.
- Learn about the different types of subqueries and see how to use them.
- See how subqueries allow you to build up very complex statements from simple components.

Types of Subqueries

There are two basic types of subqueries:

- **Single row subqueries**  Return zero or one row to the outer SQL statement.
- **Multiple row subqueries**  Return one or more rows to the outer SQL statement.

In addition, there are three subtypes of subqueries that may return single or multiple rows:

- **Multiple column subqueries**  Return more than one column to the outer SQL statement.
- **Correlated subqueries**  Reference one or more columns in the outer SQL statement. The subquery is known as a correlated subquery because the subquery is related to the outer SQL statement.
- **Nested subqueries**  Are placed within another subquery. You can nest subqueries to a depth of 255.

You'll learn about each of these types of subqueries in this chapter, and see how to add subqueries to SELECT, UPDATE, and DELETE statements. Let's plunge in and look at how to write single row subqueries.

Writing Single Row Subqueries

A single row subquery is one that returns zero or one row to the outer SQL statement. As you'll see in this section, you may place a subquery in a WHERE clause, a HAVING clause, or a FROM clause of a SELECT statement. You'll also see some errors you might encounter when issuing subqueries.

Subqueries in a WHERE Clause

You may place a subquery in the WHERE clause of another query. Let's take a look at a very simple example of a query that contains a subquery placed in its WHERE clause; notice the subquery is placed within parentheses (...):
tatement. In this

\textbf{SELECT} \textit{first_name, last_name}
\textbf{FROM} customers
\textbf{WHERE} customer_id =
  (\textbf{SELECT} customer_id
  \textbf{FROM} customers
  \textbf{WHERE} last_name = 'Brown');

\begin{tabular}{ll}
  \textbf{FIRST_NAME} & \textbf{LAST_NAME} \\
  John & Brown
\end{tabular}

This example retrieves the \textit{first_name} and \textit{last_name} of the row from the customers table whose \textit{last_name} is Brown. Let's break this query down and analyze what's going on. The subquery in the \textbf{WHERE} clause is as follows:

\textbf{SELECT} customer_id
\textbf{FROM} customers
\textbf{WHERE} last_name = 'Brown';

This subquery is executed first (and only once) and returns the customer_id for the row whose \textit{last_name} is Brown. The customer_id for this row is 1, which is passed to the \textbf{WHERE} clause of the outer query. Therefore, the outer query may be considered to be identical to the following query:

\textbf{SELECT} \textit{first_name, last_name}
\textbf{FROM} customers
\textbf{WHERE} customer_id = 1;

\textbf{Using Other Single Row Operators}

The previous example used the equality operator (=) in the \textbf{WHERE} clause. You may also use other comparison operators such as \(<\), \(<\), \(\leq\), and \(\geq\) with a single row subquery. The following example uses \(\geq\) in the outer query's \textbf{WHERE} clause; the subquery uses the \textbf{AVG()} function to get the average price of the products, which is passed to the \textbf{WHERE} clause of the outer query. The final result of the entire query is to get the product_id, name, and price of products whose price is greater than that average price.

\textbf{SELECT} \textit{product_id, name, price}
\textbf{FROM} products
\textbf{WHERE} price >
  (\textbf{SELECT} \textbf{AVG}(price)
  \textbf{FROM} products);

\begin{tabular}{llr}
  \textbf{PRODUCT_ID} & \textbf{NAME} & \textbf{PRICE} \\
  \hline
  1 & Modern Science & 19.95 \\
  2 & Chemistry & 30 \\
  3 & Supernova & 25.99 \\
  5 & Z Files & 49.99
\end{tabular}
Let's break the example down to understand how it works. The following shows the output from the subquery when it's run on its own:

```sql
SELECT AVG(price)
FROM products;
```

```
AVG(PRICE)
---------
19.7308333
```

The value of 19.7308333 is used in the `WHERE` clause of the outer query shown earlier to get the products whose price is greater than that average.

### Subqueries in a HAVING Clause

As you saw in Chapter 3, you use the `HAVING` clause to filter groups of rows. You may place a subquery in a `HAVING` clause of an outer query. This allows you to filter groups of rows based on the result returned by your subquery.

The following example uses a subquery in the `HAVING` clause of the outer query. The example retrieves the `product_type_id` and the average price for products whose average price is less than the maximum of the average for the groups of the same type product type:

```sql
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) <
    (SELECT MAX(AVG(price))
     FROM products
     GROUP BY product_type_id);
```

```
PRODUCT_TYPE_ID AVG(PRICE)
---------------------------
1             24.975
3             13.24
4             13.99
4             13.49
```

Notice the subquery uses `AVG()` to first compute the average price for each product type. The result returned by `AVG()` is then passed to `MAX()`, which returns the maximum of the averages.

Let's break the example down to understand how it works. The following shows the output from the subquery when it is run on its own:

```sql
SELECT MAX(AVG(price))
FROM products
GROUP BY product_type_id;
```

```
MAX(AVG(PRICE))
---------------
26.22
```
This value of 26.22 is used in the HAVING clause of the outer query shown earlier to filter the group's rows to those having an average price less than 26.22. The following query shows a version of the outer query that retrieves the product_type_id and average price of the products grouped by product_type_id:

```sql
SELECT product_type_id, AVG(price) FROM products GROUP BY product_type_id;
```

<table>
<thead>
<tr>
<th>PRODUCT_TYPE_ID</th>
<th>AVG(PRICE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.975</td>
</tr>
<tr>
<td>2</td>
<td>26.22</td>
</tr>
<tr>
<td>3</td>
<td>13.24</td>
</tr>
<tr>
<td>4</td>
<td>13.99</td>
</tr>
<tr>
<td></td>
<td>13.49</td>
</tr>
</tbody>
</table>

You can see that the groups with a product_type_id of 1, 3, 4, and null have an average price less than 26.22. As expected, these are the same groups returned by the query containing the subquery at the start of this section.

### Subqueries in a FROM Clause (Inline Views)

You may place a subquery in the FROM clause of an outer query. These types of subqueries are also known as inline views because the subquery provides data inline with the FROM clause. The following simple example retrieves the products whose product_id is less than 3:

```sql
SELECT product_id FROM
(SELECT product_id
 FROM products
 WHERE product_id < 3);
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Notice the subquery returns the rows from the products table whose product_id is less than 3 to the outer query, which then retrieves and displays those product_id values. As far as the FROM clause of the outer query is concerned, the output from the subquery is just another source of data.

The next example is more useful and retrieves the product_id and price from the products table in the outer query, and the subquery retrieves the number of times a product has been purchased:

```sql
SELECT prds.product_id, price, purchases_data.product_count
FROM products prds,
(SELECT product_id, COUNT(product_id) product_count
 FROM products
 WHERE product_id < 3)
```
FROM purchases
GROUP BY product_id) purchases_data
WHERE prds.product_id = purchases_data.product_id;

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRICE</th>
<th>PRODUCT_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.95</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>25.99</td>
<td>1</td>
</tr>
</tbody>
</table>

Notice the subquery retrieves the `product_id` and `COUNT(product_id)` from the `purchases` table and returns them to the outer query. As you can see, the output from subquery is just another source of data to the FROM clause of the outer query.

A Couple of Errors You Might Encounter
In this section, you’ll see some errors you might encounter. Specifically, you’ll see that a single row subquery may return a maximum of one row, and you’ll see a subquery may not contain an ORDER BY clause.

Single Row Subqueries May Return a Maximum of One Row
If your subquery returns more than one row, you’ll get the following error:

ORA-01427: single-row subquery returns more than one row.

For example, the subquery in the following statement attempts to pass multiple rows to the equality operator (=) in the outer query:

```sql
SQL> SELECT product_id, name
2  FROM products
3  WHERE product_id =
4   (SELECT product_id
5   FROM products
6   WHERE name LIKE '%e%');

ERROR at line 4:
ORA-01427: single-row subquery returns more than one row
```

There are nine rows in the `products` table whose name contains the letter e, and the subquery attempts to pass these rows to the equality operator in the outer query. Since the equality operator can only handle a single row, the query is invalid and an error is returned.

You’ll learn how to return multiple rows from a subquery later in the section “Writing Multiple Row Subqueries.”

Subqueries May Not Contain an ORDER BY Clause
A subquery may not contain an ORDER BY clause. Instead, you must do any ordering in your outer query. For example, the following outer query has an ORDER BY clause at the end that sorts on the `product_id` column:

```sql
SELECT product_id,
FROM product
WHERE prds.product_id = purchases_data.product_id;

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>1</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>s</td>
</tr>
</tbody>
</table>
```
SELECT product_id, name, price
FROM products
WHERE price >
(SELECT AVG(price)
 FROM products)
ORDER BY product_id DESC;

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>NAME</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Z Files</td>
<td>49.99</td>
</tr>
<tr>
<td>3</td>
<td>Supernova</td>
<td>25.99</td>
</tr>
<tr>
<td>2</td>
<td>Chemistry</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>Modern Science</td>
<td>19.95</td>
</tr>
</tbody>
</table>

Writing Multiple Row Subqueries

You use a multiple row subquery to return one or more rows to the outer SQL statement. To handle a subquery that returns multiple rows, your outer query may use the IN, ANY, or ALL operator. As you saw in Chapter 2, you can use these operators to check and compare values supplied in a list of literal values. As you’ll see in this section, you can also supply this list of values from a subquery.

NOTE
You can also use the EXISTS operator to check if a value is in a list returned by a correlated subquery, which you’ll learn about later in the section “Writing Correlated Subqueries.”

Using IN with a Multiple Row Subquery

As you saw in Chapter 2, you use IN to check if a value is in a specified list of values. The list of values may come from the results returned by a subquery. You can also use NOT IN to perform the logical opposite of IN: you use NOT IN to check if a value is not in a specified list of values.

The following simple example uses IN to check if a product_id is in the list of product_id values returned by the subquery; the subquery returns the product_id column values for the products whose name contains the letter e:

```sql
SELECT product_id, name
FROM products
WHERE product_id IN
(SELECT product_id
 FROM products
 WHERE name LIKE 'e%');
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Modern Science</td>
</tr>
<tr>
<td>2</td>
<td>Chemistry</td>
</tr>
<tr>
<td>3</td>
<td>Supernova</td>
</tr>
</tbody>
</table>
The next example uses `NOT IN` to check if a `product_id` is not in the list of `product_id` values in the `purchases` table:

```
SELECT product_id, name
FROM products
WHERE product_id NOT IN
(SELECT product_id
FROM purchases);
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Tank War</td>
</tr>
<tr>
<td>5</td>
<td>Z Files</td>
</tr>
<tr>
<td>6</td>
<td>2412: The Return</td>
</tr>
<tr>
<td>7</td>
<td>Space Force 9</td>
</tr>
<tr>
<td>8</td>
<td>From Another Planet</td>
</tr>
<tr>
<td>9</td>
<td>Classical Music</td>
</tr>
<tr>
<td>10</td>
<td>Pop 3</td>
</tr>
<tr>
<td>11</td>
<td>Creative Yell</td>
</tr>
<tr>
<td>12</td>
<td>My Front Line</td>
</tr>
</tbody>
</table>

Using ANY with a Multiple Row Subquery
You use the `ANY` operator to compare a value with any value in a list. You must place an `=`, `<`, `<=`, `<>`, `>`, `>=`, or `<>` operator before `ANY` in your query. The following example uses `ANY` to check if any of the employees has a salary less than any of the lowest salaries in the `salary_grades` table:

```
SELECT employee_id, last_name
FROM employees
WHERE salary < ANY
(SELECT low_salary
FROM salary_grades);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Johnson</td>
</tr>
<tr>
<td>3</td>
<td>Hobbs</td>
</tr>
<tr>
<td>4</td>
<td>Jones</td>
</tr>
</tbody>
</table>

Using ALL with a Multiple Row Subquery
You use the `ALL` operator to compare a value with any value in a list. You must place an `=`, `<`, `<=`, `<>`, `>`, `>=`, or `<>` operator before `ALL` in your query. The following example uses `ALL` to check if any of the employees has a salary greater than all of the highest salaries in the `salary_grades` table:

```
SELECT employee_id, last_name
FROM employees
WHERE salary > ALL
(SELECT high_salary
FROM salary_grades);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Writing
The subqueries to one column: retrieves the products' names

```
SELECT product_id, name
FROM products
WHERE product_id NOT IN
(SELECT product_id
FROM purchases);
```

Notice the column values.

Writing
A correlated subquery as a correlated subquery contained in the

A Correla
The following query for their product

```
SELECT product_id, name
FROM products
WHERE product_id NOT IN
(SELECT product_id
FROM purchases);
```

Notice the column values.
SELECT employee_id, last_name
FROM employees
WHERE salary > ALL
  (SELECT high_salary
   FROM salary_grades);

no rows selected

As you can see from this result, no employee has a salary greater than the highest salary. This is probably a good thing, since you don’t want an employee earning more than the highest salary!

Writing Multiple Column Subqueries

The subqueries you’ve seen so far have returned rows containing one column. You’re not limited to one column: you can write subqueries that return multiple columns. The following example retrieves the products with lowest price in each product type group:

```sql
SELECT product_id, product_type_id, name, price
FROM products
WHERE (product_type_id, price) IN
  (SELECT product_type_id, MIN(price)
   FROM products
   GROUP BY product_type_id);
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
<td>19.95</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Tank War</td>
<td>13.95</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>From Another Planet</td>
<td>12.99</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>Classical Music</td>
<td>10.99</td>
</tr>
</tbody>
</table>

Notice the subquery returns the `product_type_id` and the minimum value of the `price` column values. The outer query has a `WHERE` clause with the two columns `product_type_id` and `price` in parentheses.

Writing Correlated Subqueries

A correlated subquery references one or more columns in the outer query. The subquery is known as a correlated subquery because the subquery is related to the outer query. You typically use a correlated subquery when you need an answer to a question that depends on a value in each row contained in the outer query.

A Correlated Subquery Example

The following correlated subquery retrieves products that have a price greater than the average for their product type:

```sql
SELECT product_id, product_type_id, name, price
FROM products
WHERE product_type_id IN
  (SELECT product_type_id
   FROM products
   GROUP BY product_type_id
   HAVING AVG(price) < 15);
```
WHERE price >
(SELECT AVG(price)
FROM products inner
WHERE inner.product_type_id = outer.product_type_id);

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
<th>PRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>Chemistry</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Z Files</td>
<td>49.99</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Space Force 9</td>
<td>13.49</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>Pop 3</td>
<td>15.99</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>Creative Yell</td>
<td>14.99</td>
</tr>
</tbody>
</table>

Notice I’ve used the alias outer to label the outer query and the alias inner for the inner query. The inner and outer parts are correlated using the product_type_id column.

In a correlated subquery, each row in the outer query is passed one at a time to the subquery. The subquery reads each row in turn from the outer query and applies it to the subquery until all the rows from the outer query have been processed. The results from the entire query are then returned.

In the previous example, the outer query retrieves each row from the products table and passes each row to the inner query. Each row is read by the inner query, which calculates the average price for each product where the product_type_id in the inner query is equal to the product_type_id in the outer query.

**Using EXISTS and NOT EXISTS with a Correlated Subquery**

You use the EXISTS operator to check for the existence of rows returned by a subquery. Although you can use EXISTS with non-correlated subqueries, you’ll typically use it with correlated subqueries. NOT EXISTS does the logical opposite of EXISTS. You use NOT EXISTS when you need to check if rows do not exist in the results returned by a subquery.

**Using EXISTS with a Correlated Subquery**

The following example uses EXISTS to retrieve employees who manage other employees:

```
SELECT employee_id, last_name
FROM employees outer
WHERE EXISTS
(SELECT employee_id
FROM employees inner
WHERE inner.manager_id = outer.employee_id);
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>Johnson</td>
</tr>
</tbody>
</table>

Since EXISTS just checks for the existence of rows returned by the subquery, your subquery doesn’t have to return a column: you can just return a literal value. This can improve performance of your query returning the

```
SELECT emp1
FROM emp
WHERE EXIST
(SELECT 1
FROM emp
WHERE in
EMPLOYEE_ID
1
2
```

**Using NOT**

The following

```
SELECT prod1
FROM product
WHERE NOT EXIST
(SELECT 1
FROM purc
WHERE in
PRODUCT_ID
4
5
6
2
7
S
8
F
9
C
10
P
11
C
12
M
```

**EXISTS and NOT**

Earlier in the section, we used to check if the employee

<table>
<thead>
<tr>
<th>TIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXISTS type checks if a row exists. Therefore, you...</td>
</tr>
</tbody>
</table>
of your query. For example, the following query rewrites the previous example with the subquery returning the literal value 1:

```sql
SELECT employee_id, last_name
FROM employees outer
WHERE EXISTS
  (SELECT 1
   FROM employees inner
   WHERE inner.manager_id = outer.employee_id);
```

- EMPLOYEE_ID: 1
  - Smith
  - Johnson

Using NOT EXISTS with a Correlated Subquery

The following example uses NOT EXISTS to retrieve products that haven’t been purchased:

```sql
SELECT product_id, name
FROM products outer
WHERE NOT EXISTS
  (SELECT 1
   FROM purchases inner
   WHERE inner.product_id = outer.product_id);
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Tank War</td>
</tr>
<tr>
<td>5</td>
<td>Z Files</td>
</tr>
<tr>
<td>6</td>
<td>2412: The Return</td>
</tr>
<tr>
<td>7</td>
<td>Space Force 9</td>
</tr>
<tr>
<td>8</td>
<td>From Another Planet</td>
</tr>
<tr>
<td>9</td>
<td>Classical Music</td>
</tr>
<tr>
<td>10</td>
<td>Pop 3</td>
</tr>
<tr>
<td>11</td>
<td>Creative Yell</td>
</tr>
<tr>
<td>12</td>
<td>My Front Line</td>
</tr>
</tbody>
</table>

EXISTS and NOT EXISTS Versus IN and NOT IN

Earlier in the section “Using IN with a Multiple Row Subquery,” you saw how the IN operator is used to check if a value is contained in a list. EXISTS is different from IN: EXISTS just checks for the existence of rows, whereas IN checks actual values.

**TIP**

EXISTS typically offers better performance than IN with subqueries. Therefore you should use EXISTS rather than IN wherever possible.
You should be careful when writing queries that use NOT EXISTS or NOT IN. When a list of values contains a null, NOT EXISTS returns true, but NOT IN returns false. Consider the following example that uses NOT EXISTS and retrieves the product types that don’t have any products of that type in the products table:

```sql
SELECT product_type_id, name
FROM product_types outer
WHERE NOT EXISTS
  (SELECT 1
   FROM products inner
   WHERE inner.product_type_id = outer.product_type_id);
```

Notice one row is returned by this example. The next example rewrites the previous query to use NOT IN; notice no rows are returned:

```sql
SELECT product_type_id, name
FROM product_types
WHERE product_type_id NOT IN
  (SELECT product_type_id
   FROM products);
```

No rows are returned because the subquery returns a list of product_id values, one of which is null. The product_type_id for product #12 is null. Because of this, the NOT IN operator in the outer query returns false and therefore no rows are returned. You can get around this by using the NVL() function to convert nulls to a value. In the following example, NVL() is used to convert null product_type_id values to 0:

```sql
SELECT product_type_id, name
FROM product_types
WHERE product_type_id NOT IN
  (SELECT NVL(product_type_id, 0)
   FROM products);
```

PRODUCT_TYPE_ID NAME
------------- -------
       5 Magazine

This time the expected row appears.

### Writing Nested Subqueries

You can nest subqueries inside other subqueries to a depth of 255. But you should use this technique sparingly; you may find your query performs better using table joins. The following example contains a nested subquery.

```
SELECT product_id
FROM products
GROUP BY product_id
HAVING AVG(price) > (SELECT MAX(price) FROM product WHERE product = (SELECT pr FROM purchases WHERE quantity > 0 GROUP BY product))
```

As you can see a subquery, and then subquery, down into the third level:

```
SELECT product_id
FROM purchases
WHERE quantity > 0
```

This subquery more than once.

```
SELECT product_id
FROM products
WHERE product = (SELECT pr FROM purchases WHERE quantity > 0 GROUP BY product)
```

The subquery:

```
SELECT MAX(AVG(price))
FROM products
WHERE product = (SELECT pr FROM purchases WHERE quantity > 0 GROUP BY product)
```

This subquery the previous nesting structures.
When a list of the following products of

\[
\text{SELECT product_type_id, AVG(price)}
\text{FROM products}
\text{GROUP BY product_type_id}
\text{HAVING AVG(price) <}
\text{(SELECT MAX(AVG(price))}
\text{FROM products}
\text{WHERE product_type_id IN}
\text{(SELECT product_id}
\text{FROM purchases}
\text{WHERE quantity > 1)}
\text{GROUP BY product_type_id);}
\]

<table>
<thead>
<tr>
<th>PRODUCT_TYPE_ID</th>
<th>AVG(PRICE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.975</td>
</tr>
<tr>
<td>3</td>
<td>13.24</td>
</tr>
<tr>
<td>4</td>
<td>13.99</td>
</tr>
<tr>
<td></td>
<td>13.49</td>
</tr>
</tbody>
</table>

As you can see, this example is quite complex and contains three queries: a nested subquery, a subquery, and the outer query. These query parts are run in that order. Let's break the example down into the three parts and examine the results returned. The nested subquery is as follows:

\[
\text{SELECT product_id}
\text{FROM purchases}
\text{WHERE quantity > 1}
\]

This subquery returns the product_id values for the products that have been purchased more than once. The rows returned by this subquery are

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

The subquery that receives this output is

\[
\text{SELECT MAX(AVG(price))}
\text{FROM products}
\text{WHERE product_type_id IN}
\text{(... output from previous nested subquery ...)}
\text{GROUP BY product_type_id}
\]

This subquery returns the maximum of the averages of the prices for the products returned by the previous nested subquery. The row returned is

\[
\text{MAX(AVG(PRICE))}
\text{--------------------}
\text{26.22}
\]
This row is returned to the following outer query:

```sql
SELECT product_type_id, AVG(price)
FROM products
GROUP BY product_type_id
HAVING AVG(price) <
    (... output from previous subquery ...);
```

This query returns the `product_type_id` and average price of products that are less than average returned by the previous subquery. The rows returned are

<table>
<thead>
<tr>
<th>PRODUCT_TYPE_ID</th>
<th>AVG(PRICE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24.975</td>
</tr>
<tr>
<td>3</td>
<td>13.24</td>
</tr>
<tr>
<td>4</td>
<td>13.99</td>
</tr>
<tr>
<td>5</td>
<td>13.49</td>
</tr>
</tbody>
</table>

These are, of course, the rows returned by the complete query shown earlier.

### Writing UPDATE and DELETE Statements Containing Subqueries

So far, you've only seen subqueries contained in a `SELECT` statement. As you'll see in this section, you can also use subqueries with `UPDATE` and `DELETE` statements.

**Writing an UPDATE Statement Containing a Subquery**

In an `UPDATE` statement, you set the new column value equal to the result returned by a single row subquery. For example, the following `UPDATE` statement sets employee #4's salary to the average of the high salary grades returned by a subquery:

```sql
UPDATE employees
SET salary =
    (SELECT AVG(high_salary)
     FROM salary_grades)
WHERE employee_id = 4;
```

1 row updated.

This increases employee #4's salary from $500,000 to $625,000 (this is the average of the high salaries from the `salary_grades` table).

**NOTE**

*If you execute the UPDATE statement, remember to execute a ROLLBACK to undo the change.*
Writing a DELETE Statement Containing a Subquery

You use the results returned by the subquery in the WHERE clause of your DELETE statement. For example, the following DELETE statement removes the employee whose salary is greater than the average of the high salary grades returned by a subquery:

```
DELETE FROM employees
WHERE salary >
   (SELECT AVG(high_salary)
    FROM salary_grades);
```

1 row deleted.

This DELETE statement removes employee #1.

**NOTE**

*If you execute the DELETE statement, remember to execute a ROLLBACK to undo the removal of the row.*

**Summary**

In this chapter, you learned that

- A subquery is a query placed within a SELECT, UPDATE, or DELETE statement.
- Single row subqueries return zero or one row.
- Multiple row subqueries return one or more rows.
- Multiple column subqueries return more than one column.
- Correlated subqueries reference one or more columns in the outer SQL statement.
- Nested subqueries are subqueries placed within another subquery.

In the next chapter, you’ll learn about advanced queries.