

Database Management Systems 2

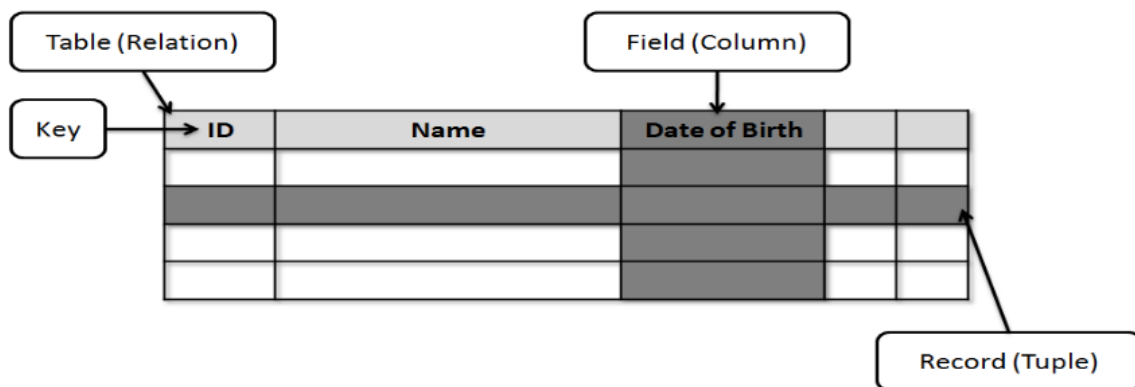
**2rd Class, CS Dept.
2nd Semester**

Introduction

A database is a structured collection of data that is organized and stored in a way that allows easy access and retrieval of information. Databases are used to store and manage data for a wide variety of applications, from small-scale personal record keeping to large-scale enterprise systems. A database is a fundamental component of modern computing.

Most databases are managed by a database management system (DBMS), which is a software system that provides tools for creating, managing, and accessing the database. Some popular DBMSs include Oracle, MySQL, and Microsoft SQL Server.

Data in a database is typically organized into tables, which consist of rows and columns. Each row represents a single record or instance of data, while each column represents a specific attribute or characteristic of that data.

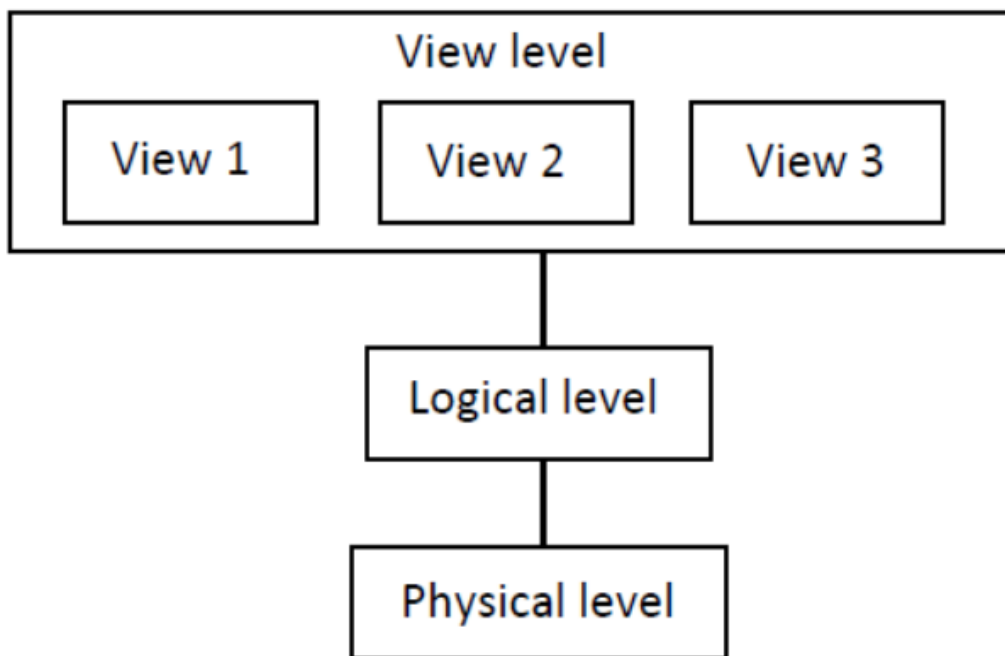


The elements of database tables

Database Instances and Database Schemas

A **schema** is a description of a particular collection of data, using the a given data model, simply means the structure or the form of the database without any data in it. Schema is of three types:



1. **Physical schema:** how the data stored in blocks of storage is described at this level.
2. **Logical schema:** programmers and database administrators work at this level, at this level data can be described as certain types of data records gets stored in data structures, however the internal details such as implementation of data structure is hidden at this level (available at physical level).
3. **View schema:** this generally describes end user interaction with database systems.



Level Database Architecture

The actual content of the database, the data, changes often over the years. A database state at a specific time defined through the currently existing content and relationship and their attributes is called a **database instance**.

The following illustration shows that a database scheme could be looked at like a template or building plan for one or several database instances.

	Real World	Database																				
Scheme	<div></div> <div>Plan for a Standard-House</div>	<table border="1"><thead><tr><th>P-ID</th><th>Name</th><th>Prename</th></tr></thead><tbody><tr><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td></tr></tbody></table> <div>Template for a Table</div>	P-ID	Name	Prename																	
P-ID	Name	Prename																				
Instances	<div></div> <div>Built Standard-Houses</div>	<table border="1"><thead><tr><th>P-ID</th><th>Name</th><th>Prename</th><th></th><th>Prename</th></tr></thead><tbody><tr><td>102356</td><td>Smith</td><td>John</td><td></td><td></td></tr><tr><td>102357</td><td>Potter</td><td>Harry</td><td></td><td>William</td></tr><tr><td></td><td></td><td></td><td>523646</td><td>Wood Lucinda</td></tr></tbody></table> <div>Data-filled Tables</div>	P-ID	Name	Prename		Prename	102356	Smith	John			102357	Potter	Harry		William				523646	Wood Lucinda
P-ID	Name	Prename		Prename																		
102356	Smith	John																				
102357	Potter	Harry		William																		
			523646	Wood Lucinda																		

Example of a database schema and instances

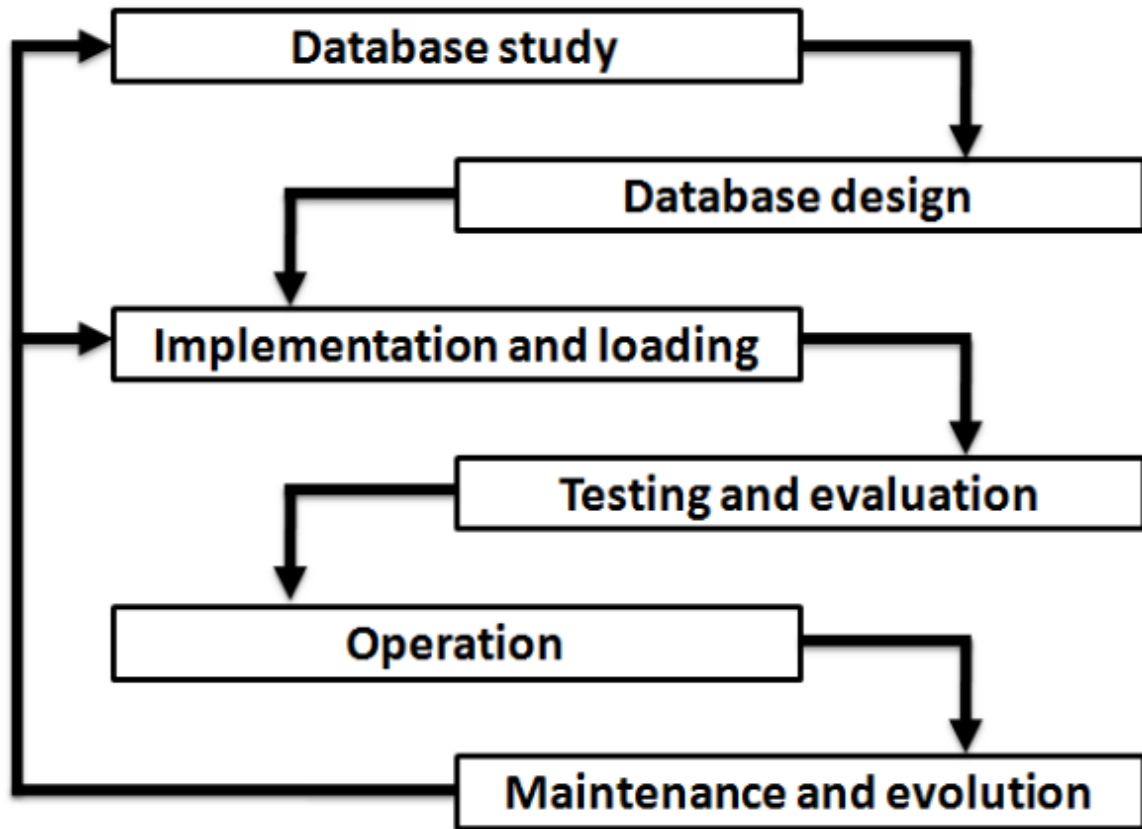
Database analysis life cycle

The purpose of any database software is to effectively manage and handle large sets of data and, for this reason, its development and implementation is carefully observed and documented so as to avoid any malfunctioning during its operational period and produce effective database software, the logical steps followed are that of the database analysis life cycle:

1. **Database study** - here the designer creates a written specification in words for the database system to be built. This involves:

- Analyzing the company situation.
 - Define problems and constraints.
 - Define objectives.
 - Define scope and boundaries.
2. **Database Design** - conceptual, logical, and physical design steps in taking specifications to physical implementable designs. This is looked at more closely in a moment.
 3. **Implementation and loading** - it is quite possible that the database is to run on a machine which as yet does not have a database management system running on it at the moment. If this is the case one must be installed on that machine. Finally, not all databases start completely empty, and thus must be loaded with the initial data set (such as the current inventory, current staff names, current customer details, etc).
 4. **Testing and evaluation** - the database, once implemented, must be tested against the specification supplied by the client. It is also useful to test the database with the client using mock data, as clients do not always have a full understanding of what they thing they have specified and how it differs from what they have actually asked for. In addition, this step in the life cycle offers the chance to the designer to fine-tune the system for best performance. Finally, it is a good idea to evaluate the database in-situ, along with any linked applications.
 5. **Operation** - this step is where the system is actually in real usage by the company.
 6. **Maintenance and evolution** - designers rarely get everything perfect first time, and it may be the case that the company requests changes to

fix problems with the system or to recommend enhancements or new requirements.



Database analysis life cycle

Overview of programming languages used for databases

Databases are important components of many software applications and systems, and there are several programming languages that are commonly used for interacting with them.

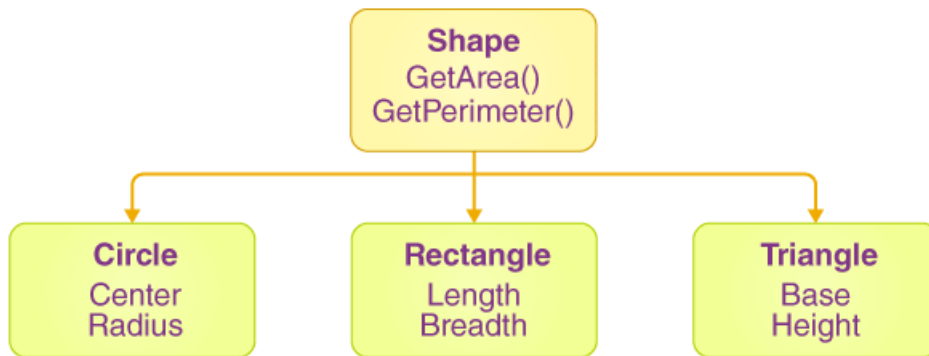
Object-oriented model

The object-oriented model is a way of organizing and structuring data based on the principles of object-oriented programming. In this model, data is represented as objects. Here are some concepts of the object-oriented model:

1. **Objects** are instances of classes that represent real-world entities or concepts. Each object has a set of properties (attributes) that define its state and behavior.
2. **Classes** are templates that define the properties and behavior of objects. They encapsulate data and behavior, and provide a way to create objects that share common properties and methods.
3. **Encapsulation** is the principle of hiding the implementation details of a class, and show only a public interface for interacting with its objects. It helps to reduce complexity and improve modularity and maintainability.
4. **Inheritance** is the principle of defining a new class based on an existing class, inheriting its attributes and behavior.
5. **Polymorphism** is the principle of using a single interface to represent multiple types of objects. It allows objects of different classes to be

treated as if they are of the same type, providing flexibility and extensibility.

6. **Methods** are functions or procedures that define the behavior of objects. They can be used to perform actions on the object's data or to modify its state.
7. **Attributes** are variables or data members that define the state of objects. They can be used to store information about the object's properties.



Object-oriented model

Overview of data interchange formats

Data interchange formats are used to exchange data between different applications or systems. Here are some of the most commonly used data interchange formats:

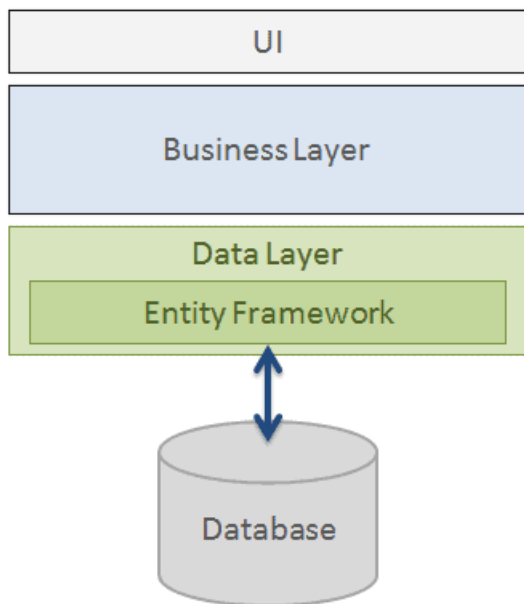
1. **JSON (JavaScript Object Notation):** JSON is a lightweight, text-based data interchange format that is widely used for web APIs and data exchange between applications. It is easy to read and write, and

- can be easily parsed by most programming languages. JSON supports simple data types such as strings, numbers, and boolean values, as well as complex data types such as objects and arrays.
2. **XML (Extensible Markup Language):** XML is a widely used data interchange format that is commonly used for document exchange and data exchange between applications. It is a markup language that uses tags to define the structure of the data. XML can be used to represent complex data structures and has support for namespaces and validation using XML schemas.
 3. **CSV (Comma Separated Values):** CSV is a simple data interchange format that uses commas to separate fields and new lines to separate records. It is commonly used for exchanging tabular data between applications and can be easily imported and exported by spreadsheet applications.

Data Access Layers and Frameworks

A data access layer (DAL) is a software component that provides a consistent interface for accessing data from various sources, such as databases, web services, and file systems, allowing developers to focus on the application logic rather than the details of the data storage technology. A DAL can help decouple the application logic from the underlying data storage technology, which can improve maintainability, scalability, and performance. Here are some popular data access layers and frameworks:

1. **Entity Framework (EF):** EF is a data access layer that is part of the Microsoft .NET framework. It provides an object-relational mapping (ORM) framework that allows developers to work with relational databases using .NET objects. EF provides automatic generation of SQL queries.
2. **Hibernate:** Hibernate is a popular data access layer for Java applications that provides an ORM framework for working with relational databases. Hibernate supports mapping of Java objects to database tables and provides caching and transaction management features.



Entity Framework

Overview of data visualization and analysis

Data visualization and analysis are critical components of data science and business intelligence.

Data Visualization: is the graphical representation of data and information. It allows users to visualize complex data sets in a simple and easy-to-understand format.



Data Visualization

Data Analysis: involves examining data to uncover patterns, relationships, and insights. It is the process of systematically analyzing data using statistical and mathematical techniques to extract meaningful insights.

Data visualization and analysis go hand in hand, as visualizing data can help identify patterns and insights that may not be immediately apparent through raw data analysis.

There are many visualization languages and tools available to help users create visual representations of data, such as: **MATLAB**, **Excel**, and **Python**.

Database connectivity

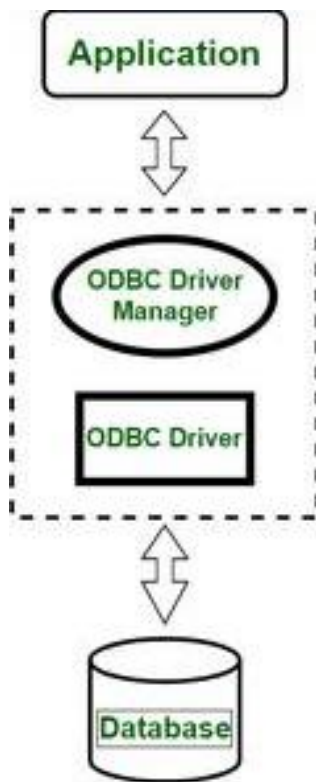
Database connectivity refers to the ability of software applications to access and manipulate data stored in a database.

The process of establishing database connectivity involves several steps, which may vary depending on the specific DBMS and programming language being used, include:

1. **Installing the appropriate database driver:** The application must have the correct driver installed for the DBMS it will be connecting to.
2. **Establishing a connection:** The application must create a connection to the database using the driver. This typically involves specifying the name of the database, the server address, and the login credentials.
3. **Executing SQL commands:** Once the connection is established, the application can send SQL (Structured Query Language) commands to the database to retrieve or manipulate data.
4. **Handling errors:** If there is an error in the connection or in executing a command, the application must be able to handle the error gracefully and provide feedback to the user.

ODBC Connectivity

ODBC (Open Database Connectivity) is a standard interface for connecting to databases. ODBC connectivity enables software applications to interact with data stored in a wide range of databases using a common interface. It provides a layer of abstraction between the application and the underlying database, allowing the application to access data without needing to know the specifics of the database management system.



ODBC Connectivity

ODBC connectivity has several advantages, including:

1. **Cross-platform compatibility:** Because ODBC is a standard interface, applications can connect to a wide range of databases across different platforms without needing to be re-written for each specific database.

2. **Reduced development time:** Using ODBC can simplify development by providing a standard interface for accessing data, rather than requiring developers to write code specific to each database.
3. **Improved performance:** ODBC drivers can often be optimized for specific databases, which can improve performance compared to using a generic database driver.

Overall, ODBC connectivity provides a flexible and powerful way to connect software applications to databases, making it a popular choice for a wide range of applications.

JDBC Connectivity

JDBC (Java Database Connectivity) is a standard interface for connecting Java applications to databases. JDBC connectivity enables Java applications to access and manipulate data stored in databases using SQL statements.

JDBC connectivity provides a robust and reliable way to connect Java applications to databases.

ADO.NET Connectivity

ADO.NET (ActiveX Data Objects for .NET) is a data access technology used to interact with relational and non-relational databases in the .NET Framework. ADO.NET allows developers to access and manipulate data in databases using a set of classes and interfaces, which are part of the .NET Framework.

ADO.NET provides a flexible and powerful way to connect .NET applications to databases.

Database Connectivity in Web Applications

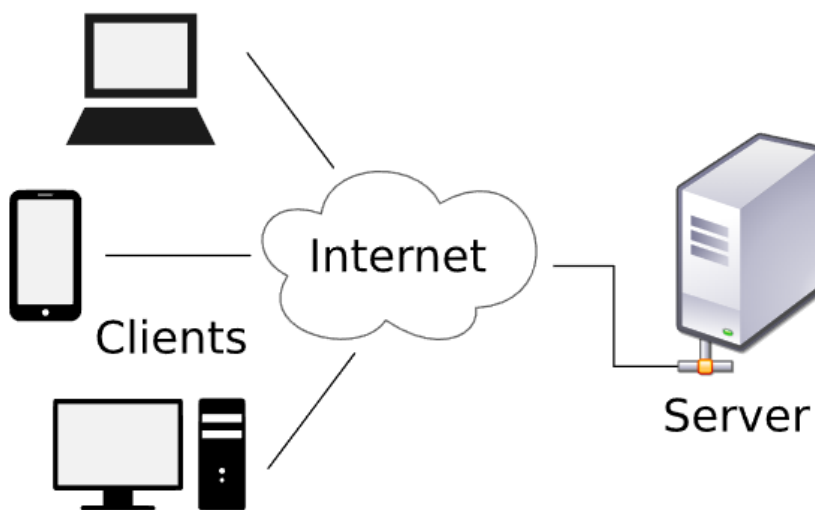
Database connectivity is a critical aspect of web application development as it enables web applications to interact with databases and store or retrieve data dynamically. In web applications, database connectivity typically involves a client-server architecture, where the web application acts as the client and the database server as the server.

Web applications can use a variety of technologies for database connectivity, including ODBC, JDBC, and ADO.NET, as well as Object-Relational Mapping (ORM) frameworks like Entity Framework or Hibernate.

In a web application, the database connectivity process involves several steps:

1. **Establishing a connection:** The web application establishes a connection to the database server using the appropriate connectivity technology, such as JDBC or ADO.NET. The connection is typically established using a connection string that contains the necessary connection information, such as the database server name, username, and password.

2. **Sending SQL queries:** Once the connection is established, the web application can send SQL queries to the database server to retrieve or store data. The queries can be simple or complex, and they can involve one or more database tables.
3. **Processing the results:** Once the database server processes the SQL queries, it returns the results to the web application. The web application can then process the results and display them to the user or store them in the application's memory for later use.
4. **Closing the connection:** When the web application is finished accessing the database, it should close the connection to the database server to free up resources and prevent security vulnerabilities.



Client-server architecture

Database connectivity in web applications is critical for many web applications, including e-commerce websites, social media platforms, and content management systems. It allows developers to create dynamic and interactive web applications that can store and retrieve data in real-time, making them more engaging and useful to users.

Functional dependencies

Functional dependencies (FDs) are a fundamental concept in relational database theory. They represent a relationship between the attributes (or columns) of a table, where the value of one attribute determines the value of another attribute. Specifically, a functional dependency between two or more attributes in a table means that the values of the first set of attributes uniquely determine the values of the second set of attributes.

For example, consider a table called "Employee" with columns "EmployeeID", "Name", "Salary", and "Department". Suppose we observe that for each EmployeeID, there is only one Department associated with it. We can then say that there is a functional dependency between EmployeeID and Department. We write this as $\text{EmployeeID} \rightarrow \text{Department}$, where " \rightarrow " means "determines".

EmployeeID	Name	Salary	Department
1	John Smith	50000	Sales
2	Jane Doe	75000	Marketing
3	Bob Johnson	60000	Sales
4	Sarah Lee	65000	HR
5	David Kim	80000	Marketing

In a database schema, functional dependencies help us to identify the **key attributes** (i.e., the attributes that uniquely identify each row in the table)

and the **non-key attributes** (i.e., the attributes that are dependent on the key attributes). They also help us to normalize a database schema by removing redundant data and improving data consistency.

By understanding functional dependencies, database designers can ensure that their databases are well-structured, efficient, and easy to maintain.

Keys and composite keys

In a relational database, a **key** is a set of one or more attributes that uniquely identifies a row (or record) in a table. A key can be a single attribute or a combination of multiple attributes, also known as a composite key. Keys are important for maintaining the integrity and consistency of the data in the database.

A primary key is a special type of key that uniquely identifies each row in a table. It must be unique and not null for each record, and cannot be duplicated in any other record in the same table. The primary key is often used as a foreign key in other tables to establish relationships between them.

A foreign key is a column or combination of columns in a table that refers to the primary key of another table. It is used to establish a link between two tables and maintain referential integrity. The foreign key in one table refers to the primary key in another table, thereby creating a relationship between the two tables.

Composite keys are used when a single attribute is not sufficient to uniquely identify a row in a table. In this case, a combination of attributes is used to create a composite key. For example, in a table of student grades, a

composite key could be created using the student ID and the course ID, as each student can take multiple courses and each course can be taken by multiple students.

Functional dependencies are closely related to keys, as they define the relationships between attributes in a table.

Formal notation

Functional dependencies (FDs) can be formally represented using arrow notation, where the left-hand side (LHS) represents the set of attributes that determines the right-hand side (RHS). For example, the functional dependency $A \rightarrow B$. Multiple functional dependencies can be represented using a comma-separated list, such as $A \rightarrow B, C \rightarrow D$. There are three rules that allow us to derive new FDs from given FDs:

Reflexivity: If X is a set of attributes, then $X \rightarrow X$ is true.

Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any set of attributes Z .

Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.

Using these axioms, we can derive additional functional dependencies from a set of given functional dependencies. For example, suppose we have the following functional dependencies:

$A \rightarrow B$

$B \rightarrow C$

We can use the augmentation axiom to derive a new functional dependency:

$A \rightarrow BC$

We can also use the transitivity axiom to derive another new functional dependency:

$A \rightarrow C$

This process of deriving new functional dependencies from given ones can be continued until no newer functional dependencies can be inferred.

These notations are important for designing and optimizing database schemas, as they provide a formal way to reason about the relationships between attributes in a table.

Example on Transitivity, let's consider a table called "Employees" with columns "EmployeeID", "DepartmentID", and "ManagerID". Here, we can observe that for each EmployeeID, there is only one DepartmentID associated with it, and for each DepartmentID, there is only one ManagerID associated with it. Therefore, we can say that there is a functional dependency between EmployeeID and DepartmentID (EmployeeID \rightarrow DepartmentID), and between DepartmentID and ManagerID (DepartmentID \rightarrow ManagerID). Using transitivity, we can infer that there is also a functional dependency between EmployeeID and ManagerID (EmployeeID \rightarrow ManagerID).

EmployeeID	DepartmentID	ManagerID
1	101	3
2	102	4
3	101	3

4	102	4
5	103	6

Partial dependencies

Partial dependencies are a property of functional dependencies in a database that occurs when a non-key attribute is functionally dependent on only a part of the primary key. In other words, a partial dependency exists when a non-key attribute is determined by only a subset of the primary key attributes, rather than the entire primary key.

For example, consider a table called "Orders" with columns "OrderID", "CustomerID", "CustomerName", and "OrderDate". Here, the primary key is composed of the "OrderID" and "CustomerID" columns. Suppose we observe that the attribute "CustomerName" is functionally dependent only on "CustomerID" and not on "OrderID". This is an example of a partial dependency because "CustomerName" is dependent on only a part of the primary key.

OrderID	CustomerID	CustomerName	OrderDate
1	1001	John Smith	2022-01-15
2	1002	Jane Doe	2022-02-20
3	1003	Bob Johnson	2022-03-05

4	1001	John Smith	2022-04-10
5	1004	Sarah Lee	2022-05-15

Partial dependencies can cause data redundancy and inconsistencies in the database, as they can result in multiple rows with the same values for some attributes but different values for others.

Normalization

Normalization is the process of organizing data in a database to minimize redundancy and dependency. It involves breaking down a large table into smaller tables and defining relationships between them to reduce data duplication and inconsistencies. The goal of normalization is to create a database schema that is efficient, flexible, and easy to maintain.

Normalization is typically achieved through a series of steps, including first normal form (1NF), second normal form (2NF), third normal form (3NF), and so on.

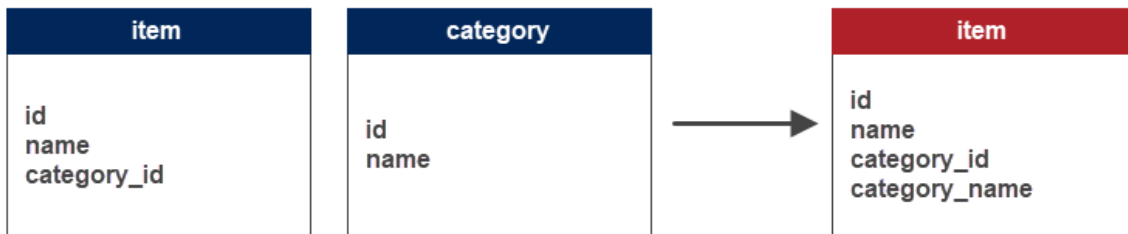
Denormalization

Denormalization is the process of intentionally introducing redundancy into a database in order to improve query performance or simplify database maintenance. In other words, denormalization is the opposite of normalization, which aims to reduce redundancy and improve data consistency.

The goal of denormalization is to improve database performance by reducing the number of tables and joins required to answer complex queries.

By duplicating data across multiple tables, queries can be simplified and faster to execute. However, this comes at the cost of increased storage requirements and the potential for data inconsistencies.

Denormalization is often used in large, complex databases with heavy read workloads, where query performance is critical. For example, in an e-commerce system, denormalization might be used to store a user's order history alongside their user profile, rather than querying multiple tables to retrieve the same information. This can reduce query complexity and improve response times.



Database Normalization

Database normalization is a process of organizing data in a database in a way that reduces redundancy and dependency, while ensuring data consistency and integrity. It involves breaking down a larger table into smaller tables and defining relationships between them to eliminate data duplication and inconsistencies.

There are several levels of normalization, including first normal form (1NF), second normal form (2NF), third normal form (3NF), and higher normal forms such as Boyce-Codd normal form (BCNF) and fourth normal form (4NF). Each level of normalization builds upon the previous one and further reduces data redundancy and dependency.

Problems without Normalization

Without normalization, a database can suffer from several problems, including:

1. **Data Redundancy:** the same data may be stored in multiple places within the database.
2. **Update Anomalies:** any updates to that data must be made in each location.
3. **Data Inconsistency:** updates are not made to all locations.
4. **Difficulty in Querying Data:** data may be stored in a way that makes it difficult to retrieve specific information or to query the database efficiently.

First normal form (1NF)

First normal form is an essential property of a relation in a relational database.

First normal form (1NF) Rules:

- A table should have a primary key that uniquely identifies each record. Identify each set of related data with a primary key.
- Each column in the table should contain only values of the same data type.
- There are no repeating groups in the individual tables.
- Each table cell should contain a single value.

Examples

Below is a table that stores the names and telephone numbers of customers. One requirement though is to retain multiple telephone numbers for some customers. The simplest way of satisfying this requirement is to allow the "Telephone Number" column in any given row to contain more than one value:

Customer

Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025, 192-122-1111

456	San	Zhang	(555) 403-1659 Ext. 53; 182-929-2929
789	John	Doe	555-808-9633

An apparent solution is to introduce more columns:

Customer

Customer ID	First Name	Surname	Telephone Number1	Telephone Number2
123	Pooja	Singh	555-861-2025	192-122-1111
456	San	Zhang	(555) 403-1659 Ext. 53	182-929-2929
789	John	Doe	555-808-9633	

Technically, this table does not violate the requirement for values to be atomic. However, informally, the two telephone number columns still form a "repeating group": they repeat what is conceptually the same attribute, namely a telephone number.

To bring the model into the first normal form, we split the strings we used to hold our telephone number information into "atomic" (i.e. indivisible)

entities: single phone numbers. And we ensure no row contains more than one phone number.

Customer

Customer ID	First Name	Surname	Telephone Number
123	Pooja	Singh	555-861-2025
123	Pooja	Singh	192-122-1111
456	San	Zhang	182-929-2929
456	San	Zhang	(555) 403-1659 Ext. 53
789	John	Doe	555-808-9633

Note that the "ID" is no longer unique in this solution with duplicated customers. An alternative design uses two tables:

Customer Name

<u>Customer ID</u>	First Name	Surname
123	Pooja	Singh
456	San	Zhang
789	John	Doe

Customer Telephone Number

Customer ID	<u>Telephone Number</u>
123	555-861-2025
123	192-122-1111
456	(555) 403-1659 Ext. 53
456	182-929-2929
789	555-808-9633

For example, suppose we have a table for **tracking sales** that contains the following columns:

Order Number	Customer Name	Item	Item Quantity
001	John Smith	Shirt, Pants	1, 2

This table does not satisfy the first normal form, because the Item and Item Quantity columns contain repeating groups of values. To normalize this table to the first normal form, we need to split it into two tables: one for Orders and one for Order Items.

Orders Table:

Order Number	Customer Name
001	John Smith

Order Items Table:

Order Number	Item	Item Quantity
001	Shirt	1
001	Pants	2

Now each table contains only atomic values, and there are no repeating groups or arrays of values. This satisfies the requirements of the first normal form.

Not in 1NF (Unnormalized table into first normal form)

Example 1:

Invoice#	Customer Information											
	Cust#	Name	Addr	Quant1	Part1	Amt1	Quant2	Part2	Amt2	Quant3	Part3	Amt3
1001	43	Jones	121 1st	200	Screw	2.00	300	Nut	2.25	100	Washr	0.75
1002	55	Smith	222 2nd	1	Motor	52.00	5	Brace	44.44			
1003	43	Jones	121 1st	10	Saw	121.00						

Example 2:

TABLE_PRODUCT

Product ID	Color	Price
1	red, green	15.99
2	yellow	23.99
3	green	17.50
4	yellow, blue	9.99
5	red	29.99

Second normal form (2NF)

The second normal form (2NF) is a database normalization rule that helps to ensure data integrity by removing partial dependencies on a composite primary key.

A table is said to be in 2NF if it meets the following two requirements:

- It is in First Normal Form (1NF).
- It includes no partial dependencies. All non-key attributes (i.e., attributes that are not part of the primary key) are fully functionally dependent on the entire primary key, and not on a part of it.

Example:

Consider the following table that contains information about students and their courses:

<u>Student ID</u>	<u>Course Code</u>	Course Name	Instructor	Instructor Email	Semester
1001	MATH101	Calculus	John Smith	jsmith@example.com	Fall 2022
1001	ENGL101	English	Jane Doe	jdoe@example.com	Fall 2022
1002	MATH101	Calculus	John Smith	jsmith@example.com	Fall 2022
1002	HIST101	History	Sarah Lee	slee@example.com	Fall 2022

In this table, the primary key is a composite key consisting of both the Student ID and the Course Code columns. However, the Instructor and Instructor Email columns are dependent only on the Course Code and not on the entire primary key.

To bring this table into 2NF, we need to separate out the Instructor and Instructor Email columns into a separate table. Here's what the two tables would look like:

Table 1: Students_Courses

<u>Student ID</u>	<u>Course Code</u>	Semester
1001	MATH101	Fall 2022
1001	ENGL101	Fall 2022
1002	MATH101	Fall 2022
1002	HIST101	Fall 2022

Table 2: Courses_Instructors

<u>Course Code</u>	Course Name	Instructor	Instructor Email
MATH101	Calculus	John Smith	jsmith@example.com
ENGL101	English	Jane Doe	jdoe@example.com
HIST101	History	Sarah Lee	slee@example.com

(It is still possible for a table in 2NF to exhibit transitive dependency; that is, one or more attributes may be functionally dependent on nonkey attributes.)

Another Example:

<u>Student Id</u>	<u>Subject Id</u>	Marks	Teacher
10	1	70	Java Teacher
10	2	75	C++ Teacher
11	1	80	Java Teacher

The simplest solution is to remove columns teacher from Score table and add it to the Subject table. Hence, the Subject table will become:

<u>Subject Id</u>	Subject_Name	Teacher
1	Java	Java Teacher
2	C++	C++ Teacher
3	Php	Php Teacher

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

<u>Student Id</u>	<u>Subject Id</u>	Marks
10	1	70
10	2	75
11	1	80

Third normal form (3NF)

A database is in third normal form if it satisfies the following conditions:

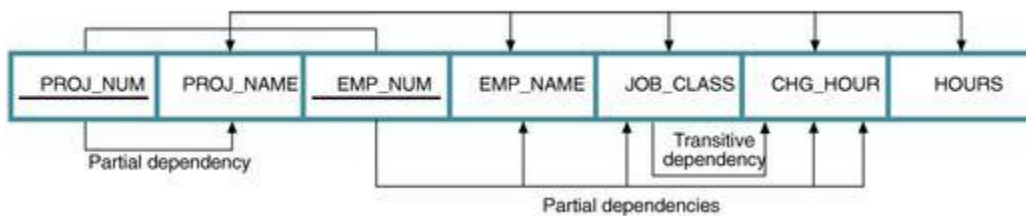
- It is in Second Normal Form
- There is no transitive functional dependency

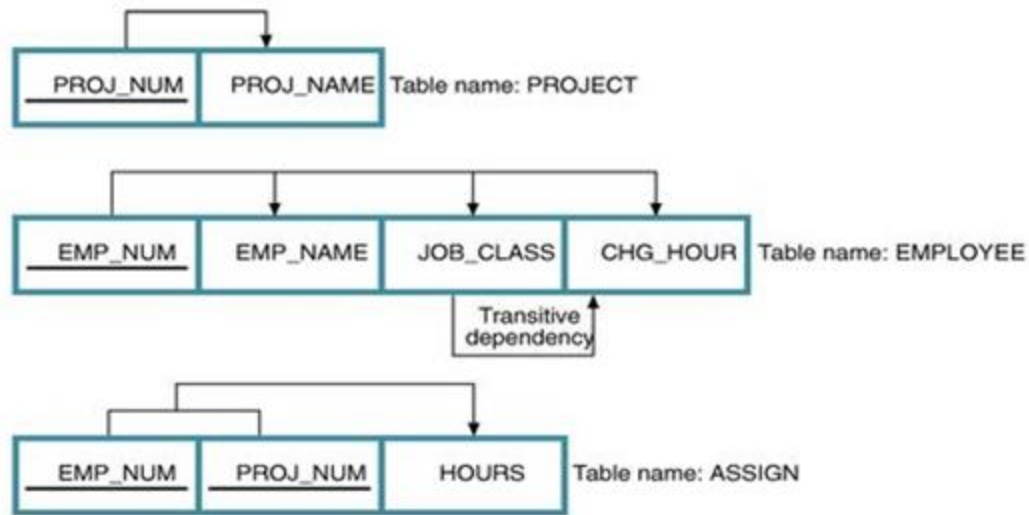
Transitive dependency

A transitive dependency is an indirect functional dependency, one in which $X \rightarrow Z$ only by virtue of $X \rightarrow Y$ and $Y \rightarrow Z$.

Dependency Diagram

- The arrows above entities indicate all desirable dependencies.
- The arrows below the dependency diagram indicate less desirable dependencies -- partial dependencies and transitive dependencies.





Example:

Tournament Winners

<u>Tournament</u>	<u>Year</u>	Winner	Winner Date of Birth
Indiana Invitational	1998	Al Fredrickson	21 July 1975
Cleveland Open	1999	Bob Albertson	28 September 1968
Des Moines Masters	1999	Al Fredrickson	21 July 1975
Indiana Invitational	1999	Chip Masterson	14 March 1977

Because each row in the table needs to tell us who won a particular Tournament in a particular Year, the composite key {Tournament, Year} is a minimal set of attributes guaranteed to uniquely identify a row. That is, {Tournament, Year} is a candidate key for the table.

The breach of 3NF occurs because the non-prime attribute Winner Date of Birth is transitively dependent on the candidate key {Tournament, Year} via the non-prime attribute Winner. The fact that Winner Date of Birth is functionally dependent on Winner makes the table vulnerable to logical inconsistencies, as there is nothing to stop the same person from being shown with different dates of birth on different records.

In order to express the same facts without violating 3NF, it is necessary to split the table into two:

Tournament Winners

<u>Tournament</u>	<u>Year</u>	Winner
Indiana Invitational	1998	Al Fredrickson
Cleveland Open	1999	Bob Albertson
Des Moines Masters	1999	Al Fredrickson
Indiana Invitational	1999	Chip Masterson

Winner Dates of Birth

<u>Winner</u>	Date of Birth
Chip Masterson	14 March 1977
Al Fredrickson	21 July 1975
Bob Albertson	28 September 1968

Update anomalies cannot occur in these tables, because unlike before, **Winner** is now a primary key in the second table, thus allowing only one value for **Date of Birth** for each **Winner**.

Customer

CustID	CustName	AccNo	BankCode	Bank
1001	Rajesh	10999901	8921	HDFC
1002	Akash	10999902	8921	HDFC

3NF 

Customer

CustID	CustName	AccNo	BankCode
1001	Rajesh	10999901	8921
1002	Akash	10999902	8921

Bank

Bank Code	Bank
8921	HDFC
8901	HDFC