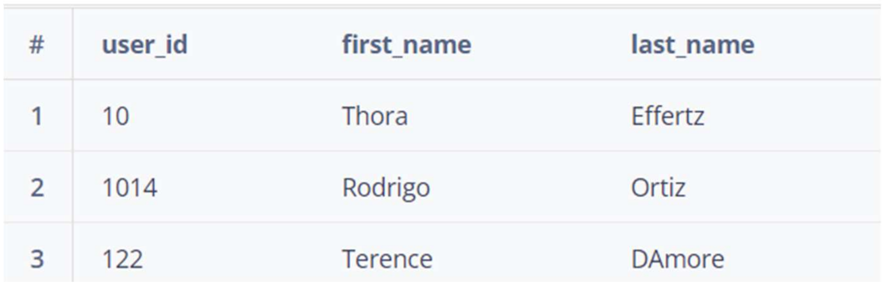
COEN 280 HW 3
Justin Li

1. [10pts] For all students majoring in ‘ART’ and having taken at least 2 classes offered by ‘COEN’ dept, list their user_id, first_name, and last_name.

SQL:
```
SELECT DISTINCT U.user_id, U.first_name, U.last_name
FROM Users U
JOIN Student S ON U.user_id = S.user_id
JOIN Takes T ON U.user_id = T.user_id
JOIN Class C ON T.class_no = C.class_no AND T.dept = C.dept AND T.cno = C.cno
WHERE S.major = 'ART' AND C.dept = 'COEN'
GROUP BY U.user_id, U.first_name, U.last_name
HAVING COUNT(T.class_no) >= 2;
```

Show result row(s):



#	user_id	first_name	last_name
1	10	Thora	Effertz
2	1014	Rodrigo	Ortiz
3	122	Terence	DAmore

Create a SQL view so that other can look at the data and draw meaningful conclusions without having to deal with all of its underlying complexity.
The view should provide access to a combination of the following information:
• Student info (user_id, first_name, last_name)
• Take info (class_no, dept, cno, grade)
• Course info (title, level)

a) [20 pts] Create the desired view (CSStudentView) by writing an appropriate CREATE VIEW statement.
```
CREATE VIEW CSStudentView AS
SELECT U.user_id, U.first_name, U.last_name, T.class_no, T.dept, T.cno, T.grade,
C.title, C.level
FROM Users U
JOIN Student S ON U.user_id = S.user_id
JOIN Takes T ON U.user_id = T.user_id
JOIN Course C ON T.dept = C.dept AND T.cno = C.cno;
```

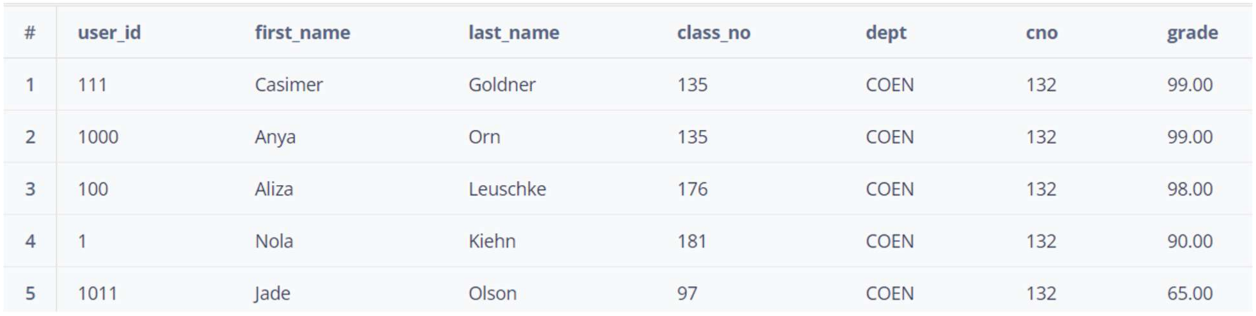
b) [5 pts] Show the usefulness of your view by writing a SELECT query against the view that prints the user_id, first_name, last_name of the students who have the highest grade in the course titled ‘Computer Networks’.

SQL:

```
SELECT user_id, first_name, last_name
FROM CSStudentView
WHERE title = 'Computer Networks'
ORDER BY grade DESC
LIMIT 1;
```

c) [10 pts]
Run some one insert and one update queries on your view. Show your query and observe if the data in underlying tables also gets updated.

```
UPDATE Takes
SET grade = 99
WHERE user_id = ‘111’ AND class_no = ‘135’ AND dept = ‘COEN’ AND cno = ‘132’;
INSERT INTO Takes (user_ID, class_no, dept, cno, grade)
VALUES (‘1000’, ‘135’, ‘COEN’, ‘132’, 99);
SELECT user_id, first_name, last_name, class_no, dept, cno, grade
FROM CSStudentView
WHERE title = ‘Computer Networks’
ORDER BY grade DESC;
```



#	user_id	first_name	last_name	class_no	dept	cno	grade
1	111	Casimer	Goldner	135	COEN	132	99.00
2	1000	Anya	Orn	135	COEN	132	99.00
3	100	Aliza	Leuschke	176	COEN	132	98.00
4	1	Nola	Kiehn	181	COEN	132	90.00
5	1011	Jade	Olson	97	COEN	132	65.00

5. Stored Procedures [20 pts]
With a normalized schema, every time a new student is registered, we have to manipulate a few tables, such as User, Student/Instructor and Phone. Seeking to simplify the process, we have noticed that most new users are students and that they only have a mobile phone. As a result, we want to create a fast path
for registering such new students using stored procedures.

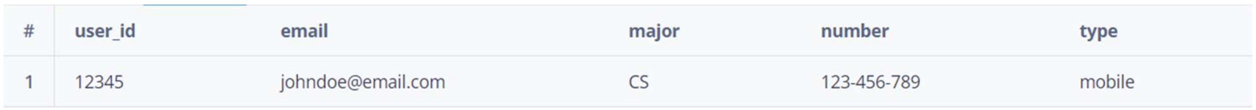
a) [15 pts] Create and exercise a SQL stored procedure called RegisterStudent(...) that the site's application can use to add a new student with a mobile phone to the database. Modify the trigger if it doesn't work for your DB setup.

```
CREATE PROCEDURE
RegisterStudent
(user_id VARCHAR(20),
email VARCHAR(50),
first_name VARCHAR(30),
last_name VARCHAR(30),
major VARCHAR(50),
mobile_number VARCHAR(20))
LANGUAGE SQL
AS $$
// Update Users, Students and Phone tables.
BEGIN
   INSERT INTO Users (user_id, email, first_name, last_name)
   VALUES (p_user_id, p_email, p_first_name, p_last_name);

   INSERT INTO Student (user_id, major)
   VALUES (p_user_id, p_major);

   INSERT INTO Phone (user_id, type, number)
   VALUES (p_user_id, 'mobile', p_mobile_number);
END;
$$;
```

b) [5pts] Verify that your new stored procedure works properly by calling it as follows to add a new student and then running a SELECT query to show the stored procedure's after-effects:

```
call RegisterStudent ('12345', 'johndoe@email.com', 'John','Doe', 'CS', '123-456-789');
select User.user_id, User.email, Student.major, Phone.number, Phone.type
from User, Student, Phone
where User.user_id = Student.user_id
AND User.user_id = Phone.user_id AND User.user_id = '12345';
```

Show the result of executing this query.



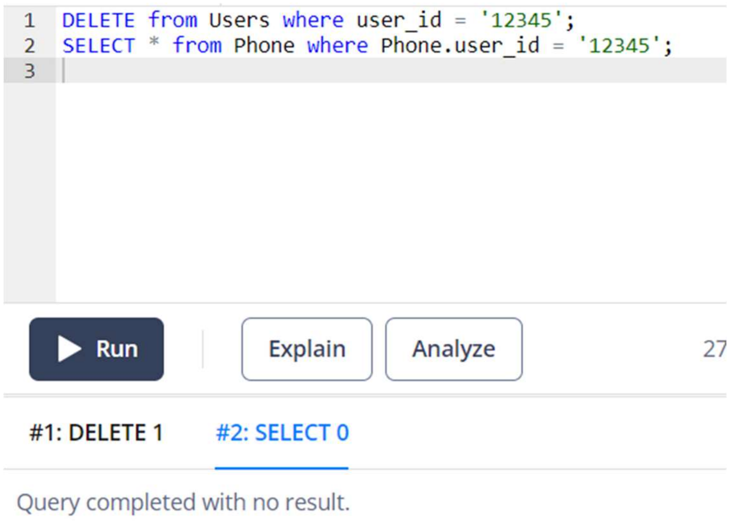
#	user_id	email	major	number	type
1	12345	johndoe@email.com	CS	123-456-789	mobile

6. Alter Table [10 pts]
Modify the original schema to accommodate a new feature. As the database already has valuable information that we need to retain, we just want to alter one of the existing tables.
The new feature request is to cause a user's phone data to be deleted when the user is deleted. What do you need to change in the Phone table to achieve that?
a) [5 pts] Write and execute the ALTER TABLE statement(s) needed to modify the Phone table so that whenever the user associated with the phone information is deleted then the phone data is deleted too. (Note: the name of the existing foreign key constraint for user_id is phone_ibfk_1).

```
ALTER TABLE Phone
DROP CONSTRAINT phone_ibfk_1,
ADD CONSTRAINT phone_ibfk_1 FOREIGN KEY (user_id)
REFERENCES Users(user_id)
ON DELETE CASCADE;
```

b) [5 pts] Execute the following DELETE and SELECT statements to show the effect of your change. Write down the result returned by the SELECT statement.

```
DELETE from User where user_id = '12345';
SELECT * from Phone where Phone.user_id = '12345';
```



```
DELETE from Users where user_id = '12345';
SELECT * from Phone where Phone.user_id = '12345';
```

#1: DELETE 1   #2: SELECT 0

Query completed with no result.

7. Triggers [20 pts]
The popularity of a post is computed based on how many likes a post receives. Previously, the popularity was being updated using a batch job which runs on a daily basis. As a result, there could be some staleness of popularities. Now we want to

update the popularity of posts in real-time when likes are inserted. (Note: You don't have to deal with the case of the deletion of a like here, or updates for that matter; just process new like insertions.)
a) [15 pt] Use the CREATE TRIGGER statement in Postgres or MYSQL to define a trigger that will do the desired job. Modify the trigger if it doesn't work with your DB,

```
CREATE FUNCTION update_popularity()
  RETURNS TRIGGER
  LANGUAGE PLPGSQL
AS $$
BEGIN
    UPDATE Post
    SET popularity = popularity + 1
    WHERE post_id = NEW.post_id;
    RETURN NEW;
END
$$;

CREATE TRIGGER update_popularity_trigger
AFTER INSERT ON Likes
FOR EACH ROW
EXECUTE FUNCTION update_popularity();
```

b) [5 pts] Execute the following INSERT and SELECT statements to show the effect of your trigger. Write down the popularity returned by each SELECT statement.
(Note: the first two selected popularity values may not differ by just 1 since the previous value is stale.)

```
SELECT popularity FROM Post where Post.post_id = '1'; Effect: NULL
INSERT INTO Likes VALUES('1141', '1');
SELECT popularity FROM Post where Post.post_id = '1'; Effect: 1
INSERT INTO Likes VALUES('1140', '1');
SELECT popularity FROM Post where Post.post_id = '1' Effect: 2
```