**Normalization of Database**

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy (repetition) and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form, removing duplicated data from the tables.

Normalization is used for mainly two purposes:

* Eliminating redundant (useless) data.
* Ensuring data dependencies make sense i.e. data is logically stored.

Problems without Normalization

If a table is not properly normalized and have data redundancy then it will not only take extra memory space but will also make it difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized. To understand these anomalies let us take an example of a Student table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **rollno** | **name** | **branch** | **hod** | **office\_tel** |
| 401 | Akon | CSE | Mr. X | 53337 |
| 402 | Bkon | CSE | Mr. X | 53337 |
| 403 | Ckon | CSE | Mr. X | 53337 |
| 404 | Dkon | CSE | Mr. X | 53337 |

In the table above, we have data of 4 Computer Science students. As we can see, data for the fields of branch, hod (Head of Department) and office\_tel is repeated for the students who are in the same branch in the college. This is Data Redundancy.

Insertion Anomaly

Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as NULL.

Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students.

These scenarios are nothing but Insertion anomalies.

Updation Anomaly

What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

Deletion Anomaly

In our Student table, two different information are kept together, Student information and Branch information. Hence, if in case, a branch is inactive and is not admitting students anymore, then we may plan to delete that branch records in our table; and in this case we will lose the student information as well. This is Deletion anomaly.

**Normalization Rules**

Normalization rules are divided into the following normal forms:

1. First Normal Form
2. Second Normal Form
3. Third Normal Form

First Normal Form (1NF)

The 1st Normal form expects us to design the table in such a way that it can easily be extended and it is easier to retrieve data from it whenever required. If tables in a database are not even in the 1st Normal Form, it is considered as bad database design.

Rules for First Normal Form: the first normal form expects to follow a few simple rules while designing the database, and they are:

Rule 1: Single Valued Attributes

Each column of the table should be atomic/single valued i.e. they should not contain multiple values.

Rule 2: Attribute Domain should not change

In each column the values stored must be of the same kind or type.

For example: If you have a column dob to save date of births of a set of people, then you cannot or you must not save 'names' in that column i.e. It should hold only 'date of birth' for all the records/rows.

Rule 3: Unique name for Attributes/Columns

This rule expects that each column in a table should have a unique name. This is to avoid confusion at the time of retrieving data or performing any other operation on the stored data. If one or more columns have same name, then the DBMS will be left confused.

Rule 4: Order doesn't matter

This rule says that the order in which you store the data in your table doesn't matter.

Example - let’s check this table:

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **subject** |
| 101 | Akon | OS, CN |
| 103 | Ckon | Java |
| 102 | Bkon | C, C++ |

This table already satisfies 3 rules out of the 4 rules, as all our column names are unique, we have stored data in the order we wanted to and we have not inter-mixed different type of data in columns.

But, out of the 3 different students in our table, 2 have opted for more than 1 subject; and we have stored the subject names in a single column.

As per the 1st Normal form each column must contain atomic value. So, to resolve this problem we have to break the values into atomic values.

Here is our updated table and it now satisfies the First Normal Form.

|  |  |  |
| --- | --- | --- |
| **roll\_no** | **name** | **subject** |
| 101 | Akon | OS |
| 101 | Akon | CN |
| 103 | Ckon | Java |
| 102 | Bkon | C |
| 102 | Bkon | C++ |

By doing so, although a few values are getting repeated but values for the subject column are now atomic for each record/row. Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

# Second Normal Form (2NF)

For a table to be in the Second Normal Form, it must satisfy two conditions:

1. The table should be in the First Normal Form.
2. There should be no Partial Dependency.

## But, what is Dependency? Let's take an example of a Student table with columns student\_id, name, reg\_no, branch, and address.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |

In this table, student\_id is the primary key and will be unique for every row, so we can use student\_id to fetch any row of data from this table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
| 10 | Akon | 07-WY | CSE | Quebec |
| 11 | Akon | 08-WY | IT | Ontario |

Hence, we can say a Primary Key for a table is the column or a group of columns (composite key) which can uniquely identify each record in the table.

We can ask for branch name of student with student\_id 10, and we can get it. Similarly, if we ask for name of student with student\_id 10 or 11, we will get it. So all we need is student\_id and every other column depends on it, or can be fetched using it.

This is Dependency and we also call it Functional Dependency.

## Now, what is Partial Dependency? For a simple table like Student, a single column like student\_id can uniquely identify all the records in a table; however, this is not true all the time. So now let's extend our example to see if more than 1 column together can act as a primary key.

Let's create another table for Subject, which will have subject\_id and subject\_name fields and subject\_id will be the primary key.

|  |  |
| --- | --- |
| **subject\_id** | **subject\_name** |
| 1 | Java |
| 2 | C++ |
| 3 | Php |

Now we have a Student table with student information and another table Subject for storing subject information.

Let's create another table Score, to store the marks obtained by students in the respective subjects. We will also be saving name of the teacher who teaches that subject along with marks.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **teacher** |
| 1 | 10 | 1 | 70 | Java Teacher |
| 2 | 10 | 2 | 75 | C++ Teacher |
| 3 | 11 | 1 | 80 | Java Teacher |

In the score table we are saving the student\_id to know which student's marks are these and subject\_id to know for which subject the marks are for.

Together, student\_id + subject\_id forms a Candidate Key for this table, which can be the Primary key.

If we ask you to get me marks of student with student\_id 10, can you get it from this table? No, because you don't know for which subject. And if I give you subject\_id, you would not know for which student. Hence, we need student\_id + subject\_id to uniquely identify any row.

### But where is Partial Dependency?

Now if you look at the Score table, we have a column name teacher which is only dependent on the subject, for Java it's Java Teacher and for C++ it's C++ Teacher & so on.

Now as we just discussed that the primary key for this table is a composition of two columns which is student\_id & subject\_id but the teacher's name only depends on subject, hence teacher's name has nothing to do with student\_id.

This is Partial Dependency, where an attribute in a table depends on only a part of the primary key and not on the whole key.

## How to remove Partial Dependency?

There can be many different solutions for this, but our objective is to remove teacher's name from Score table. The simplest solution is to remove column teacher from Score table and add it to the Subject table. Hence, the Subject table will become:

|  |  |  |
| --- | --- | --- |
| **subject\_id** | **subject\_name** | **teacher** |
| 1 | Java | Java Teacher |
| 2 | C++ | C++ Teacher |
| 3 | Php | Php Teacher |

And our Score table is now in the second normal form, with no partial dependency.

|  |  |  |  |
| --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** |
| 1 | 10 | 1 | 70 |
| 2 | 10 | 2 | 75 |
| 3 | 11 | 1 | 80 |

To summarize:

1. For a table to be in the Second Normal form, it should be in the First Normal form and it should not have Partial Dependency.
2. Partial Dependency exists, when for a composite primary key, any attribute in the table depends only on a part of the primary key and not on the complete primary key.
3. To remove Partial dependency, we can divide the table, remove the attribute which is causing partial dependency, and move it to some other table where it fits in well.

# Third Normal Form (3NF)

Third Normal Form is an upgrade to Second Normal Form. When a table is in the Second Normal Form and has no transitive dependency, then it is in the Third Normal Form.

Example: We have 3 tables, Student, Subject and Score.

#### Student Table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **name** | **reg\_no** | **branch** | **address** |
| 10 | Akon | 07-WY | CSE | Quebec |
| 11 | Akon | 08-WY | IT | Ontario |
| 12 | Bkon | 09-WY | IT | Alberta |

#### Subject Table

|  |  |  |
| --- | --- | --- |
| **subject\_id** | **subject\_name** | **teacher** |
| 1 | Java | Java Teacher |
| 2 | C++ | C++ Teacher |
| 3 | Php | Php Teacher |

#### Score Table

|  |  |  |  |
| --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** |
| 1 | 10 | 1 | 70 |
| 2 | 10 | 2 | 75 |
| 3 | 11 | 1 | 80 |

In the Score table, we need to store some more information, which is the exam name and total marks, so let's add 2 more columns to the Score table.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_name** | **total\_marks** |

As explained above, for a table to be in the third normal form,

1. It should be in the Second Normal form.
2. And it should not have Transitive Dependency.

### What is Transitive Dependency?

With exam\_name and total\_marks added to our Score table, it saves more data now. Primary key for our Score table is a composite key, which means it's made up of two attributes or columns → student\_id + subject\_id.

Our new column exam\_name depends on both student and subject. For example, a mechanical engineering student will have Workshop exam but a computer science student won't. And for some subjects you have Practical exams and for some you don't. So, we can say that exam\_name is dependent on both student\_id and subject\_id.

And what about our second new column total\_marks? Does it depend on our Score table's primary key?

Well, the column total\_marks depends on exam\_name as with exam type the total score changes. For example, practicals are of less marks while theory exams are of more marks.

But, exam\_name is just another column in the score table. It is not a primary key or even a part of the primary key, and total\_marks depends on it.

This is Transitive Dependency. When a non-prime attribute depends on other non-prime attributes rather than depending upon the prime attributes or primary key.

### How to remove Transitive Dependency?

Again the solution is very simple. Take out the columns exam\_name and total\_marks from Score table and put them in an Exam table and use the exam\_id wherever required.

#### Score Table: In 3rd Normal Form

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **score\_id** | **student\_id** | **subject\_id** | **marks** | **exam\_id** |

#### The new Exam table

|  |  |  |
| --- | --- | --- |
| **exam\_id** | **exam\_name** | **total\_marks** |
| 1 | Workshop | 200 |
| 2 | Mains | 70 |
| 3 | Practicals | 30 |

The advantage of removing transitive dependency is,

* Amount of data duplication is reduced.
* Data integrity achieved.

Example:

Imagine we're building a restaurant management application. That application needs to store data about the company's employees and it starts out by creating the following table of employees:

| **EMPLOYEE\_ID** | **NAME** | **JOB\_CODE** | **JOB** | **STATE\_CODE** | **HOME\_STATE** |
| --- | --- | --- | --- | --- | --- |
| E001 | Alice | J01, J02 | Chef, Waiter | 26 | Michigan |
| E002 | Bob | J02, J03 | Waiter, Bartender | 56 | Wyoming |
| E003 | Nick | J01 | Chef | 56 | Wyoming |