

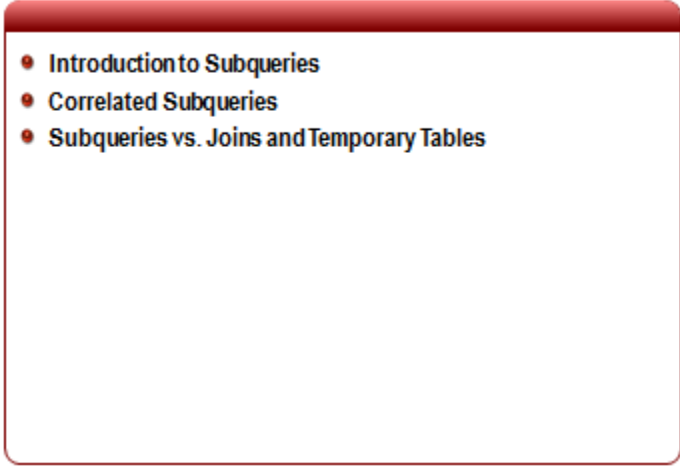
Module 5:

Working with Subqueries

Contents

Module Overview	1
Lesson 1: Introduction to Subqueries.....	2
What are Subqueries?	3
Subqueries as Expressions.....	6
ANY, ALL and SOME Operators	7
Scalar and Tabular Subqueries	9
Rules and Restrictions of Writing Subqueries	11
Practice: Simple Subquery.....	12
Lesson 2: Correlated Subqueries.....	14
What Are Correlated Subqueries?	15
Building a Correlated Subquery	16
Using Correlated Subqueries.....	17
Using the EXISTS Clause with Correlated Subqueries	18
Practice: Working with Correlated Subqueries	20
Lesson 3: Subqueries vs. Joins and Temporary Tables	22
Subqueries vs. Joins.....	23
Temporary Tables.....	26
Subqueries vs. Temporary Tables	27
Practice: Working with Subqueries vs. Joins	28
Summary	30

Module Overview

- 
- Introduction to Subqueries
 - Correlated Subqueries
 - Subqueries vs. Joins and Temporary Tables

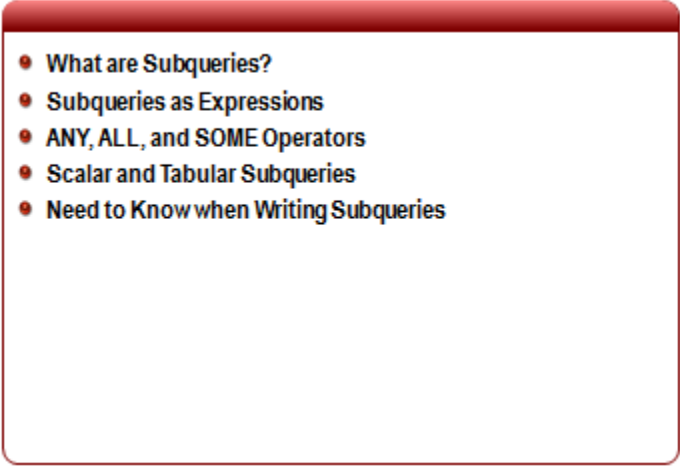
SQL Server and T-SQL provide a very powerful mechanism for creating nested queries (subqueries). A subquery is a SELECT-FROM-WHERE expression that is nested within another query. There are situations when trouble shooting a problem is very difficult to solve without using a subquery.

Objectives

After completing this module, you will be able to:

- Understand what is a subquery
- Write a correlated subquery
- Understand the difference between subquery, joins and temporary tables

Lesson 1: Introduction to Subqueries

- 
- What are Subqueries?
 - Subqueries as Expressions
 - ANY, ALL, and SOME Operators
 - Scalar and Tabular Subqueries
 - Need to Know when Writing Subqueries

A subquery is a query that is nested inside a SELECT, INSERT, UPDATE, or DELETE statement, or inside another subquery. A subquery can be used anywhere an expression is allowed.

A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.

Even a simple query can be re-written in subquery manner but it can also add unnecessary complications. In a case where the classic approach cannot fetch the desired result set, then this type of query becomes a candidate for a subquery.

Subqueries are not just reserved for SELECT statements. They also can be nested inside INSERT, UPDATE or DELETE statements or even within another subquery.

Objectives

After completing this lesson, you will be able to:

- Define what a subquery is
- Use subqueries as expressions
- Combine ANY, ALL, and SOME operators
- Write scalar and tabular subqueries

What are Subqueries?

- **SELECT statement nested within another SELECT statement**
- **We can recognize two types of subqueries:**
 - Scalar subqueries – returns a single value for the outer part
 - Multi valued subqueries – returns a list of values
- **In both cases we recognize outer and inner query parts**

```
SELECT LastName, FirstName --outer query
FROM Person.Person
WHERE Title IN
      (SELECT DISTINCT Title --inner query
       FROM Person.Person
        WHERE Title IS NOT NULL)
ORDER BY LastName
```

A common use for a subquery is within a SELECT statement that is nested within another SELECT statement. Subqueries can be placed anywhere inside a SELECT-FROM-WHERE part of a statement. The goal is to return a specific result set to the outer query. There are two types of subqueries:

- **Scalar subqueries** – returns a single value for the outer part
- **Multi valued subqueries** – returns a list of values

In both cases the outer query needs to know how to handle the input part from inner part.

From a performance perspective, the primary difference between classic queries and subqueries is that subqueries allow you to call for more detailed information from the database.

Example:

```
SELECT LastName, FirstName --outer query
FROM Person.Person
WHERE Title IN
      (SELECT DISTINCT Title --inner query
       FROM Person.Person
        WHERE Title IS NOT NULL)
ORDER BY LastName
```

Result:

LastName	FirstName
Abbas	Syed
Abel	Catherine
Abercrombie	Kim
Acevedo	Humberto
Achong	Gustavo
Ackerman	Pilar
Adams	Frances
Adams	Carla

Adams	Jay
Adams	Ben
Adina	Ronald
...	
Zubaty	Carla
Zubaty	Patricia
Zugelder	Judy
Zwilling	Michael

(1009 row(s) affected)

In the example below, the inner query returned a list of values (Titles of Persons) and passed the list on to the outer query. The outer query expects a list of values as a result of adding an IN clause inside the WHERE clause and is an example of a multi valued subquery.

The classic query equivalent for the example would be written as follows:

```
SELECT LastName, FirstName --outer query
FROM Person.Person
WHERE Title IN ('Sr.', 'Mrs.', 'Sra.', 'Ms.', 'Ms', 'Mr.')
ORDER BY LastName
```

WHERE Title IN ('Sr.', 'Mrs.', 'Sra.', 'Ms.', 'Ms', 'Mr.') –is a distinct list of values from Title column

The result is the same as the first version of the query.

Use the = operator to fetch a specific single value inner query as seen in the following example:

```
SELECT LastName, FirstName --outer query
FROM Person.Person
WHERE Title =
    (SELECT DISTINCT Title --inner query
     FROM Person.Person
     WHERE Title = 'Ms.')
ORDER BY LastName
```

Result:

....	
Zhang	Autumn
Zimmerman	Juanita
Zimmerman	Jo
Zimprich	Karin
Zubaty	Carla
Zubaty	Patricia
Zugelder	Judy

(415 row(s) affected)

The inner query returns single value where the persons title equals Ms. As a result, the outer query filters only persons whose title is equal to Ms.

Many subqueries can be written as JOINS. The next example shows a subquery and query version of the same T-SQL task. It returns a subcategory name where the category name is equal to 'Bikes'.

```
--Subquery version
SELECT Name
FROM Production.ProductSubcategory
WHERE ProductCategoryID IN
    (SELECT ProductCategoryID
     FROM Production.ProductCategory
     WHERE Name = 'Bikes')

--Join version
SELECT PS.Name
FROM Production.ProductSubcategory AS PS
     INNER JOIN Production.ProductCategory AS PC
     ON PC.ProductCategoryID = PS.ProductCategoryID
WHERE PC.Name = 'Bikes'
```

Result:

```
Name
-----
Mountain Bikes
Road Bikes
Touring Bikes

(3 row(s) affected)
```

Subqueries as Expressions

- Expressions are powerful features in TSQL
- Subqueries can be placed on many positions and forms in query, including expressions.

```
SELECT Name, Weight,
       (SELECT AVG (Weight) FROM Production.Product)
  AS 'Average',
  Weight - (SELECT AVG (Weight) FROM
Production.Product)
  AS 'Difference'
FROM Production.Product
WHERE Weight IS NOT NULL AND Weight > 800
ORDER BY Weight DESC
```

Expressions are powerful features in T-SQL that allow you to write subqueries to act as an expression? Subqueries and expressions can be placed in many positions and forms of a query.

The query below is ordered to find products where Weight is known and calculates average weight and the difference between the two. The calculations are done using subqueries placed within a SELECT list as an expression.

```
SELECT Name, Weight,
       (SELECT AVG (Weight) FROM Production.Product)
  AS 'Average',
  Weight - (SELECT AVG (Weight) FROM Production.Product)
  AS 'Difference'
FROM Production.Product
WHERE Weight IS NOT NULL
ORDER BY Weight DESC
```

Result:

Name	Weight	Average	Difference
LL Road Rear Wheel	1050.00	74.069219	975.930781
ML Road Rear Wheel	1000.00	74.069219	925.930781
LL Road Front Wheel	900.00	74.069219	825.930781
HL Road Rear Wheel	890.00	74.069219	815.930781
ML Road Front Wheel	850.00	74.069219	775.930781

Warning: Null value is eliminated by an aggregate or other SET operation.

(5 row(s) affected)

ANY, ALL and SOME Operators

- **ANY** returns TRUE when the comparison specified is TRUE for any pair
- **ALL** returns TRUE when the comparison specified is TRUE for all pairs.

```
SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice >= ALL
      (SELECT MAX (ListPrice)
       FROM Production.Product
       GROUP BY ProductSubcategoryID)
```

Query with ALL returns 5

The ALL operator compares a scalar value with a single-column set of values. Operators SOME and ANY are equivalent. SOME is an ISO standard for SQL Server T-SQL version ANY. ALL also compares a scalar value with a single-column set of values.

The difference between the ANY and ALL operators are:

- ANY returns TRUE when the comparison specified is TRUE for any pair.
- ALL returns TRUE when the comparison specified is TRUE for all pairs.

ALL requires the scalar_expression to compare positively to every value that is returned by the subquery. For instance, if the subquery returns values of 2 and 3, scalar_expression <= ALL (subquery) would evaluate as TRUE for a scalar_expression of 2. If the subquery returns values of 2 and 3, scalar_expression = ALL (subquery) would evaluate as FALSE, because some of the values of the subquery (the value of 3) would not meet the criteria of the expression. The following example uses an ANY operator:

```
SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice >= ANY
      (SELECT MAX (ListPrice)
       FROM Production.Product
       GROUP BY ProductSubcategoryID)
```

Result:

Name	ListPrice
LL Mountain Seat Assembly	133,34
ML Mountain Seat Assembly	147,14
HL Mountain Seat Assembly	196,92
LL Road Seat Assembly	133,34
...	
Road-750 Black, 44	539,99
Road-750 Black, 48	539,99
Road-750 Black, 52	539,99

(299 row(s) affected)

The same query using the ALL operator.

```
SELECT Name, ListPrice
FROM Production.Product
WHERE ListPrice >= ALL
      (SELECT MAX (ListPrice)
       FROM Production.Product
       GROUP BY ProductSubcategoryID)
```

Result:

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27

(5 row(s) affected)

ANY requires the scalar_expression to compare positively to at least one value returned by the subquery. For instance, if the subquery returns values of 2 and 3, scalar_expression = SOME (subquery) would evaluate as TRUE for a scalar_express of 2. If the subquery returns values of 2 and 3, scalar_expression = ALL (subquery) would evaluate as FALSE, because some of the values of the subquery (the value of 3) would not meet the criteria of the expression.

Scalar and Tabular Subqueries

- Subqueries can return two types of result sets, scalar and tabular.
- Depend on your needs you can choose one of them
 - Scalar subquery returns a single row of data to the outer query
 - Tabular subquery returns a tabular data to the outer query

Depending on your needs, subqueries can return two types of result sets; scalar and tabular.

- Scalar subqueries return a single row of data to the outer query
- Tabular subqueries return tabular data to the outer query

Example of scalar subquery, where result is a single row based on condition that is defined in inner query.

```
SELECT LastName
FROM Person.Person
WHERE LastName = (SELECT LastName
                  FROM Person.Person
                  WHERE LastName = 'Abbas')
ORDER BY LastName
```

Result:

```
LastName
-----
Abbas
```

(1 row(s) affected)

In the example tabular query below, the query shows all persons Last Name where their Title is equal to title of person named Abbas.

```
SELECT LastName
FROM Person.Person
WHERE Title = (SELECT Title
               FROM Person.Person
               WHERE LastName = 'Abbas')
ORDER BY LastName
```

Result:

LastName

Abbas

Achong

Adams

Adams

Adina

...

Zhang

Ziegler

Zwilling

(577 row(s) affected)

Rules and Restrictions of Writing Subqueries

The ntext, text and image data types are not allowed in the select list of subqueries.

ORDER BY can only be specified when TOP is also specified.

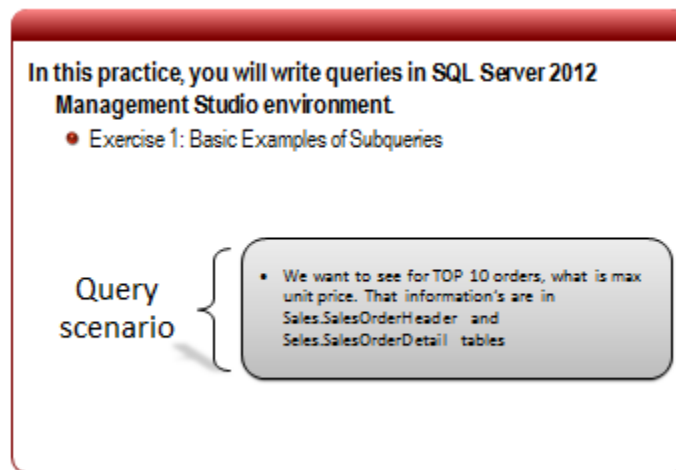
SELECT query of a subquery is always enclosed in parentheses

Subquery can be nested inside the WHERE or HAVING clause of an outer SELECT, INSERT, UPDATE, or DELETE statement

Generally, writing subqueries does not require special T-SQL skills, but you do need to understand and follow some rules and restrictions:

- When the WHERE clause of an outer query includes a column name, it must be join-compatible with the column in the subquery select list.
- ntext, text and image data types are not allowed in the select list of subqueries.
- ORDER BY can only be specified when TOP is also specified.
- SELECT query of a subquery is always enclosed in parentheses.
- Subqueries can be nested inside WHERE or HAVING clauses of an outer SELECT, INSERT, UPDATE or DELETE statement.
- A subquery can have a subquery inside.
- Up to 32 levels of nesting are possible.
- If a table appears only in a subquery and not in the outer query, the columns from that table cannot be included in the output.

Practice: Simple Subquery



In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 1.

To successfully complete this exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise: Writing a Simple Subquery

1. Click **Start**→**Microsoft SQL Server 2012**→**SQL Server Management Studio**
2. In the **Connect to Server** dialog windows, under **Server** name type the name of your local instance.
 - a. If it is default then just type (local)
3. Use Windows Authentication
4. Open **File** menu →**New** →**Database Engine Query**

Query Scenario

We want to see the TOP 10 orders and what the maximum unit price is. The information is in the Sales.SalesOrderHeader and Sales.SalesOrderDetail tables.

Write following T-SQL query:

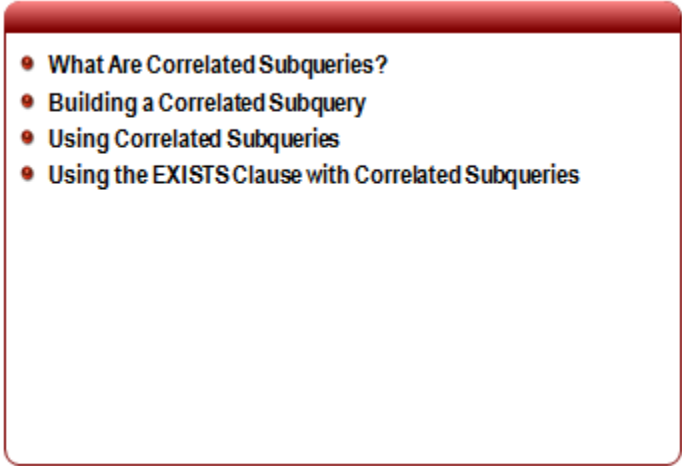
```
SELECT TOP 10 SOH.SalesOrderID, SOH.OrderDate,
      (SELECT MAX(SOD.UnitPrice)
       FROM Sales.SalesOrderDetail AS SOD
       WHERE SOH.SalesOrderID = SOD.SalesOrderID) AS MaxUnitPrice
FROM Sales.SalesOrderHeader AS SOH
ORDER BY SalesOrderID
```

Result:

SalesOrderID	OrderDate	MaxUnitPrice
43659	2005-07-01 00:00:00.000	2039,994
43660	2005-07-01 00:00:00.000	874,794
43661	2005-07-01 00:00:00.000	2039,994
43662	2005-07-01 00:00:00.000	2146,962
43663	2005-07-01 00:00:00.000	419,4589
43664	2005-07-01 00:00:00.000	2039,994
43665	2005-07-01 00:00:00.000	2039,994
43666	2005-07-01 00:00:00.000	2146,962
43667	2005-07-01 00:00:00.000	2039,994
43668	2005-07-01 00:00:00.000	2146,962

(10 row(s) affected)

Lesson 2: Correlated Subqueries

- 
- What Are Correlated Subqueries?
 - Building a Correlated Subquery
 - Using Correlated Subqueries
 - Using the EXISTS Clause with Correlated Subqueries

In many cases subqueries are stand alone and independent from the outer part of the query. You can select only the inner part and press F5 to execute and view the result set.

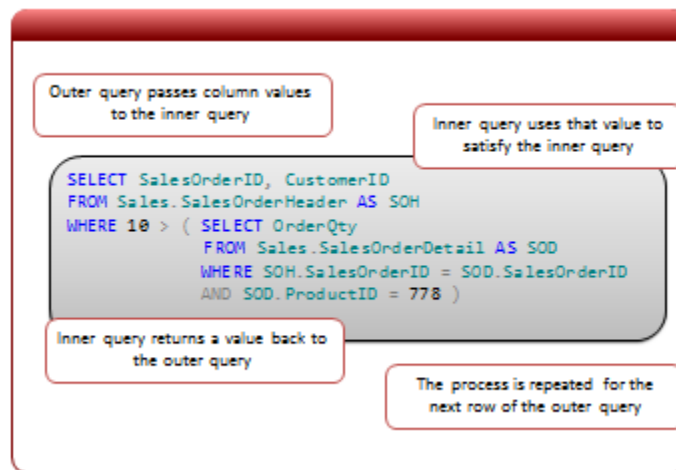
There are situations where the query “keeps” a connection with the WHERE clause while the outer query passes an input value to the inner part that is required for evaluation. This is called Correlated or Repeating Subqueries.

Objectives

After completing this lesson, you will be able to:

- Build a correlated subquery
- Use correlated subqueries
- Use the EXISTS clause with correlated subqueries

What Are Correlated Subqueries?



A Correlated subquery is a regular subquery that has a mutual relationship or connection in which one affects or depends on another. Correlated subqueries are executed multiple times (e.g. like FOR loop in programming), each time for a different row in the outer query.

Since a subquery depends on the value(s) from the outer part, you need to pay attention when writing your T-SQL code and consider the connection link between them.

Let's look at the following example:

The outer query passes column values to the inner query. The inner query then uses that value for evaluation of the condition. The inner query returns a value to the outer query. This process is repeated for the all rows in the outer query.

```

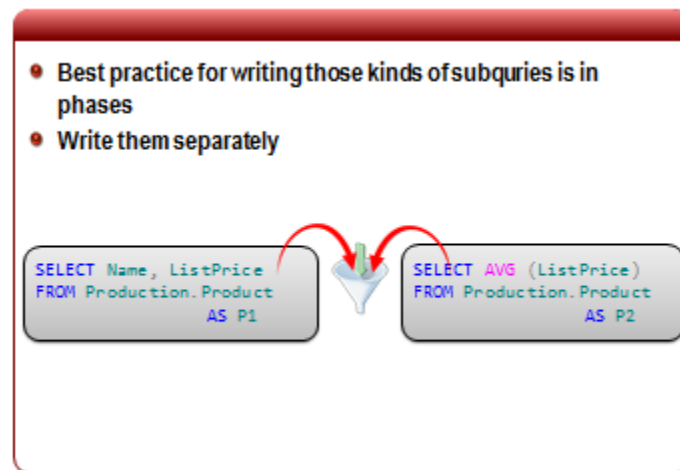
SELECT SalesOrderID, CustomerID
FROM Sales.SalesOrderHeader AS SOH
WHERE 10 > (SELECT OrderQty
            FROM Sales.SalesOrderDetail AS SOD
            WHERE SOH.SalesOrderID = SOD.SalesOrderID
            AND SOD.ProductID = 778)
  
```

Result:

SalesOrderID	CustomerID
45991	11976
46227	11990
46159	11996
46588	20035
46290	26066
...	
44294	30067
45052	30067
45792	30067

(243 row(s) affected)

Building a Correlated Subquery



The dependencies of an inner query on a value from the outer part can lead to result set errors when building correlated subqueries.

Building a correlated subquery can be a very complicated process of building, testing and debugging. A best practice to minimize errors and make trouble shooting easier is to write and test the query in phases.

In the query scenario below you want to see the correlation between product list prices and average list price of all products. You also want to show only those products where the average list price is less than 3500. The issue here is that the information for both are in the same table. Instead of writing a cross join query, you will do it using a phased approach correlated subquery.

First step, write the two separate queries below.

```
SELECT Name, ListPrice
FROM Production.Product AS P1

SELECT AVG (ListPrice)
FROM Production.Product AS P2
```

When you know that the query will deliver the expected result, you can then move to next phase of the subquery which is merging the two queries into one correlated subquery. Test the query and continue through the phases to write a query that will perform as expected.

Using Correlated Subqueries

- At this point we have to separate queries.
- Now let's connect it together

```
SELECT Name, ListPrice
FROM Production.Product AS P1
WHERE 3500 < (SELECT AVG (ListPrice)
              FROM Production.Product AS P2
              WHERE P1.ProductID = P2.ProductID)
```

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27

(5 row(s) affected)

A major limitation in building a correlated subquery process is that correlated subqueries cannot be executed separately from the outer part of T-SQL code. Before we can use the two queries from the previous topic, we need to identify the column from the outer query that will be passed into the correlated subquery. You also need to declare aliases for the tables and use it.

The two separate queries are connected together in the query below.

```
SELECT Name, ListPrice
FROM Production.Product AS P1
WHERE 3500 < (SELECT AVG (ListPrice)
              FROM Production.Product AS P2
              WHERE P1.ProductID = P2.ProductID)
```

Result:

Name	ListPrice
Road-150 Red, 62	3578,27
Road-150 Red, 44	3578,27
Road-150 Red, 48	3578,27
Road-150 Red, 52	3578,27
Road-150 Red, 56	3578,27

(5 row(s) affected)

In this case, the common column is ProductID and we are using it with aliases to connect the inner part with the outer part of query. Since the query and subquery are linked, you cannot run the subquery separately.

Using the EXISTS Clause with Correlated Subqueries

- In case of using EXISTS, SQL Server handles the results on the different way.
- There is no retrieving any kind of data, it just check is there any rows in results and returns TRUE or FALSE to the outer part.

```

SELECT Name
FROM Person.StateProvince AS PS
WHERE EXISTS (SELECT COUNT (*)
              FROM Sales.SalesTerritory AS SST
              WHERE PS.TerritoryID = SST.TerritoryID
              AND SST.Name = 'France')

```

Name
Ain
Aisne
Alpes-de-Haute Provence
...
Hauts de Seine
Seine Saint Denis
Val de Marne
Val d'Oise

(96 row(s) affected)

The EXISTS tests whether the inner query returns any rows. If it does, then the outer query proceeds. If not, the outer query does not execute, and the entire SQL statement returns nothing. It is not producing any result set. Instead it returns a Boolean TRUE or FALSE. This can be good practice for checking data for existence without retrieving actual data.

When using EXISTS, SQL Server handles the results in a different way. There is no retrieving any kind of data, it just checks if there are any rows in the result and returns TRUE or FALSE to the outer part.

In the following examples we will see the difference between solving a problem with and without using the EXISTS clause.

In this example we want to check if there are any state provinces from France in the Person.StateProvince table without using the EXISTS clause.

```

SELECT Name
FROM Person.StateProvince AS PS
WHERE (SELECT COUNT (*)
       FROM Sales.SalesTerritory AS SST
       WHERE PS.TerritoryID = SST.TerritoryID
       AND SST.Name = 'France') > 0

```

Result:

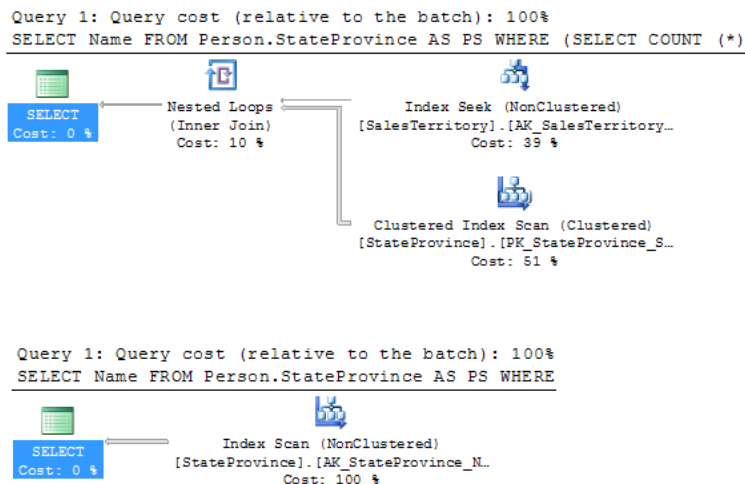
Name
Ain
Aisne
Alpes-de-Haute Provence
...
Hauts de Seine
Seine Saint Denis
Val de Marne
Val d'Oise

(96 row(s) affected)

Now rewrite the example using the EXISTS clause.

```
SELECT Name
FROM Person.StateProvince AS PS
WHERE EXISTS (SELECT COUNT (*)
              FROM Sales.SalesTerritory AS SST
              WHERE PS.TerritoryID = SST.TerritoryID
              AND SST.Name = 'France')
```

The result of both queries is the same. The second example using the EXISTS clause checks the inner query for any record based on a condition and returns the result to the outer query. There is also a difference in physical query processing. The actual execution plan is very different as shown in the image below:

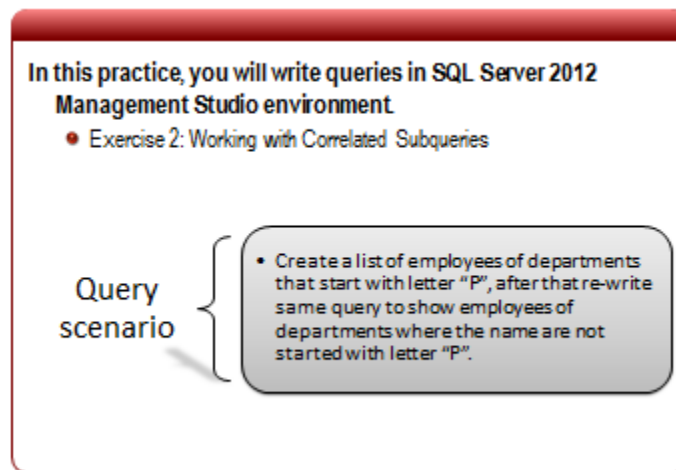


The first query example has three operations to complete, while the second has only one. This is relative and depends on many parameters. But this is good starting point for thinking about performance issues (indexes, avoiding certain T-SQL commands in some situations, database normalization, etc.).

When writing these kinds of subqueries, there is no need to write a column list in the inner query as it only returns true or false, so columns are irrelevant and using (*) is sufficient.

The NOT EXIST clause performs the exact opposite of EXISTS. If you rewrite the query example you will get only state provinces that are **not** from France.

Practice: Working with Correlated Subqueries



In this practice, you will write queries in SQL Server 2012 Management Studio environment.

- Exercise 2: Working with Correlated Subqueries

Query scenario

- Create a list of employees of departments that start with letter "P", after that re-write same query to show employees of departments where the name are not started with letter "P".

In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 2.

To successfully complete this exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise: Writing a Query with Correlated Subqueries

1. Click **Start**→**Microsoft SQL Server 2012**→**SQL Server Management Studio**
2. On the **Connect to Server** dialog windows, under **Server name** type the name of your local instance.
 - a) If it is default then just type (local)
3. Use Windows Authentication
4. Open **File** menu →**New** →**Database Engine Query**

Query Scenario

Write a query that will create a list of employees and their departments, where the department name starts with the letter "P".

```

SELECT P.FirstName, P.LastName
FROM Person.Person AS P
JOIN HumanResources.Employee AS E
    ON E.BusinessEntityID = P.BusinessEntityID
WHERE EXISTS (SELECT *
    FROM HumanResources.Department AS D
    JOIN HumanResources.EmployeeDepartmentHistory AS EDH
    ON D.DepartmentID = EDH.DepartmentID
    WHERE E.BusinessEntityID = EDH.BusinessEntityID
    AND D.Name LIKE 'P%')
ORDER BY P.LastName

```

Result:

FirstName	LastName
.....
....	
Nuan	Yu
Gary	Yukish
Eugene	Zabokritski
Kimberly	Zimmerman
Michael	Zwilling

(198 row(s) affected)

Re-write same query to list employees where the department name does not start with the letter “P”.

```

SELECT P.FirstName, P.LastName
FROM Person.Person AS P
JOIN HumanResources.Employee AS E
    ON E.BusinessEntityID = P.BusinessEntityID
WHERE NOT EXISTS (SELECT *
    FROM HumanResources.Department AS D
    JOIN HumanResources.EmployeeDepartmentHistory AS EDH
    ON D.DepartmentID = EDH.DepartmentID
    WHERE E.BusinessEntityID = EDH.BusinessEntityID
    AND D.Name LIKE 'P%')
ORDER BY P.LastName

```

Result:

FirstName	LastName
.....
....	
Jill	Williams
Dan	Wilson
John	Wood
Peng	Wu

(92 row(s) affected)

Lesson 3: Subqueries vs. Joins and Temporary Tables

- 
- Subqueries vs. Joins
 - Temporary Tables
 - Subqueries vs. Temporary Tables

Database developers and programmers most often use joins and temporary tables instead of subqueries. Subqueries can produce at least similar and sometimes even better results.

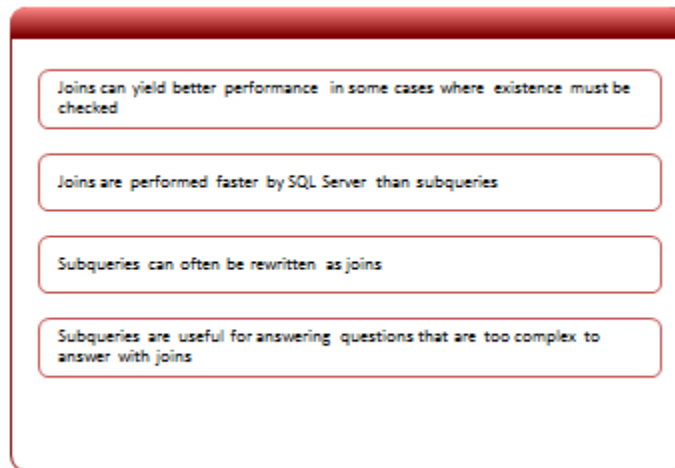
After completing this lesson you will be able to recognize the major differences between JOINS, tables and subqueries and when to use them.

Objectives

After completing this lesson, you will be able to:

- Understand the difference between subqueries and joins
- Understand temporary tables
- Recognize usage of subqueries vs. temporary tables

Subqueries vs. Joins



The first question among new people in the database world is “what is faster, subqueries or joins?” Based on the answer they form an opinion which is, in most situations wrong. In SQL Server, there is usually no significant difference in performance perspective between subqueries and join equivalents. But, yes join can produce better performance.

Subqueries can perform some operations where joins are limited or harder to implement. E.g. operators IN, EXISTS, ANY and ALL are very powerful as seen in the previous lesson.

Subqueries are used when writing joins is too complex. When multiple joins make your T-SQL code difficult to follow, understand and read, then you have a candidate for subquery rewriting.

The following are some useful guidelines for using joins and queries.

- If your report needs data from more than one table, you must perform a join. Whenever multiple tables (or views) are listed in the FROM clause, those tables become joined.
- If you need to combine related information from different rows within a table, then you can join the table with itself.
- Use subqueries when the result that you want requires more than one query and each subquery provides a subset of the table involved in the query.
- Many queries can be formulated as joins or subqueries, a join is generally more efficient to process.

For better understanding, we will use almost the same example from the last exercise. Create a list of employees from Research and Development department using joins.

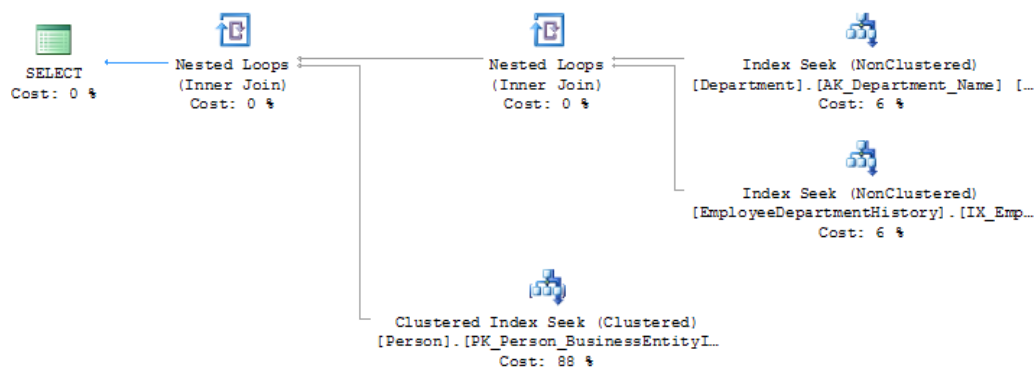
```
SELECT P.FirstName, P.LastName
FROM Person.Person AS P JOIN HumanResources.Employee AS E
    ON E.BusinessEntityID = P.BusinessEntityID
JOIN HumanResources.EmployeeDepartmentHistory AS EDH
    ON E.BusinessEntityID = EDH.BusinessEntityID
JOIN HumanResources.Department AS HD
    ON HD.DepartmentID = EDH.DepartmentID
WHERE HD.Name = 'Research and Development'
```

Result:

FirstName	LastName
Dylan	Miller
Diane	Margheim
Gigi	Matthew
Michael	Raheem

(4 row(s) affected)

Actual execution plan for join version is:



The same task in subquery version:

```

SELECT P.FirstName, P.LastName
FROM Person.Person AS P
JOIN HumanResources.Employee AS E
ON E.BusinessEntityID = P.BusinessEntityID
WHERE EXISTS (SELECT *
               FROM HumanResources.Department AS D
               JOIN HumanResources.EmployeeDepartmentHistory AS EDH
               ON D.DepartmentID = EDH.DepartmentID
               WHERE E.BusinessEntityID = EDH.BusinessEntityID
               AND D.Name = 'Research and Development')
ORDER BY P.LastName

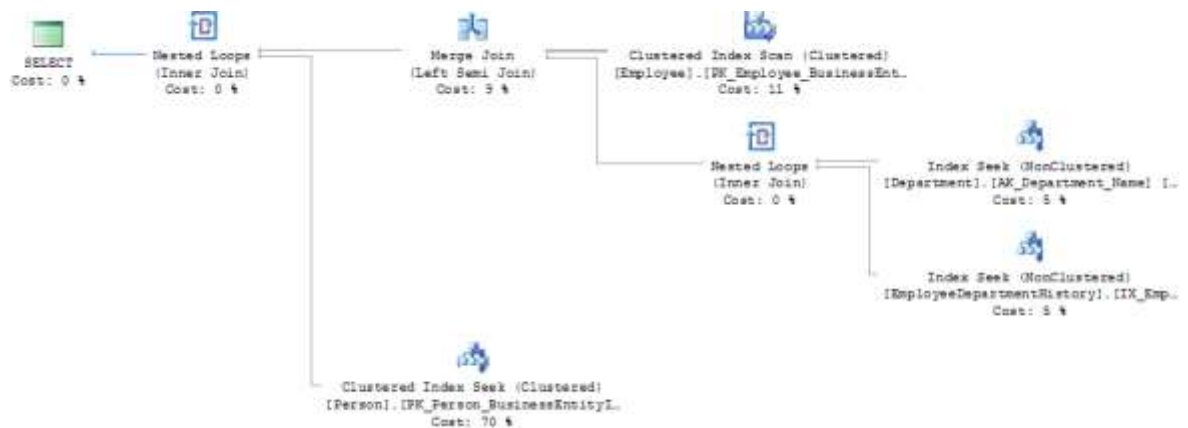
```

Result:

FirstName	LastName
Dylan	Miller
Diane	Margheim
Gigi	Matthew
Michael	Raheem

(4 row(s) affected)

Actual execution plan for subquery version is:



We can see that the query requires six operations to complete while the subquery requires eight. More operators = more time to execute = slower performance.

Temporary Tables

- **Local temporary tables**
 - This object is visible and accessible to their creators.
 - Visibility period is during same connection in which object is created. After user disconnects session, objects are deleted from tempdb.
 - Name starts with #
- **Global temporary tables.**
 - They are visible and accessible to the all user and all connection after object is created.
 - After all users disconnect sessions where they used and reference temp table, object is deleted from tempdb.
 - Name starts with ##

Regular (permanent) tables and temporary tables creation, structure and data types are the same. The single difference is, temporary tables are stored in a *tempdb* system database. Temporary tables are deleted (structure and data) when they are no longer used. We can categorize temporary tables in two major groups:

- **Global temporary tables** - are visible and accessible to all users and all connections after the object is created. The object is deleted from tempdb only after all users disconnect their sessions where they used the reference temp table.
- **Local temporary tables** – this object is visible and accessible to its creators only during the connection duration. The objects are deleted from tempdb after the user disconnects the session.

When you create a local temp table, you must use one pound sign (#) before the name and two pound signs (##) before the name for global temp tables.

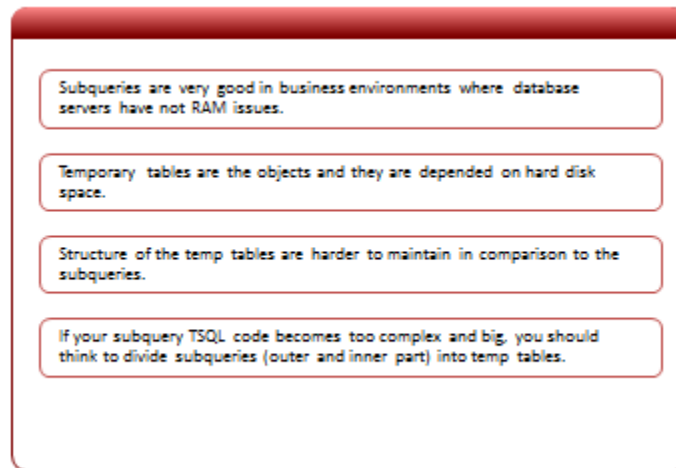
CREATE TABLE #localTemp

(SomeID Int)

CREATE TABLE ##globalTemp

(SomeID Int)

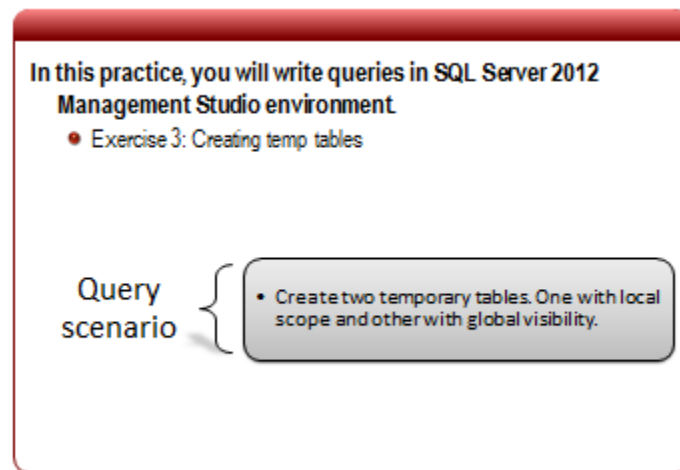
Subqueries vs. Temporary Tables



Tips to help you decide whether to use Subqueries or Temporary Tables:

- Some subqueries are very RAM dependent and therefore best used in business environments where database servers have no RAM issues.
- Temporary tables are objects that require hard disk space to execute.
- The structure of temporary tables is more difficult to handle in comparison to subqueries. If your subquery T-SQL code becomes too large and complex, you should think about dividing the subqueries (outer and inner part) into temp tables.

Practice: Working with Subqueries vs. Joins



In this practice you will write queries in SQL Server 2012 Management Studio environment. Practice is based on Lesson 3.

To successfully complete this exercise you need following resources:

- SQL Server Management Studio 2012
- AdventureWorks2012 sample database
 - <http://msftdbprodsamples.codeplex.com/>

Exercise: Writing a Query with Correlated Subqueries

1. Click Start→Microsoft SQL Server 2012→SQL Server Management Studio
2. On the Connect to Server dialog windows, under Server name type the name of your local instance.
 - i. If it is default then just type (local)
3. Use Windows Authentication
4. Open File menu →New →Database Engine Query

Query Scenario

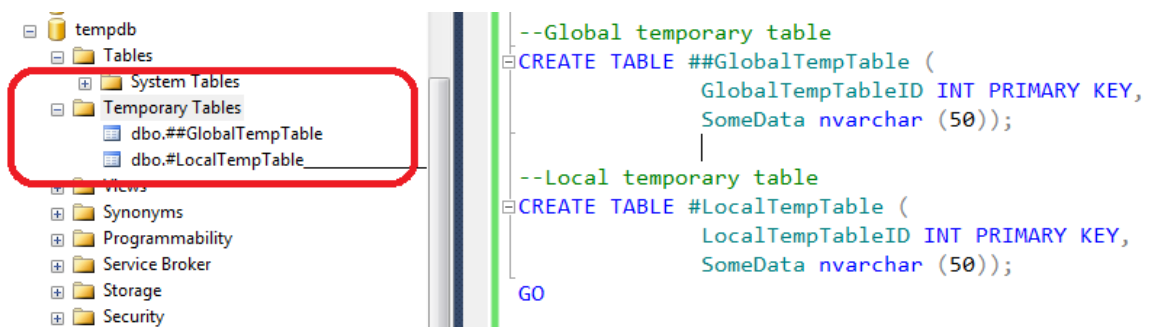
Create two temporary tables. One with local scope and other with global visibility

Write following T-SQL query:

```
--Local temporary table
CREATE TABLE #LocalTempTable (
    LocalTempTableID INT PRIMARY KEY,
    SomeData nvarchar (50));

GO

--Global temporary table
CREATE TABLE ##GlobalTempTable (
    GlobalTempTableID INT PRIMARY KEY,
    SomeData nvarchar (50));
```

Result:

Summary

In this module, you learned how to:

- How to write a subquery
- How to write a correlated subquery
- How to recognize places in TSQL code where is a need for subquery usage
- Understand difference between subquery, joins and temporary tables

In this module you have learned that SQL Server and T-SQL provide very powerful and flexible options for creating subqueries. You have learned that they can be used in place of expressions and that they are not limited to SELECT statements. Subqueries can be nested inside INSERT, UPDATE or DELETE statements as well as within another subquery.

You have learned which situations where your query needs to “keep” a connection with the WHERE clauses of the outer query and based on input values, perform some evaluation tasks. Correlated queries are powerful T-SQL tools for tasks where JOINS are too complex.

Finally, you have learned what the major difference is between joins and subqueries and when to choose temporary tables and when replace them with subqueries.

Objectives

After completing this module, you learned:

- How to write a subquery
- What is a correlated query and how to write a correlated query
- How to recognize places in T-SQL code where a subquery should be used
- Understand the difference between subquery, joins and temporary tables