

## Understanding DBMS Architecture

A Database Management system is not always directly available for users and applications to access and store data in it. A Database Management system can be **centralised** (all the data stored at one location), **decentralised** (multiple copies of database at different locations) or **hierarchical**, depending upon its architecture.

**1-tier DBMS** architecture also exist; this is when the database is directly available to the user for using it to store data.

Generally, such a setup is used for local application development, where programmers communicate directly with the database for quick response.

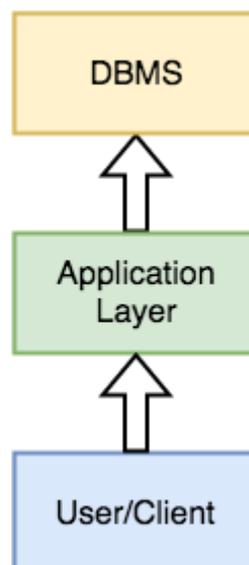
Database Architecture is logically of two types:

1. 2-tier DBMS architecture
2. 3-tier DBMS architecture

---

## 2-tier DBMS Architecture

2-tier DBMS architecture includes an **Application layer** between the user and the DBMS, which is responsible to communicate the user's request to the database management system and then send the response from the DBMS to the user. An application interface known as **ODBC** (Open Database Connectivity) provides an API that allow client side program to call the DBMS. Most DBMS vendors provide ODBC drivers for their DBMS.



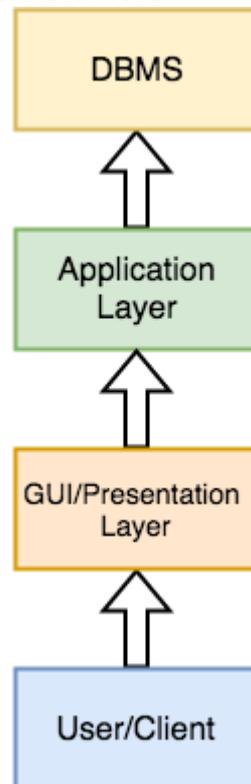
Such an architecture provides the DBMS extra security as it is not exposed to the End User directly. Also, security can be

improved by adding security and authentication checks in the Application layer too.

---

### 3-tier DBMS Architecture

3-tier DBMS architecture is the most commonly used architecture for web applications.



It is an extension of the 2-tier architecture. In the 2-tier architecture, we have an application layer which can be accessed programmatically to perform various operations on the DBMS. The application generally understands the Database Access Language and processes end users requests to the DBMS.

In 3-tier architecture, an additional Presentation or GUI Layer is added, which provides a graphical user interface for the End user to interact with the DBMS.

For the end user, the GUI layer is the Database System, and the end user has no idea about the application layer and the DBMS system.

If you have used **MySQL**, then you must have seen **PHPMyAdmin**, it is the best example of a 3-tier DBMS architecture.

## Normalization in DBMS: 1NF, 2NF, 3NF and BCNF in Database

**Normalization** is a process of organizing the data in database to avoid data redundancy, insertion anomaly, update anomaly & deletion anomaly. Let's discuss about anomalies first then we will discuss normal forms with examples.

### Anomalies in DBMS

There are three types of anomalies that occur when the database is not normalized. These are – Insertion, update and deletion anomaly. Let's take an example to understand this.

**Example:** Suppose a manufacturing company stores the employee details in a table named employee that has four attributes: emp\_id for storing employee's id, emp\_name for storing employee's name, emp\_address for storing employee's address and emp\_dept for storing the department details in which the employee works. At some point of time the table looks like this:

emp_id	emp_name	emp_address	emp_dept
101	Rick	Delhi	D001
101	Rick	Delhi	D002
123	Maggie	Agra	D890
166	Glenn	Chennai	D900
166	Glenn	Chennai	D004

The above table is not normalized. We will see the problems that we face when a table is not normalized.

**Update anomaly:** In the above table we have two rows for employee Rick as he belongs to two departments of the company. If we want to update the address of Rick then we have to update the same in two rows or the data will become inconsistent. If somehow, the correct address gets updated in one department but not in other then as per the database, Rick would be having two different addresses, which is not correct and would lead to inconsistent data.

**Insert anomaly:** Suppose a new employee joins the company, who is under training and currently not assigned to any department then we would not be able to insert the data into the table if emp\_dept field doesn't allow nulls.

**Delete anomaly:** Suppose, if at a point of time the company closes the department D890 then deleting the rows that are having emp\_dept as D890 would also delete the information of employee Maggie since she is assigned only to this department. To overcome these anomalies we need to normalize the data. In the next section we will discuss about normalization.

### Normalization

Here are the most commonly used normal forms:

- First normal form(1NF)
- Second normal form(2NF)
- Third normal form(3NF)
- Boyce & Codd normal form (BCNF)

#### First normal form (1NF)

As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

**Example:** Suppose a company wants to store the names and contact details of its employees. It creates a table that looks like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 9900012222
103	Ron	Chennai	7778881212

104	Lester	Bangalore	9990000123 8123450987
-----	--------	-----------	--------------------------

Two employees (Jon & Lester) are having two mobile numbers so the company stored them in the same field as you can see in the table above.

This table is **not in 1NF** as the rule says "each attribute of a table must have atomic (single) values", the emp\_mobile values for employees Jon & Lester violates that rule.

To make the table complies with 1NF we should have the data like this:

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

### Second normal form (2NF)

A table is said to be in 2NF if both the following conditions hold:

- Table is in 1NF (First normal form)
- No non-prime attribute is dependent on the proper subset of any candidate key of table.

An attribute that is not part of any candidate key is known as non-prime attribute.

**Example:** Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38

333	Physics	40
333	Chemistry	40

**Candidate Keys:** {teacher\_id, subject}

**Non prime attribute:** teacher\_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher\_age is dependent on teacher\_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "no non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

**teacher\_details table:**

teacher_id	teacher_age
111	38
222	38
333	40

**teacher\_subject table:**

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

**Third Normal form (3NF)**

A table design is said to be in 3NF if both the following conditions hold:

- Table must be in 2NF
- Transitive functional dependency of non-prime attribute on any super key should be removed.

An attribute that is not part of any candidate key is known as non-prime attribute.

In other words 3NF can be explained like this: A table is in 3NF if it is in 2NF and for each functional dependency  $X \rightarrow Y$  at least one of the following conditions hold:

- X is a super key of table
- Y is a prime attribute of table

An attribute that is a part of one of the candidate keys is known as prime attribute.

**Example:** Suppose a company wants to store the complete address of each employee, they create a table named employee\_details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Rohan	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

**Super keys:** {emp\_id}, {emp\_id, emp\_name}, {emp\_id, emp\_name, emp\_zip}...so on

**Candidate Keys:** {emp\_id}

**Non-prime attributes:** all attributes except emp\_id are non-prime as they are not part of any candidate keys.

Here, emp\_state, emp\_city & emp\_district dependent on emp\_zip. And, emp\_zip is dependent on emp\_id that makes non-prime attributes (emp\_state, emp\_city & emp\_district) transitively dependent on super key (emp\_id). This violates the rule of 3NF.

To make this table complies with 3NF we have to break the table into two tables to remove the transitive dependency:

**employee table:**

emp_id	emp_name	emp_zip
1001	John	282005
1002	Ajeet	222008
1006	Rohan	282007

1101	Lilly	292008
1201	Steve	222999

**employee\_zip table:**

emp_zip	emp_state	emp_city	emp_district
282005	UP	Agra	Dayal Bagh
222008	TN	Chennai	M-City
282007	TN	Chennai	Urrapakkam
292008	UK	Pauri	Bhagwan
222999	MP	Gwalior	Ratan

**Boyce Codd normal form (BCNF)**

It is an advance version of 3NF that's why it is also referred as 3.5NF. BCNF is stricter than 3NF. A table complies with BCNF if it is in 3NF and for every [functional dependency](#) X->Y, X should be the super key of the table.

**Example:** Suppose there is a company wherein employees work in **more than one department**. They store the data like this:

emp_id	emp_nationality	emp_dept	dept_type	dept_no_of_emp
1001	Austrian	Production and planning	D001	200
1001	Austrian	stores	D001	250
1002	American	design and technical support	D134	100
1002	American	Purchasing department	D134	600

**Functional dependencies in the table above:**

emp\_id -> emp\_nationality

emp\_dept -> {dept\_type, dept\_no\_of\_emp}

**Candidate key:** {emp\_id, emp\_dept}

The table is not in BCNF as neither emp\_id nor emp\_dept alone are keys.

To make the table comply with BCNF we can break the table in three tables like this:

**emp\_nationality table:**

emp_id	emp_nationality
1001	Austrian
1002	American

**emp\_dept table:**

emp_dept	dept_type	dept_no_of_emp
Production and planning	D001	200
stores	D001	250
design and technical support	D134	100
Purchasing department	D134	600

**emp\_dept\_mapping table:**

emp_id	emp_dept
1001	Production and planning
1001	stores
1002	design and technical support
1002	Purchasing department

**Functional dependencies:**

emp\_id -> emp\_nationality

emp\_dept -> {dept\_type, dept\_no\_of\_emp}

**Candidate keys:**

For first table: emp\_id

For second table: emp\_dept

For third table: {emp\_id, emp\_dept}

## ***Concurrency Control***

As we have seen above, when there are multiple transactions executing at the same time on same data, it may affect the result of the transaction. Hence it is necessary to maintain the order of execution of those transactions. In addition, it should not alter the ACID property of a transaction.

In order to maintain the concurrent access of transactions, two protocols are introduced.

- **Lock Based Protocol:** - Lock is in other words called as access. In this type of protocol any transaction will not be processed until the transaction gets the lock on the record. That means any transaction will not retrieve or insert or update or delete the data unless it gets the access to that particular data.

These locks are broadly classified as Binary locks and shared / exclusive locks.

In **binary lock** data can either be locked or unlocked. It will have only these two states. It can be locked for retrieve or insert or update or delete the data or unlocked for not using the data.

In **shared / exclusive lock** technique the data is said to be exclusively locked if for insert / update /delete. When it is exclusively locked no other transaction can read or write the data. When a data is read from the database, then its lock is shared i.e.; the data can be read by other transaction too but it cannot be changed while retrieving the data.

***Lock based protocols are of 4 types***

**Simplistic Lock Protocol:** -As the name suggests it is the simplest way of locking the data during the transaction. This protocol allows all the transaction to get the lock on the data

before insert / update /delete on it. After completing the transaction, it will unlock the data.

**Pre-claiming Protocol:** - In this protocol, it evaluates the transaction to list all the data items on which transaction needs lock. It then requests DBMS for the lock on all those data items before the transaction begins. If DBMS gives the lock on all the data, then this protocol allows the transaction to begin. Once the transaction is complete, it releases all the locks. If all locks are given by DBMS, then it reverts the transactions and waits for the lock.

For example, if we have to calculate total marks of 3 subjects, then this protocol will evaluate the transaction and list the locks on subject1 marks, subject2 marks and then subject3 marks. Once it gets all the locks, it will start the transaction.

**Two Phase Locking Protocol (2PL):** - In this type of protocol, as the transaction begins to execute, it starts requesting for the locks that it needs. It goes on requesting for the locks as and when it is needed. Hence it has a growing phase of locks. At one stage it will have all the locks. Once the transaction is complete it goes on releasing the locks. Hence it will have descending phase of locks. Thus this protocol has two phases – growing phase of locks and shrinking phase of locks.

For example, if we have to calculate total marks of 3 subjects, then this protocol will go on asking for the locks on subject1 marks, subject2 marks and then subject3 marks. As and when it gets the locks on the subject marks it reads the marks. It does not wait till all the locks are received. Then it will have total calculation. Once it is complete it release the lock on subject3 marks, subject2 marks and subject1 marks.

In this protocol, if we need to have exclusive lock on any data for writing, then we have to first get the shared lock for reading. Then we have to request / modify the lock to exclusive lock.

**Strict Two Phase Locking (Strict 2PL):** - This protocol is similar to 2PL in the first phase. Once it receives the lock on the data, it completes the transaction. Here it does not release the locks as it is used and no more required. It waits till whole transaction to complete and commit, then it releases all the locks at a time. This protocol hence does not have shrinking phase of lock release.

In the example of calculating total marks of 3 subjects, locks are achieved at growing phase of the transaction and once it receives all the locks, it executes the transaction. Once the transaction is fully complete, it releases all the locks together.

**•Time Stamp Based Protocol:** - As we have seen above in lock based protocol, it acquires locks at the time of execution. But in this method, as soon as a transaction is created it assigns the order of the transaction. The order of the transaction is nothing but the ascending order of the transaction creation. The priority for older transaction is given to execute first. This protocol uses system time or logical counter to determine the time stamp of the transaction.

Suppose there are two transactions T1 and T2. Suppose T1 has entered the system at time 0005 and T2 has entered the system at 0008 clock time. Priority will be given to T1 to execute first as it is entered the system first.

In addition to the timestamp of a transaction, this protocol also maintains the timestamp of last 'read' and 'write' operation on a data. Based on the timestamp of transaction and the data which it is accessing a timestamp ordering protocol is defined.

### **According to this protocol:**

If a transaction T is a read transaction on data X then

This algorithm states that if there is an active write operation on data X when a transaction T is requesting for X, then reject the transaction T. If the transaction T is started as soon as write is complete or no going write operation on X, then execute T.

For example, if there is an update on marks1 on MARKS table and meanwhile there is a request to read marks1, then do not perform read marks1. This is because there is an update being happening on marks1. If there was an update on marks1 which is executed long back or it is complete just now and there is a request to read marks1, then system will allow reading marks1.

- If a transaction T is a write transaction on data X then

This algorithm describes about write operation. If there is an active read or write on data X, and at the same time if the transaction T is requesting for X, then the transaction is rejected. If there is no active read / write operation on X, then execute the transaction.

Suppose T1 is reading marks1 from MARKS table. Meanwhile transaction T2 begins and tries to update marks1 in MARKS. Then the transaction T2 is rejected and rolled back.

## FUNCTION DEPENDENCY

Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. If R is a relation with attributes X and Y, a functional dependency between the attributes is represented as  $X \rightarrow Y$ , which specifies Y is functionally dependent on X.

- The attributes of a table is said to be dependent on each other when an attribute of a table uniquely identifies another attribute of the same table.

For example:

- Suppose we have a student table with attributes: Stu\_Id, Stu\_Name, Stu\_Age.
- Here Stu\_Id attribute uniquely identifies the Stu\_Name attribute of student table because if we know the student id we can tell the student name associated with it.

Functional dependency and can be written as:

$\text{Stu\_Id} \rightarrow \text{Stu\_Name}$ .

we can say Stu\_Name is functionally dependent on Stu\_Id.

Formally: If column A of a table uniquely identifies the column B of same table then it can represented as  $A \rightarrow B$

(Attribute B is functionally dependent on attribute A)

Types of Functional Dependencies:

- Trivial functional dependency
- Non-trivial functional dependency
- Multivalued dependency
- Transitive dependency

## I. Trivial functional dependency:

- The dependency of an attribute on a set of attributes is known as trivial functional dependency if the set of attributes includes that attribute. Symbolically:

- $A \rightarrow B$  is trivial functional dependency if  $B$  is a subset of  $A$ .

- The following dependencies are also trivial:

$A \rightarrow A$  &  $B \rightarrow B$

- For example: Consider a table with two columns `Student_id` and `Student_Name`.  $\{Student\_Id, Student\_Name\} \rightarrow Student\_Id$  is a trivial functional dependency as `Student_Id` is a subset of  $\{Student\_Id, Student\_Name\}$ .

- Also,  $Student\_Id \rightarrow Student\_Id$  &  $Student\_Name \rightarrow Student\_Name$  are trivial dependencies too.

## II. Non-trivial functional dependency:

- If a functional dependency  $X \rightarrow Y$  holds true where  $Y$  is not a subset of  $X$  then this dependency is called non trivial Functional dependency.

Example : An employee table with three attributes: `emp_id`, `emp_name`, `emp_address`. The following functional dependencies are non-trivial:  $emp\_id \rightarrow emp\_name$  (`emp_name` is not a subset of `emp_id`)  $emp\_id \rightarrow emp\_address$  (`emp_address` is not a subset of `emp_id`)

- On the other hand, the following dependencies are trivial:  $\{emp\_id, emp\_name\} \rightarrow emp\_name$  [`emp_name` is a subset of  $\{emp\_id, emp\_name\}$ ]

Completely non trivial FD: If a Functional dependency  $X \rightarrow Y$  holds true where  $X \cap Y$  is Null then this dependency is said to be completely non trivial function dependency.

### III. Multivalued dependency:

- Multivalued dependency occurs when there are more than one independent multivalued attributes in a table.
- a multivalued dependency is a full constraint between two sets of attributes in a relation. In contrast to the functional dependency, the multivalued dependency requires that certain tuples be present in a relation.
- Consider a bike manufacture company, which produces two colors (Black and white) in each model every year.
- Here columns `manuf_year` and `color` are independent of each other and dependent on `bike_model`. In this case these two columns are said to be multivalued dependent on `bike_model`.

These dependencies can be represented like this:

- `bike_model ->> manuf_year` • `bike_model ->> color`

IV. Transitive dependency: A functional dependency is said to be transitive if it is indirectly formed by two functional dependencies.

$X \rightarrow Z$  is a transitive dependency if the following three functional dependencies hold true:  $X \rightarrow Y$ .

$Y$  does not  $\rightarrow X$   $Y \rightarrow Z$

A transitive dependency can only occur in a relation of three or more attributes. This dependency helps us normalizing the database in 3NF (3rd Normal Form).

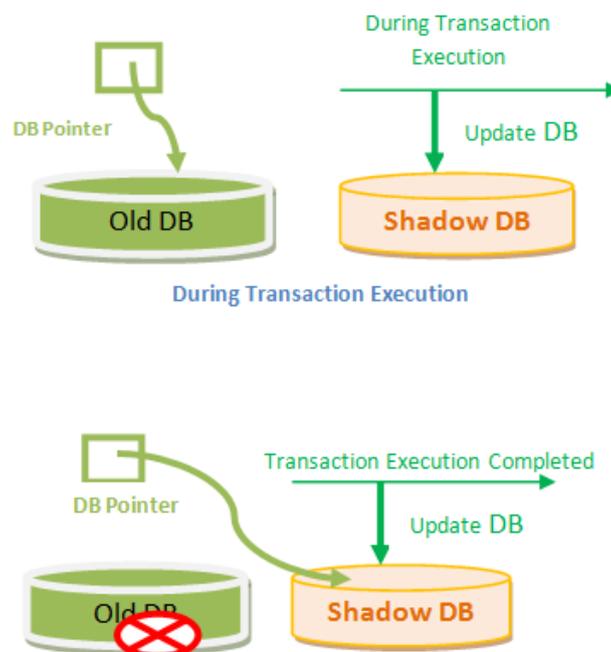
Example :-

- $\{Book\} \rightarrow \{Author\}$  (if we know the book, we know the author name)
- $\{Author\}$  does not  $\rightarrow \{Book\}$
- $\{Author\} \rightarrow \{Author\_age\}$
- Therefore as per the rule of transitive dependency:

- {Book} -> {Author\_age} should hold, that makes sense because if we know the book name we can know the author's age.

## Shadow paging in DBMS

This is the method where all the transactions are executed in the primary memory or the shadow copy of database. Once all the transactions completely executed, it will be updated to the database. Hence, if there is any failure in the middle of transaction, it will not be reflected in the database. Database will be updated after all the transaction is complete.



A database pointer will be always pointing to the consistent copy of the database, and copy of the database is used by transactions to update. Once all the transactions are complete, the DB pointer is modified to point to new copy of DB, and old copy is deleted. If there is any failure during the transaction, the pointer will be still pointing to old copy of database, and shadow database will be deleted. If the transactions are complete then the pointer is changed to point to shadow DB, and old DB is deleted.

As we can see in above diagram, the DB pointer is always pointing to consistent and stable database. This mechanism assumes that there will not be any disk failure and only one transaction executing at a time so that the shadow DB can hold

the data for that transaction. It is useful if the DB is comparatively small because shadow DB consumes same memory space as the actual DB. Hence it is not efficient for huge DBs. In addition, it cannot handle concurrent execution of transactions. It is suitable for one transaction at a time.

### ***Introduction to Query Optimization***

- Query optimization is a difficult part of the query processing.
- It determines the efficient way to execute a query with different possible query plans.
- It cannot be accessed directly by users once the queries are submitted to the database server or parsed by the parser.
- A query is passed to the query optimizer where optimization occurs.
- Main aim of Query Optimization is to minimize the cost function,  
 $I/O \text{ Cost} + CPU \text{ Cost} + \text{Communication Cost}$
- It defines how an RDBMS can improve the performance of the query by re-ordering the operations.
- It is the process of selecting the most efficient query evaluation plan from among various strategies if the query is complex.
- It computes the same result as per the given expression, but it is a least costly way of generating result.

#### Importance of Query Optimization:

- Query optimization provides faster query processing.
- It requires less cost per query.
- It gives less stress to the database.
- It provides high performance of the system.
- It consumes less memory.

1. Suppose we are given a relation scheme  $R=(A,B,C,G,H,I)$ , and the set of functional dependencies:
2.  $F=\{ (A \rightarrow B), (A \rightarrow C), \{CG \rightarrow H\}, \{CG \rightarrow I\}, (B \rightarrow H) \}$

Then the functional dependency  $A \rightarrow H$  is logically implied.

The **closure** of a set  $F$  of functional dependencies is the set of all functional dependencies logically implied by  $F$ .

3. We denote the closure of  $F$  by  $F^+$ .
4. To compute  $F^+$ , we can use some rules of inference called **Armstrong's Axioms**:
  - **Reflexivity rule:** if  $\alpha$  is a set of attributes and  $(\beta \subseteq \alpha)$   $B$  is subset of  $\alpha$ , then  $\alpha \rightarrow \beta$  holds.
  - **Augmentation rule:** if  $\alpha \rightarrow \beta$  holds, and  $\gamma$  is a set of attributes, then  $\gamma\alpha \rightarrow \gamma\beta$  holds.
  - **Transitivity rule:** if  $\alpha \rightarrow \beta$  holds, and  $\beta \rightarrow \gamma$  holds, then  $\alpha \rightarrow \gamma$  holds.
5. These rules are **sound** because they do not generate any incorrect functional dependencies. They are also **complete** as they generate all of  $F^+$ .
6. To make life easier we can use some additional rules, derivable from Armstrong's Axioms:
  - **Union rule:** if  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$ , then  $\alpha \rightarrow \beta\gamma$  holds.
  - **Decomposition rule:** if  $\alpha \rightarrow \beta\gamma$  holds, then  $\alpha \rightarrow \beta$  and  $\alpha \rightarrow \gamma$  both hold.
  - **Pseudo transitivity rule:** if  $\alpha \rightarrow \beta$  holds, and  $\gamma\beta \rightarrow \delta$  holds, then  $\alpha\gamma \rightarrow \delta$  holds.
7. Applying these rules to the scheme and set  $F$  mentioned above, we can derive the following:
  - $A \rightarrow H$ , as we saw by the transitivity rule.
  - $CG \rightarrow HI$  by the union rule.
  - $AG \rightarrow I$  by several steps:
    - Note that  $A \rightarrow C$  holds.
    - Then  $AG \rightarrow CG$ , by the augmentation rule.
    - Now by transitivity,  $AG \rightarrow I$ .

# DBMS Database Models

A Database model defines the logical design and structure of a database and defines how data will be stored, accessed and updated in a database management system. While the **Relational Model** is the most widely used database model, there are other models too:

- Hierarchical Model
  - Network Model
  - Entity-relationship Model
  - Relational Model
- 

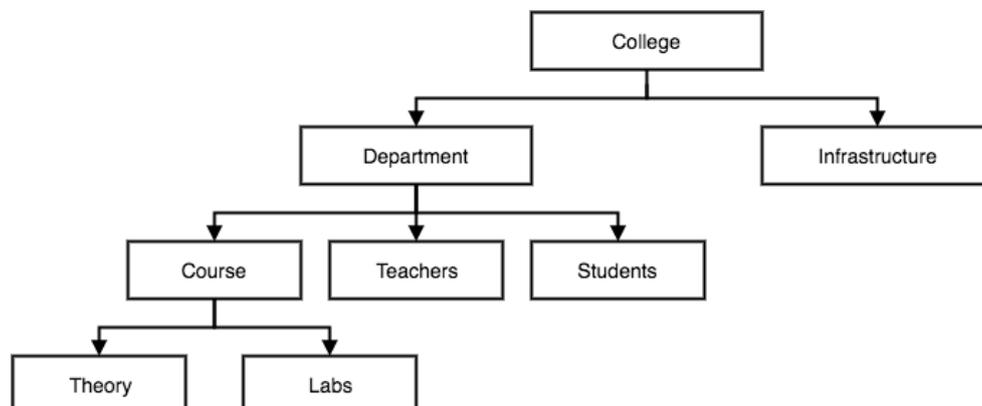
## Hierarchical Model

This database model organises data into a tree-like-structure, with a single root, to which all the other data is linked. The hierarchy starts from the **Root** data, and expands like a tree, adding child nodes to the parent nodes.

In this model, a child node will only have a single parent node.

This model efficiently describes many real-world relationships like index of a book, recipes etc.

In hierarchical model, data is organised into tree-like structure with one one-to-many relationship between two different types of data, for example, one department can have many courses, many professors and of-course many students.

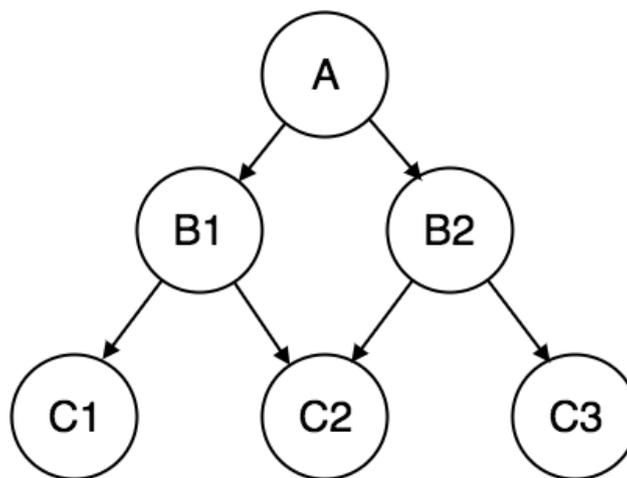


# Network Model

This is an extension of the Hierarchical model. In this model data is organised more like a graph, and are allowed to have more than one parent node.

In this database model data is more related as more relationships are established in this database model. Also, as the data is more related, hence accessing the data is also easier and fast. This database model was used to map many-to-many data relationships.

This was the most widely used database model, before Relational Model was introduced.



---

# Entity-relationship Model

In this database model, relationships are created by dividing object of interest into entity and its characteristics into attributes.

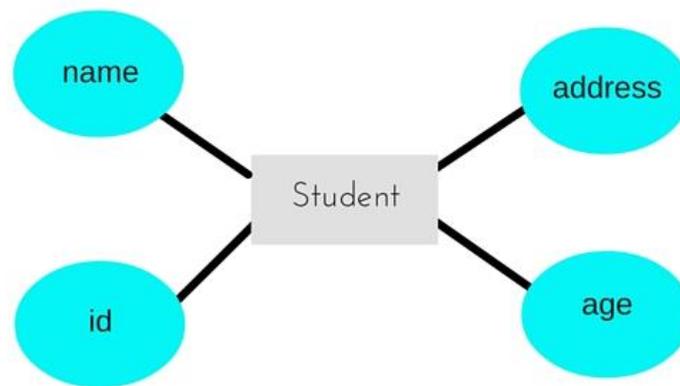
Different entities are related using relationships.

E-R Models are defined to represent the relationships into pictorial form to make it easier for different stakeholders to understand.

This model is good to design a database, which can then be turned into tables in relational model (explained below).

Let's take an example, If we have to design a School Database, then **Student** will be an **entity** with **attributes** name, age, address etc. As **Address** is generally complex, it can be another **entity** with **attributes** street name, pincode, city etc, and there will be a relationship between them.

Relationships can also be of different types.



---

## Relational Model

In this model, data is organised in two-dimensional **tables** and the relationship is maintained by storing a common field.

This model was introduced by E.F Codd in 1970, and since then it has been the most widely used database model, infact, we can say the only database model used around the world.

The basic structure of data in the relational model is tables. All the information related to a particular type is stored in rows of that table.

Hence, tables are also known as **relations** in relational model.

In the coming tutorials we will learn how to design tables, normalize them to reduce data redundancy and how to use Structured Query language to access data from tables.

student_id	name	age
1	Akon	17
2	Bkon	18
3	Ckon	17
4	Dkon	18

subject_id	name	teacher
1	Java	Mr. J
2	C++	Miss C
3	C#	Mr. C Hash
4	Php	Mr. P H P

student_id	subject_id	marks
1	1	98
1	2	78
2	1	76
3	2	88

## Database System versus File System

<b>DBMS</b>	<b>File Processing System</b>
Minimal data redundancy problem in DBMS	Data Redundancy problem exists
Data Inconsistency does not exist	Data Inconsistency exist here
Accessing database is easier	Accessing is comparatively difficult
The problem of data isolation is not found in database	Data is scattered in various files and files may be of different format, so data isolation problem exists
Transactions like insert, delete, view, updating, etc are possible in database	In file system, transactions are not possible
Concurrent access and recovery is possible in database	Concurrent access and recovery is not possible
Security of data	Security of data is not good
A database manager (administrator) stores the relationship in form of structural tables	A file manager is used to store all relationships in directories in file systems.