CS275 Final Exam

By Eric Rouse

Question 1

Compare the options:

Option 3 is perhaps most obviously the poorest choice. It happens to be the easiest to pick on because it tries to accomplish everything within a single table. Every kind of vehicle with every possible bit of information necessary is included.

This is not a very good idea for several reasons. For instance, the time a vehicle reaches sixty miles per hour is a completely meaningless value when referring to an airplane, or even most motorcycles. Using this method, you would have a database with large sections not used or filled with meaningless data.

Since it has an entry for every possible value in each row, it uses the most memory out of any of the options. In addition, it would be somewhat hard to follow because of the sheer amount of columns.

The nicest thing you can say about it is that it stores all the data in one table; you don't have any interdependencies to worry about. You don't have to look at several different tables at once.

Option 2 is much better. It breaks the data up into chunks that make sense for the type of vehicle. Each vehicle type has everything related to it in one place.

On the down side, a lot of the similar data is used in similar types of vehicles. The car and truck tables are very similar while the plane is quite a bit different.

After looking at option 2 for a while, one would think that it would be nice if there was a single table that kept all the common data in a central location.

This is what is great about option 1: one 'master' vehicle table that keeps track of the common data. Unfortunately horsepower and name are the only variables common across all known platforms. So you really don't get much "reuse" value from separating the data this way. In fact, it could be argued that really the only data point kept here is vehicle type name. This is nice in one respect: you have a list of all the types of vehicles in one place.

For instance, horsepower makes sense as a "global" variable now, but what about when we start adding other types of vehicles? For instance, motorcycle horsepower is not a very useful piece of information.

Another drawback here is that it uses a foreign key reference, so you have do a join on the vehicle and object tables to find the object's name.

Argue for the best one:

I would go with option 2 for this project. The primary reason is that with the vague specs we have at this point, option 2 is the most flexible. If we end up with more vehicle types we won't be constrained by the foreign key table of option 1.

Option 3 is just bad design. We would be adding columns every time a different type of vehicle cropped up. It really isn't worth serious consideration.

While option 1 merits further investigation, I would reject it based on the following thought experiment:

Consider adding a completely different type of vehicle, like a boat or a motorcycle or an electric car. With option 1 we need to consider the horsepower of each new vehicle type. This is fine for boats, but it is less meaningful for motorcycles. It is completely useless when talking about an electric car.

So, the added overhead of option 1 doesn't add enough benefit in my opinion.

I would probably add a "type" to each vehicle_* to make it easy to search for planes, trucks, cars, boats, etc.

Question 2

Script

(Also submitted as Q2.sql)

```
#create tables, straight from teh question

DROP TABLE IF EXISTS manager2employee;
DROP TABLE IF EXISTS employee;
DROP TABLE IF EXISTS managers;

CREATE TABLE managers(
      id INT PRIMARY KEY AUTO_INCREMENT,
      ssn INT NOT NULL,
      name VARCHAR(255),
      department VARCHAR(255)
)engine=innodb;

CREATE TABLE employee(
      id INT PRIMARY KEY AUTO_INCREMENT,
      ssn INT NOT NULL,
      name VARCHAR(255),
      boss INT,
      FOREIGN KEY (boss) REFERENCES managers(id)
)engine=innodb;

#insert dummy values
INSERT INTO managers () VALUES (1, 1, "man1", "1");
INSERT INTO managers () VALUES (2, 2, "man2", "2");
```

```
INSERT INTO managers () VALUES (3, 3, "man3", "3");
INSERT INTO employee () VALUES (1, 1, "emp1", 1);
INSERT INTO employee () VALUES (2, 2, "emp2", 2);
INSERT INTO employee () VALUES (3, 3, "emp3", 3);


#create the new relationship table
#by copying the manager table
create table manager2employee (
mID INT,
eID INT,
foreign key (mID) references managers (id),
foreign key (eID) references employee (id)
) engine = innodb;

#copy the managers into the new relationship table
INSERT manager2employee SELECT id, NULL FROM managers;

#update the table with the employee/boss info, as it exists presently
UPDATE manager2employee AS m2e
  INNER JOIN employee AS ee ON m2e.mid = ee.boss
SET m2e.eID = ee.id;

#find the foreign key constraint and store it in @const_name
SET @const_name = ( SELECT CONSTRAINT_NAME
FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS
WHERE TABLE_NAME =  'employee'
AND CONSTRAINT_NAME <>  'PRIMARY' );
```

```
#make an alter table statement from the foreign key constraint
SET @s := CONCAT(  "ALTER TABLE employee DROP FOREIGN KEY ",
@const_name ) ;
PREPARE stmt FROM @s ;

#execute the carefully devised statement
EXECUTE stmt;

#all this just to get rid of the boss column
ALTER TABLE employee DROP COLUMN boss;

#add some more dummy relationships
INSERT INTO manager2employee () VALUES (1,3);
INSERT INTO manager2employee () VALUES (3,1);
INSERT INTO manager2employee () VALUES (1,2);
INSERT INTO manager2employee () VALUES (2,3);

# Query to get a list of employees and managers
# The updated version should show all employee manager pairs
SELECT employee.name, managers.name AS `boss` FROM employee
INNER JOIN manager2employee ON manager2employee.eID = employee.id
INNER JOIN managers ON manager2employee.mID = managers.id;
```

Question 3

Discussion

The classes table is bad for several reasons.

First, it is not a great idea (in general) to use a string as a primary key. It can't auto_increment. The list must re-sort every time a new class is added, instead of it being appended to the end. In this case it is even worse because they used a CHAR(10) type which will ALWAYS use 10 bytes of data for every row. If this is a very big database it will take up a lot of unnecessary memory.

Second, if there are two sections of the same class the primary key becomes a huge problem, since both classes have the same name. These two problems can be solved by using a class_id, a unique identification number so that class name isn't so critical. Also, for class_name, use VARCHAR(255).

Next, it doesn't have much in the way of useful information. Most classes have meeting times, reference numbers (CS275 for example) and so on. These should be thought of during the database design so all records have all this data, rather than tacked on later as they are thought up.

The students table is not any better.

This table again uses a poor choice for primary key. Surely there are duplicate student names out there. This table can't handle that. Also, the full name is usually best if broken into the smallest chunks of first_name and last_name. And for the other reasons listed above I recommend the primary key be an auto_increment-ing student_id.

Age is a strange record to keep. Age should be calculated from birthday as it will change over time. For instance, in a couple of years after the entry is placed in the database this age record will be off by two. The birthday should not be a VARCHAR(10), but a DATE.

Another MAJOR problem with this setup is the on delete cascade statement appended to each class the student is enrolled in. Using this statement will delete any student in a class, if the class they are enrolled in is deleted! Yikes! Talk about major unintended data loss.

The classes to students relationship is many to many. Instead of limiting the student to 4 possibilities a third table should be used, a relationship table that pairs student_id's with class_id's. This also solves the on delete cascade dilemma: the row in the relationship table is deleted, as it should be, but the student record is left intact.

All these changes are implemented and shown here in the "Queries" section.

Queries

```
DROP TABLE IF EXISTS students2classes;

DROP TABLE IF EXISTS students;

DROP TABLE IF EXISTS classes;


CREATE TABLE classes(

class_id INT primary key auto_increment,

class_name varchar(255),

class_time TIME,

class_days varchar(255),

class_desc varchar(255),

class_num varchar(255)

)engine = innodb;


CREATE TABLE students(

student_id INT PRIMARY KEY AUTO_INCREMENT ,

fname VARCHAR( 255 ) ,

lname VARCHAR( 255 ) ,

birthday DATE

) ENGINE = INNODB;


CREATE TABLE students2classes(

class INT,

student INT,

FOREIGN KEY ( class ) REFERENCES classes( class_id ) ON DELETE
CASCADE,

FOREIGN KEY ( student ) REFERENCES students( student_id ) ON DELETE
CASCADE

) ENGINE = INNODB;
```