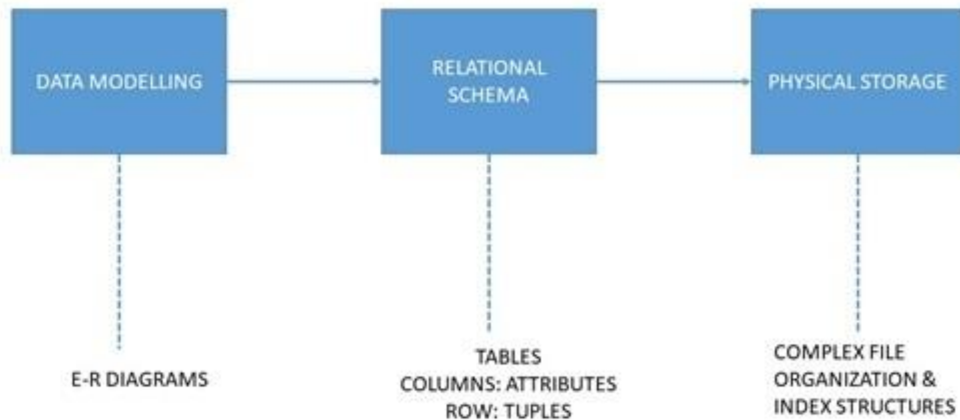
INTRODUCTION

Relational Database Model

The relational data model was introduced by C. F. Codd in 1970. Currently, it is the most widely used data model. The relational data model describes the world as “a collection of inter-related relations (or tables).” A relational data model involves the use of data tables that collect groups of elements into relations. These models work based on the idea that each table setup will include a primary key or identifier. Other tables use that identifier to provide "relational" data links and results.

Today, there are many commercial Relational Database Management System (RDBMS), such as Oracle, IBM DB2, and Microsoft SQL Server. There are also many free and open-source RDBMS, such as MySQL, mSQL (mini-SQL) and the embedded Java DB (Apache Derby). Database administrators use Structured Query Language (SQL) to retrieve data elements from a relational database.



As mentioned, the primary key is a fundamental tool in creating and using relational data models. It must be unique for each member of a data set. It must be populated for all members. Inconsistencies can cause problems in how developers retrieve data. Other issues with relational database designs include excessive duplication of data, faulty or partial data, or improper links or associations between tables. A large part of routine database administration involves evaluating all the data sets in a database to make sure that they are consistently populated and will respond well to SQL or any other data retrieval method.

For example, a conventional database row would represent a tuple, which is a set of data that revolves around an instance or virtual object so that the primary key is its unique identifier. A column name in a data table is associated with an attribute, an identifier or feature that all parts of a data set have. These and other strict conventions help to provide database administrators and designers with standards for crafting relational database setups.

Database Design Objective

- **Eliminate Data Redundancy:** the same piece of data shall not be stored in more than one place. This is because duplicate data not only waste storage spaces but also easily lead to inconsistencies.
- **Ensure Data Integrity and Accuracy:** is the maintenance of, and the assurance of the accuracy and consistency of, data over its entire life-cycle, and is a critical aspect to the design, implementation, and usage of any system which stores, processes, or retrieves data.

The relational model has provided the basis for:

- Research on the theory of data/relationship/constraint
- Numerous database design methodologies
- The standard database access language called structured query language (SQL)
- Almost all modern commercial database management systems

Relational databases go together with the development of SQL. The simplicity of SQL - where even a novice can learn to perform basic queries in a short period of time - is a large part of the reason for the popularity of the relational model.

The two tables below relate to each other through the product code field. Any two tables can relate to each other simply by creating a field they have in common.

Table 1

Product_code	Description	Price
A416	Colour Pen	₹ 25.00
C923	Pencil box	₹ 45.00

Table 2

Invoice_code	Invoice_line	Product_code	Quantity
3804	1	A416	15
3804	2	C923	24

There are four stages of an RDM which are as follows −

- **Relations and attributes** − The various tables and attributes related to each table are identified. The tables represent entities, and the attributes represent the properties of the respective entities.
- **Primary keys** − The attribute or set of attributes that help in uniquely identifying a record is identified and assigned as the primary key.

- **Relationships** −The relationships between the various tables are established with the help of foreign keys. Foreign keys are attributes occurring in a table that are primary keys of another table. The types of relationships that can exist between the relations (tables) are One to one, One to many, and Many to many
- **Normalization** − This is the process of optimizing the database structure. Normalization simplifies the database design to avoid redundancy and confusion. The different normal forms are as follows

    1. First normal form
    2. Second normal form
    3. Third normal form
    4. Boyce-Codd normal form
    5. Fifth normal form

By applying a set of rules, a table is normalized into the above normal forms in a linearly progressive fashion. The efficiency of the design gets better with each higher degree of normalization.

Advantages of Relational Databases

The main advantages of relational databases are that they enable users to easily categorize and store data that can later be queried and filtered to extract specific information for reports. Relational databases are also easy to extend and aren't reliant on the physical organization. After the original database creation, a new data category can be added without all existing applications being modified.

Other Advantages

- **Accurate** − Data is stored just once, which eliminates data deduplication.

- **Flexible** − Complex queries are easy for users to carry out.
- **Collaborative** −Multiple users can access the same database.
- **Trusted** −Relational database models are mature and well-understood.
- **Secure** − Data in tables within relational database management systems (RDBMS) can be limited to allow access by only particular users.

Relational Database Design (RDD)

Definition - What does *Relational Database Design (RDD)* mean

Relational database design (RDD) models information and data into a set of tables with rows and columns. Each row of a relation/table represents a record, and each column represents an attribute of data. The Structured Query Language (SQL) is used to manipulate relational databases. The design of a relational database is composed of four stages, where the data are modeled into a set of related tables. The stages are:

- Define relations/attributes
- Define primary keys
- Define relationships
- Normalization

Techopedia explains *Relational Database Design (RDD)*

Relational databases differ from other databases in their approach to organizing data and performing transactions. In an RDD, the data are organized into tables and all types of data access are carried out via controlled transactions. Relational database design satisfies the ACID (atomicity, consistency, integrity and durability) properties required from a database design. Relational database design mandates the use of a database server in applications for dealing with data management problems.

The four stages of an RDD are as follows:

- **Relations and attributes**: The various tables and attributes related to each table are identified. The tables represent entities, and the attributes represent the properties of the respective entities.
- **Primary keys**: The attribute or set of attributes that help in uniquely identifying a record is identified and assigned as the primary key
- **Relationships:** The relationships between the various tables are established with the help of foreign keys. Foreign keys are attributes occurring in a table that are primary keys of another table. The types of relationships that can exist between the relations (tables) are:
    - One to one
    - One to many
    - Many to many

An entity-relationship diagram can be used to depict the entities, their attributes and the relationship between the entities in a diagrammatic way.

- **Normalization**: This is the process of optimizing the database structure. Normalization simplifies the database design to avoid redundancy and confusion. The different normal forms are as follows:

  - First normal form
  - Second normal form
  - Third normal form
  - Boyce-Codd normal form
  - Fifth normal form

By applying a set of rules, a table is normalized into the above normal forms in a linearly progressive fashion. The efficiency of the design gets better with each higher degree of normalization.

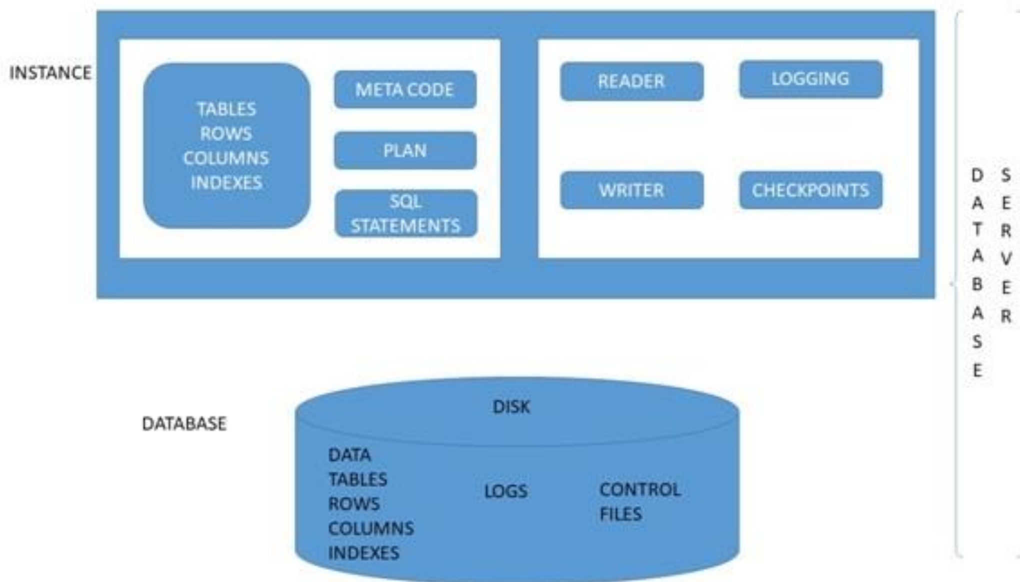
Relational Database Management System (RDMS)

Relational database design (RDD) models' information and data into a set of tables with rows and columns. Each row of a relation/table represents a record, and each column represents an attribute of data. The Structured Query Language (SQL) is used to manipulate relational databases. The design of a relational database is composed of four stages, where the data are modeled into a set of related tables. The stages are −

- Define relations/attributes
- Define primary keys
- Define relationships
- Normalization

Relational databases differ from other databases in their approach to organizing data and performing transactions. In an RDD, the data are organized into tables and all types of data access are carried out via controlled transactions. Relational

database design satisfies the ACID (atomicity, consistency, integrity, and durability) properties required from a database design. Relational database design mandates the use of a database server in applications for dealing with data management problems.



Relational Database Design Process

Database design is more art than science, as you have to make many decisions. Databases are usually customized to suit a particular application. No two customized applications are alike, and hence, no two databases are alike. Guidelines (usually in terms of what not to do instead of what to do) are provided in making these design decision, but the choices ultimately rest on the designer.

Step 1 − Define the Purpose of the Database (Requirement Analysis)

- Gather the requirements and define the objective of your database.
- Drafting out the sample input forms, queries and reports often help.

Step 2 − Gather Data, Organize in tables and Specify the Primary Keys

- Once you have decided on the purpose of the database, gather the data that are needed to be stored in the database. Divide the data into subject-based tables.
- Choose one column (or a few columns) as the so-called primary key, which uniquely identifies the each of the rows.

Step 3 − Create Relationships among Tables

A database consisting of independent and unrelated tables serves little purpose (you may consider using a spreadsheet instead). The power of a relational database lies in the relationship that can be defined between tables. The most crucial aspect in designing a relational database is to identify the relationships among tables. The types of relationship include:

- one-to-many
- many-to-many
- one-to-one

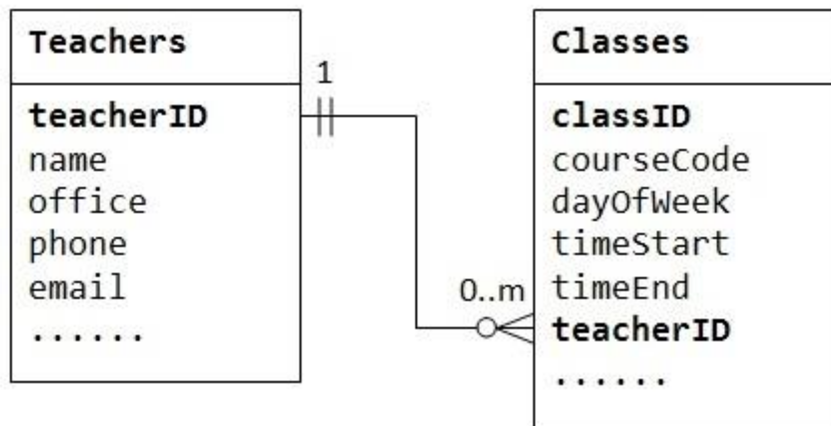
One-to-Many

In a "class roster" database, a teacher may teach zero or more classes, while a class is taught by one (and only one) teacher. In a "company" database, a manager manages zero or more employees, while an employee is managed by one (and only one) manager. In a "product sales" database, a customer may place many orders; while an order is placed by one particular customer. This kind of relationship is known as one-to-many.

The one-to-many relationship cannot be represented in a single table. For example, in a "class roster" database, we may begin with a table called Teachers, which stores information about teachers (such as name, office, phone, and email). To

store the classes taught by each teacher, we could create columns class1, class2, class3, but faces a problem immediately on how many columns to create. On the other hand, if we begin with a table called Classes, which stores information about a class, we could create additional columns to store information about the (one) teacher (such as name, office, phone, and email). However, since a teacher may teach many classes, its data would be duplicated in many rows in table Classes.

To support a one-to-many relationship, we need to design two tables: for e.g. a table Classes to store information about the classes with classID as the primary key; and a table Teachers to store information about teachers with teacherID as the primary key. We can then create the one-to-many relationship by storing the primary key of the table Teacher (i.e., teacherID) (the "one"-end or the parent table) in the table classes (the "many"-end or the child table), as illustrated below.



The column teacherID in the child table Classes is known as the foreign key. A foreign key of a child table is a primary key of a parent table, used to reference the parent table.

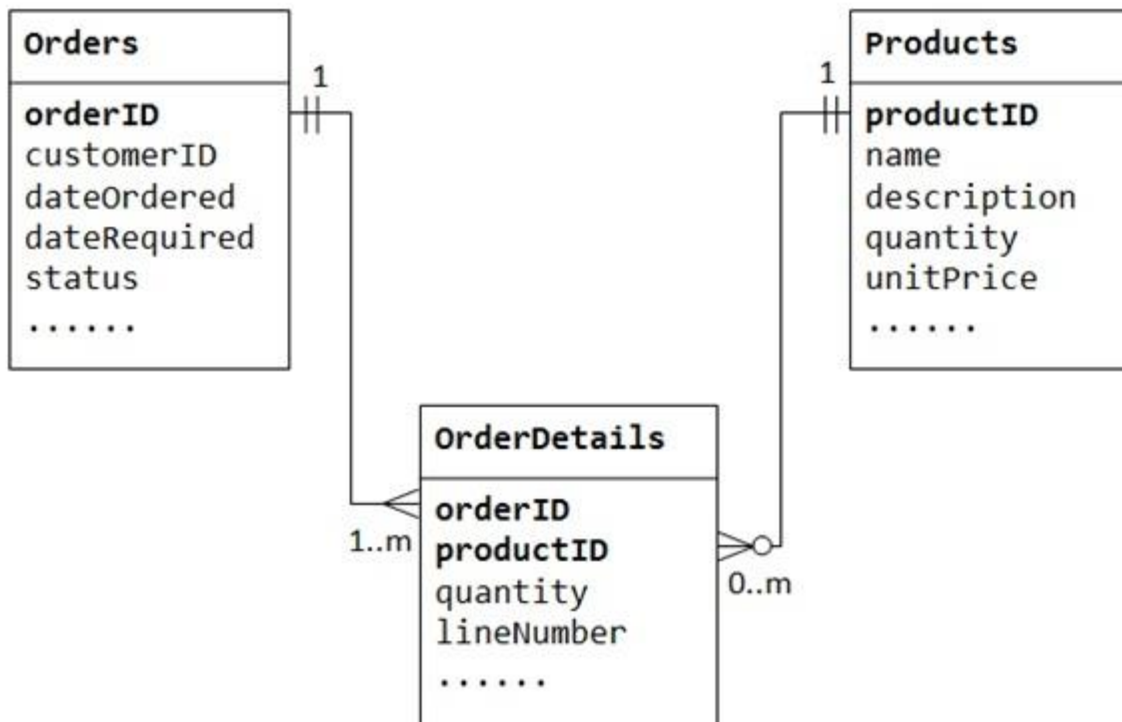
Many-to-Many

In a "product sales" database, a customer's order may contain one or more products; and a product can appear in many orders. In a "bookstore" database, a

book is written by one or more authors; while an author may write zero or more books. This kind of relationship is known as many-to-many.

Let's illustrate with a "product sales" database. We begin with two tables: Products and Orders. The table products contain information about the products (such as name, description and quantityInStock) with productID as its primary key. The table orders contain customer's orders (customerID, dateOrdered, dateRequired and status). Again, we cannot store the items ordered inside the Orders table, as we do not know how many columns to reserve for the items. We also cannot store the order information in the Products table.

To support many-to-many relationship, we need to create a third table (known as a junction table), say OrderDetails (or OrderLines), where each row represents an item of a particular order. For the OrderDetails table, the primary key consists of two columns: orderID and productID, that uniquely identify each row. The columns orderID and productID in OrderDetails table are used to reference Orders and Products tables, hence, they are also the foreign keys in the OrderDetails table.



The many-to-many relationship is, in fact, implemented as two one-to-many relationships, with the introduction of the junction table.

An order has many items in OrderDetails. An OrderDetails item belongs to one particular order.

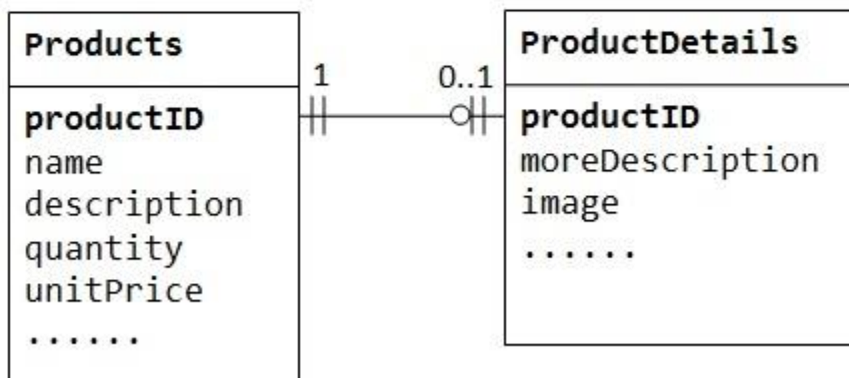
A product may appear in many OrderDetails. Each OrderDetails item specified one product.

One-to-One

In a "product sales" database, a product may have optional supplementary information such as image, more description and comment. Keeping them inside the Products table results in many empty spaces (in those records without these optional data). Furthermore, these large data may degrade the performance of the database.

Instead, we can create another table (say ProductDetails, ProductLines or ProductExtras) to store the optional data. A record will only be created for those products with optional data. The two tables, Products and ProductDetails, exhibit a one-to-one relationship. That is, for every row in the parent table, there is at most one row (possibly zero) in the child table. The same column productID should be used as the primary key for both tables.

Some databases limit the number of columns that can be created inside a table. You could use a one-to-one relationship to split the data into two tables. A one-to-one relationship is also useful for storing certain sensitive data in a secure table, while the non-sensitive ones in the main table.



Column Data Types

You need to choose an appropriate data type for each column. Commonly data types include integers, floating-point numbers, string (or text), date/time, binary, collection (such as enumeration and set).

Step 4 − Refine & Normalize the Design

For example,

- adding more columns,

- create a new table for optional data using one-to-one relationship,
- split a large table into two smaller tables,
- Other methods.

Normalization

Apply the so-called normalization rules to check whether your database is structurally correct and optimal.

First Normal Form (1NF): A table is 1NF if every cell contains a single value, not a list of values. This property is known as atomic. 1NF also prohibits a repeating group of columns such as item1, item2, itemN. Instead, you should create another table using a one-to-many relationship.

Second Normal Form (2NF) − A table is 2NF if it is 1NF and every non-key column is fully dependent on the primary key. Furthermore, if the primary key is made up of several columns, every non-key column shall depend on the entire set and not part of it.

For example, the primary key of the OrderDetails table comprising orderID and productID. If unitPrice is dependent only on productID, it shall not be kept in the OrderDetails table (but in the Products table). On the other hand, if the unit price is dependent on the product as well as the particular order, then it shall be kept in the OrderDetails table.

Third Normal Form (3NF) − A table is 3NF if it is 2NF and the non-key columns are independent of each other. In other words, the non-key columns are dependent on primary key, only on the primary key and nothing else. For example, suppose that we have a Products table with columns productID (primary key), name and unitPrice. The column discountRate shall not belong to the Products table if it is also dependent on the unitPrice, which is not part of the primary key.

Higher Normal Form: 3NF has its inadequacies, which leads to a higher Normal form, such as Boyce/Codd Normal form, Fourth Normal Form (4NF) and Fifth Normal Form (5NF), which is beyond the scope of this tutorial.

At times, you may decide to break some of the normalization rules, for performance reason (e.g., create a column called totalPrice in Orders table which can be derived from the orderDetails records); or because the end-user requested for it. Make sure that you fully aware of it, develop programming logic to handle it, and properly document the decision.

Integrity Rules

You should also apply the integrity rules to check the integrity of your design −

1. **Entity Integrity Rule** − The primary key cannot contain NULL. Otherwise, it cannot uniquely identify the row. For composite key made up of several columns, none of the columns can contain NULL. Most of the RDBMS check and enforce this rule.

2. **Referential Integrity Rule** − Each foreign key value must be matched to a primary key value in the table referenced (or parent table).

You can insert a row with a foreign key in the child table only if the value exists in the parent table.

If the value of the key changes in the parent table (e.g., the row updated or deleted), all rows with this foreign key in the child table(s) must be handled accordingly. You could either (a) disallow the changes; (b) cascade the change (or delete the records) in the child tables accordingly; (c) set the key value in the child tables to NULL.

Most RDBMS can be set up to perform the check and ensure the referential integrity, in a specified manner.

3.Business logic Integrity − Besides the above two general integrity rules, there could be integrity (validation) pertaining to the business logic, e.g., zip code shall be 5-digit within a certain ranges, delivery date and time shall fall in the business hours; quantity ordered shall be equal or less than quantity in stock, etc. These could be carried out invalidation rule (for the specific column) or programming logic.

Column Indexing

You could create an index on the selected column(s) to facilitate data searching and retrieval. An index is a structured file that speeds up data access for SELECT but may slow down INSERT, UPDATE, and DELETE. Without an index structure, to process a SELECT query with a matching criterion (e.g., SELECT * FROM Customers WHERE name='Tan Ah Teck'), the database engine needs to compare every record in the table. A specialized index (e.g., in BTREE structure) could reach the record without comparing every record. However, the index needs to be rebuilt whenever a record is changed, which results in overhead associated with using indexes.

The index can be defined on a single column, a set of columns (called concatenated index), or part of a column (e.g., first 10 characters of a VARCHAR(100)) (called partial index). You could build more than one index in a table. For example, if you often search for a customer using either customerName or phone number, you could speed up the search by building an index on column customerName, as well as phoneNumber. Most RDBMS builds an index on the primary key automatically.

DBMS - Normalization

Functional Dependency

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

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.

- **Reflexive rule** − If alpha is a set of attributes and beta is_subset_of alpha, then alpha holds beta.
- **Augmentation rule** − If a → b holds and y is attribute set, then ay → by also holds. That is adding attributes in dependencies, does not change the basic dependencies.
- **Transitivity rule** − Same as transitive rule in algebra, if a → b holds and b → c holds, then a → c also holds. a → b is called as a functionally that determines b.

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.

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.

Course	Content
Programming	Java, c++
Web	HTML, PHP, ASP

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

Course	Content
Programming	Java
Programming	c++
Web	HTML
Web	PHP
Web	ASP

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

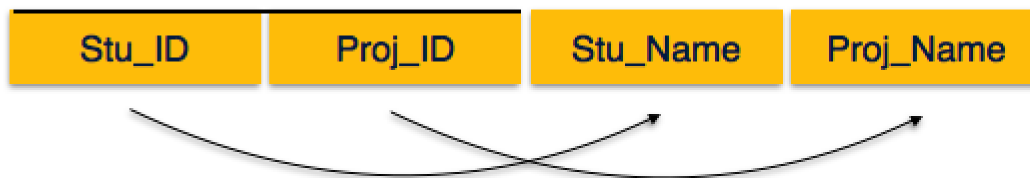
Second Normal Form

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.



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.

Student

Project

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

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 −

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.

Functional Dependencies

A *functional dependency* (FD) is a relationship between two attributes, typically between the PK and other non-key attributes within a table. For any relation R, attribute Y is functionally dependent on attribute X (usually the PK), if for every valid instance of X, that value of X uniquely determines the value of Y. This relationship is indicated by the representation below :

X ———–> Y

The left side of the above FD diagram is called the *determinant*, and the right side is the *dependent*. Here are a few examples.

In the first example, below, SIN determines Name, Address and Birthdate. Given SIN, we can determine any of the other attributes within the table.

SIN ———–> Name, Address, Birthdate

For the second example, SIN and Course determine the date completed (DateCompleted). This must also work for a composite PK.

SIN, Course ———> DateCompleted

The third example indicates that ISBN determines Title.

ISBN ———> Title

Rules of Functional Dependencies

Consider the following table of data r(R) of the relation schema R(ABCDE)

As you look at this table, ask yourself: *What kind of dependencies can we observe among the attributes in Table R?* Since the values of A are unique (a1, a2, a3, etc.), it follows from the FD definition that:

$A \rightarrow B, \quad A \rightarrow C, \quad A \rightarrow D, \quad A \rightarrow E$

- It also follows that $A \rightarrow BC$ (or any other subset of ABCDE).
- This can be summarized as $A \rightarrow BCDE$.
- From our understanding of primary keys, A is a primary key.

Since the values of E are always the same (all e1), it follows that:

$A \rightarrow E, \quad B \rightarrow E, \quad C \rightarrow E, \quad D \rightarrow E$

However, we cannot generally summarize the above with $ABCD \rightarrow E$ because, in general, $A \rightarrow E$, $B \rightarrow E$, $AB \rightarrow E$.

Other observations:

1. Combinations of BC are unique, therefore BC → ADE.
2. Combinations of BD are unique, therefore BD → ACE.
3. If C values match, so do D values.
1. Therefore, C → D
2. However, D values don't determine C values
3. So C does not determine D, and D does not determine C.

Looking at actual data can help clarify which attributes are dependent and which are determinants.

Inference Rules

Armstrong's axioms are a set of inference rules used to infer all the functional dependencies on a relational database. They were developed by William W. Armstrong. The following describes what will be used, in terms of notation, to explain these axioms.

Let R(U) be a relation scheme over the set of attributes U. We will use the letters X, Y, Z to represent any subset of and, for short, the union of two sets of attributes, instead of the usual X U Y.

Functional dependency

What is Functional Dependency

Functional dependency in DBMS, as the name suggests is a relationship between attributes of a table dependent on each other. Introduced by E. F. Codd, it helps in preventing data redundancy and gets to know about bad designs.

To understand the concept thoroughly, let us consider P is a relation with attributes A and B. Functional Dependency is represented by -> (arrow sign)

Then the following will represent the functional dependency between attributes with an arrow sign −

A -> B

Above suggests the following:

Functional Dependency

A -> B

B - functionally dependent on **A**
A - determinant set
B - dependent attribute

Example

The following is an example that would make it easier to understand functional dependency −

We have a **<Department>** table with two attributes − **DeptId** and **DeptName**.

DeptId = Department ID **DeptName** = Department Name

The **DeptId** is our primary key. Here, **DeptId** uniquely identifies the **DeptName** attribute. This is because if you want to know the department name, then at first you need to have the **DeptId**.

DeptId	DeptName
001	Finance
002	Marketing
003	HR

Therefore, the above functional dependency between **DeptId** and **DeptName** can be determined as **DeptId** is functionally dependent on **DeptName** −

```
DeptId -> DeptName
```

Types of Functional Dependency

Functional Dependency has three forms −

- Trivial Functional Dependency
- Non-Trivial Functional Dependency
- Completely Non-Trivial Functional Dependency

Let us begin with Trivial Functional Dependency −

Trivial Functional Dependency

It occurs when B is a subset of A in −

```
A ->B
```

Example

We are considering the same **<Department>** table with two attributes to understand the concept of trivial dependency.

The following is a trivial functional dependency since **DeptId** is a subset of **DeptId** and **DeptName**

{ DeptId, DeptName } -> Dept Id

Non –Trivial Functional Dependency

It occurs when B is not a subset of A in −

A ->B

Example

DeptId -> DeptName

The above is a non-trivial functional dependency since DeptName is a not a subset of DeptId.

Completely Non - Trivial Functional Dependency

It occurs when A intersection B is null in −

A ->B

Armstrong's Axioms Property of Functional Dependency

Armstrong's Axioms property was developed by William Armstrong in 1974 to reason about functional dependencies.

The property suggests rules that hold true if the following are satisfied:

- **Transitivity**
  If A->B and B->C, then A->C i.e. a transitive relation.

- **Reflexivity**

  A-> B, if B is a subset of A.

- **Augmentation**

  The last rule suggests: AC->BC, if A->B

KEYS

Types of Keys in Database Management System

There are mainly seven different types of Keys in DBMS and each key has it's different functionality:

- **Super Key -** A super key is a group of single or multiple keys which identifies rows in a table.
- **Primary Key -** is a column or group of columns in a table that uniquely identify every row in that table.
- **Candidate Key -** is a set of attributes that uniquely identify tuples in a table. Candidate Key is a super key with no repeated attributes.
- **Alternate Key -** is a column or group of columns in a table that uniquely identify every row in that table.
- **Foreign Key -** is a column that creates a relationship between two tables. The purpose of Foreign keys is to maintain data integrity and allow navigation between two different instances of an entity.

- **Compound Key -** has two or more attributes that allow you to uniquely recognize a specific record. It is possible that each column may not be unique by itself within the database.
- **Composite Key -** An artificial key which aims to uniquely identify each record is called a surrogate key. These kind of key are unique because they are created when you don't have any natural primary key.
- **Surrogate Key -** An artificial key which aims to uniquely identify each record is called a surrogate key. These kind of key are unique because they are created when you don't have any natural primary key.

What is the Super key?

A superkey is a group of single or multiple keys which identifies rows in a table. A Super key may have additional attributes that are not needed for unique identification.

Example:

EmpSSN	EmpNum	Empname
9812345098	AB05	Shown
9876512345	AB06	Roslyn
199937890	AB07	James

In the above-given example, EmpSSN and EmpNum name are superkeys.

What is a Primary Key?

PRIMARY KEY is a column or group of columns in a table that uniquely identify every row in that table. The Primary Key can't be a duplicate meaning the same value can't appear more than once in the table. A table cannot have more than one primary key.

Rules for defining Primary key:

- Two rows can't have the same primary key value
- It must for every row to have a primary key value.
- The primary key field cannot be null.
- The value in a primary key column can never be modified or updated if any foreign key refers to that primary key.

Example:

In the following example, <code>StudID</code> is a Primary Key.

StudID	Roll No	First Name	LastName	Email
1	11	Tom	Price	abc@gmail.com
2	12	Nick	Wright	xyz@gmail.com
3	13	Dana	Natan	mno@yahoo.com

What is the Alternate key?

ALTERNATE KEYS is a column or group of columns in a table that uniquely identify every row in that table. A table can have multiple choices for a primary key but only one can be set as the primary key. All the keys which are not primary key are called an Alternate Key.

Example:

In this table, StudID, Roll No, Email are qualified to become a primary key. But since StudID is the primary key, Roll No, Email becomes the alternative key.

StudID	Roll No	First Name	LastName	Email
1	11	Tom	Price	abc@gmail.com
2	12	Nick	Wright	xyz@gmail.com
3	13	Dana	Natan	mno@yahoo.com

What is a Candidate Key?

CANDIDATE KEY is a set of attributes that uniquely identify tuples in a table. Candidate Key is a super key with no repeated attributes. The Primary key should be selected from the candidate keys. Every table must have at least a single candidate key. A table can have multiple candidate keys but only a single primary key.

Properties of Candidate key:

- It must contain unique values
- Candidate key may have multiple attributes
- Must not contain null values
- It should contain minimum fields to ensure uniqueness
- Uniquely identify each record in a table

Example: In the given table Stud ID, Roll No, and email are candidate keys which help us to uniquely identify the student record in the table.

StudID	Roll No	First Name	LastName	Email
1	11	Tom	Price	abc@gmail.com
2	12	Nick	Wright	xyz@gmail.com
3	13	Dana	Natan	mno@yahoo.com

What is the Foreign key?

FOREIGN KEY is a column that creates a relationship between two tables. The purpose of Foreign keys is to maintain data integrity and allow navigation between two different instances of an entity. It acts as a cross-reference between two tables as it references the primary key of another table.

Example:

DeptCode	DeptName
001	Science
002	English
005	Computer

Teacher ID	Fname	Lname
B002	David	Warner
B017	Sara	Joseph
B009	Mike	Brunton

In this key in dbms example, we have two table, teach and department in a school. However, there is no way to see which search work in which department.

In this table, adding the foreign key in Deptcode to the Teacher name, we can create a relationship between the two tables.

Teacher ID	DeptCode	Fname	Lname

B002	002	David	Warner
B017	002	Sara	Joseph
B009	001	Mike	Brunton

This concept is also known as Referential Integrity.

What is the Compound key?

COMPOUND KEY has two or more attributes that allow you to uniquely recognize a specific record. It is possible that each column may not be unique by itself within the database. However, when combined with the other column or columns the combination of column.

Boyce-Codd Normal Form (BCNF)

Application of the general definitions of 2NF and 3NF may identify additional redundancy caused by dependencies that violate one or more candidate keys. However, despite these additional constraints, dependencies can still exist that will cause redundancy to be present in 3NF relations. This weakness in 3NF, resulted in the presentation of a stronger normal form called Boyce–Codd Normal Form (Codd, 1974).

Although, 3NF is adequate normal form for relational database, still, this (3NF) normal form may not remove 100% redundancy because of X?Y functional dependency, if X is not a candidate key of given relation. This can be solve by Boyce-Codd Normal Form (BCNF).

Boyce-Codd Normal Form (BCNF):

Boyce–Codd Normal Form (BCNF) is based on functional dependencies that take into account all candidate keys in a relation; however, BCNF also has additional constraints compared with the general definition of 3NF.

A relation is in BCNF iff, X is superkey for every functional dependency (FD) X?Y in given relation.

In other words,

A relation is in BCNF, if and only if, every determinant is a Form (BCNF) candidate key.

Note – To test whether a relation is in BCNF, we identify all the determinants and make sure that they are candidate keys.

You came across a similar hierarchy known as **Chomsky Normal Form** in Theory of Computation. Now, carefully study the hierarchy above. It can be inferred that *every relation in BCNF is also in 3NF*. To put it another way, a relation in 3NF need not to be in BCNF. Ponder over this statement for a while.

To determine the highest normal form of a given relation R with functional dependencies, the first step is to check whether the BCNF condition holds. If R is found to be in BCNF, it can be safely deduced that the relation is also in 3NF, 2NF and 1NF as the hierarchy shows. The 1NF has the least restrictive constraint – it

only requires a relation R to have atomic values in each tuple. The 2NF has a slightly more restrictive constraint.
The 3NF has more restrictive constraint than the first two normal forms but is less restrictive than the BCNF. In this manner, the restriction increases as we traverse down the hierarchy.

Example-1:

Find the highest normal form of a relation R(A, B, C, D, E) with FD set as:

```
{ BC->D, AC->BE, B->E }
```

Explanation:

- **Step-1:** As we can see, (AC)+ ={A, C, B, E, D} but none of its subset can determine all attribute of relation, So AC will be candidate key. A or C can’t be derived from any other attribute of the relation, so there will be only 1 candidate key {AC}.
- **Step-2:** Prime attributes are those attribute which are part of candidate key {A, C} in this example and others will be non-prime {B, D, E} in this example.
- **Step-3:** The relation R is in 1st normal form as a relational DBMS does not allow multi-valued or composite attribute.

The relation is in 2nd normal form because BC->D is in 2nd normal form (BC is not a proper subset of candidate key AC) and AC->BE is in 2nd normal form (AC is candidate key) and B->E is in 2nd normal form (B is not a proper subset of candidate key AC).

The relation is not in 3rd normal form because in BC->D (neither BC is a super key nor D is a prime attribute) and in B->E (neither B is a super key nor E is a prime attribute) but to satisfy 3rd normal for, either LHS of an FD should be super key or

RHS should be prime attribute. So the highest normal form of relation will be 2nd Normal form.

Note –A prime attribute cannot be transitively dependent on a key in BCNF relation.

Consider these functional dependencies of some relation R,

AB ->C

C ->B

AB ->B

Suppose, it is known that the only candidate key of R is AB. A careful observation is required to conclude that the above dependency is a **Transitive Dependency** as the prime attribute B transitively depends on the key AB through C. Now, the first and the third FD are in BCNF as they both contain the candidate key (or simply KEY) on their left sides. The second dependency, however, is not in BCNF but is definitely in 3NF due to the presence of the prime attribute on the right side. So, the highest normal form of R is 3NF as all three FD's satisfy the necessary conditions to be in 3NF.

Example-2:

For example consider relation R(A, B, C)

A -> BC,

B -> A

A and B both are super keys so above relation is in BCNF.

Note –

BCNF decomposition may always not possible with dependency preserving,

however, it always satisfies lossless join condition. For example, relation R (V, W, X, Y, Z), with functional dependencies:

V, W -> X

Y, Z -> X

W -> Y

It would not satisfy dependency preserving BCNF decomposition.

Note -: Redundancies are sometimes still present in a BCNF relation as it is not always possible to eliminate them completely.

Database Management System | Dependency Preserving Decomposition

Dependency Preservation

A Decomposition D = { R1, R2, R3….Rn } of R is dependency preserving wrt a set F of Functional dependency if

(F1 ∪ F2 ∪ … ∪ Fm)+ = F+.

Consider a relation R

R ---> F{...with some functional dependency(FD)....}

R is decomposed or divided into R1 with FD { f1 } and R2 with { f2 }, then there can be three cases:

f1 U f2 = F -----> Decomposition is dependency preserving.

f1 U f2 is a subset of F -----> Not Dependency preserving.

f1 U f2 is a super set of F -----> This case is not possible.

Problem: Let a relation R (A, B, C, D) and functional dependency {AB –> C, C –> D, D –> A}. Relation R is decomposed into R1(A, B, C) and R2(C, D). Check whether decomposition is dependency preserving or not.
Solution:

R1(A, B, C) and R2(C, D)

Let us find closure of F1 and F2

To find closure of F1, consider all combination of

ABC. i.e., find closure of A, B, C, AB, BC and AC

Note ABC is not considered as it is always ABC

closure(A) = { A } // Trivial

closure(B) = { B } // Trivial

closure(C) = {C, A, D} but D can't be in closure as D is not present R1.

= {C, A}

C--> A   // Removing C from right side as it is trivial attribute

closure(AB) = {A, B, C, D}

= {A, B, C}

AB --> C // Removing AB from right side as these are trivial attributes

closure(BC) = {B, C, D, A}

= {A, B, C}

BC --> A // Removing BC from right side as these are trivial attributes

closure(AC) = {A, C, D}

AC --> D // Removing AC from right side as these are trivial attributes

F1 {C--> A, AB --> C, BC --> A}.

Similarly F2 { C--> D }

In the original Relation Dependency { AB --> C , C --> D , D --> A}.

AB --> C is present in F1.

C --> D is present in F2.

D --> A is not preserved.

F1 U F2 is a subset of F. So **given decomposition is not dependency**

Third Normal Form (3NF)

Although Second Normal Form (2NF) relations have less redundancy than those in 1NF, they may still suffer from update anomalies. If we update only one tuple and not the other, the database would be in an inconsistent state. This update anomaly is caused by a transitive dependency. We need to remove such dependencies by progressing to Third Normal Form (3NF).

Third Normal Form (3NF):

A relation is in third normal form, if there is no transitive dependency for non-prime attributes as well as it is in second normal form.

A relation is in 3NF if at least one of the following condition holds in every non-trivial function dependency X –> Y:

1. X is a super key.
2. Y is a prime attribute (each element of Y is part of some candidate key).

In other words,

A relation that is in First and Second Normal Form and in which no non-primary-key attribute is transitively dependent on the primary key, then it is in Third Normal Form (3NF).

Note – If A->B and B->C are two FDs then A->C is called transitive dependency.

The normalization of 2NF relations to 3NF involves the removal of transitive dependencies. If a transitive dependency exists, we remove the transitively dependent attribute(s) from the relation by placing the attribute(s) in a new relation along with a copy of the determinant.

Consider the examples given below.

Example-1:

In relation STUDENT given in Table 4,

STUD-NO	STUD_NAME	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	**RAM**	**HARIYANA**	**INDIA**	**20**
2	**RAM**	**PUNCHAB**	**INDIA**	**19**
3	**SURESH**	**PUNCHAB**	**INDIA**	**21**

FD set:
{STUD_NO -> STUD_NAME, STUD_NO -> STUD_STATE, STUD_STATE -> STUD_COUNTRY, STUD_NO -> STUD_AGE}

Candidate Key:
{STUD_NO}

For this relation in table 4, STUD_NO -> STUD_STATE and STUD_STATE -> STUD_COUNTRY are true. So STUD_COUNTRY is transitively dependent on STUD_NO. It violates the third normal form. To convert it in third normal form, we will decompose the relation STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_COUNTRY_STUD_AGE) as:

STUDENT (STUD_NO, STUD_NAME, STUD_PHONE, STUD_STATE, STUD_AGE)

STATE_COUNTRY (STATE, COUNTRY)

Example-2:

Consider relation R(A, B, C, D, E)

A -> BC,

CD -> E,

B -> D,

E -> A

All possible candidate keys in above relation are {A, E, CD, BC} All attribute are on right sides of all functional dependencies are prime.

Functional dependency in DBMS

What is Functional Dependency

Functional dependency in DBMS, as the name suggests is a relationship between attributes of a table dependent on each other. Introduced by E. F. Codd, it helps in preventing data redundancy and gets to know about bad designs.

To understand the concept thoroughly, let us consider P is a relation with attributes A and B. Functional Dependency is represented by -> (arrow sign)

Then the following will represent the functional dependency between attributes with an arrow sign −

A -> B

Above suggests the following:

Functional Dependency

A -> B

B - functionally dependent on **A**
A - determinant set
B - dependent attribute

Example

The following is an example that would make it easier to understand functional dependency −

We have a **<Department>** table with two attributes − **DeptId** and **DeptName**.

DeptId = Department ID
DeptName = Department Name

The **DeptId** is our primary key. Here, **DeptId** uniquely identifies the **DeptName** attribute. This is because if you want to know the department name, then at first you need to have the **DeptId**.

DeptId	**DeptName**
001	Finance

002	Marketing
003	HR

Therefore, the above functional dependency between **DeptId** and **DeptName** can be determined as **DeptId** is functionally dependent on **DeptName** −

DeptId -> DeptName

Types of Functional Dependency

Functional Dependency has three forms −

- Trivial Functional Dependency
- Non-Trivial Functional Dependency
- Completely Non-Trivial Functional Dependency

Let us begin with Trivial Functional Dependency −

Trivial Functional Dependency

It occurs when B is a subset of A in −

A ->B

Example

We are considering the same **<Department>** table with two attributes to understand the concept of trivial dependency.

The following is a trivial functional dependency since **DeptId** is a subset of **DeptId** and **DeptName**

{ DeptId, DeptName } -> Dept Id

Non –Trivial Functional Dependency

It occurs when B is not a subset of A in −

A ->B

Example

DeptId -> DeptName

The above is a non-trivial functional dependency since DeptName is a not a subset of DeptId.

Completely Non - Trivial Functional Dependency

It occurs when A intersection B is null in −

A ->B

Armstrong's Axioms Property of Functional Dependency

Armstrong’s Axioms property was developed by William Armstrong in 1974 to reason about functional dependencies.

The property suggests rules that hold true if the following are satisfied:

- **Transitivity**
  If A->B and B->C, then A->C i.e. a transitive relation.
- **Reflexivity**
  A-> B, if B is a subset of A.
- **Augmentation**
  The last rule suggests: AC->BC, if A->B

Lossless Decomposition in DBMS

Decomposition of a relation R into R1 and R2 is a lossless-join decomposition if at least one of the following functional dependencies are in F+ (Closure of functional dependencies)

R1 ∩ R2 → R1

OR

R1 ∩ R2 → R2

Relational Decomposition

- When a relation in the relational model is not in appropriate normal form then the deco
  relation is required.
- In a database, it breaks the table into multiple tables.

- If the relation has no proper decomposition, then it may lead to problems like loss of in
- Decomposition is used to eliminate some of the problems of bad design like anomalies, inconsistencies, and redundancy.

Types of Decomposition

Lossless Decomposition

- If the information is not lost from the relation that is decomposed, then the decompositi lossless.
- The lossless decomposition guarantees that the join of relations will result in the same r was decomposed.
- The relation is said to be lossless decomposition if natural joins of all the decompositio original relation.

Example:

EMPLOYEE_DEPARTMENT table:

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEP
22	Denim	28	Mumbai	827	Sa
33	Alina	25	Delhi	438	Ma
46	Stephan	30	Bangalore	869	Fi
52	Katherine	36	Mumbai	575	Pr
60	Jack	40	Noida	678	Te

The above relation is decomposed into two relations EMPLOYEE and DEPARTMENT

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY
22	Denim	28	Mumbai
33	Alina	25	Delhi
46	Stephan	30	Bangalore
52	Katherine	36	Mumbai
60	Jack	40	Noida

DEPARTMENT table

DEPT_ID	EMP_ID	DEPT_NAME
827	22	Sales
438	33	Marketing
869	46	Finance
575	52	Production
678	60	Testing

Now, when these two relations are joined on the common column "EMP_ID", then the resulta
look like:

Employee ⋈ Department

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEP
22	Denim	28	Mumbai	827	Sa
33	Alina	25	Delhi	438	Ma
46	Stephan	30	Bangalore	869	Fi
52	Katherine	36	Mumbai	575	Pr
60	Jack	40	Noida	678	Te

Hence, the decomposition is Lossless join decomposition.

Dependency Preserving

- It is an important constraint of the database.
- In the dependency preservation, at least one decomposed table must satisfy every depen
- If a relation R is decomposed into relation R1 and R2, then the dependencies of R eithe part of R1 or R2 or must be derivable from the combination of functional dependencies
- For example, suppose there is a relation R (A, B, C, D) with functional dependency set relational R is decomposed into R1(ABC) and R2(AD) which is dependency preserving A->BC is a part of relation R1(ABC).

Decomposition Algorithms

In the previous section, we discussed decomposition and its types with the help of small examples. In the actual world, a database schema is too wide to handle. Thus, it requires algorithms that may generate appropriate databases.

Here, we will get to know the decomposition algorithms using functional dependencies for two different normal forms, which are:

- **Decomposition to BCNF**
- **Decomposition to 3NF**

Decomposition using functional dependencies aims at dependency preservation and lossless decomposition.

Let's discuss this in detail.

Decomposition to BCNF

Before applying the BCNF decomposition algorithm to the given relation, it is necessary to test if the relation is in Boyce-Codd Normal Form. After the test, if it is found that the given relation is not in BCNF, we can decompose it further to create relations in BCNF.

There are following cases which require to be tested if the given relation schema R satisfies the BCNF rule:

Case 1: Check and test, if a nontrivial dependency **α -> β** violate the BCNF rule, evaluate and compute α^+, i.e., the attribute closure of α. Also, verify that α+

includes all the attributes of the given relation R. It means it should be the superkey of relation R.

Case 2: If the given relation R is in BCNF, it is not required to test all the dependencies in F^+. It only requires determining and checking the dependencies in the provided dependency set F for the BCNF test. It is because if no dependency in F causes a violation of BCNF, consequently, none of the F^+ dependency will cause any violation of BCNF.

BCNF Decomposition Algorithm

This algorithm is used if the given relation R is decomposed in several relations R_1, R_2,…, R_n because it was not present in the BCNF. Thus,

For every subset α of attributes in the relation R_i, we need to check that α^+ (an attribute closure of α under F) either includes all the attributes of the relation R_i or no attribute of R_i-α.

```
result={R};
done=false;
compute F+;
while (not done) do
        if (there is a schema Ri in result that is not in BCNF)
                then begin
                        let α->β be a nontrivial functional dependency that holds
                        on Ri such that α->Ri is not in F+, and α ∩ β= ø;
                        result=(result-Ri) U (Ri-β) U (α,β);
```

```
        end
        else done=true;
```

This algorithm is used for decomposing the given relation R into its several decomposers. This algorithm uses dependencies that show the violation of BCNF for performing the decomposition of the relation R. Thus, such an algorithm not only generates the decomposers of relation R in BCNF but is also a lossless decomposition. It means there occurs no data loss while decomposing the given relation R into R_1, R_2, and so on…

The BCNF decomposition algorithm takes time exponential in the size of the initial relation schema R. With this, a drawback of this algorithm is that it may unnecessarily decompose the given relation R, i.e., over-normalizing the relation. Although decomposing algorithms for BCNF and 4NF are similar, except for a difference. The fourth normal form works on **multivalued dependencies,** whereas BCNF focuses on the **functional dependencies**. The multivalued dependencies help to reduce some form of repetition of the data, which is not understandable in terms of functional dependencies.

Difference between Multivalued Dependency and Functional Dependency

The difference between both dependencies is that a functional dependency expels certain tuples from being in a relation, but a multivalued dependency does not do so. It means a multivalued dependency does not expel or rule out certain tuples. Rather it requires other tuples of certain forms to exist in relation. Due to such a difference, the multivalued dependency is also referred to as **tuple-generating dependency,** and the functional dependency is referred to as **equality-generating dependency.**

Decomposition to 3NF

The decomposition algorithm for 3NF ensures the preservation of dependencies by explicitly building a schema for each dependency in the canonical cover. It guarantees that at least one schema must hold a candidate key for the one being decomposed, which in turn ensures the decomposition generated to be a lossless decomposition.

3NF Decomposition Algorithm

let Fc be a canonical cover for F;

i=0;

for each functional dependency α->β in F_c

    i = i+1;

R = αβ;

If none of the schemas R_j, j=1,2,…I holds a candidate key for R

Then

    i = i+1;

    R_i= any candidate key for R;

/* Optionally, remove the repetitive relations*/

Repeat

    If any schema R_j is contained in another schema Rk

Then

/* Delete R_j */

$R_j = R_i$;

i = i-1;

until no more Rjs can be deleted

return (R_1, R_2, . . . ,R_i)

Here, R is the given relation, and F is the given set of functional dependency for which F_c maintains the canonical cover. R_1, R_2, . . . , R_i are the decomposed parts of the given relation R. Thus, this algorithm preserves the dependency as well as generates the lossless decomposition of relation R.

A 3NF algorithm is also known as a **3NF synthesis algorithm.** It is called so because the normal form works on a dependency set, and instead of repeatedly decomposing the initial schema, it adds one schema at a time.

Drawbacks of 3NF Decomposing Algorithm

- The result of the decomposing algorithm is not uniquely defined because a set of functional dependencies can hold more than one canonical cover.
- In some cases, the result of the algorithm depends on the order in which it considers the dependencies in F_c.
- If the given relation is already present in the third normal form, then also it may decompose a relation.

Multivalued dependency in DBMS

What is Multi-valued dependency?

When existence of one or more rows in a table implies one or more other rows in the same table, then the Multi-valued dependencies occur.

If a table has attributes P, Q and R, then Q and R are multi-valued facts of P.

It is represented by double arrow −

->->

For our example:

P->->Q **P->->R**

In the above case, Multivalued Dependency exists only if Q and R are independent attributes.

A table with multivalued dependency violates the 4NF.

Example

Let us see an example &mins;

<Student>

StudentName	**CourseDiscipline**	**Activities**
Amit	Mathematics	Singing
Amit	Mathematics	Dancing
Yuvraj	Computers	Cricket

Akash	Literature	Dancing
Akash	Literature	Cricket
Akash	Literature	Singing

In the above table, we can see Students **Amit** and **Akash** have interest in more than one activity.

This is multivalued dependency because **CourseDiscipline** of a student are independent of Activities, but are dependent on the student.

Therefore, multivalued dependency −

StudentName ->-> CourseDiscipline
StudentName ->-> Activities

The above relation violates Fourth Normal Form in Normalization.

To correct it, divide the table into two separate tables and break Multivalued Dependency −

<StudentCourse>

StudentName	**CourseDiscipline**
Amit	Mathematics
Amit	Mathematics
Yuvraj	Computers

Akash	Literature
Akash	Literature
Akash	Literature

<StudentActivities>

StudentName	**Activities**
Amit	Singing
Amit	Dancing
Yuvraj	Cricket
Akash	Dancing
Akash	Cricket
Akash	Singing

This breaks the multivalued dependency and now we have two functional dependencies −

StudentName -> CourseDiscipline
StudentName - > Activities

Fourth Normal Form (4NF)

What is 4NF?

The 4NF comes after 1NF, 2NF, 3NF, and Boyce-Codd Normal Form. It was introduced by Ronald Fagin in 1977.

To be in 4NF, a relation should be in Bouce-Codd Normal Form and may not contain more than one multi-valued attribute.

Example

Let us see an example −

<Movie>

Movie_Name	**Shooting_Location**	**Listing**
MovieOne	UK	Comedy
MovieOne	UK	Thriller
MovieTwo	Australia	Action
MovieTwo	Australia	Crime

MovieThree	India	Drama

The above is not in 4NF, since

- More than one movie can have the same listing
- Many shooting locations can have the same movie

Let us convert the above table in 4NF −

<Movie_Shooting>

Movie_Name	**Shooting_Location**
MovieOne	UK
MovieOne	UK
MovieTwo	Australia
MovieTwo	Australia
MovieThree	India

<Movie_Listing>

Movie_Name	**Listing**

MovieOne	Comedy
MovieOne	Thriller
MovieTwo	Action
MovieTwo	Crime
MovieThree	Drama

Now the violation is removed and the tables are in 4NF.

Algorithm for 4NF decomposition Similar to BCNF decomposition:

Given R = (R, D) where D contains FDs and MVDs

- Take X Y violates the 4NF condition
- Decompose R into R1=(X,D1) and R2=(X∪(R-Y),D2)

where the FDs in D1 and D2 are computed in the same way as in the BCNF decomposition, see note for the computation for MVD

Decomposition and 4NF

- If X ->->Y is a 4NF violation violation for relation relation R, we can decompose R using the same technique as for BCNF

1. XY is one of the decomposed relations.
2. All but Y – X is the other.

19 Example Drinkers Drinkers(name, addr, phones, beersLiked beersLiked) FD: name -> addr MVD's: name ->-> phones name ->-> beersLiked

• Key is – { a e, n m p o es, h n bee s ed} r Liked}.

• Which dependencies violate 4NF ? All 20 – Example, Continued

• Decompose using name -> addr:

1. Drinkers1 Drinkers1(name, addr) ƒ In 4NF, only dependency is name -> addr.

2. Drinkers2 Drinkers2(name, phones, beersLiked) beersLiked) ƒ Not in 4NF. MVD's name ->-> phones and name ->-> beersLiked beersLiked apply.

ƒ Key

ƒ No FDs so all three attributes form the key 21 ƒ

No FDs, so all three attributes form the key.

Example: Decompose Drinkers2

• Either MVD name ->-> phones or name ->-> beersLiked beersLiked tells us to decompose decompose to: – Drinkers3(name, phones) – Drinkers4(name, beersLiked) 22 Relationships Among Normal Form.