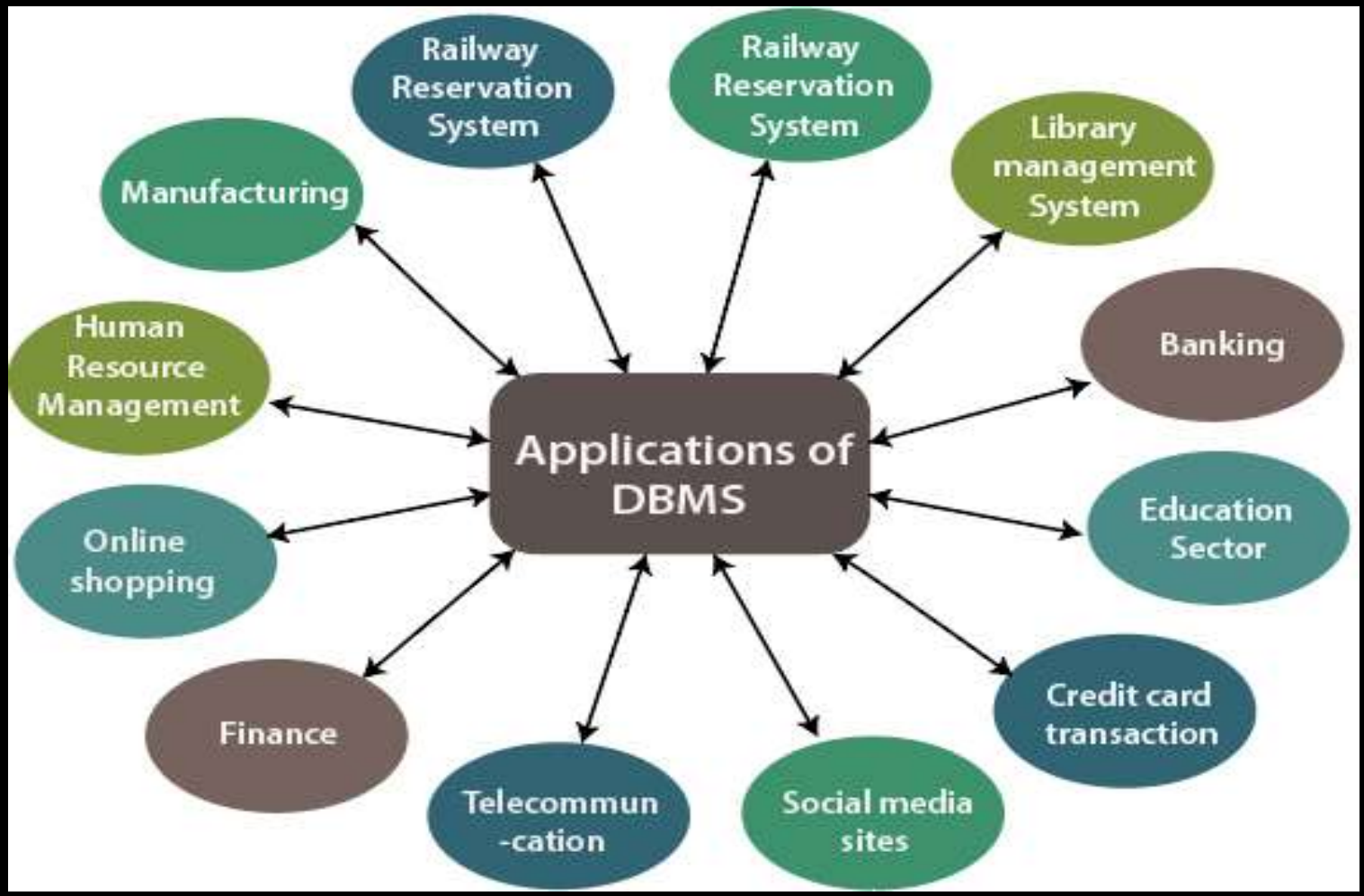
DBMS
[Database Management System]

PROF. SANJAY SHARMA

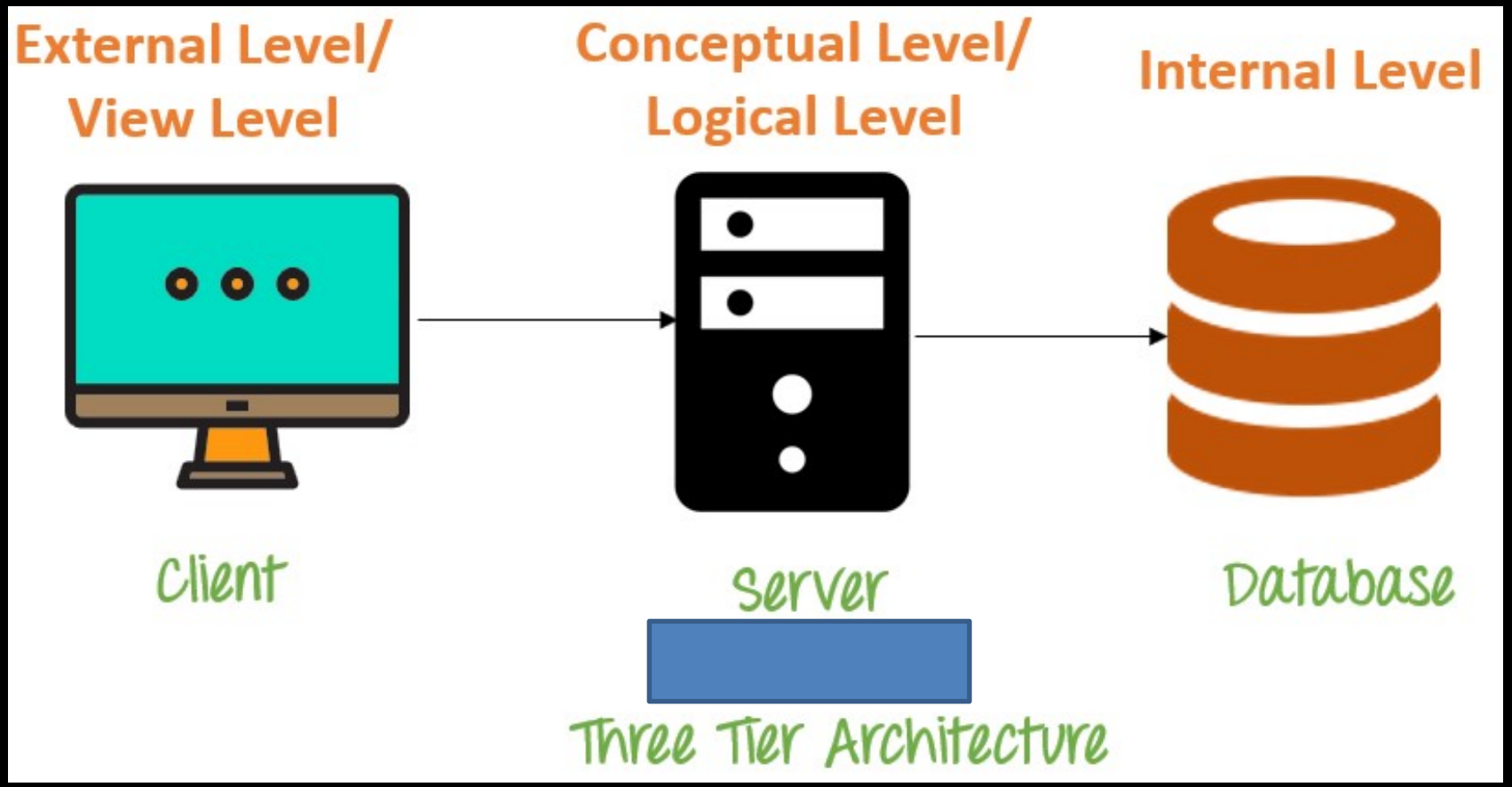
P.M.B. GUJARATI SCIENCE COLLEGE, INDORE
1 Nasia Road, Indore

Purpose of Database Systems

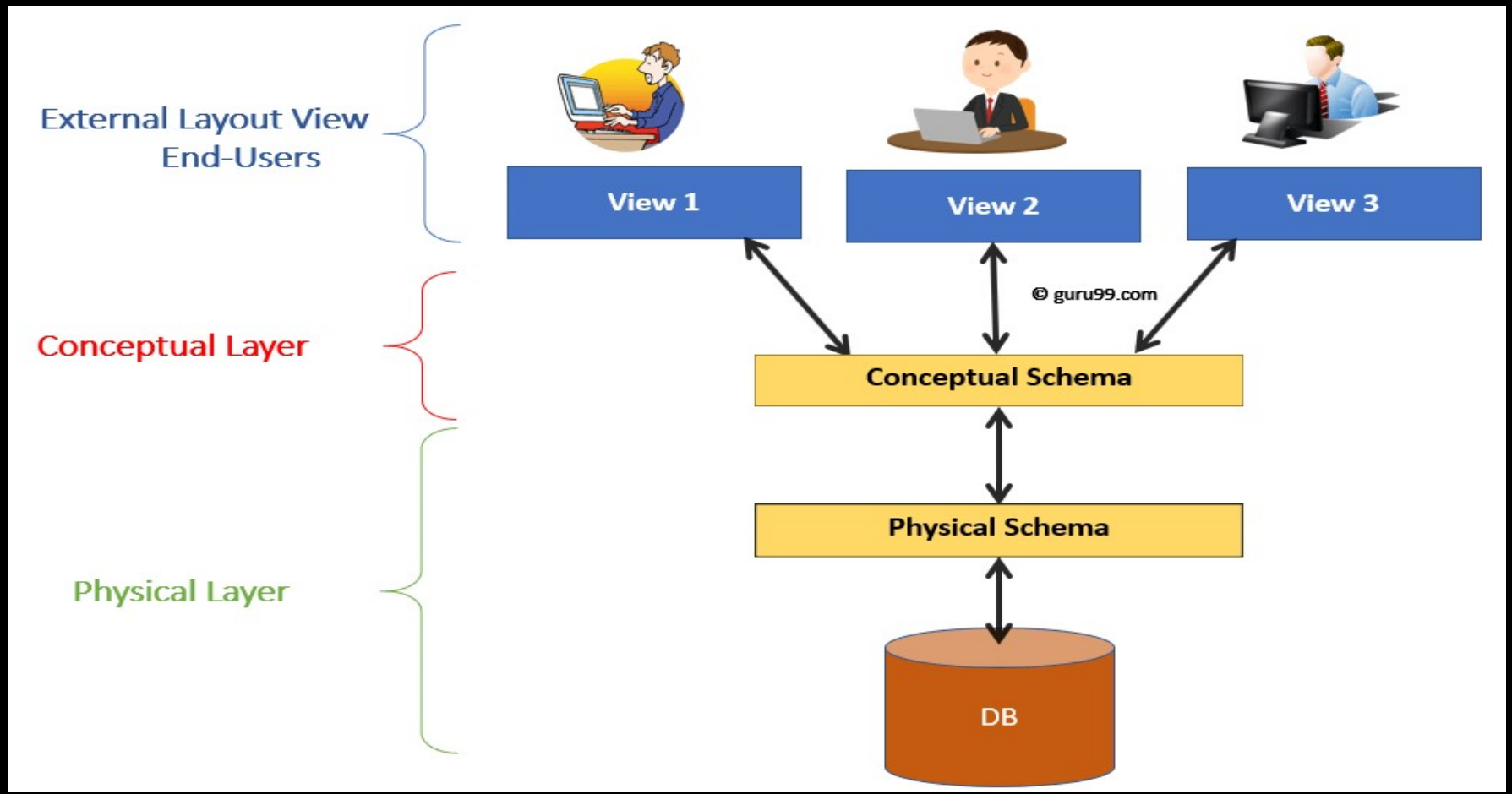
- In the early days, database applications were built directly on top of file systems
- Drawbacks of using file systems to store data:
  - Data redundancy and inconsistency
    - Multiple file formats, duplication of information in different files
  - Difficulty in accessing data
    - Need to write a new program to carry out each new task
  - Data isolation — multiple files and formats
  - Integrity problems
    - Integrity constraints (e.g. account balance > 0) become "buried" in program code rather than being stated explicitly
    - Hard to add new constraints or change existing ones


Railway Reservation System
Railway Reservation System
Manufacturing
Library management System
Human Resource Management
Banking
Applications of DBMS
Online shopping
Education Sector
Finance
Credit card transaction
Telecommun -cation
Social media sites

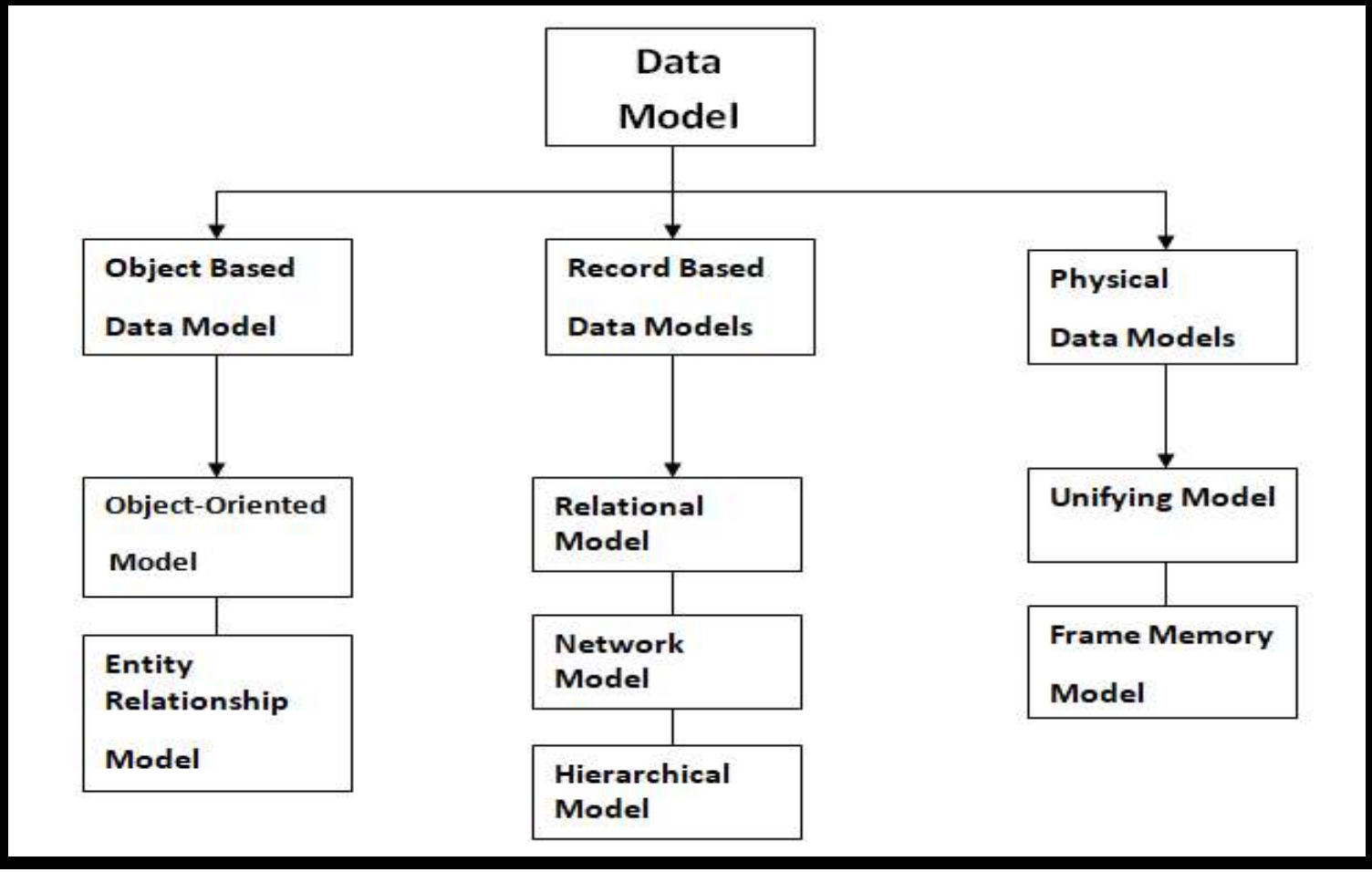
DBMS VIEWS


External Level/
View Level
Conceptual Level/
Logical Level
Internal Level
Client
Server
Database
Three Tier Architecture

DBMS VIEWS


External Layout View
End-Users
Conceptual Layer
Physical Layer
View 1
View 2
View 3
© guru99.com
Conceptual Schema
Physical Schema
DB

VARIOUS DATA MODELS



RELATIONAL DATA MODEL(Data stored in the form relations)

TYPES OF RELATIONSHIP

One : One	1 : 1
One : Many	1 : M
Many : One	N : 1
Many : Many	N : M

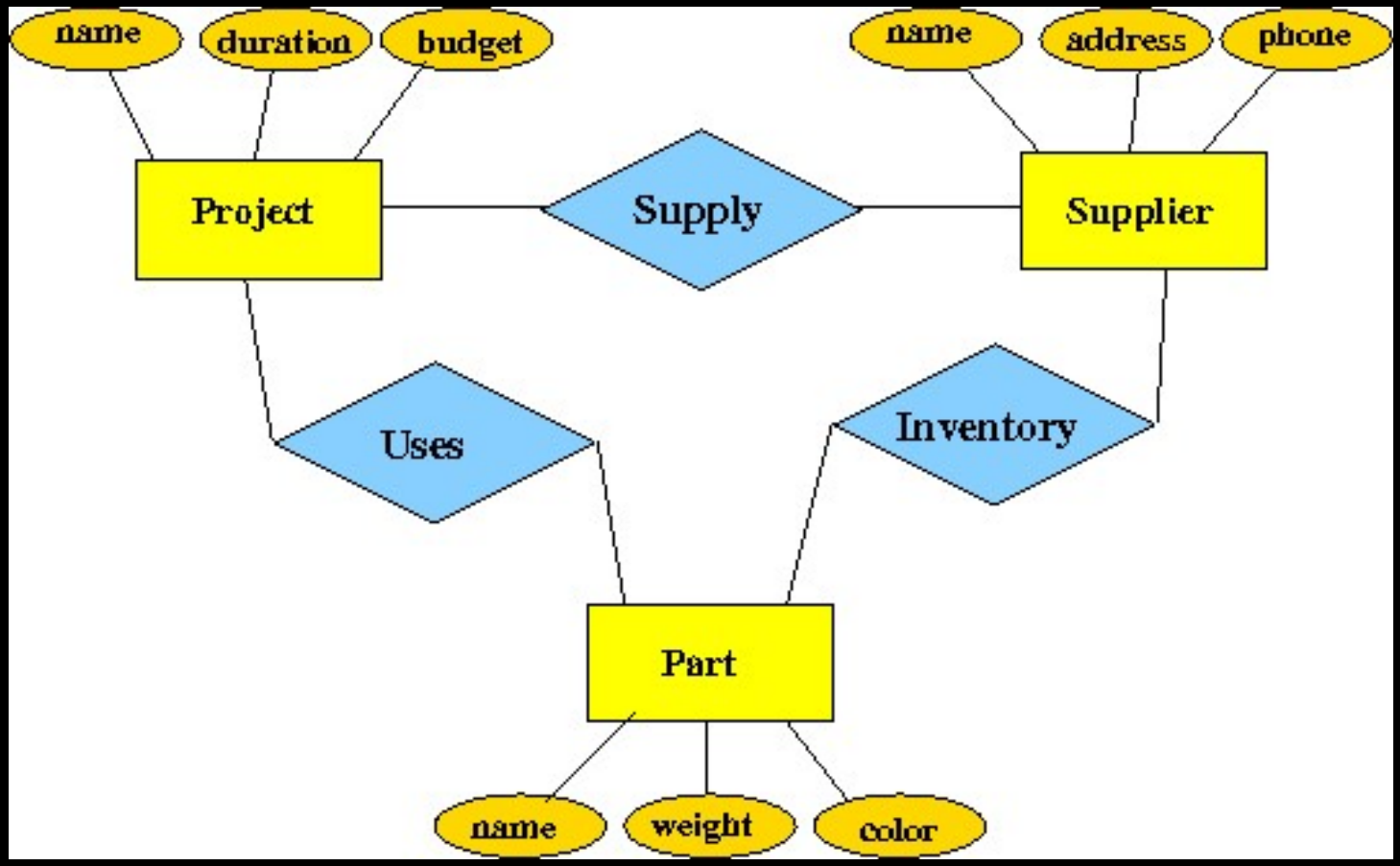
BASIC OPERATIONS ON A DATABASE

Insertion of record

Deletion of record

Updating of record

RELATIONAL DATA MODEL(Data stored in the form relations)



RELATIONAL DATA MODEL(Data stored in the form relations)

ADVANTAGES:

➨It is easy to use.
➨It is secured in nature.
➨The data manipulation can be done.
➨It limits redundancy and replication of the data.
➨It offers better data integrity.
➨It provides better physical data independence.
➨It offers logical database independence i.e. data can be viewed in different ways by the different users.
➨It provides better backup and recovery procedures.
➨It provides multiple interfaces.
➨Multiple users can access the database which is not possible in DBMS.

RELATIONAL DATA MODEL(Data stored in the form relations)

DISADVANTAGES:

Software is expensive.

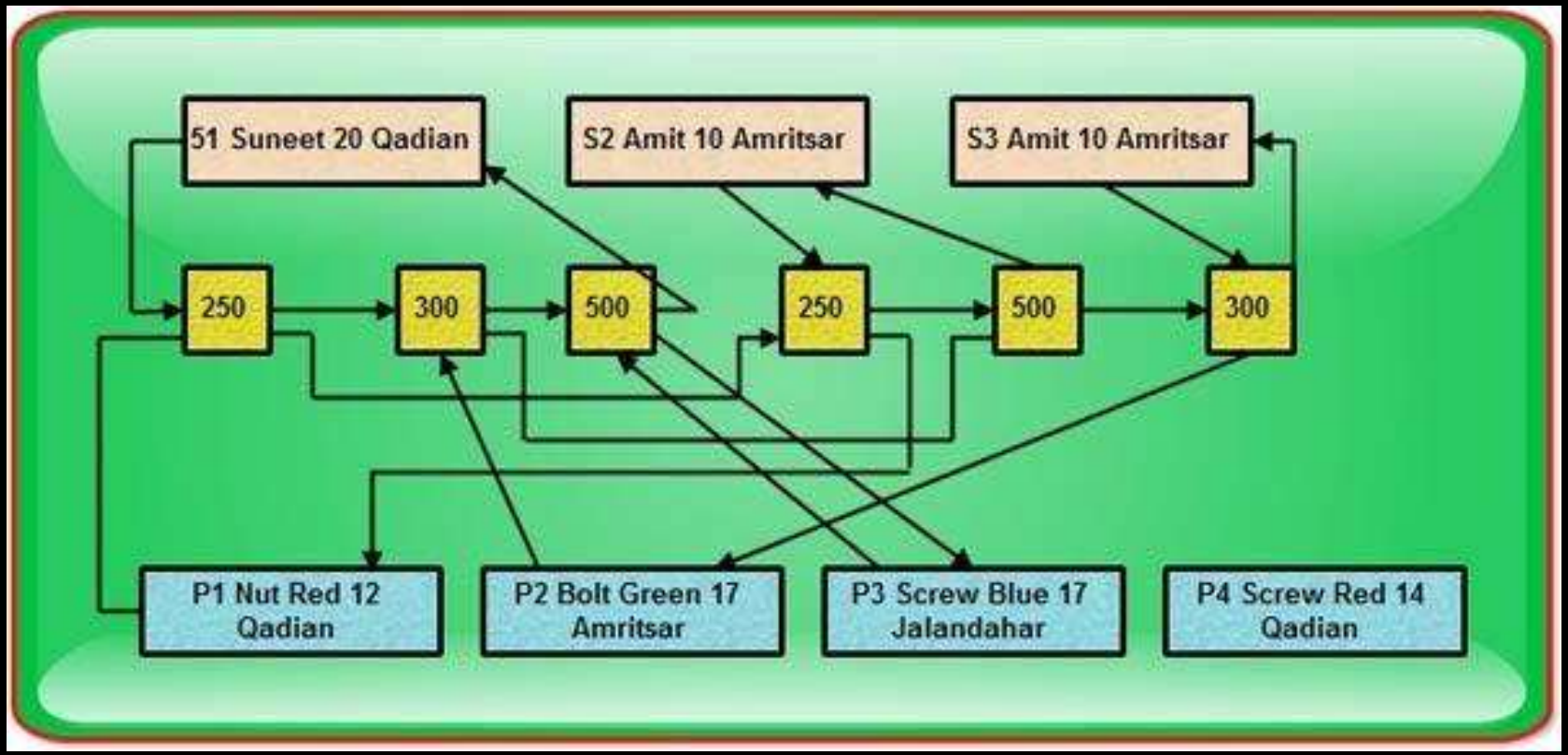
Complex software refers to expensive hardware and hence increases overall cost to avail the RDBMS service.

It requires skilled human resources to implement.

Certain applications are slow in processing.

it is difficult to recover the lost data.

NETWORK DATA MODEL(Data stored in the form of Links and Pointers)



NETWORK DATA MODEL(Data stored in the form of Links and Pointers)

ADVANTAGES OF NETWORK MODEL:

1.)Conceptual simplicity-Just like the hierarchical model,the network model is also conceptually simple and easy to design.

2.)Capability to handle more relationship types-The network model can handle the one to many and many to many relationships which is real help in modeling the real life situations.

3.) Ease of data access-The data access is easier and flexible than the hierarchical model.

4.) Data integrity- The network model does not allow a member to exist without an owner.

5.) Data independence- The network model is better than the hierarchical model in isolating the programs from the complex physical storage details.

6.) Database standards

NETWORK DATA MODEL(Data stored in the form of Links and Pointers)

DISADVANTAGE OF NETWORK MODEL:

System complexity- All the records are maintained using pointers and hence the whole database structure becomes very complex.

Operational Anomalies- The insertion, deletion and updating operations of any record require large number of pointers adjustments.

Absence of structural independence-structural changes to the database is very difficult.

NETWORK DATA MODEL(Data stored in the form hierarchy)

NETWORK DATA MODEL(Data stored in the form hierarchy)

ADVANTAGES:

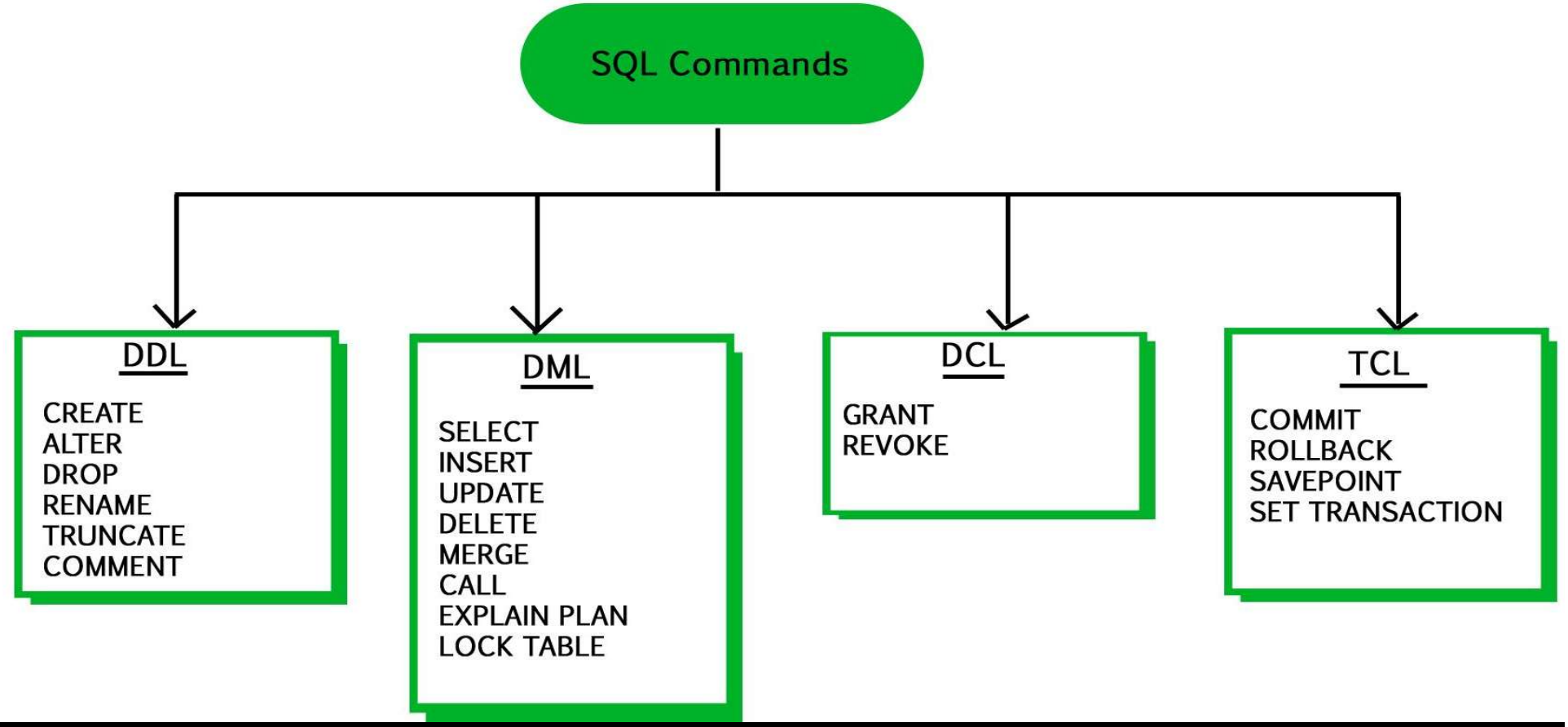
- Conceptual simplicity
- Database Security and Integrity
- Database Independence
- Efficiency

NETWORK DATA MODEL(Data stored in the form hierarchy)

DISADVANTAGES:

- One parent per child.
- Complex (users require physical representation of **database**)
- Navigation system is complex.
- Data must be organized in a **hierarchical** way without compromising the information.
- Lack structural independence.
- Many too many relationships not supported.
- Data independence.

DDL, DML, DCL, TCL



Entity Relationship

Entity:

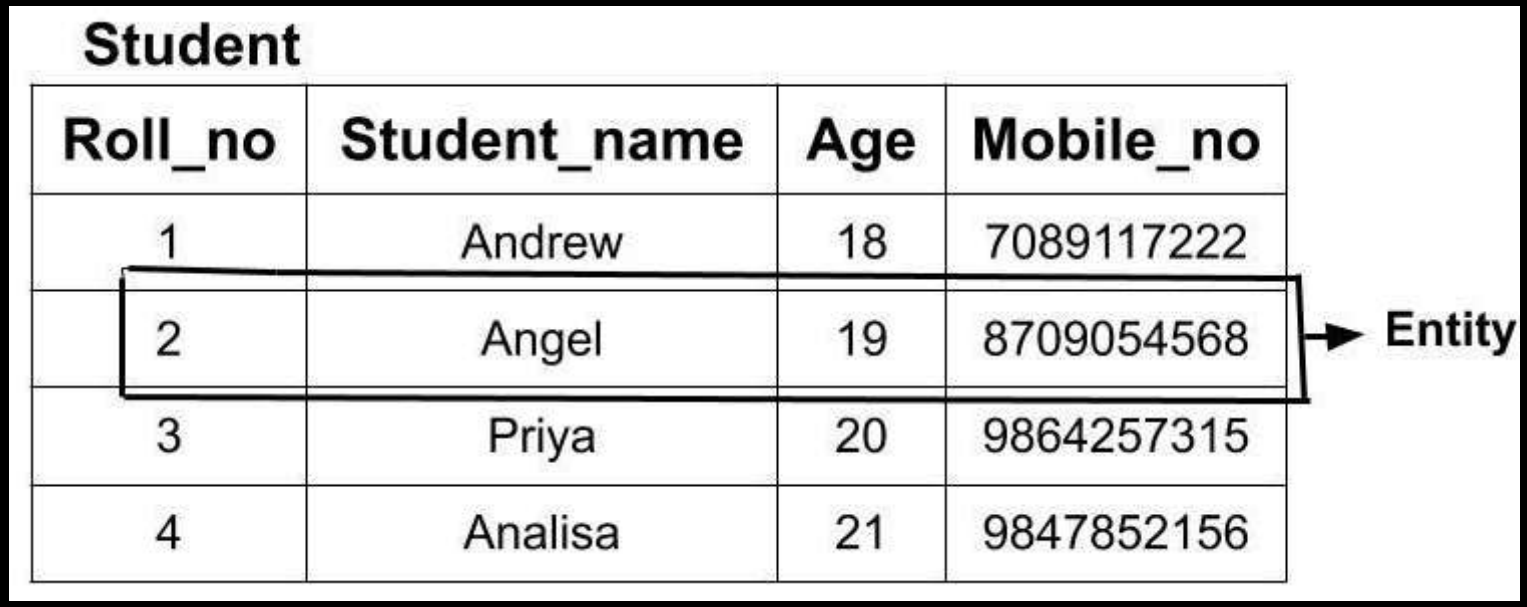
An entity is a real-world thing which can be distinctly identified like a person, place or a concept. It is an object which is distinguishable from others. If we cannot distinguish it from others then it is an object but not an entity. An entity can be of two types:

Tangible Entity: Tangible Entities are those entities which exist in the real world physically. ***Example:*** Person, car, etc.

Intangible Entity: Intangible Entities are those entities which exist only logically and have no physical existence. ***Example:*** Bank Account, etc.

Example: If we have a table of a Student (Roll_no, Student_name, Age, Mobile_no) then each student in that table is an entity and can be uniquely identified by their Roll Number i.e Roll_no.

Entity Relationship



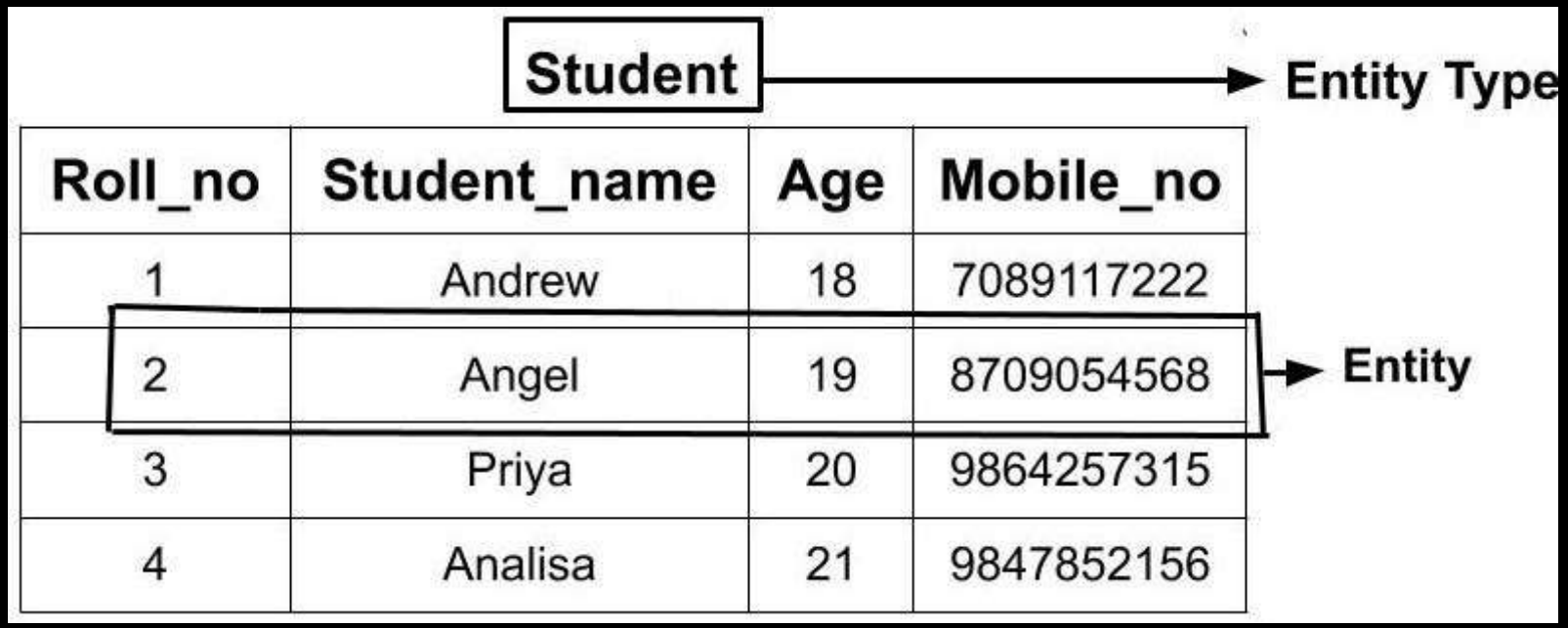
Entity Relationship

Note: In E-R model we don't represent the data but we represent the structure or schema. When we convert E-R model to relational model then data can be stored in tuple or row and hence, represented as an entity.**Entity Type**

The entity type is a collection of the entity having similar attributes. In the above Student table example, we have each row as an entity and they are having common attributes i.e each row has its own value for attributes Roll_no, Age, Student_name and Mobile_no. So, we can define the above STUDENT table as an entity type because it is a collection of entities having the same attributes.

So, an entity type in an ER diagram is defined by a name(here, STUDENT) and a set of attributes(here, Roll_no, Student_name, Age, Mobile_no). The table below shows how the data of different entities(different students) are stored.

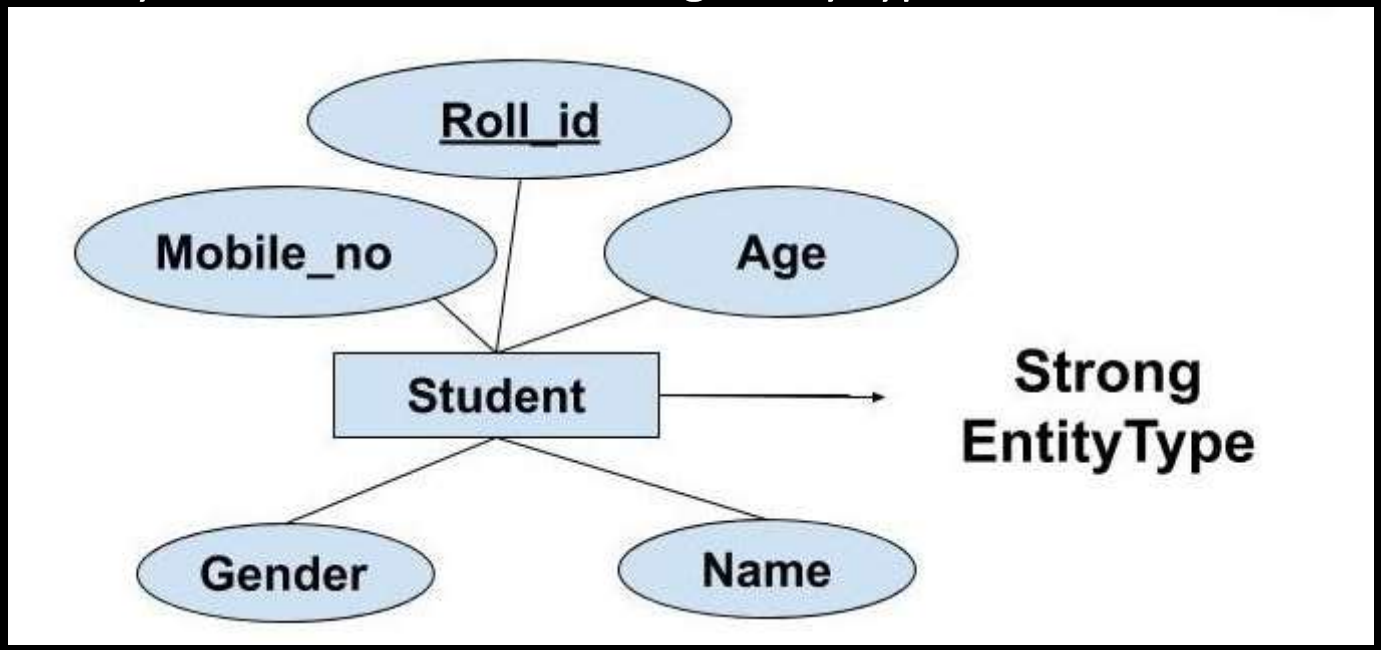
Entity Relationship



Entity Relationship

Types of Entity type Strong Entity Type Weak Entity Type

Strong Entity Type: Strong entity are those entity types which has a key attribute. The primary key helps in identifying each entity uniquely. It is represented by a rectangle. In the above example, Roll_no identifies each element of the table uniquely and hence, we can say that STUDENT is a strong entity type.

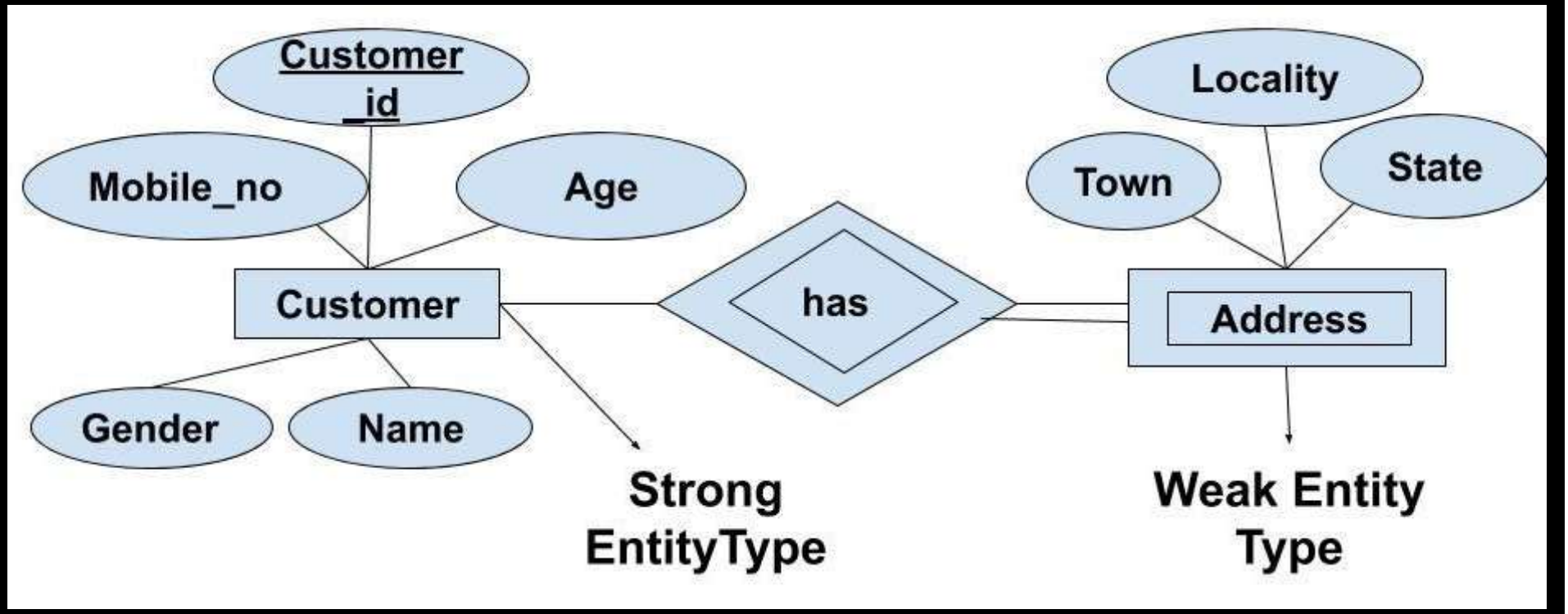


Entity Relationship

Weak Entity Type: Weak entity type doesn't have a key attribute. Weak entity type can't be identified on its own. It depends upon some other strong entity for its distinct identity. This can be understood with a real-life example. There can be children only if the parent exits. There can be no independent existence of children. There can be a room only if building exits. There can be no independent existence of a room. A weak entity is represented by a double outlined rectangle. The relationship between a weak entity type and strong entity type is called an identifying relationship and shown with a double outlined diamond instead of a single outlined diamond. This representation can be seen in the diagram below.

***Example*:** If we have two tables of Customer(Customer_id, Name, Mobile_no, Age, Gender) and Address(Locality, Town, State, Customer_id). Here we cannot identify the address uniquely as there can be many customers from the same locality. So, for this, we need an attribute of Strong Entity Type i.e ‘Customer’ here to uniquely identify entities of 'Address' Entity Type.

Entity Relationship



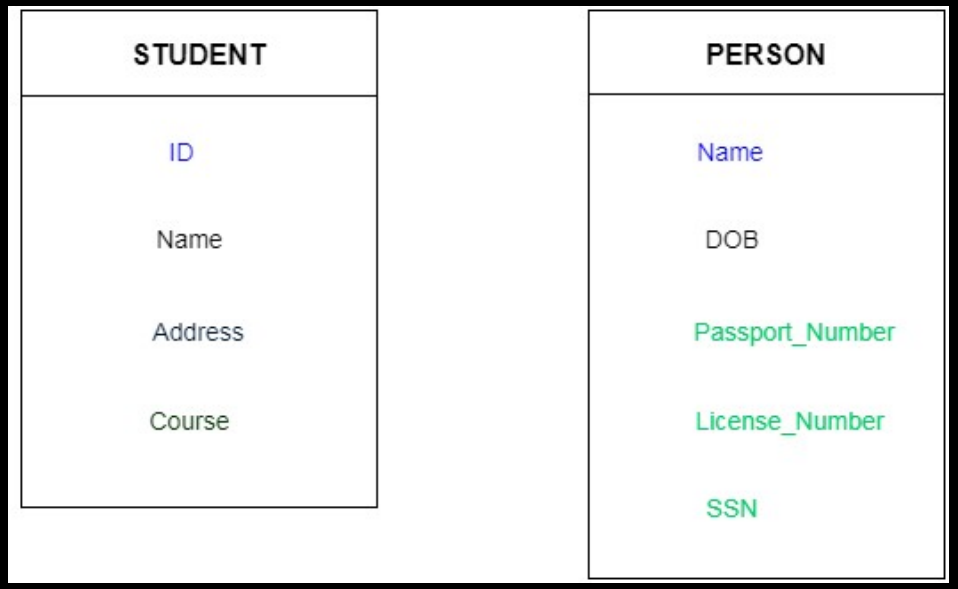
Entity Relationship

Keys

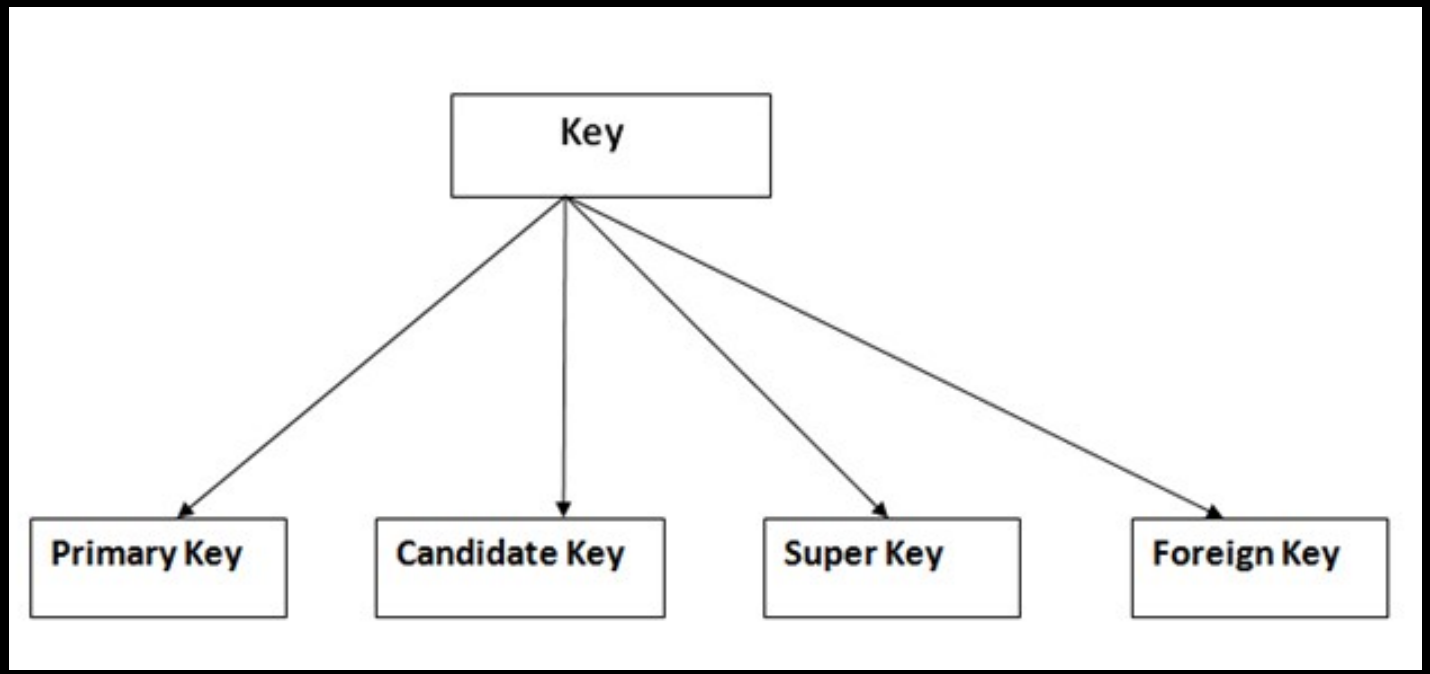
Keys play an important role in the relational database.

It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

For example: In Student table, ID is used as a key because it is unique for each student. In PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.



Entity Relationship



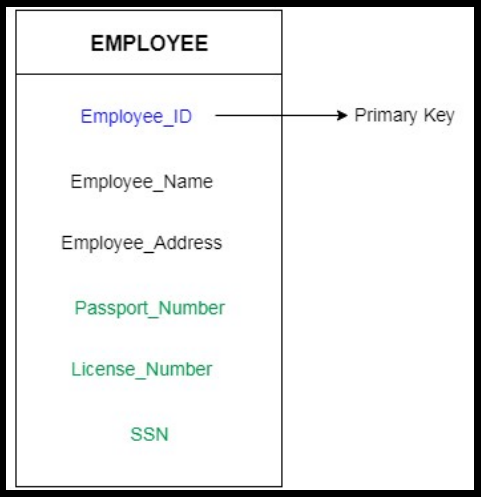
Entity Relationship

Primary key

It is the first key which is used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys as we saw in PERSON table. The key which is most suitable from those lists become a primary key.

In the EMPLOYEE table, ID can be primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary key since they are also unique.

For each entity, selection of the primary key is based on requirement and developers.

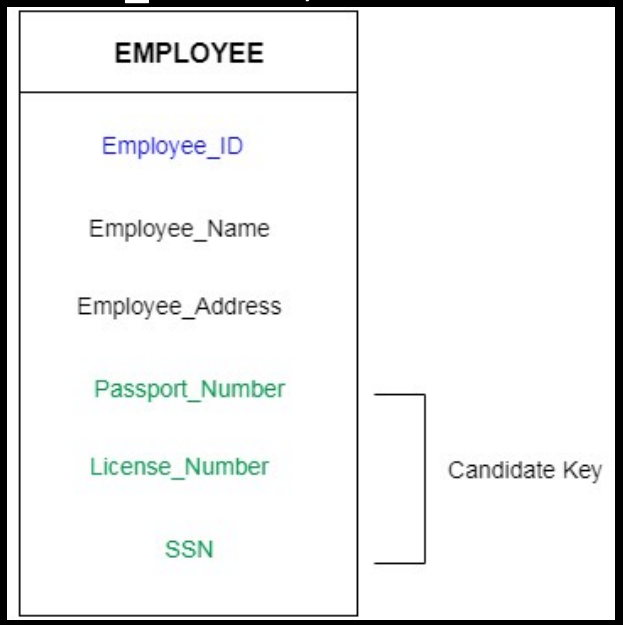


Entity Relationship

Candidate key

A candidate key is an attribute or set of an attribute which can uniquely identify a tuple. The remaining attributes except for primary key are considered as a candidate key. The candidate keys are as strong as the primary key.

For example: In the EMPLOYEE table, id is best suited for the primary key. Rest of the attributes like SSN, Passport_Number, and License_Number, etc. are considered as a candidate key.



Entity Relationship

Super Key

Super key is a set of an attribute which can uniquely identify a tuple. Super key is a superset of a candidate key.

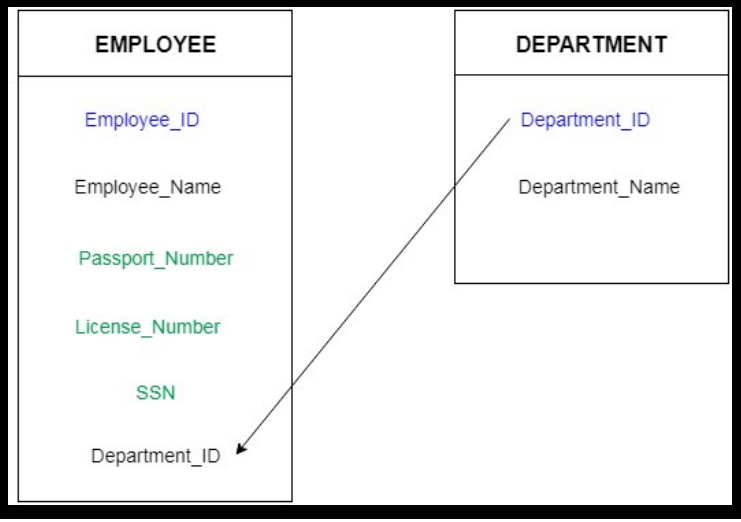
For example: In the above EMPLOYEE table, for(EMPLOEE_ID, EMPLOYEE_NAME) the name of two employees can be the same, but their EMPLYEE_ID can't be the same. Hence, this combination can also be a key.
The super key would be EMPLOYEE-ID, (EMPLOYEE_ID, EMPLOYEE-NAME), etc.

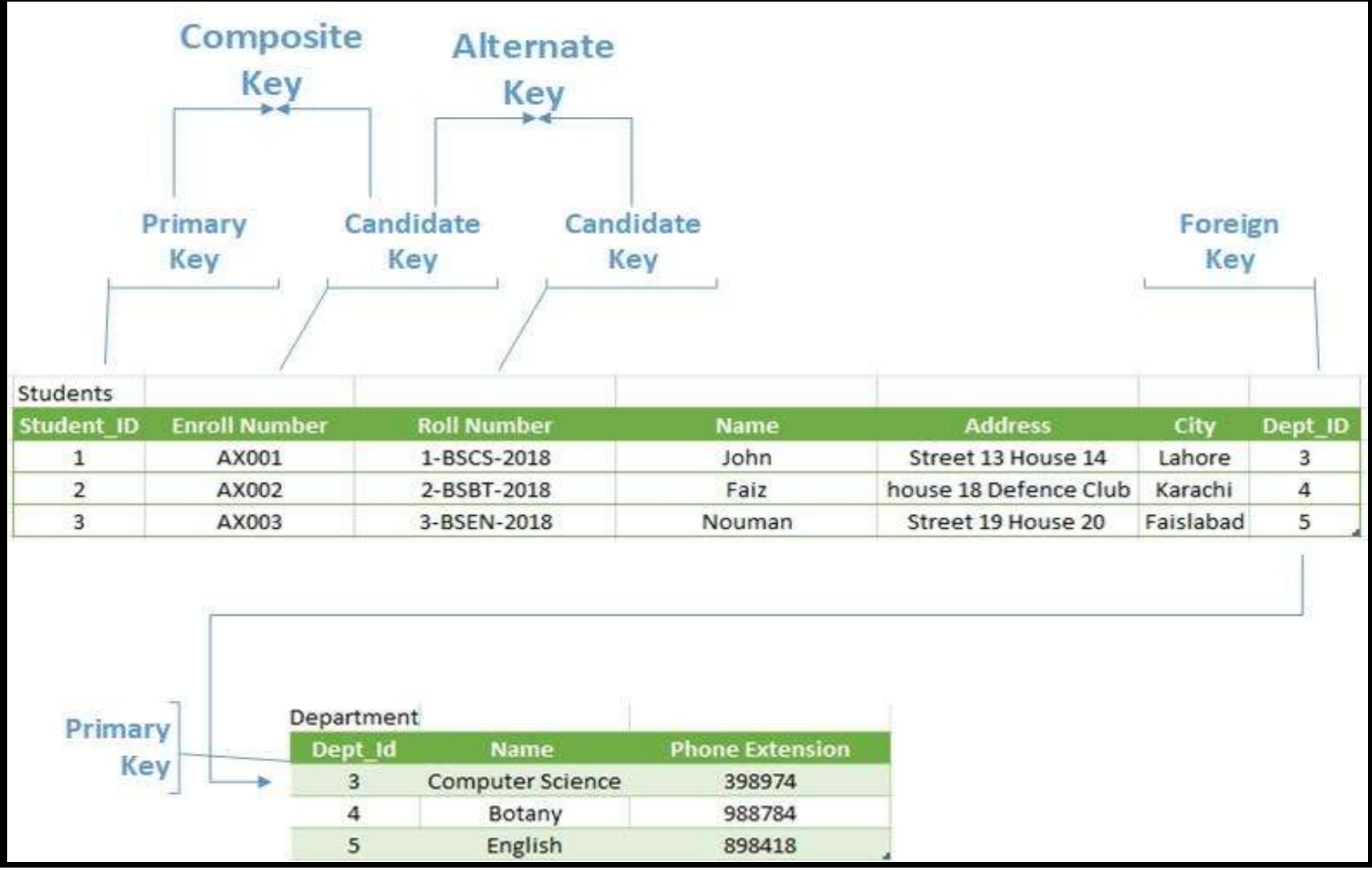
Entity Relationship

Foreign key

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

Now in the EMPLOYEE table, Department_Id is the foreign key, and both the tables are related.





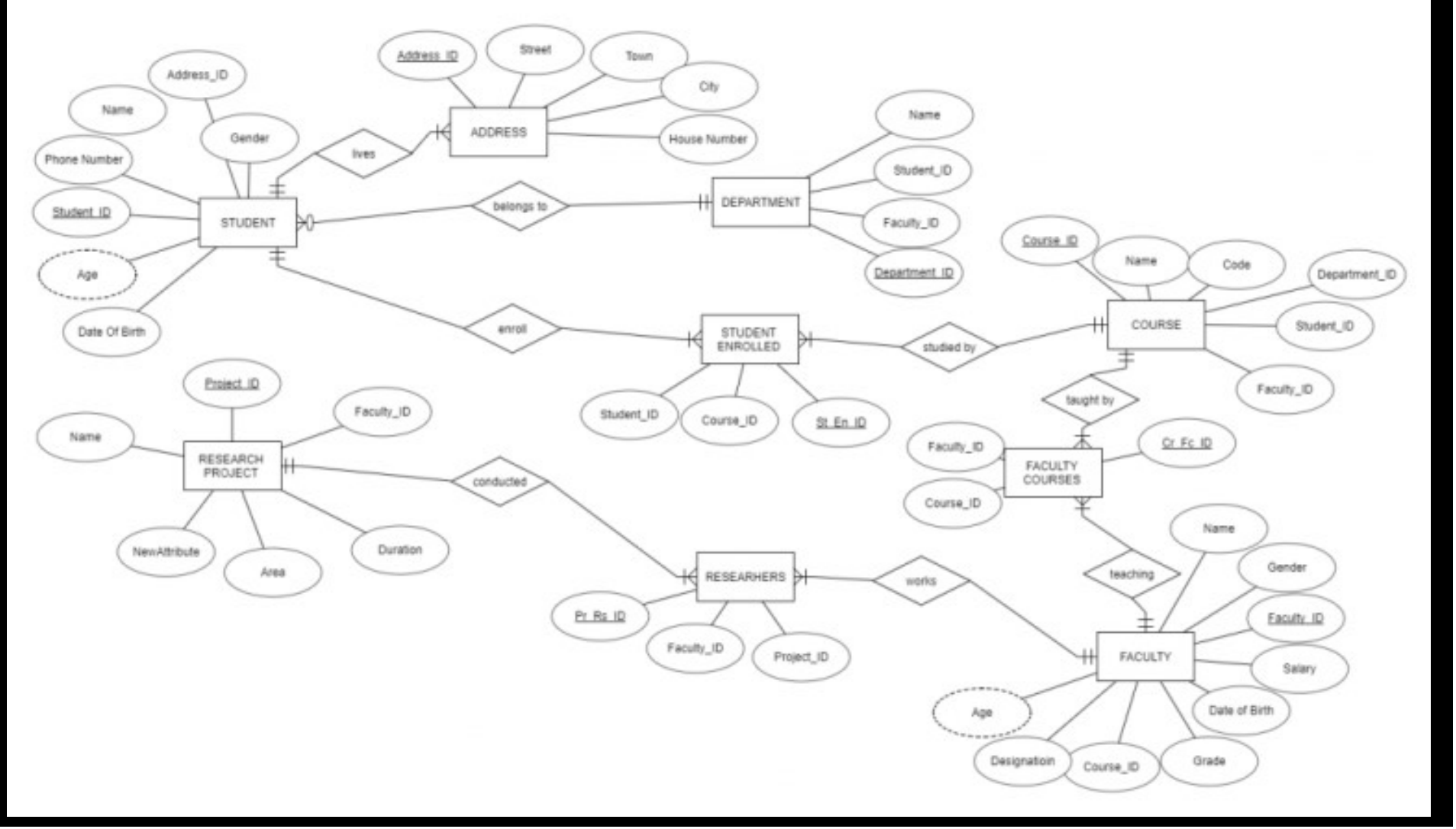
Students

Student_ID	Enroll Number	Roll Number	Name	Address	City	Dept_ID
1	AX001	1-BSCS-2018	John	Street 13 House 14	Lahore	3
2	AX002	2-BSBT-2018	Faiz	house 18 Defence Club	Karachi	4
3	AX003	3-BSEN-2018	Nouman	Street 19 House 20	Faislabad	5

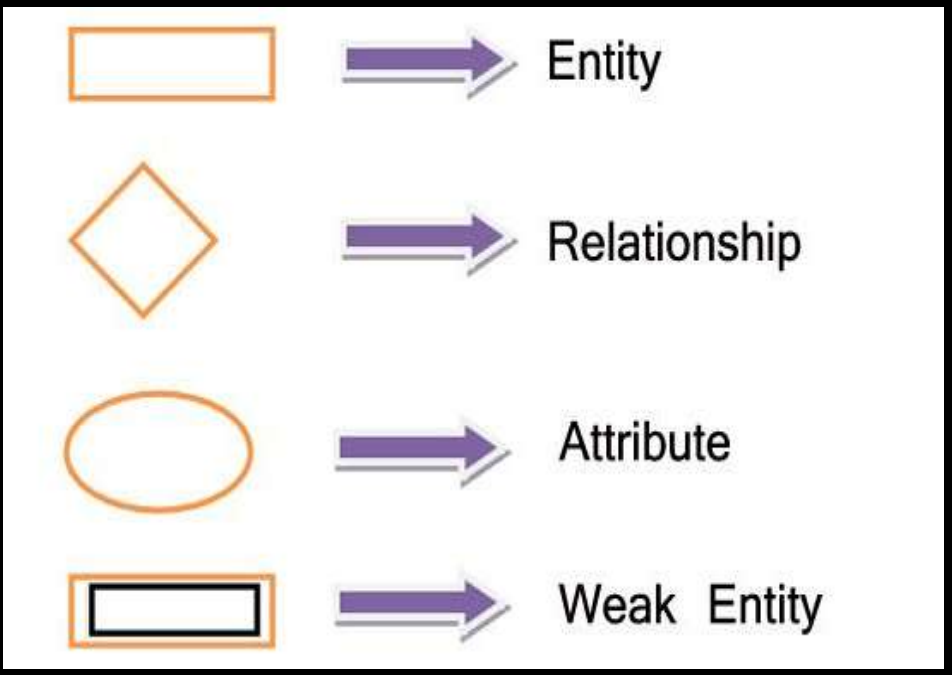
Primary Key

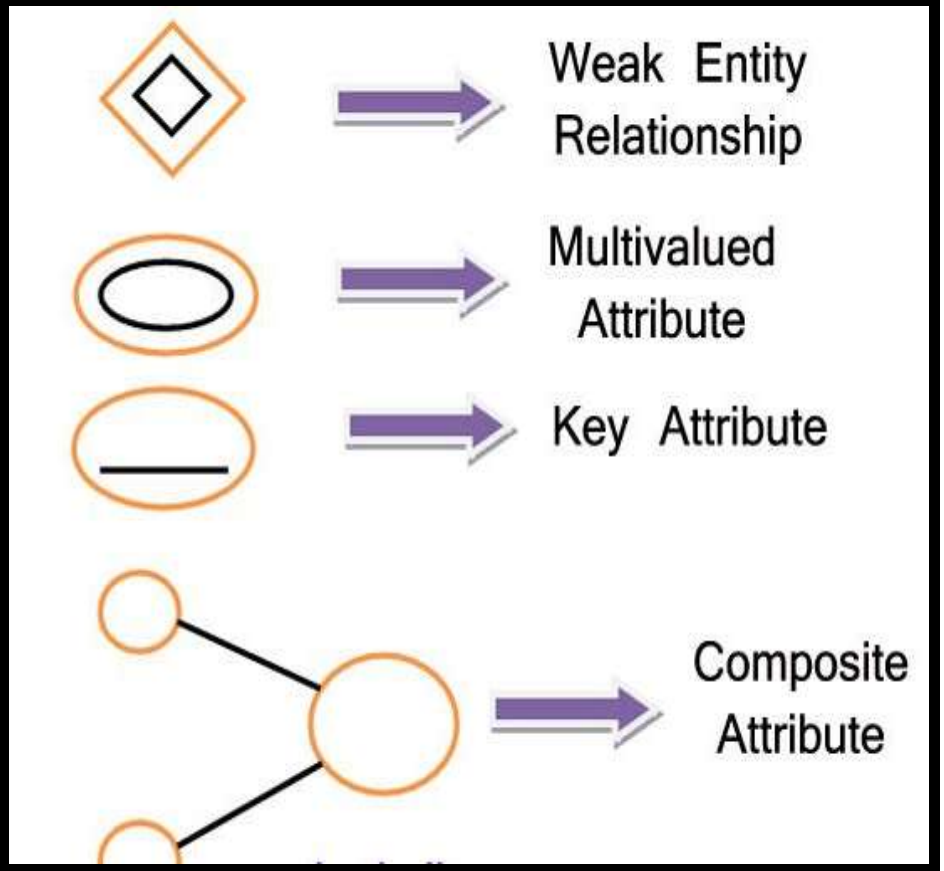
Department

Dept_Id	Name	Phone Extension
3	Computer Science	398974
4	Botany	988784
5	English	898418


Address_ID
Name
Gender
Phone Number
Student_ID
Age
Date Of Birth
STUDENT
Address_ID
Street
Town
City
House Number
ADDRESS
lives
belongs to
DEPARTMENT
Name
Student_ID
Faculty_ID
Department_ID
enroll
STUDENT ENROLLED
Student_ID
Course_ID
St_En_ID
studied by
COURSE
Course_ID
Name
Code
Department_ID
Student_ID
Faculty_ID
taught by
Project_ID
Name
Faculty_ID
RESEARCH PROJECT
NewAttribute
Area
Duration
conducted
RESEARHERS
Pr_Rs_ID
Faculty_ID
Project_ID
works
Faculty_ID
Course_ID
Cr_Fc_ID
FACULTY COURSES
teaching
Name
Gender
Faculty_ID
Salary
Date of Birth
Grade
Course_ID
Designation
Age
FACULTY

Entity Relationship (Symbol used)



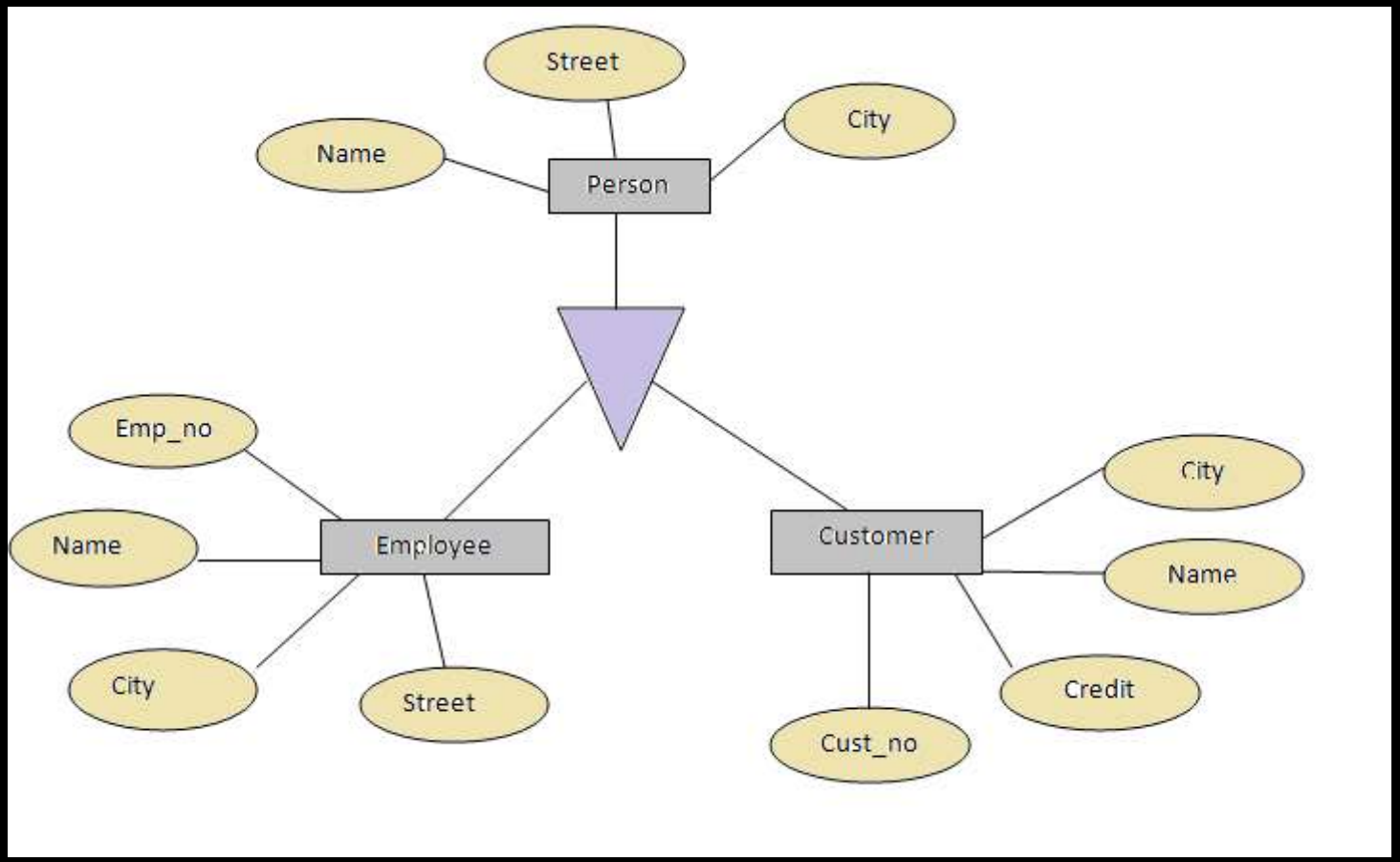


ER Modelling (Generalization)
Generalization, Specialization and Aggregation

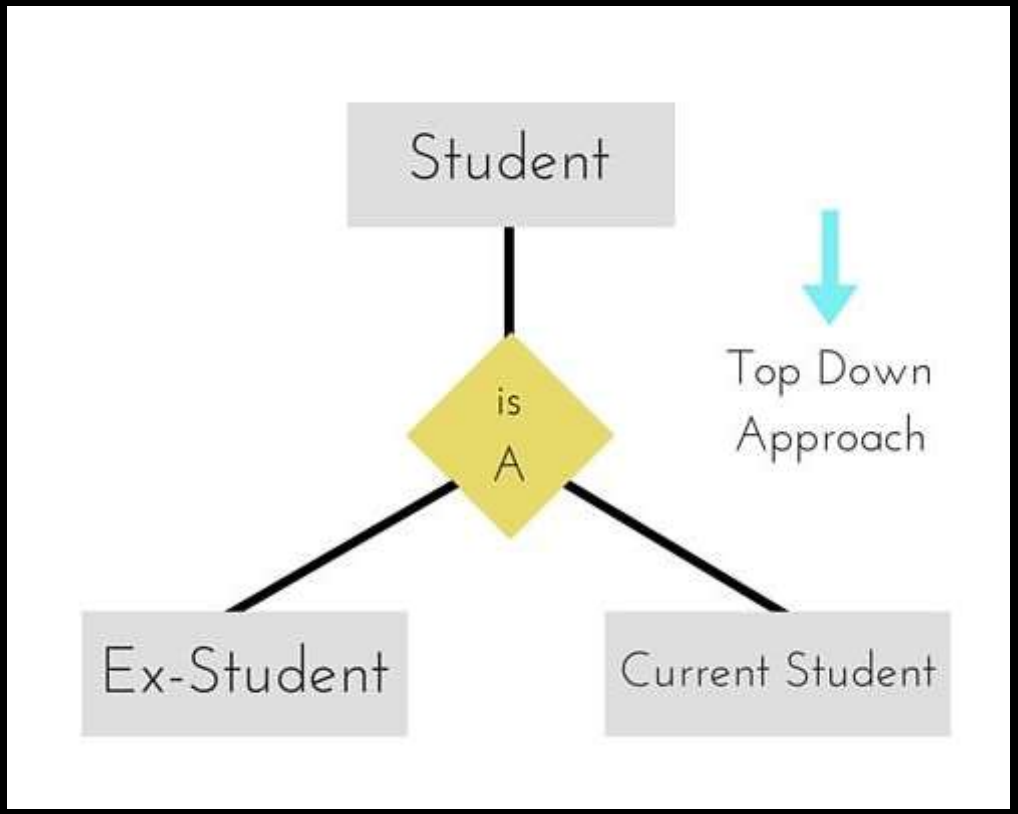
In ER model are used for data abstraction in which abstraction mechanism is used to hide details of a set of objects.

Generalization – Generalization is the process of extracting common properties from a set of entities and create a generalized entity from it

Entity Relationship (Generalization)



Entity Relationship (Generalization)



Entity Relationship (Generalization)

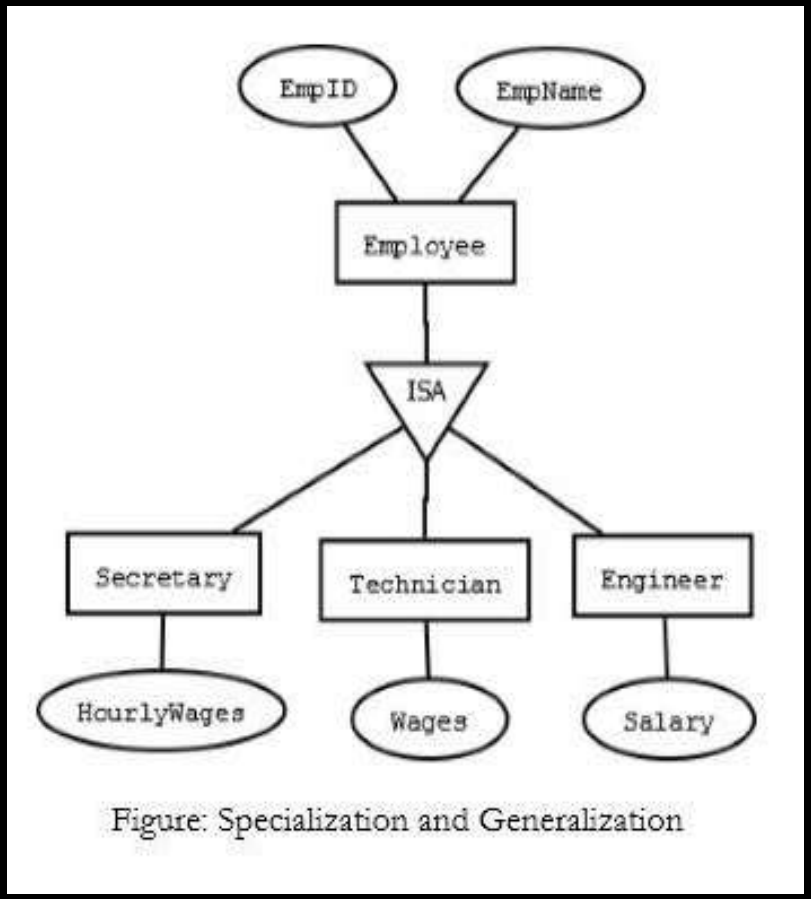


Figure: Specialization and Generalization

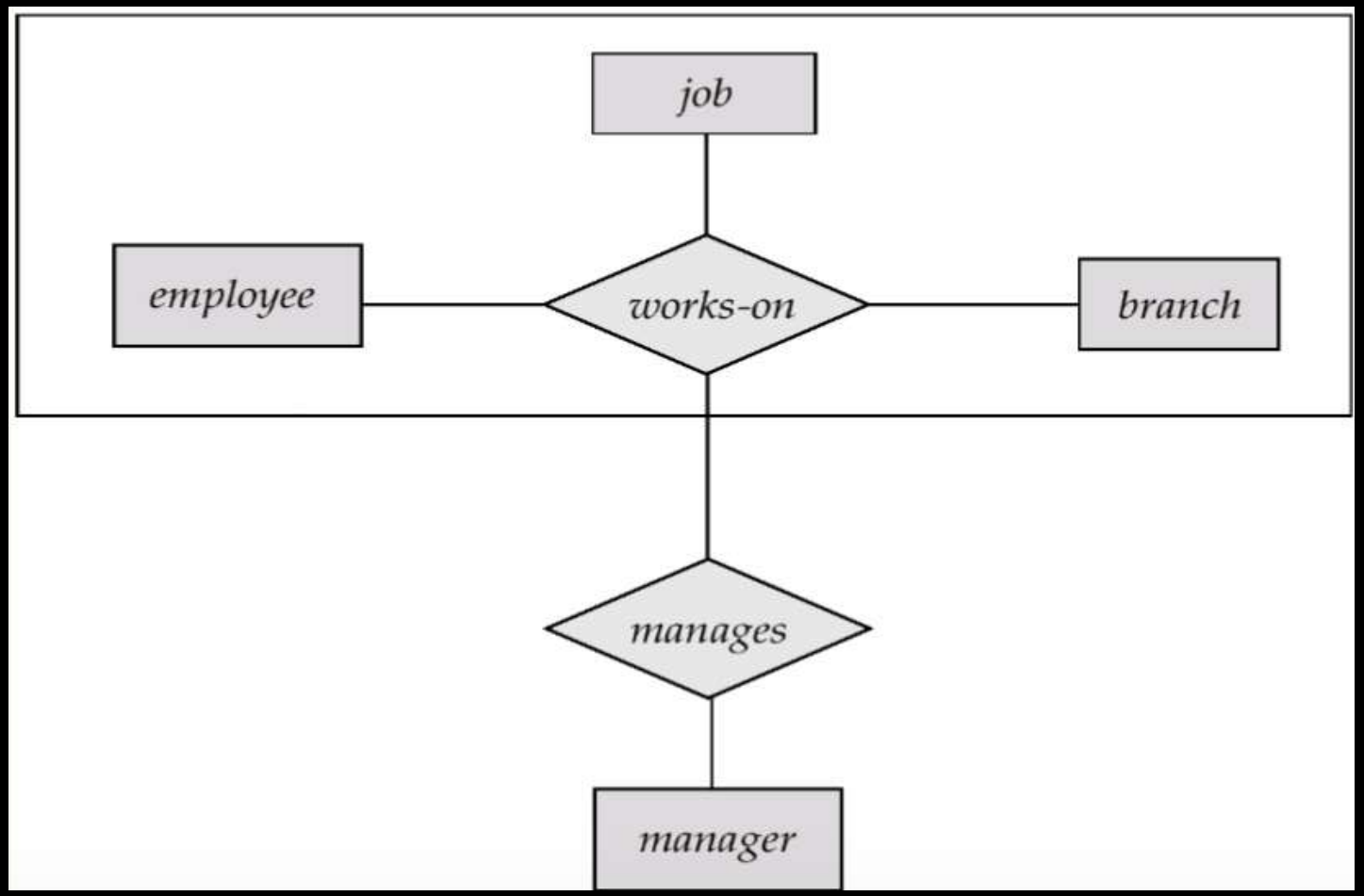
ER Modelling (Specialization)

Specialization is a process of identifying subsets of an entity that shares different characteristics. It breaks an entity into multiple entities from higher level (super class) to lower level (sub class)

ER Modelling (Aggregation)

An **ER diagram** is not capable of representing relationship between an entity and a relationship which may be required in some scenarios. In those cases, a relationship with its corresponding entities is aggregated into a higher level entity

ER Modelling (Aggregation)



ER Modelling (Aggregation)

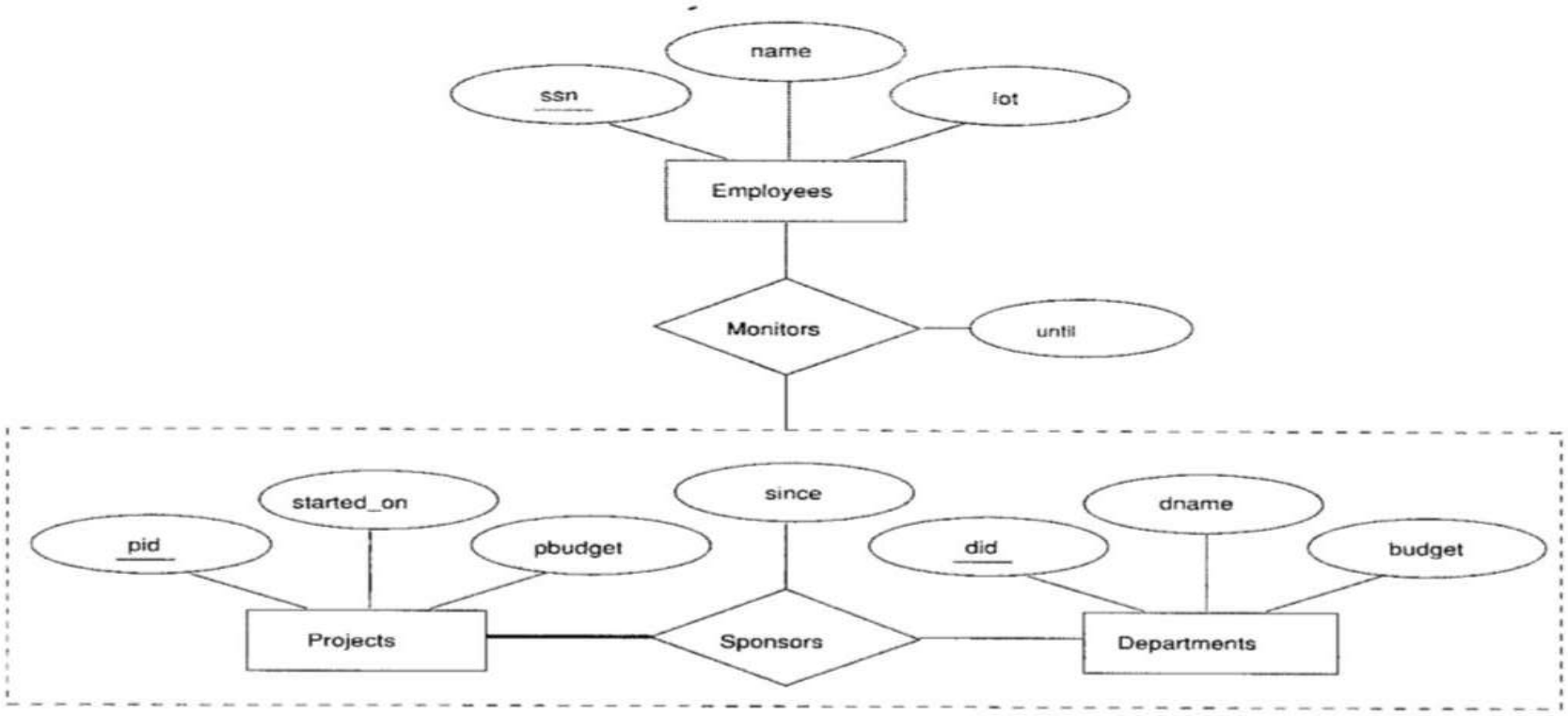
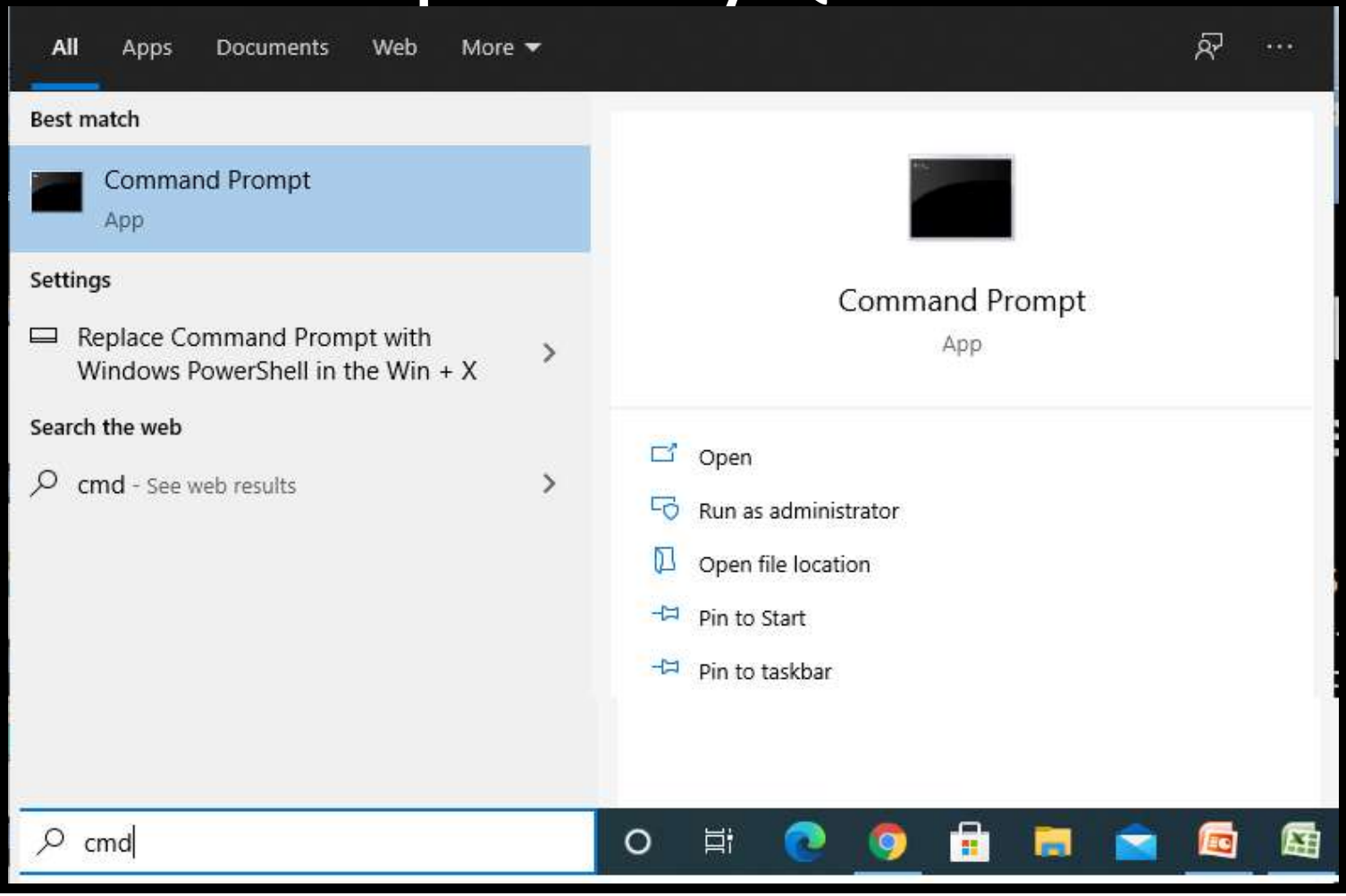


Figure 2.13 Aggregation

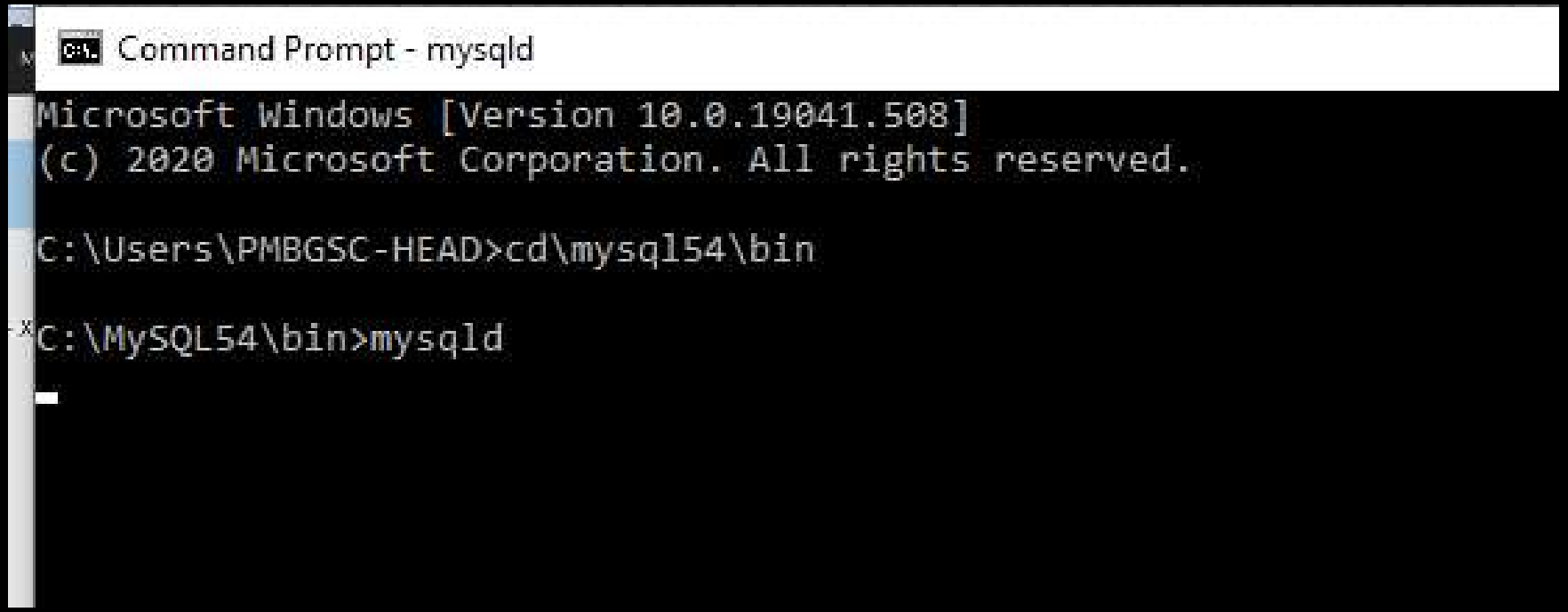
How to perform MySQL Practicals

- Demand for access
- Download folder by Right click and choosing download from it
- Assuming you have copied shared folder MySQL54
- in C:\ Drive

How to perform MySQL Practicals



How to perform MySQL Practicals



How to perform MySQL Practicals

MySQL DATA TYPES

DATE TYPE	SPEC	DATA TYPE	SPEC
CHAR	String (0 - 255)	INT	Integer (-2147483648 to 214748-3647)
VARCHAR	String (0 - 255)	BIGINT	Integer (-9223372036854775808 to 9223372036854775807)
TINYTEXT	String (0 - 255)	FLOAT	Decimal (precise to 23 digits)
TEXT	String (0 - 65535)	DOUBLE	Decimal (24 to 53 digits)
BLOB	String (0 - 65535)	DECIMAL	"DOUBLE" stored as string
MEDIUMTEXT	String (0 - 16777215)	DATE	YYYY-MM-DD
MEDIUMBLOB	String (0 - 16777215)	DATETIME	YYYY-MM-DD HH:MM:SS
LONGTEXT	String (0 - 4294967295)	TIMESTAMP	YYYYMMDDHHMMSS
LONGBLOB	String (0 - 4294967295)	TIME	HH:MM:SS
TINYINT	Integer (-128 to 127)	ENUM	One of preset options
SMALLINT	Integer (-32768 to 32767)	SET	Selection of preset options
MEDIUMINT	Integer (-8388608 to 8388607)	BOOLEAN	TINYINT(1)

How to perform MySQL Practicals

```
create table user(
    id Integer auto_increment,
    userName varchar(32) unique not null,
    password varchar(32) not null,
    realName varchar(32),
    primary key (id)
);
```

How to perform MySQL Practicals

```
ALTER TABLE employee ADD city CHAR(20);

DESCRIBE employee;

ALTER TABLE employee MODIFY name CHAR(30) NOT NULL;
```

```
ALTER TABLE new_table

DROP COLUMN column3 ;

Select * from new_table;
```

How to perform MySQL Practicals

Adding Primary Key

create table dept_tab (deptname varchar(20), deptno integer primary key) ENGINE=InnoDB;

Adding Foreign Key

create table emp_tab (empname varchar(40), empno integer, deptno integer not null, foreign key (deptno) references dept_tab(deptno)) ENGINE=InnoDB;

@FOREIGN_KEY_CHECKS is set to 1.
SELECT @@FOREIGN_KEY_CHECKS
SET FOREIGN_KEY_CHECKS=1

How to perform MySQL Practicals

1. Explain DDL, DML, DCL and TCL statements.
2. WAQ to create a table for department details having following fields. DID, DName, Description where DID would be primary key.

3. WAQ to create a table for employee data handling having following fields.

EID, DID, EName, Address, EStreet, ECity, EDOB, ContactNo, Salary, Designation where EID would be primary key and DID would be foreign key.

LOAD DATA INFILE 'cslist.csv' INTO TABLE emp fields terminated by ',' enclosed by '"' lines terminated by '\n' ignore 1 lines;

How to perform MySQL Practicals

1. ALTER TABLE mytbl ALTER j SET DEFAULT 1000;

2. ALTER TABLE mytbl ALTER j DROP DEFAULT;

3. ALTER TABLE table_name CHANGE old_column_name new_column_name <column definition>;

4. ALTER TABLE "table_name" DROP "column_name“

How to perform MySQL Practicals

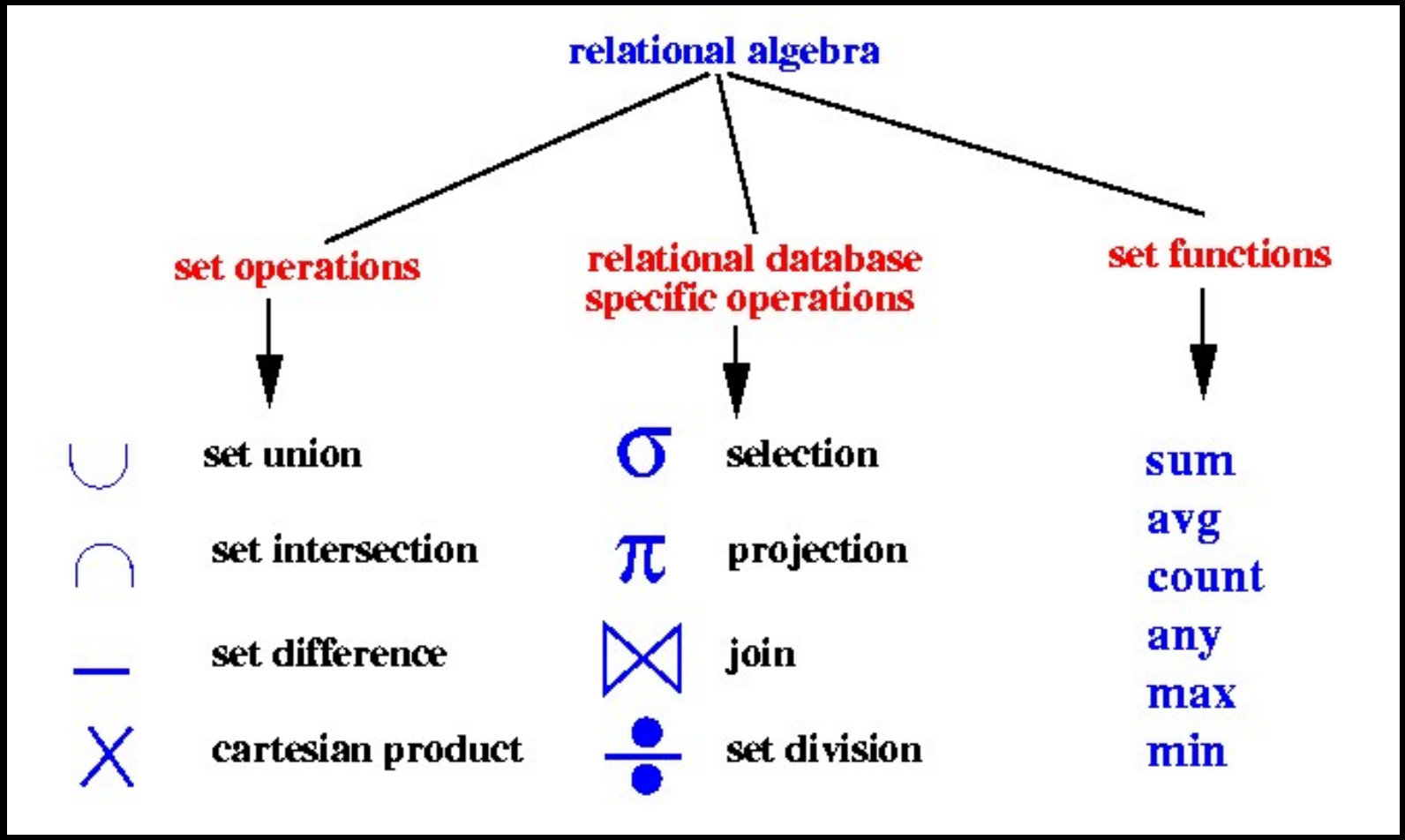
Changing column name

```
alter table emp change designation edesig varchar(20);
```

Relational Algebra

In database theory, relational algebra is a theory that uses algebraic structures with a well-founded semantics for modeling the data, and defining queries on it. The theory has been introduced by Edgar F. Codd.

Relational Algebra



Relational Algebra

Notation

Operation	My HTML	Symbol
Projection	PROJECT	π
Selection	SELECT	σ
Renaming	RENAME	ρ
Union	UNION	$\cup$
Intersection	INTERSECTION	$\cap$
Assignment	<-	$\leftarrow$

Operation	My HTML	Symbol
Cartesian product	X	$\times$
Join	JOIN	$\bowtie$
Left outer join	LEFT OUTER JOIN	$⟕$
Right outer join	RIGHT OUTER JOIN	$⟖$
Full outer join	FULL OUTER JOIN	$⟗$
Semijoin	SEMIJOIN	$\ltimes$

Relational Algebra – PROJECTION & SELECTION

A project operation selects only certain columns (fields) from a table. The result table has a subset of the available columns and can include anything from a single column to all available columns.

SELECT ENAME, SALARY FROM EMP;

SELECTION:

The ability to select rows from out of complete result set is called Selection.

SELECT * FROM EMP WHERE SALARY >= 50000

Relational Algebra - UNION

Table1	
column1	column2
a	b
a	c
a	d

∪

Table 2	
column1	column2
b	c
a	d

=

Table1 Union Table2	
column1	column2
a	b
a	c
a	d
b	c

Duplicate row not repeated in results

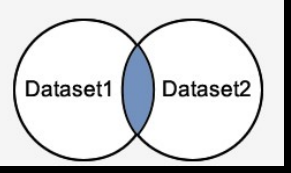
Relational Algebra - UNION

The MySQL UNION operator is used to combine the result sets of 2 or more SELECT Statements. It removes duplicate rows between the various SELECT statements.

Each SELECT statement within the UNION operator must have the same number of fields in the result sets with similar data types.

```
SELECT supplier_id FROM suppliers UNION SELECT supplier_id FROM order_details;
```

```
SELECT supplier_id, supplier_name FROM suppliers WHERE supplier_id <= 500
UNION
SELECT company_id, company_name
FROM companies WHERE company_name = 'Apple' ORDER BY 2;
```



Relational Algebra - INTERSECTION

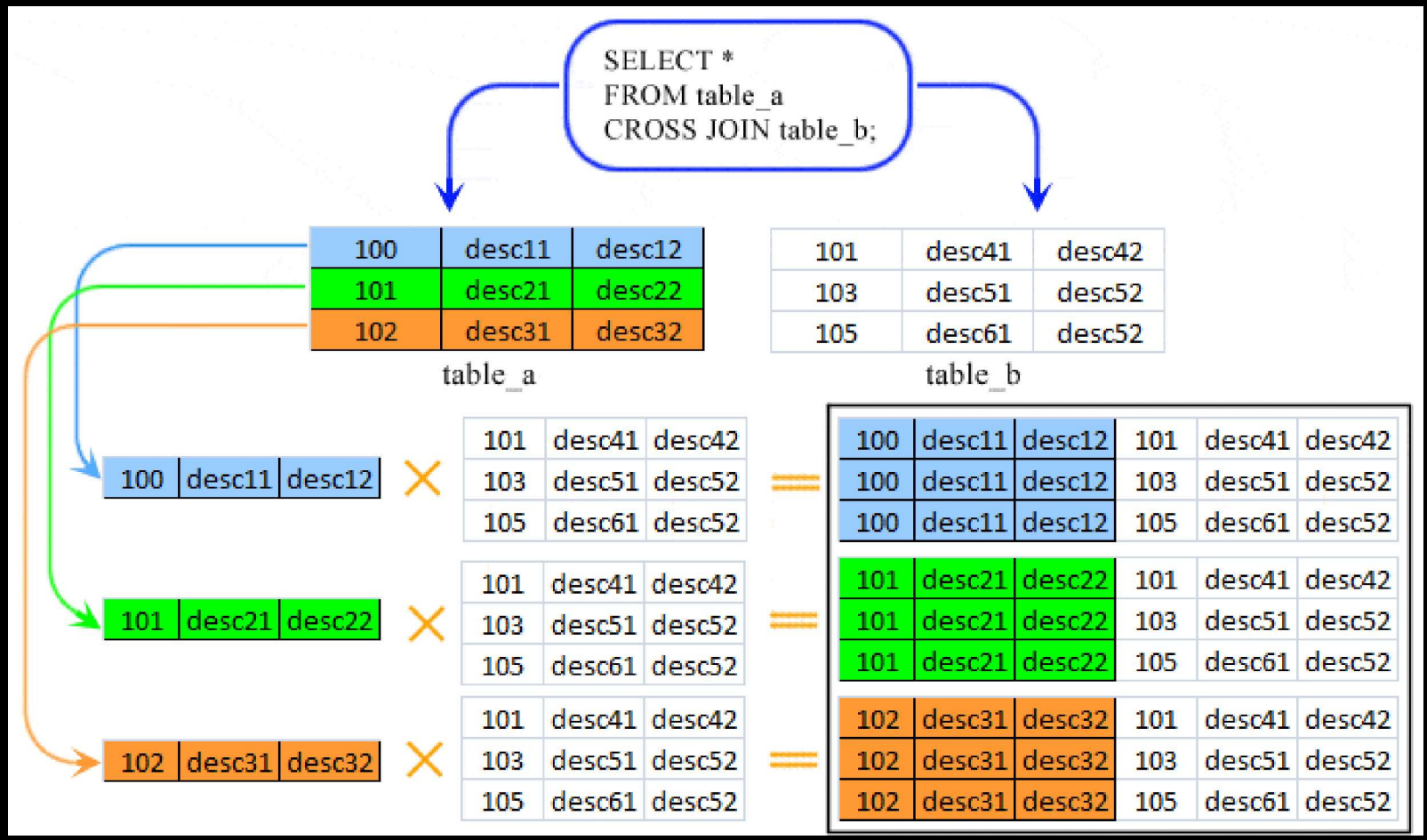
The INTERSECT query will return the records in the blue shaded area. These are the records that exist in both Dataset1 and Dataset2.

```
SELECT category_id FROM products INTERSECT SELECT category_id FROM inventory;
```

Since you can't use the INTERSECT operator in MySQL, you will use the IN operator to simulate the INTERSECT query as follows:

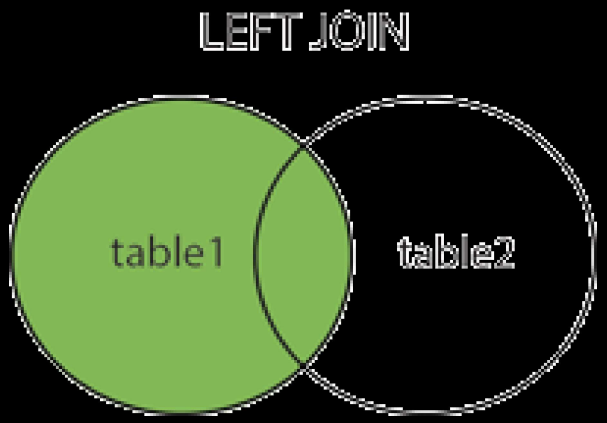
```
SELECT products.category_id FROM products WHERE products.category_id
IN
(SELECT inventory.category_id FROM inventory);
```

Relational Algebra – CARTESIAN PRODUCT



Relational Algebra – LEFT OUTHER JOIN

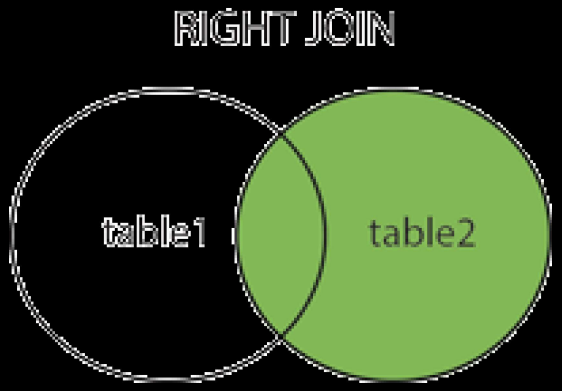
The LEFT JOIN keyword returns all records from the left table (table1), and the matched records from the right table (table2). The result is NULL from the right side, if there is no match.




```
SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID
ORDER BY Customers.CustomerName;
```

Relational Algebra – RIGHT OUTHER JOIN

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table. The result is NULL from the left side, when there is no match.




```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID =
Employees.EmployeeID
ORDER BY Orders.OrderID;
```

Relational Algebra – SELF JOIN

A self JOIN is a regular join, but the table is joined with itself.

The following SQL statement matches customers that are from the same city:

```
SELECT A.CustomerName AS CustomerName1,
B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

Database Management System

B.Sc. CS, Paper - I

Unit – IV

Functional Dependencies

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.

Functional Dependency

A -> B

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

Functional Dependencies

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**.

Functional Dependencies

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

Trivial Functional Dependency

Non-Trivial Functional Dependency

Completely Non-Trivial Functional Dependency

Functional Dependencies

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

Functional Dependencies

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 –

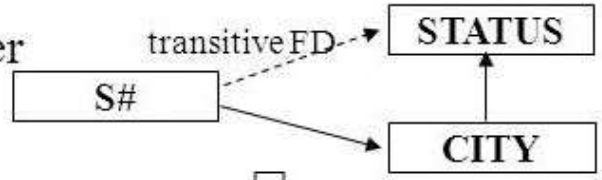
A ->B

Good and Bad decomposition in dbms

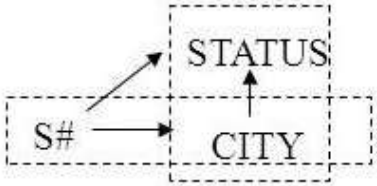
In a database, it breaks the table into multiple tables. If the relation has no proper **decomposition**, then it may lead to problems like loss of information. **Decomposition** is used to eliminate some of the problems of **bad** design like anomalies, inconsistencies, and redundancy.

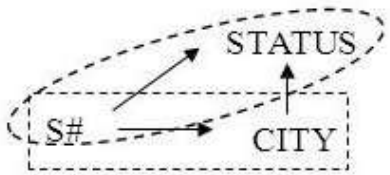
Good and Bad decomposition in dbms

- Consider



Suppose 1. CITY is the main office of the supplier.
2. STATUS is some factor of CITY (ref. p.7-9)





① Decomposition **A:**

SC: S# → CITY

CS: CITY → STATUS

Good !

(Rome, 50) can be inserted.

② Decomposition **B:**

SC: S# → CITY

CS: S# ⇢ STATUS

Bad !

(Rome, 50) can not be inserted unless there is a supplier located at Rome.

③ Decomposition **C:**

S# -> status

city -> status

'Good' or 'Bad' is dependent on item's semantic meaning!!

A consequence of bad design in dbms

A badly designed database has the following problems: Related data is scattered over various tables. ... It's possible that the information is only half present, it's there in one table, but missing in another one. Data is inconsistent or ambiguous (poly interpretable).

A consequence of bad design in dbms

- Related data is scattered over various tables. A change must be updated at many places.
- It's possible that the information is only half present, it's there in one table, but missing in another one.
- Data is inconsistent or ambiguous (poly interpretable).
- The database is unnessessary complex, hacked with lots of tricks. The database designer done unnceccessary complex things, where it could have been done so much easier.
- The database has 'hidden' information, for example by the **sequence** of rows in a table.
- The database is slow, inflexible, hard to extend and can not handle all real life situations.

Normalization

Normalization is the process of minimizing redundancy from a relation or set of relations. Redundancy in relation may cause insertion, deletion and updation anomalies. So, it helps to minimize the redundancy in relations. Normal forms are used to eliminate or reduce redundancy in **database** tables.

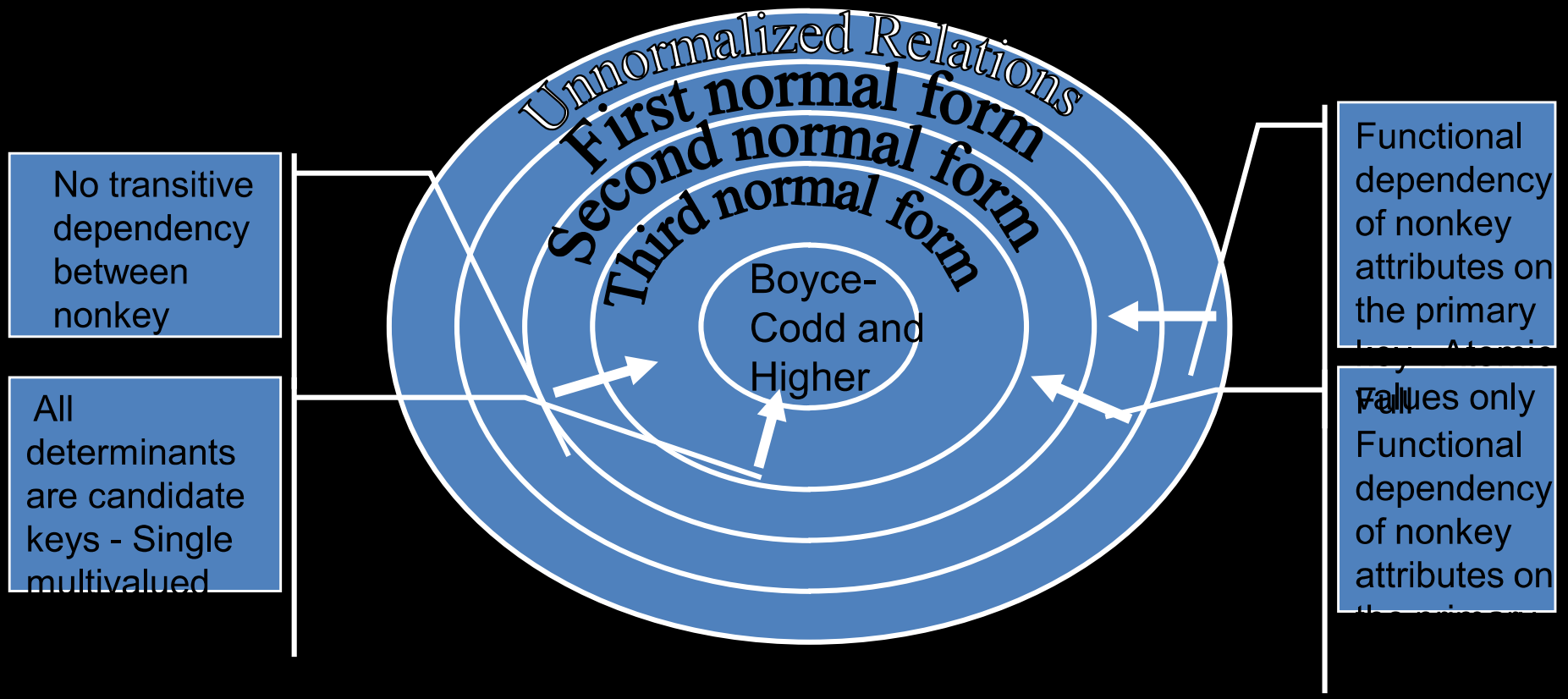
Normalization

- It is a technique for designing relational database tables to minimize duplication of information.
- Normalization is a practice to safeguard the database against logical and structural anomalies.
- Normalization is also termed as *canonical synthesis* by the experts.
- It is used to keep data consistent and check that no loss of data as well as data integrity is there.
- Its complexity may lead to higher degree of join operations which sometimes lead to the degraded throughput times.
- The normal forms like 1NF, 2NF, 3NF, BCNF, 4NF, 5NF, DKNF & 6NF are in practice.

Normal Forms

- First Normal Form (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Boyce-Codd Normal Form (BCNF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)

Normalization



Functional Dependencies

- Functional dependencies (FDs) are used to specify *formal measures* of the "goodness" of relational designs
- FDs and keys are used to define **normal forms** for relations
- FDs are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes

Functional Dependency definition

- A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y
- X →Y holds if whenever two tuples have the same value for X, they *must have* the same value for Y

  If t1[X]=t2[X], *then* t1[Y]=t2[Y] in any relation instance r(R)
- X → Y in R specifies a *constraint* on all relation instances r(R)
- FDs are derived from the real-world constraints on the attributes

Examples of FD constraints

- Social Security Number determines employee name
  SSN → ENAME
- Project Number determines project name and location
  PNUMBER → {PNAME, PLOCATION}
- Employee SSN and project number determines the hours per week that the employee works on the project
  {SSN, PNUMBER} → HOURS

Functional Dependencies and Keys

- An FD is a property of the attributes in the schema R
- The constraint must hold on *every relation instance* r(R)
- If K is a key of R, then K functionally determines all attributes in R (since we never have two distinct tuples with t1[K]=t2[K])

Inference Rules for FDs

- Given a set of FDs F, we can *infer* additional FDs that hold whenever the FDs in F hold
- Armstrong's inference rules
  - A1. (Reflexive) If Y subset-of X, then X → Y
  - A2. (Augmentation) If X → Y, then XZ → YZ
    (Notation: XZ stands for X ∪ Z)
  - A3. (Transitive) If X → Y and Y → Z, then X → Z
- A1, A2, A3 form a *sound* and *complete* set of inference rules

Additional Useful Inference Rules

- Decomposition
  - If X → YZ, then X → Y and X → Z
- Union
  - If X → Y and X → Z, then X → YZ
- Psuedotransitivity
  - If X → Y and WY → Z, then WX → Z
- **Closure** of a set F of FDs is the set F+ of all FDs that can be inferred from F

Introduction to Normalization

- **Normalization**: Process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations
- **Normal form**: Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form
  - 2NF, 3NF, BCNF based on keys and FDs of a relation schema
  - 4NF based on keys, multi-valued dependencies

Unnormalized Relations

- First step in normalization is to convert the data into a two-dimensional table
- In unnormalized relations data can repeat within a column

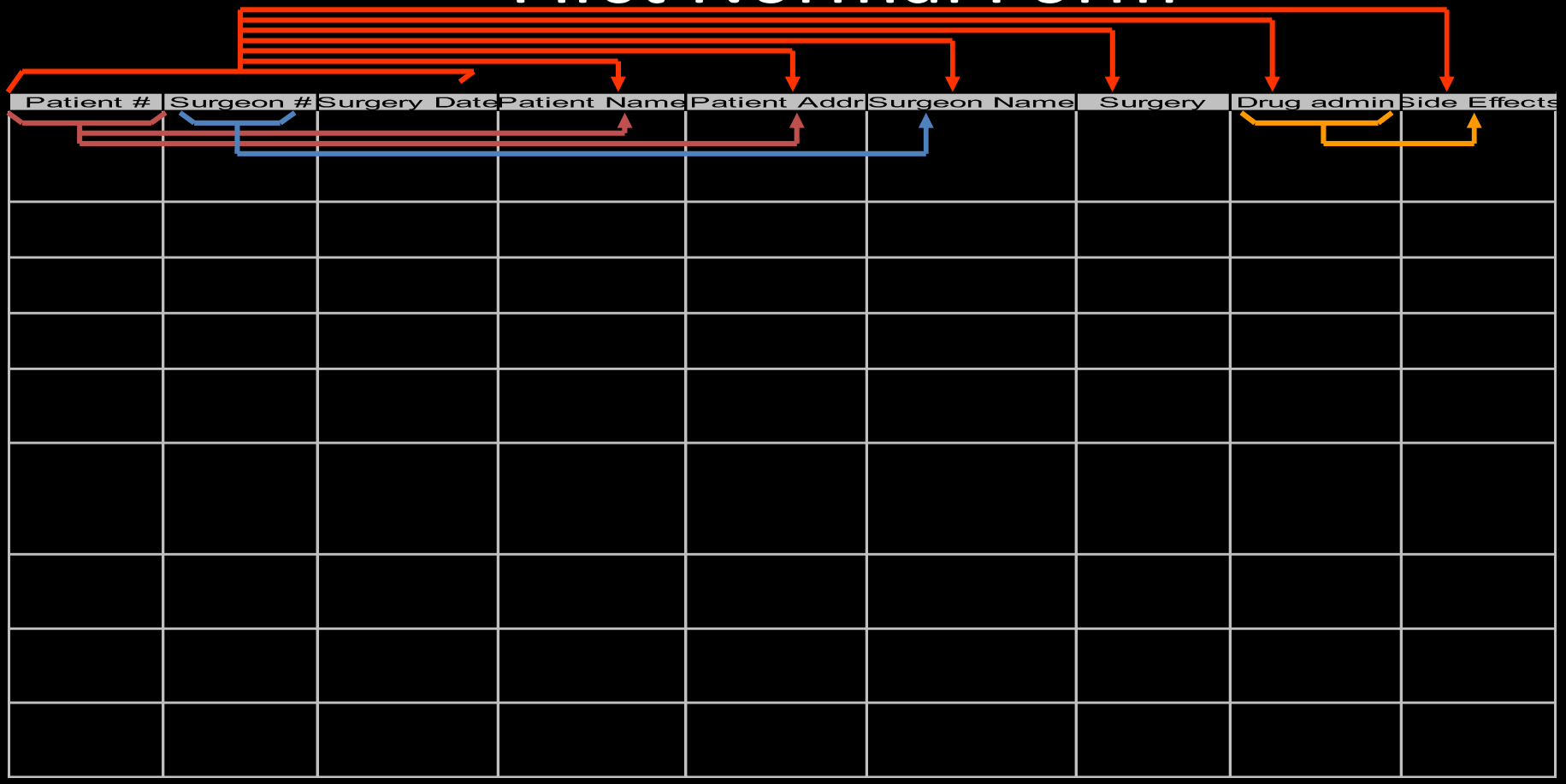
Unnormalized Relation

Patient #	Surgeon #	Surg. date	Patient Name	Patient Addr	Surgeon	Surgery	Postop drug	ug side effec

First Normal Form

- To move to First Normal Form a relation must contain only atomic values at each row and column.
  - No repeating groups
  - A column or set of columns is called a Candidate Key when its values can uniquely identify the row in the relation.

First Normal Form



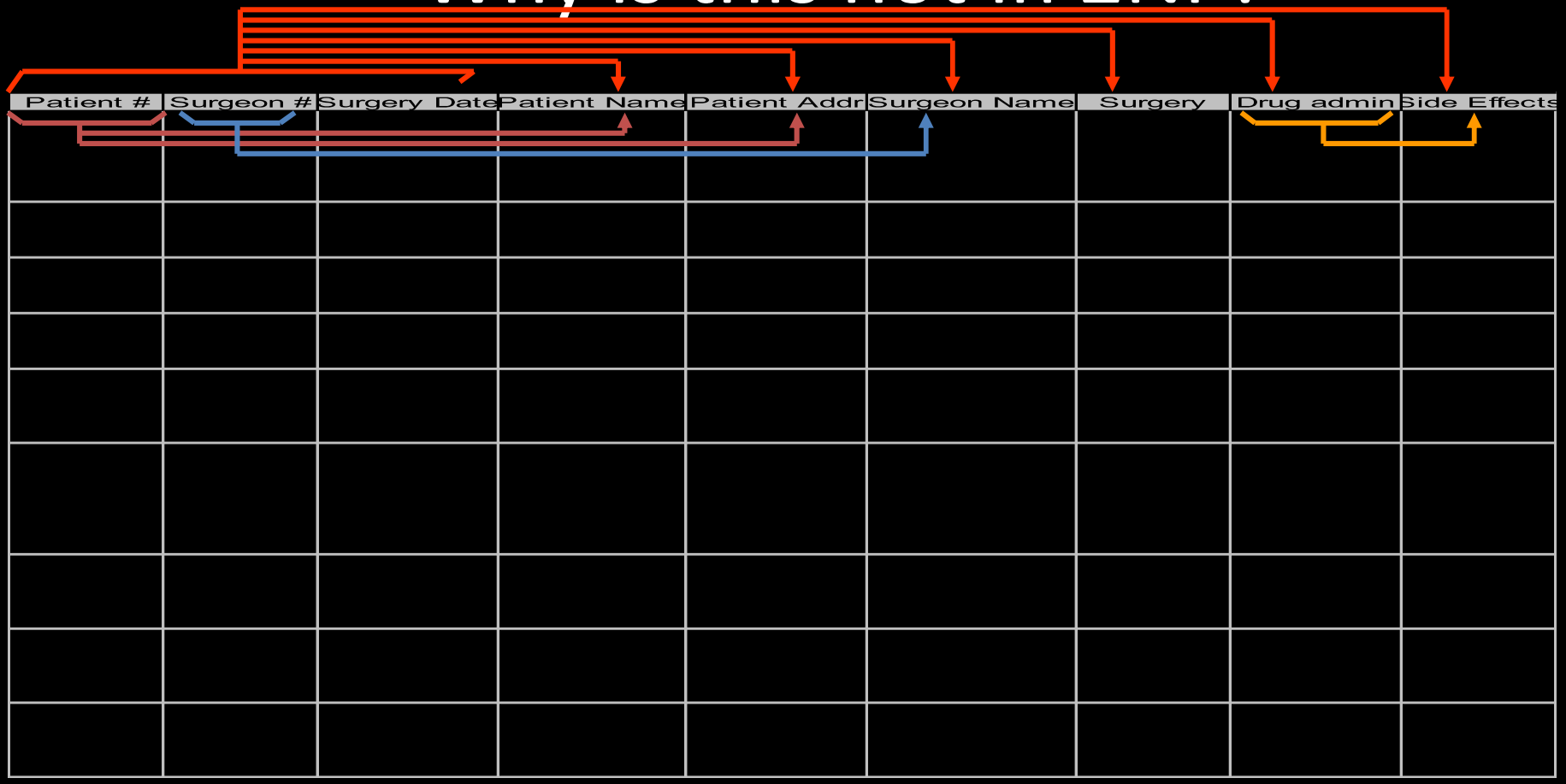
1NF Storage Anomalies

- Insertion: A new patient has not yet undergone surgery -- hence no surgeon # -- Since surgeon # is part of the key, we cannot insert.
- Insertion: If a surgeon is newly hired and has not operated yet -- there will be no way to include that person in the database.
- Update: If a patient comes in for a new procedure, and has moved, we need to change multiple address entries.
- Deletion (type 1): Deleting a patient record may also delete all info about a surgeon.
- Deletion (type 2): When there are functional dependencies (like side effects and drug) changing one item eliminates other information.

Second Normal Form

- A relation is said to be in Second Normal Form when every non-key attribute is **fully functionally dependent** on the primary key.
  - That is, every non-key attribute needs the full primary key for unique identification

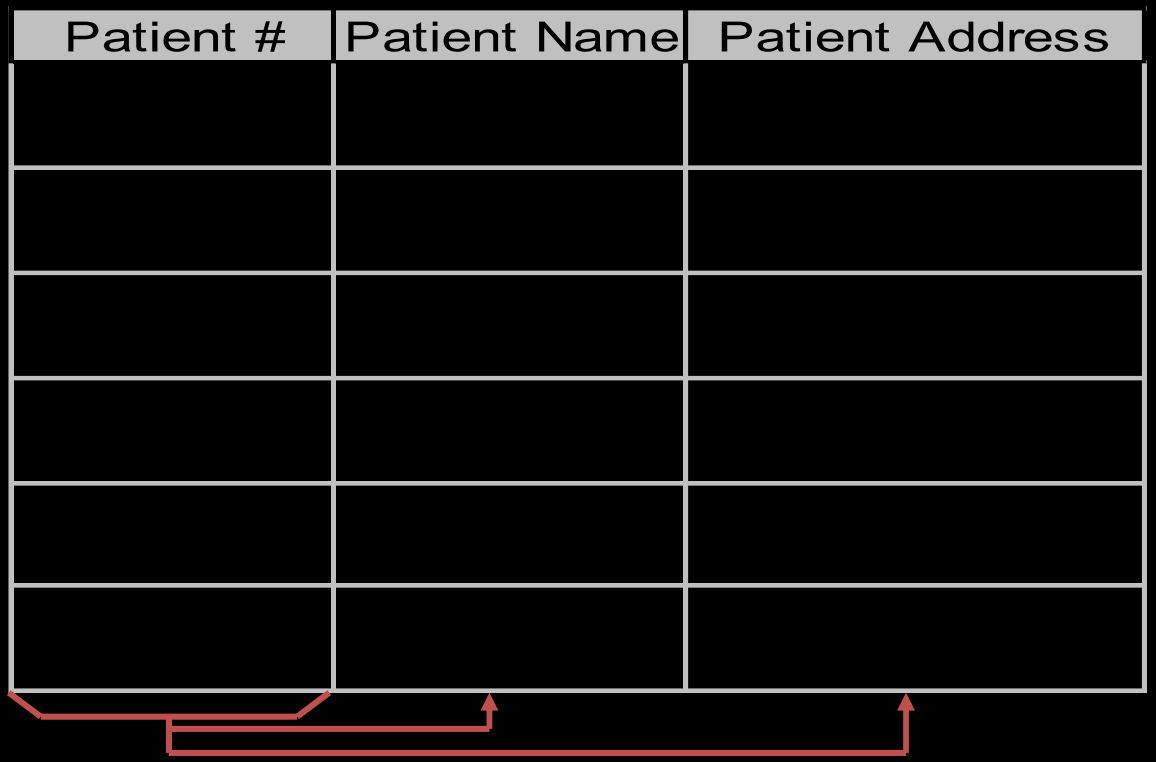
Why is this not in 2NF?



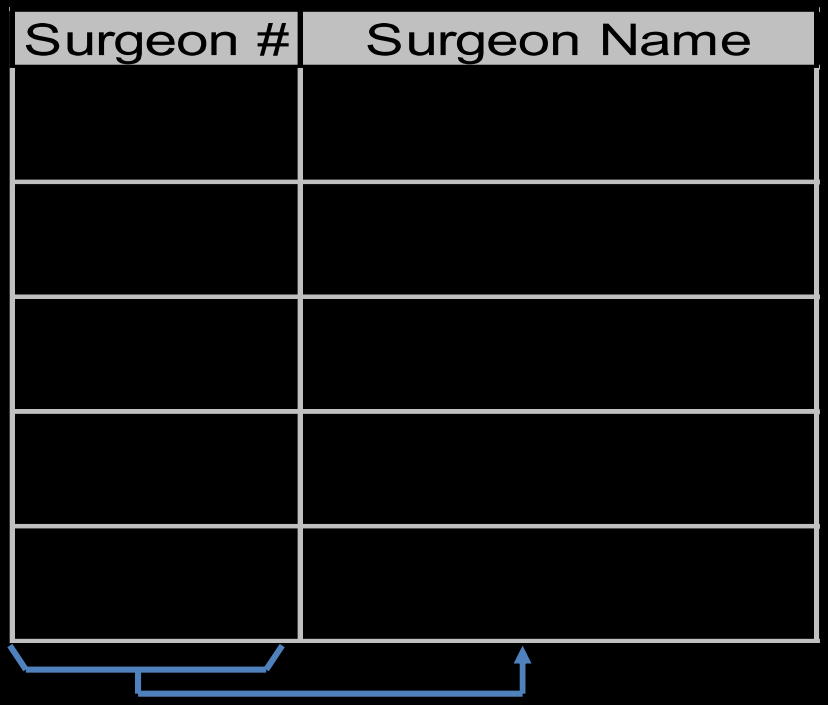
2NF Storage Anomalies

- Insertion: Cannot enter the fact that a particular drug has a particular side effect unless it is given to a patient.
- Deletion: If John White receives some other drug because of the penicillin rash, and a new drug and side effect are entered, we lose the information that penicillin can cause a rash
- Update: If drug side effects change (a new formula) we have to update multiple occurrences of side effects.

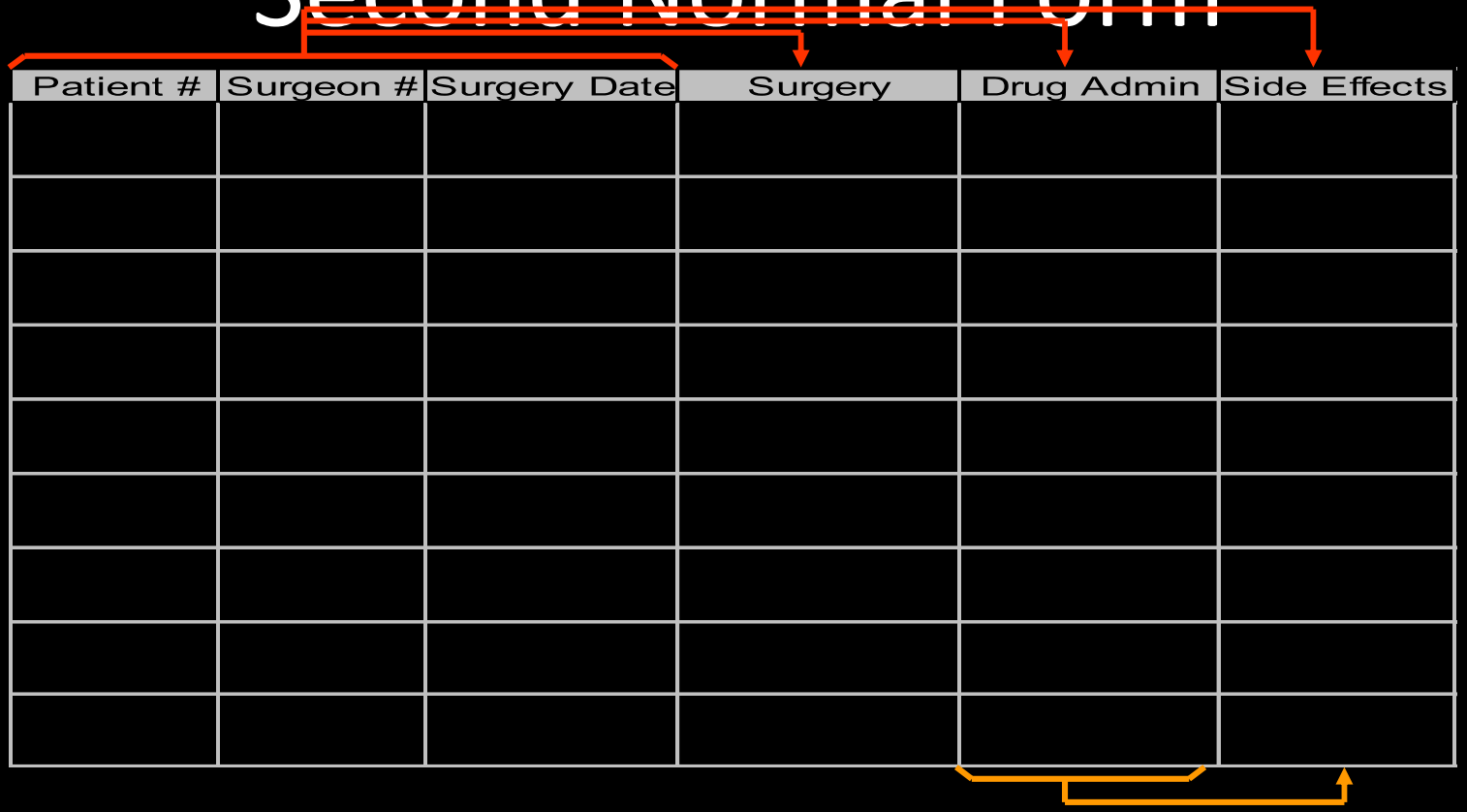
Second Normal Form



Second Normal Form



Second Normal Form



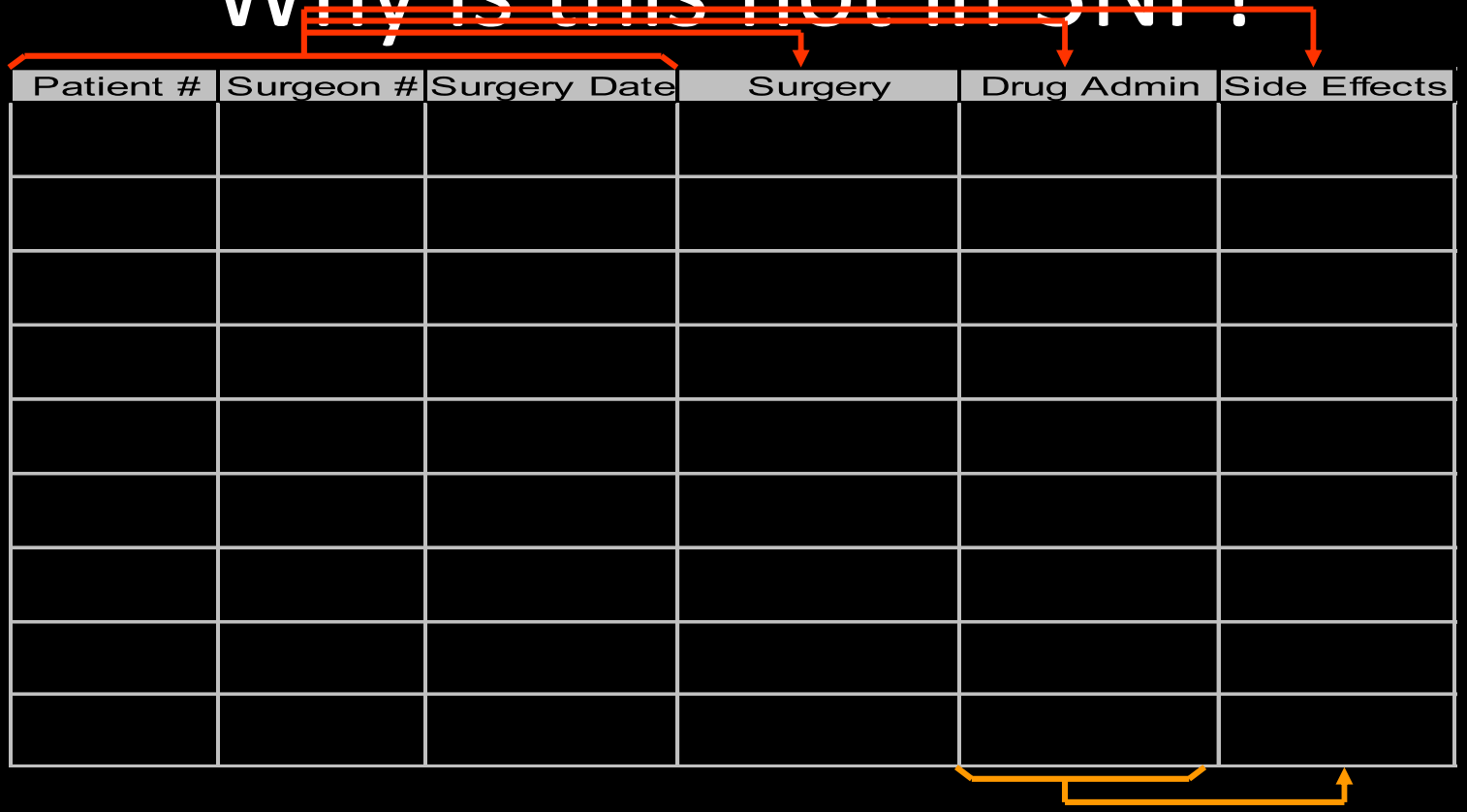
1NF Storage Anomalies Removed

- Insertion: Can now enter new patients without surgery.
- Insertion: Can now enter Surgeons who have not operated.
- Deletion (type 1): If Charles Brown dies, the corresponding tuples from Patient and Surgery tables can be deleted without losing information on David Rosen.
- Update: If John White comes in for third time, and has moved, we only need to change the Patient table

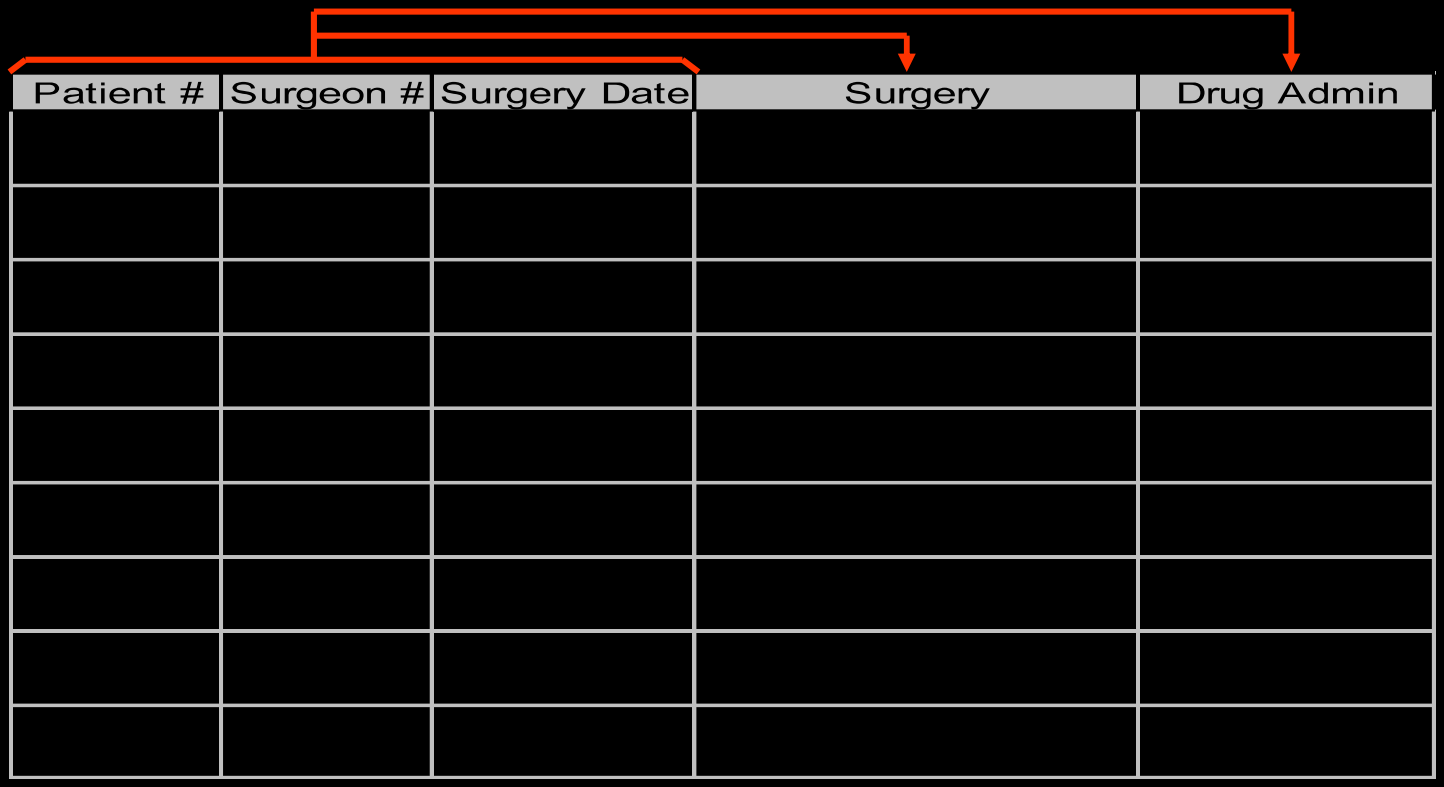
Third Normal Form

- A relation is said to be in Third Normal Form if there is no transitive functional dependency between non-key attributes
  - When one non-key attribute can be determined with one or more non-key attributes there is said to be a transitive functional dependency.
- The side effect column in the Surgery table is determined by the drug administered
  - Side effect is transitively functionally dependent on drug so Surgery is not 3NF

Why is this not in 3NF?

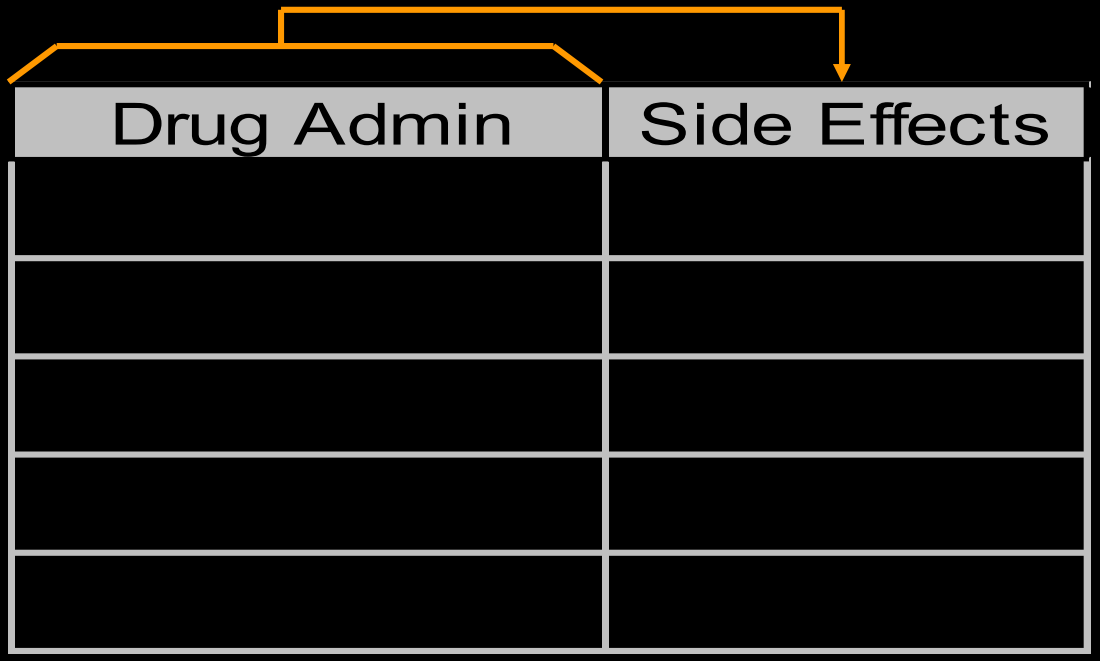


Third Normal Form



Patient #	Surgeon #	Surgery Date	Surgery	Drug Admin

Third Normal Form



2NF Storage Anomalies Removed

- Insertion: We can now enter the fact that a particular drug has a particular side effect in the Drug relation.
- Deletion: If John White receives some other drug as a result of the rash from penicillin, the information on penicillin and rash is maintained.
- Update: The side effects for each drug appear only once.

Boyce-Codd Normal Form

- Most 3NF relations are also BCNF relations.
- A 3NF relation is NOT in BCNF if:
  - Candidate keys in the relation are composite keys (they are not single attributes)
  - There is more than one candidate key in the relation, and
  - The keys are not disjoint, that is, some attributes in the keys are common

Fourth Normal Form

- Any relation is in Fourth Normal Form if it is BCNF *and* any multivalued dependencies are trivial
- Eliminate non-trivial multivalued dependencies by projecting into simpler tables

Fifth Normal Form

- A relation is in 5NF if every join dependency in the relation is implied by the keys of the relation
- *Implies that relations that have been decomposed in previous normal forms can be recombined via natural joins to recreate the original relation.*

Effectiveness and Efficiency Issues for DBMS

- Focus on the relational model
- Any column in a relational database can be searched for values.
- To improve efficiency indexes using storage structures such as BTrees and Hashing are used
- But many useful functions are not indexable and require complete scans of the the database

Example: Text Fields

- In conventional RDBMS, when a text field is indexed, only exact matching of the text field contents (or Greater-than and Less-than).
  - Can search for individual words using pattern matching, but a full scan is required.
- Text searching is still done best (and fastest) by specialized text search programs (Search Engines)

Normalization

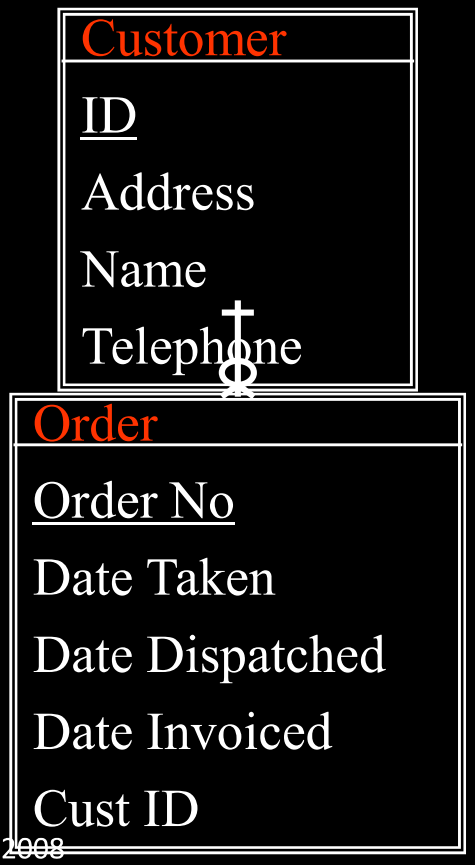
- Normalization is performed to reduce or eliminate Insertion, Deletion or Update anomalies.
- However, a completely normalized database may not be the most efficient or effective implementation.
- “Denormalization” is sometimes used to improve efficiency.

Normalizing to death

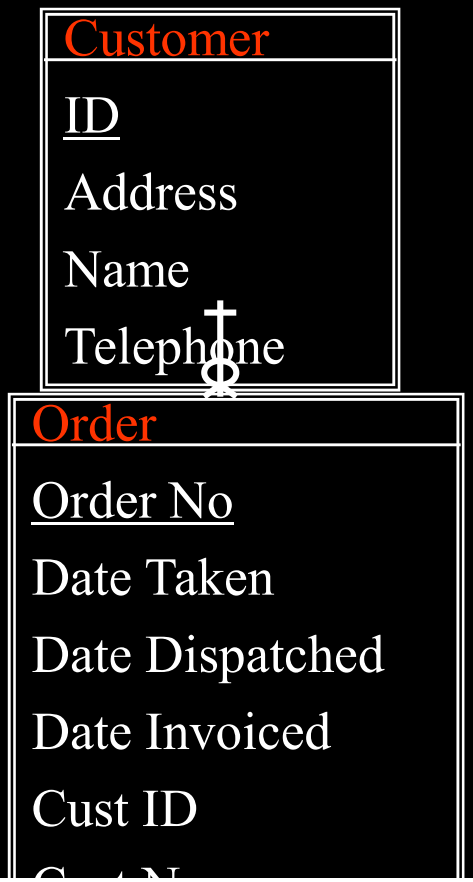
- Normalization splits database information across multiple tables.
- To retrieve complete information from a normalized database, the JOIN operation must be used.
- JOIN tends to be expensive in terms of processing time, and very large joins are very expensive.

Downward Denormalization

Before:



After:



Upward Denormalization

Denormalization

- Usually driven by the need to improve query speed
- Query speed is improved at the expense of more complex or problematic DML (Data manipulation language) for updates, deletions and insertions.

Database Management System

B.Sc. CS, Paper - I

Unit – V

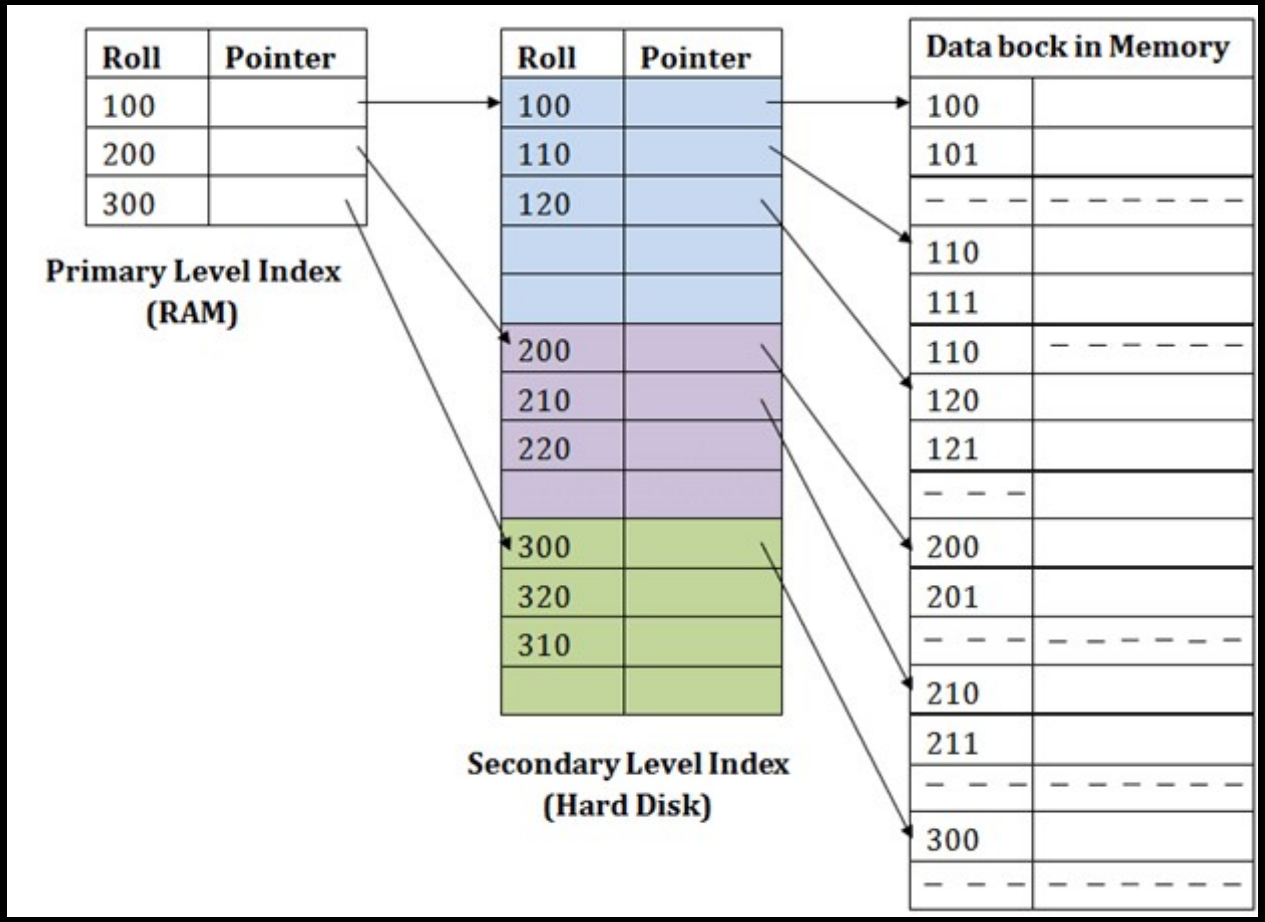
Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

INDEXING


Roll
Pointer
100
200
300
Primary Level Index
(RAM)
Roll
Pointer
100
110
120
200
210
220
300
320
310
Secondary Level Index
(Hard Disk)
Data bock in Memory
100
101
110
111
110
120
121
200
201
210
211
300

B Tree Files

B+ Tree

- The B+ **tree** is a balanced binary search **tree**. It follows a multi-level **index** format.
- In the B+ **tree**, leaf nodes denote actual data pointers. B+ **tree** ensures that all leaf nodes remain at the same height.
- In the B+ **tree**, the leaf nodes are linked using a link list.

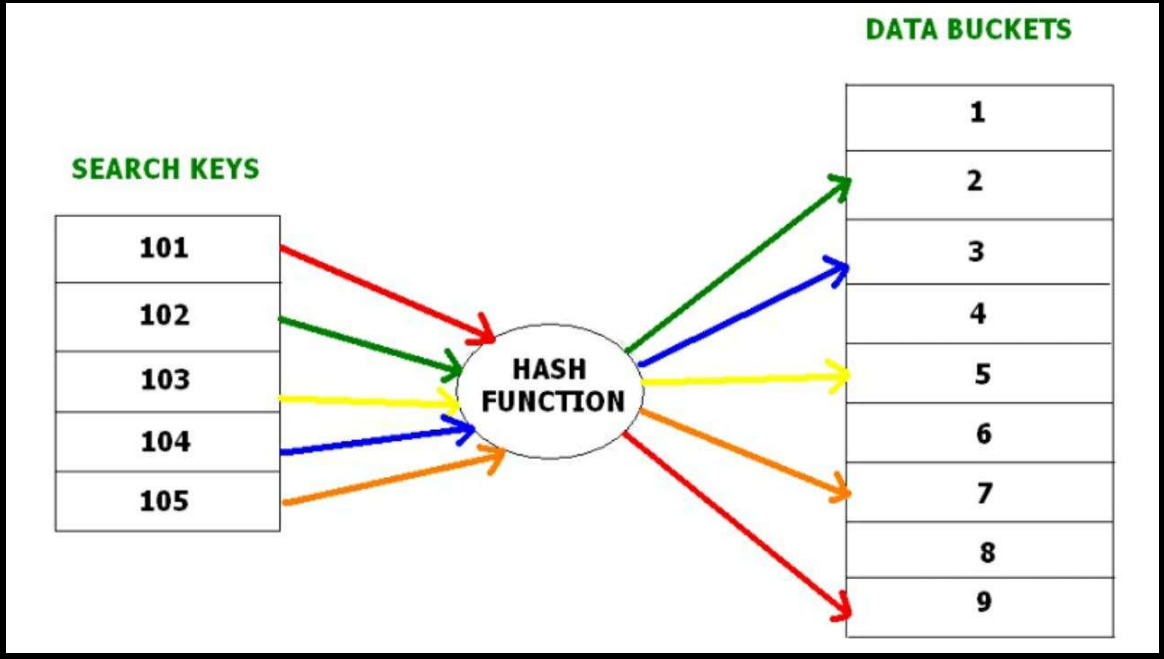
B^+-Tree Index Files

B^+-tree indices are an alternative to sequential index files.

- Disadvantage of sequential index files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

Hashing

In **DBMS**, **hashing** is a technique to directly search the location of desired data on the disk without using index structure. **Hashing** method is used to index and retrieve items in a database as it is faster to search that specific item using the shorter hashed key instead of using its original value.



Hashing

Why do we need Hashing?

Here, are the situations in the DBMS where you need to apply the Hashing method:

- For a huge database structure, it's tough to search all the index values through all its level and then you need to reach the destination data block to get the desired data.
- Hashing method is used to index and retrieve items in a database as it is faster to search that specific item using the shorter hashed key instead of using its original value.
- Hashing is an ideal method to calculate the direct location of a data record on the disk without using index structure.
- It is also a helpful technique for implementing dictionaries.

Hashing

Important Terminologies using in Hashing

Here, are important terminologies which are used in Hashing:

Data bucket – Data buckets are memory locations where the records are stored. It is also known as Unit Of Storage.

Key: A DBMS key is an attribute or set of an attribute which helps you to identify a row(tuple) in a relation(table). This allows you to find the relationship between two tables.

Hash function: A hash function, is a mapping function which maps all the set of search keys to the address where actual records are placed.

Linear Probing – Linear probing is a fixed interval between probes. In this method, the next available data block is used to enter the new record, instead of overwriting on the older record.

Hashing

Important Terminologies using in Hashing

Here, are important terminologies which are used in Hashing:

Quadratic probing- It helps you to determine the new bucket address. It helps you to add Interval between probes by adding the consecutive output of quadratic polynomial to starting value given by the original computation.

Hash index – It is an address of the data block. A hash function could be a simple mathematical function to even a complex mathematical function.

Double Hashing –Double hashing is a computer programming method used in hash tables to resolve the issues of has a collision.

Bucket Overflow: The condition of bucket-overflow is called collision. This is a fatal stage for any static has to function.

Hashing

There are mainly two types of SQL hashing methods:

Static Hashing Dynamic Hashing

Static Hashing

In the static hashing, the resultant data bucket address will always remain the same. Therefore, if you generate an address for say **Student_ID = 10** using hashing function **mod(3)**, the resultant bucket address will always be **1**. So, you will not see any change in the bucket address. Therefore, in this static hashing method, the number of data buckets in memory always remains constant.

Static Hash Functions

Inserting a record: When a new record requires to be inserted into the table, you can generate an address for the new record using its hash key. When the address is generated, the record is automatically stored in that location.

Searching: When you need to retrieve the record, the same hash function should be helpful to retrieve the address of the bucket where data should be stored.

Delete a record: Using the hash function, you can first fetch the record which is you wants to delete. Then you can remove the records for that address in memory.

Hashing

Static hashing is further divided into
Open hashing
Close hashing.

Open Hashing
In Open hashing method, Instead of overwriting older one the next available data block is used to enter the new record, This method is also known as linear probing.
For example, A2 is a new record which you wants to insert. The hash function generates address as 222. But it is already occupied by some other value. That's why the system looks for the next data bucket 501 and assigns A2 to it.

Close Hashing
In the close hashing method, when buckets are full, a new bucket is allocated for the same hash and result are linked after the previous one.

Hashing

Close Hashing

In the close hashing method, when buckets are full, a new bucket is allocated for the same hash and result are linked after the previous one.

Dynamic Hashing

Dynamic hashing offers a mechanism in which data buckets are added and removed dynamically and on demand. In this hashing, the hash function helps you to create a large number of values.

What is Collision?

Hash collision is a state when the resultant hashes from two or more data in the data set, wrongly map the same place in the hash table.

How to deal with Hashing Collision?

There are two technique which you can use to avoid a hash collision:

Rehashing: This method, invokes a secondary hash function, which is applied continuously until an empty slot is found, where a record should be placed.

Chaining: Chaining method builds a Linked list of items whose key hashes to the same value. This method requires an extra link field to each table position.

Comparison of Ordered Indexing and Hashing

Parameters	Order Indexing	Hashing
Storing of address	Addresses in the memory are sorted according to a key value called the primary key	Addresses are always generated using a hash function on the key value.
Performance	It can decrease when the data increases in the hash file. As it stores the data in a sorted form when there is any (insert/delete/update) operation performed which decreases its performance.	Performance of hashing will be best when there is a constant addition and deletion of data. However, when the database is huge, then hash file organization and its maintenance will be costlier.
Use for	Preferred for range retrieval of data- which means whenever there is retrieval data for a particular range, this method is an ideal option.	This is an ideal method when you want to retrieve a particular record based on the search key. However, it will only perform well when the hash function is on the search key.
Memory management	There will be many unused data blocks because of the delete/update operation. These data blocks can't be released for re-use. That's why regular maintenance of the memory is required.	In static and dynamic hashing methods, memory is always managed. Bucket overflow is also handled perfectly to extend static hashing.

Hashing

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89