Seminar 2. SQL Queries – DML + DDL 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.
  e.g., for *i* = 110,111,...,119:

```
SELECT MIN(S.age)
FROM   Students S
WHERE  S.gr = i
```

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* <u>conceptual evaluation:</u>

- 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 the 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
```

SQL was an important factor in the early acceptance of the relational model; more natural than earlier, procedural query languages.

SQL is relationally complete; in fact, it has a significantly more expressive power than relational algebra. Even queries that can be expressed in RA can often be expressed more naturally in SQL.

There are many alternative ways to write a query; the optimizer should look for the most efficient evaluation plan. In practice, users need to be aware of how queries are optimized and evaluated for best results.

NULL for unknown field values brings many complications.

DDL Commands

Creating Relations

Create the *Students* relation and the *Enrolled* table.

Observe that the type (domain) of each field 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)
```

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

Destroying/Altering Relations

```
DROP TABLE Students
```

Destroys the relation *Students*. The schema information *and* the tuples are deleted.

```
ALTER TABLE  Students

ADD 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.

Primary/Candidate Keys

Possibly many *candidate keys* (specified using UNIQUE), one of which is chosen as the *primary key*.

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

"*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*."

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

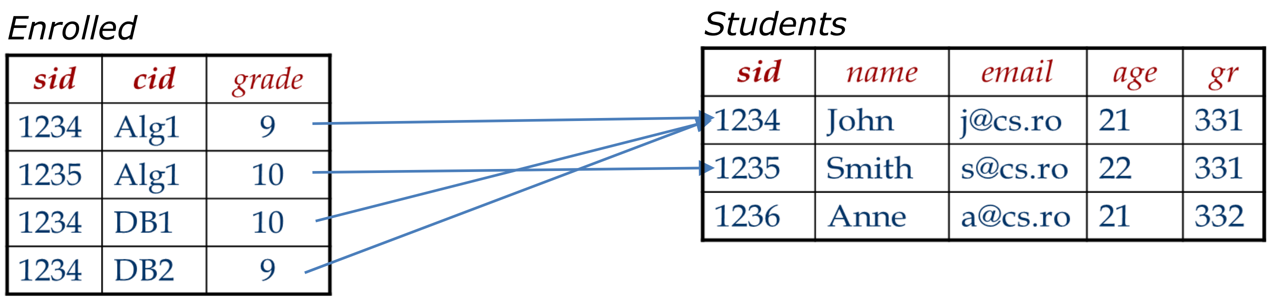
```
CREATE TABLE Enrolled
    (sid CHAR(20),
     cid  CHAR(20),
     grade CHAR(2),
 PRIMARY KEY(sid),
 UNIQUE (cid, grade))
```

!Used carelessly, an Integrity Constraint can prevent the storage of database instances that arise in practice!

<u>Foreign Keys</u>

"*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 )
```



Enrolled

sid	*cid*	*grade*
1234	Alg1	9
1235	Alg1	10
1234	DB1	10
1234	DB2	9

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

<u>Referential Integrity</u>

SQL-99 and SQL-2003 support all 4 options on deletes and updates.

The default is NO ACTION (*delete/update is rejected*).

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

They are useful when more general ICs than keys are involved.

One can use queries to express constraints.

Constraints can be named.

```
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))
```