Simple Select

In SQL a query has the following (simplified) form (components in brackets [ ] are optional):

select [distinct] <column(s)>
from <table>
[where <condition>]
[order by <column(s) [asc|desc]>]

The columns to be selected from a table are specified after the keyword select. This operation is also called **projection**

For example, the query

Select LOC, DEPTNO from DEPT; **For selecting all columns you can use "*"**

It may also contain an expression involving arithmetic operators or various SQL functions etc.

select ENAME, DEPTNO, SAL * 1.55 from EMP;

select DEPTNO from EMP;

Retrieves the department number for each record. One deptno may appear more than once, duplicate records are not automatically eliminated. We have to use distinct for eliminating duplicates.

For specifying a sorting order the order by clause is used and which has one or more attributes listed in the select clause as parameter. **desc** specifies a descending order and **asc** specifies an ascending order (this is also the default order). For example, the query

```
select ENAME, DEPTNO, HIREDATE from EMP;
from EMP
order by DEPTNO asc, HIREDATE desc;
```

ENAME	DEPTNO	HIREDATE
FORD	10	03-DEC-81
SMITH	20	17-DEC-80
BLAKE	30	01-MAY-81
WARD	30	22-FEB-81
ALLEN	30	20-FEB-81

Where clause can contain simple conditions based on comparison operators and combined using the logical connectives and, or, and not to form complex conditions. Conditions may also include pattern matching operations and even subqueries.

Some Example :

select JOB, SAL from EMP where (MGR = 7698 or MGR = 7566) and SAL > 1500;

For all data types, the comparison operators =, != or <>,<, >,<=, => are allowed in the conditions of a where clause.

Set Conditions: <column> [not] in (<list of values>)

select * from DEPT where DEPTNO in (20,30);

select * from EMP where MGR is not null; Note: the operations = null and ! = null are not defined!

Domain conditions: <column> [not] between <lower bound> and <upper bound>

select EMPNO, ENAME, SAL from EMP where SAL between 1500 and 2500;

select ENAME from EMP where HIREDATE between '02-APR-81' and '08-SEP-81';

In order to compare an attribute with a string, it is required to surround the string by apostrophes,
select ENAME from EMP where LOCATION = 'DALLAS'.

like operator for **pattern matching**

Together with this operator, two special characters are used: the percent sign **%** (also called wild card), and the **underline** , also called position marker. For example, if one is interested in all tuples of the table DEPT that contain two C in the name of the department, the condition would be where **DNAME like '%C%C%'**.

The percent sign means that any **(sub)string** is allowed there, even the **empty string**. In contrast, the **underline** stands for exactly **one character**. Thus the condition where DNAME like '%C_ C%' would require that exactly one character appears between the two Cs. To test for inequality, the **not** clause is used.

Aggregate functions are statistical functions such as count, min, max etc. They are used to compute a single value from a set of attribute values of a column:

count Counting Rows –
How many tuples are stored in the relation EMP? **select count(*) from EMP;**
How many different job titles are stored in the relation EMP?
select count(distinct JOB) from EMP;
min, max – Min, max value within selected rows
List the minimum and maximum salary. **select min(SAL), max(SAL) from EMP;**
Compute the difference between the minimum and maximum salary.
select max(SAL) - min(SAL) from EMP;
sum - Computes the sum of values (only applicable to the data type number)

Sum of all salaries of employees working in the department 30.
select sum(SAL) from EMP where DEPTNO = 30;

avg Computes average value for a column (only applicable to the data type number)

Note: avg, min and max ignore tuples that have a null value for the specified attribute, but count considers null values.

Data Modifications in SQL

Insert : The simplest way to insert a tuple into a table is to use the insert statement

insert into <table> [(<column i, . . . , column j>)] values (<value i, . . . , value j>);

For each of the listed columns, a corresponding (matching) value must be specified. Thus an insertion does not necessarily have to ***follow the order of the attributes as specified in the create table statement***. If a **column is omitted**, the **value null** is inserted instead. If **no column list is given**, however, for **each column as defined in the create table statement a value must be given**.

Examples:

```
insert into PROJECT(PNO, PNAME, PERSONS, BUDGET, PSTART)
        values(313, 'DBS', 4, 150000.42, '10-OCT-94');
or
insert into PROJECT    values(313, 'DBS', 7411, null, 150000.42, '10-OCT-94', null);
```

If there are already some data in other tables, these data can be used for insertions into a new table. For this, we write a query whose result is a set of tuples to be inserted. Such an insert statement has the form

insert into <table> [(<column i, . . . , column j>)] <query>

Example: Suppose we have defined the following table:

create table OLDEMP (ENO number(4) not null, HDATE date);

We now can use the table EMP to insert tuples into this new relation:

```
insert into OLDEMP (ENO, HDATE)
        select EMPNO, HIREDATE from EMP
        where HIREDATE < '31-DEC-60';
```

Updates : For modifying attribute values of (some) tuples in a table, we use the update statement:

update <table> set
<column i> = <expression i>, . . . , <column j> = <expression j>
[where <condition>];

An expression consists of a constant (new value), an arithmetic or string operation, or an SQL query. Note that the new value to assign to <column i> must have matching data type.

An update statement **without a where clause** results in **changing respective attributes of all tuples** in the specified table. Typically, however, only a (small) portion of the table requires an update.

Examples:

The employee JONES is transferred to the department 20 as a manager and his salary is increased by 1000:

update EMP set JOB = 'MANAGER', DEPTNO = 20, SAL = SAL +1000
where ENAME = 'JONES';

All employees working in the departments 10 and 30 get a 15% salary increase.

update EMP set SAL = SAL * 1.15 where DEPTNO in (10,30);

Analogous to the insert statement, other tables can be used to retrieve data that are used as new values. In such a case we have a <query> instead of an <expression>.

Example: All salesmen working in the department 20 get the same salary as the manager who has the lowest salary among all managers.

update EMP set SAL = (select min(SAL) from EMP where JOB = 'MANAGER')
where JOB = 'SALESMAN' and DEPTNO = 20;

Explanation: The query retrieves the minimum salary of all managers. This value then is assigned to all salesmen working in department 20.

Give all employees over the age of 50 a raise a salary rise:

Convert to number SQl Function

UPDATE EMP SET SAL = SAL * 1.02 WHERE TO_NUMBER((SYSDATE - bdate) / 365) >= 50;

It is also possible to specify **a query that retrieves more than one field value in a record.**
In this case the set clause has the form **set(<column i, . . . , column j>) = <query>.**

It is important that the order of data types and values of the selected row exactly correspond to the list of columns in the set clause.

Delete : All or selected tuples can be deleted from a table using the delete command:

delete from <table> [where <condition>];

If the where clause is omitted, all tuples are deleted from the table. An alternative command for deleting all tuples from a table is the **truncate table <table> command**. However, in this case, the deletions cannot be undone.

Example:

Delete all projects (tuples) that have been finished before the actual date (system date)

delete from PROJECT where PEND < sysdate;

Remove all employees working in

```
DELETE employee  WHERE  employee.dno IN    (SELECT dept_locations.dnumber
                    FROM   dept_locations   WHERE  dlocation = 'HOUSTON');
```

DELETE will not be successful if a constraint would be violated.
For example, consider the DNO attribute in the Employee table as a Foreign Key.
Removing a department would then be contingent upon no employees working in that department.
This is what we call enforcing *Referential Integrity.*

Oracle SQL Functions

The Oracle implementation of SQL provides a number of functions that can be used in SELECT statements. Functions are typically grouped into the following:

- **Single row functions** - Operate on column values for each row returned by a query.
- **Group functions** - Operate on a collection (group) of rows.

The following is an overview and brief description of **single row functions.**
x is some number, *s* is a string of characters and *c* is a single character.

- **Math functions include:**

  ABS (*x*) - Absolute Value of *x*
  CEIL (*x*) - Smallest integer greater than or equal to *x*.
  COS (*x*) - Cosine of *x*
  FLOOR (*x*) - Largest integer less than or equal to *x*.
  LOG (*x*) - Log of *x*
  LN (*x*) - Natural Log of *x*
  SIN (*x*) - Sine of *x*
  ROUND (*x*, *n*) - Round *x* to *n* decimal places to the right of the decimal point.
  TAN (*x*) - Tangent of *x*
  TRUNC (*x*, *n*) - Truncate *x* to *n* decimal places to the right of the decimal point.

- **Character functions include:**

  CHR (*x*) - Character for ASCII value *x*.
  INITCAP (*s*) - String *s* with the first letter of each word capitalized.
  LOWER (*s*) - Converts string *s* to all lower case letters.
  LPAD (*s*, *x*) - Pads string *s* with *x* spaces to the left.
  LTRIM (*s*) - Removes leading spaces from *s*.
  REPLACE (*s1*, *s2*, *s3*) - Replace occurrences of *s1* with *s2* in string *s*.
  RPAD (*s*, *x*) - Pads string *s* with *x* spaces to the right.
  RTRIM (*s*) - Removes trailing spaces from *s*.
  SUBSTR (*s*, *x1*, [*x2]*) - Return a portion of string *s* starting at position *x1* and ending with position *x2*. If *x2* is omitted, it's value defaults to the end of *s*.
  UPPER (*s*) - Converts string *s* to all upper case letters.

- **Character functions that return numbers include:**
  ASCII (*c*) - Returns the ASCII value of *c*
  INSTR (*s1*, *s2*, *x*) - Returns the position of *s2* in *s1* where the search starts at position *x*.
  LENGTH (*s*) - Length of *s*

- **Conversion functions include:**

  TO_CHAR (*date*, *format*) - Converts a date column to a string of characters.
  format is a set of Date formatting codes where:

YYYY	is a 4 digit year.	D	is the day of the week.
NM	is a month number.	DAY	is the name of the day.
MONTH	is the full name of the month.	HH	is the hour of the day (12 hour clock)
MON	is the abbreviated month.	HH24	is the hour of the day (24 hour clock)
DDD	is the day of the year.	MI	is the minutes.
DD	is the day of the month.	SS	is the seconds.

  TO_CHAR (number, format) - Converts a numeric column to a string of characters.
  format is a set of number formatting codes where:

Element	Description	Example	Result
9	Indicates a digit position. Blank if position value is 0. No of 9 determines the display width	99999	1234
0	Display leading zero Displays a leading (floating) currency indicator.	099999	001234
$	Floating local currency symbol	$999999	$1234
.	Decimal point specified		
L	Prints a thousand indicator	L999999	FF1234

.	Minus sign to right for negative number	999999.99	1234.00
,	Parenthesize negative number	999,999	1,234
MI	Scientific notation	999999MI	1234-
PR	Multiply by 10 n times (n = number of 9s after V)	999999PR	(1234)
EEEE	Display zero value as blank not 0	99.9999EEEE	1.234E+03
V		9999V99	123400
B		B999999.99	001234.00

SELECT TO_CHAR(salary, '$99,999.00') salary

TO_DATE (s, format) - Converts a character column (string s to a date. format is a set of Date formatting codes as above).

TO_NUMBER (s, format) - Converts a character column (string s to a Number. format is a set of Number formatting codes as above.

Some Guidelines on writing format string

- Format must be enclosed with single quotation marks and is **case sensitive**
- The names of days and months in the output are automatically padded with blanks
- To remove padded blanks or to suppress leading zero, use the fill mode **fm** element

Example :

1. **'HH24 : MI : SS AM'** → output : 15:45:32 PM
2. 'DD **"of"** MONTH' → output : 12 of OCTOBER (Double quoted string is reproduces in the result as is)
3. SELECT last_name , TO_CHAR(hire_date, 'fmDD MONTH YYYY') Hire_date

Suppress leading zero

Hire_Date → 21 May 1991 , 7 February 1999 (leading 0 is suppressed)
(blank padding is removed from the result)

- Date functions include: **SYSDATE** - Returns the current date from the system clock.
- Some **additional function** are:

1. DECODE (*s*, *search1*, *result1*, [*search2*, *result2*] ……[, d])

- Compares first argument *s* with *search1*, *search2*, etc. and returns the corresponding *result* when there is a match.

d - d is an expression to return when s does not equal to any searched value s1, s2, .. sn.

Example : DECODE(3, 1, 'One', 2, 'Two', 'Not one or two') this query returns Not one or two

```
SELECT   DECODE(country_id, 'US','United States', 'UK', 'United Kingdom', 'JP','Japan'
              , 'CA', 'Canada', 'CH','Switzerland', 'IT', 'Italy', country_id) country ,  COUNT(*)
FROM   locations
GROUP BY   country_id
HAVING  COUNT(*) > =2
ORDER BY   country_id;
```

COUNTRY_ID	COUNT(*)
CA	2
CH	2
IT	2
JP	2
UK	3
US	4

The following query uses the DECODE() function in the ORDER BY clause to sort the employees result set based on an input argument:

```
SELECT   first_name,  last_name,  job_title
FROM  employees
ORDER BY
DECODE('J', 'F', first_name, 'L', last_name, 'J', job_title);
```

We sorted the employee list by job title because we passed the character J as the first argument of the DECODE() function.

The DECODE function can be used to provide a **variety of lookup values**. In the following example, the string concatenation operator || is used to put together a sentence about each employee. The DECODE command takes the COUNT of dependents as the first argument. Then, depending on the COUNT for a given employee, it returns an appropriate ending to the sentence.

```
SELECT   fname || ' '|| lname || ' has ' ||
      DECODE(COUNT(essn),
           0, 'no dependents.',
           1, 'one dependent.',
           2, 'two dependents.',
           3, 'three dependents.') AS Sentence
FROM    employee, dependent
WHERE   employee.ssn = dependent.essn (+)
GROUP BY employee.fname, lname;
```

Output

```
SENTENCE
---------------------------------------
AHMAD JABBAR has no dependents.
ALICIA ZELAYA has no dependents.
FRANKLIN WONG has three dependents.
JAMES BORG has no dependents.
JENNIFER WALLACE has one dependent.
JOHN SMITH has three dependents.
JOYCE ENGLISH has no dependents.
RAMESH NARAYAN has no dependents.
```

2. **NVL (s, expression) - If s is NULL, return expression. If s is not null, then return s.**

```
SELECT  last_name,  NVL(TO_CHR(manager_id), 'No Manager')
FROM  emp  WHERE manager_id  IS NULL
```

3. **NVL2(expr1, expr2, expr3)** - if expr1 is not NULL returns expr2, if expr1 is NULL returns expr3. expr1 can have any data type

4. **USER - Returns the username of the current user.**

5. **COALESCE** → Return the first non-null expression in the expression list
   COALESCE (expr1, expr2, ... expr3)

6. **NULLIF (expr1, expr2)** → compares two expression. If they are equal, the function returns null. If they are not equal, the function returns the first expression.

7. **Text, dates and numbers can be combined using the various conversion functions. In the following example, the TO_CHAR function is used to convert the date BDATE into a character string.**

```
SELECT 'The oldest employee was born on ' || TO_CHAR( MIN(bdate), 'DD/MM/YY')
        AS Sentence   FROM   employee;

SENTENCE
----------------------------------------
The oldest employee was born on 10/11/27
```

   In following example Date math results in a numerical answer that must be converted to characters to concatenate with other character strings as in this next example:

```
SELECT 'The oldest employee was born on ' ||   TO_CHAR( MIN(bdate), 'DD/MM/YY')
        || ' and is now' ||  TO_CHAR( (SYSDATE - MIN(bdate)) / 365, '99')
        || ' years old.'   AS Sentence
FROM   employee;


SENTENCE
-----------------------------------------------------------------
The oldest employee was born on 10/11/27 and is now 70 years old.
```

8. The following is an overview and brief description of **multiple row (group) functions.** *col* is the name of a table column (or expression) of type NUMBER.

   - **AVG (*col*)** - Returns the average of a group of rows for *col*
   - **MAX (*col*)** - Returns the maximum of a group of rows for *col*
   - **MIN (*col*)** - Returns the minimum of a group of rows for *col*
   - **STDEV (*col*)** - Returns the standard deviation of a group of rows for *col*
   - **SUM (*col*)** - Returns the sum (total) of a group of rows for *col*
   - **VARIANCE (*col*)** - Returns the variance of a group of rows for *col*

In addition the COUNT group function counts instances of values. These values can be any type (CHAR, DATE or NUMBER):

COUNT (*columns*) - Returns the number of instances of a group of rows for (*columns*)
To use an aggregate function, a GROUP BY clause must be added to the SELECT statement.

Date Functions

1. Add_Months(dateVal, n) → dateVal is the date field, n is no. of months.
   For subtracting month use -n

   SELECT TO_CHAR(Add_Months(hire_date, 1) , 'DD-MM-YYYY') NextMonth
   FROM EMP WHERE last_name = 'Das'

   If hire_date = 7/1/2010 Result NextMonth = 7-Feb-2010, it is same day as the day of hire_date (7) as last day of Feb (28) is more than that day (7)

   If hire_date = 31/1/2010 Result NextMonth = 28-Feb-2010, it is last day of Feb , as the day of hire_date (31) is more than last day of Feb (28).

2. SYSDATE → Returns current date and time (date type)

   SELECT TO_CHAR (SYSDATE, 'MM-DD-YYYY H24:MI:SS) Cur_Date FROM DUAL;

   You cannot use SYSDATE function in CHECK constraint. You have to use trigger.

3. MONTS_BETWEEN (date1, date2) → returns no of months
   If date1 > date2 → result is +ve, If date1 < date2 → result is –ve

   If days of two dates are equal then it will return integer, otherwise result is fraction based on 31 days in a month

4. NEXT_DAY (date1, char) → Returns the date of first weekday named by char (can be abbreviated)

   Date of the next Tuesday after Feb 2, 2001 :
   SELECT Next_Day ('02-Feb-2001', 'TUESDAY') Nxt_day FROM DUAL → 06-Feb-2001

5. LAST_DAY (SYSDATE) → Last date of the month

6. Extract (YEAR/MONTH/WEEK/DAY/HOUR/MINUTE/TIMEZONE FROM Date1) : Return the value of a specified datetime field
   SELECT extract (year from sysdate) from dual Result → Extract

----
2003

Example : Display the date of the next Friday that is six months from the hire date. The resulting date should appear as Friday, Aug 13th , 1999

```
SELECT  TO_CHAR( next_day( add_months( hire_date, 6), 'Friday'),   'fmDay, Month  DDth,  YYYY')
          AS  'Next 6 Month Review'    FROM  emp
```

SubQuery

Sometime to retrieve appropriate result set using a query, we need to depend on the result of another query. The first query is known as **main query** and the second one is called **sub query (or inner query),** which is supplying a result to the main query for use. Using a subquery is equivalent to perform two sequential queries and using the result of the first query in the second query.

Subqueries are frequently used in WHERE Clause Expressions

So examples are discussed below:

This assumes the subquery returns only one tuple as a result. Typically used when aggregate functions are in the subquery.

```
SELECT name, grade
FROM   students
WHERE  grade =(  SELECT MAX(grade) FROM students );
```

Subqueries using the **IN** operator are used whenever the value of a column should be found in a *set* of values. The set of values can be explicitly listed (as in the first example) or they can be created on the fly using a subquery.

```
SELECT    employee.fname, department.dname FROM      employee, department
WHERE     employee.dno = department.dnumber
          AND     department.dname IN ('HEADQUARTERS', 'RESEARCH');
```

```
SELECT    employee.fname  FROM      employee
WHERE     employee.dno IN
        (SELECT dept_locations.dnumber
         FROM dept_locations
         WHERE dept_locations.dlocation = 'STAFFORD');
```

The subquery returns a set of tuples. The IN clause returns true when a tuple matches a member of the set.

Subqueries using EXISTS.

EXISTS will return TRUE if there is at least one row resulting from the subquery.

```
SELECT   fname, lname, salary
FROM     employee
```

```
WHERE   EXISTS  (SELECT  fname
                 FROM   EMPLOYEE e2  WHERE   e2.salary > employee.salary)
          AND EXISTS  (SELECT  fname   FROM    EMPLOYEE e3
                       WHERE   e3.salary < employee.salary);
```

```
FNAME    LNAME      SALARY
--------  --------    ---------
JOHN      SMITH      30000
FRANKLIN  WONG       40000
JENNIFER  WALLACE    43000
RAMESH    NARAYAN    38000
```

The above query shows all employees names and salaries where there is at least one person who makes more money (the first exists) and at least one person who makes less money (second exists).

Subqueries with NOT EXISTS.

NOT EXISTS will return TRUE if there are no rows returned by the subquery.

```
SELECT   fname, lname, salary
FROM     employee
WHERE    NOT EXISTS  (SELECT  fname  FROM    EMPLOYEE e2
                      WHERE   e2.salary > employee.salary);
```

This shows the highest paid employee.

```
FNAME    LNAME      SALARY
-------- -------- ---------
JAMES    BORG       55000
```

Some rules for subquery

- Subquery enclose by ()
- Place subquery on the right side of comparison
- Order By clause is permitted from 8i. It is not needed unless you are performing top-n analysis.

Types of Subquery

- Single Row Subquery – Return only one row as result from inner query
- Multiple Row Subquery - Return more than one row as result from inner query
- Multi Column Subquery - Return multiple column as result from inner query

Example :

```
SELECT  deptID,  MIN(salary)
FROM  empMaster
GROUP BY  deptID
HAVAING  MIN(salary)  > (  Select  MIN(salary) from  empMaster
                           Where  deptID = 50 )
```

Oracle will execute the inner query first and the result will be used by the outer one

```
SELECT   empID,  lastName                  It is wrong because subquery will return multiple rows
FROM empMaster                             It is single row operator with multiple row subquery
WHERE Salary =  ( Select  MIN(Salary)  FROM  empMaster
                  GROUP BY  deptID  )
```

Use of Multi-row operators IN, ANY, ALL

```
SELECT   empID,  lastName                  To correct it use multiple row operator IN
FROM empMaster
WHERE Salary IN  ( Select  MIN(Salary)  FROM  empMaster
                  GROUP BY  deptID  )
```

Employees who are not IT programmer and whose salary is less than that of any IT programmer. The maxi. Salary that a programmer earns is 9000.

```
SELECT   empID,  lastName , jobID, salary
FROM empMaster
WHERE Salary < ANY  ( Select  Salary  FROM  empMaster
                      WHERE jobID = 'IT_PROG' )
      AND  jobID <>  'IT_PROG'
```

Suppose subquery returns values 9000, 6000, 4200

(It will return all the employees whose salary < 9000 and jobID <> 'IT_PROG')

- ANY operator (its synoname SOME) compares a value to each value returned by the subquery
- < ANY means less than the maximum, > ANY means more than the minimum
- = ANY is equivalent to IN
- < ALL means less than the maximum, > ALL means more than the minimum
- NOT operator can be used with IN, ANY and ALL operators.

```
SELECT   empID,  lastName , jobID, salary
FROM empMaster
WHERE Salary < ALL  ( Select  Salary  FROM  empMaster  WHERE jobID = 'IT_PROG' )
      AND  jobID <>  'IT_PROG'
```

(It will return all the employees whose salary < 4200 and jobID <> 'IT_PROG')

Use Subquery in DML

Update rows in a table based on values from another table

```
UPDATE  copyEmp  SET  deptID  =  ( SELECT deptID FROM  empMAster  WHERE empID = 100)
```

```
WHERE    jobID =  ( SELECT  jobID FROM empMaster  WHERE empID = 200 )
```

Delete rows from a table based on values from another table

```
DELETE  FROM empMAster
WHERE  deptID  =  ( SELECT deptID FROM  dept  WHERE deptName = 'Admin' )
```

Using Subquery in an Insert Statement

```
INSERT  INTO    ( SELECT  empID, lastName,  email, hireDAte,  jobID, salary,  deptID
                  FROM empMaster
                  WHERE  deprID = 50  )
VALUES  ( 9999, 'Sarakar', 'sarkar@yahoo.co.in', '01-Apr-10',  'IT_PROG',  9000,  50  )
```

Subquery is used in place of table name in the INTO clause. The select list of the subquery must have the same number of columns as the column list of the values clause. Any rules on the columns of the base table must be followed in order for the INSERT to work successfully.

- In the following example a subquery is used to identify the table and columns of the DML statement
- The WITH CHECK OPTION keyword prohibits you from changing rows that are not in the subquery

```
INSERT  INTO    ( SELECT  empID, lastName,  email, hireDAte,  jobID, salary
                  FROM empMaster
                  WHERE  deprID = 50   WITH CHECK OPTION)
VALUES  ( 9999, 'Sarakar', 'sarkar@yahoo.co.in', '01-Apr-10',  'IT_PROG',  9000  )
```

It is also used in views

It will give error

WITH CHECK OPTION indicates if the subquery is used in place of a table in an INSERT, UPDATE or DELETE statement, no changes that would produce rows that are not included in the subquery are permitted to that table.

In the above example subquery identifies rows that are in the department 50, but the deptID is not in the SELECT list, and a value is not provided for it in the VALUES list. Inserting this row would result in a department ID of null, which is not in the subquery. So it will give error.

Nested Subquery

If a Subquery contains another subquery, then the subquery inside another subquery is called nested subquery.

Example :

Student Table

	Studentid	Firstname	Lastname	Email
1	1	Atul	Bajaj	atul@abc.com
2	2	Vivek	Johari	vivek@abc.com
3	3	Ankur	Johari	ankur@abc.com
4	4	Tarveen	Kaur	Tarveen@abc.com

Course Table

	Courseid	Coursename	CourseAdminid
1	1	Oracle	2
2	2	Automation	4
3	3	Java	2
4	4	QTP	4

Let us suppose we have another table called "**StudentCourse**" which contains the information, which student is connected to which Course. The structure of the table is:-

```
create table StudentCourse( StudentCourseid int identity(1,1), Studentid int, Courseid int)
```

```
Insert into StudentCourse values(1,3)
Insert into StudentCourse values(2,1)
Insert into StudentCourse values(3,2)
Insert into StudentCourse values(4,4)
```

Note: **-** We don't need to insert data for the column StudentCourseid since it is an identity column.

Now, if we want to get the list of all the student which belong to the course "Oracle", then the query will be,

```
select Firstname, lastname from student
      where studentid in (select studentid from studentcourse
                  where courseid in (select courseid from course where coursename='Oracle'))
```

Result

	Firstname	lastname
1	Vivek	Johari

In this example we use the nested subquery since the subquery "select courseid from course where coursename='Oracle'" is itself contained in the another subquery(Parent Subquery) "select studentid from studentcourse where courseid in (select courseid from course where coursename='Oracle')".

Correlated Sub Query

- In a regular subquery the outer query depends on values provided by the inner query
- A correlated subquery is one where the inner query depends on values provided by the outer query.
- This means that in a correlated subquery, the inner query is executed repeatedly, once for each row that might be selected by the outer query.

Correlated subqueries can produce result tables that answer complex management questions.

The subquery in below SELECT statement cannot be resolved independently of the main query. Notice that the outer query specifies that rows are selected from the employee table with an alias name of E. The inner query compares the employee department number column (department_id) of the employee table with alias E to the same column for the alias table name T.

```
SELECT EMPLOYEE_ID, salary, department_id
FROM   employees E
WHERE salary > (SELECT AVG(salary) FROM   employees T
              WHERE E.department_id = T.department_id)
```

A correlated subquery is also known as a **repeating subquery** or a **synchronized subquery**.

- **Correlated subquery in the WHERE clause**

The following query finds all products whose list price is above average for their category.

```
SELECT   product_id,  product_name,   list_price
FROM    products p
WHERE    list_price > (  SELECT  AVG( list_price )  FROM  products
                         WHERE   category_id = p.category_id  );
```

For each product from the products table, execute the correlated subquery to calculate the average price by category and then use it in outer query.

- **Correlated subquery in the SELECT clause**

The following query returns all products and the average standard cost based on the product category:

```
SELECT    product_id,  product_name,  standard_cost,
   ROUND(  ( SELECT  AVG( standard_cost )    FROM products
              WHERE   category_id = p.category_id
      ),   2 ) avg_standard_cost
FROM    products p
ORDER BY   product_name;
```

- For each product from the products table, the correlated subquery to calculate the average standard of cost for the product category.
- ROUND() function is used to round the average standard cost to two decimals.

Multiple Column Sub Query

A multiple-column subquery returns more than one column to the outer query and can be listed in the outer query's FROM, WHERE, or HAVING clause. For example, the below query shows the employee's historical details for the ones whose current salary is in range of 1000 and 2000 and working in department 10 or 20.

```
SELECT first_name, job_id, salary
FROM emp_history
WHERE (salary, department_id) in (SELECT salary, department_id    FROM employees
                                  WHERE salary BETWEEN 1000 and 2000
                                        AND department_id BETWEEN 10 and 20)
ORDER BY first_name;
```

- When a multiple-column subquery is used in the outer query's FROM clause, it creates a temporary table that can be referenced by other clauses of the outer query.
- This temporary table is more formally called an inline view. The subquery's results are treated like any other table in the FROM clause. If the temporary table contains grouped data, the grouped subsets are treated as separate rows of data in a table. Consider the FROM clause in the below query. The inline view formed by the subquery is the data source for the main query.

```
SELECT *
FROM (SELECT salary, department_id
      FROM employees
      WHERE salary BETWEEN 1000 and 2000);
```