Complex SQL

Ron Poet

Sub Queries and Set Operations

- SQL supports the standard operations on sets:
- unions (keyword **UNION**)
  - include all the rows returned by either of two **sub-queries**
- intersections (keyword **INTERSECT**)
  - include all the rows returned by both of two sub-queries
- differences (keyword **EXCEPT** or, in Oracle, **MINUS**)
  - include all the rows returned by one sub-query except those returned by another sub-query
- A sub-query generates a results table which is used as part of bigger query.
- They are also called **nested** queries.

Example of UNION

- All people living in Glasgow, whether staff or students

```
    (SELECT ID, firstN, lastN, email
            FROM Student
            WHERE city = 'Glasgow'  )
UNION
    (SELECT ID, fname, lname, email
            FROM Staff
            WHERE city = 'Glasgow‘ );
```

- Note the two sub-queries in brackets.

Restrictions on Set Operators

- Note both sides of the **UNION** must have exactly the same columns
  - the same number of columns
  - each column must be of the same type
  - the names can differ, as they are in the example.
- The **`Select-From-Where`** statement can produce duplicate rows.
- The default for union, intersection, and difference is to remove duplicates.
- If we want to retain duplicates add the word **ALL** – e.g. **UNION ALL**

Efficiency

- Detecting and removing duplicate rows involves extra work, such as sorting the table.
- The **SELECT** statement normally works one row at a time, which is why it does not remove duplicates.
- When doing intersection or difference, it is most efficient to sort the tables first
- At that point we may as well eliminate the duplicates anyway

Example

Staff

ID	Fname	Lname
11	John	Donne
12	Andrew	Marvell
13	Ben	Johnson
14	Henry	Vaughan

Student

ID	FirstN	LastN
11	John	Wayne
12	Ward	Bond
13	Ben	Johnson
14	Harry	Carey, Jr.

Staff UNION Student

ID	Fname	Lname
11	John	Donne
12	Andrew	Marvell
13	Ben	Johnson
14	Henry	Vaughan
11	John	Wayne
12	Ward	Bond
14	Harry	Carey, Jr.

Staff MINUS Student

ID	Fname	Lname
11	John	Donne
12	Andrew	Marvell
14	Henry	Vaughan

Staff INTERSECT Student

ID	Fname	Lname
13	Ben	Johnson

Staff UNION ALL Student

ID	Fname	Lname
11	John	Donne
12	Andrew	Marvell
13	Ben	Johnson
14	Henry	Vaughan
11	John	Wayne
12	Ward	Bond
13	Ben	Johnson
14	Harry	Carey, Jr.

Cursor Variables

- Each element in the **FROM** clause introduces a **cursor variable** for use in the other clauses.
  - `FROM Staff, Student` introduces two variables, `Staff` and `Student`
  - They can be used in **SELECT** and **WHERE** to get at the values in the columns of a table, especially if the column names in the different tables are the same.
  - `WHERE Staff.name = Student.name`
  - We can use them for clarity even if the two column names are different.

Identifying Attributes Using the Table Name

- Using a cursor is essential if a column name is identical in both tables
- Suppose project name and department name are both in columns called **name**
- List the names of all projects in the research dept

```
SELECT Project.name
FROM  Project, Department
WHERE (Department.dNum  = Project.conDept)
AND (Department.name = 'Research')
```

Defining Our Own Cursor Names

- Useful for cursor names that are shorter than table names
- We can also think of these cursor names as aliases for the table names

```
SELECT P.name
FROM Project P, Department D
WHERE (D.dNum  = P.pDept) AND (D.name = 'Research')
```

- They are necessary if the same table is used more than once in a query
- Example, print out the staff numbers of all pairs of people who work on the same project:

```
SELECT W1.emp, W2.emp
FROM WorksOn W1, WorksOn W2
WHERE W1.wpNum = W2.wpNum
```

Nested Sub-Queries

- In the **WHERE** clause we can test whether a value is related to the result of a nested sub-query
- Find the names of all the projects that John Smith works on

```
SELECT pName FROM Project
WHERE pNum IN
(SELECT wPNum FROM WorksOn, Employee
WHERE nin= wnin AND  name = ‘John Smith' );
```

- The sub-query (in brackets) is evaluated as a set of project numbers, and then a test for inclusion is made
- Because sub-queries are only internal to the query and never seen, **they don’t contain duplicates and they cannot be ordered**

Operators Using Sub-queries

- There are four new operators that can appear in the **WHERE** clause to test a row against a table (usually the result of a sub-query):
  - *row* **IN** *table* returns true if that row is in the table
  - **EXISTS** *table* returns true if the table has at least one row
  - *row relationship* **ANY** *table* returns true if the row stands in the stated relationship (e.g. “>”) to **at least one** of the rows in the table
  - *row relationship* **ALL** *table* returns true if the row stands in the stated relationship (e.g. “>”) to **all** of the rows in the table
- **NOT** can be used before **IN**, **EXISTS**, **ANY** and **ALL**.

The Operator IN

- **IN** tests if a row on the left hand side is one of the rows in the table on the right hand side – usually this table is returned by a sub-query, e.g.

```
SELECT pName FROM Project WHERE pNum IN
(SELECT wPNum FROM WorksOn, Employee
WHERE NIn= wnin AND  name = ‘John Smith' );
```

- **IN** on its own is rarely valuable as the above is the same as

```
SELECT pName FROM Project, WorksOn, Employee
WHERE NIn= wNỊn AND  PNum = wPNum
AND name = ‘John Smith’;
```

- **NOT IN** is much more useful

Using NOT IN

```
SELECT pName FROM Project WHERE pNum IN
(SELECT wPNum FROM WorksOn, Employee
WHERE NIn= wnin AND  name <> ‘John Smith' );
```

- Does this return the projects that John Smith does not work on
  - NO It returns the projects which everyone else works on and this may include some that John Smith works on
- To return the projects that John Smith does not work on we **must** do

```
SELECT pName FROM Project WHERE pNum NOT IN
(SELECT wPNum FROM WorksOn, Employee
WHERE NIn= wnin AND  name = ‘John Smith' );
```

Using NOT IN (2)

- Using **NOT IN** asserts that the row is not one of those in the table on the right hand side, again usually the result returned by a sub-query
  - This is the **only** way of achieving some queries
  - But it is hard for us to deal with requesting negative information
- Employees **not** managed by John Smith is achieved by

  SELECT name FROM Employee WHERE mgrnin NOT IN

  (SELECT NIn FROM Employee WHERE name = 'John Smith');
- MINUS or EXCEPT can usually be used instead

  SELECT name FROM Employee WHERE mgrnin IN

  (Employee MINUS

  (SELECT * FROM Employee WHERE name = 'John Smith'));

The Operator EXISTS

- **EXISTS** tests a table to see if it is not empty
  - The table is almost always a sub-query result
- **NOT EXISTS** tests to see if a table is empty.
- Find the names of all employees with dependents

```
SELECT name FROM Employee
WHERE EXISTS
(SELECT  * FROM  Dependent
WHERE Employee.nin = Dependent.enin);
```

The Operators ANY

- Test a single value against a table with a single column using a comparison operator, e.g. “=“ or “<“
- Returns true if the comparison operator returns true for at least 1 row
- Find the names of employees that earn more than someone on Project5

```
SELECT name FROM Employee
WHERE salary >  ANY
( SELECT salary FROM WorksOn, Employee
WHERE  wPNum=5  and nin = wNin );
```

The Operator ALL

- The operator **ALL** is similar, except it returns true if the comparison operator returns true for every row
- Names of employees that earn more than **everyone** on Project 5, replace **ANY** with **ALL**

```
SELECT name FROM Employee
WHERE salary >  ALL
( SELECT salary FROM WorksOn, Employee
WHERE  wPNum=5  and nin = wNin );
```

Using Minus

- Give the name of employees who work on all Department 5's projects
- The strategy for solving these kinds of query is
    1) Find the values of all the primary keys in the related table – e.g. the project numbers of all of department 5’s projects
    2) Find the values of the foreign keys of the row to be tested – e.g. all the projects that this employee works on
    3) Return the row if result 1 includes all of result 2. In other words, result 1 – result 2 is empty.

Using Minus (2)

SELECT name FROM Employee E

WHERE NOT EXISTS

- *Find all the projects in dept 5*

((SELECT P.pNum FROM Project P WHERE P.pdNum = 5)

MINUS

- *Find all the projects that the Employee works on*

(SELECT WO.wpNum FROM WorksOn WO

WHERE E.nin = WO.wNin))

- Note that we refer to an outer table, Employee, in the inner queries.

Division by using only NOT EXISTS

- The query can also be achieved without using **MINUS**:

SELECT name FROM Employee E

WHERE NOT EXISTS

- *Find all the workson rows related to dept 5*

(SELECT * FROM WorksOn W1

WHERE W1.Pnum IN

SELECT PNumber FROM Project WHERE Dnum = 5)

AND NOT EXISTS

- *Find all the projects that the Employee works on*

(SELECT * FROM WorksOn W2

WHERE E.nin = W2.wNin AND W1.Pnum = W2.Pnum)

Dates & Times in Oracle

- Although there are functions in Oracle for entering and extracting date and time information, they are not straightforward.
- The default format is DD-MON-YY   e.g.01-JAN-01
  - Use four figures for dates in the last century
  - However, the output default is two figures!
- Most uses of dates require the use of functions to cast to and from strings:
- Example: to find employees born in 1985, use the To_Char function which takes two parameters
  - the attribute with a DATE domain
  - the format that the character output is required in

```
SELECT * FROM  Employee
                  WHERE TO_CHAR(DateOfBirth, 'YYYY') = '1985';
```

Date Examples

- Date comparison

```
SELECT *  FROM Account
     WHERE dateOpened < '01-Jan-1997';
```

- Add/subtract number of days from a date   (NB SYSDATE =‘today’)

```
SELECT  *  FROM Account
     WHERE dateOpened > SYSDATE - 180;
```

- Add/subtract number of months to a date

```
SELECT  *  FROM  Account
     WHERE bDate > Add_Months(SYSDATE, - 3);
```

- This command returns accounts opened on a particular day of week.

```
SELECT accountno,dateOpened,inBranch
     FROM  Account
        WHERE To_Char(dateOpened,'DY') = 'TUE'
```

To_Date and To_Char

- *To_Date* and *To_Char* cast dates to and from strings
- *To_Char* takes a date and time value and a format and turns the date into a string using the format
  - The format can decorate the string with sub-strings such as “AD”, “AM”, punctuation such as “:”
  - It also determines which parts of the date or time value is returned – days, months, hours, etc
  - and what format is used –e.g. 12 hour or 24-hour clock

```
SELECT TO_CHAR(mydate, 'DD-MON-YYYY HH24:MI:SSxFF')
01-DEC-1999 10:00:00
```

- To_Date is the converse, e.g.:

```
SELECT TO_DATE( 'January 15, 1989, 11:00 A.M.', 'Month dd, YYYY, HH:MI A.M.',
    'NLS_DATE_LANGUAGE = American')
```

Time Examples

- Times are perhaps easiest stored using 2 digit integers
- Otherwise use the DATE datatype. Insert the values by using the To_Date function which needs as parameters
  - the time as a string e.g. 17:30 and the format of this string e.g. ‘HH24:MI’

```
INSERT INTO Times  VALUES (TO_DATE('17:30','HH24:MI'));
```

- To get a time out again, use the TO_CHAR function to convert the time to a character format

```
SELECT  TO_CHAR(mytime, 'HH24') AS hr FROM  Times;
```

- Oracle will automatically convert this format to a number if is involved in an arithmetic expression

```
SELECT  (TO_CHAR(mytime, 'HH24')+ 2)  AS TwoLater FROM Times;
```