Seminar 2. SQL Queries – DML Subset

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
```

Insert a single record:

```
INSERT  [INTO]  table_name [(column_list)]
VALUES ( value_list)
```

Example:

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

Bulk insert:

```
INSERT [INTO] table_name [(column_list)]
<select statement>
```

Example:

```
INSERT INTO  Enrolled  (sid, cid, grade)
SELECT sid, 'BD1', 10
FROM Students
```

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
```

DDL Commands

The following command creates the *Students* table (relation). Observe that the type (domain) of each field (attribute) is specified, and enforced by the DBMS whenever tuples are added or modified.

```
CREATE TABLE Students
        (sid CHAR(20),
         name CHAR(50),
         email CHAR(30),
         age INTEGER,
         gr INTEGER)
```

As another example, the *Enrolled* table holds information about courses that students take.

```
CREATE TABLE Enrolled
        (sid CHAR(20),
         cid CHAR(5),
         grade REAL)
```

To destroy (remove) the relation *Students* the following command could be used. Of course, both schema information and the tuples are deleted.

```
DROP TABLE Students
```

Using the following command, the schema of *Students* table is altered by adding a new field; every tuple in the current instance is extended with a *null* value in the new field.

```
ALTER TABLE  Students
ADD COLUMN firstYear INTEGER
```

The schema of *Students* is altered by adding a new field; every tuple in the current instance is extended with a *null* value in the new field.

SQL could be used to declare many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.

Sample: *For a given student and course, there is a single grade*.

```
CREATE TABLE Enrolled
   (sid CHAR(20),
    cid  CHAR(20),
    grade CHAR(2),
PRIMARY KEY (sid,cid))
```

Sample: *Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade*. (this is an example about how to not define candidate keys; used carelessly, an IC can prevent the storage of database instances that arise in practice!)

```
CREATE TABLE Enrolled
   (sid CHAR(20),
    cid  CHAR(20),
    grade CHAR(2),
 PRIMARY KEY(sid),
 UNIQUE (cid, grade))
```

Sample of defining foreign keys "*Only students listed in the Students relation should be allowed to enroll for courses*".

```
CREATE TABLE Enrolled
   (sid CHAR(20),  cid CHAR(20),  grade CHAR(2),
     PRIMARY KEY  (sid,cid),
     FOREIGN KEY (sid) REFERENCES Students )
```

Referential Integrity

Starting with SQL-99 there is support for all 4 approaches preserving referential integrity in case of deletes and updates:

- NO ACTION (*delete/update is rejected*) – it is the default approach is
- CASCADE (also delete all tuples that refer to deleted tuple)
- SET NULL/SET DEFAULT (sets foreign key value of referencing tuple)

```
CREATE TABLE Enrolled
   (sid CHAR(20),
    cid CHAR(20),
    grade CHAR(2),
    PRIMARY KEY  (sid,cid),
    FOREIGN KEY (sid)
      REFERENCES Students
  ON DELETE CASCADE
  ON UPDATE SET NULL )
```

General constraints are useful when more general ICs than keys are involved:

```
CREATE TABLE Students
```

```
(sid CHAR(20),
 name CHAR(50),
 email CHAR(30),
 age INTEGER,
 gr INTEGER,
 PRIMARY KEY (sid),
 CONSTRAINT ageInterv
 CHECK (age >= 18
       AND age<=70))
```