Slide 1

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.

Slide 2

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.

Slide 3

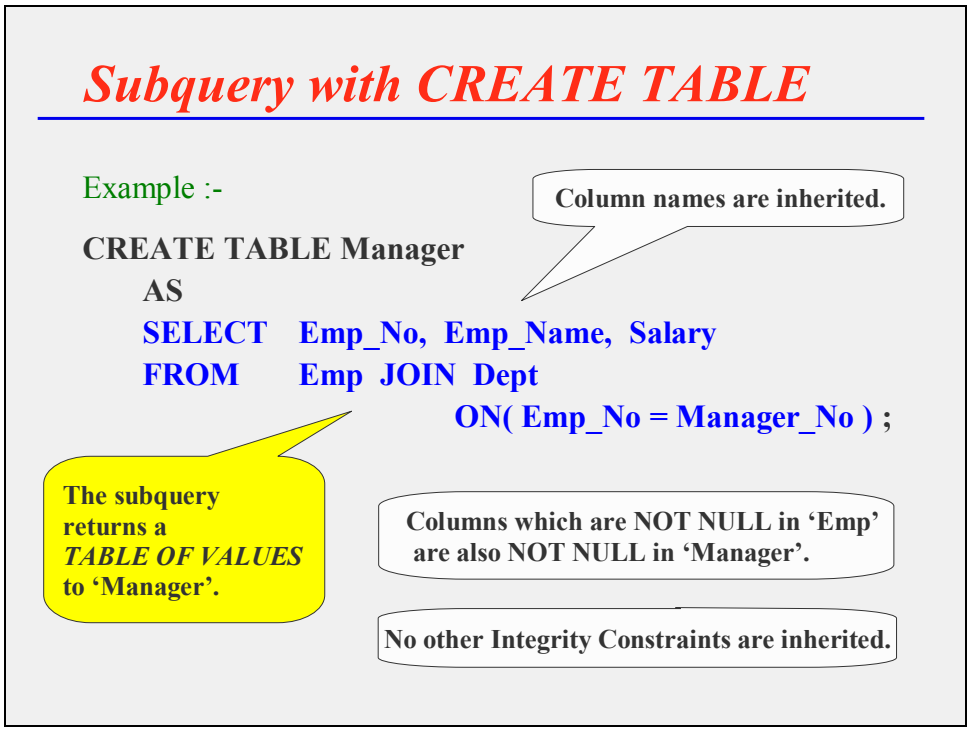
Subquery SELECT STATEMENTS

- 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.

Slide 4

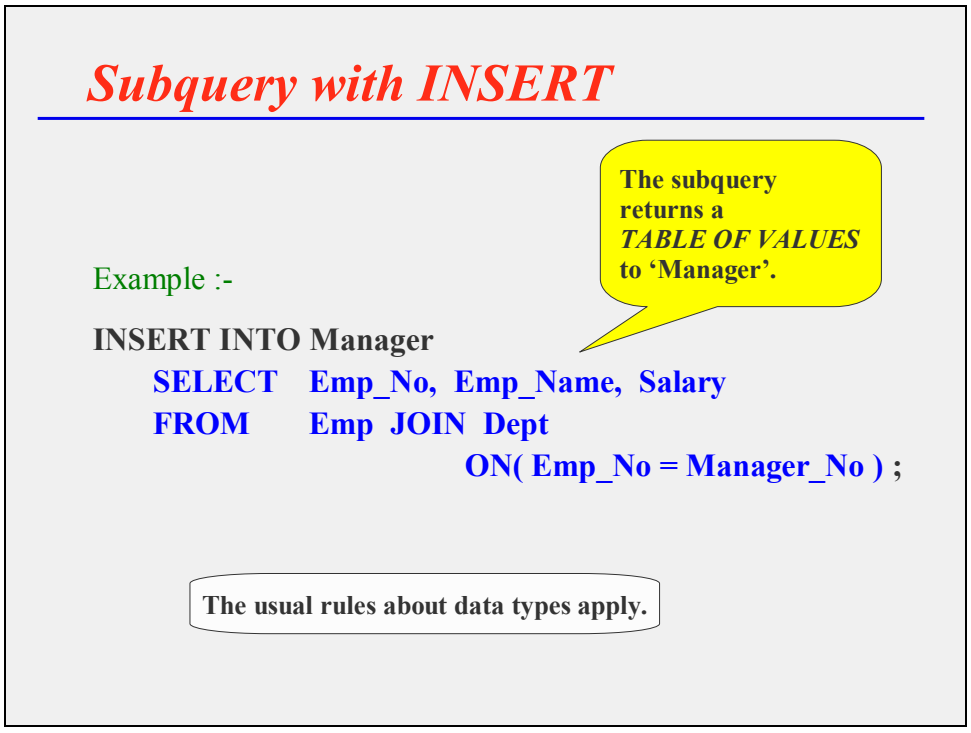


This use of **CREATE TABLE** creates an SQL table ***and*** fills it with rows of data, unlike the normal use of **CREATE TABLE** which only creates a table.

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.

Slide 5



The subquery replaces the

VALUES (……)

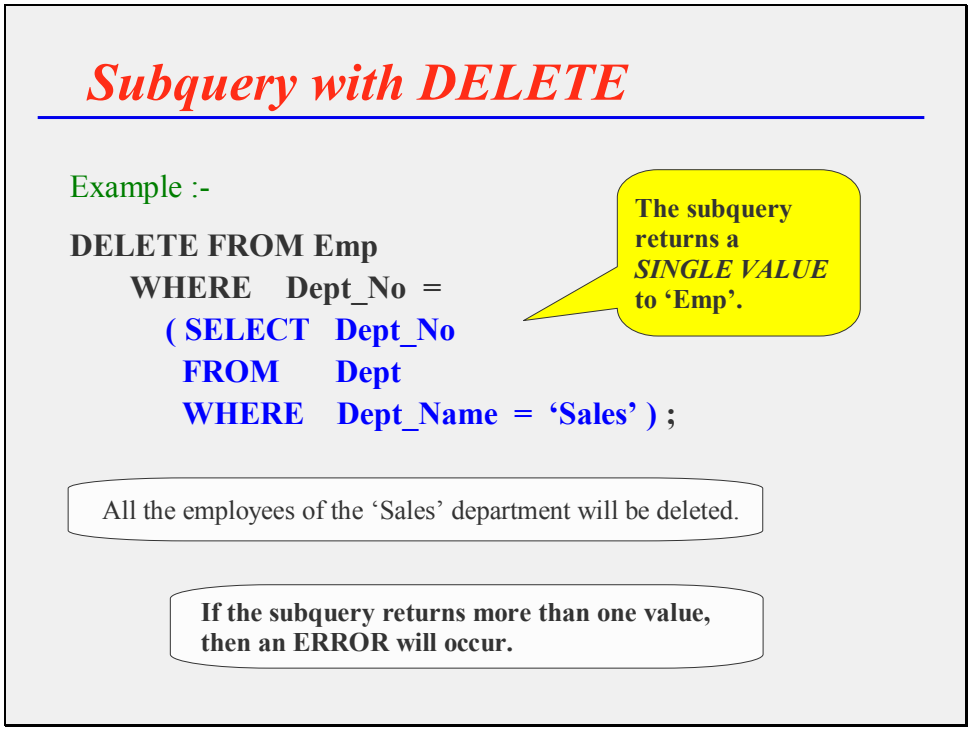
phrase of the normal **INSERT** statement.

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.

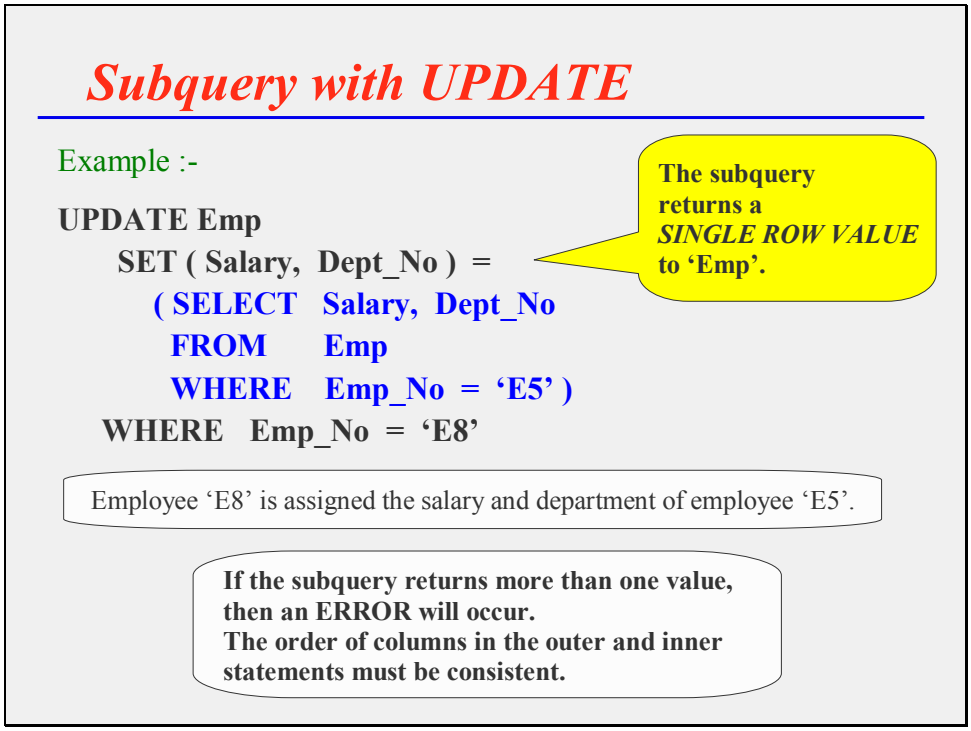
Slide 6



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.

Slide 7

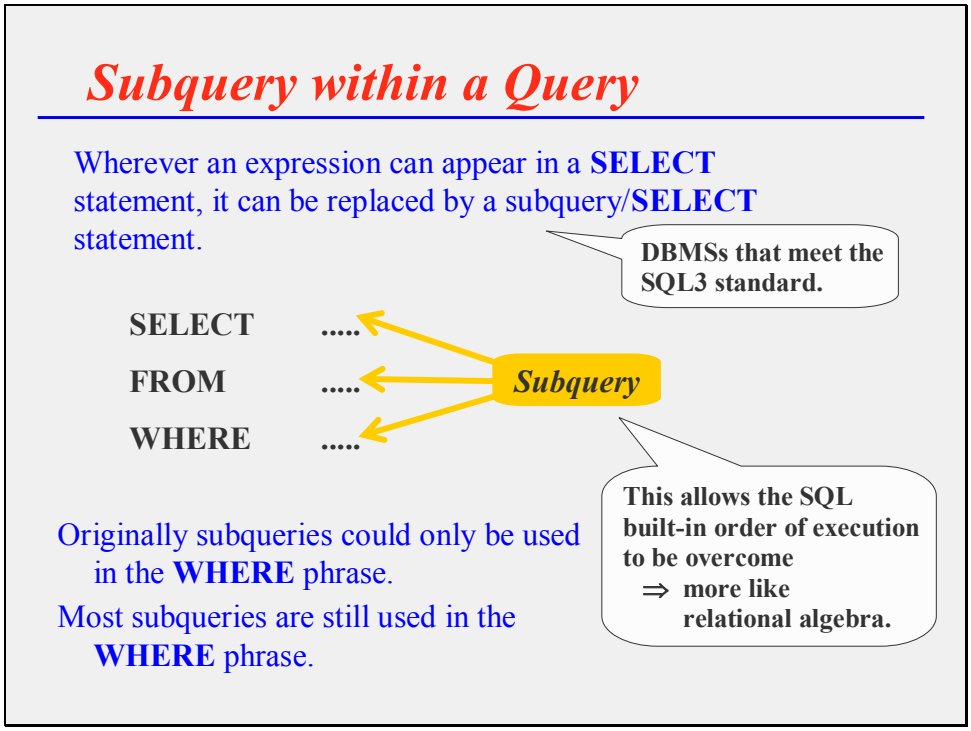


In this case, the subquery saves us from looking up the values in the DB and then typing them in to do the update.

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.

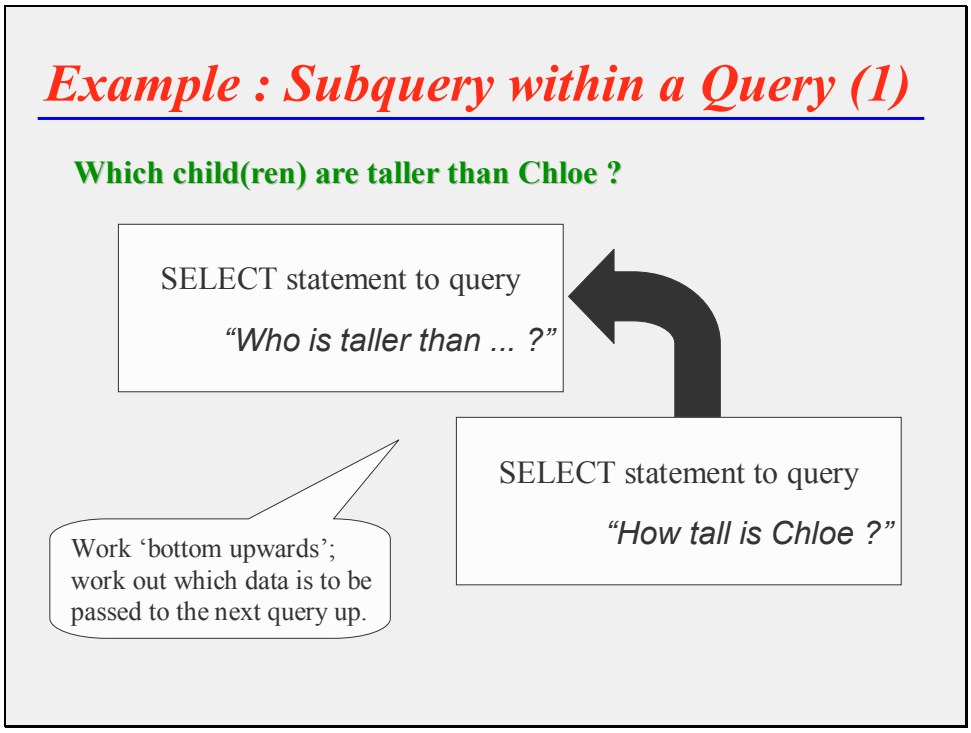
Slide 8



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

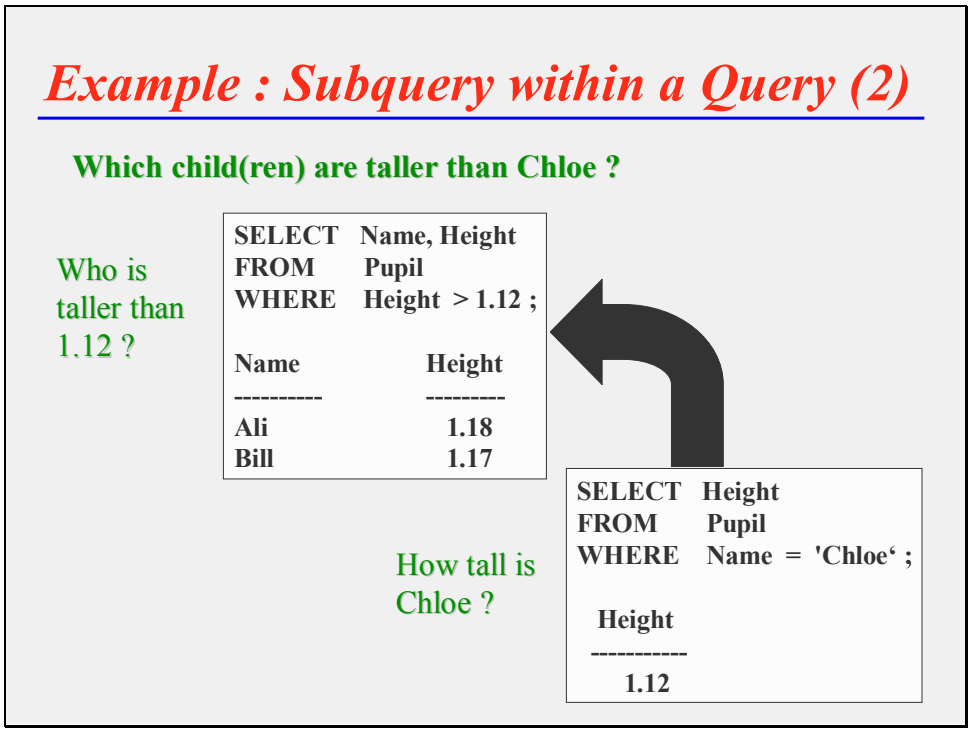
Note that not all SQL DBMSs can be relied upon to meet the SQL3 standard, but all should cope with subqueries in the **WHERE** phrase, which is the traditional location for subqueries and the most useful.

Slide 9



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

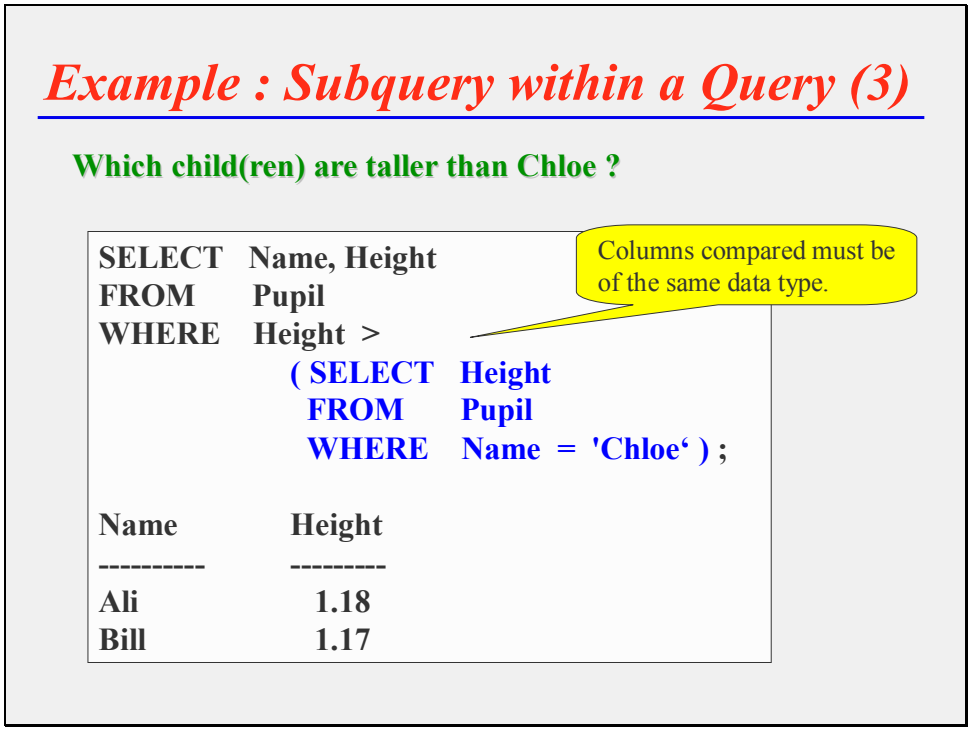
Slide 10



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).

Slide 11



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 :-

Slide 12

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
```

Slide 13

Nested Subqueries : Example

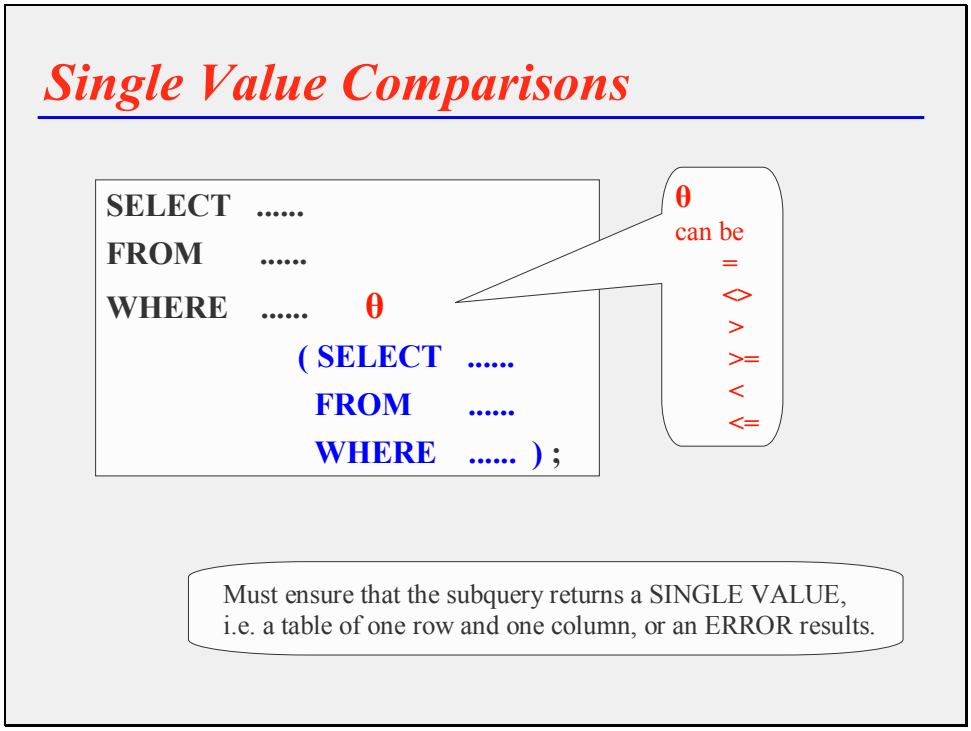
Get the names of those employees who work in the department with the biggest budget.

```
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'.

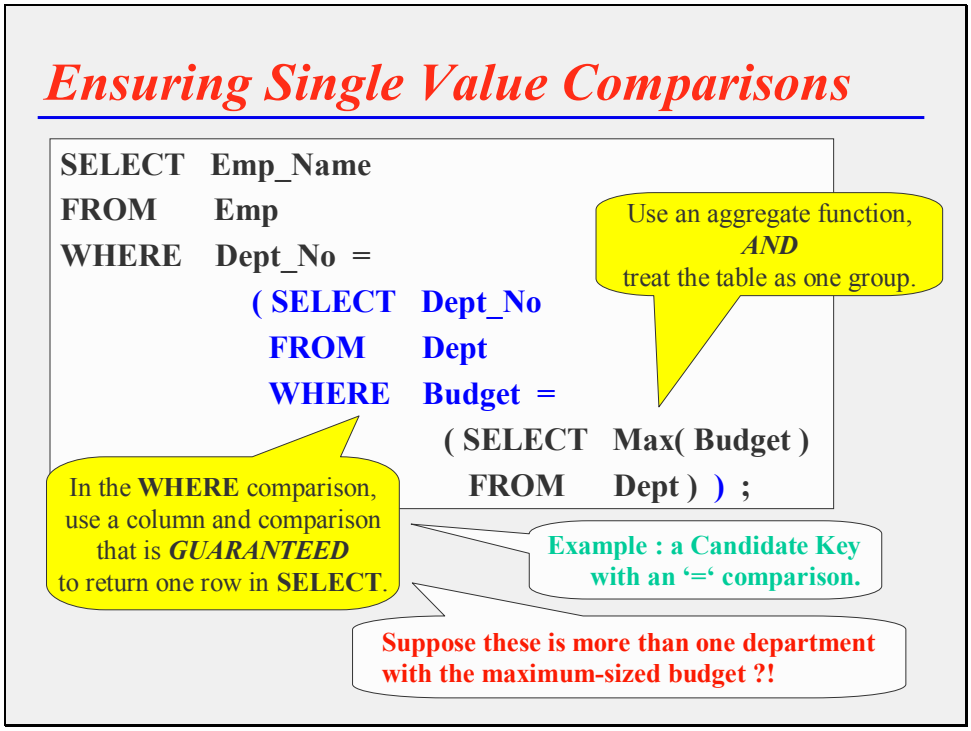
Slide 14



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 joing 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.

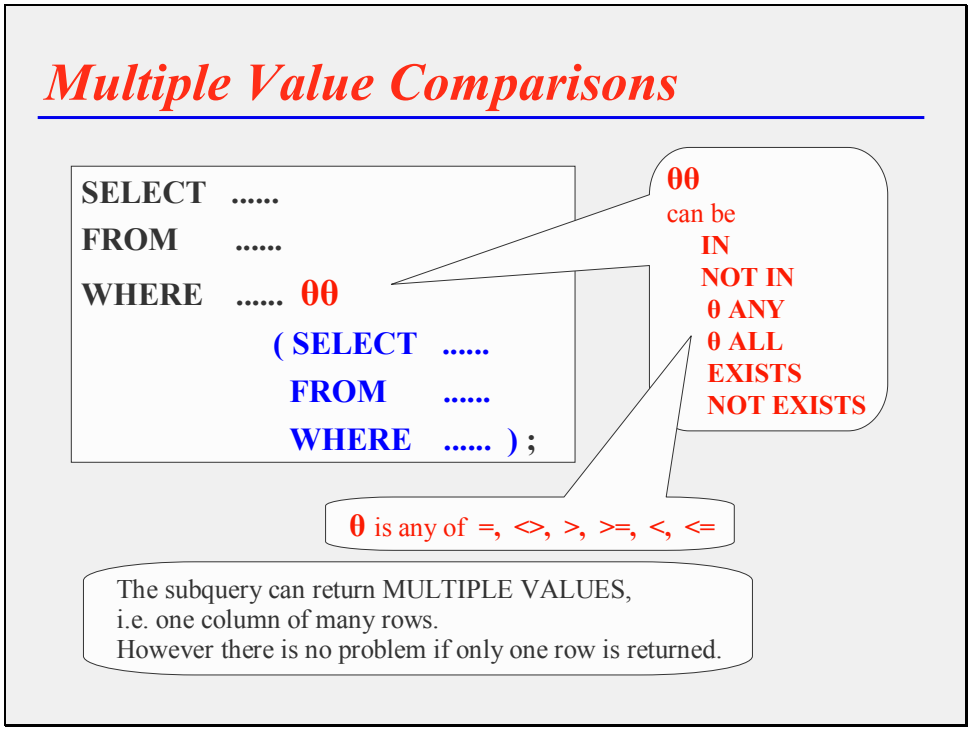
Slide 15



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.

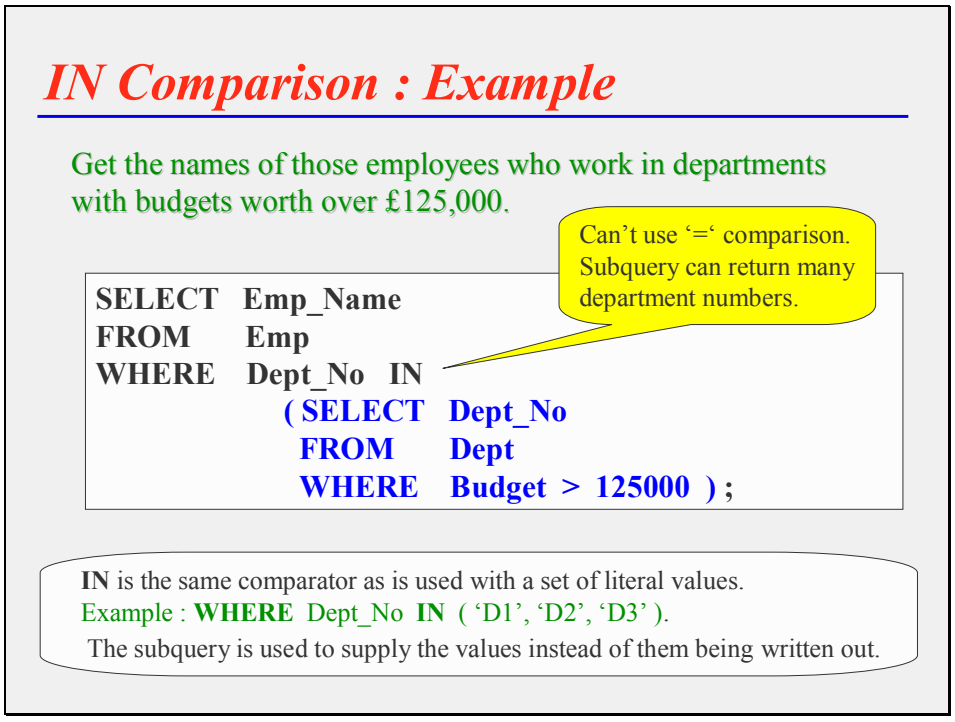
Slide 16



For each row in the table specified by the **FROM** phrase of the outer query - note that this table can be the result of joing 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.

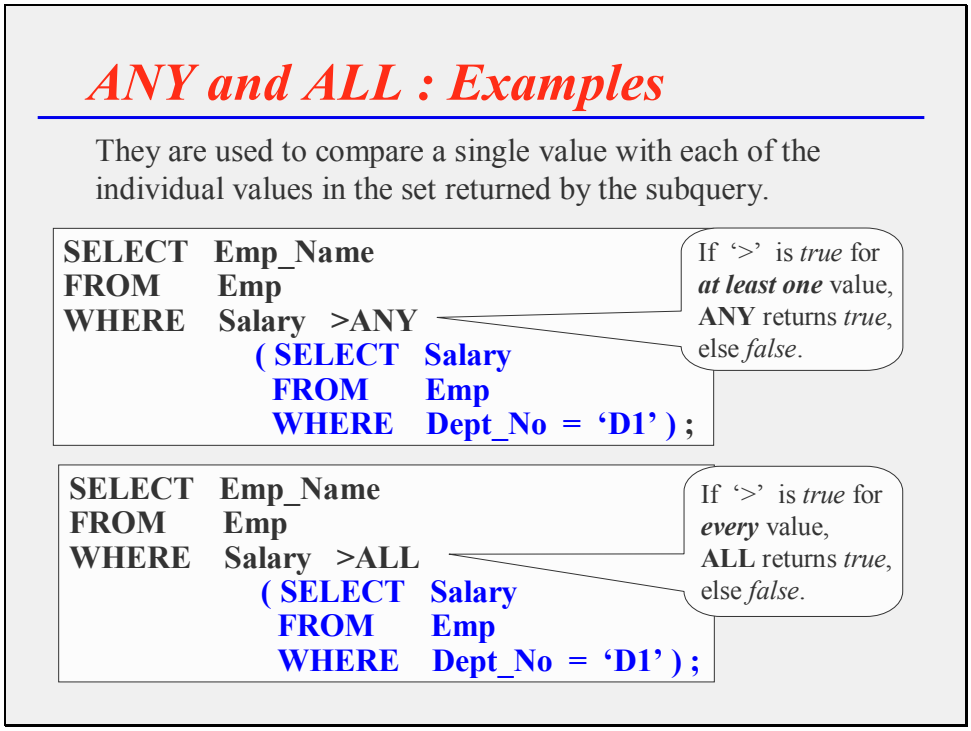
The individual kinds of comparator are now considered.

Slide 17



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

Slide 18



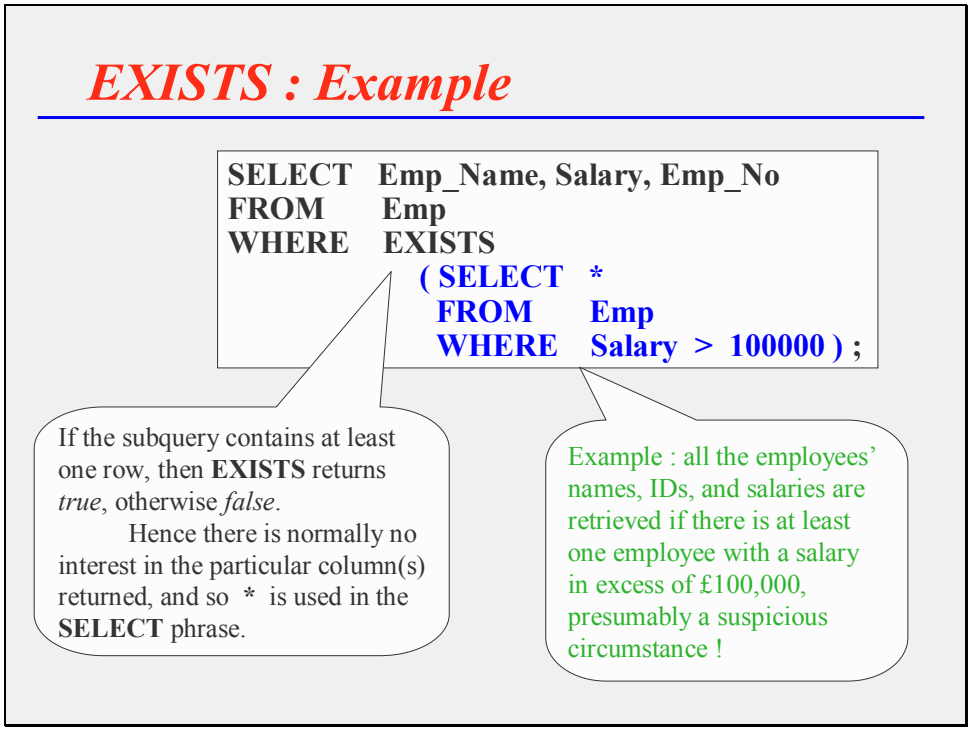
In the first example, the comparison is true if the 'outer query *Salary*' is bigger than any in 'D1', i.e. as long as it is greater than at least the smallest value returned by the subquery.

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 :

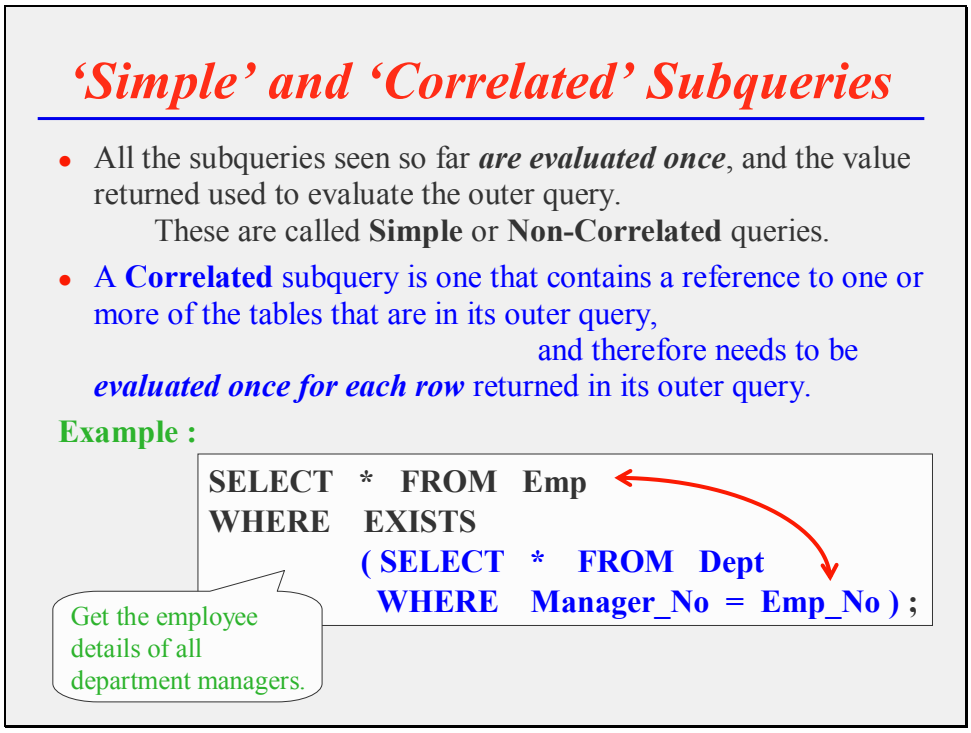
- **<ANY** means less than the maximum value in the set of values returned.
- **=ANY** means the same as **IN**.
- **>ANY** means more than the minimum value in the set of values returned.
- **<ALL** means less than the minimum value in the set of values returned.
- **>ALL** means more than the maximum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

Slide 19



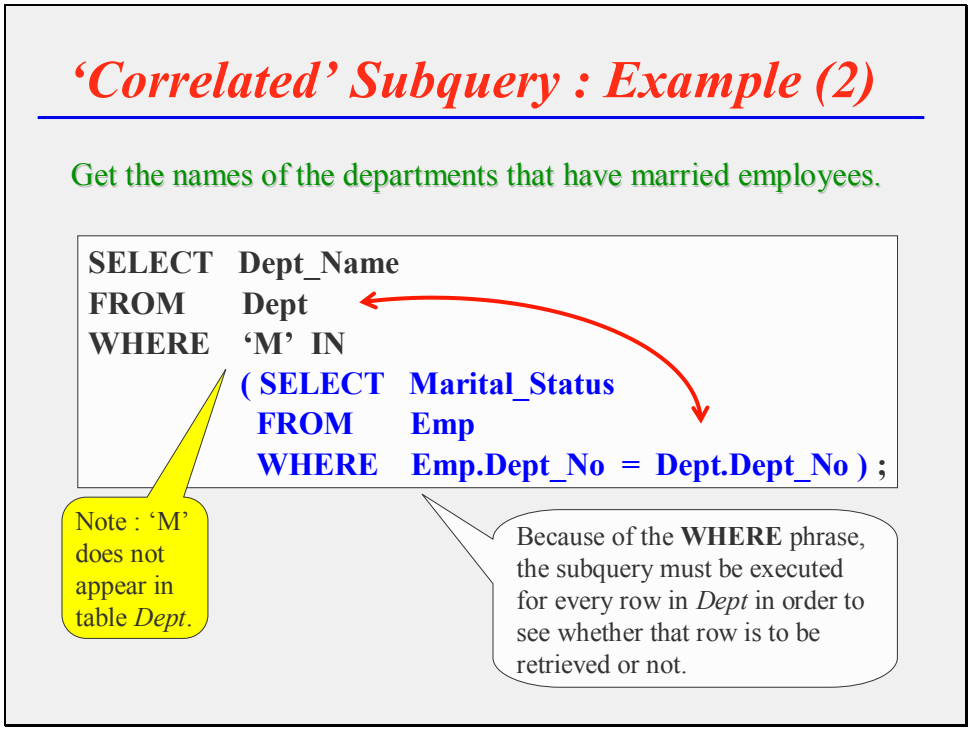
If no employee has a salary over £100,000, no rows are retrieved in the outer query.

Slide 20



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.

Slide 21



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.

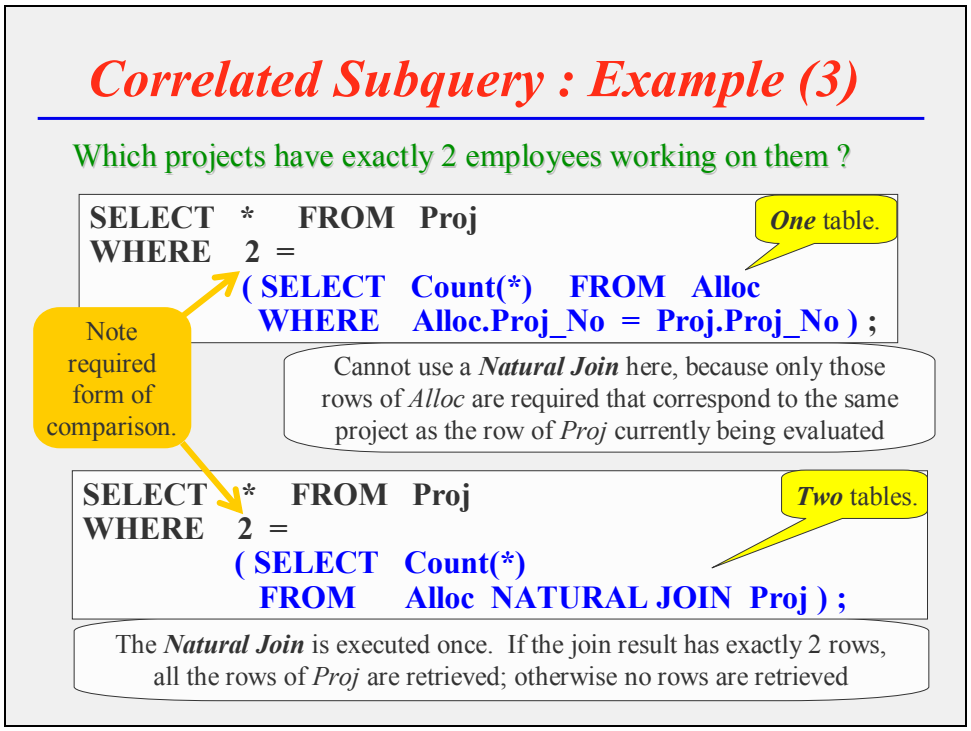
Correlated subqueries usually correspond to normal join queries, i.e. join queries that have no subqueries, and can usually be expressed as such.

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.

Slide 22



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.

Slide 23

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.

Slide 24

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.