

---

# *SQL Subqueries*

## *Objectives of the Lecture :*

- To consider the general nature of subqueries.
- To consider the various uses of subqueries.
- To contrast simple with correlated subqueries.

## ***Closure Under the Algebra / SQL***

- Every relational algebra operator takes either one or two relations as operands, and returns a relation as a result.
- Similarly in SQL, every **SELECT** statement - i.e. a query - takes one or more tables as an operand(s) and returns another table as a result.
- Thus it is possible to use a **SELECT** statement as a subquery within another SQL statement *wherever that outer statement needs a value*, i.e. within :
  - CREATE TABLE statements,
  - INSERT statements,
  - DELETE statements,
  - UPDATE statements,
  - SELECT statements.

SQL tables are not the same as relations, but a similar principle to that of the relational 'Principle of Closure' applies.

## ***Subquery SELECT STATEMENTS***

- Anything allowable in a normal **SELECT** statement is allowed in a subquery **SELECT** statement, *except* :
  - the keyword **DISTINCT** in the **SELECT** phrase,
  - an **ORDER BY** phrase.

This is because SQL can always ignore duplicate rows and row ordering in the 'sub table' result.
- For queries, subqueries are logically unnecessary in SQL. However this was not originally realised. They are retained because they can be useful for some queries, by expressing them in a natural way.
- Depending on the circumstances, the subquery may be required to return
  - a single value,
  - a column of values,
  - a row of values,
  - a table of values.

Note the exceptions, which prevent the Principle of Closure being applied with complete consistency.

## ***Subquery with CREATE TABLE***

Example :-

```
CREATE TABLE Manager
AS
SELECT Emp_No, Emp_Name, Salary
FROM Emp JOIN Dept
ON( Emp_No = Manager_No ) ;
```

Column names are inherited.

The subquery returns a **TABLE OF VALUES** to 'Manager'.

Columns which are NOT NULL in 'Emp' are also NOT NULL in 'Manager'.

No other Integrity Constraints are inherited.

This use of **CREATE TABLE** creates an SQL table *and* fills it with rows of data, unlike the normal use of **CREATE TABLE** which only creates a table.

Integrity constraints may need to be added to the table so created.

Note that the purpose of a table is to permanently store its data contents in the DB. So one needs to be careful about a table whose data comes from pre-existing DB tables, to ensure that it does not unnecessarily duplicate data in the DB.

## ***Subquery with INSERT***

Example :-

```
INSERT INTO Manager  
SELECT Emp_No, Emp_Name, Salary  
FROM Emp JOIN Dept  
ON( Emp_No = Manager_No ) ;
```

The subquery  
returns a  
***TABLE OF VALUES***  
to 'Manager'.

The usual rules about data types apply.

The subquery replaces the

**VALUES ( ..... )**

phrase of the normal **INSERT** statement.

If required, a list of column names can be put after the table name and before the subquery, just as can appear in a normal **INSERT** statement.

Naturally, whether the column names appear in the **INSERT** statement or whether the default order of column names is used (i.e. the order in which the column names appeared in the original **CREATE TABLE** statement for this table), the columns specified in the subquery must match in type the columns of the table that is receiving the result of the subquery.

This is the only way of inserting more than one row of data into a table with one **INSERT** statement.

## ***Subquery with DELETE***

Example :-

```
DELETE FROM Emp
WHERE Dept_No =
  ( SELECT Dept_No
    FROM Dept
    WHERE Dept_Name = 'Sales' );
```

The subquery  
returns a  
*SINGLE VALUE*  
to 'Emp'.

All the employees of the 'Sales' department will be deleted.

If the subquery returns more than one value,  
then an ERROR will occur.

The subquery is useful if we don't know what the ID number of the department is, and would have to look it up in the DB anyway.

An '=' comparison with the result of the subquery would be rendered illogical if the subquery were to return more than one department ID number, and so SQL will return an error if this happens. Since *Dept\_Name* is a candidate key in *Dept*, the error should not arise in this example.

## *Subquery with UPDATE*

Example :-

```
UPDATE Emp
SET ( Salary, Dept_No ) =
  ( SELECT Salary, Dept_No
    FROM Emp
    WHERE Emp_No = 'E5' )
WHERE Emp_No = 'E8'
```

The subquery returns a *SINGLE ROW VALUE* to 'Emp'.

Employee 'E8' is assigned the salary and department of employee 'E5'.

If the subquery returns more than one value, then an **ERROR** will occur.  
The order of columns in the outer and inner statements must be consistent.

In this case, the subquery saves us from looking up the values in the DB and then typing them in to do the update.

A **SET** assignment in an SQL **UPDATE** statement can only take *one* row. There would be a logical error, and hence an SQL error, if the subquery were to return more than one row. Since *Emp\_No* is a candidate key in *Emp*, the error should not arise in this example.

Our **UPDATE** is only required to update one row in the *Emp* table – namely that of the one employee 'E8'. However if it were appropriate, we could use the single row returned in the subquery to update more than one row in the *Emp* table, all of the rows being updated to the same values specified in the **SET** assignment.

## *Subquery within a Query*

---

Wherever an expression can appear in a **SELECT** statement, it can be replaced by a subquery/**SELECT** statement.

**SELECT** .....  
**FROM** .....  
**WHERE** .....

**Subquery**

DBMSs that meet the SQL3 standard.

This allows the SQL built-in order of execution to be overcome  
⇒ more like relational algebra.

Originally subqueries could only be used in the **WHERE** phrase.  
Most subqueries are still used in the **WHERE** phrase.

SQL's standard order of execution is not always what is required, as was seen in the lectures covering the SQL **GROUP BY** facility.

Note that not all SQL DBMSs can be relied upon to meet the SQL3 standard, but all should cope with subqueries in the **WHERE** phrase, which is the traditional location for subqueries and the most useful.

### Example : Subquery within a Query (1)

Which child(ren) are taller than Chloe ?

SELECT statement to query  
*"Who is taller than ... ?"*

↩

SELECT statement to query  
*"How tall is Chloe ?"*

Work 'bottom upwards';  
 work out which data is to be  
 passed to the next query up.

We don't necessarily want to know how tall Chloe is in order to want to ask the question.

### Example : Subquery within a Query (2)

Which child(ren) are taller than Chloe ?

Who is  
taller than  
1.12 ?

Name	Height
-----	-----
Ali	1.18
Bill	1.17

↩

Height
-----
1.12

How tall is  
Chloe ?

We could always literally execute the two queries one after the other to get the answer we really want.

(Assume *Pupil* is a table holding details of school children).



### Example : Subquery within a Query (3)

Which child(ren) are taller than Chloe ?

```
SELECT Name, Height
FROM Pupil
WHERE Height >
      ( SELECT Height
        FROM Pupil
        WHERE Name = 'Chloe' );
```

Columns compared must be of the same data type.

Name	Height
Ali	1.18
Bill	1.17

In the finished version of the query, a '>' comparison with the result of the subquery would be rendered illogical if the subquery were to return more than one height, and so SQL will return an error if this happens. Hopefully there is only one child named 'Chloe' in *Pupil*, or the error will occur.

A query is not limited to having *one* subquery in it. A query may contain as many subqueries as required, and a subquery may contain a nested subquery within it. The following 2 examples illustrate this :-

## *Multiple Subqueries : Example*

Which child(ren) are taller and older than Chloe ?

```
SELECT Name FROM Pupil
WHERE Height >
      ( SELECT Height FROM Pupil
        WHERE Name = 'Chloe' )
AND DoB <
      ( SELECT DoB FROM Pupil
        WHERE Name = 'Chloe' ) ;
```

Name

-----

Ali

Bill

## *Nested Subqueries : Example*

Get the names of those employees who work in the department with the biggest budget.

```
SELECT Emp_Name
FROM Emp
WHERE Dept_No =
      ( SELECT Dept_No
        FROM Dept
        WHERE Budget =
              ( SELECT Max( Budget )
                FROM Dept ) ) ;
```

Subqueries can be nested  
to any depth.

Still work 'bottom upwards'.

## Single Value Comparisons

```

SELECT .....
FROM .....
WHERE ..... θ
           ( SELECT .....
             FROM .....
             WHERE ..... );
        
```

θ  
 can be  
 =  
 <>  
 >  
 >=  
 <  
 <=

Must ensure that the subquery returns a SINGLE VALUE,  
 i.e. a table of one row and one column, or an ERROR results.

These are the possible single value comparators available in SQL.

For each row in the table specified by the **FROM** phrase of the outer query - note that this table can be the result of joining 2 or more tables - the single, scalar value specified in the **WHERE** phrase of the outer query is compared with a single value generated by executing the subquery, using one of the comparators given above. Each outer table row for which the comparison yields *true* appears in the retrieved result; if the comparison yields *false*, the outer table row does not appear in the result.

### *Ensuring Single Value Comparisons*

```
SELECT Emp_Name
FROM Emp
WHERE Dept_No =
      ( SELECT Dept_No
        FROM Dept
        WHERE Budget =
          ( SELECT Max( Budget )
            FROM Dept ) ) ;
```

Use an aggregate function, **AND** treat the table as one group.

In the **WHERE** comparison, use a column and comparison that is **GUARANTEED** to return one row in **SELECT**.

Example : a Candidate Key with an '=' comparison.

Suppose there is more than one department with the maximum-sized budget ?!

Logically it is possible for a value in the **WHERE** phrase of the outer query to be compared to multiple values returned by the subquery. What does the logic of the query require ? It is important to decide this when designing the query, and then use the appropriate comparator.

Consider now the comparators available for use with multiple values returned by the subquery.

## Multiple Value Comparisons

```

SELECT .....
FROM .....
WHERE ..... θθ
           ( SELECT .....
             FROM .....
             WHERE ..... );
    
```

**θθ**  
 can be  
**IN**  
**NOT IN**  
**θ ANY**  
**θ ALL**  
**EXISTS**  
**NOT EXISTS**

**θ** is any of =, <>, >, >=, <, <=

The subquery can return MULTIPLE VALUES,  
 i.e. one column of many rows.  
 However there is no problem if only one row is returned.

For each row in the table specified by the **FROM** phrase of the outer query - note that this table can be the result of joining 2 or more tables - the single, scalar value specified in the **WHERE** phrase of the outer query is compared with the set of values generated by executing the subquery, using one of the comparators given above. Each outer table row for which the comparison yields *true* appears in the retrieved result; if the comparison yields *false*, the outer table row does not appear in the result.

The individual kinds of comparator are now considered.

## ***IN Comparison : Example***

Get the names of those employees who work in departments with budgets worth over £125,000.

```
SELECT Emp_Name
FROM Emp
WHERE Dept_No IN
      ( SELECT Dept_No
        FROM Dept
        WHERE Budget > 125000 );
```

Can't use '=' comparison.  
Subquery can return many  
department numbers.

**IN** is the same comparator as is used with a set of literal values.

Example : **WHERE** Dept\_No **IN** ( 'D1', 'D2', 'D3' ).

The subquery is used to supply the values instead of them being written out.

**IN**, and **NOT IN**, are very common comparators to use, because they are analogous to '=' (and '<>') in single value comparisons.

## ***ANY and ALL : Examples***

They are used to compare a single value with each of the individual values in the set returned by the subquery.

<pre> SELECT Emp_Name FROM Emp WHERE Salary &gt;ANY       ( SELECT Salary         FROM Emp         WHERE Dept_No = 'D1' ); </pre>	<p>If '&gt;' is <i>true</i> for <i>at least one</i> value, ANY returns <i>true</i>, else <i>false</i>.</p>
<pre> SELECT Emp_Name FROM Emp WHERE Salary &gt;ALL       ( SELECT Salary         FROM Emp         WHERE Dept_No = 'D1' ); </pre>	<p>If '&gt;' is <i>true</i> for <i>every</i> value, ALL returns <i>true</i>, else <i>false</i>.</p>

In the first example, the comparison is true if the 'outer query *Salary*' is bigger than any in 'D1', i.e. as long as it is greater than at least the smallest value returned by the subquery.

In the second example, the comparison is true only if the 'outer query *Salary*' is bigger than all those in 'D1', i.e. if it is greater than the biggest value returned by the subquery.

The following are useful ways of understanding some of the comparators :

- <ANY means less than the maximum value in the set of values returned.
- =ANY means the same as IN.
- >ANY means more than the minimum value in the set of values returned.
- <ALL means less than the minimum value in the set of values returned.
- >ALL means more than the maximum value in the set of values returned.

## ***EXISTS : Example***

```
SELECT Emp_Name, Salary, Emp_No
FROM Emp
WHERE EXISTS
  ( SELECT *
    FROM Emp
    WHERE Salary > 100000 ) ;
```

If the subquery contains at least one row, then **EXISTS** returns *true*, otherwise *false*.

Hence there is normally no interest in the particular column(s) returned, and so **\*** is used in the **SELECT** phrase.

Example : all the employees' names, IDs, and salaries are retrieved if there is at least one employee with a salary in excess of £100,000, presumably a suspicious circumstance !

If no employee has a salary over £100,000, no rows are retrieved in the outer query.

## ***'Simple' and 'Correlated' Subqueries***

- All the subqueries seen so far *are evaluated once*, and the value returned used to evaluate the outer query.  
These are called **Simple** or **Non-Correlated** queries.

- A **Correlated** subquery is one that contains a reference to one or more of the tables that are in its outer query, and therefore needs to be *evaluated once for each row* returned in its outer query.

Example :

```
SELECT * FROM Emp
WHERE EXISTS
  ( SELECT * FROM Dept
    WHERE Manager_No = Emp_No ) ;
```

Get the employee details of all department managers.

In this query, the subquery is evaluated for each row in the outer table *Emp*, whereas in the previous query - see above - the subquery was only evaluated once.



## ***‘Correlated’ Subquery : Example (2)***

Get the names of the departments that have married employees.

```
SELECT Dept_Name
FROM Dept
WHERE 'M' IN
      ( SELECT Marital_Status
        FROM Emp
        WHERE Emp.Dept_No = Dept.Dept_No ) ;
```

Note : 'M'  
does not  
appear in  
table *Dept*.

Because of the **WHERE** phrase,  
the subquery must be executed  
for every row in *Dept* in order to  
see whether that row is to be  
retrieved or not.

This example shows more clearly the correlation between subquery and outer query, since the table name *Dept* has to appear in the subquery to distinguish the *Dept\_No* column in it from the *Dept\_No* column in *Emp*. In the previous query, the columns in the two tables could be distinguished from each other solely by their column names, and so the name of the table in the outer query did not have to appear explicitly in the subquery.

Correlated subqueries usually correspond to normal join queries, i.e. join queries that have no subqueries, and can usually be expressed as such.

It is sometimes recommended that they are translated into such join queries, because most SQL DBMSs will execute them faster.

Certain correlated subqueries can correspond to normal set queries rather than join queries, due to the comparator used with the correlated subquery.

Note that whether a subquery is correlated or not has solely to do with the tables referenced in the subquery, and nothing to do with the kind of comparator used with it.

## Correlated Subquery : Example (3)

Which projects have exactly 2 employees working on them ?

```
SELECT * FROM Proj
WHERE 2 =
      ( SELECT Count(*) FROM Alloc
        WHERE Alloc.Proj_No = Proj.Proj_No );
```

One table.

Note  
required  
form of  
comparison.

Cannot use a *Natural Join* here, because only those rows of *Alloc* are required that correspond to the same project as the row of *Proj* currently being evaluated

```
SELECT * FROM Proj
WHERE 2 =
      ( SELECT Count(*)
        FROM Alloc NATURAL JOIN Proj );
```

Two tables.

The *Natural Join* is executed once. If the join result has exactly 2 rows, all the rows of *Proj* are retrieved; otherwise no rows are retrieved

In a correlated subquery, the fact that 2 table names can be referenced in the **WHERE** phrase of the subquery can make the subquery look like an old SQL1 standard join. One might therefore be tempted to re-write the subquery with a modern SQL2 standard join in the subquery **WHERE** phrase. However this would be a mistake; it would reduce the subquery to a simple or non-correlated subquery, by putting both the tables concerned in the subquery. In a correlated query, only one of the tables whose columns are being compared comes from the subquery; the other comes from the outer query.

In the first example, the correlated subquery, a row is retrieved from *Proj* if there are exactly 2 rows in the whole of the *Alloc* table with the same *Proj\_No* column value as that row in *Proj*. This gives the right answer to the query, as a row in *Proj* corresponds to a project.

In the second example, the non-correlated query, all the rows - i.e. projects - in *Proj* will be retrieved if the natural join of *Proj* and *Alloc* has exactly 2 rows in it, and otherwise no rows will be retrieved. This gives the wrong answer to the query.

Logically we ought to be able to write

**WHERE ( subquery ) comparator value**

as well as

**WHERE value comparator ( subquery )**

which is what we always write. However SQL only allows the latter version.

## ***Subquery in a FROM Phrase***

Example : Get the *maximum* of the *average* departmental salaries.

```
SELECT  Max( AvSal ) AS MaxAvSal
FROM    ( SELECT      Avg( Salary ) AS AvSal, Dept_No
          FROM        Emp
          GROUP BY    Dept_No ) AS Emp_Averages;
```

This use of a subquery is sometimes known as an *Inline View*.

This is a useful way of applying an aggregate function to the result of an aggregate function.

## ***Subquery in a SELECT Phrase***

Example : Get the additional amount of salary that each employee receives over and above the minimum company salary.

```
SELECT  Emp_Name, ( Salary -
                  ( SELECT  Min( Salary )
                    FROM    Emp ) )
          AS Additional
FROM    Emp ;
```

This is a useful way of doing a scalar calculation in each row that requires the resulting value of an aggregation.

Although the subqueries in these two examples are non-correlated, correlated subqueries can also be used in the **SELECT** and **FROM** phrases.

