1. Teaching and Studying Database Design and Implementation at a University

Checking different curricula of Computer Science at a university you will find at least one course of database design and implementation in it. There might be some additional topics on database offered especially if you look at some master's courses. They will cover topics like distributed databases or databases and the web and so on. What is taught in these advanced courses is very much depending on the university and which environment this university offers its courses in. But if you analyse what is taught in the fundamental course of database and implementation you will find the topics:

1. Database Design,
2. Theory of Database Management Systems, and
3. SQL

This is very nice for those teaching this subject because they can go round the world in their sabbaticals taking the course they teach at their home university and teaching the topics mentioned above in any place. They might have to change the sequence of the topics according to local needs, but when ever they start the topic database design, the areas taught in there are very much the same all over the world (Table 1).

Table 1. Standardized topics in teaching database design at different universities

<table>
<thead>
<tr>
<th>Requirement Analysis</th>
<th>Data Modelling (ER - Diagrams)</th>
<th>Normalisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Tables and Test</td>
<td>DB - Tuning</td>
<td>Migration and Administration</td>
</tr>
</tbody>
</table>

This is fine for the students as well because if they want to go to another university for a semester during their studies doing the course "Database Design" or "Database Fundamentals" or "Logical Database Design" they will not have big problems to get the grade they have achieved at another university recognised by their home university.

If you look into this in more detail with a special emphasis on the topic of designing a database, then you will find that requirement analysis, data modelling (ER - Diagrams) and normalisation are taught anywhere and often in the same why. What are the reasons for doing it in the same way? Is it because there is only one way of doing it? Or is there another answer to this question too?

The reasons for teaching and studying database design in the same way in different places are because the methods are shown in Table 2:

Table 2. Reasons for the resemblance of teaching and studying database design

<table>
<thead>
<tr>
<th>Requirement Analysis</th>
<th>Data Modelling (ER - Diagrams)</th>
<th>Normalisation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create Tables and Test</td>
<td>DB - Tuning</td>
<td>Migration and Administration</td>
</tr>
</tbody>
</table>

The first three arguments for the similarity between teaching and studying are self explanatory. But as soon as it comes to the reason "easy to be interpreted" one has to ask, who is doing the interpretation. Is that the user of the database or the designer of the database? And if you add the last criterion "not easy to be falsified" then the answer to the questions "who is able to falsify an ER-diagram" and "who could answer the question whether a relation is in the first, second or even third normal form", it would come directly into your mind, that this can't be the user! So it has to be the designer. But that seems not to be the best solution. A person doing the job is checking the quality achieved as well. Is that the
correct way of doing it?

Does this mean that database design has to be done by the designer and no user participation is possible in the design process? Is there not a way of user integration or at least user participation? The answer to this is “no” if you are using the methods which are currently common around the world.

2. NIAM - A Different Way of Designing a Database

Fifteen years ago a different method of database design was developed at the Queensland University in Brisbane (Australia). In the following years this method spread to the South-East and to the United States. Even some universities in Europe got into contact with this method and checked it using it several database projects.

The NIAM method (Njissen Information Analysis Method) is a fact-oriented approach for designing a database. It is based on a fact-oriented description of the Universe of Discourse (UoD) of the Database in a structured language. Using this approach no knowledge of ER-Diagramming or Normalization is needed. In addition to that all design stages can be checked by the user of the database to verify whether these have been done correctly and whether the result of it is representing the UoD correctly.

The method is structured into ten steps. These have to be done in a sequence from 1 to 10. The output of each step can be proofed after the step has been finished. There are certain rules to indicate which step you have to go back to after a design failure has been discovered and it can be described precisely which corrections have to be done.

As it would be out of the scope of this short overview to describe all ten steps in details (see the literature mentioned at the end) the most important steps will be described here: these are the steps 1 to 4 and step 10 showing the background of this approach. So this short article concentrates on these steps only.

2.1 Describing the UoD in a Structured Language

In the first step the UoD has to be described by representative examples of facts. These have to be turned to fact types afterwards. The second step derives the first draft of the schema from step 1 by turning the fact types in to a graphical design. In a third step this first draft of the schema can be simplified by using specific rules. After that the uniqueness constraints have to be added to the schema. In order to make the schema more precise steps 5 to 9 have to be accomplished - adding additional constraints. And finally in step 10 an algorithm for turning the schema into tables of the optimal normal form is applied.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Describing the UoD by fact types using a structured language</td>
</tr>
<tr>
<td>2</td>
<td>First schema design</td>
</tr>
<tr>
<td>3</td>
<td>Simplify the schema</td>
</tr>
<tr>
<td>4</td>
<td>Add uniqueness constraints</td>
</tr>
<tr>
<td>5</td>
<td>Check clarity and add constraints</td>
</tr>
<tr>
<td>6</td>
<td>Add further constraints</td>
</tr>
<tr>
<td>7</td>
<td>First schema check by the user</td>
</tr>
<tr>
<td>8</td>
<td>Restriction on roles</td>
</tr>
<tr>
<td>9</td>
<td>Verification of the schema</td>
</tr>
<tr>
<td>10</td>
<td>Transform the scheme to optimal normal form relations</td>
</tr>
</tbody>
</table>

Table 3. The ten steps of NIAM

Step 1 involves describing the UoD by fact types using. In this step of the NIAM method, all representative facts of the UoD have to be found and described in sentences using only entity types, label types for the entity types, labels and a role. A sentence is correct if it can be turned in to a fact type by substituting the labels by dots and it stays representative and correct for all existing facts in this UoD. In addition to that, this sentence has to be atomic.

These sentences can be unary, binaries or can even contain three or more entity types and roles attached to them.

A unary fact of the UoD could be:
The person with the name John smokes.

In this fact person is an entity type, name would be the label type and John would be a label. The role smokes defines what fact about that person you are going to store in your database.

An example of a binary fact of the Uod would be:
A student with the student id 4711 is enrolled in a course with the c_name database design.

In this fact entity types are student and course. Label types are student_id and c_name.

Labels are 4711 and database design. Finally the role is is_enrolled.

And last not least here is an example of a ternary fact:
The **student** with the **stud_id** 4711 has got in the **course** with the **c_name** database _design_ **marks** in % 75.

The components of this ternary are:

<table>
<thead>
<tr>
<th>Entity types</th>
<th>Label types</th>
<th>Labels</th>
<th>Roles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>stud_id</td>
<td>4711</td>
<td>has got in</td>
</tr>
<tr>
<td>Course</td>
<td>c_name</td>
<td>database <em>design</em></td>
<td></td>
</tr>
<tr>
<td>Marks</td>
<td>%</td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>

Turning these facts into fact types can be done by omitting the labels easily. By doing this the facts are turned into fact types and these have to be correct for the UoD. So the whole UoD can be described in short sentences like this:

Person (name) ... smokes.

Student (student id) ... is enrolled in Course (c_name) ...

Student (stud_id) ... has got in Course (c_name) ... Marks (%) ...

The following restrictions for fact types apply:

1. They have to be elementary. That means they can't be split into a lower degree fact type (e.g. a ternary into two binaries) without losing information.
2. They are representative for that UoD. That means that facts of this type have to be stored in the relevant database.
3. They have to be fact types and not any sentences like conditions. (Example: If the person with the name Ana smokes she might get ill). Sentences using logical operators like: "The person with the name Ana or the person with the name Paul is smoking" are not allowed. Finally hopes (and not facts) like: "The country with the name XYZ will win medals of the kind gold in the quantity 10 in the Olympics 2004" will not be accepted.

### 2.2 Developing the First Draft of the Database Schema

Often humans understand the picture of a complex problem better than a text description of the problem. NIAM offers this option in the second step of designing a database. That means that the reality which is described in the first step in a structured language is now turned into a diagram using the circle as entity type (Figure 1).

![Figure 1. An entity type](image)

Each entity will have a name and a label type of it as well. To each entity type at least one role is attached. For describing a role an rectangle is used describing the fact which has to be stored (r1 = smokes) by a verb (Figure 2).

![Figure 2. Unary fact type](image)

If there is more than one entity type involved in a fact type, it is described as a binary or a ternary (Figures 3 and 4).

![Figure 3. Binary fact type](image)
Step 2 does nothing more than this. It transforms the verbal description the fact types into graphical ones where each fact type is represented by the symbols mentioned above.

### 2.3 Simplifying the Schema

In step 3 of NIAM the schema gets simplified by using the following rules:

1. If the same entity type shows up more than once in the schema integrate all entities of this type into one entity type (Figure 5a).

The result is shown in Figure 5b).

\[\text{student (stu-id)}\] \[\text{enrols}\] \[\text{course (title)}\] \\
\[\text{student (stu-id)}\] \[\text{enrols}\] \[\text{book (isbn-no)}\] \\

(a)

\[\text{student (stu-id)}\] \[\text{enrols}\] \[\text{course (title)}\] \\
\[\text{buys}\] \[\text{book (isbn-no)}\] \\

(b)

Figure 5a and Figure 5b. The entity type "Student" integrates two roles ("enrols" and "buys")

2. Entity types which can be compared can be represented by only one entity type as long as the name of the entity type is attached to the role. For example, the types in Figure 5c) are transformed into the types in Figure 5d).

\[\text{book (isbn-no)}\] \[\text{costs}\] \[\text{price ($)}\] \\
\[\text{pen (item-no)}\] \[\text{costs}\] \[\text{price ($)}\] \\

(c)

\[\text{book (isbn-no)}\] \[\text{has price}\] \[\text{amount ($)}\] \\
\[\text{pen}\] \[\text{has price}\] \\

(d)
Other rules for simplifying the schema can be applied for example, the rule that entities which have the same role attached can be united into one entity. And last but not least is the rule that one can define all fact types which can be derived from other fact types by documenting the rule; this makes the schema simpler and easier to understand.

2.4 Adding Uniqueness Constraints to the Schema

After having done the third step the design comes to a stage where it may be viewed and understood easily. At this point uniqueness constraints can be added to the schema. Uniqueness constraints have to be added to each fact type, defining which label types will define that constraint. Each constraint is documented by a line under the corresponding roles connected to the relevant label types. A uniqueness constraint can be defined over one role or over several roles.

The possible alternatives for a binary fact type and for a ternary fact type are shown in Figure 6 and in Figure 7 respectively.

Figure 5c. and Figure 5d. "Price" is represented by "amount"
If you find that a ternary fact type has an uniqueness constraint like the one shown in Figure 8, you should rethink this because it can not be correct. There are two solutions to this problem. One is that you have made a mistake in the first step of NIAM because you have defined a fact type which is not elementary. That means the ternary can be split into two binaries. The other alternative is that the uniqueness constraint was not defined correctly. So you have to check your design decisions again and you will find out how to correct this fault.

---

**2.5 The Optimal Normal Form (ONF) Algorithm**

Skipping steps 5 to 9 which are mentioned in Table 1, finally step 10 of the NIAM Method is reached. In this step the schema which was enhanced in steps 5 to 9 with additional constraints will be turned into relations which are in an optimal normal form. The following rules are used:

1. For each fact type without a simple key form a table. The shortest key will form the primary key of that table.
2. For all fact types with a simple key find the entity all other entities relate to. All these fact types will go into this table. The primary key is formed by the common uniqueness constraint.
3. For each fact types being left out, form a separate table.

Let us apply this rule to a small segment of a schema which is shown in Figure 9: Rule 1 cannot be applied because only simple keys appear in this segment of the schema.

---

If there were be non-simple keys, with a uniqueness constraint overstretching more than one role, for them all a
A separate table would be formed. It is expected that if there are fact types with the same uniqueness constrains (the key is overstretching the same roles), then after step 10 all entity types involved would find themselves in the same relation - if step 10 were done correctly!

As no such a uniqueness constraint is found in this example, this rule cannot be used. Instead, rule 2 can be applied. This would form a relation:

```
Customer = (Customer#, Creditcard#, Day_of_Birth, Place_of_Residence)
```

By applying rule 3 we will get

```
Town = (ZIP-Code, Town_name)
```

Now all relations created are in an optimal normal form.

### 3. A Short Comparison of NIAM to Traditional Methods of Database Design

As NIAM applies the criteria mentioned at the beginning of this paper (Table 2), it seems to be very similar to the traditional methods of database design as it is easy to teach, easy to learn and easy to understand.

But there is one main difference: the schema generated using NIAM is also easy to be read, understood and corrected by the user - which is not true for the traditional methods of database design.

By using NIAM for database design user participation in the design process is not a word only - NIAM opens the door to do it! The reason is that this method produces results which can be used and checked by the user!

### 4. Bibliography


