Seminar 1. SQL Queries – DML Subset

SELECT statement

A very simple example of using SELECT statement is:

```
SELECT  *
FROM  Students S
WHERE  S.age = 21
```

which returns all 21 years old students form ***Students*** table:

To find just names and email addresses, we should replace the first line with:

```
SELECT S.name, S.email
```

A very simple SQL query looks like:

```
SELECT [DISTINCT]target-list
FROM relation-list
WHERE qualification
```

Where:

- ***relation-list*** is a list of relation names (possibly with a range-variable after each name);
- ***target-list*** is a list of attributes of relations in relation-list;
- ***qualification*** contains logical expressions having comparisons (Attr op const or Attr1 op Attr2, where op is one of $<, >, =, \leq, \geq, \neq$) combined with logical operators AND, OR and NOT;
- ***DISTINCT*** is an optional keyword indicating that the answer should not contain duplicates.   Default is that duplicates are not eliminated!

Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:

- Compute the cross-product of relation-list;
- Discard resulting tuples if they fail qualifications;
- Delete attributes that are not in target-list;
- If DISTINCT is specified, eliminate duplicate rows.

Range variables really needed only if the same relation appears twice in the FROM clause.

It is good style, however, to always use range variables.

So, we can write the same query in two distinct ways:

```
SELECT S.name, E.cid
FROM   Students S, Enrolled E
WHERE  S.sid=E.sid AND E.grade=10
```

or

```
SELECT name, cid
FROM   Students, Enrolled
WHERE  Students.sid=Enrolled.sid
       AND grade=10
```

The following query illustrates the use of arithmetic expressions and string pattern matching: *Find triples (of ages of students and two fields defined by expressions) for students whose names begin and end with B and contain at least three characters.*

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM   Students S
WHERE  S.name LIKE 'B_%B'
```

Observations:

- Note that AS and = are two ways to name fields in result.
- LIKE operator is used for string matching.
- `_' stands for any one character
- `%' stands for 0 or more arbitrary characters.

UNION: Can be used to compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries). Duplicate rows are eliminated

Example: *Find sid of students with grades at courses with 4 or 5 credits*

```
SELECT E.sid
FROM  Enrolled E, Courses C
WHERE E.cid=C.cid
  AND C.credits=4
UNION
SELECT E.sid
FROM  Enrolled E, Courses C
WHERE E.cid=C.cid
```

```
                  AND C.credits=5
```

INTERSECT: can be used to compute the intersection of any two union-compatible sets of tuples. Included in the SQL-92 standard, but some systems don't support it.

Example: *Find sid of students with grades at both a 4 credits course and a 5 credits course*

```
SELECT  E.sid
FROM  Courses C, Enrolled E
WHERE E.cid=C.cid
   AND C.credits=4
INTERSECT
SELECT  E.sid
FROM  Courses C, Enrolled E
WHERE E.cid=C.cid
   AND C.credits=5
```

Also available: ***EXCEPT*** statement, used to obtain all the records belonging to the first set of tuples which are not part of the second set of tuples.

Nested Queries

A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)

Sample: *Find names of students who're enrolled at course 'Alg1'*

```
SELECT S.name
FROM Students S
WHERE S.sid IN (SELECT  E.sid
           FROM  Enrolled E
           WHERE  E.cid='Alg1')
```

To understand semantics of nested queries, think of a nested loops evaluation: For each Students tuple, check the qualification by computing the subquery.

Sample: *Find names of students who're enrolled at course 'Alg1'*

```
SELECT S.name
FROM Students S
WHERE EXISTS (SELECT *
           FROM Enrolled E
           WHERE E.sid=S.sid
             AND E.cid=‘Alg1’)
```

EXISTS is another set comparison operator, like IN.

The above example illustrates why, in general, subquery must be re-computed for each *Students* tuple.

Besides IN and EXISTS, and we can also use NOT IN or NOT EXISTS. There are also available:

- *operator ANY* (the value is true if the condition is true for **at least one** item of sub-query result)
- *operator ALL*(the value is true if the condition is true for **all** items of sub-query result)

Sample: *Find students whose age is greater than that of some student called ‘Joe’*:

```
SELECT   *
FROM  Students S
WHERE  S.age > ANY (SELECT S2.age
                    FROM  Students S2
                    WHERE S2.name=‘Joe’)
```

Join Queries

Students

sid	name	email	age	gr
1234	John	j@cs.ro	21	331
1235	Smith	s@cs.ro	22	331
1236	Anne	a@cs.ro	21	332

Courses

cid	cname	credits
Alg1	Algorithms1	7
DB1	Databases1	6
DB2	Databases2	6

Enrolled

sid	cid	grade
1234	Alg1	9
1235	Alg1	10
1237	DB2	9

Figure 3.1. Sample tables

Join variant	Sample query	Result

`INNER JOIN`	`SELECT S.name, C.cname`
	`FROM Students S`
	`INNER JOIN Enrolled E ON S.sid = E.sid`
	`INNER JOIN Courses C ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1

`LEFT OUTER JOIN`	`SELECT S.name, C.cname`
	`FROM Students S`
	`LEFT OUTER JOIN Enrolled E`
	`ON S.sid = E.sid,`
	`LEFT OUTER JOIN Courses C`
	`ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1
		Anne | **NULL**

`RIGHT OUTER JOIN`	`SELECT S.name, C.cname`
	`FROM Students S`
	`RIGHT OUTER JOIN Enrolled E`
	`ON S.sid = E.sid,`
	`INNER JOIN Courses C`
	`ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1
		NULL | Databases2

`FULL OUTER JOIN`	`SELECT S.name, C.cname`
	`FROM Students S`
	`FULL OUTER JOIN Enrolled E`
	`ON S.sid = E.sid,`
	`FULL OUTER JOIN Courses C`
	`ON E.cid = C.cid`
		name | *cname*
		John | Algorithms1
		Smith | Algorithms1
		NULL | Databases2
		NULL | Databases1
		Anne | **NULL**

NULL values

Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *inapplicable*. SQL provides a special value <u>*null*</u> for such situations.

The presence of *null* complicates many issues. E.g.:

- Special operators needed to check if value is/is not *null*.
- Is *rating>8* true or false when *rating* is equal to *null*? What about AND, OR and NOT connectives?

Solution: we need a <u>3-valued logic</u> (**true**, **false** and ***unknown***). Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.). New operators (in particular *outer joins*) are possible/needed.

Aggregate Operators

Most used aggregate operators are (A is a table field name):

```
-   COUNT (*)
-   COUNT ( [DISTINCT] A)
-   SUM ( [DISTINCT] A)
-   AVG ( [DISTINCT] A)
-   MAX (A)
-   MIN (A)
```

Sample: *Get the total number of students*

```
SELECT  COUNT (*)
FROM  Students S
```

Sample: *Get age average of group 311*

```
SELECT  AVG (S.age)
FROM  Students S
WHERE  S.gr=311
```

Sample: *Find how many groups have assigned at least one student named Bob*

```
SELECT COUNT (DISTINCT S.gr)
FROM  Students S
WHERE S.name='Bob'
```

Sample: *Find the names of oldest students*

```
SELECT S.name
FROM Students S
WHERE S.age = ANY
      (SELECT MAX(S2.age)
       FROM  Students S2)
```

GROUP BY and HAVING

So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.

Consider: *Find the age of the youngest student for <u>each</u> group.*

- In general, we don't know how many groups exist
- Suppose we know that group values go from 110 to 119, we can write 10 similar queries. But when another group is added, a new query should be created.

Group By and *Having* clauses allow us to solve problems like this in only one SQL query. General syntax is:

```
SELECT [DISTINCT] target-list
FROM   relation-list
WHERE  qualification
GROUP BY  grouping-list
HAVING    group-qualification
```

The *target-list* contains

- <u>attribute names</u> (the <u>attribute names</u> must be a subset of *grouping-list*);
- terms with aggregate operations (e.g., MIN (*S.age*)).

Intuitively, each answer tuple corresponds to a *group*, and these attributes must have a single value per group. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Group By / *Having* conceptual evaluation:

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary'* fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- The *group-qualification* is then applied to eliminate some groups. Expressions in *group-qualification* must have a <u>*single value per group*</u>!
    - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Sample: *Find the age of the youngest student with age ≥ 20 for each group with at least 2 such students*

```
SELECT  S.gr,  MIN (S.age)
FROM  Students S
WHERE  S.age >= 20
GROUP BY  S.gr
HAVING  COUNT (*) > 1
```

- Only S.gr and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes `*unnecessary*'.
- 2nd column of result is unnamed. (Use AS to name it.)

Sample: *Find the number of enrolled students and the grade average for each course with 6 credits*

```
SELECT  C.cid,  COUNT (*) AS scount, AVG(grade)
```

```
FROM  Students S, Enrolled E, Courses C
WHERE  S.sid=E.sid AND E.cid=C.cid AND C.credits=6
GROUP BY  C.cid
```

Adding, Deleting and Updating Tuples

Insert a single tuple using:

```
INSERT INTO  Students (sid, name, email, age, gr)
VALUES  (53688, ‘Smith’, ‘smith@math’, 18, 311)
```

Delete all tuples satisfying some condition:

```
DELETE  FROM Students S
WHERE S.name = ‘Smith’
```

Modify the columns values using:

```
UPDATE Students S
SET S.age=S.age+1
WHERE S.sid = 53688
```