2.7 Subqueries

2.7.1 Simple Subqueries in the WHERE clause

Decision makers often wish to identify special records in tables:

- What is the product with the highest price?
- What is the product with the lowest price?
- What product has the most unitsInStock?
- Who is the lightest female in the person table?

Using aggregates, we can get part of the way:

```
SELECT MAX(unitPrice)
FROM Products
```

Suppose the above SQL returns $100 as the maximum unitPrice in the Products table. We know the maximum unitPrice value, but don’t know which product it is, or any other details about it. However, knowing that the product has a unitPrice=100 allows us to do this:

```
SELECT ProductName, unitPrice
FROM Products
WHERE unitPrice = 100
```

Unfortunately, this method takes two separate operations. Also, there’s no guarantee that between executing the first and second SELECT statements, that a new, higher-priced product hasn’t been inserted in the table. We can make sure that this doesn’t happen by executing both SELECT statements at the same time:

```
SELECT ProductName, unitPrice
FROM Products
WHERE unitPrice =
    (SELECT MAX(unitPrice)
     FROM Products)
```

The SELECT statement that returns the maximum price has been embedded as a subquery within the SELECT statement that gives details about products. The result is a single SELECT statement that returns the details of the product that has the highest unitPrice.

This same technique can be used to show above average priced items:

```
SELECT ProductName, unitPrice
FROM Products
WHERE unitPrice >
    (SELECT AVG(unitPrice)
     FROM Products)
```

We can also accomplish something similar to a join using the IN operator:

```
SELECT CompanyName
```
FROM Suppliers
WHERE supplierID IN
    (SELECT supplierID
     FROM Products)

Here the subquery provides a list of supplierIDs in the Products table. Then records in the Suppliers table are limited to only those that appear in the result of the subquery. The result is “Suppliers who supply us a product.” The same result could be accomplished with a join:

```
SELECT CompanyName
FROM Suppliers JOIN Products
    ON Suppliers.SupplierID = Products.SupplierID
```

### 2.7.2 Subqueries in the SELECT or FROM clause

It’s possible to put subqueries in the SELECT or FROM clause of a SELECT statement. In most cases, there is clearer way to phrase the SELECT statement, but we’ll cover some possibilities here for completeness.

Here is a simple statement to demonstrate a subquery in the SELECT clause:

```
SELECT 17     AS [Number],
       (SELECT 'fred') AS [Name]
```

Which gives:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>fred</td>
</tr>
</tbody>
</table>

This could be more clearly phrased, without the subquery, this way:

```
SELECT 17     AS [Number],
       'fred' AS [Name]
```

Perhaps a more realistic example:

```
SELECT ProductID,
       ProductName,
       (SELECT COUNT(SupplierID)
        FROM   Suppliers) AS [SupplierCount]
FROM   Products
```

Which gives:

<table>
<thead>
<tr>
<th>ProductID</th>
<th>ProductName</th>
<th>SupplierCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Alice Mutton</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>Aniseed Syrup</td>
<td>31</td>
</tr>
<tr>
<td>40</td>
<td>Boston Crab Meat</td>
<td>31</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Notice that the subquery is in no way related to the main (outer) query. You can see this by noting that the SupplierCount does not change from product record to product record.

Here is another simple, if odd, example that shows how a subquery can be placed in the FROM clause:

```
SELECT [Number]
FROM (SELECT 17 AS [Number]) [TableAlias]
```

Which gives:

<table>
<thead>
<tr>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
</tr>
</tbody>
</table>

This takes a slight bit of trickery to execute successfully. The [TableAlias] portion is merely there to give a “table name” to the subquery, within the context of the main query. The “SELECT [Number]” in the main query is specifying the [Number] column of a table named “[TableAlias]”, which consists of the results of the subquery.

Here’s another example of a subquery in the FROM clause:

```
SELECT   SupplierID,
          ProductCount
FROM     (SELECT SupplierID       AS [SupplierID],
               COUNT(ProductID) AS [ProductCount]
               FROM   Products
               GROUP BY SupplierID) [SupplierProductCounts]
WHERE    ProductCount > 2
```

In this case, the subquery creates a table called [SupplierProductCounts], from which the main query selects only some of the records. You might look at this and realize that there is a special SQL word created to do exactly this kind of operation, without using a subquery: HAVING. Here is how to write a logically equivalent SELECT statement that is more direct:

```
SELECT   SupplierID,
          COUNT(ProductID) AS [ProductCount]
FROM   Products
GROUP BY SupplierID
HAVING   COUNT(ProductID) > 2
```

So, the HAVING clause is not strictly necessary in Structured Query Language, because it can be accomplished with a subquery and a WHERE clause. However, the HAVING clause is much clearer and more succinct.

In summary, subqueries can be used in both the SELECT clause and the FROM clause of a SELECT statement:

- In the SELECT clause, to generate a value in each row
- In the FROM clause to generate a “temporary” table from which to extract rows
Using subqueries in the SELECT and FROM clause is less common and generally less useful than using a subquery in the WHERE clause. With the primary goal of creating clear, understandable SQL statements, these type of subqueries should be avoided where a more standard phrasing is available. However, sometimes it is easier to work through the logic, or more clearly document your intentions by using a subquery this way. In these cases, it makes sense to use a subquery.

### 2.7.3 Correlated Subqueries

In the subqueries from the previous section, you can think of the subqueries being executed first, then the results provided to the outer queries. In this regard, they are relatively simple subqueries, and they are separable – the subquery is a full-fledged, syntactically correct SQL statement in its own right. It could be executed independently of the outer SQL that contains it.

However, a subquery can placed in almost any place that an expression is used. Furthermore, the subquery can be made to relate to the parent query by creating a correlated subquery. A correlated subquery is different because it needs information from the parent query to execute. In other words, it cannot be executed independently of the outer SQL that contains it.

Consider this simple SELECT statement:

```
SELECT CategoryName
FROM Categories
```

Suppose we also wish to know the highest price that appears in a Products record in each category. That can be written using GROUP BY and a JOIN:

```
SELECT Categories.CategoryName,  
    MAX(Products.unitPrice) AS [Highest Price]
FROM Categories JOIN Products  
ON Categories.CategoryID=Products.CategoryID
GROUP BY Categories.CategoryName
```

We can write an equivalent SELECT statement using a correlated subquery:

```
SELECT CategoryName,  
    (SELECT MAX(unitPrice) 
    FROM Products  
    WHERE Categories.CategoryID=Products.CategoryID)  
    AS [Highest Price]
FROM Categories
```

Which has this subquery:

```
(SELECT MAX(unitPrice)  
FROM Products  
WHERE Categories.CategoryID=Products.CategoryID)
```

Unlike simple subqueries, correlated subqueries cannot execute on their own. In this example, the SQL interpreter will indicate that the Categories table should be in the FROM statement if we want to use
Categories.CategoryID in the WHERE clause. So where does Categories.CategoryID come from? It comes from the outer query, as indicated here:

```sql
SELECT CategoryName,
       (SELECT MAX(unitPrice)
        FROM Products
        WHERE Categories.CategoryID=Products.CategoryID)
       AS [Highest Price]
FROM Categories
```

Another difference between simple subqueries and correlated subqueries is the way the results are produced. In a simple subquery, the subquery is executed first and the results used in the parent query. In other words, the subquery is executed first.

In correlated subqueries, the subquery must know which parent query record is being used to be able to execute. In other words, the parent query (sometimes called the outer query) is executed concurrently with the subquery. As each record in the parent query is produced, the necessary value for the subquery is passed to the subquery. The subquery executes based on the value provided, and returns the result.

In the example above, think of the database engine executing the correlated subquery like this:

1. Get the first record in Categories
2. Get the categoryID from this record
3. Pass the categoryID to the subquery
4. Execute the subquery with this categoryID
5. Return the MAX(unitPrice) of products with this categoryID to the outer query
6. Get the second record in Categories
7. Etc.

As mentioned above, there are some correlated subqueries which can be stated in another way, such as a GROUP BY with a JOIN. However, what if there are two tables involved? Consider this data model:
Suppose that we wish to show each person, their highest skill level, and the number of clubs they are involved with. Here is the SELECT statement, with two correlated subqueries in the SELECT clause:

```sql
SELECT Person.firstname,
       Person.lastname,
       (SELECT Max(skilllevel)
        FROM   PersonSkill
        WHERE  PersonSkill.personid = Person.personid)
       AS [Highest Skill Level],
       (SELECT COUNT(clubid)
        FROM   Clubmembership
        WHERE  Clubmembership.personid = Person.personid)
       AS [Club Count]
FROM   Person
```

It may seem like this can be rephrased with a GROUP BY and several JOINs, like this:

```sql
SELECT Person.firstName,
       Person.lastName,
       MAX(PersonSkill.skillLevel) AS [Highest Skill Level],
       COUNT(ClubMembership.clubID) AS [Club Count]
FROM   Person, PersonSkill, ClubMembership
WHERE  Person.PersonID = PersonSkill.PersonID
       AND Person.PersonID = ClubMembership.PersonID
GROUP BY Person.firstName, Person.LastName
```

However, there are several important differences. First, the GROUP BY version will treat two people with the same first and last name as the same person. (This can be prevented by including Person.personID in the GROUP BY clause.) Second, in the case where a Person is not in a club, the entire query will return zero records. Or in the case where the person does not have an entry in PersonSkill,
the entire query will return zero records. In other words, the GROUP BY version will only work if every person has a PersonSkill record and a ClubMembership record associated with them.

In the correlated subquery version, all people will always be shown, even though the [Highest Skill Level] may be NULL, or the [Club Count] may be zero. There are ways to get around this issue using outer joins (covered later), but the SQL becomes more complex. In short, there are certain SQL statements that are most clearly expressed by a correlated subquery.

Another common use of correlated subqueries is in combination with aggregation to find special records within groups. For example, consider this simple subquery (not correlated):

```
SELECT Suppliers.CompanyName, Products.ProductName, Products.unitPrice
FROM Products JOIN Suppliers
ON Products.SupplierID = Suppliers.SupplierID
WHERE unitPrice = (SELECT MAX(unitPrice)
                 FROM Products)
```

This shows the highest price product in the Products table, and the Supplier’s company name for that product. What if we wish to see, for each supplier, the highest priced product from that supplier? Here is a correlated subquery that will produce that:

```
SELECT Suppliers.CompanyName, Products.ProductName, Products.unitPrice
FROM Products JOIN Suppliers
ON Products.SupplierID = Suppliers.SupplierID
WHERE unitPrice = (SELECT MAX(unitPrice)
                 FROM Products
                 WHERE Products.SupplierID = Suppliers.SupplierID)
```

Notice the emphasized table Suppliers. The Suppliers table is not included in the FROM statement of the subquery. The subquery is correlated with the parent query by this reference to Suppliers in the parent query.

Think of the database engine satisfying this SQL request in this way:

1. Perform the outer query, joining Products and Suppliers, ignoring the WHERE clause in the outer query
2. Look at the first row, R1, from Step 1
3. Get the SupplierID from R1
4. Execute the subquery by finding the MAX(unitPrice) of all products with the same SupplierID as R1
5. Does R1’s unitPrice equal the MAX(unitPrice) of all products with the same SupplierID as R1?
6. If so, keep it, otherwise throw it out
7. Go to the second row, R2
8. Etc.

In other words, the subquery gets executed for each record in the parent query. In a simple subquery, the subquery is only executed once, regardless of how many records are in the parent query.

Here are several examples comparing the simple subquery in the WHERE clause with the correlated subquery in the WHERE clause.

*Show all details of products that have an above average unitPrice.*

```sql
SELECT * 
FROM Products JOIN Categories 
    ON Products.CategoryID = Categories.CategoryID 
WHERE unitPrice > 
    (SELECT AVG(unitPrice) 
     FROM Products)
```

*Show all details of products that have an above average unitPrice within their category.*

```sql
SELECT * 
FROM Products JOIN Categories 
    ON Products.CategoryID = Categories.CategoryID 
WHERE unitPrice > 
    (SELECT AVG(unitPrice) 
     FROM Products 
     WHERE Products.CategoryID = Categories.CategoryID)
```

*Show all details of people who have below average weight.*

```sql
SELECT * 
FROM Person 
WHERE weight > 
    (SELECT AVG(weight) 
     FROM Person)
```

*Show all details of people who have below average weight compared to people of the same gender.*

```sql
SELECT * 
FROM Person OUTERPerson 
WHERE weight > 
    (SELECT AVG(weight) 
     FROM Person 
     WHERE Person.gender = OUTERPerson.gender)
```

In this last example, we used a table alias to rename the table in the outer SELECT statement. This lets us refer to the Person table from the outer SELECT as OUTERPerson. Within the subquery, OUTERPerson is the Person table from the parent SELECT statement.
In summary, subqueries are a way of including one query within another. In general, a subquery can be substituted for many expressions in a SQL statement. Subqueries make it possible to rephrase other SQL statements in a more clear, elegant manner. In some cases, subqueries allow for easy creation of information that may be very difficult to produce otherwise.

Simple subqueries do not depend on the parent (outer) query, and can generally be run as an independent statement. Simple subqueries should be thought of as executing before the parent query.

Correlated subqueries are special, in that they depend on results from the parent (outer) query to execute. Correlated subqueries are dependent on the parent query. Correlated subqueries should be thought of as executing once for each row in the parent query.

2.7.4 EXISTS

EXISTS is a SQL statement that is commonly used with correlated subqueries. EXISTS is a logical operator that returns True if a set of records is non-empty, and False if the set of records is empty. In other words if records exist in the set of records, EXISTS returns true, otherwise it returns false.

Here is a simple example:

```
SELECT *
FROM Suppliers
WHERE EXISTS (SELECT 17, 'Fred')
```

This SELECT statement always returns all the records in Suppliers table. Recall that the WHERE clause is evaluated for every record. Here is how the database goes about executing this query:

1. For each record in the Suppliers table
   a. Evaluate the subquery, which results in a 1-row, 2-column table containing the values 17 and 'Fred', always
   b. Does this 1-row, 2-column table have any records in it? Yes.
   c. EXISTS returns true
   d. Include this record

As you can see the EXISTS clause will evaluate to true for every record in the Suppliers table, so every record will be returned. This example isn’t very interesting. Could we create an example where the subquery depends on the main query? Sure – it’s called a correlated subquery.

Here’s a correlated subquery using EXISTS that is a little more interesting:

```
SELECT *
FROM Suppliers
WHERE EXISTS (SELECT ProductID
               FROM Products
               WHERE Products.SupplierID = Suppliers.SupplierID)
```
This SELECT statement returns “All Suppliers who supply us a product”; or more technically stated, all supplier records that match up with at least one product record. Note that the subquery is correlated with the main query. Here is how the database goes about executing this query:

1. For each record in the Suppliers table
   a. Evaluate the subquery, using the Suppliers.SupplierID from the main query. The subquery may contain records, or may not, depending on the value of Suppliers.SupplierID from the outer query.
   b. EXISTS returns true if the subquery has at least one record
   c. EXISTS returns false if the subquery has zero records in it
   d. Include the record if EXISTS = true for the record

You might say “But this is just another way to do a JOIN!” You would be right. Here’s how to phrase the same thing using a JOIN statement:

```sql
SELECT Suppliers.*
FROM Suppliers JOIN Products ON Products.SupplierID = Suppliers.SupplierID
```

However, there is a difference from the database engine’s point of view. Let’s say that there are 5 records in the Suppliers table, and 5 million records in the Products table. The database engine will perform the JOIN, generating 5 million records. (Remember, if a supplier supplies 1 million products, the JOIN will show that supplier 1 million times.)

You might say, but we can fix that using DISTINCT this:

```sql
SELECT DISTINCT Suppliers.*
FROM Suppliers JOIN Products ON Products.SupplierID = Suppliers.SupplierID
```

Now the database engine will return only 5 records. But how did the database engine get those 5 records? It generated 5 million matching records, then threw out duplicates to get the 5 DISTINCT records.

This shows a very important benefit of using EXISTS: performance! Here’s the EXISTS form again:

```sql
SELECT * FROM Suppliers
WHERE EXISTS (SELECT ProductID FROM Products WHERE Products.SupplierID = Suppliers.SupplierID)
```

Here is how the database engine performs this query:

1. Supplier Record #1:
   a. Start looking for a product from that supplier
   b. Found one? Stop!
2. Supplier Record #2:
a. Start looking for a product from that supplier
b. Found one? Stop!

3. Etc.

Notice that the database engine need not perform the entire JOIN. All it needs to do is find at least one record to satisfy the EXIST statement.

In summary, EXISTS can be used in place of a JOIN, and can give better performance. But there’s another use: show Suppliers that DO NOT match with any Products. This is where EXISTS is really helpful:

```sql
SELECT *
FROM Suppliers
WHERE NOT EXISTS (SELECT ProductID
                  FROM Products
                  WHERE Products.SupplierID = Suppliers.SupplierID)
```

This is a very common question for organizations to ask. It is sometimes called a “Find Un-matched Records” query. Here are some examples:

- Show students who are not enrolled in classes
- Show customers who have not placed an order
- Show products that have not been sold

Ironically, the NOT EXISTS does not give a performance improvement. Think of it this way: How can you prove that a certain Supplier doesn’t exist in the Products table? You have to check every Product record. So, there is a way to do a Find Un-matched query using outer joins, which will be covered later.

Finally, there is one last benefit to the EXISTS statement: it is sometimes clearer. For some queries, it is easier to work out complex logic using EXISTS. Or, once the query is complete, it is easier to read using EXISTS.
2.8 EXISTS, Recursive joins, UNION

2.8.1 EXISTS

2.8.1.1 The existential quantifier

2.8.1.2 Logical operator / function

2.8.1.3 WHERE EXISTS(SELECT * FROM ...) -> returns true if there are any records in the subquery

2.8.1.4 SELECT EXISTS(...),

2.8.1.5 Can be extremely fast in some circumstances

2.8.1.6 Commonly phrased as a correlated subquery

Consider a query that finds all suppliers that supply us products:

**Version A: DISTINCT/JION version**

```sql
SELECT DISTINCT Suppliers.CompanyName
FROM Suppliers JOIN
    Products ON Suppliers.SupplierID = Products.SupplierID
```

This can be rephrased as a correlated subquery with the EXISTS statement:

**Version B: Correlated Subquery/EXISTS version**

```sql
SELECT Suppliers.CompanyName
FROM Suppliers
WHERE EXISTS ( SELECT *
```
FROM Products
WHERE Products.SupplierID=Suppliers.SupplierID)

Even though the results of these two queries will always be the same, they are quite different in what we are asking for. Using the DISTINCT/JOIN version, we are asking that a JOIN be performed, at that duplicates be removed. Using the EXISTS version, we are asking to list suppliers that appear in the Products table. Note that the DISTINCT/JOIN version must find ALL matching products to perform the JOIN, while the EXISTS version must only find ONE matching product for each supplier. This can be a significant time savings in some cases.

(This is not to say that we are determining how the database engine satisfies the request. In SQL, we merely declare the set of records that we’d like to have returned; what steps the engine takes to satisfy the request is its’ own business. However, it is clear that Version B is giving the database engine a way of saving time.)

Consider the find unmatched query pattern that can be performed with a LEFT OUTER JOIN:

**Version A: LEFT JOIN / WHERE IS NULL**

```
SELECT Suppliers.supplierID
FROM Suppliers LEFT JOIN Products
    ON Suppliers.SupplierID = Products.SupplierID
WHERE Products.SupplierID IS NULL
```

This can also be phrased using a correlated subquery and NOT EXISTS:

**Version B: Correlated subquery/NOT EXISTS version**

```
SELECT Suppliers.supplierID
FROM Suppliers
WHERE NOT EXISTS (
    SELECT *
    FROM Products
    WHERE Products.SupplierID = Suppliers.SupplierID)
```

Similar to the previous example, Version A performs an entire LEFT OUTER JOIN, identifying ALL products that satisfy the JOIN (then the WHERE clause is applied). Version B checks each supplier, and if it finds only a single product for that supplier, it can stop looking at product records.

Another common rephrasing using EXISTS is for a dynamic IN clause:

**Version A: Dynamic IN**

```
SELECT Suppliers.SupplierID
FROM Suppliers
WHERE Suppliers.SupplierID IN (  
    SELECT Products.SupplierID
    FROM Products
    WHERE Products.UnitsInStock>50)
```
Version B: EXISTS/Correlated Subquery

```sql
SELECT Suppliers.SupplierID
FROM Suppliers
WHERE EXISTS (  
    SELECT *  
    FROM Products  
    WHERE Products.SupplierID = Suppliers.SupplierID  
    AND Products.UnitsInStock > 50)
```

2.8.2 Recursive Joins

2.8.2.1 Used for recursive relationships

2.8.2.2 Requires aliasing of one or more tables

2.8.2.3 Think of as creating multiple copies of the table & contents

2.8.2.4 Normal join rules apply

2.8.3 UNION / UNION ALL

2.8.3.1 Used to combine records from two record sets

2.8.3.2 Record sets must have the same number of columns

2.8.3.3 Columns must be type-compatible (not necessarily identical)

2.8.3.4 Uses

2.8.3.4.1 to simplify logic in SQL phrasing

2.8.3.4.2 Combine records from separate base tables

2.8.4