Last Name: First Name: Student ID:

1. [10pts] For all students who enrolled in at least 3 courses after the date 2020-09-24, list the students' ids, first_name and last_name.

a) [7pts] SQL Query:

```
SELECT s.user_id, u.first_name, u.last_name
FROM users u, student s, enrolledin e, course c
WHERE u.user_id = s.user_id AND s.user_id = e.user_id AND
      e.course_id = c.course_id AND e.enroll_date > '2020-09-24'
GROUP BY s.user_id, u.first_name, u.last_name
HAVING count(c.course_id) >= 3
```

b) [3pts] Result:

	user_id text	first_name text	last_name text
1	292	Brian	Cowan
2	419	Christina	Robinson

2. [10pts] Find the longest meeting duration for each course taught by the instructor with user_id '486'. Show only the course_id, course_name, and the maximum duration. Rank the results by the durations from highest to lowest.

a) [7pts] SQL Query:

```
SELECT c.course_id, c.course_name, max(m.duration)
FROM instructor i, teaches t, course c, meeting m
WHERE i.user_id = t.user_id AND c.course_id = t.course_id AND
      c.course_id = m.course_id AND i.user_id = '486'
GROUP BY c.course_id, c.course_name
ORDER BY max(m.duration) desc
```

b) [3pts] Result:

	course_id [PK] text	course_name text	max integer
1	40	Geology I	120
2	0	Communications 101	90
3	46	Communications 102	90
4	39	German II	60
5	23	Informatics	60
6	74	Physics 102	60
7	31	Finance 102	60
8	38	Programming	30
9	54	Medicine	30
10	4	Geography II	30

3. [10pts] For all posts that are liked by at least 0.4% of all users, list the post_id and the number of users who thumbed up the post. Rank your results by the number of users (likers) from the highest to the lowest.

a) [7pts] SQL Query:

```
SELECT p.post_id, count(t.user_id)
FROM post p, thumbsup t, users u
WHERE p.post_id = t.post_id AND u.user_id = t.user_id
GROUP BY p.post_id
HAVING COUNT(t.user_id) >= (
      SELECT COUNT(u2.user_id)*0.004
      FROM users u2
)
ORDER BY count(t.user_id) desc
```

b) [3pts] Result:

	post_id [PK] text	count bigint
1	66	4
2	101	4
3	53	4
4	0	3
5	25	3
6	33	3

4. **Views** [20 pts]

It's finally time to complete the last piece of the puzzle - the "good_posts" attribute from the E-R diagram that we designed earlier. Recall that "good_posts" is a derived attribute, and it's an accounting of the total number of "thumbs up"s that each student has **received**. Note that while each post can get multiple "thumbs up"s, we want to consider each such good post as **only one good post**. In addition, if a student does not have any good posts, their number of good posts should be zero. To implement this, we need to create a view. The view should include each students' user_id and their number of good posts.

a) [15 pts] Create the desired view (StudentView) by writing an appropriate CREATE VIEW statement.

```
CREATE VIEW swoosh.StudentView (user_id, occupation, good_posts) AS (
      SELECT s.user_id, s.occupation, count(distinct p.post_id)
      FROM student s join users u on s.user_id = u.user_id left join (post
p join thumbsup t on t.post_id = p.post_id) on u.user_id = p.user_id
      GROUP BY s.user_id
);
```

b) [5 pts] Show the usefulness of your view by writing a SELECT query against the view that prints the user_id, first_name, and last_name of the student who received the largest number of good posts.

```
SELECT s.user_id, u.first_name, u.last_name
FROM StudentView s, users u
WHERE s.user_id = u.user_id AND s.good_posts = (
SELECT MAX(s2.good_posts)
FROM StudentView s2
)
```

	user_id text	first_name text	last_name text
1	213	Dominique	Harper

5. **Stored Procedures** [20 pts]

a) [15 pts] Create and exercise a SQL stored procedure called RegisterInstructor(…) that the application developer can then use to add a new instructor with one piece of initial optional education info to the

database.

```
CREATE PROCEDURE RegisterInstructor(
      IN user_id text,
      IN email text,
      IN first_name text,
      IN last_name text,
      IN title text,
      IN "degree" text,
      IN graduation_year integer,
      IN major text,
      IN school text,
      IN education_id text
)
LANGUAGE PLPGSQL
AS $$
      BEGIN
      INSERT INTO users(email, first_name, last_name, user_id) VALUES
(email, first_name, last_name, user_id);
      INSERT INTO instructor(title, user_id) VALUES (title, user_id);
      IF ("degree" IS NOT NULL OR major IS NOT NULL OR graduation_year IS
NOT NULL OR school IS NOT NULL) THEN
      INSERT INTO instructoreducation("degree", education_id,
graduation_year, instructor_id, major, school)
      VALUES ("degree", education_id, graduation_year, user_id, major,
school);
      END IF;
      END;
$$;
```

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

```
CALL registerinstructor('999', 'ins@uci.edu', 'peter', 'anteater',
'mascot', 'BS', 2099, 'CS', 'ICS', '998');

SELECT i.user_id, u.first_name, u.last_name, u.email, ie.degree, ie.school
FROM instructor i, users u, instructoreducation ie
WHERE i.user_id = u.user_id AND i.user_id = ie.instructor_id AND i.user_id
= '999';
```

Result:

	user_id text	first_name text	last_name text	email text	degree text	school text
1	999	peter	anteater	ins@uci.edu	BS	ICS

6. **Alter Table** [10 pts]

a) [5 pts] Write and execute the ALTER TABLE statement(s) needed to modify the recording table so that when the meeting associated with a recording is deleted, the recording will ***not*** also be deleted. It should be retained instead. (*Note:* The name of the existing foreign key constraint for the meeting_id field is *recording_meeting_id_fkey*).

```
ALTER TABLE recording DROP CONSTRAINT recording_meeting_id_fkey;
```

b) [5 pts] Execute the following DELETE and SELECT statements to show the effect of your change. Report the COUNT query's result (just the number) returned by the SELECT statement both before and after your DELETE.

```
SELECT COUNT(*)
FROM recording r
WHERE r.meeting_id = '50';

DELETE FROM meeting
WHERE meeting_id = '50';

SELECT COUNT(*)
FROM recording r
WHERE r.meeting_id = '50';
```

Result: 1 and 1

7. **Triggers** [20 pts]

a) [15 pt] Create a table watchedvideo(recording_id, user_id, PRIMARY KEY(recording_id, user_id)) that stores the recordings watched by students as well as the students who watched them. Then write a CREATE TRIGGER statement (**by hand** of course!) to define a trigger that will do the following job: after a student has watched a segment of a recording -- indicated by an insert into the watchedsegment table -- if the sum of the student's watched segment durations of a recording add up to at least the recording length, we say the student has (in fact) watched the recording, and we therefore insert the recording_id and the student's user_id into the watchedvideo table. The new table is only responsible for keeping the watching records after the trigger is created. Use the CREATE FUNCTION statement as well as needed. Your function should ignore duplicate inserts to the new table. HINT: use "...ON CONFLICT..." to handle insertion conflicts.

```
CREATE TABLE watchedvideo(
      recording_id text,
      user_id text,
      PRIMARY KEY (recording_id, user_id),
      FOREIGN KEY (recording_id) REFERENCES recording(recording_id) ON
DELETE CASCADE,
      FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);

CREATE FUNCTION AddWatched()
RETURNS Trigger
AS $$
      BEGIN
            INSERT INTO watchedvideo(recording_id, user_id)
            SELECT NEW.recording_id, NEW.user_id
            WHERE (SELECT r.end_time - r.start_time
                   FROM recording r
                   WHERE r.recording_id = NEW.recording_id)
                  <=
                  (SELECT SUM(w.watched_to - w.watched_from)
                  FROM watchedsegment w
                  WHERE recording_id = NEW.recording_id AND user_id =
NEW.user_id) ON CONFLICT DO NOTHING;
            RETURN NEW;
      END;
$$
LANGUAGE PLPGSQL;

CREATE TRIGGER WatchedRecordingLogger
AFTER INSERT ON watchedsegment
```

```
FOR EACH ROW
EXECUTE FUNCTION AddWatched();
```

b) [5 pts] Execute the following INSERT and SELECT statements to show the effect of your trigger. Report the results.

```
SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
```

recording_id [PK] text	user_id [PK] text

```
INSERT INTO watchedsegment(recording_id, user_id, segment_id, watched_from,
watched_to)
VALUES ('2252', '0', '10', '17:00:00', '17:20:00');

SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
```

recording_id [PK] text	user_id [PK] text

```
INSERT INTO watchedsegment(recording_id, user_id, segment_id, watched_from,
watched_to)
VALUES ('2252', '0', '11', '17:00:00', '19:00:00');

SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
```

	recording_id [PK] text	user_id [PK] text
1	2252	0

```
INSERT INTO watchedsegment(recording_id, user_id, segment_id, watched_from,
watched_to)
VALUES ('2252', '0', '12', '17:00:00', '18:09:00');

SELECT *
FROM watchedvideo
WHERE recording_id = '2252';
```

	recording_id [PK] text	user_id [PK] text
1	2252	0