Database: The database is a collection of interrelated data which is used to retrieve, insert and delete the data efficiently. It is also used to organize the data in the form of a table, schema, views, and reports, etc.

ER model

* ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
* It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
* In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

Components of the ER Diagram

This model is based on three basic concepts:

* Entities
* Attributes
* Relationships

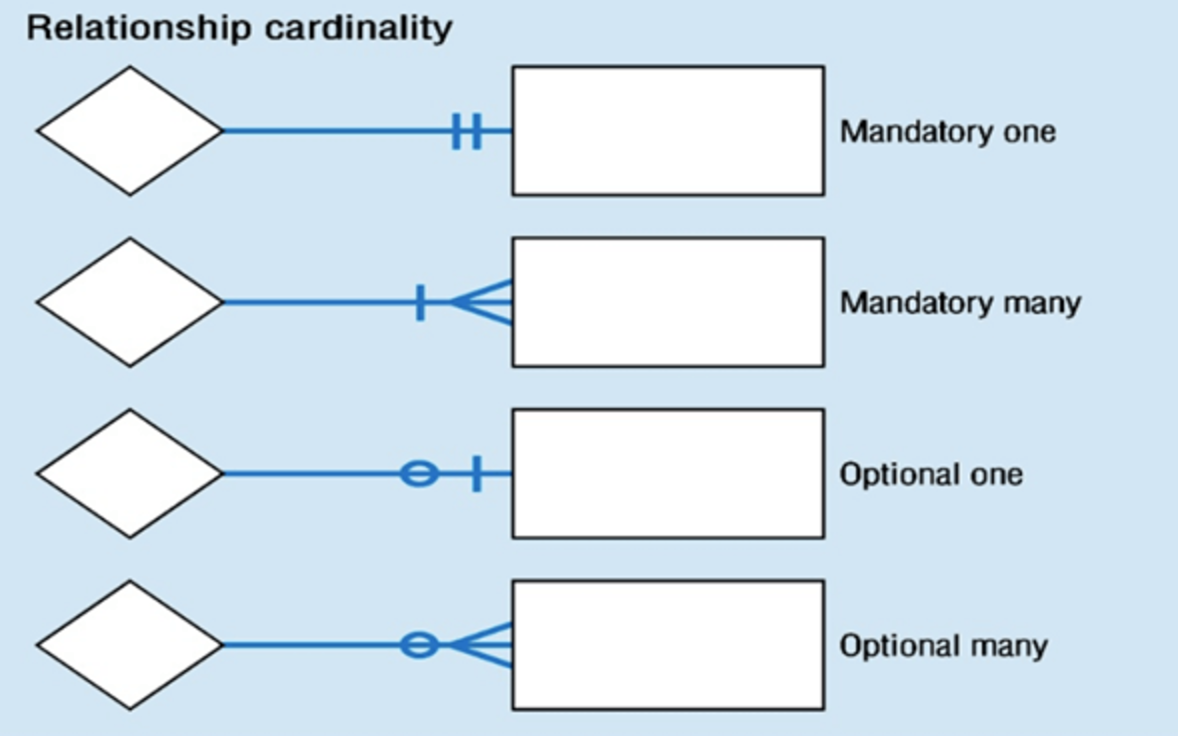
|  |  |
| --- | --- |
| **Types of Attributes** | **Description** |
| **Simple attribute** | Simple attributes can't be divided any further. For example, a student's contact number. It is also called an atomic value. |
| **Composite attribute** | It is possible to break down composite attribute. For example, a student's full name may be further divided into first name, last name, and last name. |
| **Derived attribute** | This type of attribute does not include in the physical database. However, their values are derived from other attributes present in the database. For example, age should not be stored directly. Instead, it should be derived from the DOB of that employee. |
| **Multivalued attribute** | Multivalued attributes can have more than one values. For example, a student can have more than one mobile number, email address, etc. |

Cardinality

Defines the numerical attributes of the relationship between two entities or entity sets.

Different types of cardinal relationships are:

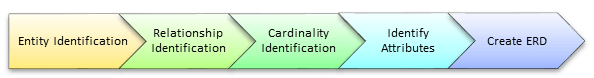
* One-to-One Relationships
* One-to-Many Relationships
* May to One Relationships
* Many-to-Many Relationships



ER- Diagram is a visual representation of data that describe how data is related to each other.

* **Rectangles:** This symbol represent entity types
* **Ellipses :** Symbolrepresent attributes
* **Diamonds:** This symbol represents relationship types
* **Lines:** It links attributes to entity types and entity types with other relationship types
* **Primary key:** attributes are underlined
* **Double Ellipses:** Represent multi-valued attributes

Following are the steps to create an ERD.

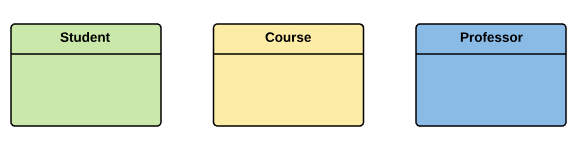


In a university, a Student enrolls in Courses. A student must be assigned to at least one or more Courses. Each course is taught by a single Professor. To maintain instruction quality, a Professor can deliver only one course

Step 1) Entity Identification

We have three entities

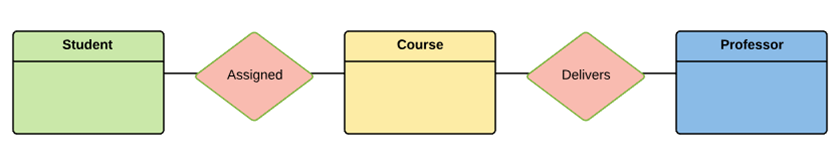
* Student
* Course
* Professor



Step 2) Relationship Identification

We have the following two relationships

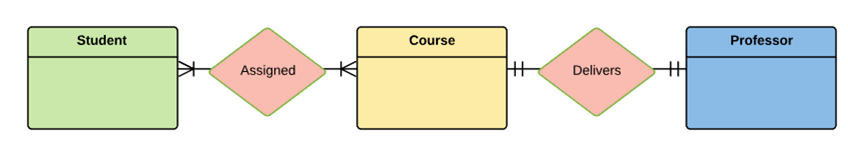
* The student is **assigned** a course
* Professor **delivers** a course



Step 3) Cardinality Identification

For them problem statement we know that,

* A student can be assigned **multiple** courses
* A Professor can deliver only **one** course



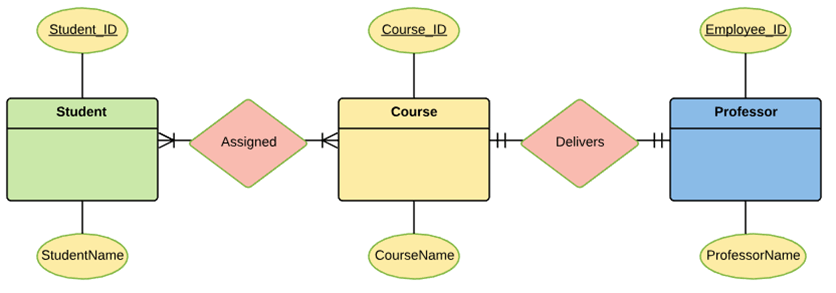
Step 4) Identify Attributes

You need to study the files, forms, reports, data currently maintained by the organization to identify attributes. You can also conduct interviews with various stakeholders to identify entities. Initially, it's important to identify the attributes without mapping them to a particular entity.

Once, you have a list of Attributes, you need to map them to the identified entities. Ensure an attribute is to be paired with exactly one entity. If you think an attribute should belong to more than one entity, use a modifier to make it unique.

Once the mapping is done, identify the primary Keys. If a unique key is not readily available, create one.

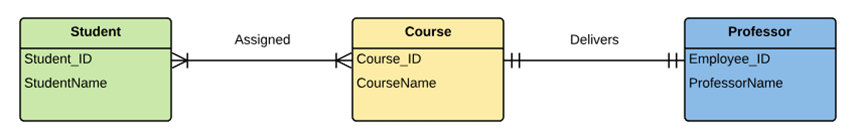
|  |  |  |
| --- | --- | --- |
| Entity | Primary Key | Attribute |
| Student | Student\_ID | StudentName |
| Professor | Employee\_ID | ProfessorName |
| Course | Course\_ID | CourseName |



For Course Entity, attributes could be Duration, Credits, Assignments, etc. For the sake of ease we have considered just one attribute.

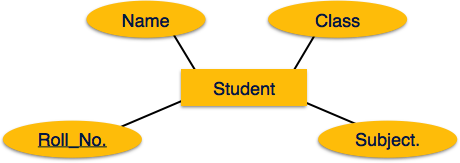
Step 5) Create the ERD

A more modern representation of ERD Diagram



Mapping Entity

An entity is a real-world object with some attributes.

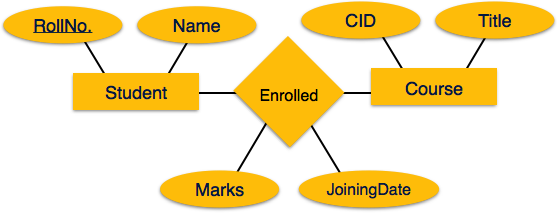


Mapping Process (Algorithm)

* Create table for each entity.
* Entity's attributes should become fields of tables with their respective data types.
* Declare primary key.

Mapping Relationship

A relationship is an association among entities.



Mapping Process

* Create a table for every entity
* Create table for a relationship.
* Add the primary keys of all participating Entities as fields of table with their respective data types.
* If relationship has any attribute, add each attribute as field of table.
* Declare a primary key composing all the primary keys of participating entities.
* Declare all foreign key constraints.

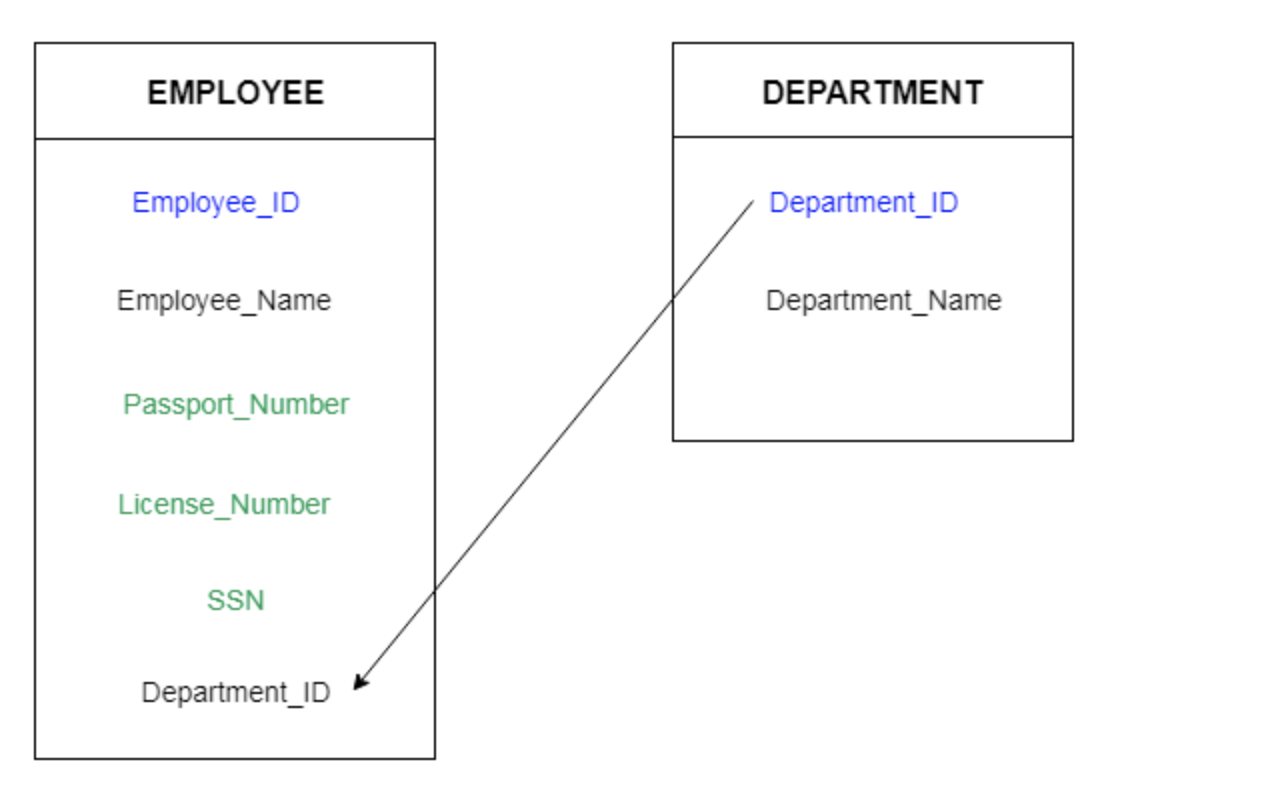
**Super Key:** An attribute or set of attributes that uniquely identifies a tuple within a relation

**Candidate key:** A super key such that no proper subset is a super key within the relation

**Primary key:** The candidate key that is selected to identify tuples uniquely within the relation, the candidate keys which are not selected as PKs are called "Alternate keys"

**Foreign key**. One or more attributes in an entity type that represents a key, either primary or secondary, in another entity type.

* Foreign keys are the column of the table which is used to point to the primary key of another table.
* In a company, every employee works in a specific department, and employee and department are two different entities. So we can't store the information of the department in the employee table. That's why we link these two tables through the primary key of one table.
* We add the primary key of the DEPARTMENT table, Department\_Id as a new attribute in the EMPLOYEE table.
* Now in the EMPLOYEE table, Department\_Id is the foreign key, and both the tables are related.



## Functional Dependency

Functional dependency (FD) is a set of constraints between two attributes in a relation. Functional dependency says that if two tuples have the same values for attributes A1, A2,..., An, then those two tuples must have to have the same values for attributes B1, B2, ..., Bn.

Functional dependency is represented by an arrow sign (→) that is, X→Y, where X functionally determines Y. The left-hand side attributes determine the values of attributes on the right-hand side.

X → Y   
Emp\_id -> Emp\_name

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

For example:

Assume we have an employee table with attributes: Emp\_Id, Emp\_Name, Emp\_Address.

1. Emp\_Id, Emp\_Name -> Emp\_Name, Emp\_Address

2. Emp\_Id -> Empl\_Address

Here Emp\_Id attribute can uniquely identify the Emp\_Name attribute of employee table because if we know the Emp\_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

Emp\_Id → Emp\_Name

Armstrong's Axioms

If F is a set of functional dependencies then the closure of F, denoted as F+, is the set of all functional dependencies logically implied by F. Armstrong's Axioms are a set of rules, that when applied repeatedly, generates a closure of functional dependencies.

F:

X -> Y  
X->Z

Z->A

-------------------------------------------------------

X->Y,Z  
X->Y,Z,A

X->X  
Y->Y  
X->XZ

## 

A,B ->C

B,C->D

A,B->A,B,C,D

## 1. Reflexive Rule (IR1)

In the reflexive rule, if Y is a subset of X, then X determines Y.

X -> a, b, c

Y-> a,b

1. If X ⊇ Y then X → Y

**Example:**

1. X = {a, b, c, d, e}
2. Y = {a, b, c}

X->Y

1. X = {a, b, c, d, e}
2. Y = {a, b, c, f, g, h}
3. XY = {a,b,c,d,e,f,g,h}

## 2. Augmentation Rule (IR2)

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

1. If X → Y then XZ → YZ

**Example:**

1. For R(ABCD), **if** A → B then AC → BC

## 3. Transitive Rule (IR3)

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

1. If X → Y and Y → Z then X → Z

## 4. Union Rule (IR4)

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

1. If X → Y and X → Z then X → YZ

**Proof:**

1. X → Y (given)

2. X → Z (given)

3. X → XY (using IR2 on 1 by augmentation with X. Where XX = X)

4. XY → YZ (using IR2 on 2 by augmentation with Y)

5. X → YZ (using IR3 on 3 and 4)

## 5. Decomposition Rule (IR5)

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

1. If X → YZ then X → Y and X → Z

**Proof:**

1. X → YZ (given)

2. YZ → Y (using IR1 Rule)

3. X → Y (using IR3 on 1 and 2)

## 6. Pseudo transitive Rule (IR6)

In Pseudo transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

1. If X → Y and YZ → W then XZ → W

**Proof:**

1. X → Y (given)

2. WY → Z (given)

3. WX → WY (using IR2 on 1 by augmenting with W)

4. WX → Z (using IR3 on 3 and 2)

## Trivial Functional Dependency

* **Trivial** − If a functional dependency (FD) X → Y holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.
* **Non-trivial** − If an FD X → Y holds, where Y is not a subset of X, then it is called a non-trivial FD.
* **Completely non-trivial** − If an FD X → Y holds, where x intersect Y = Φ, it is said to be a completely non-trivial FD.

## Normalization

If a database design is not perfect, it may contain anomalies, which are like a bad dream for any database administrator. Managing a database with anomalies is next to impossible.

* **Update anomalies** − If data items are scattered and are not linked to each other properly, then it could lead to strange situations. For example, when we try to update one data item having its copies scattered over several places, a few instances get updated properly while a few others are left with old values. Such instances leave the database in an inconsistent state.
* **Deletion anomalies** − We tried to delete a record, but parts of it was left undeleted because of unawareness, the data is also saved somewhere else.
* **Insert anomalies** − We tried to insert data in a record that does not exist at all.

Normalization is a method to remove all these anomalies and bring the database to a consistent state.

* Normalization is the process of organizing the data in the database.
* Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.
* Normalization divides the larger table into the smaller table and links them using relationship.
* The normal form is used to reduce redundancy from the database table.

# 

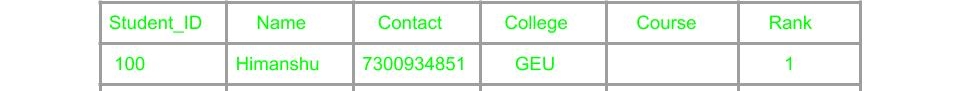
# The Problem of redundancy in Database

**Redundancy** means having multiple copies of the same data in the database. This problem arises when a database is not normalized. Suppose a table of student details attributes are: student Id, student name, college name, college rank, course opted.



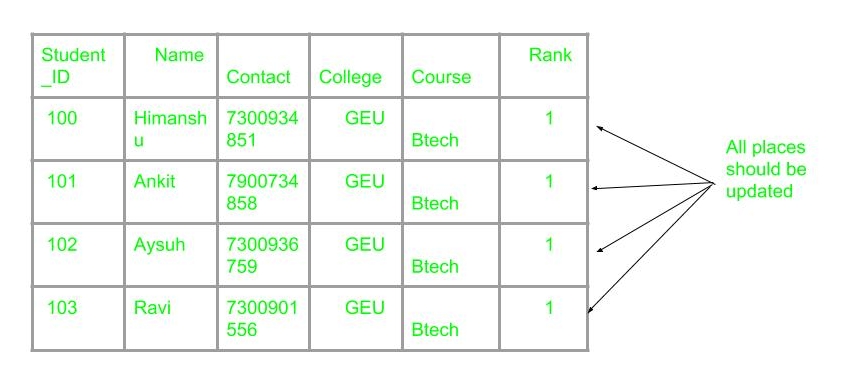
As it can be observed that values of attribute college name, college rank, course is being repeated which can lead to problems. Problems caused due to redundancy are: Insertion anomaly, Deletion anomaly, and Updation anomaly.

1. **Insertion Anomaly –**If a student detail has to be inserted whose course is not being decided yet then insertion will not be possible till the time course is decided for student.



This problem happens when the insertion of a data record is not possible without adding some additional unrelated data to the record.

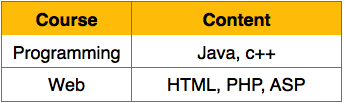
1. **Deletion Anomaly –**If the details of students in this table is deleted then the details of college will also get deleted which should not occur by common sense.  
   This anomaly happens when deletion of a data record results in losing some unrelated information that was stored as part of the record that was deleted from a table.
2. **Updation Anomaly –**Suppose if the rank of the college changes then changes will have to be all over the database which will be time-consuming and computationally costly.



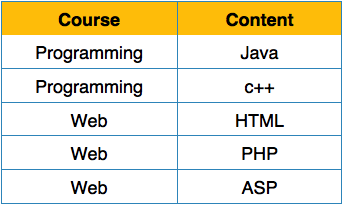
If updation do not occur at all places then database will be in inconsistent state.

## First Normal Form

First Normal Form is defined in the definition of relations (tables) itself. This rule defines that all the attributes in a relation must have atomic domains. The values in an atomic domain are indivisible units.



We re-arrange the relation (table) as below, to convert it to First Normal Form.



Each attribute must contain only a single value from its pre-defined domain.

* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
* First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

**Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP\_PHONE.

**EMPLOYEE table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385,  9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389,  8589830302 | Punjab |

The decomposition of the EMPLOYEE table into 1NF has been shown below:

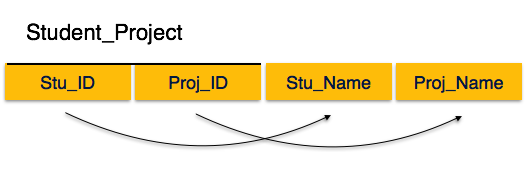
|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_NAME** | **EMP\_PHONE** | **EMP\_STATE** |
| 14 | John | 7272826385 | UP |
| 14 | John | 9064738238 | UP |
| 20 | Harry | 8574783832 | Bihar |
| 12 | Sam | 7390372389 | Punjab |
| 12 | Sam | 8589830302 | Punjab |

## Second Normal Form

Before we learn about the second normal form, we need to understand the following −

* **Prime attribute** − An attribute, which is a part of the candidate-key, is known as a prime attribute.
* **Non-prime attribute** − An attribute, which is not a part of the prime-key, is said to be a non-prime attribute.

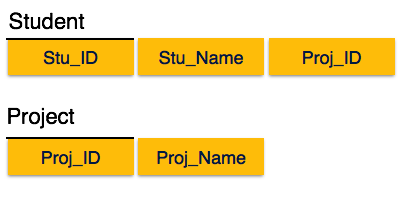
If we follow second normal form, then every non-prime attribute should be fully functionally dependent on prime key attribute. That is, if X → A holds, then there should not be any proper subset Y of X, for which Y → A also holds true.

  
107 101 Tarun LiFi

108 101 Aaravind LiFi

107 102 Tarun Road Detection  
  
BADDDD

We see here in Student\_Project relation that the prime key attributes are Stu\_ID and Proj\_ID. According to the rule, non-key attributes, i.e. Stu\_Name and Proj\_Name must be dependent upon both and not on any of the prime key attribute individually. But we find that Stu\_Name can be identified by Stu\_ID and Proj\_Name can be identified by Proj\_ID independently. This is called **partial dependency**, which is not allowed in Second Normal Form.



We broke the relation in two as depicted in the above picture. So there exists no partial dependency.

* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key

**Example:** Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

**TEACHER table**

|  |  |  |
| --- | --- | --- |
| **TEACHER\_ID** | **SUBJECT** | **TEACHER\_AGE** |
| 25 | Chemistry | 30 |
| 25 | Biology | 30 |
| 47 | English | 35 |
| 83 | Math | 38 |
| 83 | Computer | 38 |

In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

**TEACHER\_DETAIL table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **TEACHER\_AGE** |
| 25 | 30 |
| 47 | 35 |
| 83 | 38 |

**TEACHER\_SUBJECT table:**

|  |  |
| --- | --- |
| **TEACHER\_ID** | **SUBJECT** |
| 25 | Chemistry |
| 25 | Biology |
| 47 | English |
| 83 | Math |
| 83 | Computer |

## Third Normal Form

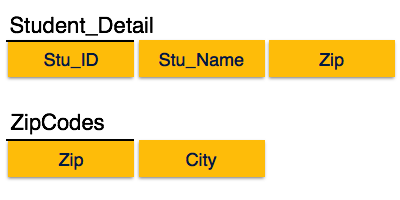
For a relation to be in Third Normal Form, it must be in Second Normal form and the following must satisfy −

* No non-prime attribute is transitively dependent on prime key attribute.
* For any non-trivial functional dependency, X → A, then either −
  + X is a superkey or,A is prime attribute.



We find that in the above Student\_detail relation, Stu\_ID is the key and only prime key attribute. We find that City can be identified by Stu\_ID as well as Zip itself. Neither Zip is a superkey nor is City a prime attribute. Additionally, Stu\_ID → Zip → City, so there exists **transitive dependency**.

To bring this relation into third normal form, we break the relation into two relations as follows −



# 

* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
* If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

**Example:**

**EMPLOYEE\_DETAIL table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. **EMP\_ID** | 1. **EMP\_NAME** | 1. **EMP\_ZIP** | 1. **EMP\_STATE** | 1. **EMP\_CITY** |
| 1. 222 | 1. Harry | 1. 201010 | 1. UP | 1. Noida |
| 1. 333 | 1. Stephan | 1. 02228 | 1. US | 1. Boston |
| 1. 444 | 1. Lan | 1. 60007 | 1. US | 1. Chicago |
| 1. 555 | 1. Katharine | 1. 06389 | 1. UK | 1. Norwich |
| 1. 666 | 1. John | 1. 462007 | 1. MP | 1. Bhopal |

**Super key in the table above:**

* 1. {EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME, EMP\_ZIP}....so on

**Candidate key:** {EMP\_ID}

**Non-prime attributes:** In the given table, all attributes except EMP\_ID are non-prime.

Here, EMP\_STATE & EMP\_CITY dependent on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively dependent on super key(EMP\_ID). It violates the rule of third normal form.

That's why we need to move the EMP\_CITY and EMP\_STATE to the new <EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key.

**EMPLOYEE table:**

|  |  |  |
| --- | --- | --- |
| 1. **EMP\_ID** | 1. **EMP\_NAME** | 1. **EMP\_ZIP** |
| 1. 222 | 1. Harry | 1. 201010 |
| 1. 333 | 1. Stephan | 1. 02228 |
| 1. 444 | 1. Lan | 1. 60007 |
| 1. 555 | 1. Katharine | 1. 06389 |
| 1. 666 | 1. John | 1. 462007 |

1. **EMPLOYEE\_ZIP table:**

|  |  |  |
| --- | --- | --- |
| 1. **EMP\_ZIP** | 1. **EMP\_STATE** | 1. **EMP\_CITY** |
| 1. 201010 | 1. UP | 1. Noida |
| 1. 02228 | 1. US | 1. Boston |
| 1. 60007 | 1. US | 1. Chicago |
| 1. 06389 | 1. UK | 1. Norwich |
| 1. 462007 | 1. MP | 1. Bhopal |

## Boyce-Codd Normal Form

Boyce-Codd Normal Form (BCNF) is an extension of Third Normal Form on strict terms. BCNF states that −

* For any non-trivial functional dependency, X → A, X must be a super-key.

In the above image, Stu\_ID is the super-key in the relation Student\_Detail and Zip is the super-key in the relation ZipCodes. So,

Stu\_ID → Stu\_Name, Zip

and

Zip → City

Which confirms that both the relations are in BCNF.

* BCNF is the advance version of 3NF. It is stricter than 3NF.
* A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
* For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

**Example:** Let's assume there is a company where employees work in more than one department.

**EMPLOYEE table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** | **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| 264 | India | Designing | D394 | 283 |
| 264 | India | Testing | D394 | 300 |
| 364 | UK | Stores | D283 | 232 |
| 364 | UK | Developing | D283 | 549 |

**In the above table Functional dependencies are as follows:**

1. EMP\_ID → EMP\_COUNTRY
2. EMP\_DEPT → {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate key: {EMP-ID, EMP-DEPT}**

The table is not in BCNF because neither EMP\_DEPT nor EMP\_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

**EMP\_COUNTRY table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_COUNTRY** |
| 264 | India |
|  |  |

**EMP\_DEPT table:**

|  |  |  |
| --- | --- | --- |
| **EMP\_DEPT** | **DEPT\_TYPE** | **EMP\_DEPT\_NO** |
| Designing | D394 | 283 |
| Testing | D394 | 300 |
| Stores | D283 | 232 |
| Developing | D283 | 549 |

**EMP\_DEPT\_MAPPING table:**

|  |  |
| --- | --- |
| **EMP\_ID** | **EMP\_DEPT** |
| D394 | 283 |
| D394 | 300 |
| D283 | 232 |
| D283 | 549 |

**Functional dependencies:**

1. EMP\_ID → EMP\_COUNTRY
2. EMP\_DEPT → {DEPT\_TYPE, EMP\_DEPT\_NO}

**Candidate keys:**

**For the first table:** EMP\_ID

**For the second table:** EMP\_DEPT

**For the third table:** {EMP\_ID, EMP\_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

**From a purist point of view you want to normalize your data structures as much as possible, but from a practical point of view you will find that you need to 'back out" of some of your normalizations for performance reasons. This is called "denormalization".**

# 

# Transaction

* The transaction is a set of logically related operation. It contains a group of tasks.
* A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

**Example:** Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

**X's Account**

1. Open\_Account(X)
2. Old\_Balance = X.balance
3. New\_Balance = Old\_Balance - 800
4. Gave you the money
5. X.balance = New\_Balance
6. Close\_Account(X)

**Y's Account**

1. Open\_Account(Y)
2. Old\_Balance = Y.balance
3. New\_Balance = Old\_Balance + 800
4. Y.balance = New\_Balance
5. Close\_Account(Y)

## Operations of Transaction:

Following are the main operations of transaction:

**Read(X):** Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

**Write(X):** Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. 1. R(X);
2. 2. X = X - 500;
3. 3. W(X);

Let's assume the value of X before starting of the transaction is 4000.

* The first operation reads X's value from database and stores it in a buffer.
* The second operation will decrease the value of X by 500. So buffer will contain 3500.
* The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

**For example:** If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

**Commit:** It is used to save the work done permanently.

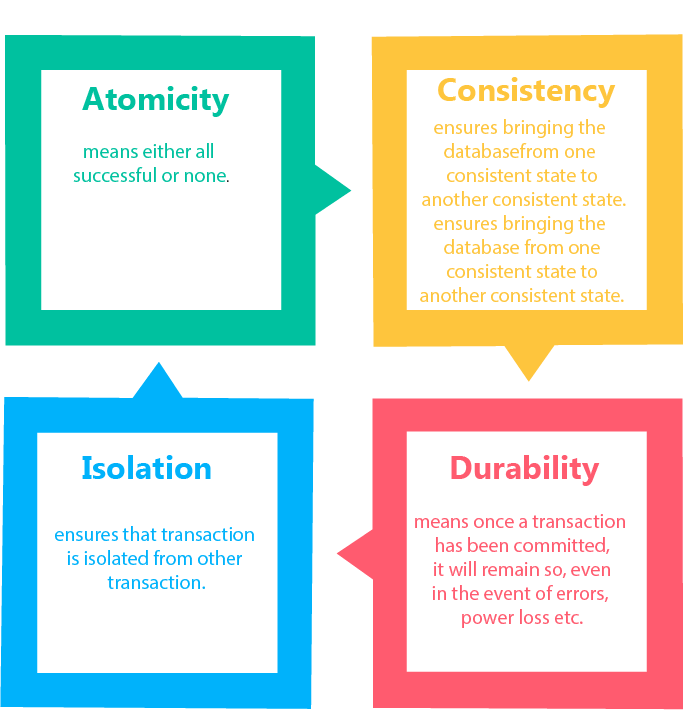
**Rollback:** It is used to undo the work done.

# Transaction property

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

## Property of Transaction

1. Atomicity
2. Consistency
3. Isolation
4. Durability



## Atomicity

* It states that all operations of the transaction take place at once if not, the transaction is aborted.
* There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

**Abort:** If a transaction aborts then all the changes made are not visible.

**Commit:** If a transaction commits then all the changes made are visible.

**Example:** Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

|  |  |
| --- | --- |
| **T1** | **T2** |
| Read(A)  A:= A-100  Write(A) | Read(B)  Y:= Y+100  Write(B) |

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.

If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

## Consistency

* The integrity constraints are maintained so that the database is consistent before and after the transaction.
* The execution of a transaction will leave a database in either its prior stable state or a new stable state.
* The consistent property of database states that every transaction sees a consistent database instance.
* The transaction is used to transform the database from one consistent state to another consistent state.

**For example:** The total amount must be maintained before or after the transaction.

1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

## Isolation

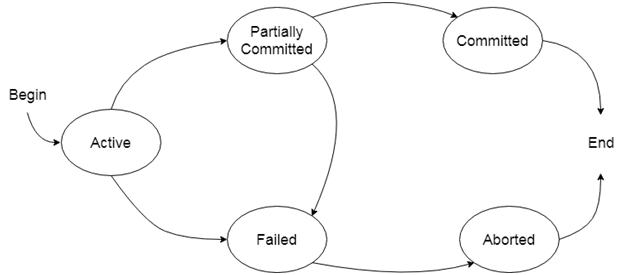
* It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
* In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
* The concurrency control subsystem of the DBMS enforced the isolation property.

## Durability

* The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
* They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
* The recovery subsystem of the DBMS has the responsibility of Durability property.

# States of Transaction

In a database, the transaction can be in one of the following states -



### Active state

* The active state is the first state of every transaction. In this state, the transaction is being executed.
* For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

### Partially committed

* In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
* In the total mark calculation example, a final display of the total marks step is executed in this state.

### Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

### Failed state

* If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
* In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

### Aborted

* If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
* If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
* After aborting the transaction, the database recovery module will select one of the two operations:
  1. Re-start the transaction
  2. Kill the transaction

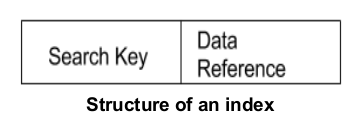
### Indexing

Indexing is a way to optimize performance of a database by minimizing the number of disk accesses required when a query is processed.

An index or database index is a data structure which is used to quickly locate and access the data in a database table.

Indexes are created using some database columns.

* The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly (Note that the data may or may not be stored in sorted order).
* The second column is the Data Reference which contains a set of pointers holding the address of the disk block where that particular key value can be found.



### B-Tree

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like [AVL](https://www.geeksforgeeks.org/avl-tree-set-1-insertion/) and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc ) require O(h) disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

**Properties of B-Tree**

**1)** All leaves are at the same level.

**2)** A B-Tree is defined by the term *minimum degree* ‘t’. The value of t depends upon disk block size.

**3)** Every node except root must contain at least t-1 keys. Root may contain minimum 1 key.

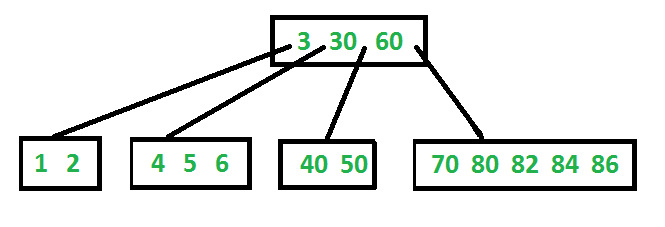
**4)** All nodes (including root) may contain at most 2t – 1 keys.

**5)** Number of children of a node is equal to the number of keys in it plus 1.

**6)** All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

**7)** Like other balanced Binary Search Trees, time complexity to search, insert and delete is O(Logn).

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



**Search**

Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don’t find k in the leaf node, we return NULL.

**Traverse**

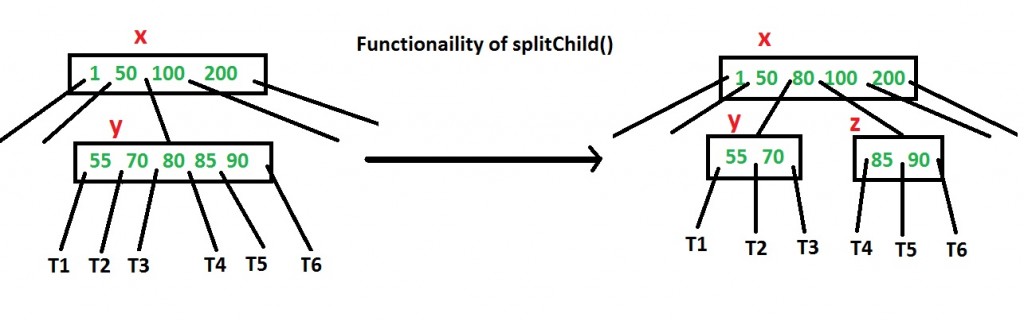
Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

**Insert**

A new key is always inserted at the leaf node. Let the key to be inserted be k.

Like BST, we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on the number of keys that a node can contain. So before inserting a key to the node, we make sure that the node has extra space.

*How to make sure that a node has space available for a key before the key is inserted?* We use an operation called splitChild() that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z. Note that the splitChild operation moves a key up and this is the reason B-Trees grow up, unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is the complete algorithm.

**Insertion**

**1)** Initialize x as root.

**2)** While x is not leaf, do following

..**a)** Find the child of x that is going to be traversed next. Let the child be y.

..**b)** If y is not full, change x to point to y.

..**c)** If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.

**3)** The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

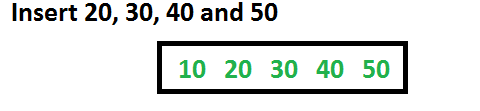
Note that the algorithm follows the Cormen book. It is actually a proactive insertion algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don’t split a node before going down to it and split it only if a new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from the root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because the parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree ‘t’ as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.

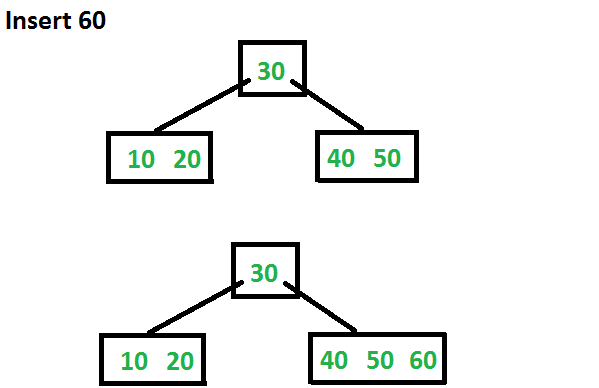
Initially root is NULL. Let us first insert 10.



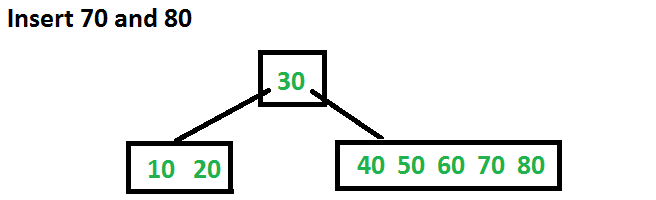
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because the maximum number of keys a node can accommodate is 2\*t – 1 which is 5.



Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.



Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

