

# Queries Within Queries

Course: INFO6210 Data Management and Database Design

Week: 7

Instructor: Mutsalklisana

# Queries within Queries

- SQL allows queries within queries, or subqueries, which are SELECT statements inside SELECT statements.
- Subqueries can actually be very useful
  - however, is that they can consume a lot of processing, disk, and memory resources.
- A subquery's syntax is just the same as a normal SELECT query's syntax.
- As with a normal SELECT statement, a subquery can contain joins, WHERE clauses, HAVING clauses, and GROUP BY clauses.
- Subqueries are particularly powerful when coupled with SQL operators such as IN, ANY, SOME, and ALL.
- Note: versions of MySQL prior to version 4.1 do not fully support subqueries

# Subquery Terminology

- In this lecture we'll use references to the outer and inner subqueries.
- The outer query is the main SELECT statement, and you could say that so far all of your SELECT statements have been outer queries.
- Shown below is a standard query:  
`SELECT MemberId FROM Members;`
- Using the standard query, you can nest—that is, place inside the outer query—a subquery, which is termed the inner query:  
`SELECT MemberId FROM MemberDetails  
WHERE MemberId = (SELECT MAX(FilmId) FROM Films);`
- WHERE clause is added to the outer query, and it specifies that MemberId must equal the value returned by the nested inner query, which is contained within brackets

# Nest Subquery

- It is also possible to nest a subquery inside the inner query
- Consider the following example:

```
SELECT MemberId FROM MemberDetails
WHERE MemberId = (SELECT MAX(FilmId) FROM Films
                  WHERE FilmId IN (SELECT LocationId FROM Location));
```

- In the preceding example, a subquery is added to the WHERE clause of the inner query.
- A subquery inside a subquery is referred to as the ***innermost query***

# Subqueries in a SELECT list

- You can include a subquery as one of the expressions returning a value in a SELECT query, just as you can include a single column.
- However, the subquery must return just one record in one expression, in what is known as a scalar subquery.
- The subquery must also be enclosed in brackets. An example:

```
SELECT Category,  
       (SELECT MAX(DVDPrice) FROM Films WHERE Films.CategoryId =  
        Category.CategoryId),  
       CategoryId  
FROM Category;
```

# Subqueries in a SELECT list

- The SELECT query starts off by selecting the Category column, much as you've already seen many times before.
- However, the next item in the list is not another column but rather a subquery.
- This query inside the main query returns the maximum price of a DVD.
- An aggregate function returns only one value, complying with the need for a subquery in a SELECT statement to be a scalar subquery.
- The subquery is also linked to the outer SELECT query using a WHERE clause.
- Because of this link, MAX(DVDPrice) returns the maximum price for each category in the Category table

# DVD Category and Price Table

- Result of Subquery with SELECT

Category	DVDPrice	Category.CategoryId
Thriller	12.99	1
Romance	12.99	2
Horror	9.99	3
War	12.99	4
Sci-fi	12.99	5
Historical	15.99	6
Comedy	NULL	7
Film Noir	NULL	9

# What do we get if whole query is executed

- Starting with the first row in the results, the **Category** is ***Thriller*** and the **CategoryId** is **1**
- The subquery is joined to the outer query by the CategoryId column present in both the Films and Category tables
- For the first row, the CategoryId is 1, so the subquery finds the maximum DVDPrice for all films in the Films table where the CategoryId is 1
- Moving to the next row in the outer query, the Category is Romance and the CategoryId is 2. This time, the subquery finds the maximum DVDPrice for all records where the CategoryId is 2
- The process continues for every row in the Category table.



Without the `WHERE` clause in the subquery linking the subquery to the outer query, the result would simply be the maximum value of all rows returned by the subquery. If you change the SQL and remove the `WHERE` clause in the subquery, you get the following statement:

```
SELECT Category,  
       (SELECT MAX(DVDPrice) FROM Films),  
       CategoryId  
FROM Category;
```

Executing this query provides the following results:

Category	MAX(DVDPrice)	Category.CategoryId
Thriller	15.99	1
Romance	15.99	2
Horror	15.99	3
War	15.99	4
Sci-fi	15.99	5
Historical	15.99	6
Comedy	15.99	7
Film Noir	15.99	9

`MAX(DVDPrice)` is now simply the maximum `DVDPrice` for all records in the `Films` table and is not specifically related to any category.

Although aggregate functions such as MAX, MIN, AVG, and so on are ideal for subqueries because they return just one value, any expression or column is suitable as long as the results set consists of just one row. For example, the following subquery works because it returns only one row:

```
SELECT FilmName, PlotSummary, (SELECT Email FROM MemberDetails WHERE MemberId = 1)
FROM Films;
```

MemberId is unique in the MemberDetails table, and therefore WHERE MemberId = 1 returns only one row, and the query works:

FilmName	PlotSummary	Email
The Dirty Half Dozen	Six men go to war wearing unwashed uniforms. The horror!	katie@mail.com
On Golden Puddle	A couple finds love while wading through a puddle.	katie@mail.com
The Lion, the Witch, and the Chest of Drawers	A fun film for all those interested in zoo/ magic/furniture drama.	katie@mail.com
Nightmare on Oak Street, Part 23	The murderous Terry stalks Oak Street.	katie@mail.com
The Wide-Brimmed Hat	Fascinating life story of a wide-brimmed hat	katie@mail.com
Sense and Insensitivity	She longs for a new life with Mr. Arcy; he longs for a small cottage in the Hamptons.	katie@mail.com
Planet of the Japes	Earth has been destroyed, to be taken over by a species of comedians.	katie@mail.com
The Maltese Poodle	A mysterious bite m... guilty-looking poodle. ... class thriller	katie@mail.com

# Subquery returning more than 1 value

- If, however, you change the MemberDetails table to the Attendance table, where MemberId appears more than once, you get the following query:

```
SELECT FilmName, PlotSummary,  
(SELECT MeetingDate FROM Attendance WHERE MemberId = 1)  
FROM Films;
```

- Executing this query results in an error message similar to the following:

“Subquery returned more than 1 value. This is not permitted when the subquery follows =, !=, <, <=, >, >= or when the subquery is used as an expression.”

# Linking inner query to outer query

- An alternative to using a literal value would be to link the inner query to the outer query, so long as the result of the inner query produces only one row.
- For example, the following query shows the CategoryId from the FavCategory table, and in the subquery, it shows FirstName from the MemberDetails table.
- The MemberId in the subquery is linked to the FavCategory table's MemberId for each row in the outer query—thereby ensuring that the subquery returns only one row, as MemberIds are unique in the MemberDetails table:

```
SELECT CategoryId,  
(SELECT FirstName FROM MemberDetails WHERE MemberId =  
FavCategory.MemberId)  
FROM FavCategory;
```

CategoryId	FirstName
1	John
1	Susie
1	Stuart
2	Katie

# Subquery versus Join

- Join work just as well and in some cases more efficiently than a subquery
- In general, it would make more sense to use a join, but we are doing it here to demonstrate how to use a subquery!

# Subqueries in WHERE Clause

- Subqueries are at their most useful in WHERE clauses
- The syntax and form that subqueries take in WHERE clauses are identical to how they are used in SELECT statements, with one difference: now ***subqueries are used in comparisons***
- Example:
  - Imagine that you need to find out the name or names, if there are two or more of identical value, of the cheapest DVDs for each category
  - You want the results to display the category name, the name of the DVD, and its price
  - The data you need comes from the Category and Films tables

# How to choose only cheapest DVD from each Category

- You could use a GROUP BY clause, as shown in the following query:

```
SELECT Category, MIN(DVDPrice)
FROM Category INNER JOIN Films
ON Category.CategoryId = Films.CategoryId
GROUP BY Category;
```

- The query's results, however, don't supply the name of the film, just the category name and the price of the cheapest DVD:

Category	MIN(DVDPrice)
Historical	8.95
Horror	8.95
Romance	12.99
Sci-fi	12.99
Thriller	2.99
War	12.99



- Add FilmName to the list of columns in the SELECT statement.
- Category and FilmName must appear in the GROUP BY clause as shown below:

```
SELECT Category, FilmName, MIN(DVDPrice)
FROM Category INNER JOIN Films
ON Category.CategoryId = Films.CategoryId
GROUP BY Category, FilmName;
```

Category	FilmName	MIN(DVDPrice)
Historical	15th Late Afternoon	NULL
Historical	Gone with the Window Cleaner	9.99
Horror	Nightmare on Oak Street, Part 23	9.99
Romance	On Golden Puddle	12.99
Horror	One Flew over the Crow's Nest	8.95
War	Planet of the Japes	12.99
Thriller	Raging Bullocks	NULL
Historical	Sense and Insensitivity	15.99
Sci-fi	Soylent Yellow	12.99
War	The Dirty Half Dozen	NULL
Historical	The Good, the Bad, and the Facially Challenged	8.95
Thriller	The Life of Bob	12.99
Horror	The Lion, the Witch, and the Chest of Drawers	NULL
Thriller	The Maltese Poodle	2.99
Sci-fi	The Wide-Brimmed Hat	NULL

- The results are wrong because they're grouped by Category and FilmName, so the MIN(DVDPrice) value is not the minimum price for a particular category but rather the minimum price for a particular film in a particular category!



# A list of all the lowest prices for a DVD per Category

- In the SQL query, you need to compare the price of a DVD with the minimum price for a DVD in that category, and you want the query to return a record only if they match:

```
SELECT Category, FilmName, DVDPrice
FROM Category INNER JOIN Films
ON Category.CategoryId = Films.CategoryId
WHERE Films.DVDPrice =
      (SELECT MIN(DVDPrice) FROM Films WHERE Films.CategoryId = Category.CategoryId);
```

Category	FilmName	DVDPrice
Thriller	The Maltese Poodle	2.99
Romance	On Golden Puddle	12.99
Horror	One Flew over the Crow's Nest	8.95
War	Planet of the Japes	12.99
Sci-fi	Soylent Yellow	12.99
Historical	The Good, the Bad, and the Facially Challenged	8.95

# Operators in Subqueries

- So far, all the subqueries we have discussed have been scalar subqueries
  - that is, queries that only return only one row.
- If more than one row is returned, you end up with an error.
- Next, we'll discuss about operators that allow you to make comparisons against a multi-record results set

# IN Operator

- The IN operator allows you to specify that you want to match one item from any of those in a list of items.
- For example, the following SQL finds all the members born in 1967, 1992, or 1937:

```
SELECT FirstName, LastName, YEAR(DateOfBirth)  
FROM MemberDetails  
WHERE YEAR(DateOfBirth) IN (1967, 1992, 1937);
```

FirstName	LastName	YEAR(DateOfBirth)
Steve	Gee	1967
Susie	Simons	1937
Jamie	Hills	1992

*Note that this code, and any code that contains the YEAR() function, won't work in Oracle because it doesn't support the YEAR() function*

# IN operator could be used with subqueries

- Instead of providing a list of literal values, a SELECT query provides the list of values. For example, if you want to know which members were born in the same year that a film in the Films table was released, you'd use the following SQL query Again, these examples won't work in Oracle because it doesn't support the YEAR ) function:

```
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails
WHERE YEAR(DateOfBirth) IN (SELECT YearReleased FROM Films);
```

- Executing this query gives the following results:

FirstName	LastName	YEAR(DateOfBirth)
Katie	Smith	1977
Steve	Gee	1967
Doris	Night	1997

- The subquery (SELECT YearReleased FROM Films) returns a list of years from the Films table. If a member's year of birth matches one of the items in that list, then the WHERE clause is true and the record is included in the final results

# JOIN versus Subquery

- An INNER JOIN coupled with a GROUP BY statement could be used instead, as shown below:

```
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails JOIN Films ON YEAR(DateOfBirth) = YearReleased
GROUP BY FirstName, LastName, YEAR(DateOfBirth);
```

- Running this query gives the same results as the previous query. So which is best?
- Unfortunately, there's no definitive answer; very much depends on the circumstances, the data involved, and the database system involved
- A lot of SQL programmers prefer a join to a subquery and believe that to be the most efficient!

# JOIN versus Subquery – Cont'd

- However, if you compare the speed of the two using MS SQL Server 2000, in this case, on that system, the subquery is faster by roughly 15 percent.
- Given how few rows there are in the database, the difference was negligible in this example, but it might be significant with a lot of records.
- Which way should you go?
- You should go with the way that you find easiest, and fine-tune your SQL code only if problems occur during testing.
- If you find on a test system with a million records that your SQL runs like an arthritic snail with heavy shopping, then you should go back and see whether you can improve your query

There is one area in which subqueries are pretty much essential: when you want to find something is not in a list, something very hard to do with joins. For example, if you want a list of all members who were not born in the same year that any of the films in the Films table were released, you'd simply change your previous subquery example from an IN operator to a NOT IN operator:

```
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails
WHERE YEAR(DateOfBirth) NOT IN (SELECT YearReleased FROM Films);
```

The new query gives the following results:

FirstName	LastName	YEAR(DateOfBirth)
John	Jones	1952
Jenny	Jones	1953
John	Jackson	1974
Jack	Johnson	1945
Seymour	Botts	1956
Susie	Simons	1937
Jamie	Hills	1992
Stuart	Dales	1956
William	Doors	1994

Now a match occurs only if YEAR(DateOfBirth) is not found in the list produced by the subquery SELECT YearReleased FROM Films.

# Getting same result using JOIN (OUTER)

```
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails
LEFT OUTER JOIN Films
ON Films.YearReleased = Year(MemberDetails.DateOfBirth)
WHERE YearReleased IS NULL;
```

- The query produces almost identical results, except for any member, whose date of birth is NULL.
- The way the query works is that the left outer join returns all rows from MemberDetails, regardless of whether YearReleased from the Films table finds a match in the MemberDetails table.
- NULL is returned when a match isn't found.
- NULLs indicate that YEAR(DateOfBirth) isn't found in the Films table, so by adding a WHERE clause that returns rows only when there's a NULL value in YearReleased, you make sure that the query will find all the rows in MemberDetails where there's no matching year of birth in the Films table's YearReleased column.
- The query below modifies the SQL to remove the WHERE clause and show the YearReleased column

```
SELECT FirstName, LastName, YearReleased
FROM MemberDetails
LEFT OUTER JOIN Films
ON Films.YearReleased = Year(MemberDetails.DateOfBirth)
ORDER BY YearReleased;
```



If you execute this query, you can see how the results came about:

FirstName	LastName	YearReleased
John	Jones	NULL
Jenny	Jones	NULL
John	Jackson	NULL
Jack	Johnson	NULL
Seymour	Botts	NULL
Susie	Simons	NULL
Jamie	Hills	NULL
Stuart	Dales	NULL
William	Doors	NULL
Catherine	Hawthorn	NULL
Steve	Gee	1967
Steve	Gee	1967
Steve	Gee	1967
Katie	Smith	1977
Doris	Night	1997

The advantage of using an outer join in this situation is that it's quite often more efficient, which can make a big difference when there are a lot of records involved. The other advantage if you're using MySQL before version 4.1 is that subqueries are not supported, so an outer join is the only option. The advantage of using subqueries is that they are easier to write and easier to read.

# ANY, SOME and ALL Operators

- The IN operator allows a simple comparison to see whether a value matches one in a list of values returned by a subquery.
- 
- The ***ANY, SOME, and ALL*** operators not only allow an equality match but also allow any comparison operator to be used

# ANY and SOME Operators

- First, **ANY and SOME** are identical; they do the same thing but have a different name.
- The examples refer to the **ANY** operator, but you can use the **SOME** operator without it making a bit of difference.
- a For **ANY** to return true to a match, the value being compared needs to match any one of the values returned by the subquery.
- You must place the comparison operator before the **ANY** keyword.
- For example, the following SQL uses the equality (=) operator to find out if any members have the same birth year as the release date of a film in the Films table:

```
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails
WHERE YEAR(DateOfBirth) =ANY (SELECT YearReleased FROM Films);
```

- a The WHERE clause specifies that YEAR(DateOfBirth) must equal any one of the values returned by the subquery (SELECT YearReleased FROM Films)

# ANY and SOME Operators

If `YEAR(DateOfBirth)` equals any one of these values, then the condition returns true and the `WHERE` clause allows the record into the final results. The final results of the query are as follows:

FirstName	LastName	YEAR(DateOfBirth)
Katie	Smith	1977
Steve	Gee	1967
Doris	Night	1997

Seem familiar? Yup, that's right, the results from `= ANY` are the same as using the `IN` operator:

```
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails
WHERE YEAR(DateOfBirth) IN (SELECT YearReleased FROM Films);
```

To obtain the same results as you would with the NOT IN operator, simply use the not equal (<>) operator, as shown in the following code:

```
SELECT FirstName, LastName, YEAR(DateOfBirth)
FROM MemberDetails
WHERE YEAR(DateOfBirth) <> ANY (SELECT YearReleased FROM Films);
```

Before you write off the ANY operator as just another way of using the IN operator, remember that you can use ANY with operators other than equal (=) and not equal (<>). The following example query finds a list of members who, while they were members, had the opportunity to attend a meeting:

```
SELECT FirstName, LastName
FROM MemberDetails
WHERE DateOfJoining < ANY (SELECT MeetingDate FROM Attendance);
```

The query checks to see whether the member joined before any one of the meeting dates. This query uses the less than (<) operator with ANY, and the condition evaluates to true if a member's DateOfJoining is less than any of the values returned by the subquery (SELECT MeetingDate FROM Attendance).

The results are shown in the following table:

FirstName	LastName
Katie	Smith
Steve	Gee

# ALL Operator

## ALL Operator

The ALL operator requires that every item in the list (all the results of a subquery) comply with the condition set by the comparison operator used with ALL. For example, if a subquery returns 3, 9, and 15, then the following WHERE clause would evaluate to true because 2 is less than all the numbers in the list:

```
WHERE 2 < ALL (3, 9, 15)
```

However, the following WHERE clause would evaluate to false because 7 is not less than all of the numbers in the list:

```
WHERE 7 < ALL (3, 9, 15)
```

This is just an example, though, and you can't use ALL with literal numbers, only with a subquery.

Put the ALL operator into an example where you select MemberIds that are less than all the values returned by the subquery (SELECT FilmId FROM Films WHERE FilmId > 5):

```
SELECT MemberId
FROM MemberDetails
WHERE MemberId < ALL (SELECT FilmId FROM Films WHERE FilmId > 5);
```

# ALL Operator

The subquery (`SELECT FilmId FROM Films WHERE FilmId > 5`) returns the following values:

FilmId
6
7
8
9
10
11
12
13
14
15

Essentially, this means that the WHERE clause is as follows:

```
MemberId < ALL (6,7,8,9,10,11,12,13,14,15);
```

# ALL Operator

Essentially, this means that the WHERE clause is as follows:

```
MemberId < ALL (6,7,8,9,10,11,12,13,14,15);
```

MemberId must be less than all of the listed values if the condition is to evaluate to true. The full results for the example query are displayed in the following table:

MemberId
1
4
5

There is something to be aware of when using ALL: the situation when the subquery returns no results at all. In this case, ALL will be true, which may seem a little weird, but it's based on fundamental principles of logic. So, if you change the subquery so that it returns no values and update the previous example, you end up with the following code:

```
SELECT MemberId  
FROM MemberDetails  
WHERE MemberId < ALL (SELECT FilmId FROM Films WHERE FilmId > 99);
```

You know for a fact that there are no FilmIds with a value higher than 99, so the subquery returns an empty set. However, the results set for the example is shown in the following table:



# ALL Operator

MemberId
1
10
14
6
5
8
7
11
15
4
13
12
9

The table represents every single row in the MemberDetails table; indeed, `MemberId < ALL (SELECT FilmId FROM Films WHERE FilmId > 99)` has evaluated to true every time due to an empty results set returned by the subquery.

# ALL Operator

## Using the EXISTS Operator

The EXISTS operator is unusual in that it checks for rows and does not compare columns. So far you've seen lots of clauses that compare one column to another. On the other hand, EXISTS simply checks to see whether a subquery has returned one or more rows. If it has, then the clause returns true; if not, then it returns false.

This is best demonstrated with three very simple examples:

```
SELECT City
FROM Location
WHERE EXISTS (SELECT * FROM MemberDetails WHERE MemberId < 5);

SELECT City
FROM Location
WHERE EXISTS (SELECT * FROM MemberDetails WHERE MemberId > 99);

SELECT City
FROM Location
WHERE EXISTS (SELECT * FROM MemberDetails WHERE MemberId = 15);
```

Notice the use of the asterisk (\*), which returns all columns, in the inner subqueries. EXISTS is row-based, not column-based, so it doesn't matter which columns are returned.

# EXISTS Operator

The first example uses the following `SELECT` query as its inner subquery:

```
(SELECT * FROM MemberDetails WHERE MemberId < 5);
```

This subquery provides a results set with three rows. Therefore, `EXISTS` evaluates to true, and you get the following results:

City
Orange Town
Windy Village
Big City

The second example uses the following inner subquery:

```
SELECT * FROM MemberDetails WHERE MemberId > 99
```

This returns no results because no records in `MemberDetails` have a `MemberId` greater than 99. Therefore, `EXISTS` returns false, and the outer `SELECT` statement returns no results.

# EXISTS Operator

Finally, the third example uses the following query as its inner subquery:

```
SELECT * FROM MemberDetails WHERE MemberId = 15
```

This query returns just one row, and even though some of the columns contain `NULL` values, it's still a valid row, and `EXISTS` returns `true`. In fact, even if the whole row contained `NULL`s, `EXISTS` would still return `true`. Remember, `NULL` doesn't mean no value; it means an unknown value. So, the results for the full query in the example are as follows:

City
Orange Town
Windy Village
Big City

# NOT EXISTS Operator

You can reverse the logic of `EXISTS` by using the `NOT` operator, essentially checking to see whether no results are returned by the subquery. Modify the second example described previously, where the subquery returns no results, adding a `NOT` operator to the `EXISTS` keyword:

```
SELECT City
FROM Location
WHERE NOT EXISTS (SELECT * FROM MemberDetails WHERE MemberId > 99);
```

Now `NOT EXISTS` returns true, and as a result the `WHERE` clause is also true. The final results are shown in the following table:

City
Orange Town
Windy Village
Big City

# Sample Coding

Build the necessary query by following these steps:

1. First of all, formulate a `SELECT` statement that returns the values you want. In this case, you just want the `Category` from the `Category` table:

```
SELECT Category
FROM Category
```

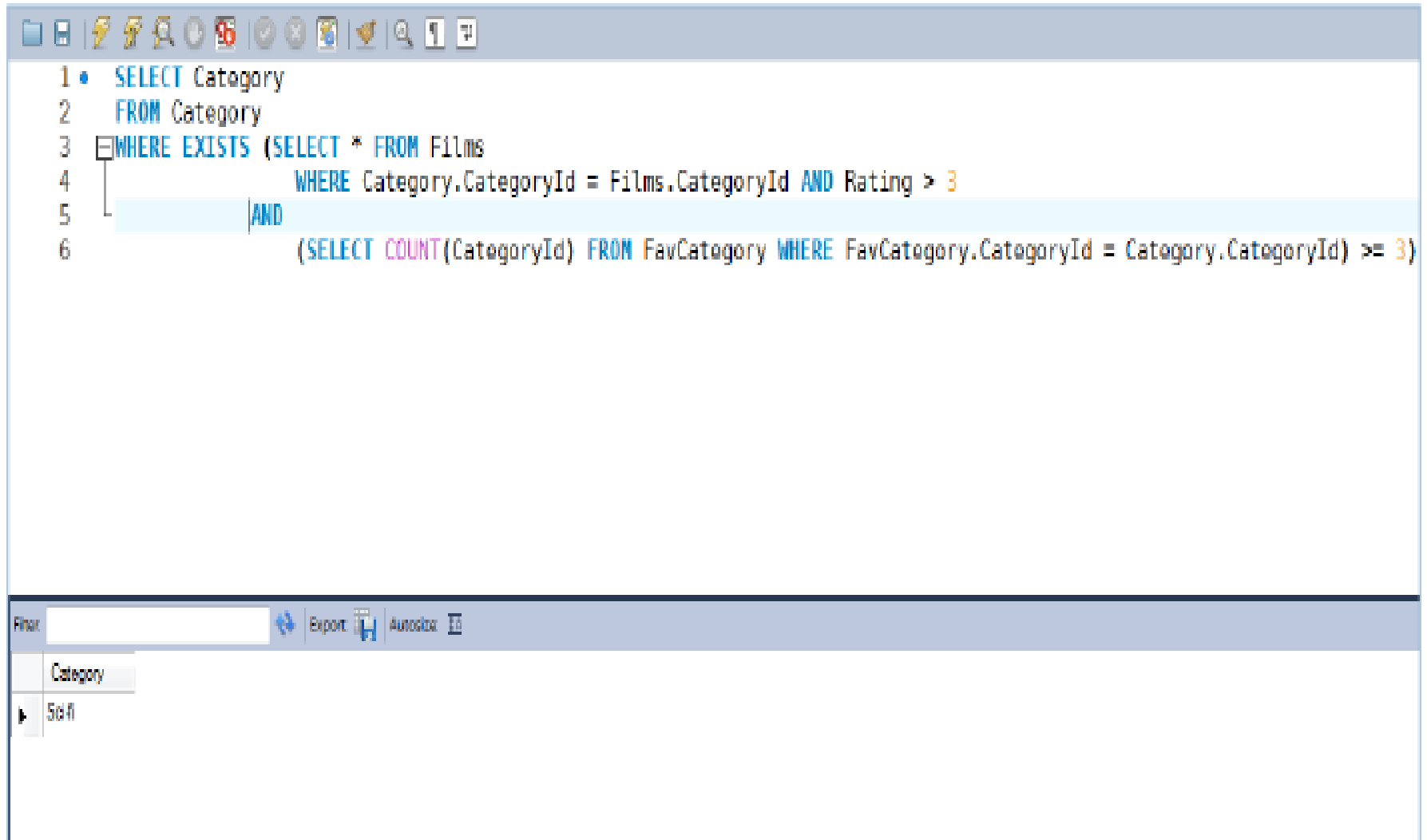
2. Now you need to add a `WHERE` clause to ensure that you get only categories that have films rated 4 or higher and that have also been selected as a favorite by three or more members.

```
SELECT Category
FROM Category
WHERE EXISTS (SELECT * FROM Films
              WHERE Category.CategoryId = Films.CategoryId
              AND Rating > 3
              )
```

3. Finally, you just want categories that three or more members have chosen as their favorite.

```
SELECT Category
FROM Category
WHERE EXISTS (SELECT * FROM Films
              WHERE Category.CategoryId = Films.CategoryId
              AND Rating > 3
              AND (SELECT COUNT(CategoryId)
                  FROM FavCategory
                  WHERE FavCategory.CategoryId = Category.CategoryId)
              >= 3
              );
```

# Sample Coding – Cont'd



The screenshot shows a SQL IDE window. The top toolbar contains icons for file operations, execution, and debugging. The main text area displays a SQL query with line numbers 1 through 6. The query is a SELECT statement that filters categories based on the existence of films with a rating greater than 3 and the count of favorite categories. The bottom toolbar includes buttons for 'Find', 'Export', and 'Autosave'. Below the toolbar, a results pane shows a table with two columns: 'Category' and '508 fi'.

```
1 • SELECT Category
2   FROM Category
3  WHERE EXISTS (SELECT * FROM Films
4                WHERE Category.CategoryId = Films.CategoryId AND Rating > 3
5                AND
6                (SELECT COUNT(CategoryId) FROM FavCategory WHERE FavCategory.CategoryId = Category.CategoryId) >= 3)
```

Category
508 fi

# HAVING Clause

- The **HAVING** clause was added to SQL because the WHERE keyword could not be used with aggregate functions.
- SQL **HAVING** Syntax

SELECT column name, aggregate function(column name)  
FROM table name  
WHERE column name operator value  
GROUP BY column name  
HAVING aggregate\_function(column\_name) operator value



# SQL HAVING Coding Example

- Assume you create the following table:

O_Id	OrderDate	OrderPrice	Customer
1	2008/11/12	1000	Hansen
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Hansen
4	2008/09/03	300	Hansen
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

- Now we want to find if any of the customers have a total order of less than 2000. We use the following SQL statement:

```
SELECT Customer,SUM(OrderPrice) FROM Orders  
GROUP BY Customer  
HAVING SUM(OrderPrice)<2000
```

- The result will look like this

Customer	SUM(OrderPrice)
Nilsen	1700

## Using the *HAVING* Clause with Subqueries

```
SELECT City
FROM MemberDetails
GROUP BY City
HAVING AVG(YEAR(DateOfBirth)) > 1990;
```

The preceding query compares against an actual value, 1990. A subquery is useful where you want to compare against a value extracted from the database itself rather than a predetermined value. For example, you might be asked to create a list of cities where the average year of birth is later than the average for the membership as a whole. To do this, you could use a *HAVING* clause plus a subquery that finds out the average year of birth of members:

```
SELECT City
FROM MemberDetails
GROUP BY City
HAVING AVG(YEAR(DateOfBirth)) >
    (SELECT AVG(YEAR(DateOfBirth)) FROM MemberDetails);
```

This is the same as the query just mentioned, except 1990 is replaced with a subquery that returns the average year of birth of all members, which happens to be 1965. The final results table is as follows:

City
------

Big City
----------

Dover
-------

New Town
----------

Orange Town
-------------

# Correlated Subquery

Sometimes it's necessary to refer to the outer query, or even a query nested inside a subquery. To do this, you must give aliases to the tables involved

*A correlated subquery* is a subquery that references the outer query. A *correlation variable* is an alias given to tables and used to reference those tables in the subquery. In Chapter 3 you learned how to give tables a correlation, or alias name.

The following example demonstrates a correlated subquery. This example isn't necessarily the only way to get the result, but it does demonstrate a nested subquery. The query obtains from each category the cheapest possible DVD with the highest rating:

```
SELECT FilmName, Rating, DVDPrice, Category
FROM Films AS FM1 INNER JOIN Category AS C1 ON C1.CategoryId = FM1.CategoryId
WHERE FM1.DVDPrice =
    (SELECT MIN(DVDPrice)
     FROM Films AS FM2
     WHERE FM2.DVDPrice IS NOT NULL
           AND FM1.CategoryId = FM2.CategoryId
           AND FM2.Rating =
               (SELECT MAX(FM3.Rating)
                FROM Films AS FM3
                WHERE FM3.DVDPrice IS NOT NULL
                      AND FM2.CategoryId = FM3.CategoryId)
     GROUP BY FM2.CategoryId)
ORDER BY FM1.CategoryId;
```

*If you're using Oracle, remove the AS keywords because Oracle doesn't require or support them when creating an alias.*

This is a big query, but when broken down it's actually fairly simple. There are three queries involved: the outer query and two inner subqueries. Begin with the most nested inner query:

```
(SELECT MAX(FM3.Rating)
  FROM Films AS FM3
 WHERE FM3.DVDPrice IS NOT NULL
       AND FM2.CategoryId = FM3.CategoryId
 GROUP BY FM3.CategoryId)
```

This inner query returns the highest-rated film within each category. Films that have no price (where DVDPrice is NULL, for example) are excluded, because you don't want them forming part of the final results. The Films table is given the alias FM3. The query also refers to the FM2 table, which is the alias given to the subquery in which this query is nested. The CategoryId columns of FM2 and FM3 are inner-joined to ensure that the MAX rating and MIN price are both from the same category. If you expand the SQL and now include the innermost query just described, and also the nested subquery that it was inside, you arrive at the following SQL:

```
(SELECT MIN(DVDPrice)
  FROM Films AS FM2
 WHERE FM2.DVDPrice IS NOT NULL
       AND FM1.CategoryId = FM2.CategoryId
       AND FM2.Rating =
         (SELECT MAX(FM3.Rating)
          FROM Films AS FM3
         WHERE FM3.DVDPrice IS NOT NULL
               AND FM2.CategoryId = FM3.CategoryId
          GROUP BY FM3.CategoryId)
 GROUP BY FM2.CategoryId)
```

# Correlated Subquery

The outer query here returns the minimum DVD price for the highest-rated film in each category. Remember that the innermost subquery returns the rating of the highest-rated DVD. The outer query's WHERE clause is set to specify that the film should also have a rating equal to the one returned by the subquery — thereby ensuring that it has a rating equal to the highest in the same category. Note that although an inner query can refer to any alias of a table outside the query, an outer query can't reference any tables in a query nested inside it. For example, this SQL won't work:

```
(SELECT MIN(DVDPrice)
  FROM Films AS FM2
 WHERE FM2.CategoryId = FM3.CategoryId
 ...
 ...
 ...
```

The table with alias FM3 is referenced, but that table is a nested subquery inside, and therefore it can't be referenced.

This is a big query, but when broken down it's actually fairly simple. There are three queries involved: the outer query and two inner subqueries. Begin with the most nested inner query:

```
(SELECT MAX(FM3.Rating)
  FROM Films AS FM3
 WHERE FM3.DVDPrice IS NOT NULL
    AND FM2.CategoryId = FM3.CategoryId
 GROUP BY FM3.CategoryId)
```

This inner query returns the highest-rated film within each category. Films that have no price (where DVDPrice is NULL, for example) are excluded, because you don't want them forming part of the final results. The Films table is given the alias FM3. The query also refers to the FM2 table, which is the alias given to the subquery in which this query is nested. The CategoryId columns of FM2 and FM3 are inner-joined to ensure that the MAX rating and MIN price are both from the same category. If you expand the SQL and now include the innermost query just described, and also the nested subquery that it was inside, you arrive at the following SQL:

```
(SELECT MIN(DVDPrice)
  FROM Films AS FM2
 WHERE FM2.DVDPrice IS NOT NULL
    AND FM1.CategoryId = FM2.CategoryId
    AND FM2.Rating =
      (SELECT MAX(FM3.Rating)
        FROM Films AS FM3
       WHERE FM3.DVDPrice IS NOT NULL
          AND FM2.CategoryId = FM3.CategoryId
        GROUP BY FM3.CategoryId)
 GROUP BY FM2.CategoryId)
```



# Correlated Subquery

The outer query here returns the minimum DVD price for the highest-rated film in each category. Remember that the innermost subquery returns the rating of the highest-rated DVD. The outer query's WHERE clause is set to specify that the film should also have a rating equal to the one returned by the subquery — thereby ensuring that it has a rating equal to the highest in the same category. Note that although an inner query can refer to any alias of a table outside the query, an outer query can't reference any tables in a query nested inside it. For example, this SQL won't work:

```
(SELECT MIN(DVDPrice)
  FROM Films AS FM2
  WHERE FM2.CategoryId = FM3.CategoryId
  ...
  ...
  ...)
```

The table with alias FM3 is referenced, but that table is a nested subquery inside, and therefore it can't be referenced.

Finally, the entire query, with the outermost `SELECT` statement, is as follows:

```
SELECT FilmName, Rating, DVDPrice, Category
FROM Films AS FM1 INNER JOIN Category AS C1 ON C1.CategoryId = FM1.CategoryId
WHERE FM1.DVDPrice =
    (SELECT MIN(DVDPrice)
      ...
      ...
      ...
     ORDER BY FM1.CategoryId;
```

The outermost query is a join between the `Films` table with an alias of `FM1` and the `Category` table with an alias of `C1`. A `WHERE` clause specifies that `FM1.DVDPrice` must be equal to the DVD price returned by the subquery, the lowest price of a DVD in that category. This inner query, as you've seen, has a `WHERE` clause specifying that the film's rating must be equal to the highest-rated film for that category, as returned by the innermost subquery. So, three queries later, you have the lowest-priced, highest-rated film for each category. The results are displayed in the following table:

FilmName	Rating	DVDPrice	Category
The Maltese Poodle	1	2.99	Thriller
On Golden Puddle	4	12.99	Romance
One Flew over the Crow's Nest	2	8.95	Horror
Planet of the Japes	5	12.99	War
Soylent Yellow	5	12.99	Sci-fi
The Good, the Bad, and the Facially Challenged	5	8.95	Historical



# CAUTION: INSERT INTO & SELECT

- It's quite important not to make a mistake when using INSERT INTO with SELECT queries, or else a whole load of incorrect data might get inserted.
- Unless your query is very simple, it's preferable to ***create the SELECT part of the SQL first***, double-check that it is giving the correct results, and ***then add the INSERT INTO bit***, which is very simple anyway.
- For the SELECT part of the query, you need to extract a MemberId from the MemberDetails table, where that member has selected the Thriller category, that is, where CategoryId equals 1.
- To prevent errors, you need to make sure that the member hasn't already selected the Film Noir category, a CategoryId of 9.
- It's playing safe, as currently no one has selected the Film Noir category, but just in case someone updates the database in the meantime, you should check that a member hasn't selected CategoryId 9 as a favorite before trying to make CategoryId 9 one of their favorites.

In order to create the necessary statement, follow these steps:

1. Begin by building the outer part of the SELECT query:

```
SELECT 9, MemberId FROM MemberDetails AS MD1;
```

This portion of the query returns all the rows from the MemberDetails table. Oracle doesn't support the AS keyword as a way of defining an alias — it just needs the alias name — so remove the AS after MemberDetails if you're using Oracle.

2. Now add the first part of your WHERE clause that checks to see whether the member has selected Thriller as one of their favorite categories:

```
SELECT 9, MemberId FROM MemberDetails AS MD1
WHERE EXISTS
    (SELECT * from FavCategory FC1
     WHERE FC1.CategoryId = 1 AND FC1.MemberId = MD1.MemberId);
```

3. Now modify the subquery so that it returns rows only if the member hasn't already selected CategoryId 9 as one of their favorites:

```
SELECT 9, MemberId FROM MemberDetails AS MD1
WHERE EXISTS
    (SELECT * from FavCategory FC1
     WHERE FC1.CategoryId = 1 AND FC1.MemberId = MD1.MemberId
     AND NOT EXISTS
        (SELECT * FROM FavCategory AS FC2
         WHERE FC2.MemberId = FC1.MemberId AND
         FC2.CategoryId = 9));
```

Notice the nested subquery inside the other subquery that checks that there are no rows returned (that they do not exist), where the current MemberId has selected a favorite CategoryId of 9. Execute this query and you get the following results:

Literal Value of 9	MemberId
9	5
9	10
9	12

4. Next, quickly double-check a few of the results to see whether they really are correct. Having confirmed that they are, you can add the INSERT INTO bit:

```
INSERT INTO FavCategory (CategoryId, MemberId)
SELECT 9, MemberId FROM MemberDetails AS MD1
WHERE EXISTS
    (SELECT * from FavCategory FC1
     WHERE FC1.CategoryId = 1 AND FC1.MemberId = MD1.MemberId
     AND NOT EXISTS
         (SELECT * FROM FavCategory AS FC2
          WHERE FC2.MemberId = FC1.MemberId AND
                FC2.CategoryId = 9));
```

The INSERT INTO inserts the literal value 9 (representing the Film Noir's CategoryId) and MemberId into the FavCategory table. When you execute the SQL, you should find that four rows are added. Execute it a second, a third, or however many times, and no more rows will be added as the query checks for duplication, a good safeguard.

## Using Subqueries with the UPDATE Statement

As with `INSERT INTO`, you can use subqueries to supply values for updating or to determine the `WHERE` clause condition in the same way that subqueries can be used with regular queries' `WHERE` clauses. An example makes this clear. Imagine that the film club chairperson has decided to boost profits by selling DVDs. To maximize profits, she wants films that are rated higher than 3 and that appear in four or more members' favorite categories to have their prices hiked up to that of the highest price in the database.

To tackle this query, you need to break it down. First, the new `DVDPrice` is to be the maximum price of any film in the `Films` table. For this, you need a subquery that returns the maximum `DVDPrice` using the `MAX()` function. Note that neither MS Access nor MySQL support updating a column using a subquery:

```
SELECT MAX(DVDPrice) FROM Films;
```

This query is used as a subquery in the `UPDATE` statement's `SET` statement, but before executing this statement, you need to add the `WHERE` clause:

```
UPDATE Films  
SET DVDPrice = (SELECT MAX(DVDPrice) FROM Films);
```

*You should create and test the `WHERE` clause inside a `SELECT` query before adding the `WHERE` clause to the `UPDATE` statement. A mistake risks changing, and potentially losing, a lot of `DVDPrice` data that shouldn't be changed.*

# Using Subqueries with the *INSERT* Statement

You no longer have to insert literal values, like this

```
INSERT INTO FavCategory (CategoryId, MemberId) VALUES (7,15)
```

Instead, you can use data from the database:

```
INSERT INTO FavCategory (CategoryId, MemberId) SELECT 7, MemberId FROM  
MemberDetails WHERE LastName = 'Hawthorn' AND FirstName = 'Catherine';
```

You can take this one step further and use subqueries to provide the data to be inserted. The film club chairperson has discovered that no one has selected the Film Noir category as their favorite. She's decided that people who like thrillers might also like film noir, so she wants all members who put the Thriller category down as a favorite to also now have Film Noir added as one of their favorite categories.

Film Noir has a CategoryId of 9; Thriller has a CategoryId of 1. You could use SQL to extract these values, but to keep the SQL slightly shorter and clearer for an example, use the CategoryId values without looking them up.



Now create the *WHERE* clause. It needs to limit rows to those where the film's rating is higher than 3 and where the film is in a category selected as a favorite by three or more members. You also need to check whether the film is available on DVD—no point updating prices of films not for sale! The rating and availability parts are easy enough. You simply need to specify that the *AvailableOnDVD* column be set to *Y* and that the *Films.Rating* column be greater than 3:

```
SELECT CategoryId, FilmName FROM Films
WHERE AvailableOnDVD = 'Y' AND Films.Rating > 3;
```

Now for the harder part: selecting films that are in a category chosen as a favorite by three or more members. For this, use a subquery that counts how many members have chosen a particular category as their favorite:

```
SELECT CategoryId, FilmName FROM Films
WHERE (SELECT COUNT(*) FROM FavCategory
       WHERE FavCategory.CategoryId = Films.CategoryId) >= 3
AND AvailableOnDVD = 'Y' AND Films.Rating > 3;
```

The *COUNT(\*)* function counts the number of rows returned by the subquery; it counts how many members have chosen each category as their favorite. The *WHERE* clause of the subquery makes sure that the *Films* table in the outer query is linked to the *FavCategory* table of the inner query.

Execute this query and you should get the following results:

CategoryId	FilmName
4	Planet of the Japes
6	The Good, the Bad, and the Facially Challenged
5	Soylent Yellow

# Subqueries with UPDATE

Now that you've confirmed that the statement works and double-checked the results, all that's left to do is to add the *WHERE* part of the query to the *UPDATE* statement you created earlier:

```
UPDATE Films SET DVDPrice = (SELECT MAX(DVDPrice) FROM Films)
WHERE (SELECT COUNT(*) FROM FavCategory WHERE FavCategory.CategoryId =
Films.CategoryId) >= 3
AND AvailableOnDVD = 'Y' AND Films.Rating > 3;
```

Execute the final SQL and you should find that three rows are updated with *DVDPrice* being set to 15.99. This SQL won't work on MySQL or MS Access, as they don't support using subqueries to update a column, so you'll need to change the query to the following if you want the results displayed to match those shown later in the book:

```
UPDATE Films SET DVDPrice = 15.99
WHERE (SELECT COUNT(*) FROM FavCategory WHERE FavCategory.CategoryId =
Films.CategoryId) >= 3
AND AvailableOnDVD = 'Y' AND Films.Rating > 3;
```

## Using Subqueries with the **DELETE FROM** Statement

The only place for a subquery to go in a **DELETE** statement is in the **WHERE** clause. Everything you've learned so far about subqueries in a **WHERE** clause applies to its use with a **DELETE** statement, so you can launch straight into an example. In this example, you want to delete all locations where one or fewer members live and where a meeting has never been held.

As before, create the **SELECT** queries first to double-check your results, and then use the **WHERE** clauses with a **DELETE** statement. First, you need a query that returns the number of members living in each city:

```
SELECT COUNT(*), City
FROM MemberDetails
GROUP BY City, State;
```

This query is used as a subquery to check whether one or fewer members live in a particular city. It provides the following results:

COUNT(*)	City
1	NULL
1	Dover
2	Windy Village
2	Big City
2	Townsville
1	New Town
4	Orange Town



Now you need to find a list of cities in which a meeting has never been held:

```
SELECT LocationId, City
FROM Location
WHERE LocationId NOT IN (SELECT LocationId FROM Attendance);
```

The subquery is used to find all LocationIds from the Location table that are not in the Attendance table. This subquery provides the following results:

LocationId	City
3	Big City

Now you need to combine the WHERE clauses and add them to a DELETE statement. Doing so combines the WHERE clause of the preceding query, which included the following condition:

```
LocationId NOT IN (SELECT LocationId FROM Attendance);
```

Likewise, it combines the SELECT statement of the first example as a subquery:

```
SELECT COUNT(*), City
FROM MemberDetails
GROUP BY City, State;
```

The condition and the subquery are merged into the WHERE clause of the DELETE statement:

```
DELETE FROM Location
WHERE (SELECT COUNT(*) FROM MemberDetails
      WHERE Location.City = MemberDetails.City
      AND
      Location.State = MemberDetails.State
      GROUP BY City, State) <= 1
AND
LocationId NOT IN (SELECT LocationId FROM Attendance);
```

In the first subquery, you count how many members live in a particular location from the Location table. If it's one or less, then that part of the WHERE clause evaluates to true. Note that the Location table of the DELETE statement and the MemberDetails tables of the subquery have been joined on the State and City fields:

```
Location.State = MemberDetails.State
```

This is absolutely vital, otherwise the subquery returns more than one result.

The second condition in the WHERE clause specifies that LocationId must not be in the Attendance table. Because an AND clause joins the two conditions in the WHERE clause, both conditions must be true.

If you execute the query, you find that no records match both conditions, and therefore no records are deleted from the Location table.