CS5530/6530 Database Systems
Fall 2005
Key to Assignment 2 – Relational Languages: SQL and Functional Dependencies
Due: Tue-Oct 4

Aside from *regular exercises*, your assignment contains *practice exercises* which will help you cover the material in the textbook and lecture notes. Only regular exercises will be corrected -- you should not turn in the practice exercises. I will provide the key to practice exercises and I may use some of these in your exams.

Some questions are intended only for students taking cs6530. Unless explicitly stated in the question, cs5530 should not turn in these questions – they will not be graded.

NOTE: A given query can be expressed in many different ways in SQL, thus the solutions provided below give one, and in some cases 2 expressions for the queries. Other solutions are possible.

Regular Exercises

1. **The university of Utah maintains a database with information about students, professors and courses. The SQL descriptions of each relation in the database is given below. A sample database is provided for you to run your queries against. Besides the actual queries, you should also submit *electronically* (using the submit system) the results of the queries.**

```
create table Student (
        Sid             int,
        Sname           varchar(30),
        GPA             real,
        Saddr           varchar(100),
        primary key (Sid)
);


create table Professor (
        Pid             int,
        Pname           varchar(30),
        Paddr           varchar(100),
        primary key (Pid)
);
```

```
create table Course (
        Cid             int,
        Cname           varchar(30),
        required        int,
        primary key (Cid)
);

create table TA (
        Sid             int,
        Cid             int,
        Year            int,
        foreign key (Sid) references Student(Sid),
        foreign key (Cid) references Course(Cid)
);

create table Takes (
        Sid             int,
        Cid             int,
        grade           real,
        Year            int,
        foreign key (Sid) references Student(Sid),
        foreign key (Cid) references Course(Cid)
);

create table Teaches (
        Pid             int,
        Cid             int,
        Year            int,
        foreign key (Pid) references Professor(Pid),
        foreign key (Cid) references Course(Cid)
);
```

The meaning of these tables and their attributes should be obvious. Write the following queries in SQL: [60 points]

a. **List the names of all the people in the university database whose address contains the string "salt".**

```
(select name from student where saddr LIKE '%salt%')
  UNION ALL
  (select name from professor where saddr LIKE '%salt%')
```

Note here the use of UNION ALL, which will keep duplicated names in the results.

b. **List the names of all professors, and the names of all the courses they have taught (if any).**

```
 select pname,cname
 from professor P OUTERJOIN teaches T JOIN course C
 where P.pid = T.pid AND T.cid = C.cid
```

c. **List the names of professors that have not taught a course since 2000 in descending order of name.**

```
select Pname
   from Professor
   where Pid not in (select Pid
        from Teaches
        where year > 2000)
   order by Pname desc;
```

A more involved solution:

```
 select Pname from
  (select Pname
  from Professor P, Teaches T
  where P.pid = T.pid
  group by Pname
  having MAX(year) < 2000)
  UNION
  (select Pname
  from Professor
         where Pid not in (select Pid
              from Teaches)
  )
order by Pname desc
```

d. **Since the database was just created, students have no GPAs in their records (the Student table). Using the information about the courses each student took, compute their GPA. Keep in mind that when students audit courses, they do not receive a grade for that course.**

```
select sid, (sum(grade)/count(grade)) as gpa
from takes
group by sid
```

e. **Using the GPAs computed in the previous query, update the GPA column in the Student table.**

```
UPDATE student
SET gpa =  (SELECT (sum(grade)/count(grade))
   FROM  takes
   WHERE Student.sid = Takes.sid);
```

NOTE: This is an example of a ***correlated update***

f. **List the name of all students whose GPA is different from 3.0. (this should include students whose GPA is NULL)**

```
(Select name from student)
Minus
(Select name from student where gpa = 3.0)
```

g. **List the name and address of students whose GPA are the highest in the database.**

```
Select name,gpa from student
Where gpa = (select max(gpa) from student)
```

h. **List all CS courses and for each course, compute the average score obtained by the students each time the course was taught. A sample result could be:**

```
Databases 2003    2.8
Databases 2004    3.5
Graphics   2001   4.0
```

```
Create view GradeAvg as
        Select cid, year, (sum(grade)/count(grade)) as averg
        from takes
        group by cid, year;

    select Cname, year, averg
    from Course natural join GradeAvg;
```

An alternative would be to use a subquery:

```
select Cname, year, averg
from Course natural join
    (Select cid, year, (sum(grade)/count(grade)) as averg
    from takes
    group by cid, year)
```

i. **List the names of students and the names of the courses they TA'ed but did not take.**

```
 create or replace view TANotTakes as
((select Sid, Cid
        from TA)
 minus
(select TA.Sid, Takes.Cid
          from Takes, TA
 where Takes.Sid = TA.Sid));

select S.Sname, C.Cname
          from Student S, Course C, TANotTakes TNT
where S.Sid = TNT.Sid and C.Cid = TNT.Cid;
```

What about the following query?

```
(select s.sname
 from student s
 where s.sid in (
          select ta.sid
          from ta,course
          where ta.cid = course.cid))
 MINUS
 (select s.sname
 from student s
 where s.sid in
          (select takes.sid
          from takes,course
          where takes.cid = course.cid));
```

j. **[cs6530, and extra credit for 5530] List the names of all students who have taken all the required CS courses.**

(this requires division)

```
SELECT  S.sname
FROM  Student S
WHERE  NOT EXISTS
          ((SELECT  C.cid
            FROM  Course C
            WHERE C.required = 1)
           MINUS
            (SELECT  T.cid
             FROM  Takes T
             WHERE  S.sid=T.sid ));
```

2. **[cs6530] Propose 2 improvements for the design of the university database that address some of the complications you have encountered while formulating the queries of question 1. (10 points)**

There are several possible improvements to the tables defined above. For example:

- *You could disallow NULL values for grades, either by constraining the attribute grade to be non-NULL*

```
create table Takes (
        Sid             int,
        Cid             int,
        grade           real NOT NULL,
        Year            int,
        foreign key (Sid) references Student(Sid),
        foreign key (Cid) references Course(Cid)
);
```

Or by creating a separate table for storing the grades.

- *You could specify primary keys for TA, Takes and Teaches*

- *You could create a separate relation that keeps track of course sections:*

```
create table CourseSection (
        Cid             int,
        Year            int,
        Primary key (Cid,Year)
        foreign key (Cid) references Course(Cid)
);
```

Assuming that a course is taught only once a year, i.e., the primary key for CourseSection is (Cid,Year). To ensure the tuples in Takes are valid and correspond to courses that were taught, you could create a foreign key from Takes to CourseSection:

```
create table Takes (
    Sid             int,
    Cid             int,
    grade           real,
    Year            int,
    primary key (Sid,Cid,Year),
    foreign key (Sid) references Student(Sid),
    foreign key (Cid,Year) references CourseSection(Cid,Year) );
```

And to ensure the tuples in Teaches are valid and correspond to courses that had actual sections, you could create a foreign key from Teaches to CourseSection:

```
create table Teaches (
    Pid             int,
    Cid             int,
    Year            int,
    primary key (Pid,Cid,Year),
    foreign key (Pid) references Professor(Pid),
    foreign key (Cid,Year) references Course(Cid,Year) );
```

3. **Express each of the following integrity constraints in SQL unless it is implied by the primary and foreign key constraint; if so, explain how it is implied. If the constraint cannot be expressed in SQL, say so. For each constraint, state what operations (inserts, deletes, and updates on specific relations) must be monitored to enforce the constraint. (20 points)**

   a. **Every class has a minimum enrollment of 5 students and a maximum enrollment of 30 students.**

   The takes table should be modified as follows:

```
CREATE TABLE Takes (
        Sid             int,
        Cid             int,
        grade           real,
        Year            int,
        PRIMARY KEY (snum, cname),
        foreign key (Sid) references Student(Sid),
        foreign key (Cid) references Course(Cid)
CHECK ( ( SELECT COUNT (T.sid)
FROM Takes T
GROUP BY T.cid, T.year) >= 5),
CHECK ( ( SELECT COUNT (T.sid)
FROM Takes T
GROUP BY T.cid, T.year)<=30)
```

   b. **Every faculty member must teach at least two courses every year.**

   Note that we also need to include professors who did not teach any course and whose pid is not present in the Teaches table.

```
CREATE ASSERTION TeachTwo
CHECK ( ( SELECT COUNT (*) FROM
                (SELECT Professor P, Teaches T
                WHERE P.pid = T.pid
                GROUP BY T.Pid, T.year
                HAVING COUNT (*) < 2))=0)
            AND
                (SELECT COUNT(*) FROM
                    (SELECT pid
                    FROM Professor
                    WHERE pid NOT IN (SELECT pid FROM
Teaches)))=0))
```

   The assertion below is INCORRECT because it ignores professors who never taught a course.

```
CREATE ASSERTION TeachTwo
CHECK ( ( SELECT COUNT (*) FROM
                (SELECT Professor P, Teaches T
                WHERE P.pid = T.pid
                GROUP BY T.Pid, T.year
                HAVING COUNT (*) < 2)))=0)
```

c. **Only faculty in that do not live in Salt Lake City can teach fewer than 2 courses per year.**

```
CREATE ASSERTION TeachLessThanTwo
CHECK (
        NOT EXISTS(
                (SELECT pid
                FROM Professor P, Teaches T
                WHERE P.pid = T.pid and P.Paddr LIKE  '%salt lake
                city%'
                GROUP BY T.pid,T.year
                HAVING COUNT (*) < 2)
                                        UNION ALL
              (SELECT pid
              FROM Professor
               WHERE P.Paddr LIKE  '%salt lake%'
                  AND  pid NOT IN (SELECT pid FROM Teaches))
           ))
```

d. **Every student must take either Databases or Advanced Databases.**

```
CREATE ASSERTION ReqDB
```

```
CHECK (NOT EXISTS (select S.sid, count (*)
From Student S, Takes T
Where S.sid = T.sid and (T.cid = 1 or T.cid = 5)
Group by S.sid
Having count(*) < 1))
```

e. **[cs6530] A student must be enrolled in more required courses than in non-required courses.**

```
CREATE ASSERTION ReqEnroll
CHECK (NOT EXISTS (
select SReq.sid from
(select T1.sid, count (*) AS C1
 From Course C, takes T1
Where C.cid = T1.cid and C.required = 1
Group by T1.sid) SReq,
(select T2.sid, count (*) AS C2
 From Course C, takes T2
Where C.cid = T2.cid and C.required = 0
Group by T2.sid) SNotReq
Where SReq.sid = SNotReq.sid AND SReq.C1 <= SNotReq.C2;))
```

4. **Given the ER diagram below and a possible set of relations to represent this database: (20 points)**

Movies(<u>title, year</u>, length)
Sequel(title<u>_orig,year_orig,title_seq,year_seq</u>) – fkeys into Movies
Owns(title, year, studio_name) – fkey(title,year)→ Movies;
fkey(studio_name)→Studios
Studios(<u>name</u>, address)
StarsIn(title, year, star_name, salary) – fkey(title, year)→ Movies;
fkey(star_name)→Stars
Stars(<u>name</u>, address)

a. **Write in SQL statements to create these relations. You need to specify all the keys and foreign keys.**

```
Create table Movies (
      Title            varchar(30),
      Year             int,
      Length           real,
      Primary key (title, year)
);

Create table Sequel (
      Title_orig       varchar(30),
      Year_orig        int,
      Title_seq        varchar(30),
     Year_seq        int,
     Primary key (Title_orig,Year_orig,Title_seq,Year_seq),
     Foreign key (title_orig,year_orig) references Movies(title,year),
     Foreign key (title_seq, year_seq) references Movies(title,year)
);


Create table Studios (
     Name            varchar(30) PRIMARY KEY,
     Address         varchar(100)
);

Create table Owns (
     Title           varchar(30),
     Year            int,
     Studio_name     varchar(30),
     Primary key(title,year,studio_name),
     Foreign key (title,year) references Movies(title,year),
     Foreign key (studio_name) references Studios(name)
);


Create table Stars (
     Name            varchar(30) PRIMARY KEY,
     Address         varchar(100)
);

Create table StarsIn (
     Title           varchar(30),
     Year            int,
     star_name       varchar(30),
     salary          real,
     Primary key (title,year,star_name),
     Foreign key (title,year) references Movies(title,year),
     Foreign key (star_name) references Stars(name)
);
```

b. **Suppose your database system does not allow the definition of foreign keys. How would you express the foreign keys above using the other integrity constraints checking mechanisms in SQL?**

You would need to use assertions. For example, to enforce the following foreign key in the Owns relation:

```
          Foreign key (studio_name) references Studios(name)
```

you could create the following assertion:

```
CREATE ASSERTION
CHECK (NOT EXISTS (
                (SELECT studio_name
                FROM Owns
                WHERE studio_name NOT IN
                        (SELECT name from Studios))))
```

We might suppose that we could simulate a referential integrity constraint by an attribute-based CHECK constraint that requires the existence of the referred value. The following is an **erroneous** *attempt to simulate the foreign key constraint in the Owns relation:*

```
        Studio_name   varchar(30)
                CHECK ( studio_name IN (SELECT name FROM Studios))
```

This constraint guarantees that if we add a new tuple or modify the studio_name value of a tuple in Owns, the update is only allowed if the value for studio_name is in the Studios table. However, if we change the Studios relation, say, by deleting a tuple for one studio, this is **invisible** *to the CHECK constraint above. Thus, the deletion is permitted even though the attribute-based CHECK constraint on studio_name is now violated*

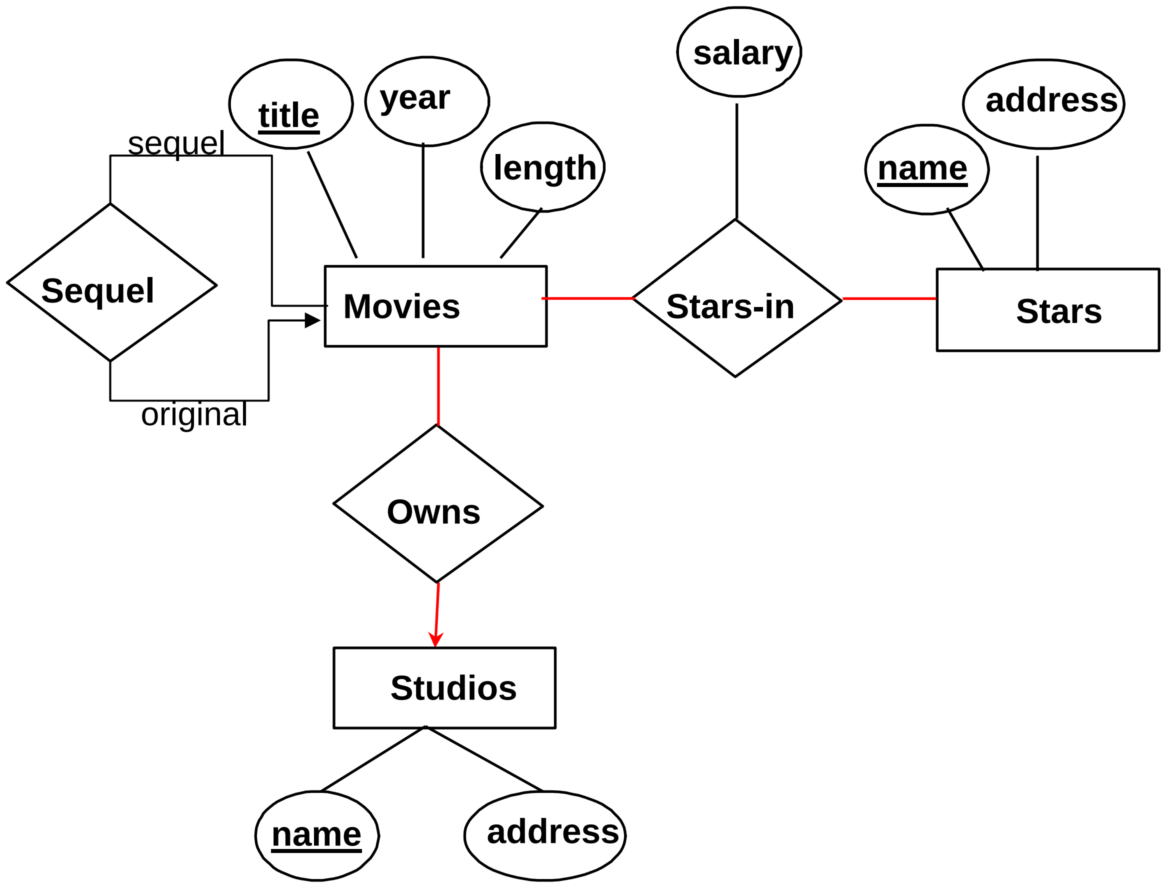
Unlike value- and tuple-based constraints, which are only checked when the table in which they were defined is updated, assertions are checked when *any* of the tables they refer to is modified.

c. [cs6530] Write an SQL query that lists all the sequels for the movie Friday the 13th which was released in 1980.
(this requires a recursive query)

```
WITH RECURSIVE Mov(Title_orig, Year_orig, Title_seq, Year_seq) AS

(select S1.Title_orig, S1.Year_orig, S1.Title_seq, S1.Year_seq
    From Sequel S1
 Where S1.Title_orig = 'Friday the 13th' and S1.Year_orig = 1980)
UNION ALL
(select S2. Title_orig, S2.Year_orig, M1.Title_seq, M1.Year_seq
    From Sequel S2, Mov M1
 Where S2. Title_seq = M1.Title_orig and S2.Year_seq = M1.Year_orig)

Select * from Mov;
```

Another alternative:

```
WITH RECURSIVE AllSequels(Title_orig, Year_orig, Title_seq, Year_seq) AS

(select S1.Title_orig, S1.Year_orig, S1.Title_seq, S1.Year_seq
    From Sequel S1)
UNION ALL
(select S2. Title_orig, S2.Year_orig, AS.Title_seq, AS.Year_seq
    From Sequel S2, AllSequels AS
 Where S2. Title_seq = AS.Title_orig and S2.Year_seq = AS.Year_orig)

Select Title_seq, Year_seq  from AllSequels
Where Title_orig = ‘Friday the 13th’ and Year_orig = 1980;
```



Practice Exercises

Consider the following schema from Assignment 1:
Suppliers(<u>*sid*</u>*:* integer, *sname:* string, *address:* string)
Parts(<u>*pid*</u>*:* integer, *pname:* string, *color:* string)
Catalog(<u>*sid*</u>*:* integer, <u>*pid*</u>*:* integer, *cost:* real)

The key fields are underlined, and the domain of each field is listed after the field name. Therefore *sid* is the key for Suppliers, *pid* is the key for Parts, and *sid* and *pid* together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in SQL:

1. Find the *name*s of suppliers who supply some red part.
2. Find the *sid*s of suppliers who supply some red or green part.
3. Find the *sid*s of suppliers who supply some red part or are at 221 Packer Street.
4. Find the *sid*s of suppliers who supply some red part and some green part.
5. Find the *sid*s of suppliers who supply every part.
6. Find the *sid*s of suppliers who supply every red part.
7. Find pairs of *sid*s such that the supplier with the first *sid* charges more for some part than the supplier with the second *sid*.
8. Find the *pid*s of parts supplied by at least two different suppliers.
9. Find the *pid*s of the most expensive parts supplied by suppliers named Yosemite Sham.
10. Find how many suppliers supply some red part.

Answer:

```
1. SELECT S.sname
FROM Suppliers S, Parts P, Catalog C
WHERE P.color='red' AND C.pid=P.pid AND C.sid=S.sid

2. SELECT C.sid
FROM Catalog C, Parts P
WHERE (P.color = 'red' OR P.color = 'green')
AND P.pid = C.pid

3. SELECT S.sid
FROM Suppliers S
WHERE S.address = '221 Packer street'
OR S.sid IN ( SELECT C.sid
FROM Parts P, Catalog C
WHERE P.color='red' AND P.pid = C.pid )

4. SELECT C.sid
FROM Parts P, Catalog C
WHERE P.color = 'red' AND P.pid = C.pid
AND EXISTS ( SELECT P2.pid
FROM Parts P2, Catalog C2
WHERE P2.color = 'green' AND C2.sid = C.sid
AND P2.pid = C2.pid )

5. SELECT C.sid
FROM Catalog C
WHERE NOT EXISTS (SELECT P.pid
FROM Parts P
WHERE NOT EXISTS (SELECT C1.sid
FROM Catalog C1
WHERE C1.sid = C.sid
AND C1.pid = P.pid))

6. SELECT C.sid
FROM Catalog C
WHERE NOT EXISTS (SELECT P.pid
FROM Parts P
WHERE P.color = ‘red’
AND (NOT EXISTS (SELECT C1.sid
FROM Catalog C1
WHERE C1.sid = C.sid AND
C1.pid = P.pid)))


7. SELECT C1.sid, C2.sid
FROM Catalog C1, Catalog C2
WHERE C1.pid = C2.pid AND C1.sid  <> C2.sid
AND C1.cost > C2.cost

8. SELECT C.pid
FROM Catalog C
WHERE EXISTS (SELECT C1.sid
FROM Catalog C1
WHERE C1.pid = C.pid AND C1.sid  <> C.sid )

9. SELECT C.pid
FROM Catalog C, Suppliers S
WHERE S.sname = ‘Yosemite Sham’ AND C.sid = S.sid
AND C.cost ≥ ALL (Select C2.cost
FROM Catalog C2, Suppliers S2
WHERE S2.sname = ‘Yosemite Sham’
AND C2.sid = S2.sid)

10. SELECT COUNT(S.sid)
FROM Suppliers S, Parts P, Catalog C
WHERE P.color=’red’ AND C.pid=P.pid AND C.sid=S.sid
```