CS27020 Assignment – AberTutors

Veronika Karsanova, vek1@aber.ac.uk

7th of November 2019

Contents

1 Modelling the problem

1.1 UNF

Student	Name	Contact details	Subject	Location
0 (False)	Jackie	01970666543	Maths, English	Aberystwyth
0 (False)	Jeff	jeff@stemtutors.net	Maths, Science	Machynlleth, Talybont, Bow Street
0 (False)	Ahmed	07723456788	Cymraeg, Art	Borth, Aberystwyth, Bow Street, Llandre, Talybont
0 (False)	Dafydd	07845333444	Cymraeg	Machynlleth
0 (False)	Peter	pete@languagetutors.com	English	Bow Street
0 (False)	Brian	brian@arttutors.org	Art	Aberystwyth, Llanrhystud
1 (True)	Elin Davies	elin@example.com	Maths, Science	Machynlleth
1 (True)	Steve Smith	steve@example.com	English	Llanrhystud

Figure 1: All the data in UNF

1.2 Attributes and domains

Header for the figure 1 is represented as such:

$$\{(Student : Boolean), (Name : String(30)), (Contact\ details : String(40)),$$
$$(Subject : String(30)),(Location : String(100))\}$$

Student attribute is of type *boolean* (BOOL) and returns 'true' (yes) if the tuple belongs to a student and 'false' (no) if it belongs to a tutor. This allows us to avoid null values throughout the table, unites all the information together (which is the goal, when it comes to UNF) and, in fact, is not violating any properties, since it is not a surrogate key: we cannot use given attribute as a primary key and it is data that we will only use for the sake of UNF and further normalisation.

Name attribute is represented by a *String* (VARCHAT) of 30 characters, and in UNF is representing either a name (for tutors) or a name with a surname (for students). Hence, the string is purposefully being assigned to be quiet long in order to allow some particularly long names.

Contact details attribute stores information about telephone numbers or e-mails addresses of both tutors and students. It allows input of type *String* (VARCAHAR) up to 40 characters long, again, to allow even the longest of e-mails to be properly processed.

Subject attribute is in charge of storing either all the subjects being taught (in the case of person being a tutor) or the subjects that are wanted (if person is a student). *String* (VARCHAR) is nominated as the domain of choice once again, but this time its length does not exceed 30 characters, since there is no such subject which would require more space than that.

Location attribute in UNF is used to store all the cities in which tutors are willing to teach or the cities in which students are located. It is allowed to store *String* (VARCHAR) up to 100 charactes long, because due it the table currently being unnormalised, we are storing the towns in one line, hence occasionally it is more than one or two towns per tutor.

2 Candidate keys and functional dependencies within the data

2.1 Relational keys

Relational keys are the main component to ensure the uniqueness within relational data model.

Superkeys – the set of all present attributes $\{Student, Name, Contact\ details, Subject, Location\}$ defines a superkey of the relation. Other possible superkeys are $\{Name, Contact\ details, Subj$ as well as $\{Name, Contact\ details, Location\}$.

Candidate keys – were chosen accordingly and are $\{Name, Contact\ details\}$, $\{Contact\ details, Subject\}$, since any information on its own would not define uniqueness of the record.

Primary key – $\{Name, Contact\ details\}$ is nominated to be a *composite primary key*. Together attributes will uniquely identify tuples in the relation. No attribute on its own in this data model can promise us uniqueness: name can repeat, so do contact details (landlines as an example), tutors can teach in the same locations and same subjects, and it is also relevant to the students. That is why it is required to use a composite primary key.

2.2 Functional dependencies

If we know the name of the person and their contact details we can ensure that we are talking about a unique entrie (specific tutor or student, if we are talking about UNF) in the database. Knowing that, we can tell which subject is it that tutor is teaching or what is the subject that student wanted.

$$Name, Contact\ details \rightarrow Subject$$

Same goes to the Location. Knowing name and contact details, we can tell in which location is it that tutor is teaching or where student is located.

$$Name, Contact\ details \rightarrow Location$$

3 From UNF to 3NF: an account of the normalisation process

3.1 1NF

In order to ensure that our data base is in 1NF we need to make sure that three requirements are being fulfilled. Requirements are as follows: the value of each column is atomic; there are no repeating groups (meaning, two columns do not contain similar information in the same table); each new table has an assigned primary key.

Let's now make sure that the 4 tables we ended up with are fulfilling needed requirements, discuss foreign keys and primary keys as well as functional dependencies once again.

So, first thing we needed to do is to place all the items that appear to be non-atomic in a new table. In our case, two attributes had such problems: location and subject. Hence, it was decided to put them in separate tables: see figures 2(c) and 2(d). From the first glance one could thing that there would be no problem in making one table for both of the attributes: unfortunately, that would lead to repeating groups, since, for example, same subject can be taught in the same town by two different people.

Alongside the new formed tables (or better say lists at this stage), we needed to duplicate in the new tables the PK of the table from which the repeating group was extracted (meaning, from our UNF table from figure 1). Since our primary key is composed: we duplicating both name and contact details into the location and into the subject tables. Notice, that now this is our so called *foreign key* and our headers are as follows.

Location table:

$$\{(Name : String(30)), (Contact\ details : String(40)), (Location : String(40))\}$$

Students	
Name	**Contact details**
Elin Davies	elin@example.com
Steve Smith	steve@example.com

(a) Student table

Tutors	
Name	**Contact details**
Jackie	01970666543
Jeff	jeff@stemtutors.net
Ahmed	07723456788
Dafydd	07845333444
Peter	pete@languagetutors.com
Brian	brian@arttutors.org

(b) Tutor table

Location Table		
Name	**Contact details**	**Location**
Jackie	01970666543	Aberystwyth
Jeff	jeff@stemtutors.net	Machynlleth
Jeff	jeff@stemtutors.net	Talybont
Jeff	jeff@stemtutors.net	Bow Street
Ahmed	07723456788	Borth
Ahmed	07723456788	Aberystwyth
Ahmed	07723456788	Bow Street
Ahmed	07723456788	Llandre
Ahmed	07723456788	Talybont
Dafydd	07845333444	Machynlleth
Peter	pete@languagetutors.com	Bow Street
Brian	brian@arttutors.org	Aberystwyth
Brian	brian@arttutors.org	Llanrhystud
Elin Davies	elin@example.com	Machynlleth
Steve Smith	steve@example.com	Llanrhystud

(c) Location table

Subject Table		
Name	**Contact details**	**Subject**
Jackie	01970666543	Maths
Jackie	01970666543	English
Jeff	jeff@stemtutors.net	Maths
Jeff	jeff@stemtutors.net	Science
Ahmed	07723456788	Cymraeg
Ahmed	07723456788	Art
Dafydd	07845333444	Cymraeg
Peter	pete@languagetutors.com	Englsih
Brian	brian@arttutors.org	Art
Elin Davies	elin@example.com	Maths
Elin Davies	elin@example.com	Science
Steve Smith	steve@example.com	English

(d) Subject table

Figure 2: The tables after 1NF

Subject table:

$$\{(Name : String(30)), (Contact\ details : String(40)), (Subject : String(30))\}$$

Next step would be to identify the primary keys for the new tables. The tricky bit here is that, essentially, since we do not have that much attributes to choose from, we need to use another composite key. That means that the composite primary key for the location table is: $\{Name, Contact\ details, Location\}$, while composite primary key for the subject table is $\{Name, Contact\ details, Subject\}$.

It is also worth clarifying that the in order to avoid use of a natural key $(Student : Boolean)$, we now have two tables to store information about names and contact details: one for students and another one for tutors (see figures 2(a) and 2(b)). Somehow, I did not find that affecting any of the predescribed functional dependencies. The only difference is due to the performed normalisation and need to connect tables in between each other, which means that for the subject table our FD is:

$$Name, Contact\ details \rightarrow Name, Contact\ details, Subject$$

And for Location table our FD is:

$$Name, Contact\ details \rightarrow Name, Contact\ details, Location$$

3.2 2NF

In order to ensure that our database is in 2NF we need to make sure that two requirements are being fulfilled. Requirements are as follows: the database is in 1NF (we ensured that in section 3.1); all non-prime attributes (attributes that are not part of CK) in table are fully functionally dependent on primary key.

If our current database is not in 2NF, then we would need to move the data item which is violationing 2NF and part of the PK to a separate table in order to normalise the DB.

But what we encounter in this case is that our current 4 tables (see figure **??**) are already in 2NF. Database is in 1NF and there are no non-prime attributes that are not fully functionnaly dependent on the primary key.

That means that our Functional dependencies remain the same for the subject table:

$$Name, Contact\ details \rightarrow Name, Contact\ details, Subject$$

And for Location table:

$$Name, Contact\ details \rightarrow Name, Contact\ details, Location$$

3.3 3NF

In order to ensure that our database is in 3NF we need to make sure that two requirements are being fulfilled. Requirements are as follows: the database is in 2NF (we ensured that in section 3.2); there is no transitive FD, meaning there can be no dependencies between non-prime attributes.

Finally, if we take a look at our tables once again, we realise that dependencies between non-prime attributes would not be possible whatsoever. So, we can conclude that by normalising the DB to 1NF, we also normalised it to 2NF and 3NF.

That means that our Functional dependencies remain the same for the subject table:

$$Name, Contact\ details \rightarrow Name, Contact\ details, Subject$$

And for Location table:

$$Name, Contact\ details \rightarrow Name, Contact\ details, Location$$

4 ER diagram

Tutor can teach one or many (1..*) subjects (e.g. Jacki teaches Maths and English). We assume that if the tutor would not be able to teach any subjects, then they would not be part of the system in the first place.

We assume that if subject is offered to be teached then it has at least one tutor, but it can have multiple tutors as well, so cardinality is 1..* (e.g. both Jackie and Jeff teach Maths).

Tutor can teach in one or many locations (1..*), but location can have zero or many (0..*) tutors teaching at it, since, for examle, a student can live in a location, which does not have tutors.

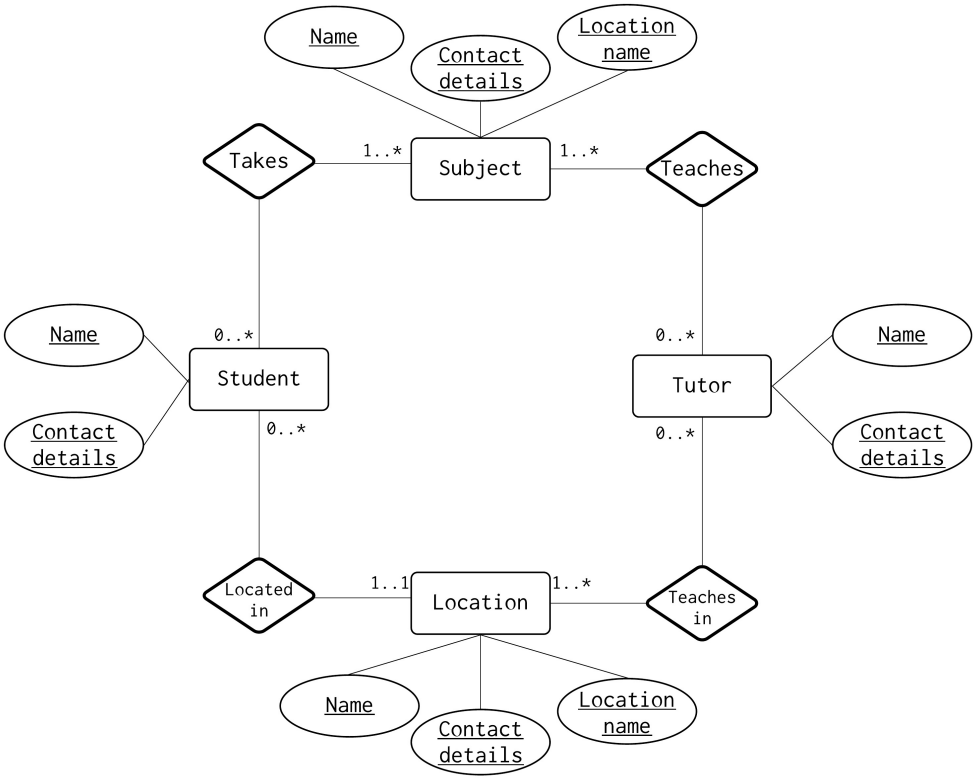

Figure 3: Basic ER Diagram

We assume that if student contacted the system, then they are willing to take one or many subjects (1..*).

But subject will not necessarily have students who will want to take it, so subject can have zero or many (0..*) students enrolled. Same thing with tutors: we assume that student can request a subject without a present subject tutor, meaning that subject can have 0..* tutors.

We say that each student is settled at one particular location and they are not travelling to the tutors, so students will have one and only one location (1..1).

But it is possible that there are no students in the particular location, so we say that location has 0..* students.

5 Implementation in PostgreSQL

5.1 Tutor Related Tables

All the tables that are connect to the tutors.

Input:

```
CREATE TABLE Tutors(Tutor_Name VARCHAR(30), Tutor_Contacts VARCHAR(40));

ALTER TABLE Tutors ADD CONSTRAINT Tutor_PK PRIMARY KEY
(Tutor_Name, Tutor_Contacts);

INSERT INTO Tutors(Tutor_Name, Tutor_Contacts)
```

```
VALUES ('Jackie', '01970666543'),
('Jeff', 'jeff@stemtutors.net'), ('Ahmed','07723456788'),
('Dafydd','07845333444'), ('Peter', 'pete@languagetutors.com'),
('Brian', 'brian@arttutors.org');
```

Output:

```
tutor_name |     tutor_contacts
------------+------------------------
Jackie     | 01970666543
Jeff       | jeff@stemtutors.net
Ahmed      | 07723456788
Dafydd     | 07845333444
Peter      | pete@languagetutors.com
Brian      | brian@arttutors.org
```

Input:

```
CREATE TABLE Tutor_Subject(Tutor_Name VARCHAR(30),
Tutor_Contacts VARCHAR(40), Subject_Teached VARCHAR(30) NOT NULL,
CONSTRAINT Tutor_Subject_FK FOREIGN KEY (Tutor_Name, Tutor_Contacts)
REFERENCES Tutors(Tutor_Name, Tutor_Contacts));

ALTER TABLE Tutor_Subject ADD CONSTRAINT Tutor_Subject_PK
PRIMARY KEY (Tutor_Name, Tutor_Contacts, Subject_Teached);

INSERT INTO Tutor_Subject(Tutor_Name, Tutor_Contacts, Subject_Teached)
VALUES ('Jackie', '01970666543', 'Maths'),
('Jackie', '01970666543', 'English'),
('Jeff', 'jeff@stemtutors.net', 'Maths'),
('Jeff', 'jeff@stemtutors.net', 'Science'),
('Ahmed','07723456788', 'Cymraeg'),
('Ahmed','07723456788', 'Art'),
('Dafydd','07845333444', 'Cymraeg'),
('Peter', 'pete@languagetutors.com', 'English'),
('Brian', 'brian@arttutors.org', 'Art');
```

Output:

```
tutor_name |     tutor_contacts      | subject_teached
------------+-------------------------+-----------------
Jackie     | 01970666543             | Maths
Jackie     | 01970666543             | English
Jeff       | jeff@stemtutors.net     | Maths
Jeff       | jeff@stemtutors.net     | Science
Ahmed      | 07723456788             | Cymraeg
Ahmed      | 07723456788             | Art
Dafydd     | 07845333444             | Cymraeg
Peter      | pete@languagetutors.com | English
Brian      | brian@arttutors.org     | Art
```

Input:

```
CREATE TABLE Tutor_Location(Tutor_Name VARCHAR(30),
Tutor_Contacts VARCHAR(40), Tutor_Located VARCHAR(40) NOT NULL,
CONSTRAINT Tutor_Location_FK FOREIGN KEY (Tutor_Name, Tutor_Contacts)
REFERENCES Tutors(Tutor_Name, Tutor_Contacts));

ALTER TABLE Tutor_Location ADD CONSTRAINT Tutor_Location_PK
PRIMARY KEY (Tutor_Name, Tutor_Contacts, Tutor_Located);

INSERT INTO Tutor_Location(Tutor_Name, Tutor_Contacts, Tutor_Located)
VALUES ('Jackie', '01970666543', 'Aberystwyth'),
('Jeff', 'jeff@stemtutors.net', 'Machynlleth'),
('Jeff', 'jeff@stemtutors.net', 'Talybont'),
('Jeff', 'jeff@stemtutors.net', 'Bow Street'),
('Ahmed','07723456788', 'Borth'),
('Ahmed','07723456788', 'Aberystwyth'),
('Ahmed','07723456788', 'Bow Street'),
('Ahmed','07723456788', 'Llandre'),
('Ahmed','07723456788', 'Talybont'),
('Dafydd','07845333444', 'Machynlleth'),
('Peter', 'pete@languagetutors.com', 'Bow Street'),
('Brian', 'brian@arttutors.org', 'Aberystwyth'),
('Brian', 'brian@arttutors.org', 'Llanrhystud');
```

Output:

```
tutor_name |     tutor_contacts      | tutor_located
-----------+-------------------------+---------------
Jackie     | 01970666543             | Aberystwyth
Jeff       | jeff@stemtutors.net     | Machynlleth
Jeff       | jeff@stemtutors.net     | Talybont
Jeff       | jeff@stemtutors.net     | Bow Street
Ahmed      | 07723456788             | Borth
Ahmed      | 07723456788             | Aberystwyth
Ahmed      | 07723456788             | Bow Street
Ahmed      | 07723456788             | Llandre
Ahmed      | 07723456788             | Talybont
Dafydd     | 07845333444             | Machynlleth
Peter      | pete@languagetutors.com | Bow Street
Brian      | brian@arttutors.org     | Aberystwyth
Brian      | brian@arttutors.org     | Llanrhystud
```

5.2 Student Related Tables

All the tables that are connected to the students.

Input:

```
CREATE TABLE Students(Students_Name VARCHAR(30), Student_Contacts VARCHAR(40));

ALTER TABLE Students ADD CONSTRAINT Students_PK PRIMARY KEY
(Student_Name, Student_Contacts);

INSERT INTO Students(Student_Name, Student_Contacts) VALUES
('Elin Davies', 'elin@example.com'), ('Steve Smith', 'steve@example.com');
```

Output:

```
student_name | student_contacts
-------------+------------------
Elin Davies  | elin@example.com
Steve Smith  | steve@example.com
```

Input:

```
CREATE TABLE Student_Subject (Student_Name VARCHAR(30),
Student_Contacts VARCHAR(40), Subject_Wanted VARCHAR(30) NOT NULL,
CONSTRAINT Student_Subject_FK FOREIGN KEY (Student_Name, Student_Contacts)
REFERENCES Students(Student_Name, Student_Contacts));

ALTER TABLE Student_Location ADD CONSTRAINT Student_Location_PK
PRIMARY KEY (Student_Name, Student_Contacts, Subject_Wanted);

INSERT INTO Student_Subject(Student_Name, Student_Contacts, Subject_Wanted)
VALUES ('Elin Davies', 'elin@example.com', 'Maths'),
('Elin Davies', 'elin@example.com', 'Science'),
('Steve Smith', 'steve@example.com', 'Llanrhystud');

INSERT INTO Student_Subject(Student_Name, Student_Contacts, Subject_Wanted)
VALUES ('Joseph Joestar', 'jojo@example.com', 'Geography');
```

Output:

```
student_name | student_contacts  | subject_wanted
-------------+-------------------+----------------
Elin Davies  | elin@example.com  | Maths
Elin Davies  | elin@example.com  | Science
Steve Smith  | steve@example.com | English
Joseph Joestar  | jojo@example.com | Geography
```

Input:

```
CREATE TABLE Student_Location (Student_Name VARCHAR(30),
Student_Contacts VARCHAR(40), Student_Located VARCHAR(40) NOT NULL,
CONSTRAINT Student_Location_FK FOREIGN KEY (Student_Name, Student_Contacts)
REFERENCES Students(Student_Name, Student_Contacts));

ALTER TABLE Student_Location ADD CONSTRAINT Student_Location_PK
PRIMARY KEY (Student_Name, Student_Contacts, Student_Located);

INSERT INTO Student_Location(Student_Name, Student_Contacts, Student_Located)
VALUES ('Elin Davies', 'elin@example.com', 'Machynlleth'),
('Steve Smith', 'steve@example.com', 'Llanrhystud');

INSERT INTO Student_Location(Student_Name, Student_Contacts, Student_Located)
VALUES ('Joseph Joestar', 'jojo@example.com', 'Cardiff');
```

Output:

```
student_name | student_contacts  | student_located
-------------+-------------------+-----------------
Elin Davies  | elin@example.com  | Machynlleth
Steve Smith  | steve@example.com | Llanrhystud
Joseph Joestar  | jojo@example.com | Cardiff
```

5.3 Decribing Tables

The outputs of 'describe table" command.

```
Table "vek1.tutors"
Column     |         Type          | Modifiers
---------------+----------------------+-----------
tutor_name     | character varying(30) | not null
tutor_contacts | character varying(40) | not null

Indexes:
"tutor_pk" PRIMARY KEY, btree (tutor_name, tutor_contacts)
Referenced by:
TABLE "tutor_location" CONSTRAINT "tutor_location_fk"
FOREIGN KEY (tutor_name, tutor_contacts)
REFERENCES tutors(tutor_name, tutor_contacts)
TABLE "tutor_subject" CONSTRAINT "tutor_subject_fk"
FOREIGN KEY (tutor_name, tutor_contacts)
REFERENCES tutors(tutor_name, tutor_contacts)
```

```
Table "vek1.students"
Column      |         Type          | Modifiers
-----------------+----------------------+-----------
student_name     | character varying(30) | not null
student_contacts | character varying(40) | not null

Indexes:
"students_pk" PRIMARY KEY, btree (student_name, student_contacts)
Referenced by:
TABLE "student_location" CONSTRAINT "student_location_fk"
FOREIGN KEY (student_name, student_contacts)
REFERENCES students(student_name, student_contacts)
TABLE "student_subject" CONSTRAINT "student_subject_fk"
FOREIGN KEY (student_name, student_contacts)
REFERENCES students(student_name, student_contacts)
```

```
Table "vek1.tutor_subject"
Column      |         Type          | Modifiers
----------------+----------------------+-----------
tutor_name      | character varying(30) | not null
tutor_contacts  | character varying(40) | not null
subject_teached | character varying(30) | not null
Indexes:
"tutor_subject_pk" PRIMARY KEY, btree (tutor_name,
tutor_contacts, subject_teached)
```

```
Foreign-key constraints:
"tutor_subject_fk" FOREIGN KEY (tutor_name, tutor_contacts)
REFERENCES tutors(tutor_name, tutor_contacts)
```

```
Table "vek1.student_subject"
Column      |         Type          | Modifiers
------------------+----------------------+-----------
student_name     | character varying(30) | not null
student_contacts | character varying(40) | not null
subject_wanted   | character varying(30) | not null

Indexes:
"student_subject_pk" PRIMARY KEY, btree (student_name,
student_contacts, subject_wanted)
Foreign-key constraints:
"student_subject_fk" FOREIGN KEY (student_name, student_contacts)
REFERENCES students(student_name, student_contacts)
```

```
Table "vek1.tutor_location"
Column     |         Type          | Modifiers
---------------+----------------------+-----------
tutor_name     | character varying(30) | not null
tutor_contacts | character varying(40) | not null
tutor_located  | character varying(40) | not null
Indexes:
"tutor_location_pk" PRIMARY KEY, btree (tutor_name,
tutor_contacts, tutor_located)
Foreign-key constraints:
"tutor_location_fk" FOREIGN KEY (tutor_name, tutor_contacts)
REFERENCES tutors(tutor_name, tutor_contacts)
```

```
Table "vek1.student_location"
Column      |         Type          | Modifiers
------------------+----------------------+-----------
student_name     | character varying(30) | not null
student_contacts | character varying(40) | not null
student_located  | character varying(40) | not null
Indexes:
"student_location_pk" PRIMARY KEY, btree (student_name,
student_contacts, student_located)
Foreign-key constraints:
"student_location_fk" FOREIGN KEY (student_name, student_contacts)
REFERENCES students(student_name, student_contacts)
```

5.4 Implementation

PostgreSQL Implementation was quiet a roller-coaster. Not because it was necessarily hard, but because first my laptop and then my Problem Model gave up on me. I had a fun time connecting to the database:

I would not be able to type anything in, everything would weirdly glitch and, as I panickly was trying to save my work on a USB drive, my laptop was acting as if it is about to die. Luckily, it did, but I did lose some nerves.

Fun part started when I realised that my model, that looked good on paper, does not really fit into the world of SQL: you cannot assign two foreign keys from two different tables; you cannot somehow join them and allow access the original tables in order to determine whether the person is student or tutor. That was somewhat dissapointing, but I just needed to accept it and do the first thing that came to mind: separate the big Location and Subject tables (see figure **??**) into two smaller ones, so that both student and tutor have individual copies of those. 4 tables became 6, as you can see from the structure above.

From their, it was just monotenous table creation and their population, assignment of PKs and FKs. Worth mentioning somehow, that I am still left with the feeling that I missed something big and could have implemented everything better. But at the moment of actual database creation – nothing really seemed to cause any problems, so I left everything as is.

6 Some sample queries

6.1 Query I

We are required to write a query to find all the tutors who teach math. It is pretty straightforward: we are showing all the rows from the table 'Tutor Subject' and then, using keyword *WHERE*, defining which values in the column 'Subject Teached' we are looking for (in our case it would be 'Maths').

```
SELECT *
FROM Tutor_Subject
WHERE Subject_Teached='Maths';
```

Output of the query is a table with rows from the table, containing information about name, contact details and subject which the tutor is teaching. Notice how both of the tutors have 'Maths' listed in the column 'Subject Teached'. I think it is safe to say that the query was successful and we received the expected output.

```
tutor_name |   tutor_contacts    | subject_teached
-----------+---------------------+-----------------
Jackie     | 01970666543         | Maths
Jeff       | jeff@stemtutors.net | Maths
(2 rows)
```

6.2 Query II

We are required to write a query to find all the tutors in 'Aberystwyth', who are teaching 'Art'. Even though, this query is more complex than the first one, the steps taken are quiet easy to understand. First, the same way we did in the first query, we specifying what we want to be shown in the output. Then, in order to access both tables simultaneously we need to join them: for that purpose we are making use of keyword *JOIN* (quiet obvious). After that we specifying *ON* using Foreign Key and Primary Key of tables: in our case they are holding the same values despite being in separate tables. And after that, the only thing left, is to make use of keyword *WHERE* once again and define what we are looking for (attributes 'Subject Teached' and 'Tutor Located').

```
SELECT Tutor_Subject.Tutor_Name, Tutor_Subject.Tutor_Contacts,
Tutor_Subject.Subject_Teached,Tutor_Location.Tutor_Located
FROM Tutor_Subject
```

```
JOIN Tutor_Location
ON Tutor_Subject.Tutor_Name = Tutor_Location.Tutor_Name
WHERE Tutor_Subject.Subject_Teached='Art'
AND Tutor_Location.Tutor_Located='Aberystwyth';
```

Output of the query is a table of 4 colums, which identifies the name, contact details, subject and location for the tutors. Both of the tutors have 'Art' listed as one of the subject they teach in the 'Aberystwyth' location. Query was succesful.

```
tutor_name |   tutor_contacts    | subject_teached | tutor_located
-----------+---------------------+-----------------+---------------
Ahmed      | 07723456788         | Art             | Aberystwyth
Brian      | brian@arttutors.org | Art             | Aberystwyth
(2 rows)
```

6.3 Query III

I did encounter difficulties with this query and was unable to complete it. I did understand that that input will be essentially joining all the tables together and then search through them, but I did not figure out the order in which they would connect. However, see my failed attempts below:

```
SELECT Student_Subject, Tutor_Subject, Student_Location.Student_Located,
Tutor_Location.Tutor_Located
FROM Student_Subject
JOIN Tutor_Subject
ON Student_Subject.Student_Name =Tutor_Subject.Tutor_Name
JOIN Student_Location
ON Student_Subject.Student_Name = Student_Location.Student_Name
JOIN  Tutor_Location
ON Student_Subject.Student_Name =Tutor_Location.Tutor_Name
WHERE Student_Subject.Subject_Wanted = Tutor_Subject.Subject_Teached;

SELECT *
FROM Student_Subject
CROSS JOIN Tutor_Subject
WHERE Student_Subject.Subject_Wanted = Tutor_Subject.Subject_Teached;

SELECT Student_Location.Student_Name, Student_Location.Student_Contacts
FROM Student_Location
JOIN Tutor_Location
ON Student_Location.Student_Located = Tutor_Location.Tutor_Located
WHERE NOT Student_Location.Student_Located = Tutor_Location.Tutor_Located;
```

7 Self evaluation

Modelling the problem – I believe that I did manage to fulfill all of the requirements for this section: it contains both the table for all of the data we were given in UNF and the list of attributes with notes of types. Just because I always think that there is something to improve, I would give myself 18/20 marks.

Candidate keys and functional dependencies within the data – again, I am listing attributes in unnormalised structure, specifying the choice of relational keys (SK, CK, PK), explaining current Functional Dependencies and describing every provided piece of information.

For this part, I would give myself 8/10 marks, because I could have potentially explained things a bit more in depth.

From UNF to 3NF: an account of the normalisation process – despite this section seeming quiet confusing in terms of lack of steps after 1NF, I think that I managed to present my point in full: I gave a description of what I was supposed to do, what I did, what was happening with attributes and Functional Dependencies. For this section, I would give myself 14/15 marks.

ER diagram – I tend to think that this section turned out to be one of the weaker once, because I was not sure about the style of the diagram, how advanced it is supposed to be and did the way of discussing cardinality mattered. I do believe that I still did an acceptable job, so I would award myself 6/10 for this section.

Implementation in PostgreSQL – I would give myself 20/30 marks, due to my prior model not being able to get implimented and lack of junction tables. At the same time, I feel like I did explain everything rather well and included all the needed tables.

Some sample queries – for the successful queries I did not only list the code and the output, but also gave a good description for everything that was happening and what we received in the end. For the third one: described the thinking process. I would award myself 6/10 marks.

Self evaluation – it seems to me like that the given explanations and self-reflection was detailed enough for me to earn full marks, 5/5 for this section.

Based on the version 1.2 of Assessment Criteria for Development, even though my work most certainly is not flawless or exceptional, I think that I have managed to show a good understanding of the problem. As always, if I would have put more effort – I would have been able to present a better work, but at this point, judging on the self evaluation, I have earned myself 77 percent of the total mark.