SQL Part 2

Set Operators : Union

A sub-query is a complete query used as part of bigger query

SQL supports the standard operations on sets:

- unions (keyword UNION) - include all the records returned by either of two **sub-queries**
- intersections (keyword INTERSECT) - include all the records returned by both of two sub-queries
- differences (keyword EXCEPT or, in Oracle, MINUS) - include all the records returned by one sub-query excluding those returned by another sub-query

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' );
```


Restrictions on Set Operators

Note both sides of the UNION must have exactly the same columns

i.e. the same number of columns
and each column must be of the same domain
the names can differ as above

Although the `Select-From-Where` statement uses bag semantics, the default for union, intersection, and difference is set semantics

- i.e. duplicates are removed. Why? **Efficiency**
- When doing a projection in relational algebra, it is easier to avoid eliminating duplicates
  - Just work tuple-at-a-time
    - Saves time of not comparing tuples as we generate them.
- When doing intersection or difference, it is most efficient to sort the relations first
  - At that point you may as well eliminate the duplicates anyway.

If you want to retain duplicates add the word ALL – e.g. UNION ALL

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.

The Semantics of Single Table Queries

Key Slide

To emphasise the meaning of the a single table, let's compare it to a Java method defined on a class with the same variables as columns in the table

– e.g.

```
SELECT dateofBirth, house, street
    FROM  Employee
        WHERE city = 'Glasgow';
```

```
Employee[] emps;  // i.e. an array of Employee objects
public void employeeDetails()
{  println( "dateofBirth" + "  " + "house"  + "  "  + "street" )      // heading
   for (int e=0; e<emps.length; e++)     // once through the loop for each emp
        if (emps[e].city == "Glasgow")
                println(emps[e]. dateofBirth + "  " + emps[e]. house  +
                        "  "  + emps[e]. street );
}
```

– i.e. we iterate through the set of tuples
  - for each one we evaluate the boolean expression in the where clause
  - if it returns TRUE, print the values specified in the SELECT clause expressions

Multi-Table Queries

Key Slide

List the names and salaries of all employees in the research dept

```
SELECT name, salary            // columns in answer – this is RA project
FROM Employee, Department      // Cartesian Project in RA
WHERE dept = dNum              // foreign keys – RA join
AND dName = 'Research';        // restriction on data - RA select
```

This style is usual in SQL. It uses the Cartesian Product followed by selection then projection in Relational Algebra

A recommended style is to have all the foreign keys restrictions first, nearest the list of tables

The Semantics of Multi-Table Queries

Again, let's compare it to a Java method defined on a class with the same variables as columns in the table

– e.g.
```
SELECT Ename, Pname  FROM  Employee, WorksOn, Project
  WHERE city = 'Glasgow' and Employee.Enumber = WorksOn.emp
                AND WorksOn.proj = Project.Pnumber;
```

– and assume we have a class with arrays Es, WOs and Ps holding three sets of objects equivalent to the three sets of records in the tables

```
public void employeeAndProjectNames()
{  println( "Ename" + "  " + "Pname")  // a heading for the output
   for (int e=0; e < Es.length; e++)
      for (int wo=0; wo < WOs.length; wo++)    // nested for loops
         for (int p=0; p < Ps.length; p++)
            if (Es[e].city = "Glasgow" and Es[e].Enumber = WOs[wo].emp
                  and WOs[wo].proj = Ps[p].Pnumber)
                        println( Es[e].Ename + "  " + Ps[p].Pname );
}
```


Tuple Cursor Variables

Key Slide

Each element in the FROM clause introduces a **cursor variable**, for use in the other clauses, which ranges over the tuples in its relation

– this is used in SELECT and WHERE to get at the values in the column of a relation

A tuple variable may be:

– **invisible** (as in the examples so far)
  • If the column names can't be confused, they can be used without identifying which relation they come from
    – e.g. there was no confusion about where the column *dName* was
– **implicit**
  • If there are two columns with the same name in different tables, they must be distinguished with a tuple variable and the table name can be used for this
    – e.g. *Department.dName* would work fine
– **explicit**
  • Either to shorten the code or to disambiguate two uses of the **same** table, we can introduce an explicit variable name:
    – e.g. given *FROM Department D* we can use *D.dName*

Example Identifying Attributes Using the Table Name

Using an implicit or explicit cursor is essential if a field 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                          // essential
FROM  Project, Department
   WHERE (Department.dNum  = Project.conDept)   // optional
      AND (Department.name = 'Research')        // essential
```

Your Own Cursor Names

Useful if you'd like cursors that are shorter than table names

You 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 Queries

Key Slide

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 ni#= wni# 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 Subqueries

Key Slide

There are then four new operators that can appear in the WHERE clause to test a tuple against a relation (usually the result of a sub-query):

- *tuple* IN *relation* returns true if that tuple is in the relation
- EXISTS *relation* returns true if the relation has at least one tuple
- *tuple relationship* ANY *relation* returns true if the tuple stands in the stated relationship (e.g. ">") to **at least one** of the tuples in the relation
- *tuple relationship* ALL *relation* returns true if the tuple stands in the stated relationship (e.g. ">") to **all** of the tuples in the relation

Again NOT can be used before these *<---- this is where it gets very hard*

There is also the use of arithmetic comparisons if the subquery returns a single value – i.e. it is a **scalar subquery**

- if the query returns just one value, we can think of the comparison as being of just two numbers although, formally, these are still a relation with one row and one column

The Operator IN

IN tests if a tuple on the left hand side is one of the tuples in the relation on the right hand side – usually this is returned by a subquery, e.g.

```
SELECT pName FROM Project
      WHERE pNum IN
            (SELECT wPNum FROM WorksOn, Employee
                  WHERE NI#= wni# AND  name = 'John Smith' );
```

IN on its own is rarely valuable as this is the same as:

```
SELECT pName FROM Project, WorksOn, Employee
   WHERE NI#= wNI# AND  PNum = wPNum
                  AND name = 'John Smith';
```

NOT IN is much more useful

Using NOT IN

Consider:

```
SELECT pName FROM Project
    WHERE pNum IN
        (SELECT wPNum FROM WorksOn, Employee
                WHERE NI#= wni# AND  name <> 'John Smith' );
```

Does this return the projects that John Smith does not work on

- Answer: NO It returns the projects which everyone else works on and this may include some that John Smith works on

To achieve 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 NI#= wni# AND  name = 'John Smith' );
```


Using NOT IN

Using NOT IN asserts that the tuple is not one of those in the relation on the right hand side, again usually the result returned by a subquery

- This is the **only** way of achieving some queries
- But it is very hard for us to deal with requesting negative information

For instance, asking for employees **not** managed by John Smith is achieved by:

```
SELECT name FROM Employee
    WHERE mgrni# NOT IN
      (SELECT NI# FROM Employee WHERE name = 'John Smith');
```

MINUS or EXCEPT can usually be used instead:

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


The Operator EXISTS

EXISTS tests a relation to see if it is empty

- The relation is almost always a subquery result

For instance to find the names of all employees with dependents

```
SELECT name FROM Employee
    WHERE EXISTS
          (SELECT  * FROM  Dependent
                WHERE Employee.ni# = Dependent.eni#);
```

Again NOT EXISTS is going to prove more useful (and harder to understand)

The Operators ANY, SOME and ALL

The operator ANY or SOME is used

- to test a single value against a single column relation using a comparison operator, e.g. "=" or "<"
- and returns true if the comparison operator returns true for at least 1 record,
- e.g. Find the names of employees that earn more than someone on Project 5

```
SELECT name FROM Employee
      WHERE salary >  ANY
            ( SELECT salary FROM WorksOn, Employee
                  WHERE  wPNum=5  and ni# = wNi# );
```

The operator ALL is similar, except it returns true if the comparison operator returns true for every record

- e.g. to 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 ni# = wNi# );
```

Division Queries

Queries which try to find tuples which relate to every record in another relation are called division queries

- because they are solved in the relational algebra by the division operator
- they are the most difficult to understand of all SQL queries
- example:
  - 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 relation – e.g. the project numbers of all of department 5's projects
2) Find the values of the foreign keys of the tuple to be tested – e.g. all the projects that this employee works on
3) Return the tuple if result 2 includes all of result 1

There are two ways of achieving this:

- although the deprecated CONTAINS operator would have made things much simpler

Division by using MINUS

Key Slide

The query is achieved by saying that there are no records in the target relation that are not related to the target tuple

For the example:

```
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.ni# = WO.wNi# )  )
```


Division by using only NOT EXISTS

The query is achieved by saying that there are no records in *WorksOn* where the project is controlled by department 5 and there is no matching record in *WorksOn* relating the same project and the employee being tested, i.e.

- For the example:

```
SELECT name FROM Employee E
  WHERE NOT EXISTS
            Find all the works on record 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.ni# = W2.wNi# AND W1.Pnum = W2.Pnum)
```

- This is as hard as it gets in SQL and is needed only if MINUS or EXCEPT are unavailable

Aggregate Functions

Key Slide

We can write

```
SELECT AVG (salary) FROM Employee
```

- which returns one value from a column in the table
- **SUM, MIN & MAX** are also available

COUNT is slightly different – it counts the records, e.g.

```
SELECT COUNT (*) from Employee
```

returns the number of records in the table, while

```
SELECT COUNT (salary) from Employee
```

returns the number of records in which salary is not NULL and

```
SELECT COUNT (distinct salary) from Employee
```

which returns the number of different salaries in the table

Group By and Having Clauses

Key Slide

```
SELECT AVG(salary), dept
FROM Employee
GROUP BY dept
```

produces the average salary for each department — one record for each

When using GROUP BY, we are, however, restricted in what is in the SELECT to those things known to be **single valued** within each group:

- aggregates or the columns explicitly named in GROUP BY
- even if we know that another column is unique in the group we cannot use it
  - e.g. if we put a column *Dname* in *Employee* as well then we can't use it even though we know that for each group *Dname* is unique
  - i.e. the following would throw up an error :

```
SELECT Dnum, Dname, COUNT(*), AVG(Salary)
    FROM Employee
        GROUP BY Dnum;
```

Appending `HAVING COUNT ( * ) > 2` restricts the query to departments with more than two employees.

- **HAVING** is like a **WHERE** for aggregate functions

The Order By Clause

Key Slide

```
SELECT * FROM Employee
    ORDER BY salary DESC, lName ASC
```

This returns the results ordered first in descending order of salary and then in alphabetical order of name.

- **ASC** is the default, so can be omitted

Ordering is useful for ALL queries which return more than a few lines

Note that ordering should **never** be used in a sub-query

- since the internal workings of a query are not ordered
- the only place where ordering matters is the part that involves the user
  - i.e. the outermost select

Scalar Subqueries

Key Slide

If only one tuple with one column is returned it can be treated like a constant

- This makes most sense if it also returns one component, e.g.

```
SELECT street FROM Employee WHERE Enumber = 1234;
```

or

```
SELECT MAX(salary) FROM Employee;
```

- The result can therefore be used in simple comparisons, e.g.

```
SELECT name FROM Employee WHERE street =
    (SELECT street FROM Employee WHERE Enumber = 1234);
```

gives the names of employees living on the same street as emp 1234

```
SELECT name FROM Employee
    WHERE salary = (SELECT MAX(salary) FROM Employee);
```

gives the name of the highest paid employee(s)

- Typically, a single tuple is guaranteed by the presence of key attributes or the use of aggregate functions
  - A run-time error occurs if there is no tuple or more than one tuple

Parameterised Queries

Many queries will be more versatile if you could change the value searched for at run-time

- these are the basis of "canned queries"

In Oracle, replace the literal with a prompt preceded by '&'

The prompt may often be the same as the attribute name, but doesn't have to be.

e.g. Alter which city you are interested in

```
SELECT * FROM Employee WHERE city = &name_of_city;
```

- When this query is run, the user is prompted to 'Enter name_of_city : '
- Running a parameterised query in the Query Analyser is slightly complex.
  - Execute the query
  - The prompt appears in the output panel
  - Type the parameter into the input panel, overwriting anything that is there already
  - Press 'Execute'
    - The results should appear in the output panel