Chapter 7
Advanced SQL

Objectives

Define terms
Write single and multiple table SQL queries
Write noncorrelated and correlated subqueries
Define and use three types of joins
Nested Queries (Subqueries)

Subquery:
- Placing an inner query (SELECT statement) inside an outer query
- Can be used in place of a table name, or an expression that returns a list, for operators that accept a list (IN, EXISTS, =ANY, etc.)

Subqueries can be:
- Noncorrelated—executed once for the entire outer query
- Correlated—executed once for each row returned by the outer query

Subquery Example
Show all customers who have placed an order

```
SELECT CustomerName
FROM Customer_T
WHERE CustomerID IN
(SELECT DISTINCT CustomerID
FROM Order_T);
```

The IN operator will test to see if the CUSTOMER_ID value of a row is included in the list returned from the subquery

Subquery is embedded in parentheses. In this case it returns a list that will be used in the WHERE clause of the outer query
Set Membership: Existential

Show all faculty who qualified to teach some course before 10/30/2010.

```
SELECT FacultyID, FacultyName
FROM Faculty_T
WHERE FacultyID IN
  ( SELECT DISTINCT FacultyID
    FROM Qualified_T
    WHERE DateQualified < '2010-10-30' );
```

Subquery returns the IDs of all faculty who qualified to teach some course before 10/30/2010

Existential: Easy! 😊

Set Membership: Universal

Show all faculty who got all their qualifications before 10/30/2010.

Universal: Hard! ☹️
Set Membership: Universal

Find the other faculty, who got some qualification after 10/30/2010.

SELECT FacultyName FROM Faculty_T
WHERE FacultyID IN ( SELECT DISTINCT FacultyID FROM Qualified_T
WHERE DateQualified >= '2010-10-30' );

Find all faculty who got all qualifications before 10/30/2010.

SELECT FacultyName FROM Faculty_T
WHERE FacultyID NOT IN ( SELECT DISTINCT FacultyID FROM Qualified_T
WHERE DateQualified >= '2010-10-30' );

Figure 7-7a Processing a noncorrelated subquery

A noncorrelated subquery processes completely before the outer query begins
Correlated vs. Noncorrelated Subqueries

Noncorrelated subqueries:
  - Do not depend on data from the outer query
  - Execute once for the entire outer query

Correlated subqueries:
  - Make use of data from the outer query
  - Execute once for each row of the outer query
  - Can use the EXISTS operator

Correlated Subquery Example

Show all courses offered in the first semester of 2008.

The EXISTS operator will return a TRUE value if the subquery resulted in a non-empty set, otherwise it returns a FALSE

```sql
SELECT DISTINCT CourseName FROM Course_T
WHERE EXISTS
  (SELECT * FROM Section_T
   WHERE CourseID = Course_T.CourseID
   AND Semester = 'I-2008');
```

The subquery is testing for a value that comes from the outer query

⇒ A correlated subquery always refers to an attribute from a table referenced in the outer query
Correlated Subquery Example

Show all courses offered in the first semester of 2008.

```sql
SELECT DISTINCT CourseName FROM Course_T
WHERE EXISTS
  (SELECT * FROM Section_T
   WHERE CourseID = Course_T.CourseID
   AND Semester = 'I-2008');
```

1. The first CourseID is selected from Course_T (ISM 3113).
2. The subquery is evaluated to see if any sections exist for semester ‘I-2008’. EXISTS is valued as TRUE and the CourseName is added to the results table.
3. The next CourseID is selected from Course_T (ISM 3112), etc.

NULLs in SQL

- Whenever we don’t have a value, we can use a NULL
- Can mean many things
  - Value does not exist
  - Value exists but is unknown
  - Value not applicable
- How does SQL cope with tables that have NULLs?
NULL Values

- If \( x = \text{NULL} \) then \( 4*(3-x)/7 \) is still \( \text{NULL} \)
- If \( x = \text{NULL} \) then \( x > 6 \) is \( \text{UNKNOWN} \)
- In SQL there are three boolean values:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>= 0</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>= 0.5</td>
</tr>
<tr>
<td>TRUE</td>
<td>= 1</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{C1 AND C2} &= \min(C1, C2) \\
\text{C1 OR C2} &= \max(C1, C2) \\
\text{NOT C1} &= 1 - C1
\end{align*}
\]

```
SELECT * 
FROM Person 
WHERE (age < 25) AND 
      (height > 6 OR weight > 190);
```

Rule in SQL: include only records that yields TRUE.

NULL Values

- Unexpected behavior:

```
SELECT * 
FROM Person 
WHERE (age < 25) OR (age >= 25);
```

Some Persons not included!

- Can test for NULL explicitly:

```
SELECT * 
FROM Person 
WHERE (age < 25) OR (age >= 25) OR (age IS NULL);
```
Quick Note: GROUP-BY vs. Subquery

- Find students who take \( \geq 2 \) classes
- Attempt 1 (with subqueries):

```sql
SELECT StudentName
FROM Student_T
WHERE (SELECT COUNT(*) FROM
    (SELECT R.StudentID
     FROM Registration_T AS R, Student_T AS S
     WHERE S.StudentID = R.StudentID) AS CT
    ) \( \geq 2 \);
```

This is SQL by a novice

Quick Note (contd.)

- Find students who take \( \geq 2 \) classes
- Attempt 2 (with GROUP BY):

```sql
SELECT StudentName
FROM Student_T AS S, Registration_T AS R
WHERE S.StudentID = R.StudentID
GROUP BY S.StudentID
HAVING COUNT(R.SectionNo) \( \geq 2 \);
```

This is SQL by an expert
Union Queries

Combine the output (union of multiple queries) together into a single result table.

**First query**

```sql
SELECT C1.CustomerID, CustomerName, OrderedQuantity, ‘Largest Quantity’ AS Quantity
FROM Customer_T C1, Order_T O1, OrderLine_T Q1
AND O1.OrderID = Q1.OrderID
AND OrderedQuantity = (SELECT MAX(OrderedQuantity)
                        FROM OrderLine_T)
```

**Second query**

```sql
SELECT C1.CustomerID, CustomerName, OrderedQuantity, ‘Smallest Quantity’
FROM Customer_T C1, Order_T O1, OrderLine_T Q1
AND O1.OrderID = Q1.OrderID
AND OrderedQuantity = (SELECT MIN(OrderedQuantity)
                        FROM OrderLine_T)
ORDER BY 3;
```

**Figure 7-8 Combining queries using UNION**

1. In the above query, the subquery is processed first and an intermediate results table created. It contains the maximum quantity ordered from OrderLine_T and has a value of 10.
2. Next the main query selects customer information for the customer or customers who ordered 10 of any item. Contemporary Casuals had ordered 10 of some unspecified item.

**Third query**

```sql
SELECT C1.CustomerID, CustomerName, OrderedQuantity, ‘Smallest Quantity’
FROM Customer_T C1, Order_T O1, OrderLine_T Q1
AND O1.OrderID = Q1.OrderID
AND OrderedQuantity = (SELECT MIN(OrderedQuantity)
                        FROM OrderLine_T)
ORDER BY 3;
```

1. In the second main query, the same process is followed but the result returned is for the minimum order quantity.
2. The results of the two queries are joined together using the UNION command.
3. The results are then ordered according to the value in OrderedQuantity. The default is ascending value, so the orders with the smallest quantity, 1, are listed first.

Note: with UNION queries, the quantity and data types of the attributes in the SELECT clauses of both queries must be identical.
Join – Cartesian product of two tables (which gets us all pairs of rows, with one coming from each table) followed by a selection process according to some predicate.

Implicit Join – two or more tables appear in the FROM clause, separated by commas; the join condition (that is, the predicate that decides which rows match) is added to the WHERE clause.

Implicit Join Example

For each student registered in the first semester of 2008, what is the student name and ID?

```
SELECT DISTINCT S.StudentID, StudentName
FROM Student_T AS S, Registration_T AS R
WHERE S.StudentID = R.StudentID;
```
Explicit Join

For each student registered in the first semester of 2008, what is the student name and ID?

SELECT DISTINCT S.StudentID, StudentName
FROM Student_T S JOIN Registration_T R ON (S.StudentID = R.StudentID);

NATURAL JOIN

Most join conditions involve the equality predicate. SQL supports the concept of a NATURAL JOIN, which is a join in which the join condition is created implicitly, by requiring equality of all fields with the same name in both tables.

SELECT DISTINCT StudentID, StudentName
FROM Student_T NATURAL JOIN Registration_T;

StudentID is a field common to both tables. Only registered students will be listed.
Natural Join and USING

Natural joins are too brittle. SQL also supports a much better JOIN ... USING syntax. Rather than writing ON and the join condition, we write USING and then a list of fields which must match.

```sql
SELECT StudentID,  StudentName
FROM Student_T JOIN Registration_T USING (StudentID);
```

Natural Join Flaw

Natural joins in SQL are *inner joins* (a.k.a. *equi-joins*), meaning that rows must have matching values to appear in the result table.

```sql
SELECT StudentID,  StudentName
FROM Student_T JOIN Registration_T USING (StudentID);
/* Students who never registered will be lost */
```

Many times we want all rows from a certain table appear in a join, even when there is no corresponding join on the other table.

This can be achieved with an OUTER JOIN.
Outer Joins

- Joins in which rows with no matching values in common columns are nonetheless included in the result table
  - **Left Outer Join**: Include the left tuple even if there is no match
  - **Right Outer Join**: Include the right tuple even if there is no match
  - **Full Outer Join**: Include both left and right tuple even if there is no match

Outer Join Example

List the student ID, name and registration information for each student. Include students even if not yet registered.

```sql
SELECT * FROM Student_T S LEFT OUTER JOIN Registration_T R
ON (S.StudentID = R.StudentID);
```

All students will appear in the results table.

Note: Same principles apply to more than two tables involved in a JOIN operation.
Multiple Table Join Example

Assemble all information necessary to create an invoice for order number 1006

```
FROM Customer T, Order T, OrderLine T, Product T
WHERE Order_T.CustomerID = Customer_T.CustomerID
    AND Order_T.OrderID = OrderLine_T.OrderID
    AND OrderLine_T.ProductID = Product_T.ProductID
    AND Order_T.OrderID = 1006;
```

Four tables involved in this join

Each pair of tables requires an equality-check condition in the WHERE clause, matching primary keys against foreign keys

---

Figure 7-4 Results from a four-table join (edited for readability)

From CUSTOMER_T table

<table>
<thead>
<tr>
<th>CUSTOMERID</th>
<th>CUSTOMERNAME</th>
<th>CUSTOMERADDRESS</th>
<th>CUSTOMERCITY</th>
<th>CUSTOMERSTATE</th>
<th>CUSTOMERPOSTALCODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Value Furniture</td>
<td>15145 S. W. 17th St.</td>
<td>Plano</td>
<td>TX</td>
<td>75094 7743</td>
</tr>
<tr>
<td>2</td>
<td>Value Furniture</td>
<td>15145 S. W. 17th St.</td>
<td>Plano</td>
<td>TX</td>
<td>75094 7743</td>
</tr>
<tr>
<td>2</td>
<td>Value Furniture</td>
<td>15145 S. W. 17th St.</td>
<td>Plano</td>
<td>TX</td>
<td>75094 7743</td>
</tr>
</tbody>
</table>

From ORDER_T table

<table>
<thead>
<tr>
<th>ORDERID</th>
<th>ORDERDATE</th>
<th>ORDERED QUANTITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1006</td>
<td>24-OCT-10</td>
<td>1</td>
</tr>
<tr>
<td>1006</td>
<td>24-OCT-10</td>
<td>2</td>
</tr>
<tr>
<td>1006</td>
<td>24-OCT-10</td>
<td>2</td>
</tr>
</tbody>
</table>

From PRODUCT_T table

<table>
<thead>
<tr>
<th>PRODUCTID</th>
<th>PRODUCTNAME</th>
<th>PRODUCTSTANDARDPRICE</th>
<th>QUANTITY*STANDARDPRICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Entertainment Center</td>
<td>650</td>
<td>650</td>
</tr>
<tr>
<td>2</td>
<td>Writer's Desk</td>
<td>325</td>
<td>650</td>
</tr>
<tr>
<td>3</td>
<td>Dining Table</td>
<td>800</td>
<td>1600</td>
</tr>
</tbody>
</table>
Self-Join Example

Query: What are the employee ID and name of each employee and the name of his or her supervisor (label the supervisor’s name Manager)?

```sql
SELECT E.EmployeeID, E.EmployeeName, M.EmployeeName AS Manager
FROM Employee_E, Employee_M
WHERE E.EmployeeSupervisor = M.EmployeeID;
```

Result:

<table>
<thead>
<tr>
<th>EMPLOYEEID</th>
<th>EMPLOYEENAME</th>
<th>MANAGER</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-44-347</td>
<td>Jim Jason</td>
<td>Robert Lewis</td>
</tr>
</tbody>
</table>

Self-joins are usually used on tables with unary relationships.