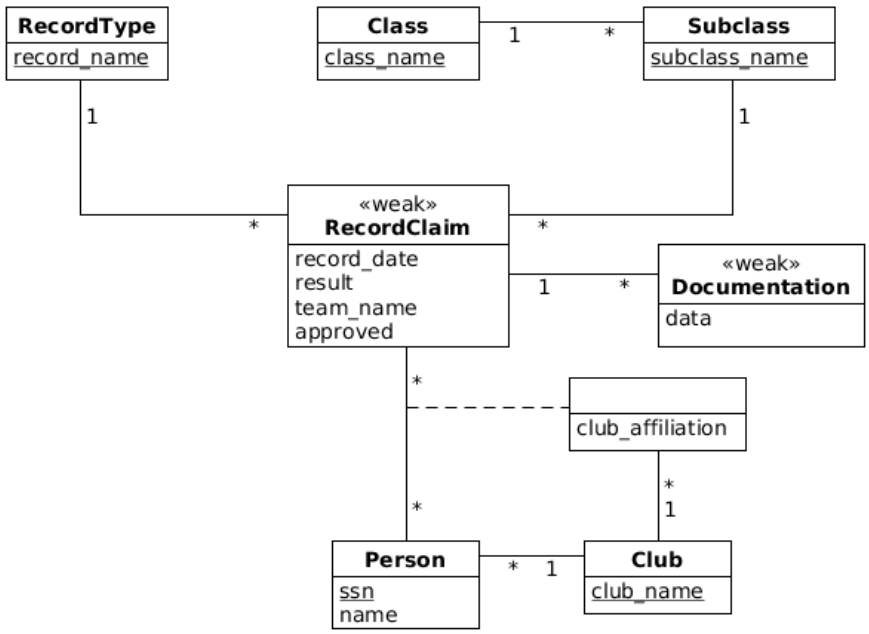
Solutions to EDA216 exam

March 15, 2016

Solution 1

(a) We can draw the ER-model several ways, this is one attempt:



(b) From the model, our entity sets immediately give the following relations (primary keys are underlined, foreign keys will be *italicized*):

```
record_types(record_name)
classes(class_name)
subclasses(subclass_name)
record_claims(record_date, result, team_name, approved)
people(ssn, name)
clubs(club_name)
```

It looks as if `record_claims` will get a long and unwieldy primary key, so we might as well invent a key for it:

```
record_claims(claim_id, record_date, result, team_name, approved)
```

Now we can look at "many-to-1"-associations, to avoid creating unnecessary relations:

```
record_types(record_name)
classes(class_name)
subclasses(subclass_name, class_name)
record_claims(claim_id, record_date, result, approved, record_name, subclass_name)
people(ssn, name, club_name)
clubs(club_name)
```

Finally we implement our only "many-to-many" association by creating one more relation:

```
record_setters(ssn, claim_id, club_affiliation)
```

There are no functional dependencies in these relations where the left hand side isn't a key in the relation, so they are all in BCNF[1].

(c) I prefer to break it down using views, first all approved records:

```
DROP VIEW IF EXISTS approved_records;
CREATE VIEW approved_records AS
SELECT  claim_id, record_name, subclass_name
FROM    record_claims
WHERE   approved;
```

Now we can find each person with any kind of record in any kind of subclass:

```
DROP VIEW IF EXISTS all_individual_records;
CREATE VIEW all_individual_records AS
SELECT DISTINCT  ssn, record_name, subclass_name
FROM             approved_records
JOIN             record_settings
USING            (claim_id);
```

It's time to count how many different kinds of records each person have, and only keep track of those with more than one record:

```
DROP VIEW IF EXISTS records_count;
CREATE VIEW records_count AS
SELECT    ssn, COUNT() AS no_of_records
FROM      all_individual_records
GROUP BY  ssn
HAVING    no_of_records > 1;
```

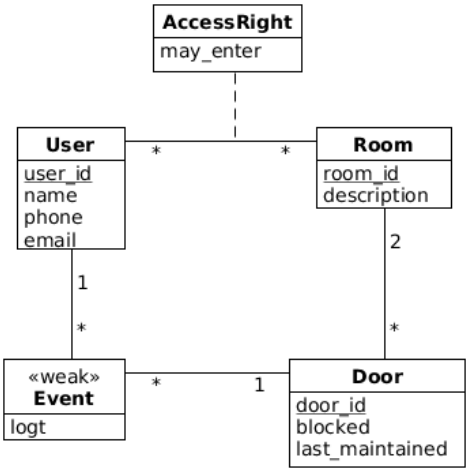
Now the rest is easy, we only need to join in `people` to get the name of the record setter:

```
SELECT    name, no_of_records
FROM      records_count
JOIN      people
USING     (ssn)
ORDER BY  no_of_records DESC;
```

Solution 2

(a) We'll use integers for the door identification, and use the value 0 to denote everywhere outside of the house (so, the rest of the world has `room_id = 0`):

[1] It would have sufficed to show that the left hand sides ware *superkeys* (i.e., supersets of a key), but here they're all keys



(b) The following code works in SQLite:

```
DROP TABLE IF EXISTS access_rights;
CREATE TABLE access_rights (
  user_id    TEXT,
  room_id    INT,
  may_enter  INT DEFAULT (0),
  PRIMARY KEY (user_id, room_id),
  FOREIGN KEY (user_id) REFERENCES users(user_id),
  FOREIGN KEY (room_id) REFERENCES rooms(room_id)
);
```

(c) We need to look through all events, and join in the `doors` table to see which rooms are entered:

```
SELECT    enter_room, logt
FROM      events
JOIN      doors
USING     (door_id)
WHERE     user_id = 'alex' AND enter_room <> 0
ORDER BY  logt;
```

(d) There are several ways of solving this, such as using a left outer join, using a subquery, or using set operations, we'll try these three alternatives:

- Let's start with the solution based on a left outer join. For the outer join we need:
  - a left table with the user id's of everyone with access to room 5,
  - a right table with all events leading into room 5.

  We introduce the view `allowed_into_room_5`:

  ```
  DROP VIEW IF EXISTS allowed_into_room_5;
  CREATE VIEW allowed_into_room_5 AS
  SELECT  user_id, room_id
  FROM    access_rights
  WHERE   room_id = 5 AND may_enter;
  ```

  Next we create the view `events_leading_into_5`:

  ```
  DROP VIEW IF EXISTS walks_into_room_5;
  CREATE VIEW walks_into_room_5 AS
  ```

```
SELECT user_id, logt      -- we don't really need logt... (but want
                          -- something to be NULL in our left join)
FROM   events
JOIN   doors
USING  (door_id)
WHERE  enter_room = 5;
```

And finally it's time for our left outer join:

```
SELECT     user_id
FROM       allowed_into_room_5
LEFT JOIN  walks_into_room_5
USING      (user_id)
WHERE      logt IS NULL;
```

- Using a subquery (and reusing our `allowed_into_room_5`):

```
SELECT user_id
FROM   allowed_into_room_5
WHERE  user_id NOT IN
        (SELECT user_id
         FROM   walks_into_room_5);
```

- Using set operations:

```
SELECT user_id
FROM   allowed_into_room_5
EXCEPT
SELECT user_id
FROM   walks_into_room_5;
```

(e) Here we can use an arithmetic expression (a simple subtraction) in an outer `SELECT`, and the outer `SELECT` uses two subqueries to count the number of people leaving outside (i.e., entering the house), and the number of people entering outside (i.e., leaving the house):

```
SELECT
  (SELECT  COUNT()
   FROM    events
   JOIN    doors
   USING   (door_id)
   WHERE   logt <= datetime('now') AND exit_room = 0)
-
  (SELECT  COUNT()
   FROM    events
   JOIN    doors
   USING   (door_id)
   WHERE   logt <= datetime('now') AND enter_room = 0);
```

(f) We start out by defining a view containing all 'heavily' used doors:

```
DROP VIEW IF EXISTS heavily_used_doors;
CREATE VIEW heavily_used_doors AS
SELECT    door_id
FROM      events
WHERE     logt LIKE '2016-02%'
GROUP BY  door_id
HAVING    COUNT() > 1000;
```

Now we can update our doors accordingly:

```
UPDATE    doors
SET       blocked = 1
WHERE     door_id IN
          (SELECT door_id
           FROM heavily_used_doors);
```

(g) Since we're not using MySQL, it's tempting to use `INTERSECT`, so:

```
SELECT DISTINCT exit_room
FROM    doors
WHERE   NOT blocked
INTERSECT
SELECT DISTINCT enter_room
FROM    doors
WHERE   NOT blocked;
```

This is what we would do in SQLite or PostgreSQL, in MySQL we could instead use a 'nested loop' through our non-blocked doors. As usual we can use a view to make things (arguably) slightly easier:

```
DROP VIEW IF EXISTS nonblocked_doors;
CREATE VIEW nonblocked_doors AS
SELECT door_id, exit_room, enter_room
FROM   doors
WHERE  NOT blocked;
```

and then:

```
SELECT DISTINCT nd1.enter_room
FROM   nonblocked_doors nd1, nonblocked_doors nd2
WHERE  nd1.enter_room = nd2.exit_room
ORDER BY nd1.enter_room;
```

Solution 3

(a) See the official solution on the web site.

Solution 4

SQL is not well suited to 'loop' as required in this problem (using recursive queries, it is actually possible in some DBMS's today, but it's easier to do it in another language).

A simple Java solution (using JDBC) is:

```
public boolean check(String userId,
                     Timestamp start,
                     Timpstamp finish,
                     Connection conn) {
    boolean ok = true;
    PreparedStatement ps;
    try {
        String sql =
            "SELECT exit_room, enter_room " +
            "FROM   events " +
            "JOIN   doors " +
            "USING  (door_id) " +
            "WHERE  user_id = ? " +
            "       AND ? <= logt AND logt <= ?";
        ps = conn.prepareStatement(sql);
```

```
            ps.set(1, userId);
            ps.set(2, start);
            ps.set(3, finish);
            ResultSet rs = ps.executeQuery();
            if (!rs.next()) {
                return true;    // no walk to report
            }
            String src = rs.getString("enter_room");
            while (rs.next()) {
                String dst = rs.getString("exit_room");
                if (!dst.equals(src)) {
                    ok = false;
                    System.out.println("...");
                }
                src = dst;
            }
            return ok;
        } catch (SQLException e) {
            e.printStackTrace();
            return false;
        } finally {
            try {
                ps.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

Solution 5

See official solution (but XML and its validation is not a part of the course this year).

Solution 6

See official solution.