SQL: Structured Query Language
Nested Queries

One of the most important features of SQL is that SQL SELECT statements can be nested within each other to produce complex queries. While some of the queries discussed below can be written without nesting, some queries require nested structure.

Return Types

In general, a SELECT statement returns a relational table as a result. However, there are situations, where the returned table can be cast to a more simple structure. In general, the following four cases are possible. *We note, that in all four cases, it is always possible to treat the returned result as a relational table.*

Consider, for example, the following table:

`Univerersity(Id INT, NAME CHAR(60), City CHAR(15), State CHAR(2), NumStudents INT)`

The return type of a SELECT statement may be:

1. **An atomic value**. Returned when a single column of a relation is requested in the `SELECT` clause, and the information requested is unique in the table (e.g., a key value). For example, the following query

```
SELECT Id  FROM University
WHERE Name = 'University of Montana';
```

   returns a single number: the `Id` attribute for the University of Montana record.

2. **A single tuple**. Returned when the `SELECT` clause contains multiple attributes, but the information requested is unique in the table. For example, consider the following query:

```
SELECT *  FROM University
WHERE Id = 10;
```

   This query returns a single tuple, whose value for the primary key attribute is 10.

3. **A single column**. Returned when a single attribute is listed in the `SELECT` clause, and when the result contains more than one entry. This can be viewed as a set of atomic values. For example, the query below, returns the list of IDs for all universities in California.

```
SELECT Id  FROM University
WHERE State ='CA';
```

4. **A relational table**. This is the most general return type, used when the `SELECT` clause contains multiple columns, and the result of the query is multiple tuples. This return type can be viewed as a set of tuples. The following query returns as its result the list of all California campuses and their corresponding enrollments:

```
SELECT Name, NumStudents FROM University
WHERE State = 'CA';
```

Note: For some queries, the return type can be established statically, by analyzing constraints on the relational table. For some other queries, the actual return type of the query run on some instances can be different than the statically predicted return type (e.g., the last query will return a single tuple if only one record about a California university exists in the table).

Nested Queries in the FROM clause

Since the return of any SELECT statement can be viewed as a relational table, a nested SELECT statement can be used in the `FROM` clause. This allows to construct a temporary table which is useful for the purposes of the query, but does not need to be made persistent. Any attribute of the table resulting from the nested SELECT statement can be accessed from within the `WHERE` , `GROUP BY`, `HAVING` and `SELECT` clauses of the main query, just like attributes of regular relational tables.

Syntax:

```
SELECT <select-list>
FROM (SELECT query) <Alias>, ...
[WHERE <cond> ]
[GROUP BY <Attlist>
[HAVING <cond> ] ]
```

Two rules must be followed:

- The nested SELECT statement must be enclosed in partentheses.
- The result of the nested SELECT statement must be given a table alias and all aggregate/expression attributes in the result of the nested SELECT statement that are to be referenced in the outer query must be given column aliases.

Example. Consider the following tables (of university courses, students and student enrollments in courses):

```
Course(Major, Num, Name, Room, Instructor, Capacity)
Student(SSN, FirstName, LastName, Major, Year)
Enrollments(CourseMajor, CourseNum,StudentId)
```

Consider the following query.

```
SELECT C.Major, C.Num, Room
FROM Course C, (SELECT E.CourseMajor, E.CourseNum, COUNT(*) AS Enrolled
                FROM Student S, Enrollments E
                WHERE S.Year > 3 AND E.StudentId = S.SSN
                GROUP BY E.CourseMajor, E.CourseNum )  S
WHERE C.Major = S.CourseMajor AND C.Num = S.CourseNumber AND
      S.Enrolled >= Capacity/2;
```

The nested query:

```
SELECT E.CourseMajor, E.CourseNum, COUNT(*) AS Enrolled
FROM Student S, Enrollments E
WHERE S.Year > 3 AND E.StudentId = S.SSN
GROUP BY E.CourseMajor, E.CourseNum
```

Finds all seniors (`Year>3`) and joins their information with the information about the courses they are taking. This information is then grouped by course id/number, resulting in the relation that reports a list of courses and the number of senior students enrolled in each [1].

This information is reported back as a three-attribute relation with alias `S` to the main query, which, in turn, finds and reports the courses and the rooms of the courses, in which senior students occupy more than one half of the designated capacity.

The same query can be rewritten as

```
SELECT C.Major, C.Num, Room
FROM Course C, Student S, Enrollments E
WHERE S.Year > 3 AND E.StudentId = S.SSN AND
      C.Major = E.CourseMajor AND C.Num = E.CourseNumber
GROUP BY C.Major, E.CNum, Capacity
HAVING    COUNT(*) >= Capacity/2;
```

Note, that in this query, `Capacity` has to be included in the `GROUP BY` clause, to allow for its use in the `HAVING` clause.

Nested Queries in the WHERE Clause

Nested queries that return atomic values

A SELECT statement that returns a single value can be used in any expression in the `WHERE` clause where the value of this type is expected.

This, in particular, can be used to write joins in nested form.

[1] For courses that have at least one senior student enrolled

Example Consider the following tables (a list of bank accounts and a list of transactions for the account: each transaction either adds money (`After > Before`) or removes money (`After < Before`):

```
Accounts(AcctNo, Owner, Type, DateOpened, Balance)
Transactions(AcctNo, TDate, TType, Before, After)
```

Let us also assume that we have a `UNIQUE(Owner, Type)` constraint on `Accounts`. We can write the query that finds all dates on which John Smith added money to his savings account as follows:

```
SELECT TDate
FROM Transactions T
WHERE T.Before < T.After AND
      AcctNo = (SELECT AcctNo FROM Accounts A
                WHERE A.Owner = 'John Smith' AND A.Type = 'Savings');
```

Return Value	Allowed comparisons
numeric	<, >, <=, >=, ! =, <>, =
strings	=, ! =, <>, LIKE
other	=, ! =, <>

Nested queries that return single tuples

SQL has a tuple (list) construct: (<Att>, <Att>,...,<Att>). When a SELECT statement returns a single tuple, tuples formed using this construct can be compared to the returned tuple.

Example Consider the following tables:

`People(Id, FirstName, LastName, Age, Gender)` `Interests(FName, LName, Interest)`

The following query returns all interests of the person who has the id number 1 in the `People` table.

```
SELECT i.Interest
FROM Interests i
WHERE (i.Fname, i.LName) = (SELECT FirstName, LastName
                            FROM People
                            WHERE id = 1);
```

Return Value	Allowed comparisons
tuple (list)	=, ! =, <>

Nested queries that return single columns

When a SELECT statement returns a single column as a result, the result can be treated as a set of atomic values. SQL allows for a number of comparison operations between atomic values and sets:

Operation	Explanation
EXISTS <SubQuery>	true if <SubQuery> returns a non-empty table
<AtomicValue> IN <SubQuery>	true if <AtomicValue> is one of the elements in the result of <SubQuery>
<AtomicValue> <Op> ALL (<SubQuery>)	<Op>$\in \{>,<,>=,<=,=,!=,<>\}$. universal comparison
<AtomicValue> <Op> ANY (<SubQuery>)	<Op>$\in \{>,<,>=,<=,=,!=,<>\}$. existential comparison

A universal comparison operation evaluates to true iff <AtomicValue> is in the required relationship (<Op>) with **every** member of the set returned by <Expression>.

An existential comparison operation evaluates to true iff <AtomicValue> is in the required relationship (<Op>) with **at least one** member of the set returned by <Expression>.

Examples. Consider the relational tables Accounts and Transactions defined above:

```
Accounts(AcctNo, Owner, Type, DateOpened, Balance)
Transactions(AcctNo, TDate, TType, Before, After)
```

The following query finds the account information for all accounts that had at least one $1000 deposit.

```
SELECT A.AcctNo, A.Owner, A.Type
FROM Accounts A
WHERE A.AcctNo IN (SELECT DISTINCT AcctNo FROM Transactions
                   WHERE After - Before = 1000 and TType='deposit');
```

This query finds the information about all accounts with the maximum balance.

```
 SELECT A.AcctNo, A.Owner, A.Balance
 FROM Accounts
 WHERE Balance >= ALL (SELECT Balance FROM Accounts);
```

Correlated Queries

In examples above, nested queries can be evaluated once for the entire "lifetime" of the full query. In some situations, though, the result of the inner query depends changes as the evaluation of the outer query proceeds. Such queries are called *correlated queries*.

Examples. The query below reports the highest balance for each type of the account.

```
SELECT DISTINCT A.Type, A.Balance
FROM Accounts A
WHERE A.Balance >= (SELECT Balance FROM Accounts AA
                    WHERE AA.Type = A.Type);
```

Note, that in this case, correlating subquery and the main query is unavoidable — the subquery has to select only the balances of accounts of the same type as the current account number being considered in the outer query.

The next query is one possible way of finding all accounts that had transactions on October 1, 2007.

```
SELECT A.AcctNo, A.Owner
FROM Accounts A
WHERE EXISTS (SELECT * FROM Transactions t
              WHERE t.TDate = '01-Oct-2007' AND t.AcctNo = A.AcctNo);
```

In this case the use of `EXISTS` operation naturally represents our intuition: we are not interested in the specific details of transactions that happened on October 1, 2007, rather, we are simply verifying that such transactions exist for each account.

Nested queries that return relational tables

SELECT statements that return multicolumn relational tables can also be used as subqueries. The table below shows operations which they can be used:

Expression	Allowed Operators
`EXISTS (<Subquery>)`	N/A
`<Tuple> IN (<SubQuery>)`	N/A
`<Tuple> <OP> ALL (<SubQuery>)`	$=, <>, ! =$
`<Tuple> <OP> ANY (<SubQuery>)`	$=, <>, ! =$

Examples. Consider the `Accounts` and `Transactions` tables discussed above.

```
Accounts(AcctNo, Owner, Type, DateOpened, Balance)
Transactions(AcctNo, TDate, TType, Before, After)
```

The following query finds all dates when transactions *like* (same account type, same amount) any transactions on the accounts owned by Bob Brown occured.

```
SELECT TDate
FROM  Transactions
WHERE  (TType, After - Before) = ANY (SELECT TType, After-Before
                                      FROM Transactions T, Accounts A
                                      WHERE A.AcctNo = T.AcctNo and A.Owner ='Bob Brown');
```

Other uses of subqueries

Subqueries in Set Operation statements

We note briefly that the set operations `UNION`, `INTERSECT` and `MINUS` (or `EXCEPT`) are instances of the use of SELECT subqueries.

Subqueries in CREATE TABLE statements

New tables can be created to store results of specific queries. The syntax is:

```
CREATE TABLE <Name> AS (<SubQuery>);
```

Example. The following query creates a report of account activity for the month of October. The table contains the account number, owner's name and the account type, and the total activity information.

```
CREATE TABLE OctoberActivity AS
    (SELECT A.AcctNo, A.Owner, A.Type, SUM(T.Change) AS Activity
     FROM  Accounts A, (SELECT AcctNo, After - Before AS Change
                        FROM Transactions
                        WHERE TDate >= '01-Oct-2007' and TDate<= '31-Oct-2007') T
     WHERE A.AcctNo = T.AcctNo
     GROUP BY A.AcctNo, A.Owner, A.Type);
```