**Chapter 8: Relational Database Design**

Table of Contents

[Good Relational Database Aspects 2](#_Toc49498837)

[Design Alternatives: Smaller to Larger Schema Options 2](#_Toc49498838)

[Design Alternatives: Smaller Schema Option 3](#_Toc49498839)

[Schema Decomposition 4](#_Toc49498840)

[Normal Forms 5](#_Toc49498841)

[Atomic Domains (First Normal Form) 5](#_Toc49498842)

[Decomposition Using Functional Dependencies 6](#_Toc49498843)

[Boyce – Codd Normal Form 9](#_Toc49498844)

## Good Relational Database Aspects

The goal of relational database is to generate a set of relation schemas that meet the following points:

* Allow the storage of data without unnecessary redundancy
* Allow quick data retrieval

The standard method to evaluate a design to ensure it meets this goal is called Normal Forms. This is a theory that allows us to ensure our design meets the intended goals.

### Design Alternatives: Smaller to Larger Schema Options

Consider the following two schemas:

department(dept\_name, building, budget)  
instructor(id, name, dept\_name, salary)

SQL

The problem with this design is that it involves a natural join to retrieve some data. This is a very expensive process. On a large scale, it will make retrieval of data slow, which fails out goal.

In order to solve this problem, we will merge the two schemas into one.

inst\_dept(id, name, salary, dept\_name, building, budget)

SQL

But even this has problems. Firstly, there will be data redundancy, since a lot of the department information will be repeated. This leads to the second problem, inconsistency. Any updates to the department information must be made everywhere, including in any other schemas. This is both time-consuming and error-prone. Thirdly, it introduces some weird logic into our program. Specifically, we will be unable to add data for a new department without also adding at least one teacher to the department.

### Design Alternatives: Smaller Schema Option

Suppose instead, we start with the larger, combined schema.

inst\_dept(id, name, salary, dept\_name, building, budget)

SQL

We need to answer a few major questions before we decide to split up this schema:

* How do we know that it contains repeated data and should be split up? It may feel like it is easy to spot repetitions in some sample data, but in real-life designs, there are often millions of records involved. This makes it next to impossible to spot repetitions.
* How do we know that the ‘repetitions’ we are seeing are genuinely repetitions and not just coincidences? For example, how can we know for certain that a department is supposed to reside in just one specific building and that it is not purely coincidental that all the entries for that department have the same building? Maybe we have three different departments that happen to have the same budget and also share a single building.

To know that the data we are seeing genuinely does have repetitions, we need a formal method. This means we need to allow the database designer to specify some rules, such as ‘each department must have a single budget’, ‘each department must be in a single building’, etc. Essentially, we need to have a rule that says if there was a schema (dept\_name, budget), dept\_name would be able to serve as the primary key.

This form of rule is called functional dependency. It is expressed as dept\_name 🡪 budget (read as ‘dept\_name controls budget’). This means that each value of dept\_name will always give us a single value for budget.

### Schema Decomposition

It must be easy to see that inst\_dept­ really should be decomposed into instructor and department, as in the original decision. However, this is because we are looking at a very small scope and we can almost intuitively tell from the given example what should be done. It is not always so easy to make this decision especially in less obvious situations, where there are a large number of attributes and several functional dependencies. We need some formal methodology, which is where Normal Forms comes in.

Consider the following schema:

employee(id, name, street, city, salary)

SQL

Say we decompose it into the following two schemas:

employee1(id, name)  
employee2(name, street, city, salary)

SQL

The main problem with this decomposition is the fact that name has been selected as the joining attribute. It is entirely possible that two employees have the exact same name. In that situation, the join will be performed on every occurrence of the names in the two tables, which will give us double the records, since there will be incorrect records included. We failed to produce the original records using a natural join.

Problematic decompositions like this one are called lossy decomposition. We need to find the correct decomposition method, which is called lossless decomposition.

## Normal Forms

### Atomic Domains (First Normal Form)

An atomic domain is one where the elements of the domain are indivisible units, meaning they cannot be subdivided. For example, a person’s height, weight, country of origin, etc.

A relation R is said to be in the first normal form (1NF) if:

* All the attributes of R are atomic
* The value of each attribute only contains a single value from that domain (no multiple values)

In the case of multiple values, it is possible to simply add another attribute to make the relation 1NF. However, if we do that, it will most likely lead to wastage of space. For example, we may encounter the problem of multiple values with phone numbers, but on the other hand, most people have just one phone number. This will cause a lot of NULL values. An even bigger problem is if someone has five different phone numbers. It is infeasible to simply keep adding attributes. The ideal solution to the problem of multiple values is to use a separate table for that attribute. This will solve the problem of wastage of space, while still keeping the relation in 1NF.

Integers are considered to be atomic, as long as they do not have sub-parts. For example, student IDs could be of the form ‘CSE001’. This clearly has sub-parts. The problem with leaving this like this, is that it takes extra programming to extract the separate parts if we needed to. Thus, such an integer is not considered to be atomic.

The exception to this is if the non-atomic integer is actually made up of parts that are also separately available in the schema as attributes. In the case of the student ID ‘CSE001’, if the department name is also separately another attribute, this will be considered to be atomic. Of course, there is redundancy here, but this is ignored since it is intentional and serves the end-users. Thus, whether or not the attribute is actually atomic depends on how we are designing our schema.

### Decomposition Using Functional Dependencies

So far, we have seen sample data and decided from the data if the schema should be decomposed or not. However, as mentioned before, we need a formal method to make this decision. This formal method is based on the concepts of keys and functional dependencies. Keys refer to attributes that are used to identify records, such as the primary key, and functional dependencies have already been discussed.

The notation we use in this section are as follows:

* Greek letters (, etc.) are used for sets of attributes that may or may not form a schema.
* Roman Letters (, etc.) are used for sets of attributes that form a schema.
* A set of attributes is a superkey, is the corresponding relation.
* Lowercase names are used for relations.
* The term ‘instance of ’ refers to the particular value of a relation at a given time.

Keys and functional dependencies are mainly used to ensure that the design follows some rules or constraints. For example,

* Students and instructors are uniquely identified by their ID
* Each student and instructor has only one name
* Each instructor and student is primarily associated with only one department (exceptions are being ignored here)
* Each department has only one budget and only one building

An instance of a relation that satisfies all the constraints set on it is called a legal instance of the relation. If all the relation instances in a database are legal instances, that database is said to have a legal instance.

A superkey is a set of one or more attributes that, collectively, allows us to uniquely identify a tuple in the relation.

Formally, for the relation schema , a subset of is a superkey of if, in any legal instance of , for all pairs of tuples and in that instance of , if , then .

The formal definition for functional dependencies is as follows:

In a relation schema , let and . Given an instance of , the instance satisfies the functional dependency for all pairs of tuples and in that instance if, when , then . We say that the functional dependency holds on the schema if, in every instance of , the functional dependency is satisfied. Essentially, that the property is true for all data at all times in that relation.

|  |  |  |  |
| --- | --- | --- | --- |
| **recordNo()** |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Note that, even though in the example above, the reverse is not true. This is because there are records for which the value d for .

A functional dependency is called a trivial functional dependency if it is satisfied by all relations. For example, is satisfied by all relations involving the attribute . Similarly, is also satisfied by all relations involving .

Formally, a functional dependency is trivial if .

### Boyce – Codd Normal Form

A relation schema is in BCNF with respect to a set of functional dependencies if, for all functional dependencies in (no clue what this is) of the form , at least one of the following conditions holds:

* is a trivial functional dependency
* is a superkey

Consider the following schema:

inst\_dept(id, name, salary, dept\_name, building, budget)

SQL

This schema is not in BCNF. However, this schema can be split into two parts

instructor(id, name, dept\_name, salary)  
dept(dept\_name, building, budget)

SQL

Both of these parts are in BCNF.

Say is a schema that is not in BCNF. This means that there is at least one non-trivial functional dependency such that is not a superkey for . In this situation, we replace the schema with two schemas:

In the context of our example, let be dept\_name and be building, budget. Here, , since there are no common attributes. Thus, we get:

* (dept\_name, building, budget)
* (id, name, dept\_name, salary)