TDA357/DIT620 Databases lab assignment

Last updated: February 27, 2017

Contents

1 Introduction

The introduction was last updated on 20 January 2017

1.1 Purpose

The purpose of this assignment is for you to get hands-on experience with designing, constructing and using a database for a real-world domain. You will see all aspects of database creation, from understanding the domain to using the final database from external applications.

The final result of this assignment is a playable computer game. The game is set in a world with towns and cities, which are interconnected by roads. From time to time, money is awarded to the first player that reaches a certain city or town. Players can build roads and hotels. Traveling across a road costs money, and so does staying at a hotel. More details about the game-play of this game will be provided in one of the later tasks.

1.2 Structure

This assignment consists of four distinct tasks:

Task 0: Getting started This task gets you set up to successfully complete this lab assignment by finding a lab partner, registering in Fire and getting PostgreSQL credentials.

Task 1: Designing the database schema From the domain description, you will create an E/R diagram which correctly models the domain model and translate it into a relational database schema.

Task 2: Constructing the database From the E/R diagram and relational database schema, you will implement the database schema in the PostgreSQL database engine.

Task 3: Writing a front end application Based on a working database implementation in PostgreSQL, you will write a front end application in Java using JDBC that communicates with the database.

Contrary to previous lab assignments for this course, a task does not build on the solution of the previous task.
For instance, do not assume that the database schema you design in Task 1, will be the database schema you implement in Task 2.

To make it easier to grade, we will select a good solution for a given task and ask all teams to start from that.
For instance, after the deadline for Task 1, we select a good solution for Task 1 and distribute it to everyone. Task 2 will then consist of implementing the database schema we provide, instead of your own.

1.3 Assignment submission and deadlines

To pass the programming assignment, you must pass all four tasks described in this document. You will do the assignment in groups of two.

You must submit your solutions through the Fire reporting system. You must submit your groups solutions to each task by the given deadline. Each task will list what files to submit to Fire.

Each task has a first deadline and last deadline. You should submit a solution by the first deadline, even if it is incomplete. Not submitting a solution by the first deadline means you fail that task (and thus the lab).

After submission, your assignment will be graded (pass or reject) and you will receive comments on your solution. If your submission is rejected, you are allowed to refine your solution and re-submit it until the last deadline. After the last deadline, no more solutions for that task are accepted.

To reduce the TAs workload for grading lab assignments, some tasks will be automatically graded. This means that your Fire submission is processed using an automated tool which runs a collection of tests on your submission. This automated tool will report any problems found with your submission, just like a human grader would. Since this is a new way of grading for this course, we expect that there may be some issues along the way. Please give us some leeway on this.

The advantage of automated grading is that you receive feedback on your submissions a lot faster.

2 Task 0: Getting started

Task 0 was last updated on 20 January 2017

Deadline for first submission:	**Sunday 22 January 2017**
Deadline for last resubmission:	**Sunday 12 March 2017**

2.1 Your job

Find a lab partner This lab should be completed in teams of 2. Exceptions can be made for extraordinary reasons, after consulting with the examiner. Do not wait until the last minute: the sooner you find a lab partner, the sooner you can start working on Task 1.

Register in the Fire system We use the Fire system to handle submissions and grading of the tasks in this lab assignment. You need to sign up to Fire in order to register your lab team, and later, submit your solutions for the different tasks. Read the separate notes on registering in the Fire system and submitting assignment tasks [3].

Get your PostgreSQL credentials Each lab team has its own PostgreSQL username and password. These credentials are distributed through the Fire system, after you have registered your lab team and submitted Task 0 in Fire (send in an empty file). You will need these credentials for Tasks 2 and 3. Read the separate notes on requesting your groups PostgreSQL account [3].

2.2 What to submit

Just an empty file. You will receive the PostgreSQL credentials as a reply.

3 Task 1: Designing the database schema

Task 1 was last updated on 20 January 2017

Deadline for first submission:	**Sunday 5 February 2017**
Deadline for last resubmission:	**Sunday 12 March 2017**

3.1 Domain description

The domain to model is that of a fictional board game.

Countries Countries consist of areas which are either cities or towns. Countries have unique names.

Areas Areas can be interconnected by roads. There can be many roads between 2 areas. Areas have names that are only unique in the country they are located in. Areas also have a population size associated with them.

Roads Roads can be owned by a person, or be a public road (not owned). Roads have a roadtax associated with them.

Hotels Hotels have names, but they are not unique in any way. Hotels also have a price per night.

People A Person is always a citizen of a country. People travel around, but are always located in a certain city or town (an area). People can own hotels, which are located in cities (not towns). A person can own multiple roads, but cannot own more than 1 road between 2 areas. A person can own maximum 1 hotel in a city. Persons have a name and a personnummer. A person's name is not unique, but the personnummer is unique in their country. Each person can also speak several languages and have a budget.

3.2 Your job

Your first task is to design the database that your application will use. The goal of this task and the next is to reach a correct database schema that could later be implemented in PostgreSQL.

E/R diagram You are to create an E/R diagram that correctly models the domain described above. Hint: if your diagram does not contain (at least) one weak entity, (at least) one ISA relationship, and (at least) one many-to-at-most-one relationship, you have done something wrong. You can use any tool you like for this task, as long as you hand in your solution as an image in one of the formats .png, .jpg, .gif or .pdf. The tool Dia [1] is available on the school computer system, has a mode for ER diagrams, and can export diagrams to image files, but using any other tool is also OK.

Relational schema When your E/R diagram is complete, you should translate it, using the (mostly) mechanical translation rules [2], into a database schema consisting of a set of relations, complete with column names, keys and references.

Functional dependencies Identify the functional dependencies that you expect should hold for the domain. Do this starting from the domain description. Do not only look for functional dependencies in your relation schemas! If you do, then any potential errors in your schema would not be caught, nor will you find any extra constraints not already captured by the relational structure! At least one such constraint exists in the domain. You should always look for the functional dependencies in the domain description. Remember to check for functional dependencies with multiple attributes on the left-hand side of the arrow.

Update based on constraints Update your relational schema with the extra constraints discovered from your functional dependencies. Alternatively, argue why your schema should not capture the constraint (see e.g. the difference between 3NF and BCNF for why not all constraints can be captured). Further, do the same for any constraints needed to handle cyclic relationships in your diagram.

3.3 What to submit

You should submit 4 files. Files with a ".txt" suffix are text files.

diagram.X your E/R diagram, where .X is one of .png, .jpg, .gif or .pdf.

schema1.txt the database schema that you get from translating the diagram into tables. Clearly mark keys (e.g. _key_). If any attribute can be null, indicate so by writing (nullable) or similar next to it. If you have made any non-obvious choices, when choosing which translation approach to use (e.g. ER or Null) include a comment about why you decided on using it.

fds.txt the full list of functional dependencies that you have identified for the domain.

schema2.txt the database schema that you get after updating your schema1.txt with any added constraints found because of the functional dependency analysis.

4 Task 2: Constructing the database

Task 2 was last updated on 1 February 2017

Deadline for first submission:	**Sunday 19 February 2017**
Deadline for last resubmission:	**Sunday 12 March 2017**

4.1 Description

The previous task modeled a world with countries, cities, towns, roads, hotels and people. This world is the stage for a game where some rules apply.

In this task, you will implement part of this game as a PostgreSQL database with tables, views and triggers. To avoid unnecessary overhead because of wildly divergent solutions, everyone will start Task 2 from this given relational database schema:

```
Countries(name)

Areas(country, name, population)
    country -> Countries.name

Towns(country, name)
    country, name -> Areas.(country, name)

Cities(country, name, visitbonus)
    country, name -> Areas.(country, name)

Persons(country, personnummer, name, locationcountry, locationarea, budget)
    country -> Countries.name
    locationcountry, locationarea -> Areas.(country, name)

Hotels(name, locationcountry, locationname, ownercountry,
   ownerpersonnummer)
    locationcountry, locationname -> Cities.(country, name)
    ownercountry, ownerpersonnummer -> Persons.(country, personnummer)

Roads(fromcountry, fromarea, tocountry, toarea, ownercountry,
   ownerpersonnummer, roadtax)
    fromcountry, fromarea -> Areas.(country, name)
    tocountry, toarea -> Areas.(country, name)
    ownercountry, ownerpersonnummer -> Persons.(country, personnummer)
```

Please note that this relational database schema is different from Task 1's solution.

4.1.1 Predefined constants and functions

The game uses a number of constants (such as the price of a road) that should not be hard-coded into your solution. Instead, we provide an SQL file with an SQL table (called `Constants`) with constants and their values, and a handy SQL function (called `getval()` to access them.

You should use the `getval()` function whenever you require a certain constant in a computation.

For instance, if you wish to decrease a Person's budget with the price of a road, you should use the following statement which retrieves the constant `'roadprice'` using the function `getval()`:

```
UPDATE Persons SET budget = budget - getval('roadprice') WHERE ...
```

To facilitate testing, we also provide a function `assert()` which compares two given values. If the values match, nothing happens. If they do not match, an exception is raised. The `assert()` function can be used in your unit tests to ensure that a computed value is what you expect it to be.

For instance, if you want to ensure that a Person's budget is 123.45, you can use the following `assert()` invocation:

```
SELECT assert( (SELECT budget FROM Persons WHERE ...), 123.45 );
```

(Note: `assert()` can not be used by itself in SQL, since it is not a valid SQL statement. You have to use it as part of a `SELECT` statement.)

The SQL file with predefined table (`Constants`) and functions (`getval()` and `assert()`) will be preloaded into your database during testing (See further), and the file is also available on the website for your own needs.

4.1.2 Game rules

In our little game world, there are some rules that define how Persons can interact with the world. For this Task, these rules are important:

Traveling over roads costs money if the person travels over a road owned by another person, the roadtax associated with that road is deducted from the traveling person's budget and added to the budget of the owner of the road. If the road is public, or owned by the traveling person, no money is transferred.

Creating roads costs money Persons can create a new road between the area they are currently in, and any other area, as long as they do not already own a road between those two areas. When a new road is created, the price of a road (`getval('roadprice')`) is automatically deducted from the owner's budget. The roadtax for new roads is set to `getval('roadtax')`.

Visiting a city can cost money If a person visits a city with hotels in it, the visiting person must pay money (`getval('cityvisit')`). That money is distributed equally among all people who own a hotel in that city. It does not matter whether a person visiting a city owns a hotel in that city, the money must still be paid and distributed to the hotel owners (i.e. the visiting person also gets a part of that money if he owns a hotel in that city). If there are no hotels in the city, then the visiting person does not have to pay.

Persons can buy/sell a hotel Each person can buy one hotel per city. Hotels cost a fixed amount (`getval('hotelprice')`), which is deducted from the Person's budget. Persons do not need to be in the city they wish to build a hotel in. When a Hotel is sold, the owner gets back a fraction of the original price (`getval('hotelrefund')`, a value between 0 and 1, indicates how much to refund of the hotel price. E.g. 0.4 means 40%).

Cities can offer visiting bonuses From time to time, a city may issue a "visiting bonus" to attract visitors. This bonus is the visitbonus attribute of a city. The first person to reach the city receives the full bonus. Any persons already in the city when the bonus is issued, receive nothing. When a city issues a bonus to a visitor, the city's visitbonus is transfered fully to the Person's budget. A city's visiting bonus is increased through the course of the game, but more on that in Task 3.

4.1.3 Personnummers

In contradiction to reality, we will assume that all countries use a personnummer with the exact format "XXXXXXXX-XXXX", where each X is a digit (0-9).

4.1.4 Negative numbers

Keep in mind that budgets, citybonus, population and roadtax can not be negative.

4.1.5 Unidirectional roads

In the game, roads are bidirectional (you can travel over them in both directions), but the roads in the Roads table are unidirectional (they have a distinct start and end). For a certain bidirectional road (denoted as `A--B`), we will not have 2 unidirectional roads in the Roads table (denoted as `A->B` and `B->A`), but only one. Whenever we deal with roads, our program logic will take into account the reverse road too.

4.1.6 Rounding

Unlike in mathematics, working with floating point numbers (real numbers with decimals after the decimal point) is not always exact. Because of this, we consider two numbers equal when they are equal up to two decimals after the decimal point. For instance, for our purposes we consider **15.78**1641365 equal to **15.78**52313453.

We use SQL's built-in `trunc()` function for this, as you can see in the definition of our predefined `assert()` function.

4.1.7 The government (public roads)

Roads can be public or owned by a Person. Because of our design, all roads must have an owner, including public roads.

To get around this, we introduce "The Government" as an articial Person living in Country `' '` (the empty string) and with Personnummer `' '` (the empty string). Note that this violates the constraint about the format of a personnummer, and you should make an exception for this specific case.

The government has an infinite budget which you do not need to be concerned about. Whenever money is about to added or removed from The government's budget, you can just skip that operation.

Because this artificial person can only be created after the `Persons` table is created (which is your job), you must also create this person into that table.

You may use the following statements to insert this artificial person (Note that we create an Area, since all Persons are located in an Area):

```
INSERT INTO Countries VALUES('');
INSERT INTO Countries VALUES('Sweden');
INSERT INTO Areas VALUES('Sweden', 'Gothenburg', 491630);
INSERT INTO Persons VALUES('', '', 'The␣government', 'Sweden', '
    Gothenburg', 100000000000);
```

4.2 Your job

You should create all tables, marking key and foreign key constraints in the process, and you should also insert checks that ensure that only valid data can be inserted in the database. Examples of invalid data would be a negative population for an Area.

Make sure that you follow the naming scheme of the tables and the attributes, as well as the order of the attributes exactly.

Budgets and roadtax should not be stored as whole numbers. It is best to use the SQL type `NUMERIC` for them.

We advise that you implement the views before the triggers, and that you implement and maintain a set of unit tests as you progress.

4.2.1 Views

Create the following views:

NextMoves Gives an overview of the possible destinations a Person can go to, together with the cost. This VIEW can be described as:

```
NextMoves(personcountry, personnummer, country, area, destcountry,
    destarea, cost)
     personcountry, personnummer -> Persons.(country, personnummer)
     destcountry, destarea -> Areas.(country, name)
     country, area -> Areas.(country, name)
```

{`personcountry, personnummer`} indicates the Person, {`country, area`} the current location of that person, {`destcountry, destarea`} the possible destination and {`cost`} the lowest cost to get to that destination. Keep in mind that Roads have a start and end-point, but that it is possible to traverse them in the opposite direction too!

No `INSERT`, `DELETE` or `UPDATE` operations are allowed on this view.

AssetSummary Gives an overview of each person's budget and assets. This VIEW can be described as:

```
AssetSummary(country, personnummer, budget, assets, reclaimable)
     country, personnummer -> Persons.(country, personnummer)
```

{`country, personnummer`} indicates the Person, {`budget`} the current budget of that Person, {`assets`} the summed price of all Roads and Hotels owned by that Person, {`reclaimable`} the total value that can be reclaimed by selling all hotels owned by that Person.

No `INSERT`, `DELETE` or `UPDATE` operations are allowed on this view.

4.2.2 Triggers

INSERT/DELETE/UPDATE Roads When a road (`A->B`) is added, you must ensure that the reverse road(`B->A`) is not already present for the same owner. Likewise, if a road (`A--B`) is about to be deleted, either `A->B` or `B->A` may be present in the Roads table. You should remove whichever one of those is present.

When a road is created by a Person, ensure that that Person is located at either the start-point or end-point of that road, and deduct the price of the road (`getval('roadprice')`) from that Person's budget.

Finally, make sure that only the roadtax field in a road can be updated, since the game does not allow a road to change the start-point, end-point or owner.

UPDATE Persons A Person is always located in an Area. When a person moves, you must ensure that there is a road between the old and new area that Person is in. If there are multiple roads, use a free one if possible (a public road, or a road owned by that person), otherwise find the cheapest road and deduct the roadtax from the Person's budget before updating the Person's location to the new area.

When a person moves to a city, and there are hotels in that city, deduct `getval('cityvisit')` for visiting a city from the Person's budget and transfer that money equally to all hotelowners that have a hotel in that city.

Finally, if a city has a visiting bonus, transfer it to the visiting Person.

INSERT/UPDATE Hotels When a hotel is created, the price of the hotel must be deducted from that Person's budget. Hotels can not be moved to a new city, but they can change owner. Keep in mind that a Person can only own one Hotel per City. Persons can sell their hotel, in which case the hotel is deleted from the Hotels table. When that happens, the Person get refunded with a fraction (`getval('hotelrefund')`) of the price of the hotel (`getval('hotelprice')`).

4.2.3 Unit tests

You should write unit tests to ensure that your database works as expected. This can be time-consuming, but it is an important part of the development of a database. Each unit test is a set of `INSERT`, `UPDATE` and/or `DELETE` statements, which fill the database with example data. Unit tests should always assume an empty database (except for the predefined table and functions discussed in Section 4.1.1).

Suggestion: Start a transaction at the beginning of a unit test and rollback at the end.

There should be unit tests to test success (E.g. creating a country with a valid name should work), but also failure (E.g. creating a road with the same start-point and end-point should fail).

It is a good idea to implement unit tests first, describing behavior that should or should not work, before implementing the database schema.

4.3 What to submit

You should submit a single ZIP file with the following contents:

a file named `task2.sql` The SQL definition of your database, including all views and triggers.

a directory named `shouldwork` A directory with .sql files containing your unit tests that should result in success.

a directory named `shouldfail` A directory with .sql files containing your unit tests that should result in failure.

5 Task 3: Writing a front end application

Task 3 was last updated on 6 February 2017

Deadline for first submission:	**Sunday 5 March 2017**
Deadline for last resubmission:	**Sunday 12 March 2017**

5.1 Your job

Your job for Task 3 is to implement the game in Java. We provide a skeleton program that you should complete.

5.1.1 Game loop

The game goes through these steps:

1. Read a world definition file and create the world in the database.
2. Read player names and enter them in the database. Each player starts in a randomly selected area in the world and receives a start budget of 1000.
3. Repeat YYY times:
   (a) For each player, in the order they were registered: allow the player to inspect the world and other players as much as wanted, and allow either 1 move, 1 purchase (build a road or hotel) or 1 sale (sell a hotel). A player is not required to make a move, purchase or sale.
   (b) Finally, a visiting bonus of ZZZ is added to a randomly selected city.
4. The player with the highest budget wins the game.

5.1.2 World file format

Instead of hard-coding a fixed world into the game, the game will read a text-file containing a "world description", that is: a list of towns, cities and roads between them.

The world definition file contains a list of statements, one per line. These statements can be any of the following:

```
TOWN <town name> <population>
CITY <city name> <population>
ROAD <start town or city name> <end town or city name>
```

To make it easier for you, assume there are no spaces in the town or city names.

An example world definition with only 2 areas and 1 road is the following:

```
TOWN Sigtuna 8444
CITY Gothenburg 549789
ROAD Sigtuna Gothenburg
```

5.1.3 Player turn

During each turn, the player can inspect the world and other players, as well as make a single move, purchase or sale.

Show possible moves Give a list of directly-reachable destinations and the road-tax associated with it, for a player-specified area. Not specifying an area means the player wants to see this information for the area he is in right now.

Show player assets Give a list of roads and hotels owned by a certain player, or for the current player if no player is specified.

Show scores Display an overview of all players with their current budget, assets and the refunded value if all that player's hotels are sold.

5.2 What to submit

You should submit a single ZIP file with the following contents:

a file named `task3.sql` The SQL definition of your database, including all views and triggers, as implemented (and accepted) from Task 2.

a file named `Game.java` The Java program to play the game. There should be a Game class with a main() method to start the game, just like in the skeleton program.

References

[1] Dia. `http://www.cse.chalmers.se/edu/course/TDA357/VT2017/lab/dia.html`.

[2] ER diagram translation rules. `http://www.cse.chalmers.se/edu/course/TDA357/VT2017/_downloads/Translation.pdf`.

[3] How to use Fire, submit solutions and request PostgreSQL credentials. `http://www.cse.chalmers.se/edu/course/TDA357/VT2017/lab/submission.html`.