CS352 - DATABASE SYSTEMS

Database Design Project - Various parts due as shown in the syllabus

Purposes: To give you experience with developing a database to model a real domain

At your option, this project may be done by a team of two students.

Requirements

This project is deliberately made quite open-ended. It consists of four parts

1. Choose a domain with which you are familiar - other than one of the domains used for examples in the book (banking) or lecture (a library). (See discussion on "Choosing an Appropriate Domain" below.) Identify a set of requirements for a system that is approriate for this domain. (If you wish, you may choose an appropriate subset of a larger domain.)
2. Develop an E-R diagram for a database that models this domain (or the chosen subset).
3. Turn your E-R diagram into a normalized relational database design for the (subset of the) domain - i.e. a set of tables, each with appropriate attributes, a primary key, and appropriate foreign keys. The database should be a based on the E-R diagram, but one-to-one and one-to-many relationships may be implemented by appropriate attributes in the "one" entity, rather than as separate tables. Of course, your relational database must be 4NF!
4. Implement your database design

   a. Create a file of SQL `create ____` statements.

   b. Create your database under your username (or the username of one of the partners) as a schema in the `design` database (i.e. you won't actually create the database itself, just the tables, etc, within a schema in it.)

   c. Populate your database with sample data to allow testing of the schema and the various transactions. Each table should have a *minimum* of five rows

   d. Create SQL select/insert/update/delete statements that correspond to your initial requirements.

   e. Thoroughly test your queries and your constraints using your sample data. Among other things, this means ensuring that your constraints correctly catch various kinds of invalid insert/update/delete operations, while allowing legitimate ones. Be sure to test using both "good" and "bad" data! Note that the comprehensiveness of your test data will be a factor in the grading of this part of the project!

Turn in:

Your project will be turned in in four parts. You may wish to modify your approach/design for subsequent parts based on professor feedback from each part. (You do not need to redo the prior part unless you totally change the project!) The value of each part in the final course grade will be as shown below (20% total for the whole project.)

On the due date for each of the first three parts, you will present your work orally to your classmates. For the first part, we will also spend some time as a class discussing the next part of the project - e.g. what might be appropriate entities and relationships for modelling this domain. (We may also have brief class discussion on the other two parts, though not as extensively as on the first part.)

To facilitate oral presentation,/discussion you will need to prepare appropriate material for projection, or handouts.

Part I - Requirements (3% of final course grade)

- A description of the problem domain (written using terminology that a user of the system would use, not techical database terminology.
- A statement of requirements. You may express this in "requirements language", or you may find it useful to prepare a use case diagram and perhaps simplified use cases. (See example systems from CPS211).

Part II - E-R Diagram (5% of final course grade)

- An ER diagram for your database, including attributes for entities and relationships.

Part III - Relational Database Design (5% of final course grade)

- A list of the functional and multivalued dependencies for your scheme.
- A schema diagram for your database, with primary and foreign keys specified appropriately.

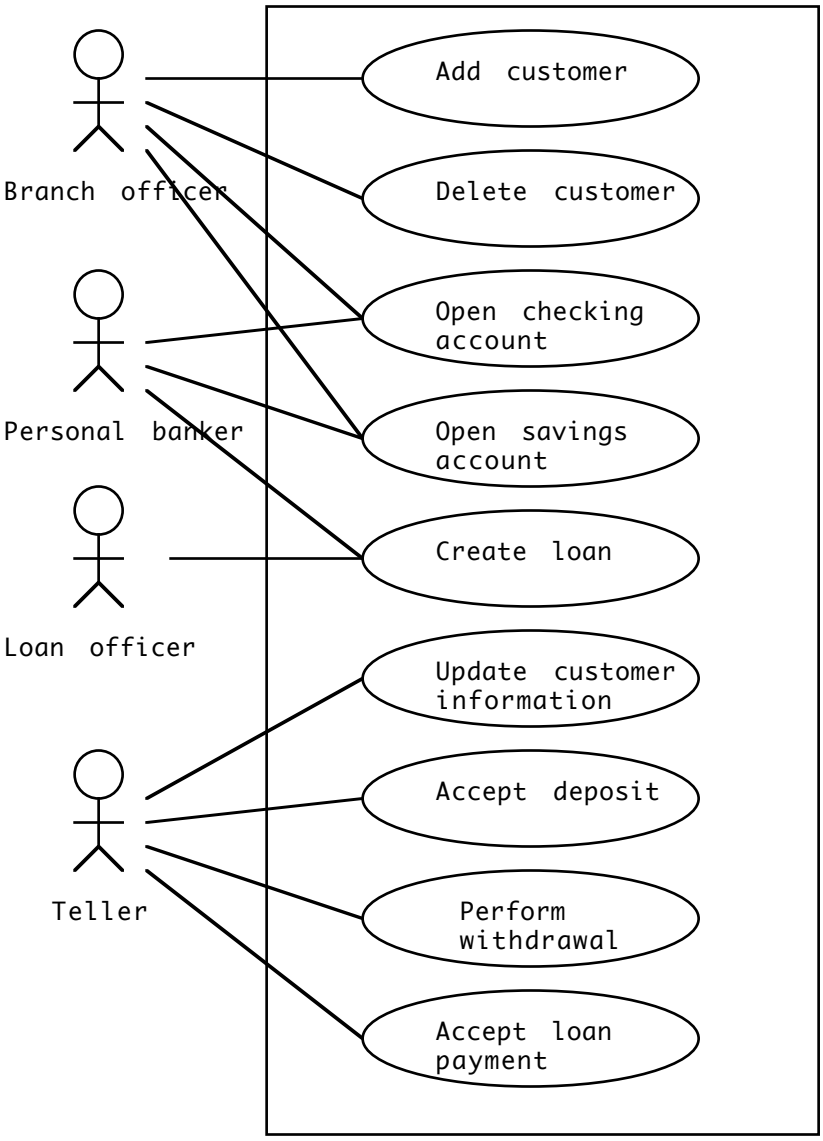
Part IV - Relational Database Implementation/Sample Data (7% of final course grade)

- A printout of a file containing your SQL `"create"` statements that create the database.
- The SQL statements corresponding to each of your requirements. (If your requirements call for more than a dozen or so statements, you can do a subset). Your SQL statements should require a variety of SQL capabilities, such as varous kinds of join, aggregate functions, etc. [This presupposes a good initial domain choice, of course!]
- The test data you used for testing. You must exercise each of your SQL statements, and must also exercise each of your constraints, being sure it correctly catches errors while allowing legitimate data. In the case of testing constraints, you should include some comments indicating what specific problem you were testing for with "bad" data - but be sure to include "good" data as well! (Note: you do not need to test `not null` constraints.)
- Printouts of the sample data contents of the various tables (the results of `select * from ...`) and of the results of your tests indicating that the SQL statements for your requirements worked correctly and that your constraints correctly allowed good data and caught bad data. (You can interweave some of this with #3 above if you wish.)

An Example:

(Note: this example is **much** simpler than what you would actually do for the project - it's only meant to illustrate what the various pieces of the project need to include.)

1. Suppose you chose a subset of the domain of banking, as in the examples in the book. (As noted above, you can't actually choose this domain, of course!) Section 1.6.2 in the book describes this system; you would also need to spell out the requirements, perhaps by using a use case diagram like that shown on the next page. (Cases pertaining to managing the set of branches and to managing information about employees are excluded, as is any consideration of access via an ATM)



2. Your design might be based on an E-R diagram like figure 6.25 in the book. (Note: other examples in the book use the customer name as the primary key, rather than having a separate `customer_id` attribute. This is actually not a good idea, so we will follow the approach of the other book examples to avoid confusion! Therefore, pretend that the E-R diagram does not have a `customer_id` attribute. We'll also omit features of the E-R diagram pertaining to employees, different types of accounts and loan payments from the rest of this example.)

3. The following FD's hold on the E-R diagram, modified/abbreviated as noted above.

```
branch_name  →  branch_city, assets
customer_name  →  customer_street, customer_city
account_number  →  branch_name, balance
loan_number  →  branch_name, amount
customer_name  ->>   account_number
account_number  ->>  customer_name
customer_name  ->>  loan_number
loan_number  ->>  customer_name
```

A database schema based on these dependencies might look like figure 2.8 in the book. (Note: this does not correspond exactly to the E-R diagram - the book authors have not tried to develop a single example consistently throughout the book!) No tables are needed corresponding to account_branch or loan_branch in the E-R diagram - attributes of these relationships can be folded into the account and loan tables because each relationship is 1:many.

4. • This database could be created by the following SQL statements.
   • Named domains are created for each of the primary keys of the four entities, plus one for representing money amounts.
   • DB2 automatically creates indexes for each primary key, so we don't need to do this here (but on some systems it might be necessary to explicitly do so.) We also index depositor and borrower by loan_number (but not by customer_name - assume we most often lookup depositor and borrower information on accounts and loans by number.)
   • When an account is deleted, the specification of cascading delete in the references clause of depositor causes the relevant depositor row(s) to also be deleted. The same holds for loan and borrower.
   • A trigger is used so that when a loan is deleted (which cascades to delete the borrower row(s)), the borrower(s) is/are also deleted from the customer table if they have no other outstanding loans and are not depositors. (Something similar could also be done for accounts)>
   • A view is created to allow clerical employees of the Perryridge branch to access full customer information on all the borrowers who have loans at the branch, but not details about the loan.
   • For this problem, it is reasonable to constrain all attributes to be not null. (This would not necessarily always be the case - in some cases it is meaningful for a particular attribute to be null).

```
create distinct type branch_name_t as char(10) with comparisons;
create distinct type customer_name_t as char(30) with comparisons;
create distinct type account_number_t as integer with comparisons;
create distinct type loan_number_t as integer with comparisons;
create distinct type money_t as decimal(12, 2) with comparisons;

create table branch(
   branch_name branch_name_t not null primary key,
   branch_city char(30) not null,
   assets money_t not null);

create table customer(
   customer_name customer_name_t not null primary key,
   customer_street char(30) not null,
   customer_city char(30) not null);

create table account(
   account_number account_number_t not null primary key,
   branch_name branch_name_t not null references branch,
   balance money_t not null);

create table loan(
   loan_number loan_number_t not null primary key,
   branch_name branch_name_t not null references branch,
   amount money_t not null);

create table depositor(
   customer_name customer_name_t not null references customer,
   account_number account_number_t not null references account
         on delete cascade);
```

```
create table borrower(
   customer_name customer_name_t not null references customer,
   loan_number loan_number_t not null references loan
         on delete cascade);

-- db2 automatically creates the primary key indexes for three tables
create unique index dep_acct_no_index on depositor(account_number);
create unique index borr_loan_no_index on borrower(loan_number);

create trigger del_if_not_cust
   after delete on borrower referencing old as delrow
    for each row mode db2sql
     when (delrow.customer_name not in (select customer_name from borrower)
         and delrow.customer_name not in (select customer_name from depositor))
          delete from customer
             where customer.customer_name = delrow.customer_name;

create view perryridge_borrowers as
   select customer.*
       from (loan join borrower on loan.loan_number = borrower.loan_number)
         join customer on customer.customer_name = borrower.customer_name
       where branch_name = cast('Perryridge' as branch_name_t);
```

• This example will not include SQL for all requirements - but just to illustrate the point

- For the requirement "Open savings account":

```
insert into account values(account number, branch, initial deposit)
```

• This example will not include test data for all requirements and constraint - but just to illustrate the point, the following would test the primary key and foreign key constraints on `account` (assuming that the `Perryridge` branch was already created.)

```
insert into account values(12345, 'Perryridge', 100.00);
insert into account values(12345, 'Perryridge', 200.00);        (fails PK)
insert into account values(98765, 'No such', 100.00);           (fails FK)
```

Documentation for this test would consist of a screen shot showing DB2 accepting the first insert and giving an error message for each of the two invalid inserts.

The trigger could be tested by the following:

```
insert into customer values ('Aardvark', 'Jenks subbasement', 'Wenham');
insert into loan values (1234, 'Perryridge', 100.00);
insert into borrower values ('Aardvark', 1234);
delete from borrower where loan_number = cast (1234 as loan_number_t);
```

the last statement should also result in the deletion of the `customer` row for `'Aardvark'` - i.e. a `select * from customer` should show a row for `Aardvark` before the deletion is done, but not after. Documentation for this test would consist of a screen shot showing the result of both select statements, with the delete statement between them.

Choosing an Appropriate Domain

The domain (or subset of a domain) that you choose for your project should have the following characteristics:

1. Size. An appropriate-size domain will result in a database having about a dozen tables (more or less).

2. The entities comprising your domain should be interrelated.

3. Your schema should include some attributes which make it possible to include some transactions that involve aggegate functions. (For example, the schema developed above would allow for a queries to calculate the total of all loans for each branch, or the average loan size of each branch, or the total of all loans for each customer, etc.), and should also make interesting constraints and triggers possible.

   Important: look over the requirements for Part IV when choosing a domain, to ensure it will allow you to do a good job on the final part!