

Introduction to DBMS

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

Introduction to DBMS

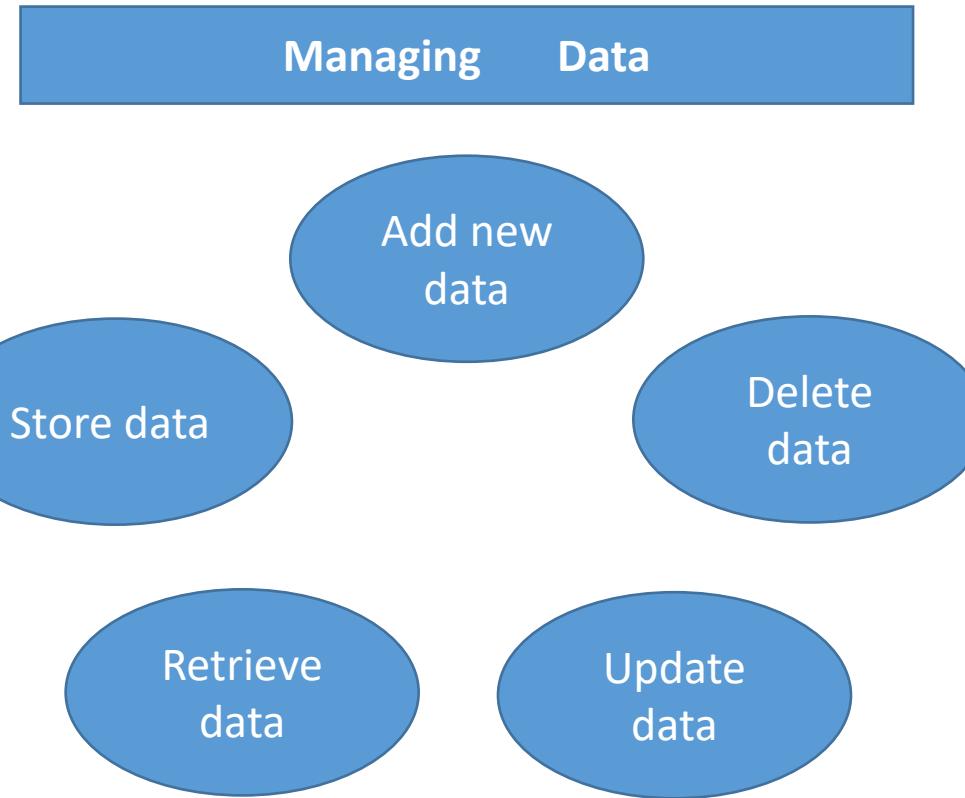
- DBMS stands for Database Management System.
- DBMS = Database + Management System.
- Database is a collection of data and Management System is a set of programs to store and retrieve those data.
- Based on this we can define DBMS like this: DBMS is a collection of inter-related data and set of programs to store & access those data in an easy and effective manner.

What is the need of DBMS?

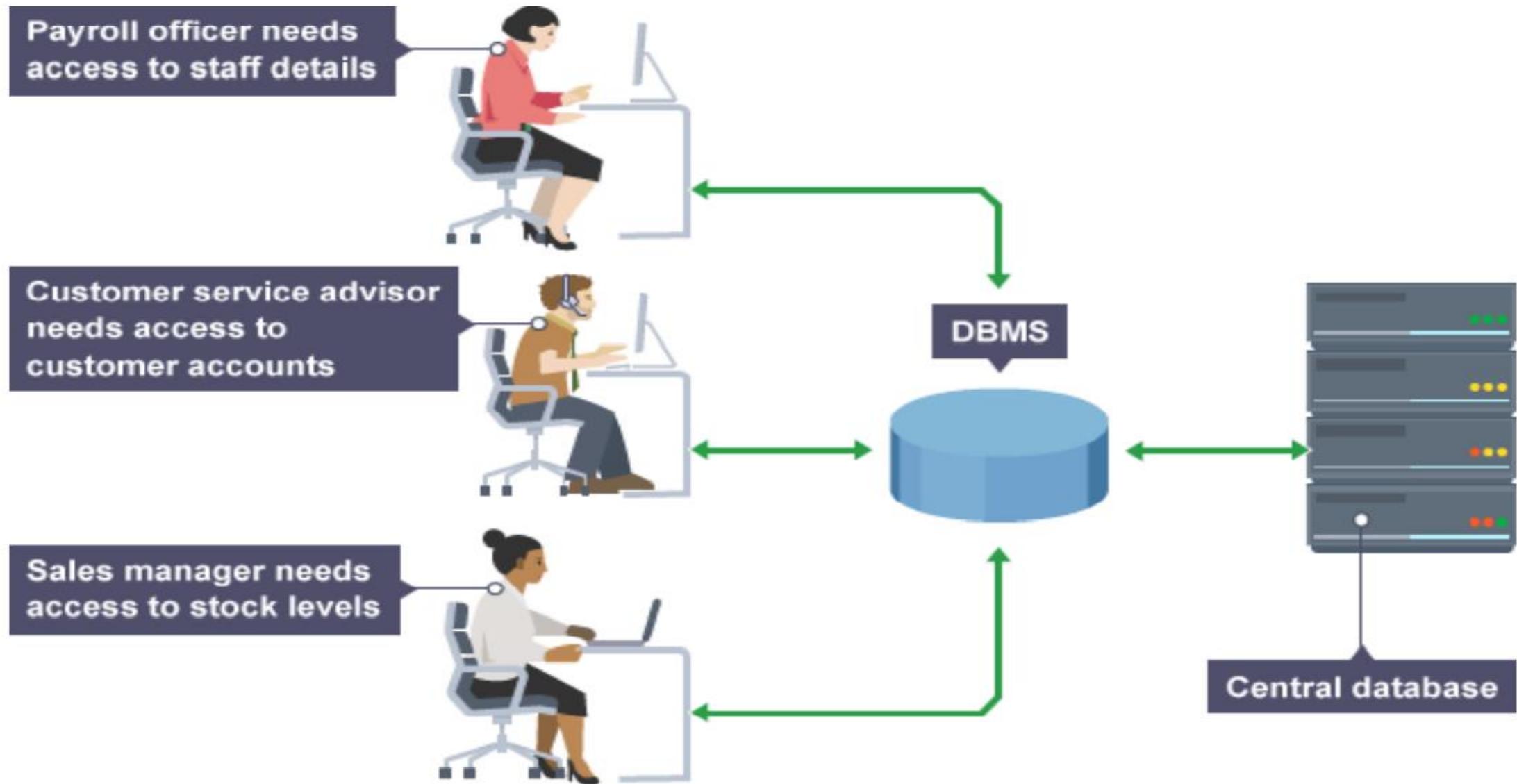
- **Storage:** Acquires less space as the redundant data (duplicate data) has been removed before storage.
Example: A person having 2 accounts and the details
- **Fast Retrieval of data:** Faster retrieval of data through queries

Purpose of Database Systems

- To manage the data.
- Consider a University that keeps the data of students, teachers, courses, books etc.
- For all the above , we need a **Database Management System.**

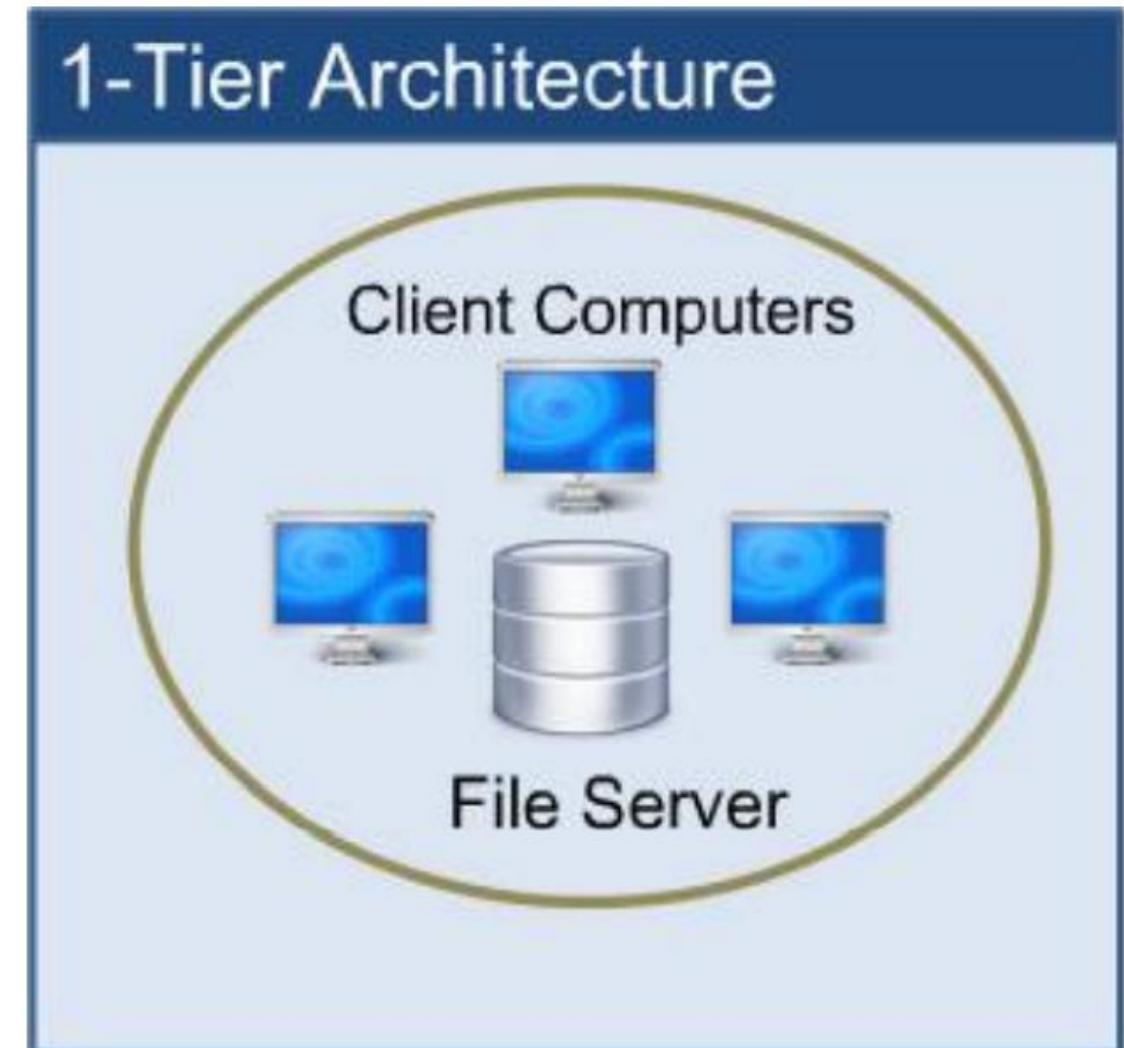


Examples for DBMS



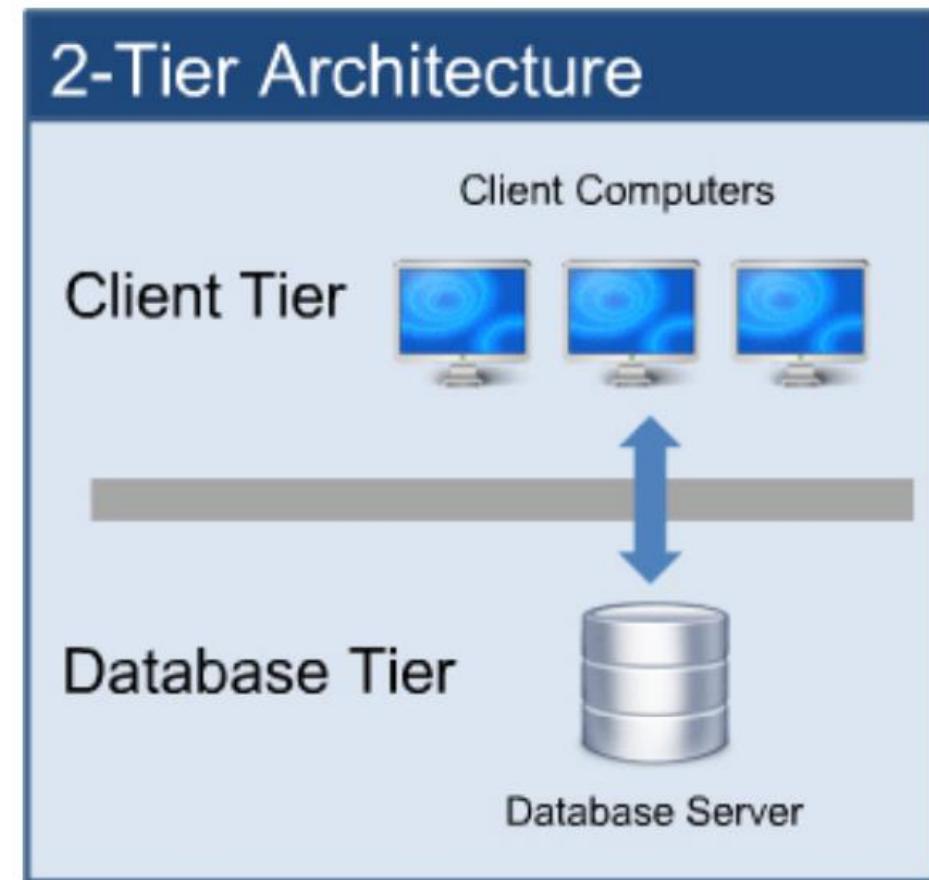
One Tier Architecture

- Simple Architecture
- Run on a single computer system and do not interact with other computer systems.
- Install a Database in your system and access it to practice SQL queries.
- Rarely used in production.



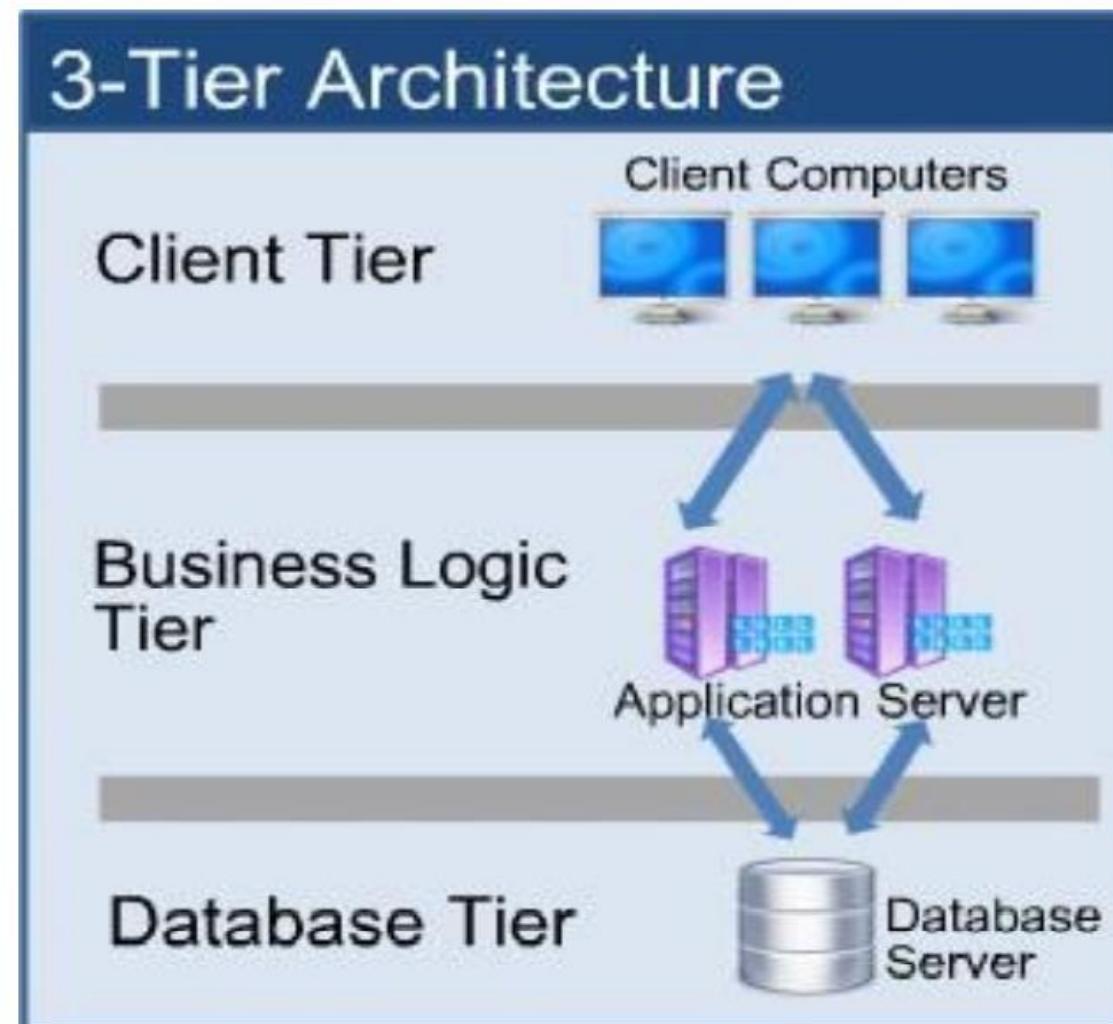
Two Tier Architecture

- Client-server architecture
- Client – machine used by end-user.
- Server – high end machine holds software and hardware
- The application programs run on the client side
- Client will send Query / request transaction through ODBC / JDBC



Three Tier Architecture

- Client machine just acts as a front end and does not contain any direct database calls
- client end communicates with an application server
- application server in turn communicates with a database system to access Data
- applications that run on the World Wide Web



Few more Applications of DBMS

- **Airlines:** reservations, schedules, etc
- **Telecom:** calls made, customer details, network usage, etc
- **Universities:** registration, results, grades, etc
- **Sales:** products, purchases, customers, etc
- **Banking:** all transactions etc
- **Education sector:** staff ,student & course details, registration, etc
- **Online shopping:** product details, billing, shipping, etc

DBMS vs File System

Drawbacks of File system

- **Data redundancy:** Refers to the duplication of data.

Ex: Students registers for 2 courses, the details of that student has to be saved in two files, this will take more storage than needed. Data redundancy often leads to higher storage costs and poor access time.

- **Data inconsistency:** Data redundancy leads to data inconsistency.

Ex: Considering the student in the above example, if the student requests to change his address. If the address is changed at one place and not on all the records then this can lead to data inconsistency.

- **Data Isolation:** Data are scattered in various files and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.
- **Dependency on application programs:** Changing files would lead to change in application programs.

Drawbacks of File system

- **Atomicity issues:** All the operations in a transaction executes or none.
Ex: Fund transfer across accounts involves debit and credit operations, incase after debit, system fails. It is difficult to achieve atomicity in file processing systems.
- **Data Security:** Unauthorized access should be avoided.
Ex: A student in a college should not be able to see the payroll details of the teachers, such kind of security constraints are difficult to apply in file processing systems.

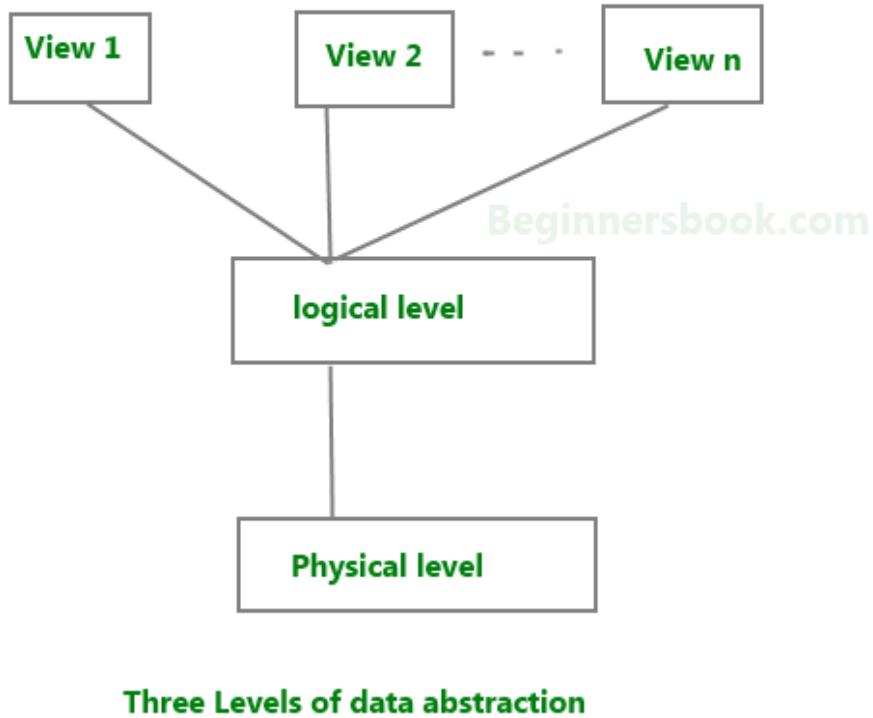
Advantages of DBMS over file system

- **No redundant data:** Redundancy removed by data normalization. No data duplication saves storage and improves access time.
- **Data Consistency and Integrity:** Since data normalization takes care of the data redundancy, data inconsistency is checked
- **Data Security:** It is easier to apply access constraints in database systems so that only authorized user is able to access the data.
- **Privacy:** Limited access means privacy of data.
- **Easy access to data** – Database systems manages data in such a way so that the data is easily accessible with fast response times.
- **Easy recovery:** Since database systems keeps the backup of data, it is easier to do a full recovery of data in case of a failure.
- **Flexible:** Database systems are more flexible than file processing systems.

When to use a relational database

- Your data can be organised in tabular form
 - E.g. information about things that share common properties
- You are interested in multiple types of entity
 - And the relationships between them
 - Entities may be concrete or more abstract
- You want to identify instances of things that meet certain criteria
- You want to be able to present one dataset in multiple different ways
 - Query results can be exported and used elsewhere

Data Abstraction in DBMS



We have three levels of abstraction:

Physical level: Lowest level of data abstraction.
It describes how data is actually stored in database.

Logical level: Describes what data is stored in database.

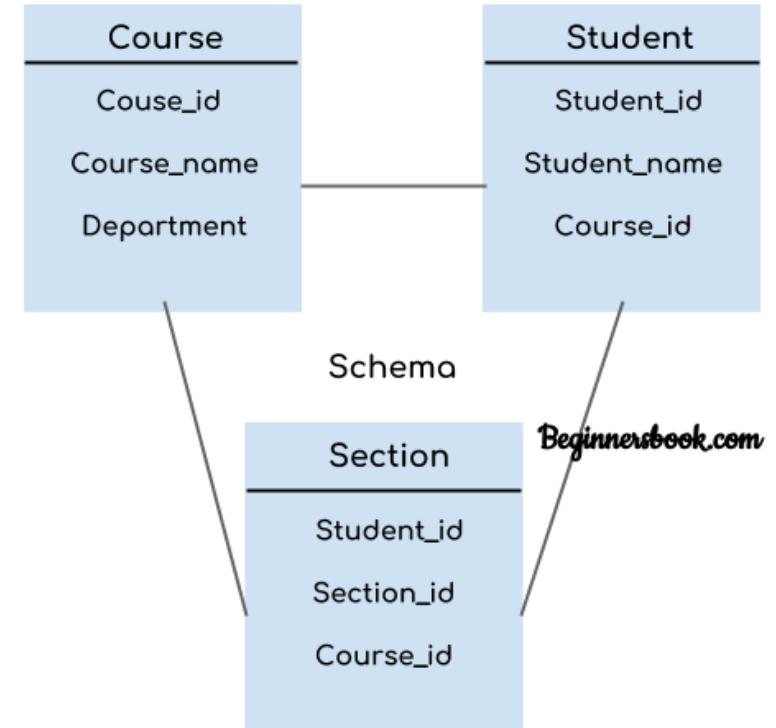
View level: Highest level of data abstraction.
Describes the user interaction with database system.

Schema

Design of a database is called the schema.

Schema is of three types:

- Physical schema
- Logical schema
- View schema

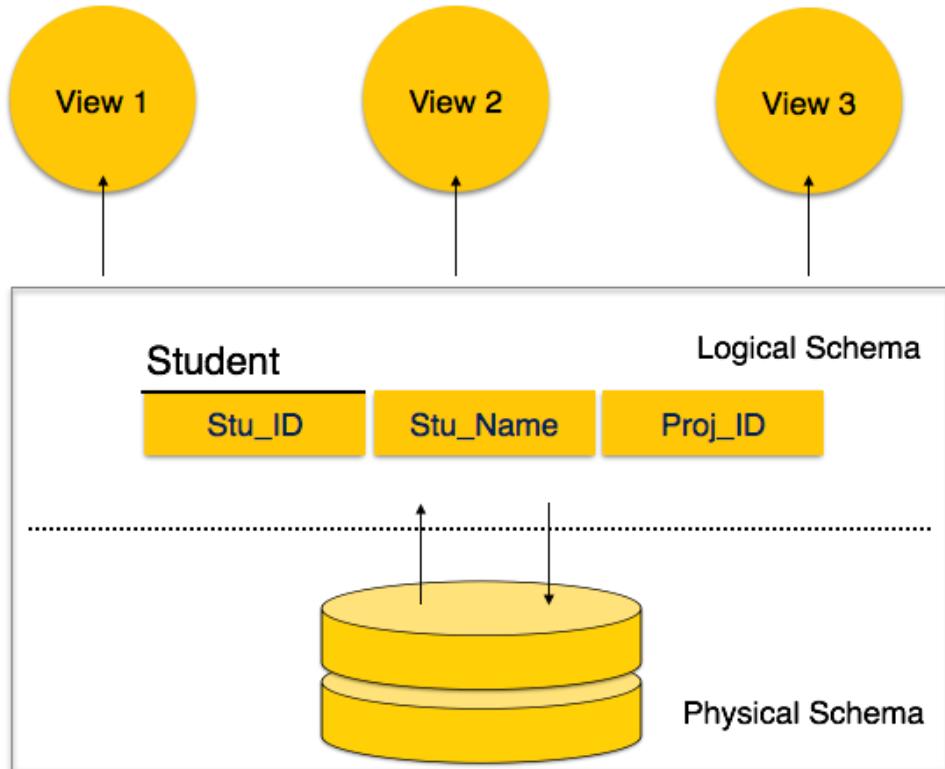


For example: In the diagram, we have a schema showing the relationship between three tables: Course, Student and Section. It does not show the data present in those tables. Schema is only a structural view/design of a DB

Types of Schema

View schema: Design of database at view level

Describes end user interaction with database systems



Physical schema : The design of a database at physical level

How the data stored in blocks of storage is described at this level.

Logical schema: Design of database at logical level

Programmers and database administrators work at this level, which describes the table structure , how attributes/columns are interrelated, however the internal details such as implementation of data structure is hidden at this level (available at physical level).

Instance of a DB

Definition

The data stored in database at a particular moment of time.

For example, lets say we have a single table student in the database, today the table has 100 records, so today the instance of the database has 100 records. Lets say we are going to add another 100 records in this table by tomorrow so the instance of database tomorrow will have 200 records in table. In short, at a particular moment the data stored in database is called the instance, that changes over time when we add or delete data from the database.



Question????

Is it schema that changes frequently or is it instance?????



Difference between schema and instance

S C H E M A

V E R S U S

I N S T A N C E

SCHEMA

Visual representation of a database which is a set of rules that govern a database

Formal description of the structure of the database

Does not change frequently

INSTANCE

Data stored in a database at a particular time

Set of information stored in a database at a particular time

Changes frequently

Difference between schema and instance

Analogy

P-ID	Name	Pname

Template for a Table

Data Independence

What is Data Independence ??????

Property of DBMS that helps you to change the Database schema at one level of a database system without requiring to change the schema at the next higher level.

Data Independence

Why Data Independence ??????

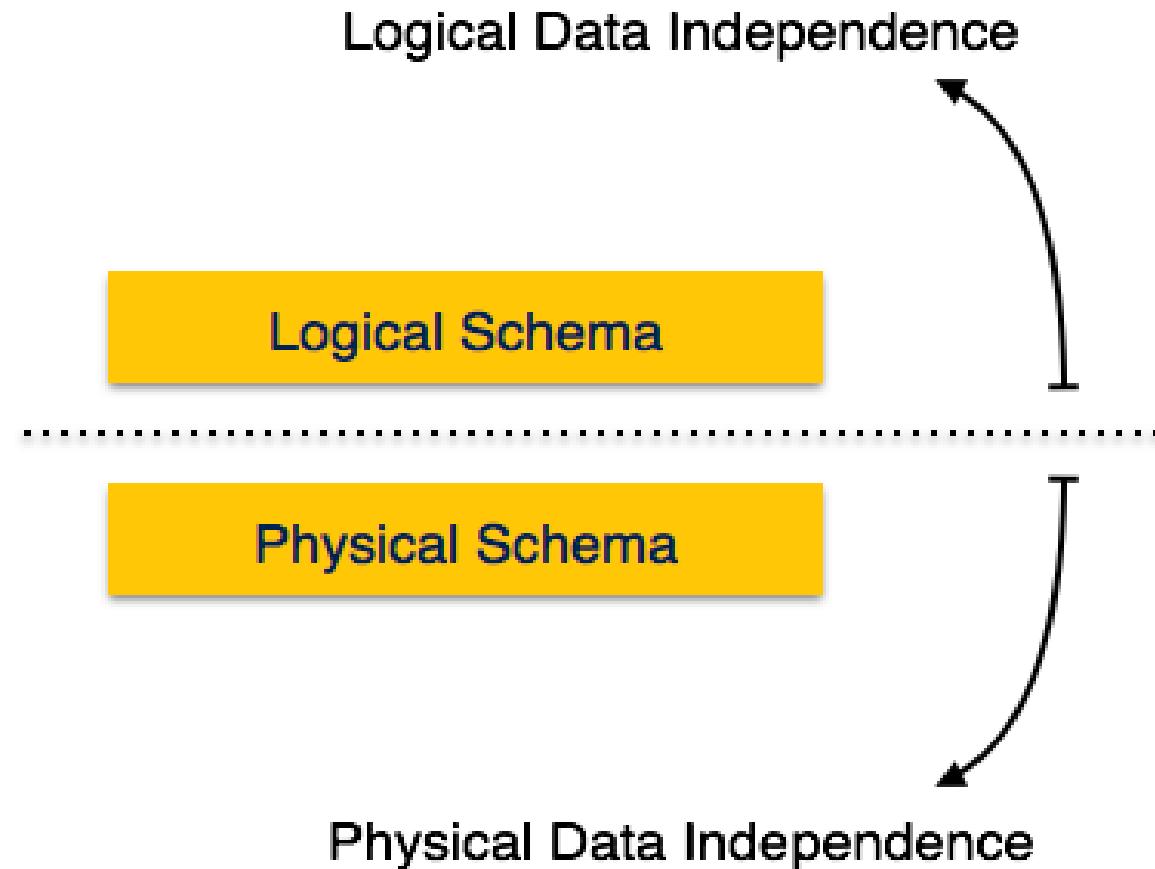
A database system normally contains a lot of data in addition to users' data.

Stores data about data, known as metadata, to locate and retrieve data easily.

It is rather difficult to modify or update a set of metadata once it is stored in the database.

But as a DBMS expands, it needs to change over time to satisfy the requirements of the users. If the entire data is dependent, it would become a tedious and highly complex job.

Data Independence



Data Independence

Physical Data Independence

2006 - AUMS – accommodated data for say 1000 students.... X units of memory

2016 –AUMS - accommodates data for say 3000 students..... X+Y units of memory

Logical Data Independence

Student(RegNo, name, marks1, marks 2, total, addr, fee)

Note: Total is defined as marks1 + marks 2

Admin:

RegNo (R),
name(R),
addr(R),
fee (R & W)

Stud:

RegNo (R),
name (R),
marks1 (R),
marks2 (R),
total (R),
addr(R & W),
fee (R)

Teacher:

RegNo (R),
name (R),
marks1 (R),
marks2 (R),
total (R),
addr(R)

Note: Total is later defined as marks1 + marks 2 + 5

Data Independence

Physical Data Independence

All the schemas are logical, and the actual data is stored in bit format on the disk.

The ability to change the physical data without impacting the schema at the logical level.

For example, in case we want to change or upgrade the storage system itself – suppose we want to replace hard-disks with SSD – it should not have any impact on the logical data or schemas. Or

may be we increase the memory capacity of the hard disk, and this is no way requires an explicit change in the logical level

Logical Data Independence

Logical data is data about database, that is, it stores information about how data is managed inside.

Logical data independence is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

The ability to change the logical data without impacting the schema at the view level

For example, a table (relation) stored in the database and all its constraints, applied on that relation. Or

We enforce a constraint in the logical level

What is SQL

- Structured Query Language
- Not a programming language
- A computer language for storing, manipulating and retrieving data stored in relational database.
- Standard language for Relation Database System. All relational database management systems like “MySQL, MS Access, Oracle, Sybase, Informix, postgres and SQL Server” use SQL as standard database language.

Terminology

- Database - contains one or more tables
- Relation (or table) - contains tuple and attributes
- Tuple (or row) - set of attributes that represent an “ object”
- Attribute (or column or field) - one of possibly elements of the data concerning the object represented by the tuple

Sample relational database

<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account-number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

<i>customer-id</i>	<i>account-number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Designing a relational database

- Relation or Table
- Record or rows
- Columns or fields
- Datatypes
- NULL
- Constraints
- Uniqueness
- Transaction (ACID)
- Index
- ERD
- RDBMS

Designing the tables

People				
Surname	Wilson	Temple	Sterling	Elliott
First name	Adam	Thos	Oliver	Justin
Middle initial(s)	T	G	J K	W
Date of birth	3/8/1697	6/10/1705	23/5/1720	24/2/1718
...				
Notes	Born France	London		landowner

SQL datatypes

- String types
 - CHAR(n) – fixed-length character data, n characters long Maximum length = 2000 bytes
 - VARCHAR2(n) – variable length character data, maximum 4000 bytes
 - LONG – variable-length character data, up to 4GB. Maximum 1 per table
- Numeric types
 - NUMBER(p,q) – general purpose numeric data type
 - INTEGER(p) – signed integer, p digits wide
 - FLOAT(p) – floating point in scientific notation with p binary digits precision
- Date/time type
 - DATE – fixed-length date/time in dd-mm-yy form

Types of data

People	
Surname	text
First name	text
Middle initial(s)	text
Date of birth	date
...	
Notes	memo

Books	
Title	text
Author	text
DatePub	date
...	
Place	text
ISBN	text
...	
...	

Set a data type for each field:
Text, Number,
Date/time,
Currency, Yes/No

Person
Surname
First name
Middle initial(s)
Date of birth
Notes

Publication
Title
Author(s)
Publisher
Date of publication
Place of publication
Edition
Format
Type of publication
Price
Sales
Notes

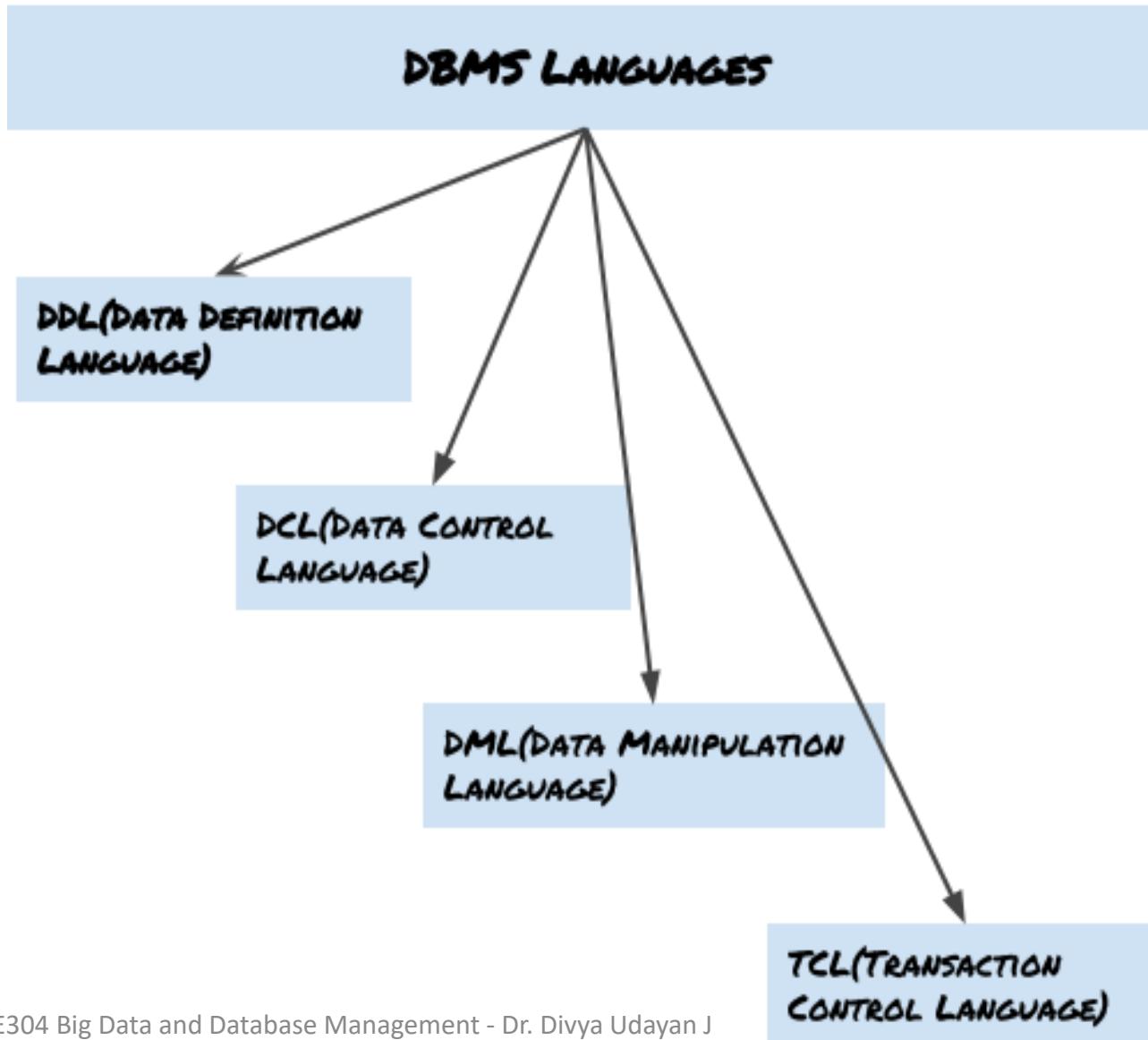
Publisher
Name
Staff
Founded
Ceased
Address
Notes

Reference
Author(s)
Title
Date of publication
Edition
Volume
Page(s)
URL
Notes

Database Operations (CRUD)

- Consider IMDB
- Create data in a table
 - A new actor has just appeared in a film
- Read data from a table
 - Somebody has searched for an actor
- Update data in a table
 - An actor has appeared in a new movie
- Delete data in a table
 - A planned movie is cancelled

DBMS Languages



DBMS Languages

Data Definition Language (DDL)

DDL is used for specifying the database schema.

It is used for creating tables, schema, indexes, constraints etc. in database.

- To create the database instance – **CREATE**
- To alter the structure of database – **ALTER**
- To drop the structure of database – **DROP**
- To delete tables in a database instance – **TRUNCATE**

DBMS Languages

Data Manipulation Language (DML)

DML is used for accessing and manipulating data in a database.

The following operations on database comes under DML:

- To read records from table(s) – **SELECT**
- To insert record(s) into the table(s) – **INSERT**
- Update the data in table(s) – **UPDATE**
- Delete all the records from the table – **DELETE**

DBMS Languages

Data Control language (DCL)

DCL is used for **granting and revoking user access** on a database –

- To grant access to user – **GRANT**
- To revoke access from user – **REVOKE**

DBMS Languages

Transaction Control Language(TCL)

The changes in the database that we made using DML commands are either performed or rolled back using TCL.

- To persist the changes made by DML commands in database – **COMMIT**
- To rollback the changes made to the database – **ROLLBACK**

ACID

- **Atomicity**
 - All or none
- **Consistency**
 - Always in a legal state
- **Isolation**
 - Each user is isolated from each other user
- **Durability**
 - Can recover after a crash or power failure

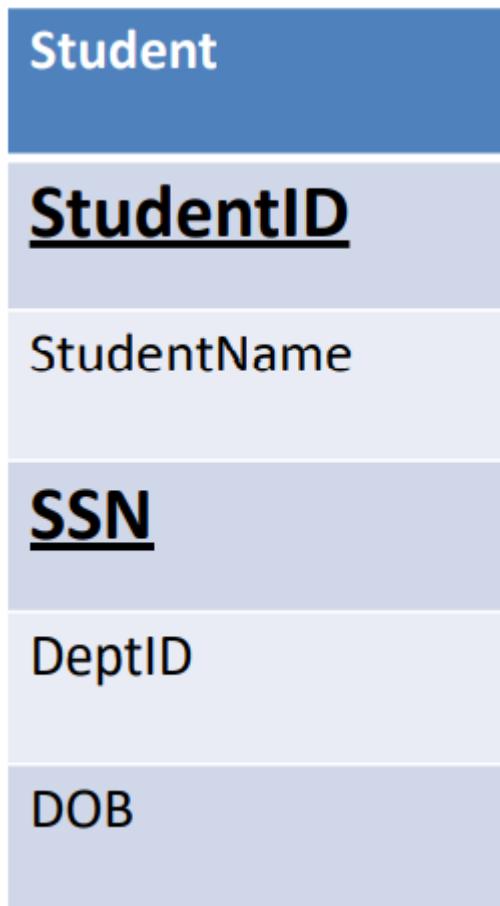
Importance of ACID properties

- Atomicity preserves the “completeness” of the business process.
- Consistency refers to the state of the data both before and after the transaction is executed.
- A transaction maintains the consistency of the state of the data. In other words, after running a transaction, all data in the database is “correct.”

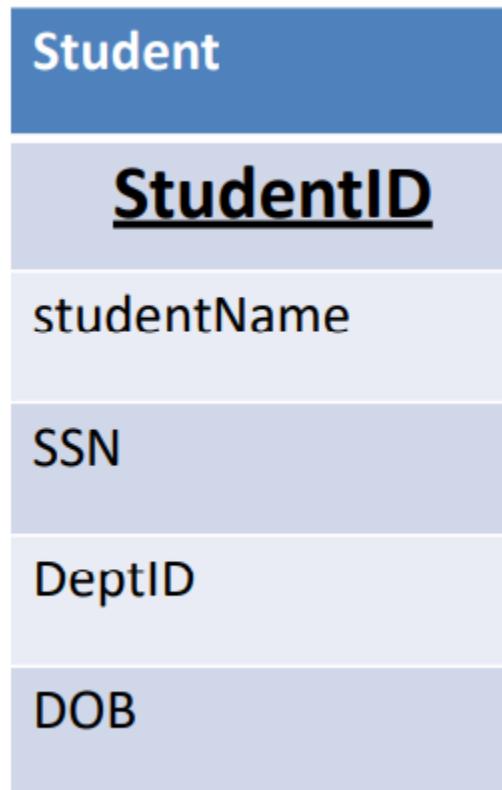
Keys

- Super Key
- Candidate Key
- Primary Key
- Foreign Key
- Composite Key
- Alternate Key
- Unique Key or Surrogate Key

Candidate Key



Primary Key



Conditions to be met : Primary Key

- No two rows can have the same primary key value.
- Every row must have a primary key value.
- The primary key field cannot be null.
- Value in a primary key column can never be modified or updated, if any foreign key refers to that primary key.

Composite Key

Student

StudentID

StudentName

SSN

DeptID

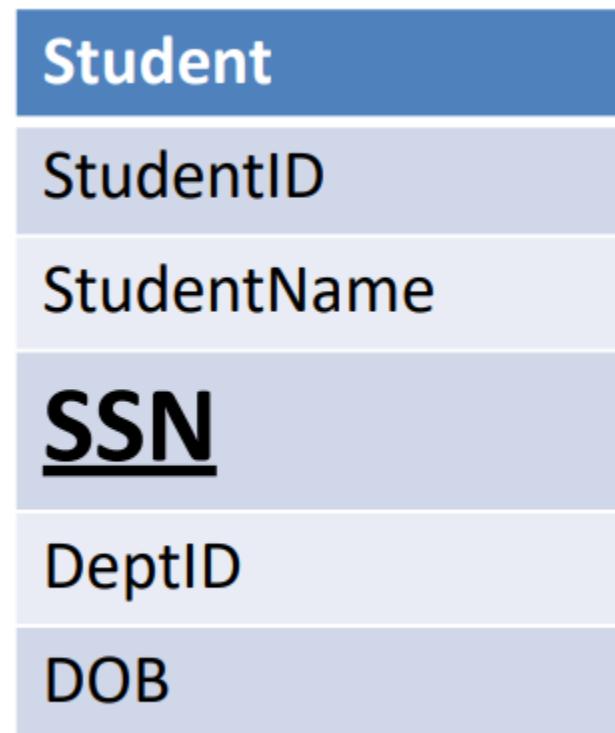
DOB

Foreign Key

Student
StudentID
StudentName
SSN
<u>DeptID</u>
DOB

Department
<u>DeptID</u>
DeptName

Alternate Key/Secondary Key



Super Key

Student

StudentID

StudentName

SSN

DeptID

DOB

Unique Key

Student

StudentID

StudentName

SSN

EmailID

DOB

Database Management Systems

Chapter 2 *Entity Relationship Diagram* *Or* *Entity Relationship Model*



Entity:

- Entity is any *real world object* with *well defined property*.
- Entity have their independent existence.
- An Entity can be a thing, person, place, unit, object or any item about which the data should be captured and stored in the form of properties and tables.
- Entity without properties is not an entity.

Examples of Entities:

- **Person:** Employee, Student, Patient
- **Place:** Store, Building
- **Object:** Machine, product, and Car
- **Event:** Sale, Registration, Renewal
- **Concept:** Account, Course

What is an entity type

- A database has a group of entities that are similar.
- An entity type defines a collection (or set) of entities that have the same attributes.
- Each entity type in the database is described by its name and attributes.

What is an entity set?

- The collection of all entities of a particular entity type in the database at any point of time is called an entity set
- The entity set is usually referred to using the same name as the entity type.

Entity type name: EMPLOYEE
(Name, age, salary)

Entity set:
(Extension)

e1.
(Raju, 25, 25000)
e2.
(Arial, 28, 29000)
 → e3.
(Jeet, 24, 22000)

.

Example

University database, we have entities : Students, Courses, and Lecturers.

- Students entity can have attributes : Rollno, Name and Sem.
 - Course entity can have attributes : CourseCode, CourseName and Credit.
 - Lecturer entity can have attributes: Id, lec_Name and Area_of_Spl
-
- They might have relationships with Courses and Lecturers.

(Student ***registers*** for a course ***taught*** by lecturer.)

(***registers*** and ***taught*** are relationships)



Entity

Person, place, object, event or concept about which data is to be maintained
Example: Car, Student



Attribute

Property or characteristic of an entity
Example: Color of car Entity
Name of Student Entity



Relation



Association between the instances of one or more entity types

Example: Blue Car Belongs to Student Jack

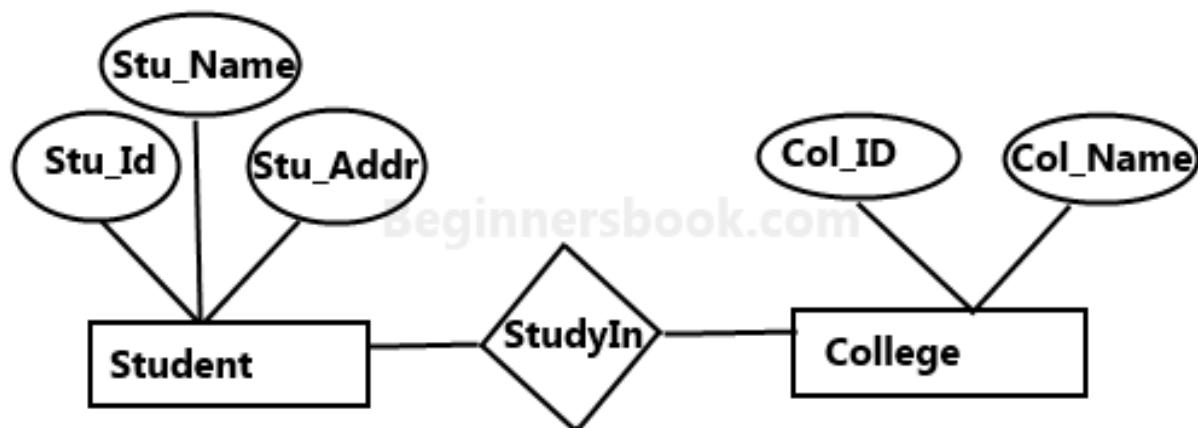


Example:

Two Entities: Student and College and their relationship.

Student entity has attributes such as Stu_Id, Stu_Name & Stu_Addr and
College entity has attributes such as Col_ID & Col_Name.

The relationship between Student and College is that the students study in colleges.



Sample E-R Diagram

What is an Entity Relationship Diagram (ER Diagram)?

- An ER diagram shows the relationship among entities & relationships.
- ER diagram shows the complete logical structure of a database.

Components of the ER Diagram

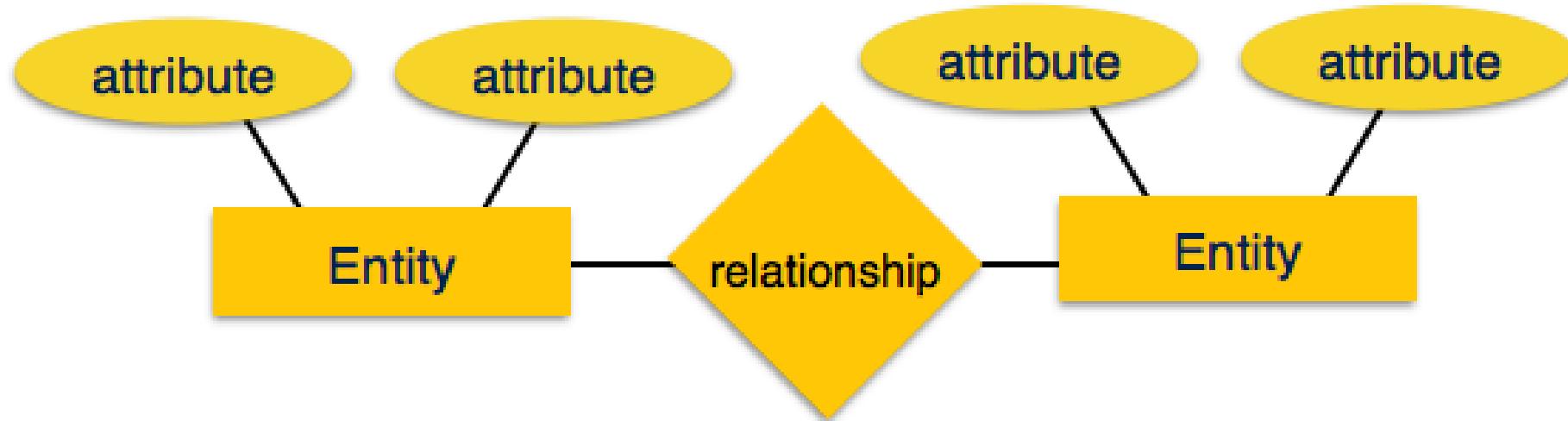
This model is based on three basic concepts:

- Entities
- Attributes
- Relationships

Why use ER Diagrams?

- ✓ ER Diagram is allowed you to communicate with the logical structure of the database to users
- ✓ Provide a preview of how all your tables should connect, what fields are going to be on each table
- ✓ ER diagrams are translatable into relational tables which allows you to build databases quickly
- ✓ The database designer gains a better understanding of the information to be contained in the database with the help of ER diagram

ER Diagram : Basic Skeleton

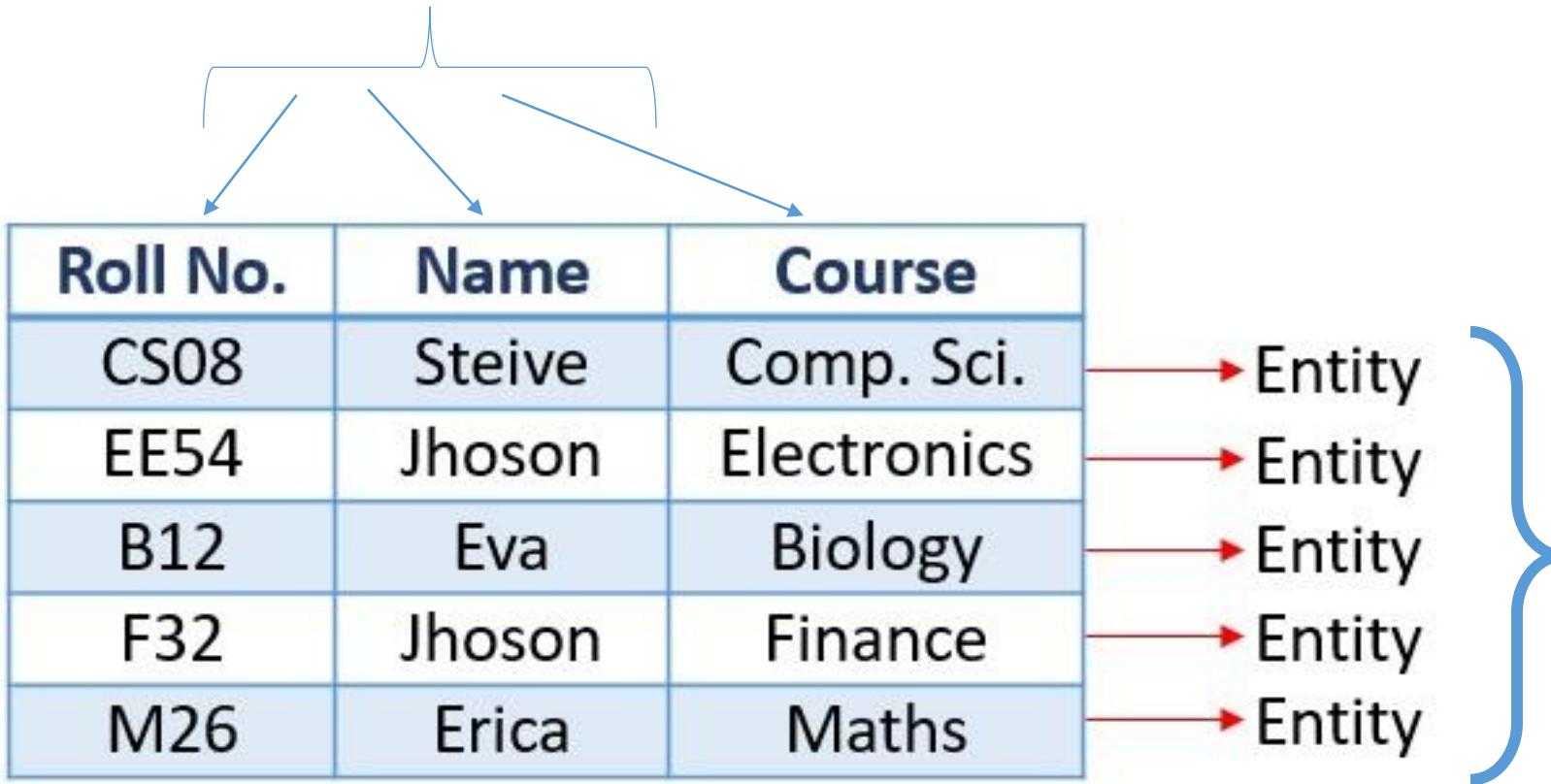


What is an Entity.....?????

What is Relationship.....?????

What is an Attribute.....?????

Attributes



Student Table

Rectangle: Represents Entity sets.



Ellipses: Attributes



Diamonds: Relationship Set



Double Ellipses: Multivalued Attributes



Dashed Ellipses: Derived Attribute



Double Rectangles: Weak Entity Sets



Lines: They link attributes to Entity Sets & Entity sets to Relationship Set / Partial participation of an entity in a relationship



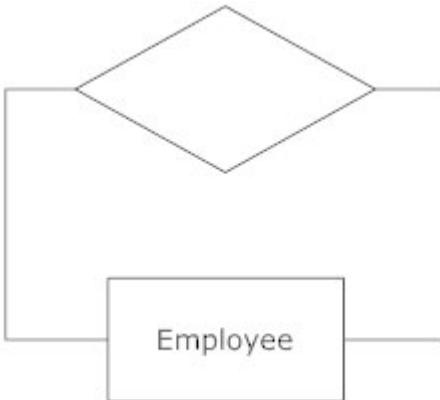
Double Lines: Total participation of an entity in a relationship



Cardinality: specifies how many instances of an entity relate to one instance of another entity.

1:1 , 1:M / M:1, M:N

In some cases, entities can be self-linked. For example, employees can supervise other employees.



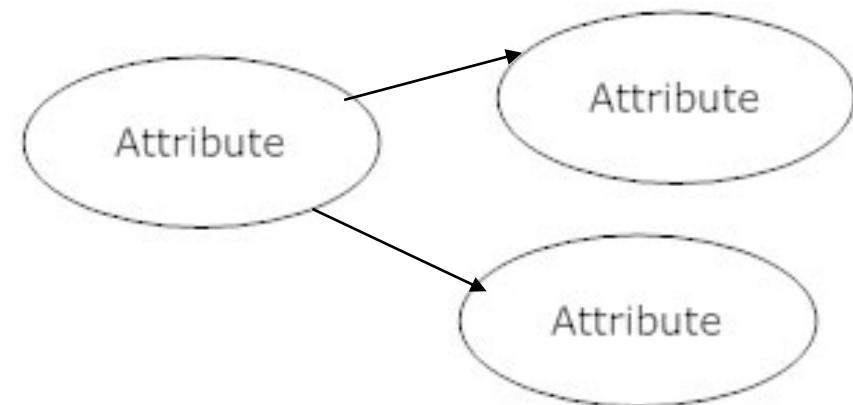
Ellipse with name underlined (dashed): Discriminator/Partial key



Ellipse with name underlined (solid): Primary Key



Connected Ellipses: Composite Attribute



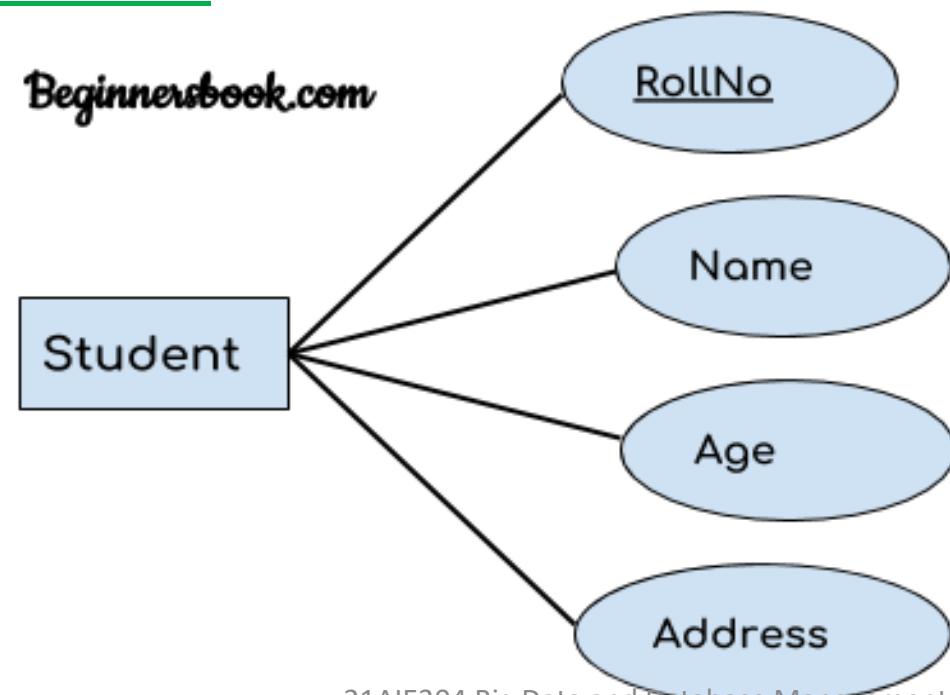
Attributes:

There are five types of attributes:

1. Key attribute
2. Simple and single valued attribute
3. Composite attribute
4. Multivalued attribute
5. Derived attribute and stored attributes

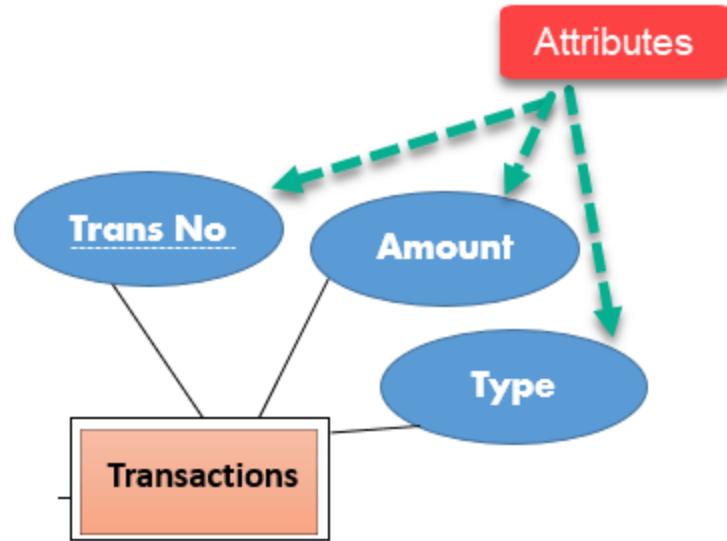
1. Key attribute:

- A key attribute can uniquely identify an entity from an entity set.
- For example, student roll number can uniquely identify a student from a set of students.
- Key attribute is represented by oval same as other attributes however the **text of key attribute is underlined**.



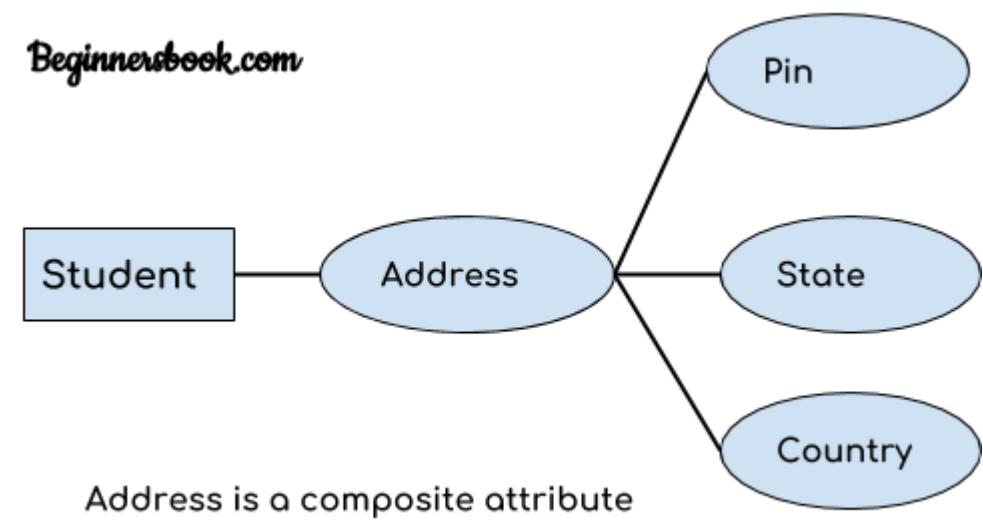
2. Simple attribute/single valued:

- Attribute not divided in to sub parts



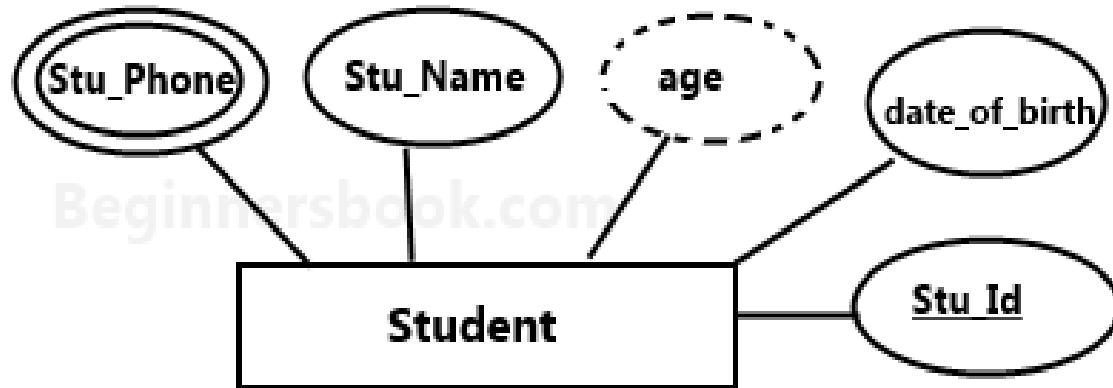
2. Composite attribute:

- An attribute that is a combination of other attributes is known as composite attribute.
- For example, In student entity, the student address is a composite attribute as an address is composed of other attributes such as pin code, state, country.



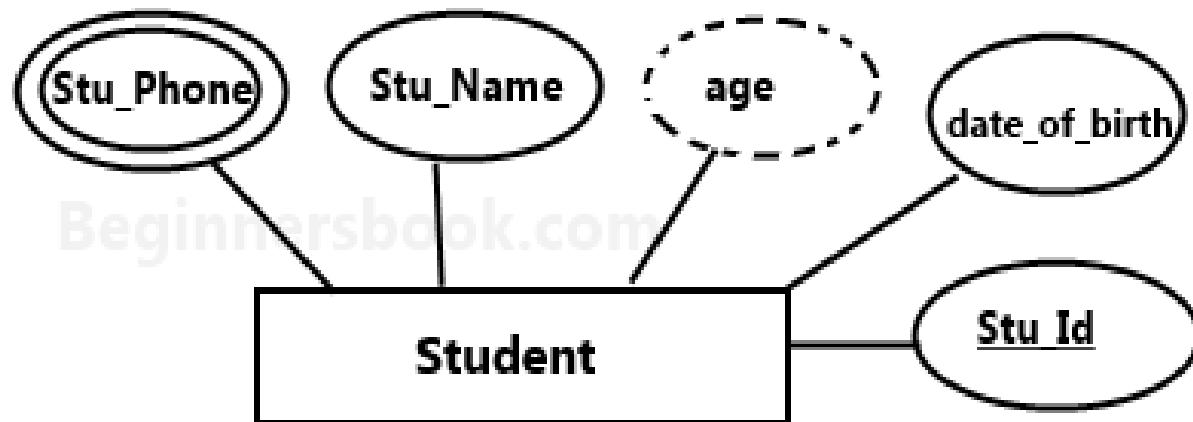
3. Multivalued attribute:

- An attribute that can hold multiple values is known as multivalued attribute.
- It is represented with double ovals in an ER Diagram.
- For example – A person can have more than one phone numbers so the phone number attribute is multivalued.



4. Derived attribute and stored attribute:

- A derived attribute is one whose value is dynamic and derived from another attribute. It is represented by dashed oval in an ER Diagram.
- For example – Person age is a derived attribute as it changes over time and can be derived from another attribute (Date of birth).



Mapping Cardinality / Cardinality Ratio / Cardinality :

Specifies how many instances of an entity relate to how many instance of another entity.

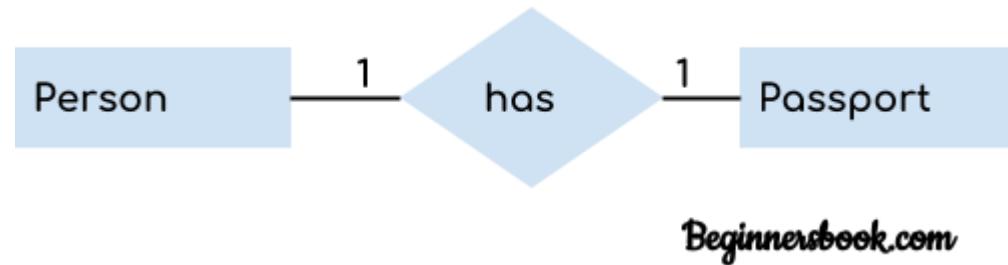
- *There are four types of Mapping Cardinalities:*

1. One to One
2. One to Many
3. Many to One
4. Many to Many

- **One to One (1 : 1):**

When a single instance of an entity is associated with a single instance of another entity then it is called one to one relationship.

For example, a person has only one passport and a passport is given to one person



- **One to Many (1: M)**

When a single instance of an entity is associated with more than one instances of another entity then it is called one to many relationship.

For example – a customer can place many orders but a order cannot be placed by many customers.

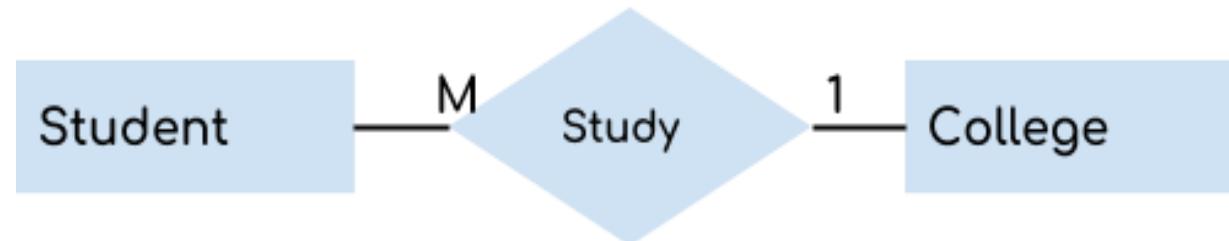


Beginnersbook.com

- Many to (M:1)

When more than one instances of an entity is associated with a single instance of another entity then it is called many to one relationship.

For example – many students can study in a single college but a student cannot study in many colleges at the same time. Many students can register for 1 course



Beginnerbook.com

- **Many to Many (M: N)**

When more than one instances of an entity is associated with more than one instances of another entity then it is called many to many relationship.

For example, a student can be assigned to many projects and a project can be assigned to many students.



Beginnersbook.com

Date _____
Page _____

For each attribute there is a set of permitted values called domain or value of that attribute. [constraint]

Domain of attribute can name might be set of all text strings of certain length.

Domain value: 1 set of an attribute. [data dictionary]

Allowable value of an attribute: datatype, data size constraints.

Example:- ename, if char
 if Integ (no redundancy)
 if size was 20 can't exceed more than 20

constraints [limitations]

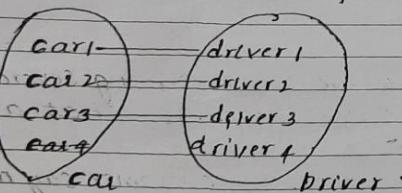
- I Mapping constraints
- II participation constraints.

ordinality defines the no. of entities in an entity set which can be associated with no. of entities in another entity set

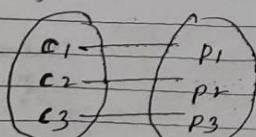
- I Mapping constraints → (on mapping cardinalities express the no. of entities to which another entity can be associated via a relationship)
- (1) one to one.
- (2) one to many
- (3) Many to one.
- (4) Many to Many → Most useful for describing binary relationship

(1) one to one Relationship → for binary relationship R b/w 2 entity sets A & B the mapping cardinalities are

At a particular point of time

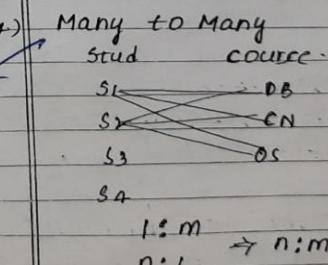
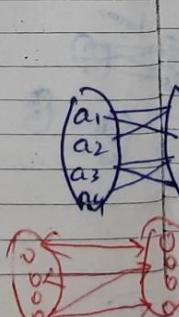


An entity in A is associated with at most one entity in B. An entity in B is associated with at most one entity in A.



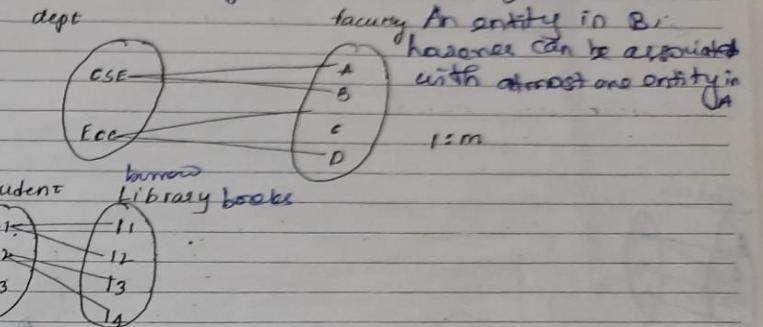
country - president

(3)

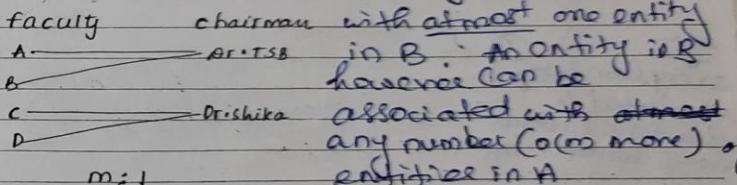


An entity in A is associated with any number (0 or more) of entities in B, and an entity in B is associated with any number (0 or more) of entities in A.

(2) One to Many An entity A is associated with any no (zero or more) of entities in B.



Many to one.



Participation Constraints :

Specifies if all the instances or only few of the instances of an entity are participating in a relationship.

- *There are two types of Participation:*

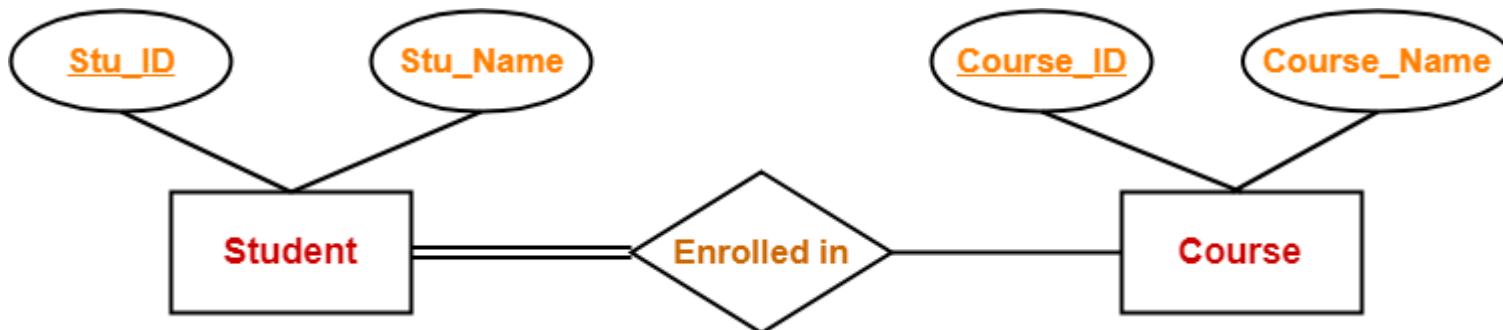
1. Total Participation

2. Partial Participation

Total Participation

A Total participation of an entity set represents that each instance in entity must participate in the relationship

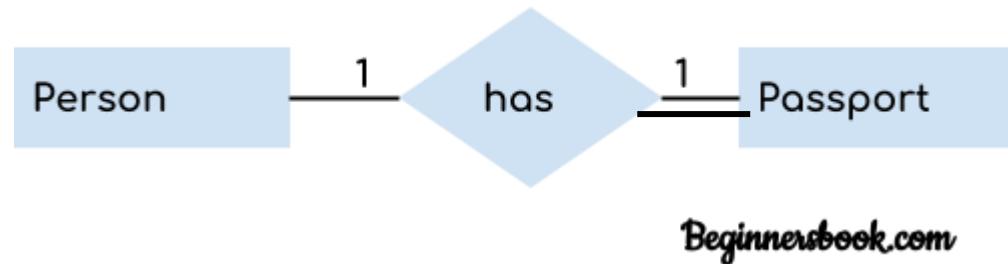
For example: In the below diagram, each student must be enrolled in at least one course. So, total participation. There might exist some courses for which no enrollments are made



Total Participation

A Total participation of an entity set represents that each instance in entity must participate in the relationship

For example: In the below diagram each Passport must belong to a person, hence total participation

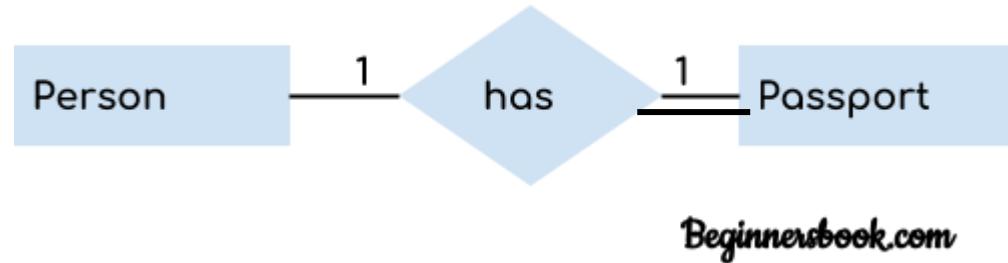


Beginnersbook.com

Partial Participation

A Partial participation of an entity set represents that only few or none of the instances participate in the relationship.

For example: In the below diagram every person may not have a Passport, hence partial participation



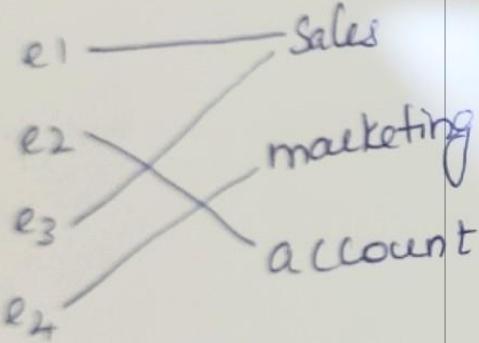
Reset **Save**

...

Total Participation

emp

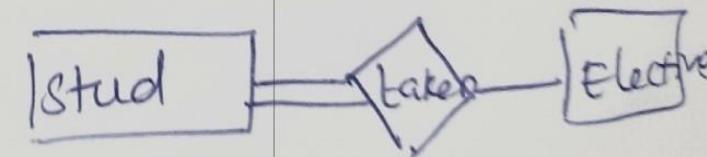
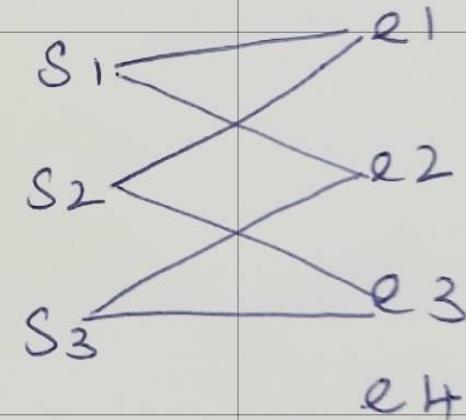
Dept



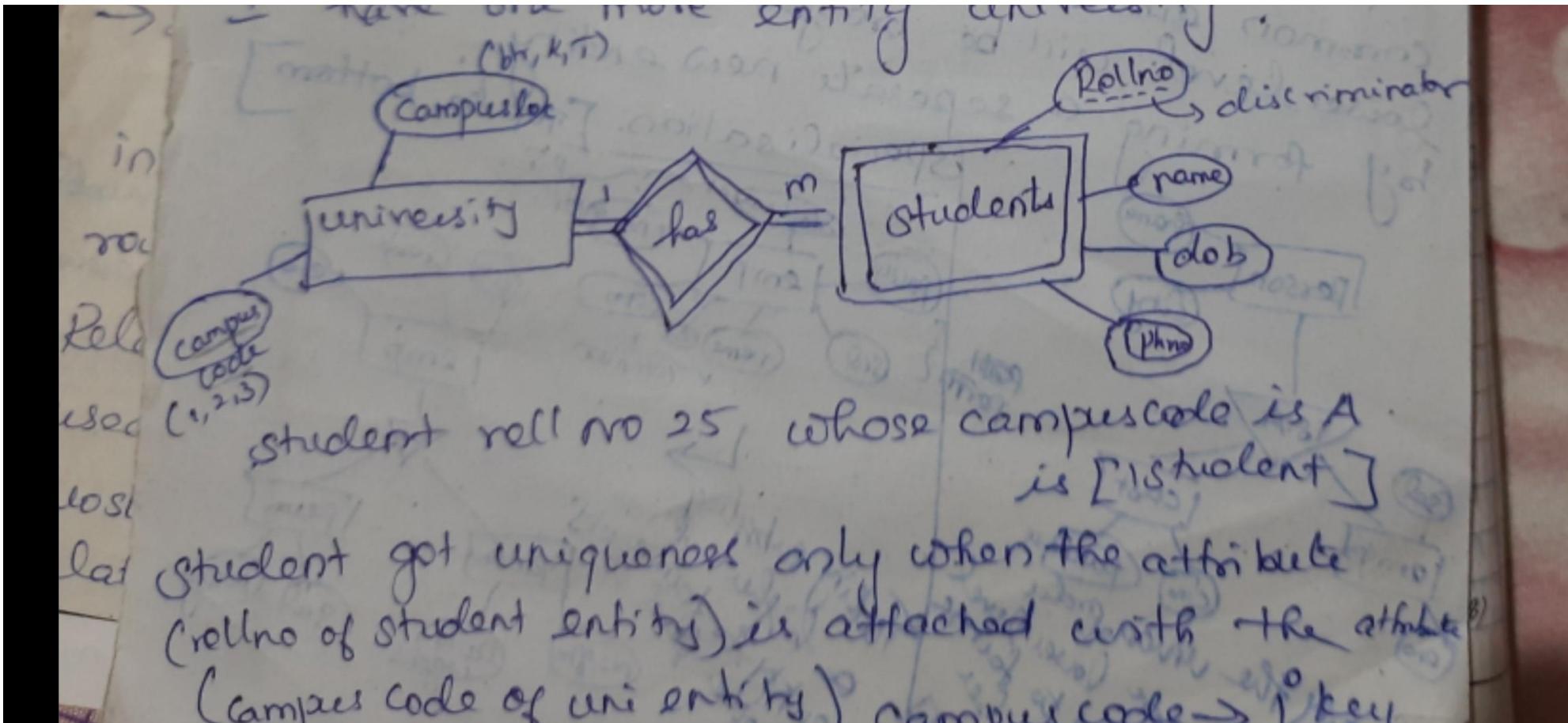
partial participation

stud

Elective

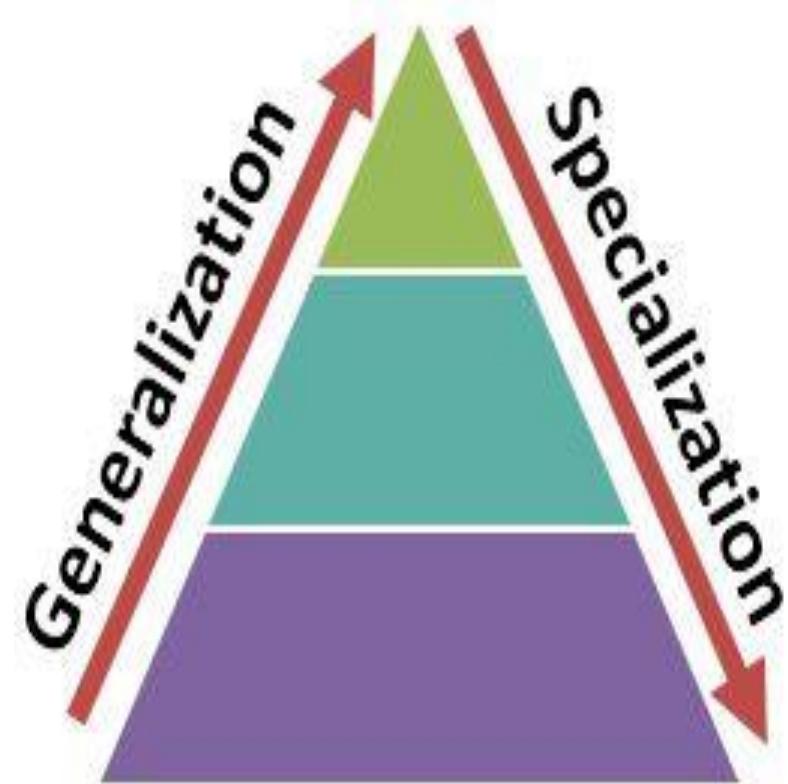


Strong entity and weak entity



Specialization and Generalization:

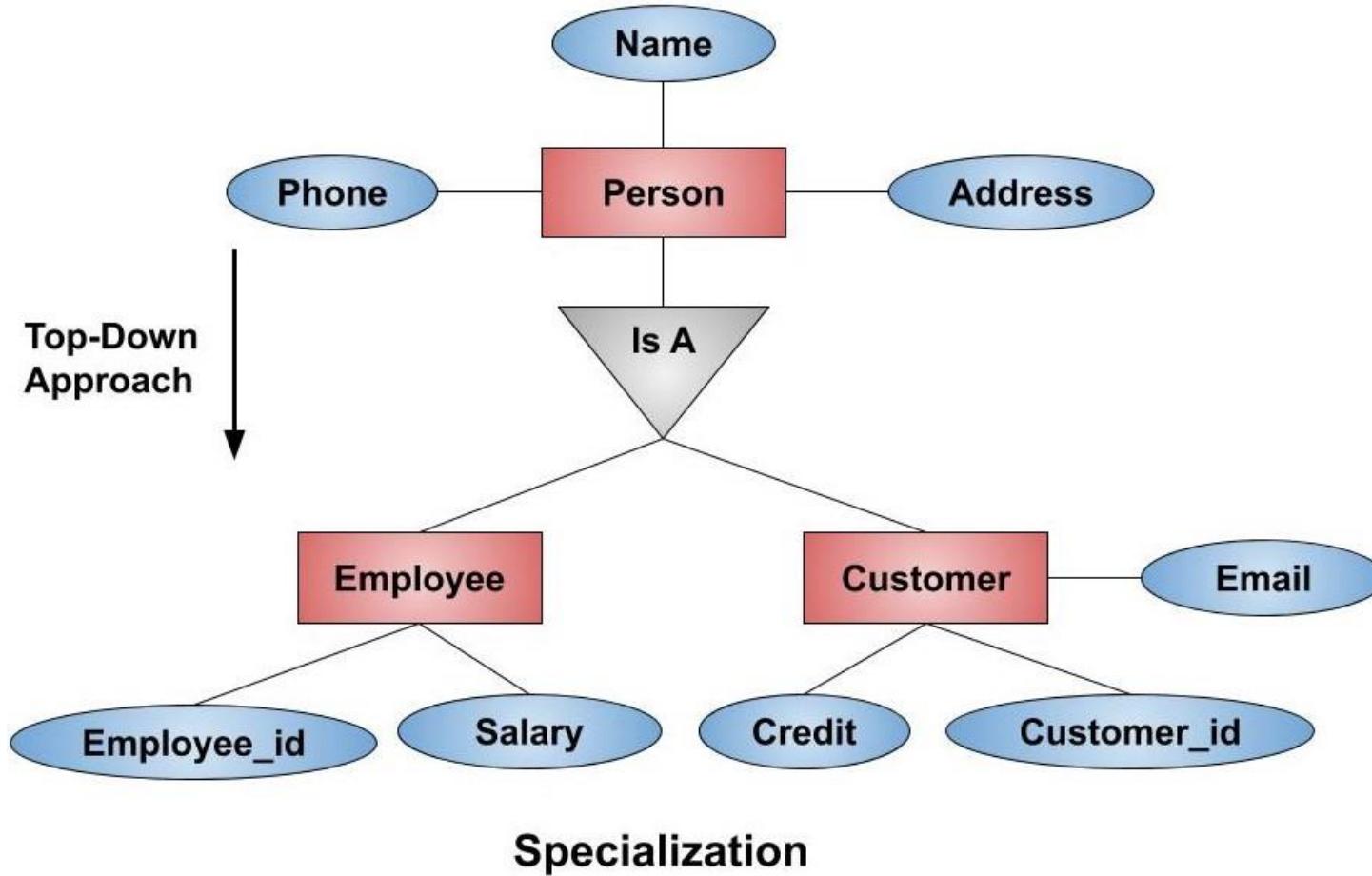
- **Specialization** and **Generalization** are fundamental concepts in database modeling that are useful for establishing superclass-subclass relationships.
- They are extended ER features



Specialization :

- **Specialization** is a **top-down approach** in which a **higher**-level entity is **divided** into **multiple specialized lower**-level entities.
- In addition to sharing the attributes of the higher-level entity, these lower-level entities have *specific* attributes of their own.
- Specialization is usually used to find subsets of an entity that has a few different or additional attributes.

Specialization :

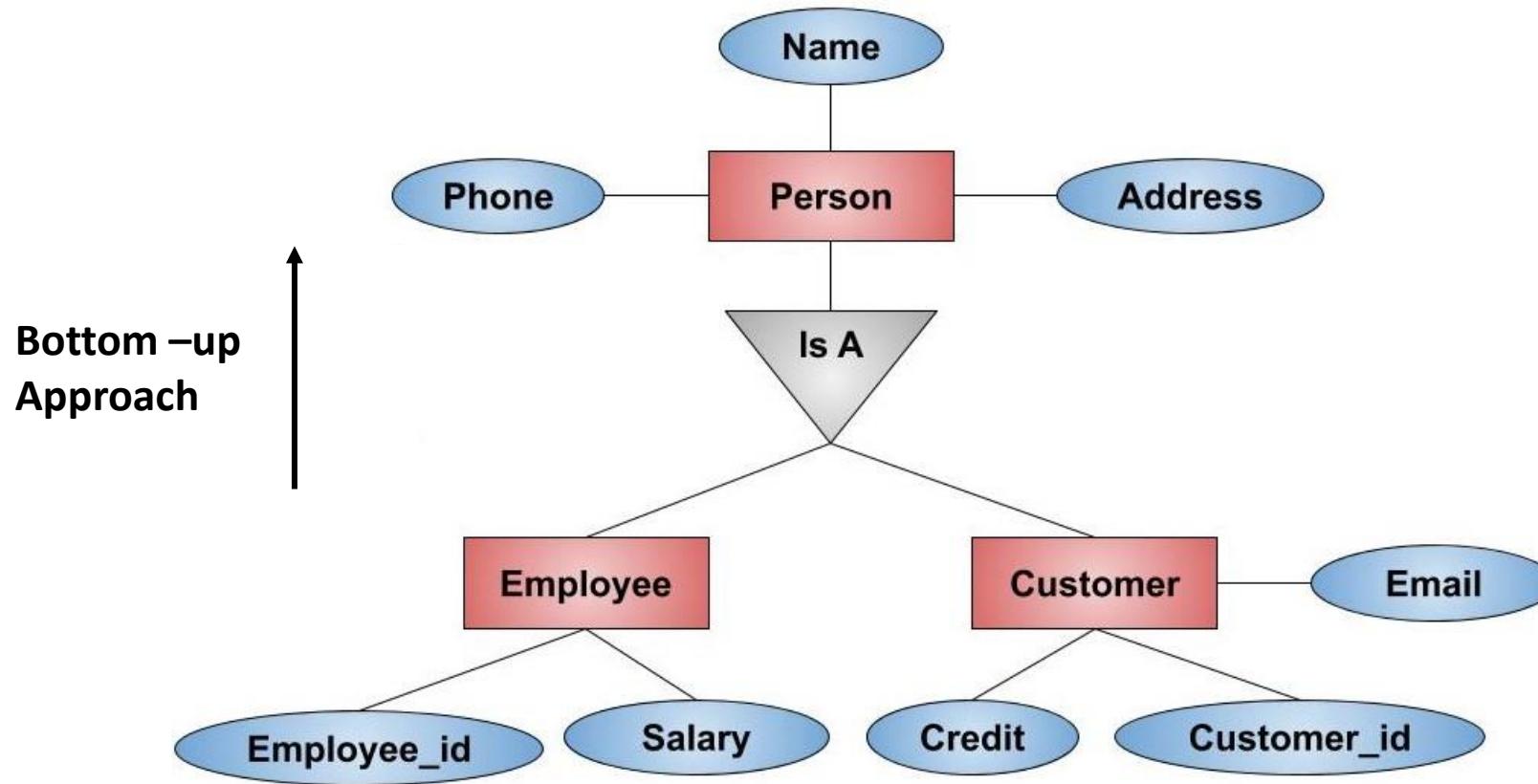


- *Generalization:*

Generalization

- Generalization is a **bottom-up approach** in which **multiple lower-level entities are combined** to form a **single higher-level entity**.
- Generalization is usually used to find common attributes among entities to form a generalized entity. It can also be thought of as the opposite of specialization.
- The following enhanced entity relationship diagram expresses entities in a hierarchical database to demonstrate generalization:

Generalization :



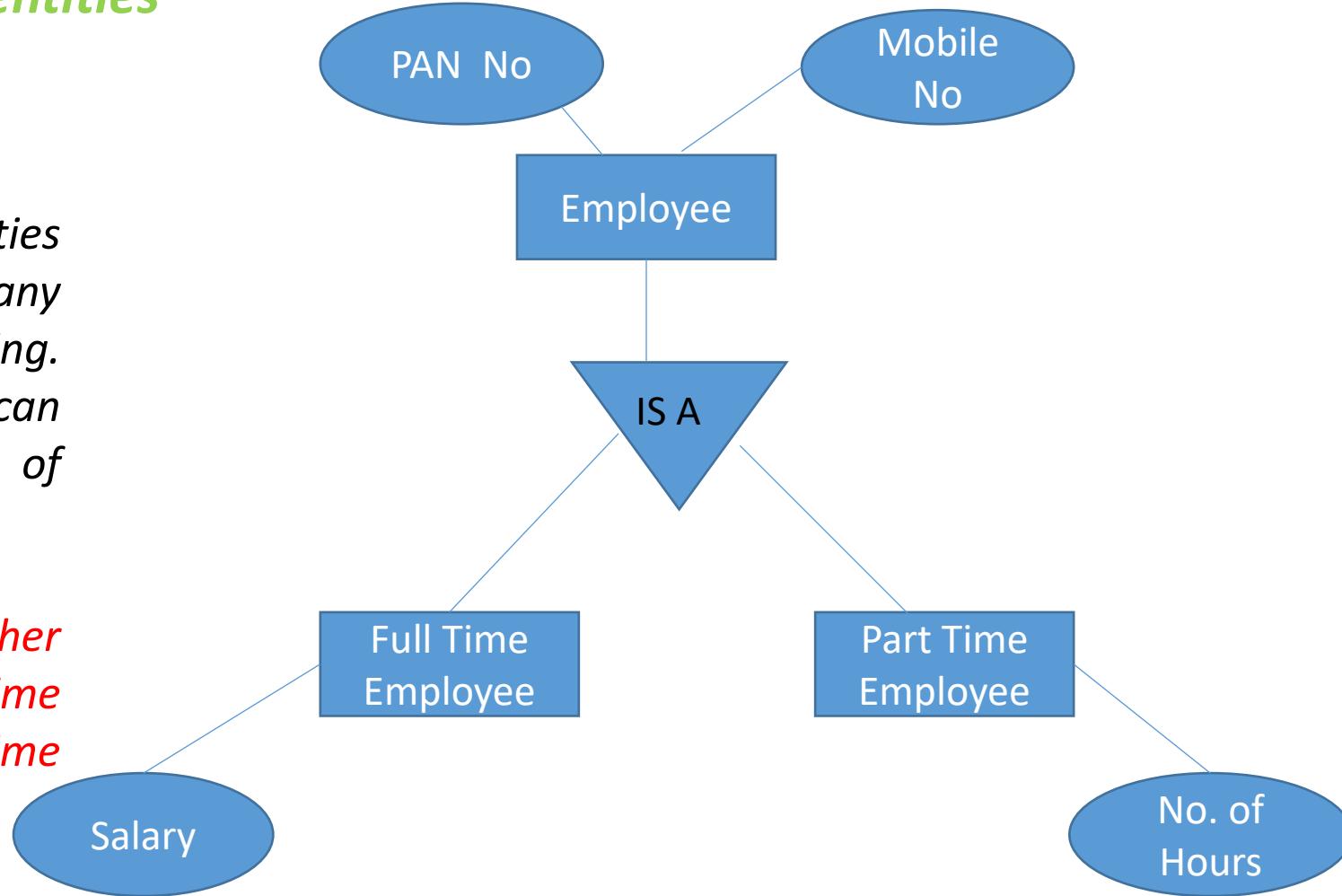
Generalization

Constraints on Generalization / Specialization:

- Condition based Vs User Defined
- Overlapping Vs Disjoint

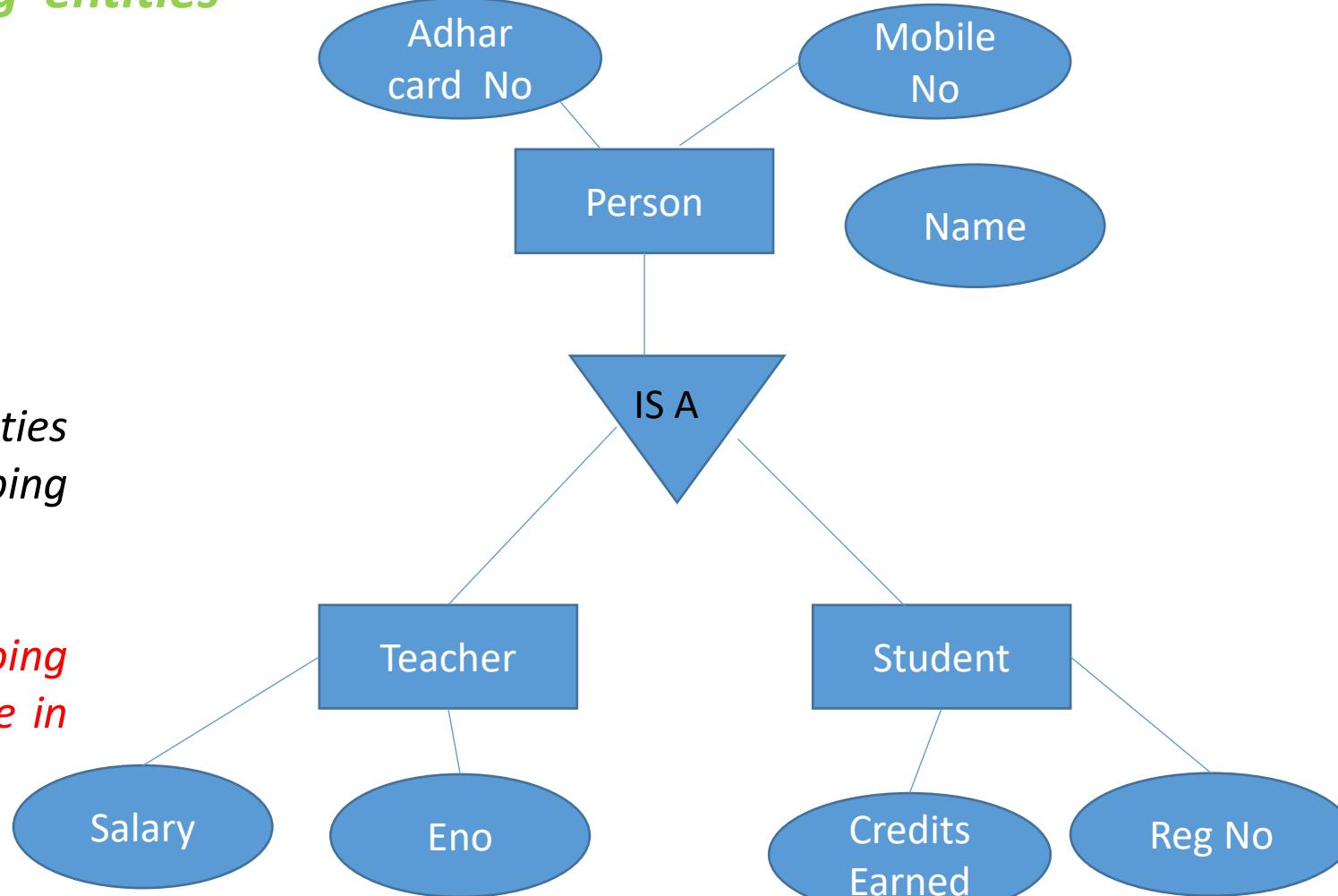
disjoint entities

The *lower level entities* will not have any instances overlapping. i.e., the instance can belong to only one of the lower entities

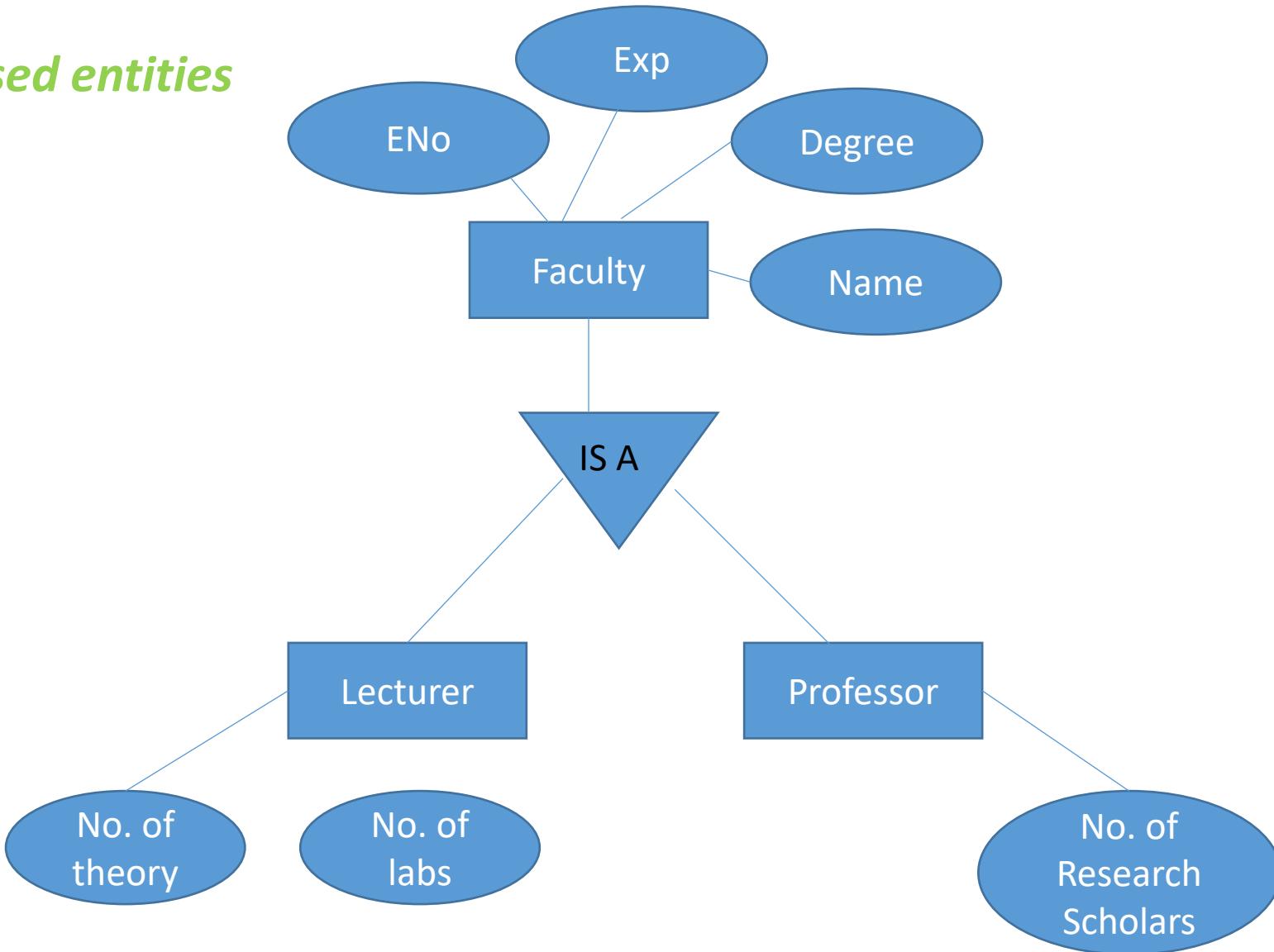


An employee can either be a part time employee or full time employee

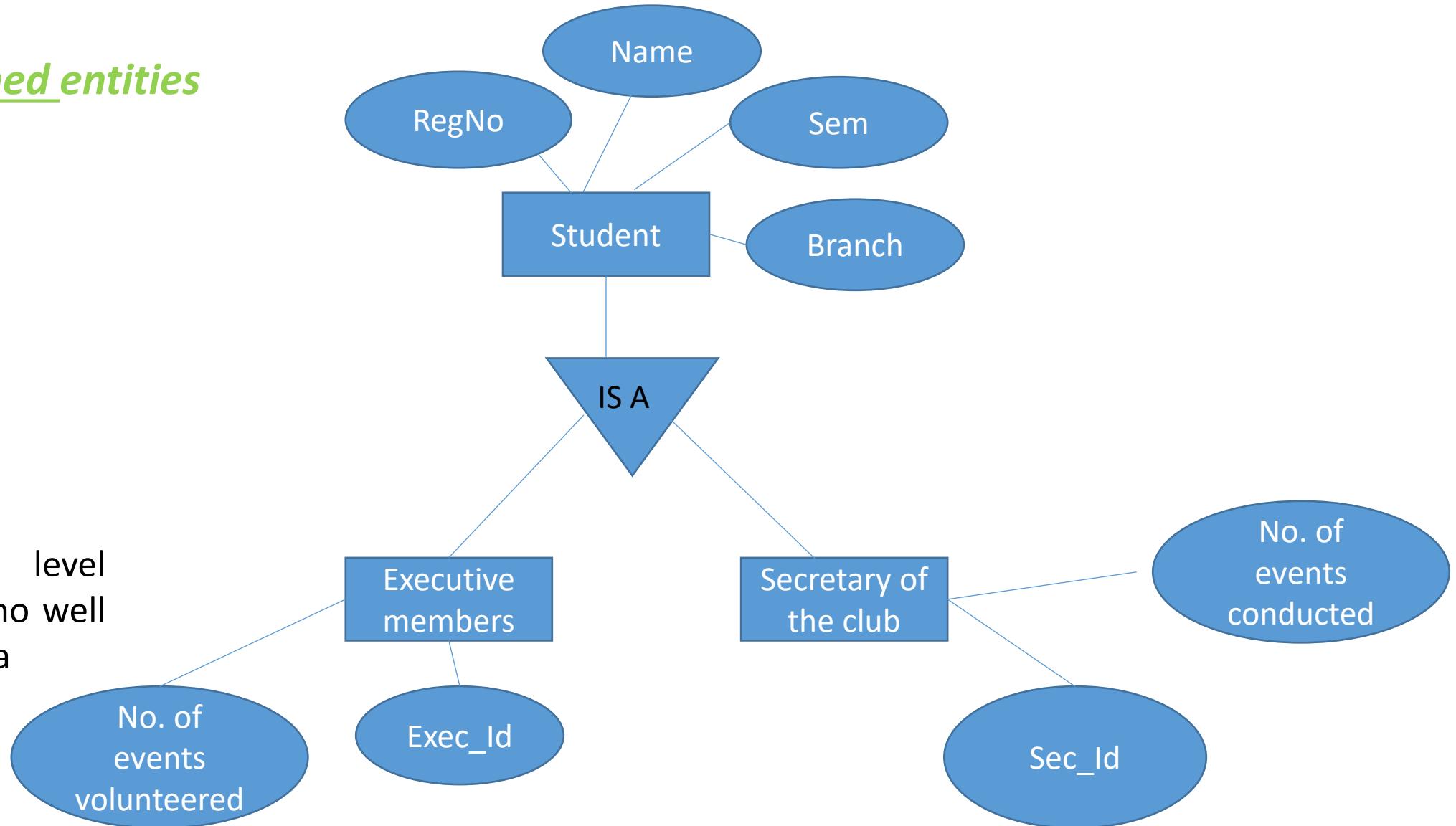
overlapping entities



condition based entities



user defined entities

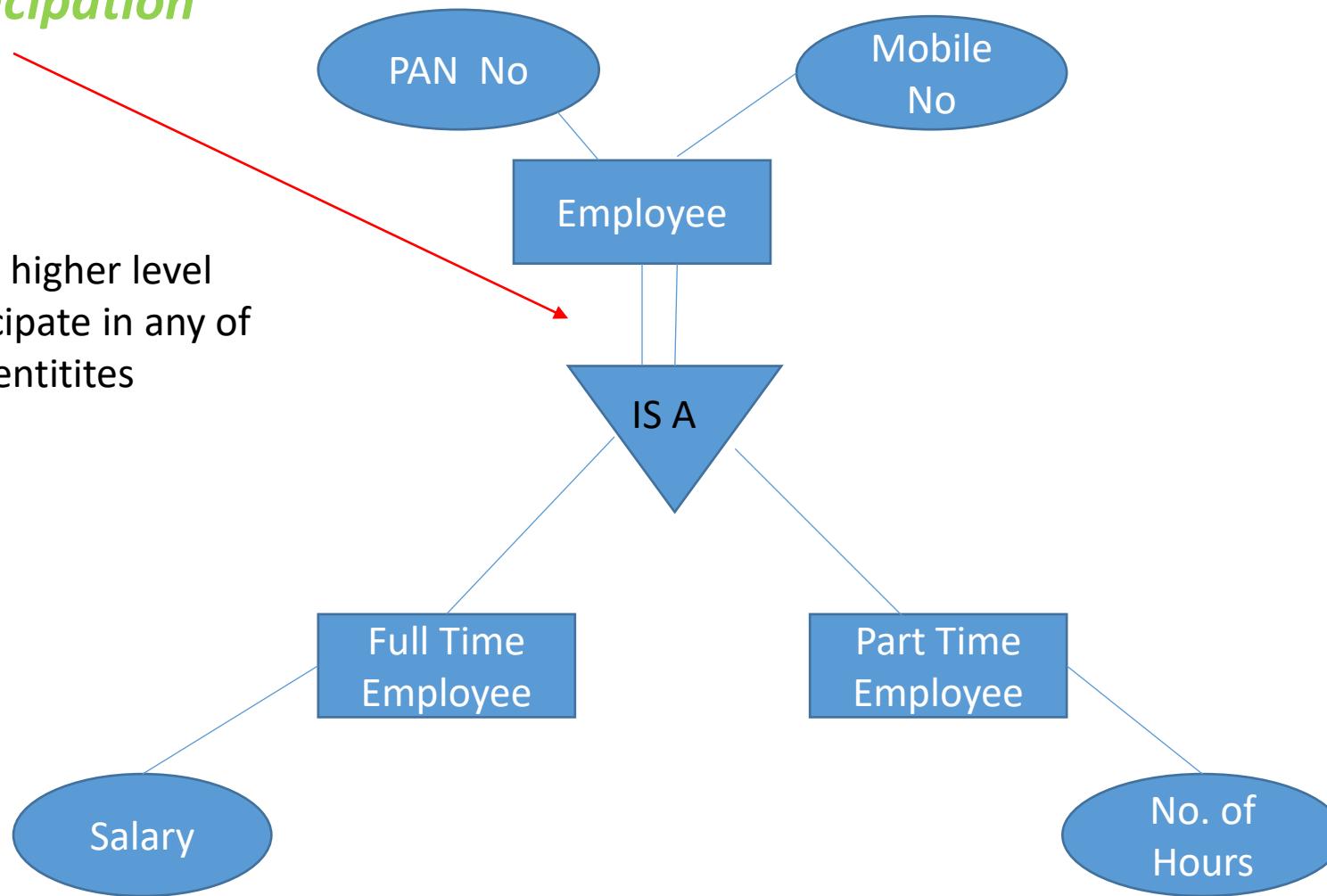


Participation Constraints on Generalization / Specialization:

- Total Participation
- Partial Participation

Total Participation

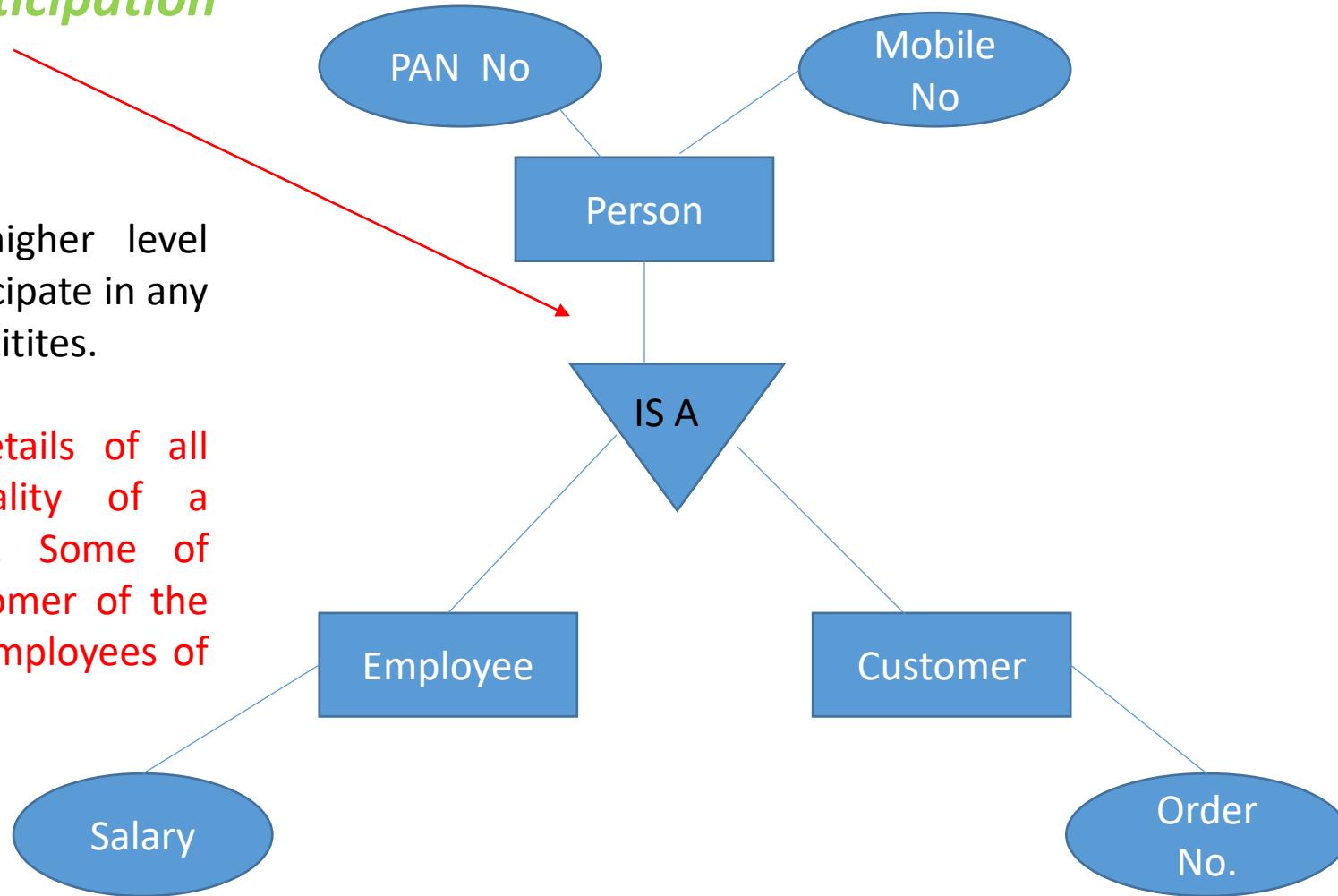
All Entities from higher level entity will participate in any of the lower level entities



Partial Participation

All Entities from higher level entity may not participate in any of the lower level entities.

Ex: I may have details of all people in a locality of a departmental store. Some of whom may be customer of the store, few may be employees of that store



Steps for constructing an ERD:

Identify the entities. The first step in making an ERD is to identify all of the entities. An entity is nothing more than a rectangle with a description of something that your system store information about. This could be a customer, a manager, an invoice, a schedule, etc.

Identify and describe relationships. Look at two entities, are they related? How are the entities related? Draw an action diamond between the two entities on the line you just added. In the diamond write a brief description of how they are related.

Cardinality: Specify the mapping cardinalities between the entities in relationship , also specify the participation constraints.

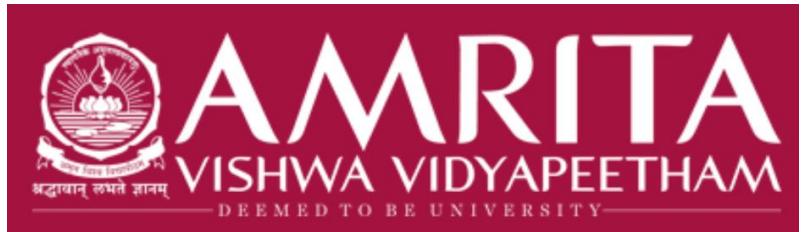
Add attributes. Any key attributes of entities should be added using oval-shaped symbols.

Complete the diagram. Continue to connect the entities with lines, and adding diamonds to describe each relationship until all relationships have been described. Each of your entities may not have any relationships, some may have multiple relationships. That is okay.

How to Draw a complete ER Diagram????

Tips for Effective ER Diagrams

- *Make sure that each entity only appears once per diagram.*
- *Name every entity, relationship, and attribute on your diagram.*
- *Examine relationships between entities closely. Are they necessary?*
- *Are there any relationships missing? Eliminate any redundant relationships.*
- *Don't connect relationships to each other.*



Introduction to SQL

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,

Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

What is SQL?

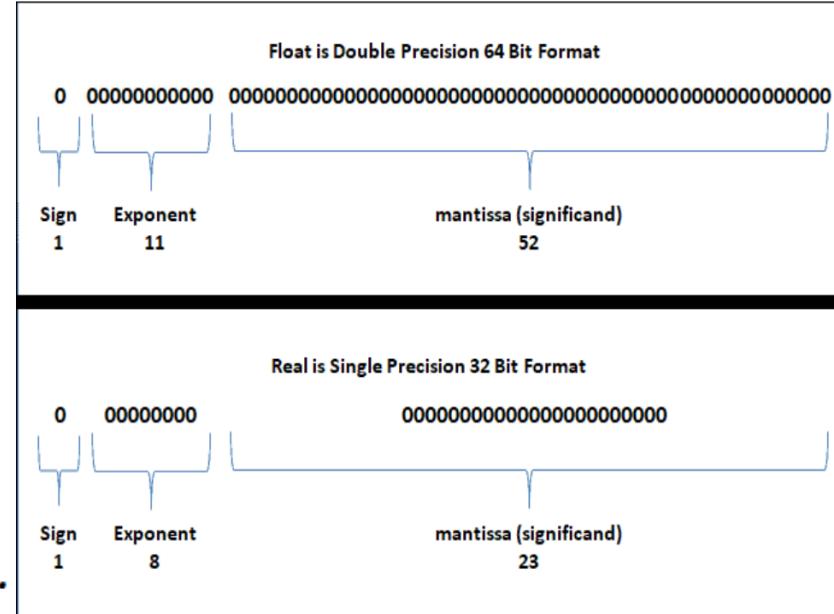
SQL is short for Structured Query Language. Originally, it used to be called SEQUEL (Structured English Query Language) and was used for storing and manipulating data in databases. Today SQL is used to perform all types of data operations in relational database management systems (RDBMS).

SQL is a powerful language where we can perform a wide range of operations:

- execute queries
- fetch data
- insert, update, and delete records in a database (DML operations)
- create new objects in a database (DDL operations)
- set permissions on tables, procedures, functions, and views
- and much, much more...

Domain types in SQL

- **char(*n*)**. Fixed length character string, with user-specified length *n*.
- **varchar(*n*)**. Variable length character strings, with user-specified maximum length *n*.
- **int**. Integer (a finite subset of the integers that is machine-dependent).
- **smallint**. Small integer (a machine-dependent subset of the integer domain type).
- **numeric(*p,d*)**. Fixed point number, with user-specified precision of *p* digits, with *d* digits to the right of decimal point. (ex., **numeric(3,1)**, allows 44.5 to be stored exactly, but not 444.5 or 0.32)
- **real, double precision**. Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(*n*)**. Floating point number, with user-specified precision of at least *n* digits.



Install PostgreSQL on Windows and Ubuntu

(Document uploaded in sharepoint)

Testing the installation

- Command to check version of installation – `select version();`
- To display all databases - `\l` (meaning - **back slash l**)
- Create new database -
`create database sql_demo;`
- Connect to database
`\c sql_demo`

SQL Query examples

CREATE TABLE query

CREATE TABLE is a keyword that will create a new, initially empty table in the database. The table will be owned by the user who has issued this command.

```
postgres=# create table dummy_table(name varchar(20),address text,age int);  
CREATE TABLE
```

INSERT query

The [INSERT](#) command is used to insert data into a table:

```
postgres=# insert into dummy_table values('XYZ','location-A',25);
INSERT 0 1
postgres=# insert into dummy_table values('ABC','location-B',35);
INSERT 0 1
postgres=# insert into dummy_table values('DEF','location-C',40);
INSERT 0 1
postgres=# insert into dummy_table values('PQR','location-D',54);
INSERT 0 1
```

SELECT query without WHERE condition

The [SELECT command](#) (when used without the optional WHERE condition) is used to fetch all data from a database table:

```
postgres=# select * from dummy_table;
 name | address | age
-----+-----+ -----
 XYZ  | location-A | 25
 ABC  | location-B | 35
 DEF  | location-C | 40
 PQR  | location-D | 54
(4 rows)
```

UPDATE query

UPDATE is used to make updates to the data or row(s) of a database table. In the example below we use UPDATE to change the age of a person whose name is ‘PQR’:

```
postgres=# update dummy_table set age=50 where name='PQR';
UPDATE 1
postgres=# select * from dummy_table;
name | address | age
-----+-----+-----
XYZ  | location-A | 25
ABC  | location-B | 35
DEF  | location-C | 40
PQR  | location-D | 50
(4 rows)
```

Q) Change the name and age of a person whose address is 'location-D'

```
postgres=# update dummy_table set name='GHI',age=54 where address='location-D';
UPDATE 1
postgres=# select * from dummy_table;
   name  |  address   | age
-----+-----+-----
  XYZ  | location-A | 25
  ABC  | location-B | 35
  DEF  | location-C | 40
  GHI  | location-D | 54
(4 rows)

postgres=#

```

If we want to modify all the values in the address and age columns in dummy_table, then we do not need to use the WHERE clause. The UPDATE query would look like this:

```
postgres=# update dummy_table set age=54,address='location-X';
UPDATE 4

postgres=# select * from dummy_table ;
   name  |  address  |  age
-----+-----+-----
  XYZ  | location-X |  54
  ABC  | location-X |  54
  DEF  | location-X |  54
  GHI  | location-X |  54
(4 rows)

postgres=#

```

A **RETURNING** clause returns the updated rows. This is optional in UPDATE:

```
postgres=# update dummy_table set age=30 where name='XYZ' returning age as age_no;  
age_no  
-----  
     30  
(1 row)  
  
UPDATE 1
```

DELETE query

The `DELETE` command is used to delete row(s). It can be used with or without the optional `WHERE` condition, but take note: if the `WHERE` condition is missing, the command will delete all rows, leaving you with an empty table.

In this example, we are deleting one row whose age column has the value 65:

```
postgres=# delete from dummy_table where age=65;  
DELETE 1  
postgres=#
```

Comparison Operators

In PostgreSQL, with the help of **comparison operators** we can find results where the value in a column is not equal to the specified condition or value.

Less than or equal to query:

```
postgres=# select * from dummy_table where age <=50;
 name | address | age
-----+-----+-----
 XYZ  | location-A | 25
 ABC  | location-B | 35
 DEF  | location-C | 40
 PQR  | location-D | 50
(4 rows)
```

Greater than or equal to query:

```
postgres=# select * from dummy_table where age>=50;
 name | address | age
-----+-----+-----
 PQR  | location-D | 50
(1 row)
```

Not equal to query:

```
postgres=# select * from dummy_table where age<>50;
 name | address | age
-----+-----+-----
 XYZ  | location-A | 25
 ABC  | location-B | 35
 DEF  | location-C | 40
(3 rows)
```

Equal to query:

```
postgres=# select * from dummy_table where age=50;
 name | address | age
-----+-----+-----
 PQR  | location-D | 50
(1 row)
```

SELECT DISTINCT query

- The SQL **DISTINCT** keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.
- There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only those unique records instead of fetching duplicate records.

Syntax

The basic syntax of DISTINCT keyword to eliminate the duplicate records is as follows –

```
SELECT DISTINCT column1, column2,.....columnN  
FROM table_name  
WHERE [condition]
```

Example

Consider the CUSTOMERS table having the following records –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

```
SELECT SALARY FROM CUSTOMERS  
ORDER BY SALARY;
```

```
SELECT SALARY FROM CUSTOMERS  
ORDER BY SALARY;
```

SALARY
1500.00
2000.00
2000.00
4500.00
6500.00
8500.00
10000.00

```
SQL> SELECT DISTINCT SALARY FROM CUSTOMERS  
      ORDER BY SALARY;
```

This would produce the following result where we do not have any duplicate entry.

SALARY
1500.00
2000.00
4500.00
6500.00
8500.00
10000.00

TRUNCATE query

The [TRUNCATE command](#) is used to empty a table:

```
postgres=# truncate table dummy_table;  
TRUNCATE TABLE
```

DROP query

Syntax

The basic syntax of this DROP TABLE statement is as follows –

```
DROP TABLE table_name;
```

Syntax

The basic syntax of DROP DATABASE statement is as follows –

```
DROP DATABASE DatabaseName;
```

Always the database name should be unique within the RDBMS.

Exporting query result to a text file

With the help of the [COPY command](#), we can export data from a table to an outside text file as well as import data from a text file into a table.

Exporting data from a table to a text file

```
postgres=# copy dummy_table to '/tmp/abc.txt';
COPY 5
```

```
postgres=# \! cat /tmp/abc.txt
XYZ      location-A      25
ABC      location-B      35
DEF      location-C      40
PQR      location-D      50
CXC      1              50
```

Importing data from a text file into a table

```
postgres=# copy dummy_table from '/tmp/abc.txt';
COPY 5
```

EmployeeID	EmployeeName	Emergency ContactName	PhoneNumber	Address	City	Country
01	Shanaya	Abhinay	9898765612	Oberoi Street 23	Mumbai	India
02	Anay	Soumya	9432156783	Marathalli House No 23	Delhi	India
03	Preeti	Rohan	9764234519	Queens Road 45	Bangalore	India
04	Vihaan	Akriti	9966442211	Brigade Road Block 4	Hyderabad	India
05	Manasa	Shourya	9543176246	Mayo Road 23	Kolkata	India



Database Design using ER Model –Part 1

Design Phases



- **Initial Phase**
 - characterize fully the data needs of the prospective database users.
 - output- User requirements specification.
- **Conceptual Design Phase**
 - Chooses a data model
 - Applying the concepts of the chosen data model for translating these requirements into a conceptual schema of the database.
 - A fully developed conceptual schema indicates the functional requirements of the enterprise.
 - Describe the kinds of operations (or transactions) that will be performed on the data.

Design Process



- Final Phase -- Moving from an abstract data model to the implementation of the database
- Logical Design – Deciding on the database schema.
 - The designer maps the high level conceptual schema into relational schema.
- Physical Design – Deciding on the physical layout of the database
 - Form of file organization, choice of index structures etc.

Design Alternatives



- In designing a database schema, we must ensure that we avoid two major pitfalls:
 - Redundancy: a bad design may result in repeat information.
 - Redundant representation of information may lead to data inconsistency among the various copies of information
 - Incompleteness: a bad design may make certain aspects of the enterprise difficult or impossible to model.

ER Model



- Widely used conceptual level data model
 - proposed by Peter P Chen in 1970s
- Data model to describe the database system at the requirements collection stage.
- The ER data model employs three basic concepts:
 - entity ,
 - relationship ,
 - attributes.
- The ER model also has an associated diagrammatic representation, the **ER diagram**, which can express the overall logical structure of a database graphically.

Entity



- An **entity** is an object that exists and is distinguishable from other objects.
 - Example: specific person, company, event, plant
- An **entity set** is a set of entities of the same type that share the same properties.
 - Example: set of all persons, companies, trees, holidays

76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

student

Entity Sets -- *instructor* and *student*

Attributes



- Each entity is described by a set of attributes; i.e., descriptive properties possessed by all members of an entity set.
 - Example:
 - $\text{instructor} = (\text{ID}, \text{name}, \text{salary})$
 $\text{course} = (\text{course_id}, \text{title}, \text{credits})$
- A subset of the attributes form a **primary key** of the entity set; i.e., uniquely identifying each member of the set.

Types of Attributes



- Simple Attributes

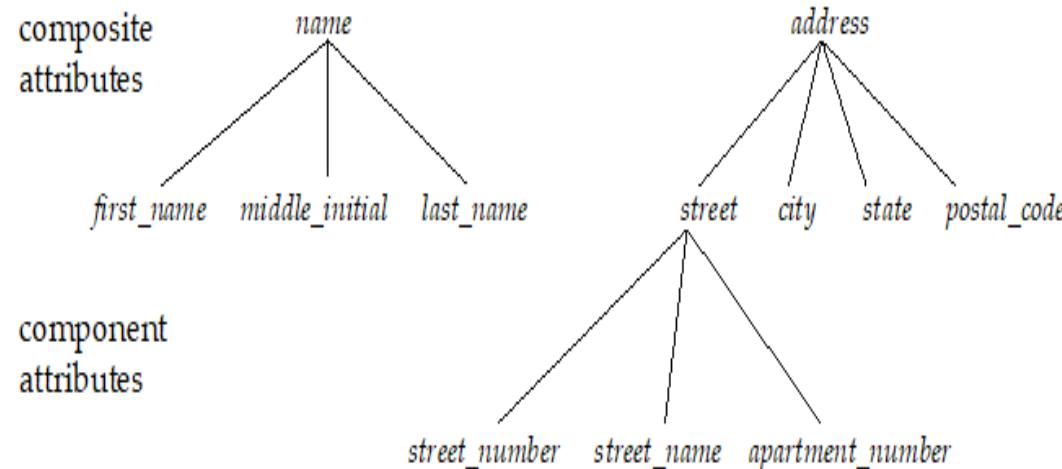
having atomic or indivisible values.

example: Dept—a string, PhoneNumber—an eight digit number

- Composite Attributes

having several components in the value.

example:



Types of Attributes



- Single-valued

having only one value rather than a set of values.
for instance, PlaceOfBirth—single string value.

- Multi-valued

having a set of values rather than a single value.
for instance, CoursesEnrolled attribute for student
EmailAddress attribute for student

- Derived Attributes

- Attribute value is dependent on some other attribute.
- example: Age depends on DateOfBirth. So age is a derived attribute.

Relationships



- A **relationship** is an association among several entities

- Example:

44553 (Peltier)

advisor

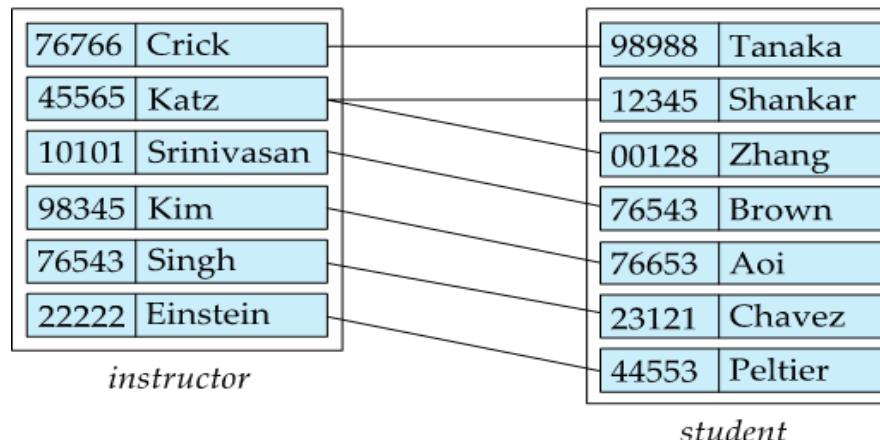
22222 (Einstein)

student entity

relationship set

instructor entity

- we define the relationship set *advisor* to denote the associations between students and the instructors who act as their advisors.



Relationships

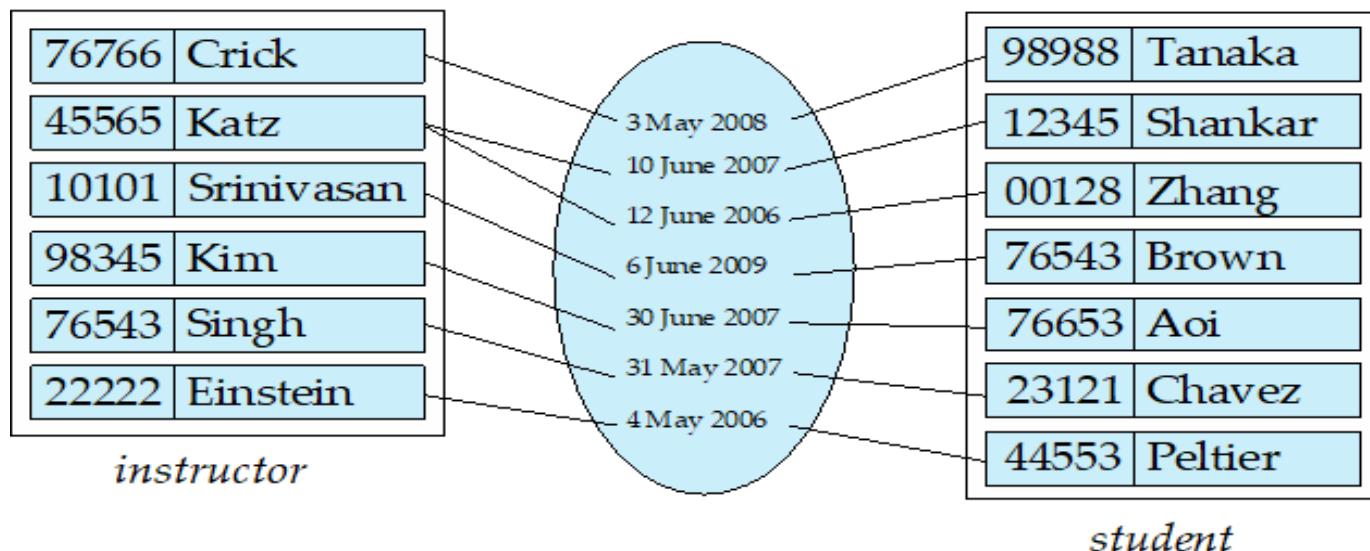


- A **relationship set** is a mathematical relation among more than two or more entities, each taken from entity sets.
- **Degree of a relationship**
 - the number of participating entities.
 - Degree 2: binary
 - Degree 3: ternary
 - Degree n: n-ary
 - Binary relationships are very common and widely used.

Relationship Sets



- An attribute can also be associated with a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have the attribute *date* which tracks when the student started being associated with the advisor

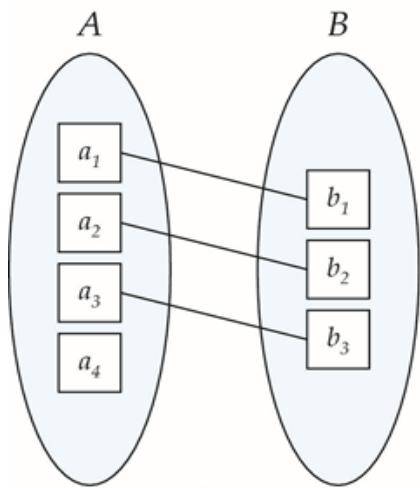


Mapping Cardinality Constraints (Cardinality ratio)

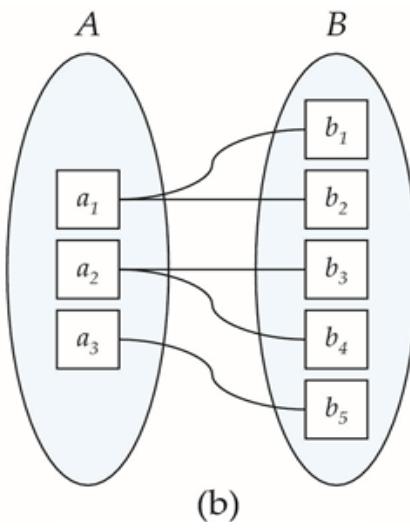


- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
 - One to one
 - One to many
 - Many to one
 - Many to many

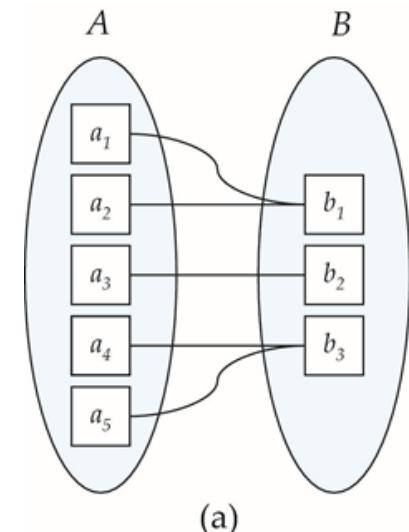
Mapping Cardinalities



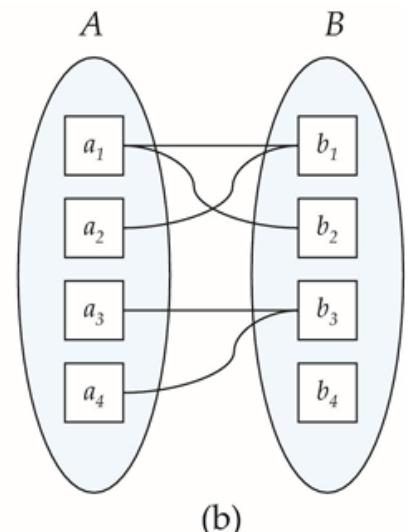
One to one



One to many



Many to one



Many to many

Participation Constraints



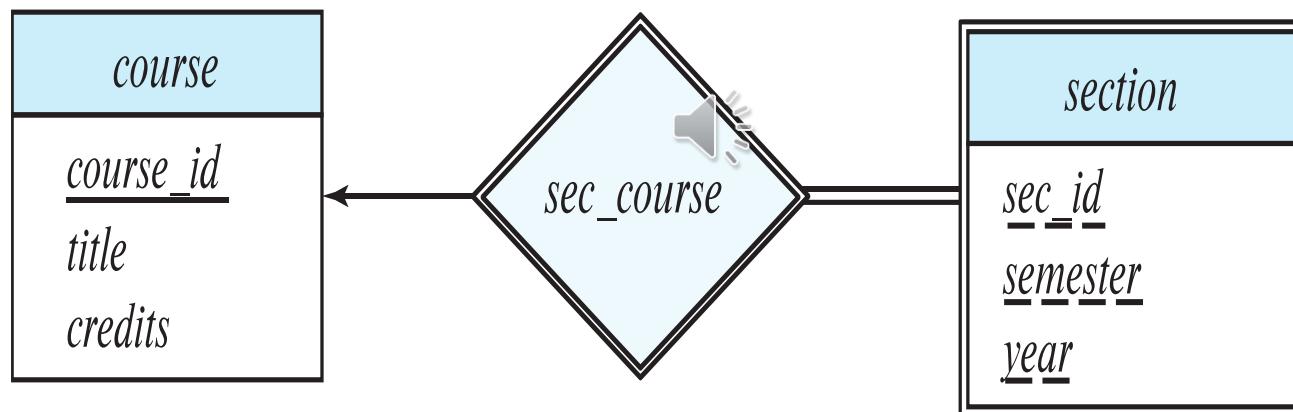
- **Total participation** : every entity in the entity set participates in at least one relationship in the relationship set.
 - participation of *student* in *advisor* relation is total
 - every *student* must have an associated instructor
- **Partial participation**: some entities may not participate in any relationship in the relationship set
 - Example: participation of *instructor* in *advisor* is partial

Weak Entity Sets



- A **weak entity set** is one whose existence is dependent on another entity, called its **identifying entity**
- Instead of associating a primary key with a weak entity, we use the identifying entity, along with extra attributes called **discriminator** to uniquely identify a weak entity.
- An entity set that is not a weak entity set is termed a **strong entity set**.
- Every weak entity must be associated with an identifying entity; that is, the weak entity set is said to be **existence dependent** on the identifying entity set.
- The identifying entity set is said to **own** the weak entity set that it identifies.
- The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**.

Weak Entity- Example



Database Design using ER Model –Part 2

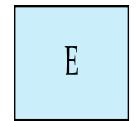


ER Diagrams

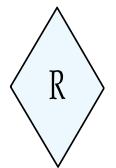
- Can express the overall logical structure of a database graphically.



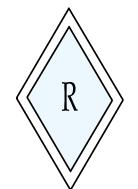
Symbols Used in E-R Notation



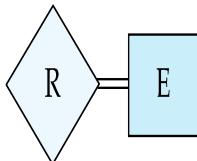
entity set



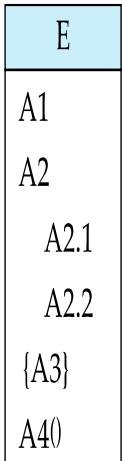
relationship set



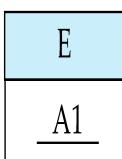
identifying
relationship set
for weak entity set



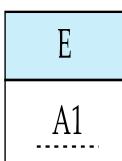
total participation
of entity set in
relationship



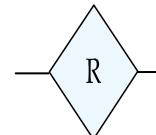
attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)



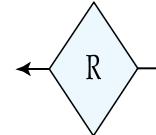
primary key



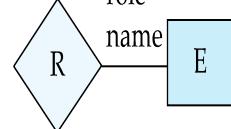
discriminating
attribute of
weak entity set



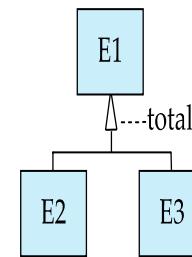
many-to-many
relationship



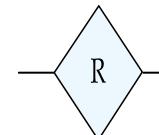
one-to-one
relationship



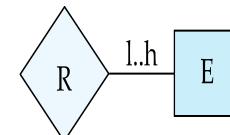
role indicator



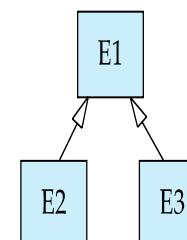
total (disjoint)
generalization



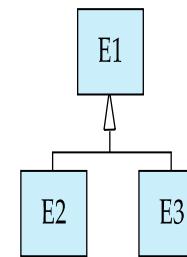
many-to-one
relationship



cardinality
limits



ISA: generalization
or specialization



disjoint
generalization

Representing Entity sets in ER Diagram

- Entity sets can be represented graphically as follows:
 - Rectangles represent entity sets.
 - Attributes listed inside entity rectangle
 - Underline indicates primary key attributes



<i>instructor</i>
<u>ID</u>
<i>name</i>
<i>salary</i>

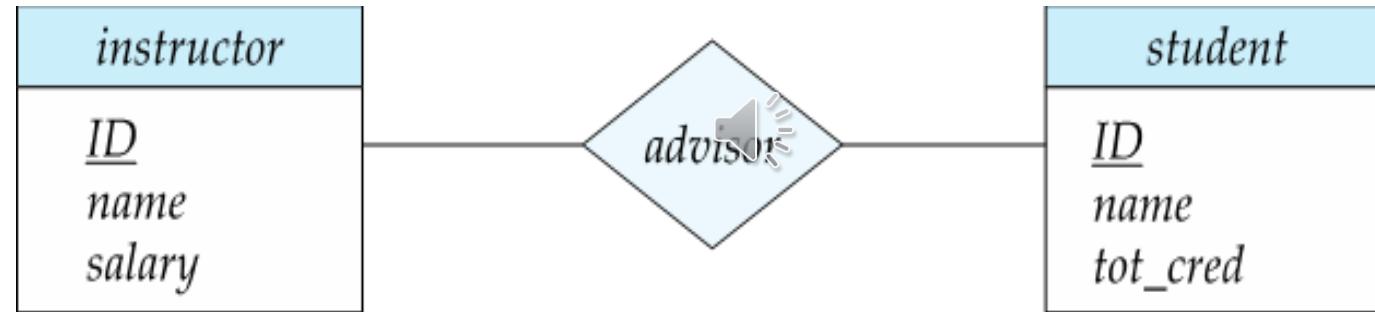
<i>student</i>
<u>ID</u>
<i>name</i>
<i>tot_cred</i>

Representing Complex Attributes in ER Diagram

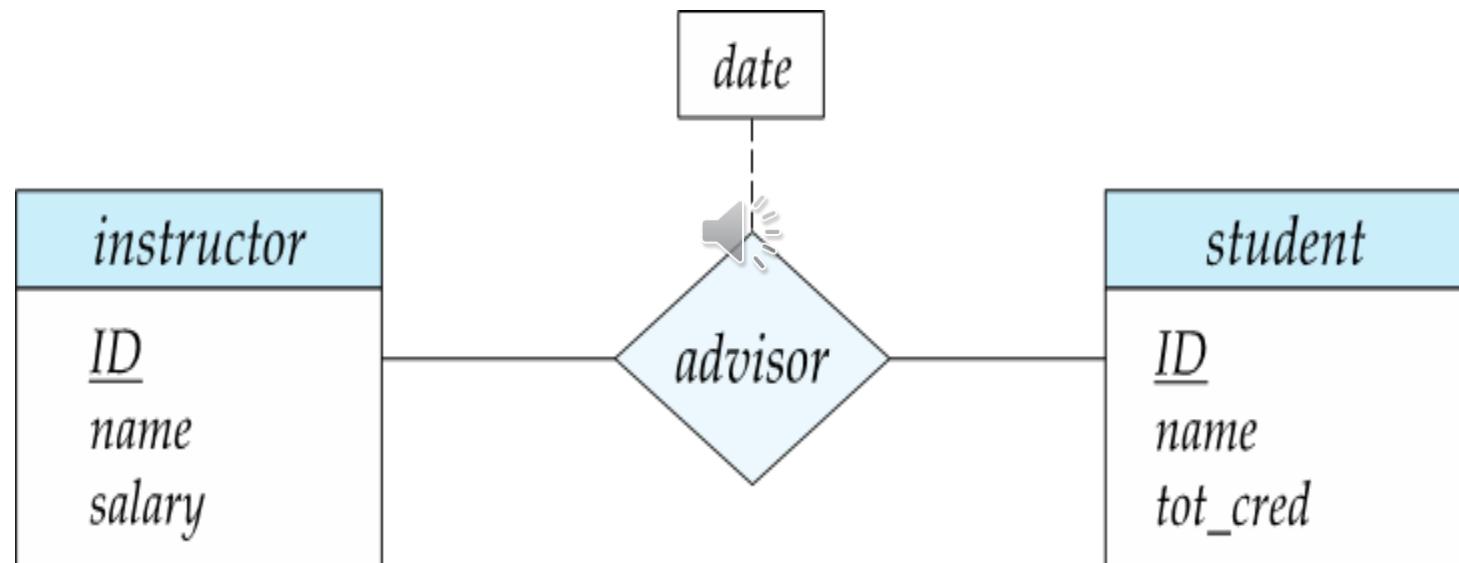
<i>instructor</i>	
<u><i>ID</i></u>	
<i>name</i>	
<i>first_name</i>	
<i>middle_initial</i>	
<i>last_name</i>	
<i>address</i>	🔊
<i>street</i>	
<i>street_number</i>	
<i>street_name</i>	
<i>apt_number</i>	
<i>city</i>	
<i>state</i>	
<i>zip</i>	
{ <i>phone_number</i> }	
<i>date_of_birth</i>	
<i>age ()</i>	

Representing Relationship Sets via ER Diagrams

- Diamonds represent relationship sets.

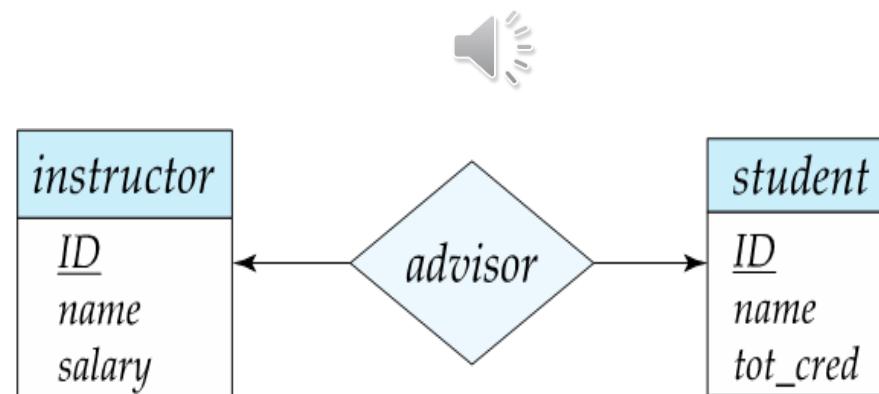


Relationship Sets with Attributes



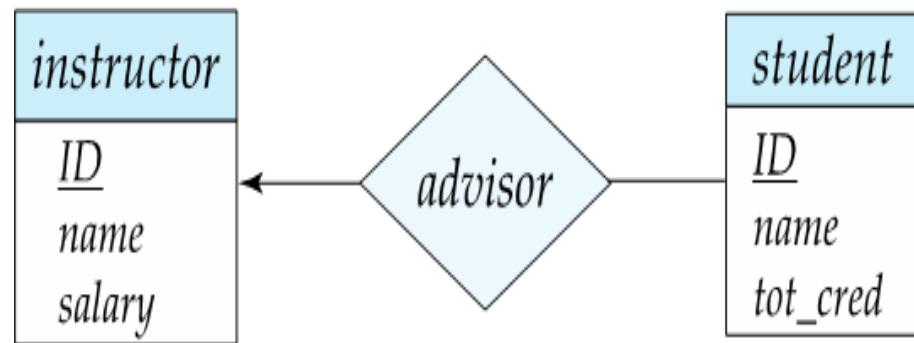
Representing Cardinality Constraints in ER Diagram

- We express cardinality constraints by drawing either a directed line (\rightarrow), signifying “one,” or an undirected line ($-$), signifying “many,” between the relationship set and the entity set.
- One-to-one relationship between an *instructor* and a *student* :
 - A student is associated with at most one *instructor* via the relationship *advisor*



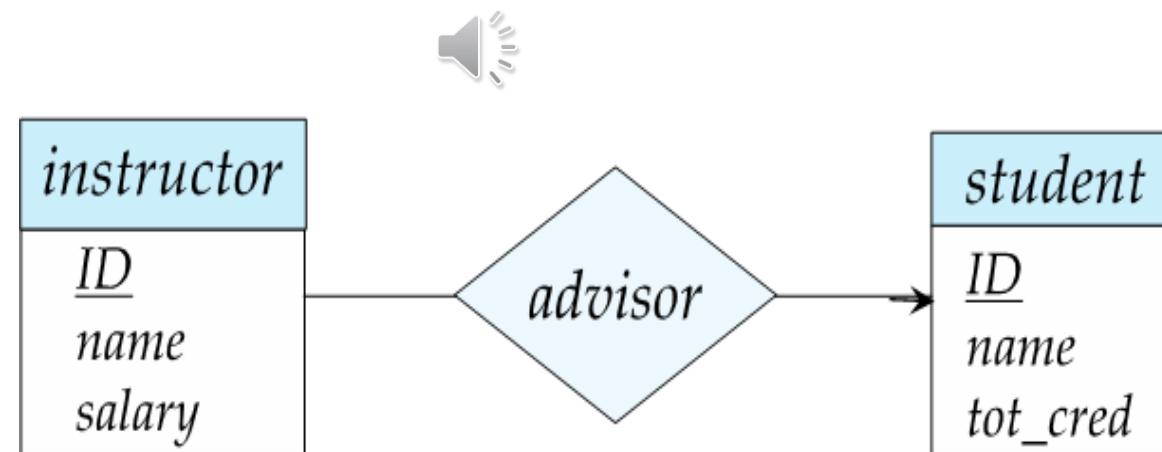
One-to-Many Relationship

- one-to-many relationship between an *instructor* and a *student*
 - an *instructor* is associated with several (including 0) *students* via *advisor*
 - a *student* is associated with ~~at~~ *almost one* *instructor* via *advisor*,



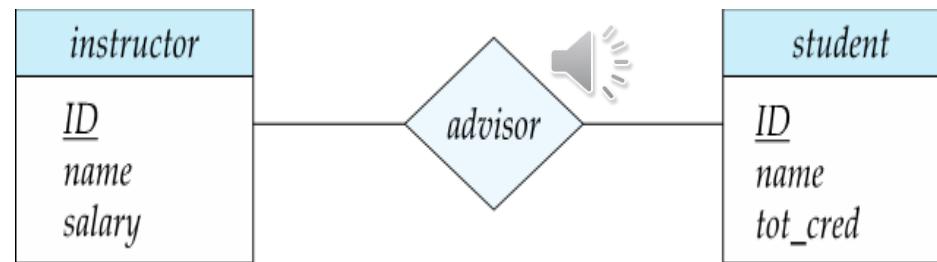
Many-to-One Relationships

- In a many-to-one relationship between an *instructor* and a *student*,
 - an *instructor* is associated with at most one *student* via *advisor*,
 - and a *student* is associated with several (including 0) *instructors* via *advisor*



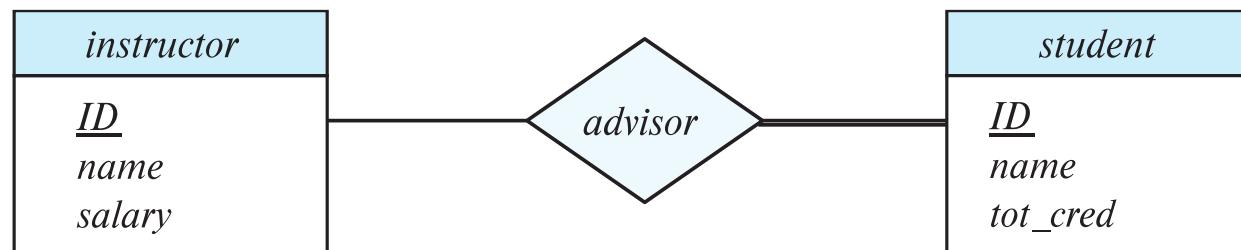
Many-to-Many Relationship

- An instructor is associated with several (possibly 0) students via *advisor*
- A student is associated with several (possibly 0) instructors via *advisor*



Total and Partial Participation

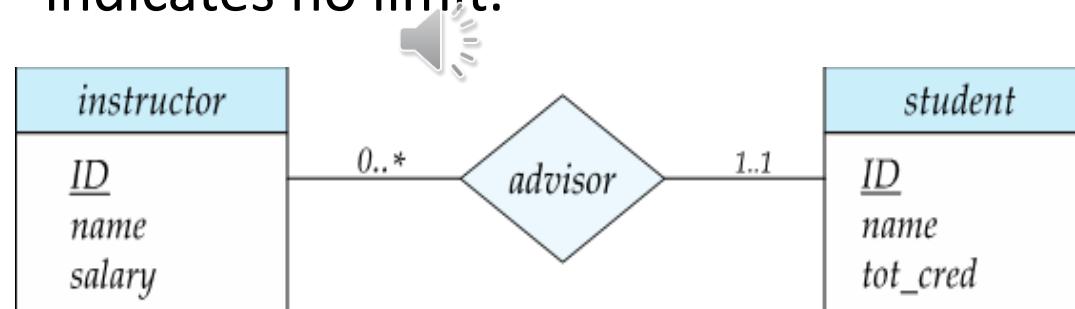
- **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
- **Partial participation**: some entities may not participate in any relationship in the relationship set



Notation for Expressing More Complex Constraints

- A line may have an associated minimum and maximum cardinality, shown in the form $l..h$, where l is the minimum and h the maximum cardinality
 - A minimum value of 1 indicates total participation.
 - A maximum value of 1 indicates that the entity participates in at most one relationship
- A maximum value of * indicates no limit.

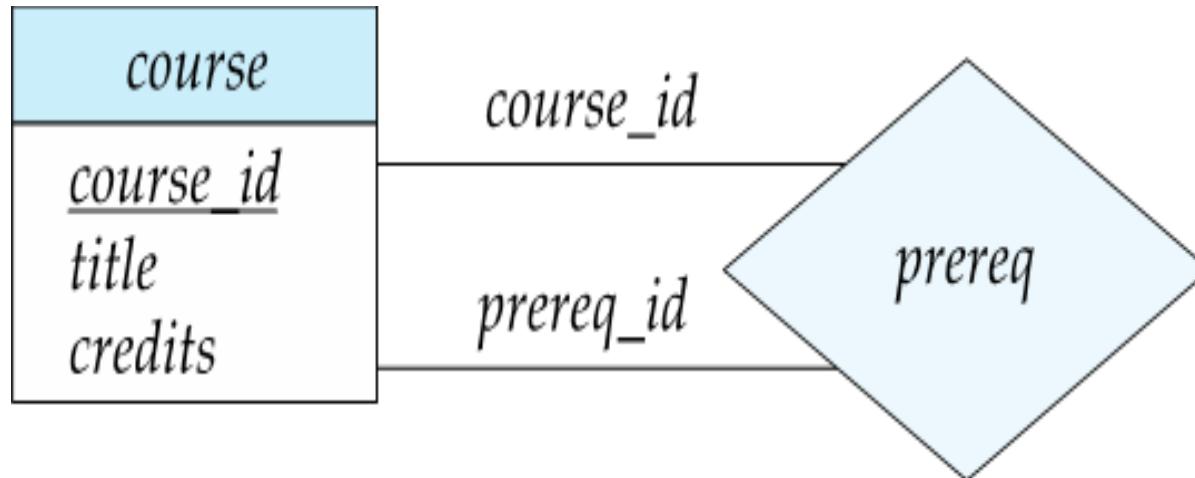
- Example



- Instructor can advise 0 or more students. A student must have 1 advisor; cannot have multiple advisors

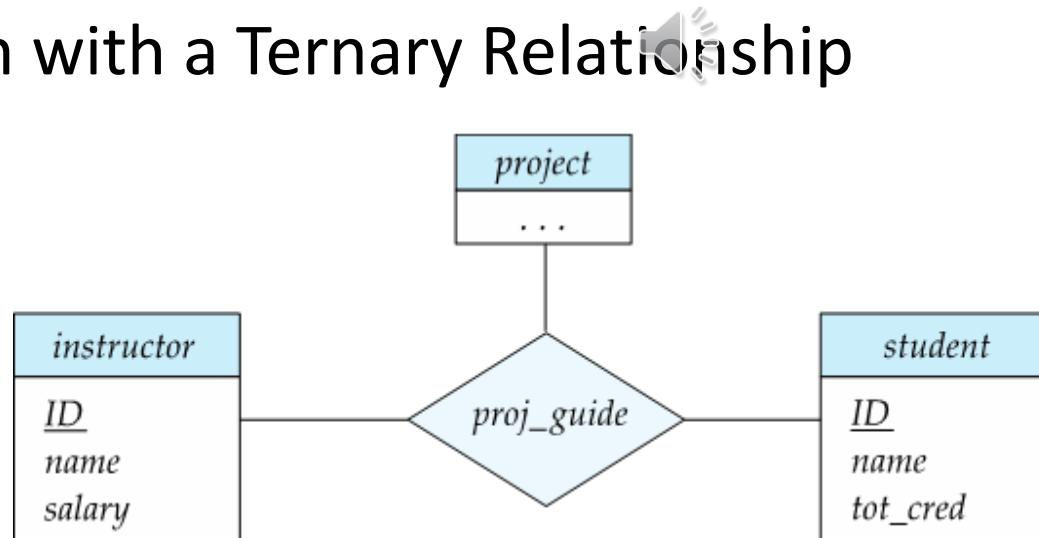
Roles

- Entity sets of a relationship need not be distinct
- Each occurrence of an entity set plays a “role” in the relationship
- The labels “*course_id*” and “*prereq_id*” are called **roles**.



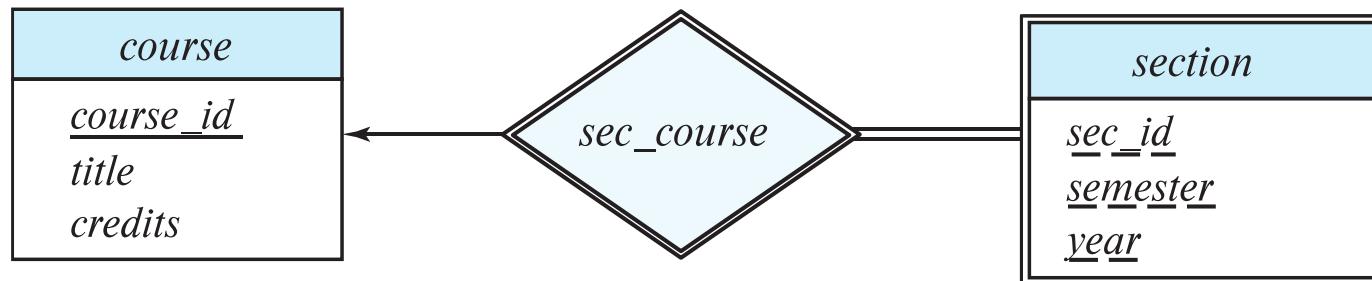
Non-binary Relationship Sets

- Most relationship sets are binary
- There are occasions when it is more convenient to represent relationships as non-binary.
- E-R Diagram with a Ternary Relationship



Expressing Weak Entity Sets

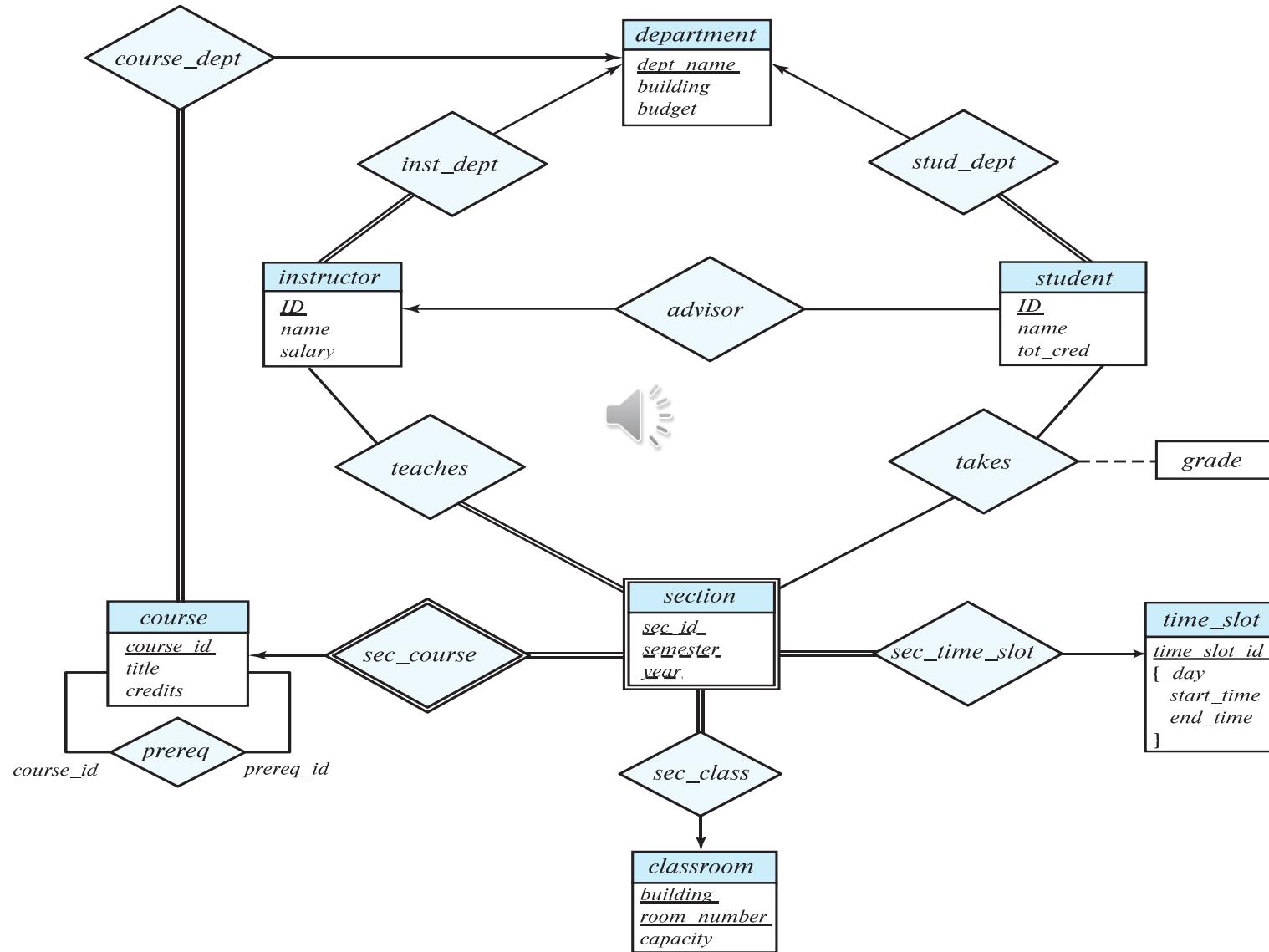
- In E-R diagrams, a weak entity set is depicted via a double rectangle.
- We underline the discriminator of a weak entity set with a dashed line.
- The relationship set connecting the weak entity set to the identifying strong entity set is depicted by a double diamond.
- Primary key for *section* – (*course_id*, *sec_id*, *semester*, *year*)



Database Design for a University Organization

- The university is organized into departments. Each department is identified by a unique name (*dept name*), is located in a particular *building*, and has a *budget*.
- Each department has a list of courses it offers. Each course has associated with it a *course id*, *title*, *dept name*, and *credits*, and may also have have associated *prerequisites*.
- Instructors are identified by their unique *ID*. Each instructor has *name*, associated department (*dept name*), and *salary*.
- Students are identified by their unique *ID*. Each student has a *name*, an associated major department (*dept name*), and *tot cred* (total credit hours the student earned thus far).
- The university maintains a list of classrooms, specifying the name of the *building*, *room number*, and *room capacity*.
- The university maintains a list of all classes (sections) taught. Each section is identified by a *course id*, *sec id*, *year*, and *semester*, and has associated with it a *semester*, *year*, *building*, *room number*, and *time slot id* (the time slot when the class meets).
- The department has a list of teaching assignments specifying, for each instructor, the sections the instructor is teaching.
- The university has a list of all student course registrations, specifying, for each student, the courses and the associated sections that the student has taken (registered for).
- In our university database, we have a constraint that each instructor must have exactly one associated department.

E-R Diagram for a University Enterprise



ER Model-Part 3

Reduction of ER Model to
Relational Schema



ER to Relational Mapping

- A database which conforms to an E-R diagram can be represented by a collection of schemas.



Representing Entity Sets

- A strong entity set reduces to a schema with the same attributes

student(ID, name, tot_cred)

Instructor(ID, name, salary)

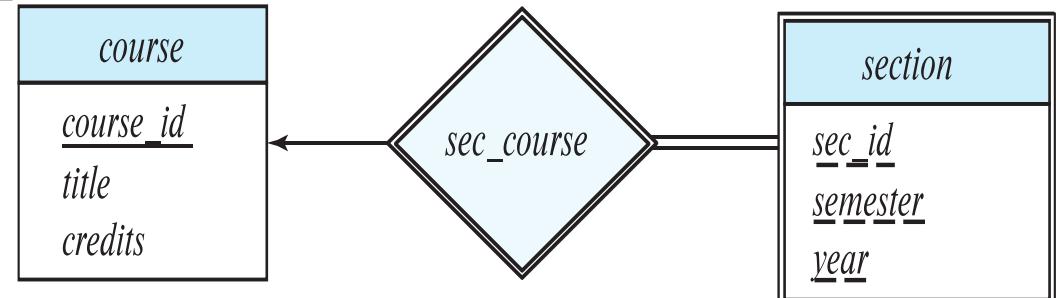


instructor
<u>ID</u>
name
salary

student
<u>ID</u>
name
tot_cred

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

section (course id, sec_id, sem, year)



Representation of Entity Sets with Composite Attributes

<i>instructor</i>
<i>ID</i>
<i>name</i>
<i>first_name</i>
<i>middle_initial</i>
<i>last_name</i>
<i>address</i>
<i>street</i>
<i>street_number</i>
<i>street_name</i>
<i>apt_number</i>
<i>city</i>
<i>state</i>
<i>zip</i>
{ <i>phone_number</i> }
<i>date_of_birth</i>
<i>age ()</i>

- Composite attributes are flattened out by creating a separate attribute for each component attribute
 - Example: given entity set *instructor* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes *name_first_name* and *name_last_name*
 - Prefix omitted if there is no ambiguity (*name_first_name* could be *first_name*)
- Ignoring multivalued attributes, extended instructor schema is
 - *instructor(ID, first_name, middle_initial, last_name, street_number, street_name, apt_number, city, state, zip_code, date_of_birth)*

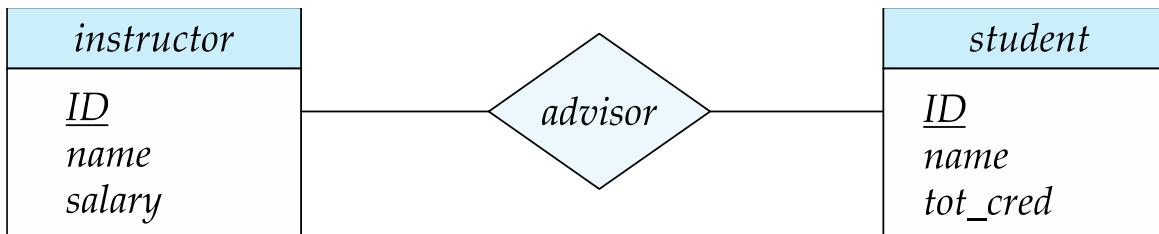
Representation of Entity Sets with Multivalued Attributes

- A multivalued attribute M of an entity E is represented by a separate schema EM
- Schema EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
- Example: Multivalued attribute $phone_number$ of $instructor$ is represented by a schema:
 $inst_phone= (\underline{ID}, \underline{phone_number})$
- Each value of the multivalued attribute maps to a separate tuple of the relation on schema EM
 - For example, an $instructor$ entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:
(22222, 456-7890) and (22222, 123-4567)

Representing Relationship Sets

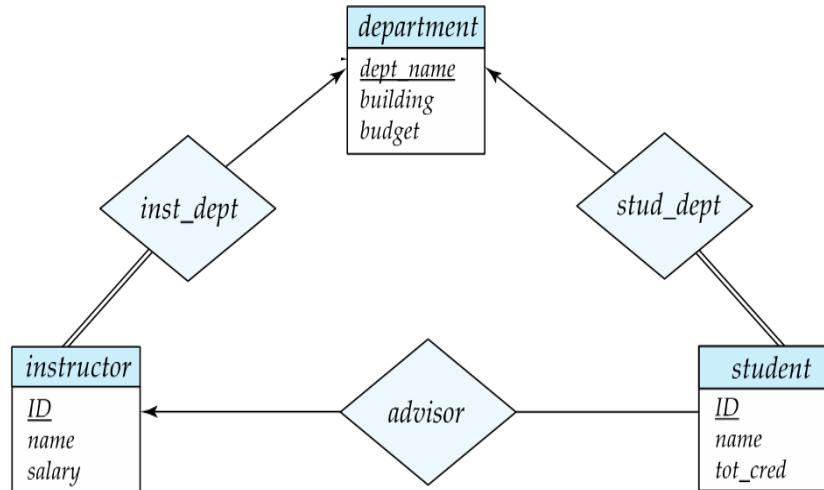
- A **many-to-many relationship set** is represented as a schema with attributes for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.
- Example: schema for relationship set *advisor*

advisor = (s_id, i_id)



Representing Relationship Sets

- **Many-to-one and one-to-many relationship sets** that are total on the many-side can be represented by adding an extra attribute to the “many” side, containing the primary key of the “one” side
- Example: Instead of creating a schema for relationship set *inst_dept*, add an attribute *dept_name* to the schema arising from entity set *instructor*
- Example



Instructor(id,name,salary,dept_name)

Representing Relationship Sets

- For **one-to-one relationship** sets, either side can be chosen to act as the “many” side
 - That is, an extra attribute can be added to either of the tables corresponding to the two entity sets
 - It is better to add the primary key of entity set that is partially participating in the relationship set as a foreign key in the relation corresponding to entity at total participating side.

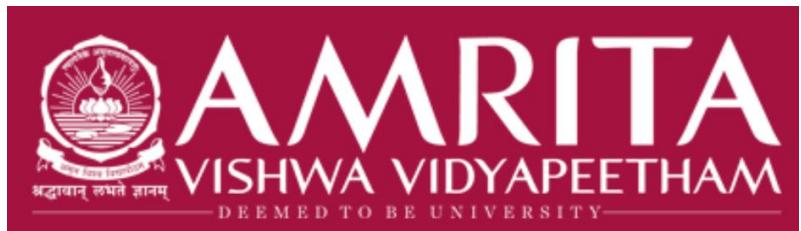
Mapping of N-ary Relationship Types.

- For each n-ary relationship type R, where $n > 2$, create a new relation S to represent R.
 - Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types.
 - Also include any simple attributes of the n-ary relationship type (or simple components of composite attributes) as attributes of S.

Summary – Mapping ER to Relational Schema

Table 9.1 Correspondence between ER and Relational Models

ER MODEL	RELATIONAL MODEL
Entity type	<i>Entity</i> relation
1:1 or 1:N relationship type	Foreign key (or <i>relationship</i> relation)
M:N relationship type	<i>Relationship</i> relation and <i>two</i> foreign keys
<i>n</i> -ary relationship type	<i>Relationship</i> relation and <i>n</i> foreign keys
Simple attribute	Attribute
Composite attribute	Set of simple component attributes
Multivalued attribute	Relation and foreign key
Value set	Domain
Key attribute	Primary (or secondary) key



Overview of NoSQL

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

What is NoSQL?

- The NoSQL name was introduced during an open-source event on distributed databases
- It doesn't mean 'No SQL' - it means 'Not only SQL'

NoSQL



'Not only
SQL'

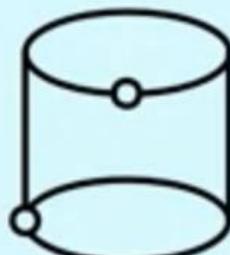
What is NoSQL?

- Refers to family of databases that vary widely in style and technology
- However, they share a common trait
 - Non-relational
 - Not standard row and column type RDBMS
- Could be referred to as 'Non-relational databases'

What is NoSQL?

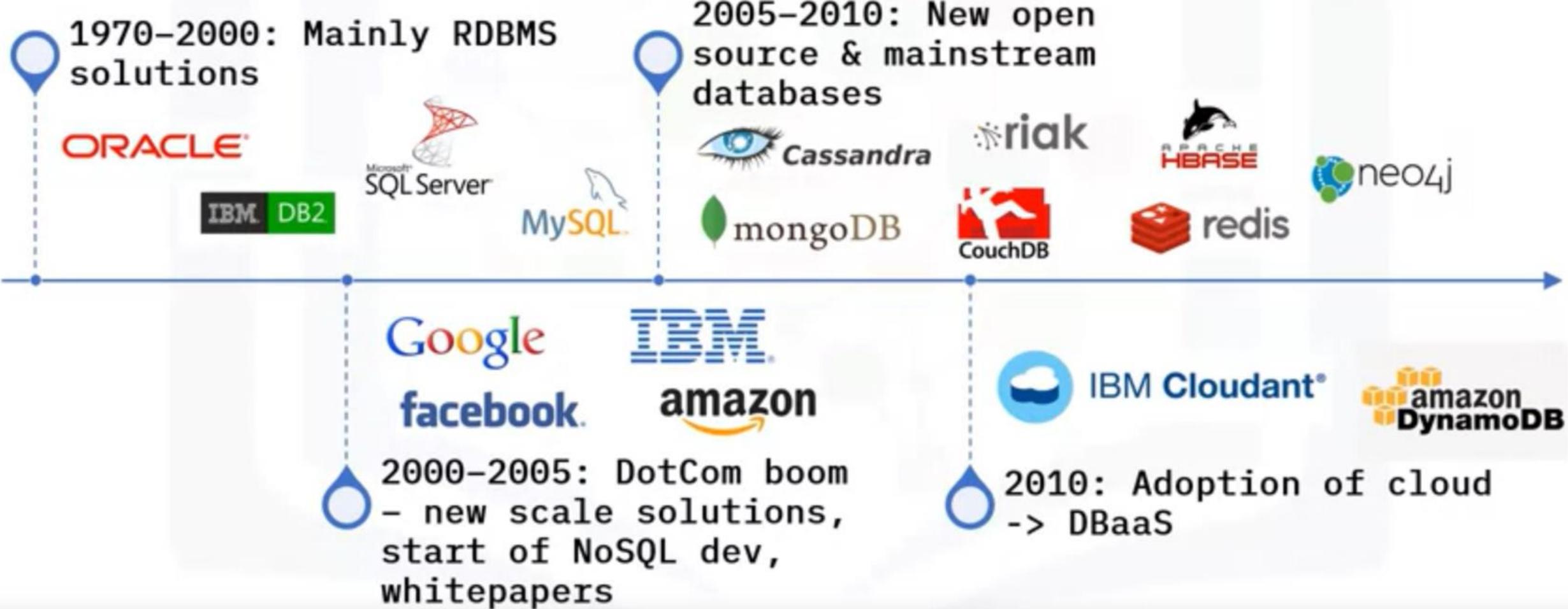
NoSQL databases:

- Provide new ways of storing and querying data
 - Address several issues for modern apps
- Are designed to handle a different breed of scale - 'Big Data'
- Typically address more specialized use cases
- Simpler to develop app functionality for than RDBMS



NoSQL
Databases

History of NoSQL



Summary

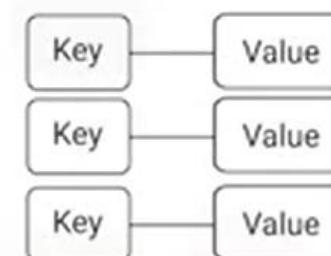
- The name ‘NoSQL’ stands for Not only SQL
- NoSQL refers to a class of databases that are non-relational in architecture
- Implementations of NoSQL databases differ technically, but share common traits
- Since 2000 NoSQL databases have become more popular in the marketplace, due to scale demands of Big Data

NoSQL Database Categories

- The most common trait amongst NoSQL databases is that they are non-relational in architecture
- What types of NoSQL databases are available?
- What is common to them?

NoSQL Database Categories

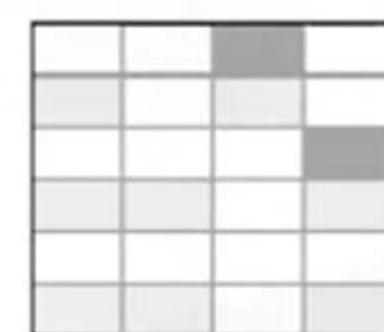
General consensus is...
...NoSQL databases fit
into four categories



Key-Value



Document

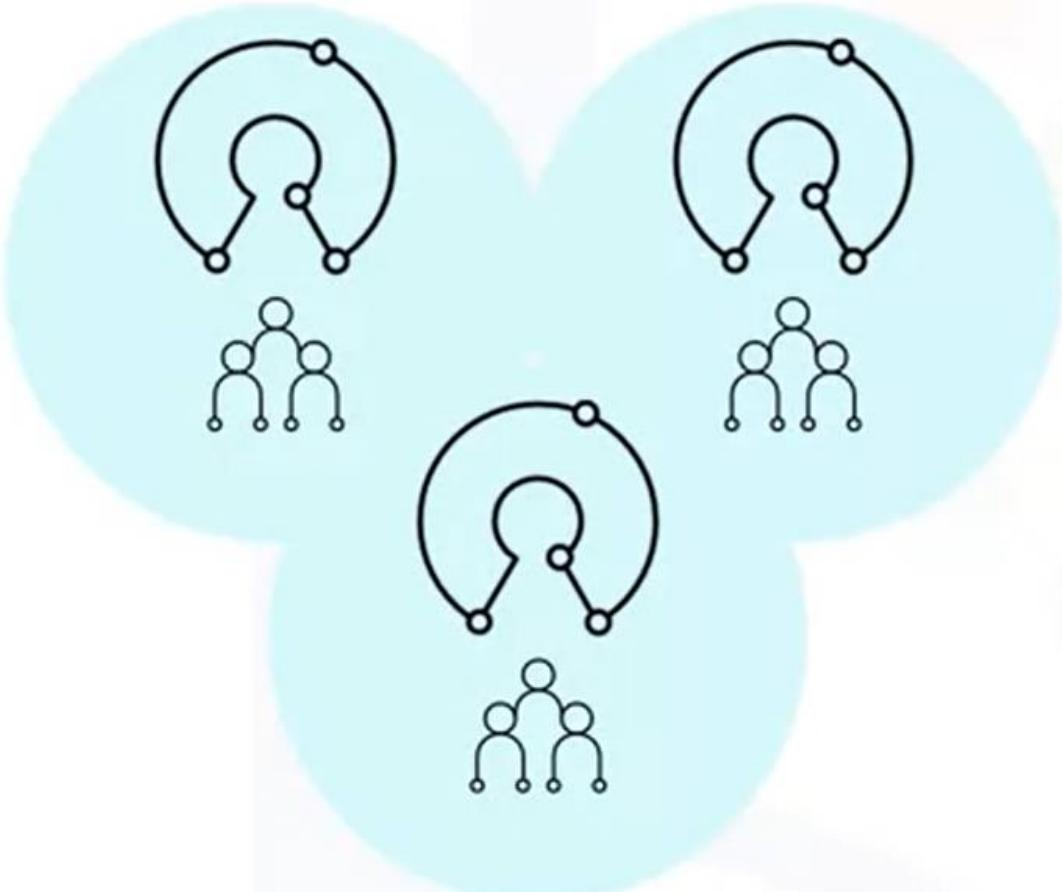


Column



Graph

NoSQL Database Characteristics



Open Source
Communities

But what ties NoSQL databases together?

- Majority have their roots in the open source community
- Many have been used and leveraged in an open source manner

Benefits of NoSQL Databases

Why use a NoSQL database?

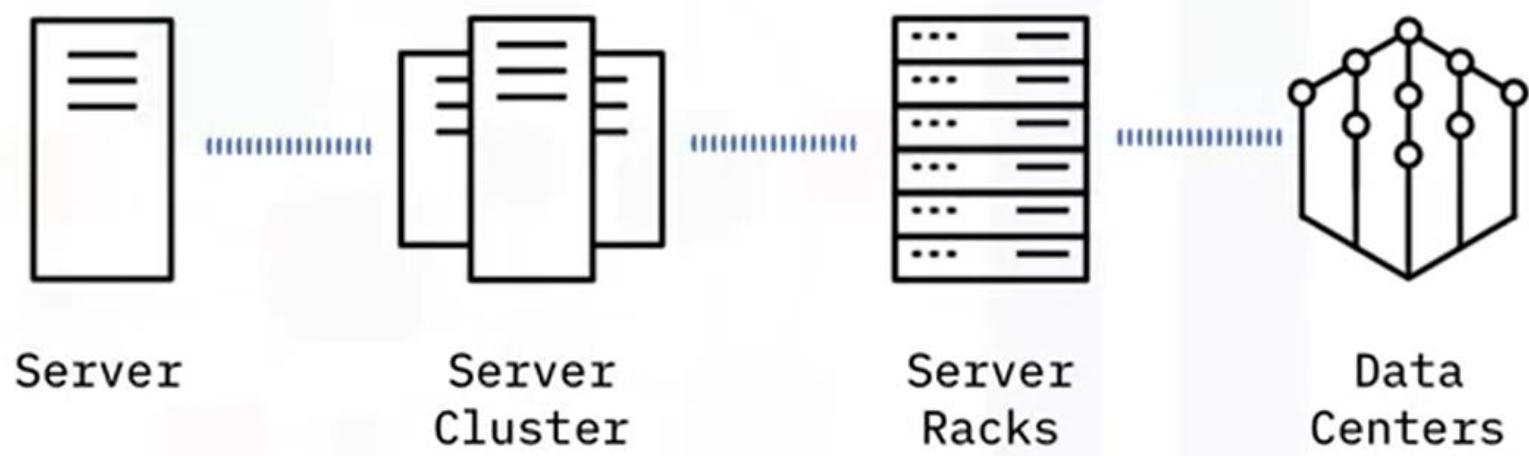
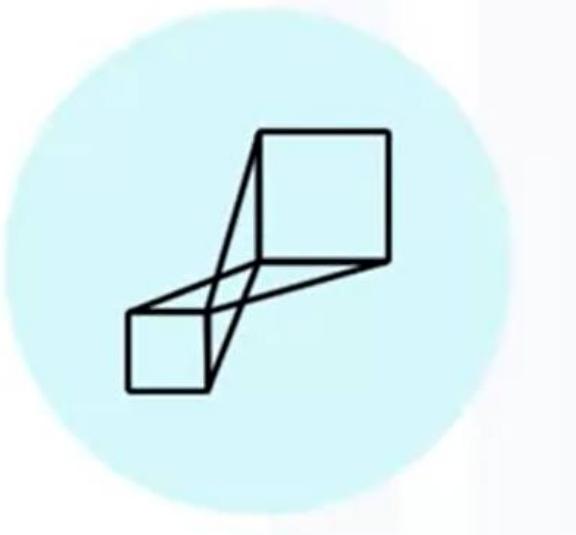


Why is their popularity growing so rapidly?



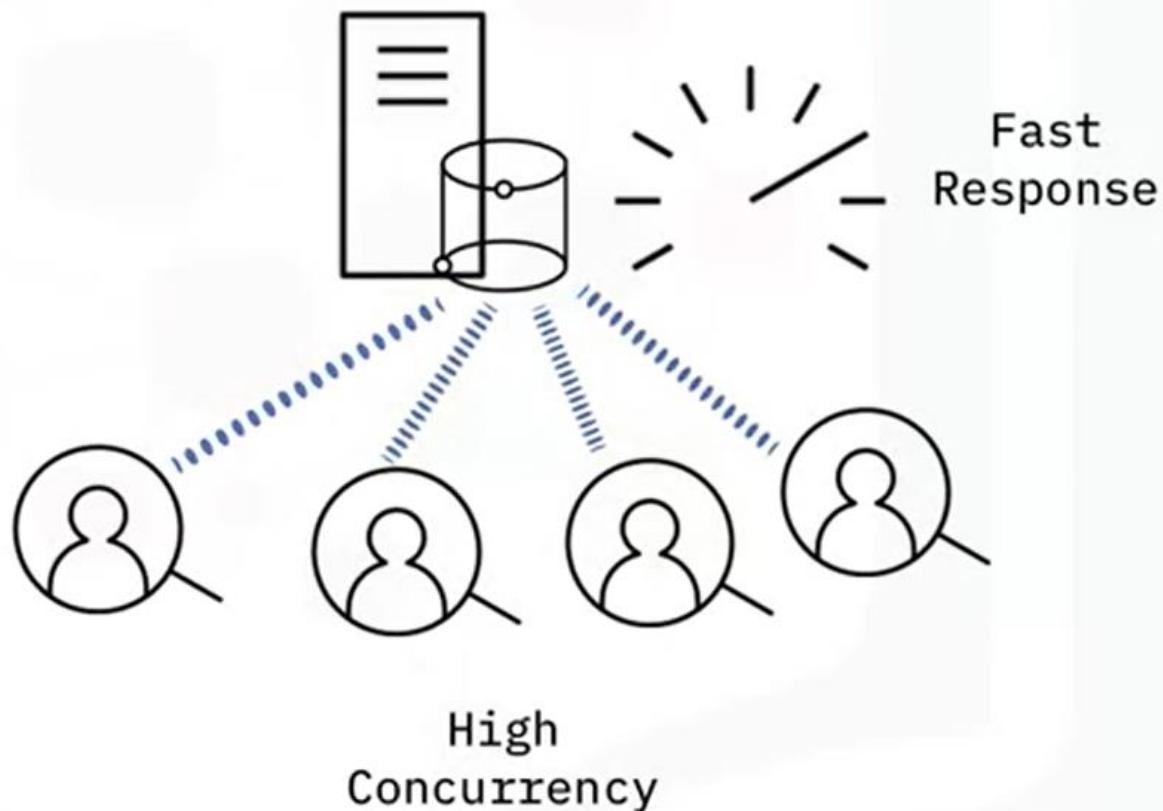
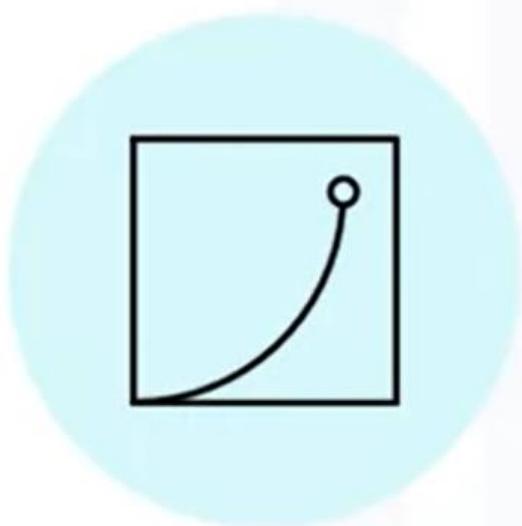
Benefits of NoSQL Databases

Scalability



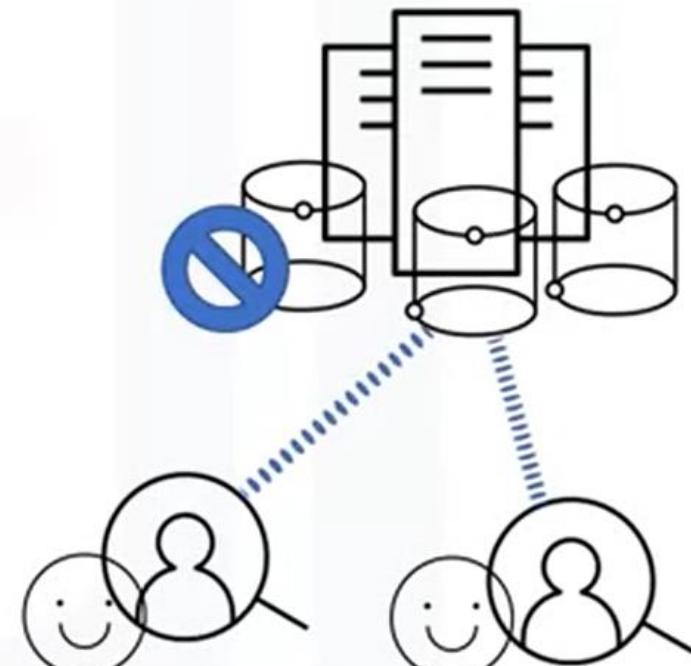
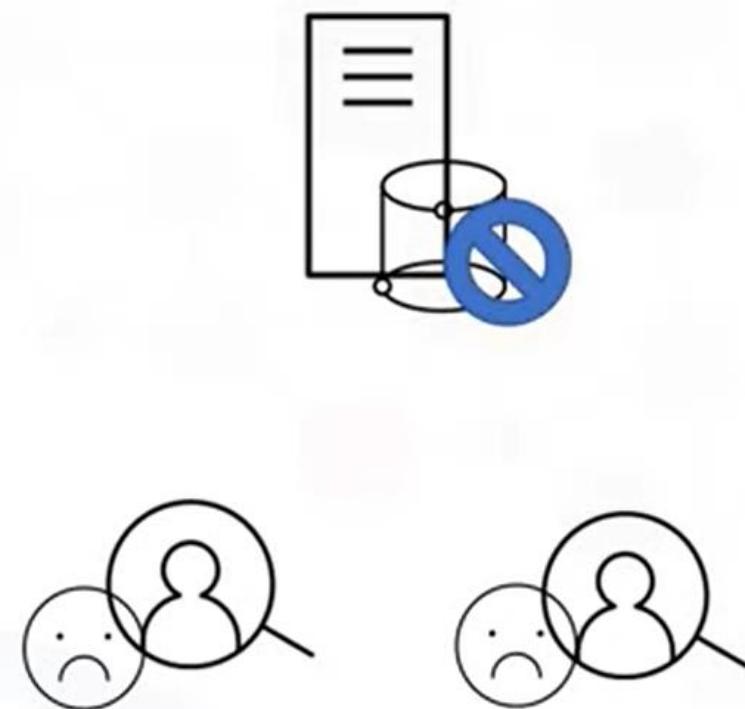
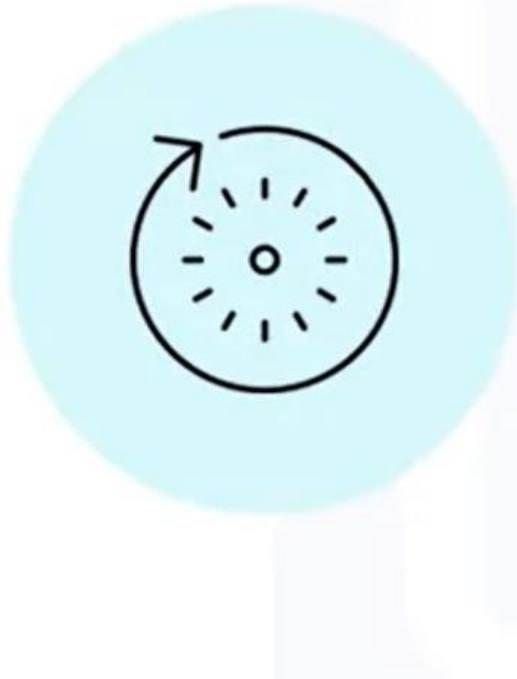
Benefits of NoSQL Databases

Performance



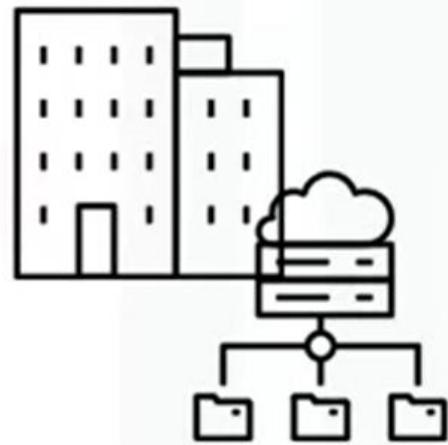
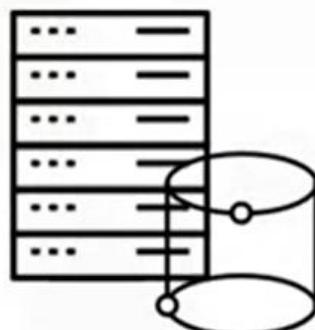
Benefits of NoSQL Databases

Availability



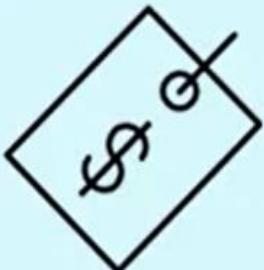
Benefits of NoSQL Databases

Cloud Architecture

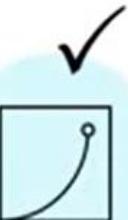
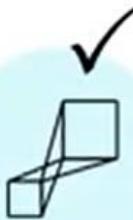


Benefits of NoSQL Databases

Cost

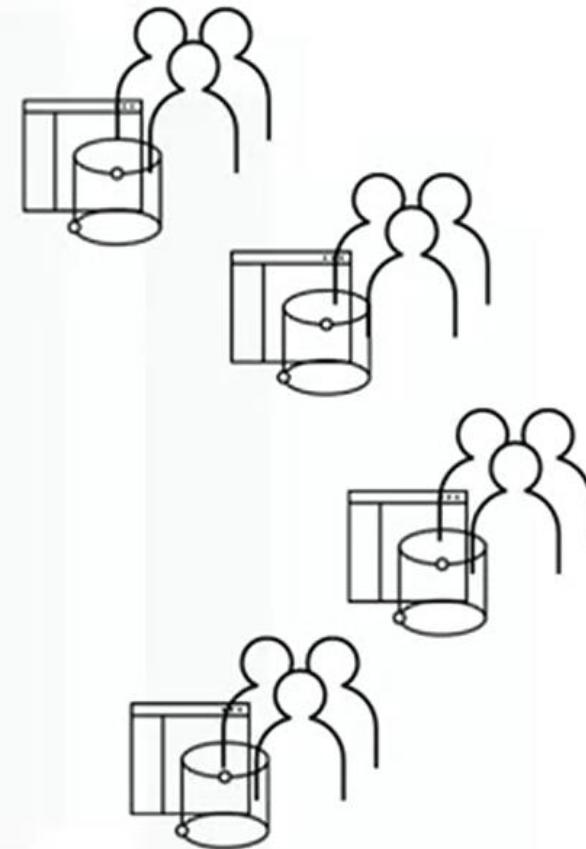


NoSQL



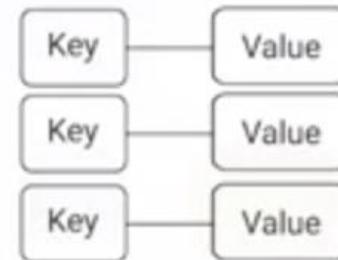
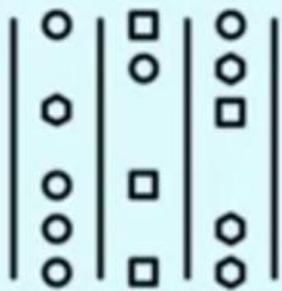
Benefits of NoSQL Databases

Flexible Schema



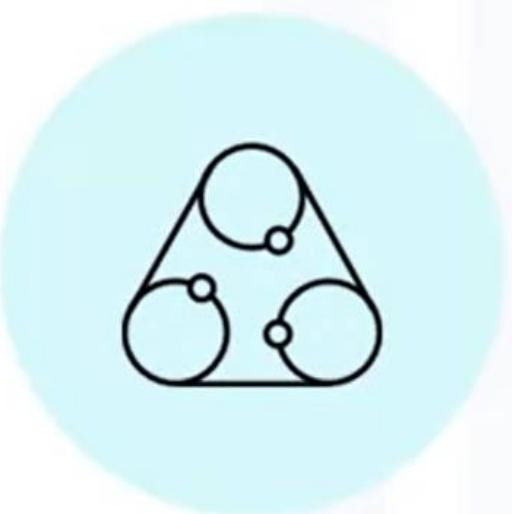
Benefits of NoSQL Databases

Varied Data Structures

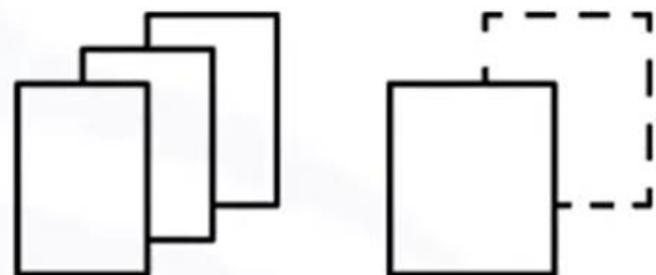


Benefits of NoSQL Databases

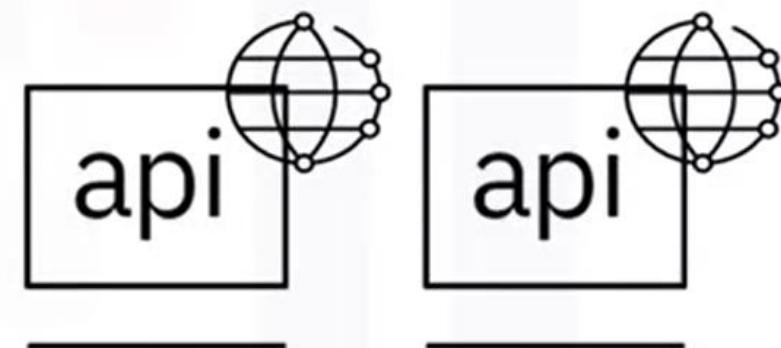
Specialized Capabilities



Indexing and Querying



Data Replication Robustness



Modern HTTP APIs

Summary

- NoSQL databases are non-relational
- There are four categories of NoSQL database
- NoSQL databases have their roots in the open-source community
- NoSQL database implementations are technically different
- There are several benefits to adopting NoSQL databases



Overview of MongoDB

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

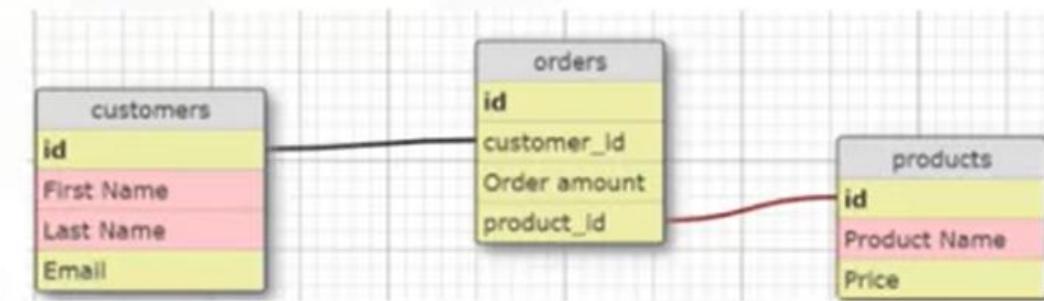
Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

What is MongoDB Database?

- A document and a NoSQL database



Order
document

What are documents?

Associative arrays like JSON objects or Python dictionaries

For example, a student document

```
{  
    "firstName": "John",  
    "lastName": "Doe",  
    "email": "john.doe@email.com",  
    "studentId": 20217484  
}
```

What is a Collection?

A group of stored documents

For example, all student records in Students section (**collection**)

and staff records in Employees section (**collection**)



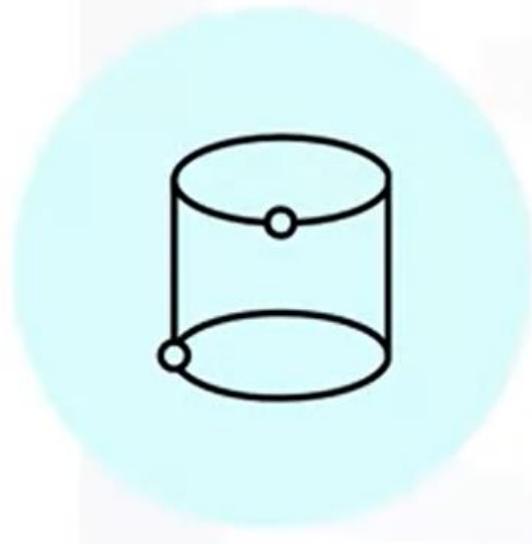
Students



Employees

What is a Database?

Database stores Collections



Students and Employees collections stored in a database called CampusManagementDB

Documents in detail - 1/2

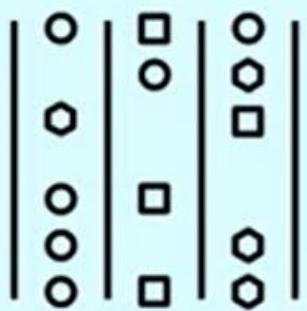
Fields in the document below:
firstName, lastName, email and studentId

```
{  
    "firstName": "John",  
    "lastName": "Doe",  
    "email":      "john.doe@email.com",  
    "studentId": 20217484  
}
```

Documents in detail - 2/2

MongoDB supports various data types:

```
{  
  "name": "John",  
  "dateOfBirth": ISODate("2000-01-01T14:45:00.000Z"),  
  "studentId": 20217484,  
  "enrolled": true,  
  "balance": 20.01,  
  "address": {  
    "city": "Stonefield",  
    "country": "UK"  
  },  
  "interests": ["football", "skiing", "travelling"]  
}
```



Why use MongoDB?

- Model data as you read/write, not the other way
 - Traditional relational databases: create schema first, then create tables
 - To store another field, you have to alter tables

```
{  
    "orderId": "AXU-1311",  
    "customer": {  
        "name": "John Doe",  
        "email": "john.doe@email.com"  
    }  
}
```

Why use MongoDB?

- Model data as you read/write, not the other way
 - Traditional relational databases: create schema first, then create tables
 - To store another field, you have to alter tables

```
SELECT * FROM Orders INNER JOIN Customers...
```

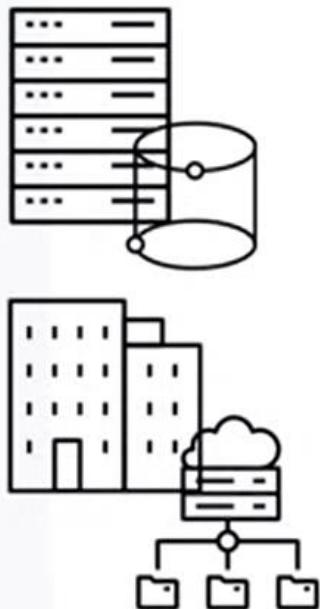
```
ALTER TABLE Customers...
```

In MongoDB , you change the field as you go along.....

Where to use MongoDB

MongoDB is a popular choice of database for

- Large and unstructured
- Complex
- Flexible
- Highly scalable applications
- Self-managed, hybrid, or cloud hosted



- **From where to install**

- <https://www.mongodb.com/download-center/community>
- <https://www.mongodb.org/dl/win32/i386>

- **How to install**

- <https://www.guru99.com/installation-configuration-mongodb.html#1>

- **After Installation**

- Open command propmt
- D:\set up\mongodb\bin>mongod.exe --dbpath "C:\dbdata"
- Open another command prompt
- D:\set up\mongodb\bin>mongo.exe

What is MongoDB?

MongoDB is a document-oriented NoSQL database used for high volume data storage. Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents. Documents consist of key-value pairs which are the basic unit of data in MongoDB. Collections contain sets of documents and function which is the equivalent of relational database tables. MongoDB is a database which came into light around the mid-2000s.

MongoDB Features

1. Each database contains collections which in turn contains documents. Each document can be different with a varying number of fields. The size and content of each document can be different from each other.
2. The document structure is more in line with how developers construct their classes and objects in their respective programming languages. Developers will often say that their classes are not rows and columns but have a clear structure with key-value pairs.
3. The rows (or documents as called in MongoDB) doesn't need to have a schema defined beforehand. Instead, the fields can be created on the fly.
4. The data model available within MongoDB allows you to represent hierarchical relationships, to store arrays, and other more complex structures more easily.
5. Scalability – The MongoDB environments are very scalable. Companies across the world have defined clusters with some of them running 100+ nodes with around millions of documents within the database

The below example shows how a document can be modeled in MongoDB.

1. The `_id` field is added by MongoDB to uniquely identify the document in the collection.
2. What you can note is that the Order Data (OrderID, Product, and Quantity) which in RDBMS will normally be stored in a separate

table, while in MongoDB it is actually stored as an embedded document in the collection itself. This is one of the key differences in how data is modeled in MongoDB.

```
{  
    _id : <ObjectId> ,  
  
    CustomerName : Guru99 ,  
  
    Order:  
        {  
            OrderID: 111  
            Product: ProductA  
            Quantity: 5  
        }  
}
```

OrderID: 111
Product: ProductA
Quantity: 5

Example of how data can be embedded in a document

Key Components of MongoDB Architecture

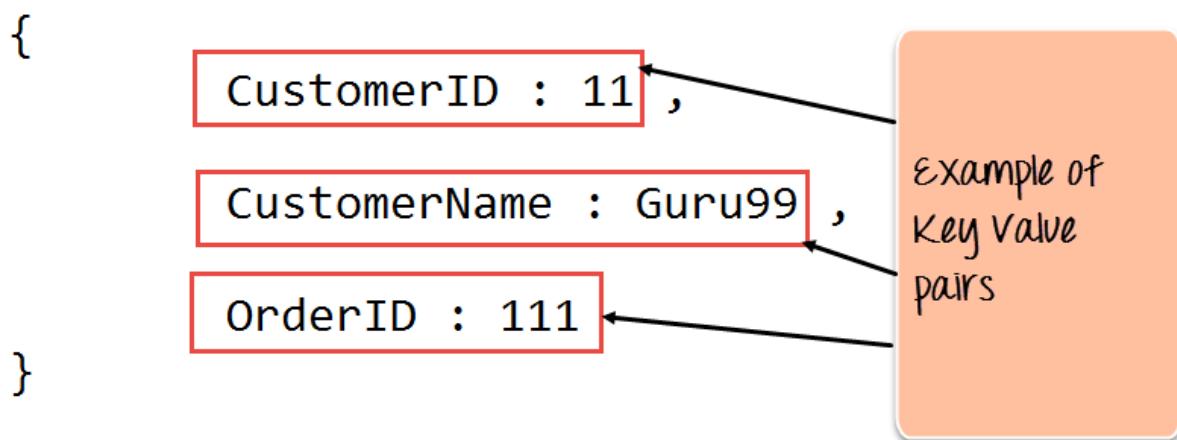
Below are a few of the common terms used in MongoDB

1. **_id** – This is a field required in every MongoDB document. The _id field represents a unique value in the MongoDB document. The _id field is like the document's primary key. If you create a new document without an _id field, MongoDB will automatically create the field. So for example, if we see the example of the above customer table, Mongo DB will add a 24 digit unique identifier to each document in the collection.

_Id	CustomerID	CustomerName
563479cc8a8a4246bd27d784	11	Guru99
563479cc7a8a4246bd47d784	22	Trevor Smith
563479cc9a8a4246bd57d784	33	Nicole

2. **Collection** – This is a grouping of MongoDB documents. A collection is the equivalent of a table which is created in any other RDMS such as Oracle or MS SQL. A collection exists within a single database. As seen from the introduction collections don't enforce any sort of structure.

3. **Cursor** – This is a pointer to the result set of a query. Clients can iterate through a cursor to retrieve results.
4. **Database** – This is a container for collections like in RDMS wherein it is a container for tables. Each database gets its own set of files on the file system. A MongoDB server can store multiple databases.
5. **Document** – A record in a MongoDB collection is basically called a document. The document, in turn, will consist of field name and values.
6. **Field** – A name-value pair in a document. A document has zero or more fields. Fields are analogous to columns in relational databases. The following diagram shows an example of Fields with Key value pairs. So in the example below CustomerID and 11 is one of the key value pair's defined in the document.



7. **JSON** – This is known as [JavaScript](#) Object Notation. This is a human-readable, plain text format for expressing structured data. JSON is currently supported in many programming languages.

Just a quick note on the key difference between the `_id` field and a normal collection field. The `_id` field is used to uniquely identify the documents in a collection and is automatically added by MongoDB when the collection is created.

Why Use MongoDB?

Below are the few of the reasons as to why one should start using MongoDB

1. Document-oriented – Since MongoDB is a [NoSQL](#) type database, instead of having data in a relational type format, it stores the data in documents. This makes MongoDB very flexible and adaptable to real business world situation and requirements.
2. Ad hoc queries – MongoDB supports searching by field, range queries, and regular expression searches. Queries can be made to return specific fields within documents.
3. Indexing – Indexes can be created to improve the performance of searches within MongoDB. Any field in a MongoDB document can be indexed.
4. Replication – MongoDB can provide high availability with replica sets. A replica set consists of two or more mongo DB instances. Each replica set member may act in the role of the primary or secondary replica at any time. The primary replica is the main server which interacts with the client and performs all the read/write operations. The Secondary replicas maintain a copy of the data of the primary using built-in replication. When a primary replica fails, the replica set automatically switches over to the secondary and then it becomes the primary server.
5. Load balancing – MongoDB uses the concept of sharding to scale horizontally by splitting data across multiple MongoDB instances. MongoDB can run over multiple servers, balancing the load and/or duplicating data to keep the system up and running in case of hardware failure.

Data Modelling in MongoDB

As we have seen from the Introduction section, the data in MongoDB has a flexible schema. Unlike in [SQL](#) databases, where you must have a table's schema declared before inserting data, MongoDB's collections do not enforce document structure. This sort of flexibility is what makes MongoDB so powerful.

When modeling data in Mongo, keep the following things in mind

1. What are the needs of the application – Look at the business needs of the application and see what data and the type of data needed

for the application. Based on this, ensure that the structure of the document is decided accordingly.

2. What are data retrieval patterns – If you foresee a heavy query usage then consider the use of indexes in your data model to improve the efficiency of queries.
3. Are frequent inserts, updates and removals happening in the database? Reconsider the use of indexes or incorporate sharding if required in your data modeling design to improve the efficiency of your overall MongoDB environment.

Difference between MongoDB & RDBMS

Below are some of the key term differences between MongoDB and RDBMS

RDBMS	MongoDB	Difference
Table	Collection	In RDBMS , the table contains the columns and rows which are used to store the data whereas, in MongoDB, this same structure is known as a collection. The collection contains documents which in turn contains Fields, which in turn are key-value pairs.
Row	Document	In RDBMS, the row represents a single, implicitly structured data item in a table. In MongoDB, the data is stored in documents.
Column	Field	In RDBMS, the column denotes a set of data values. These in MongoDB are known as Fields.
Joins	Embedded documents	In RDBMS, data is sometimes spread across various tables and in order to show a complete view of all data, a join is sometimes formed across tables to get the data. In MongoDB, the data is normally stored in a single collection, but separated by using Embedded documents. So there is no concept of joins in MongoDB.

Apart from the terms differences, a few other differences are shown below

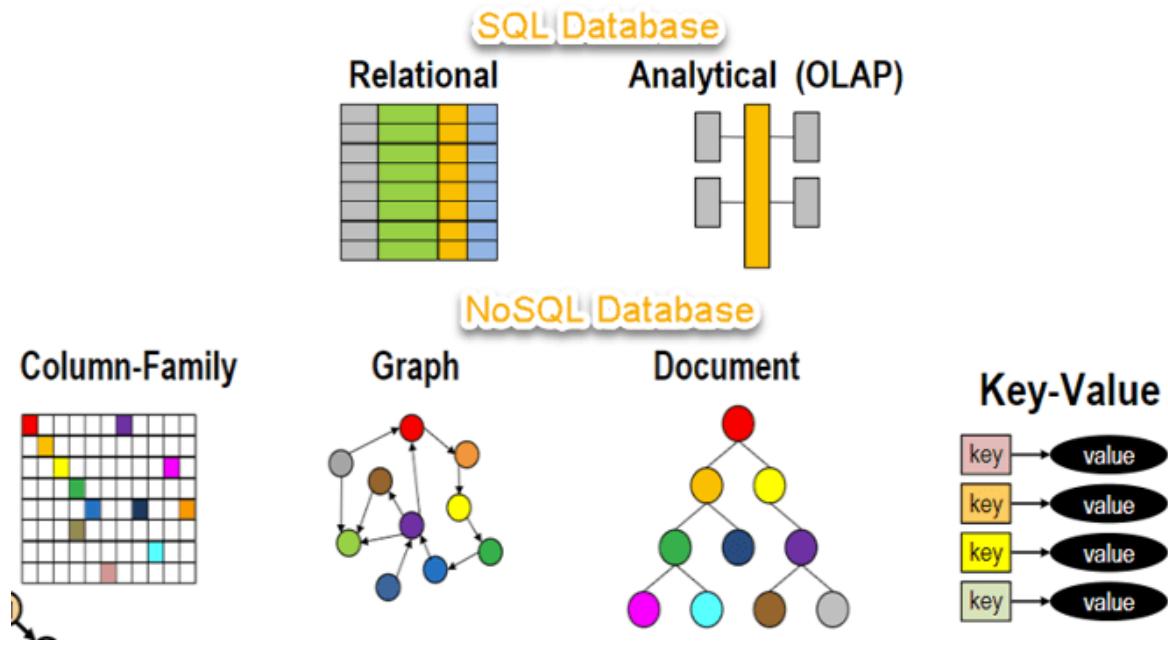
1. Relational databases are known for enforcing data integrity. This is not an explicit requirement in MongoDB.
2. RDBMS requires that data be [normalized](#) first so that it can prevent orphan records and duplicates. Normalizing data then has the requirement of more tables, which will then result in more table joins, thus requiring more keys and indexes. As databases start to grow, performance can start becoming an issue. Again this is not an explicit requirement in MongoDB. MongoDB is flexible and does not need the data to be normalized first.

What is NoSQL?

NoSQL Database is a non-relational Data Management System, that does not require a fixed schema. It avoids joins, and is easy to scale. The major purpose of using a NoSQL database is for distributed data stores with humongous data storage needs. NoSQL is used for Big data and real-time web apps. For example, companies like Twitter, Facebook and Google collect terabytes of user data every single day.

NoSQL database stands for “Not Only SQL” or “Not SQL.” Though a better term would be “NoREL”, NoSQL caught on. Carl Strozz introduced the NoSQL concept in 1998.

Traditional RDBMS uses SQL syntax to store and retrieve data for further insights. Instead, a NoSQL database system encompasses a wide range of database technologies that can store structured, semi-structured, unstructured and polymorphic data.



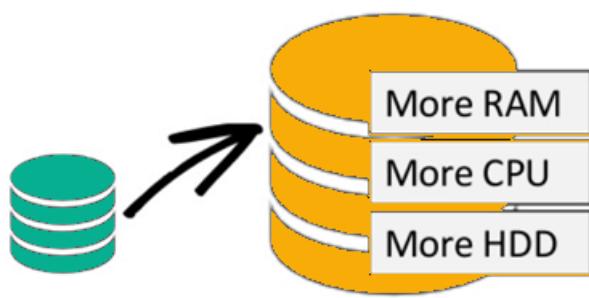
Why NoSQL?

The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data. The system response time becomes slow when you use RDBMS for massive volumes of data.

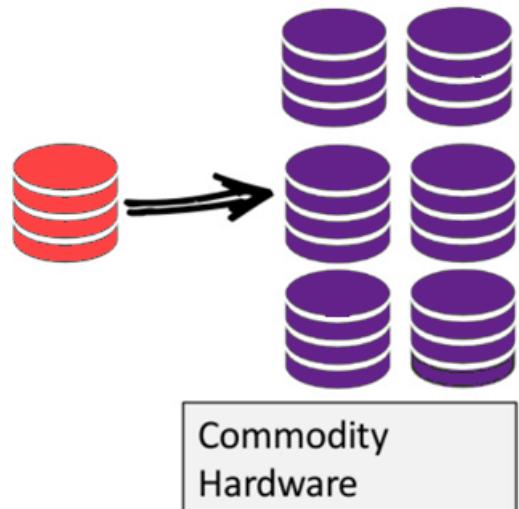
To resolve this problem, we could “scale up” our systems by upgrading our existing hardware. This process is expensive.

The alternative for this issue is to distribute database load on multiple hosts whenever the load increases. This method is known as “scaling out.”

Scale-Up (*vertical* scaling):



Scale-Out (*horizontal* scaling):



NoSQL database is non-relational, so it scales out better than relational databases as they are designed with web applications in mind.

Features of NoSQL

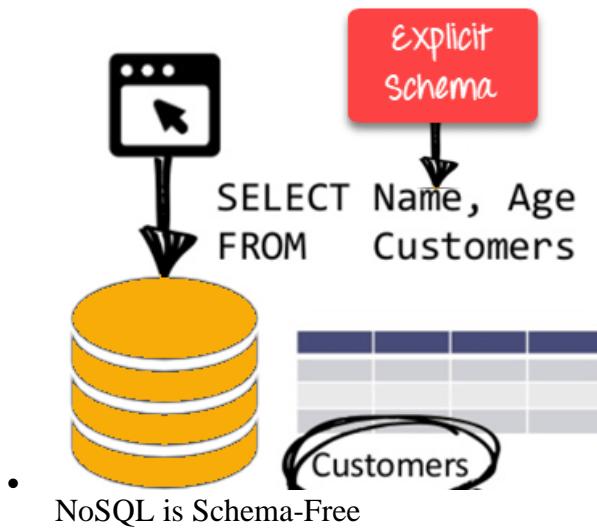
Non-relational

- NoSQL databases never follow the [relational model](#)
- Never provide tables with flat fixed-column records
- Work with self-contained aggregates or BLOBs
- Doesn't require object-relational mapping and data normalization
- No complex features like query languages, query planners, referential integrity joins, ACID

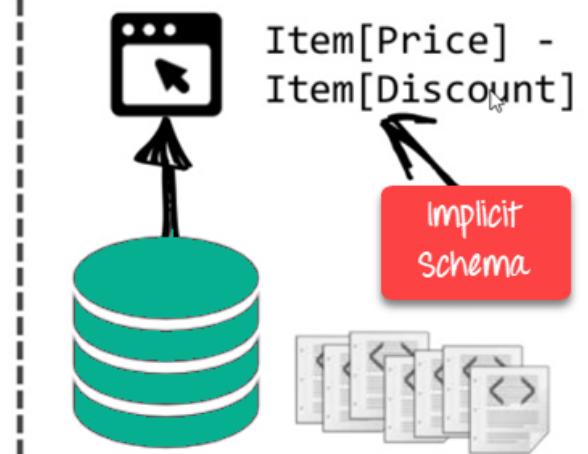
Schema-free

- NoSQL databases are either schema-free or have relaxed schemas
- Do not require any sort of definition of the schema of the data
- Offers heterogeneous structures of data in the same domain

RDBMS:



NoSQL DB:

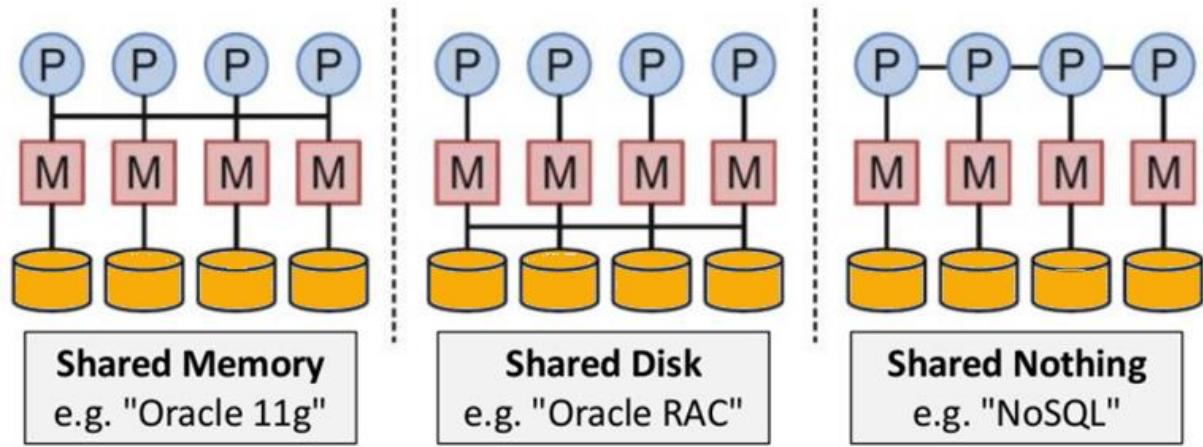


Simple API

- Offers easy to use interfaces for storage and querying data provided
- APIs allow low-level data manipulation & selection methods
- Text-based protocols mostly used with HTTP REST with JSON
- Mostly used no standard based NoSQL query language
- Web-enabled databases running as internet-facing services

Distributed

- Multiple NoSQL databases can be executed in a distributed fashion
- Offers auto-scaling and fail-over capabilities
- Often ACID concept can be sacrificed for scalability and throughput
- Mostly no synchronous replication between distributed nodes
Asynchronous Multi-Master Replication, peer-to-peer, HDFS Replication
- Only providing eventual consistency
- Shared Nothing Architecture. This enables less coordination and higher distribution.



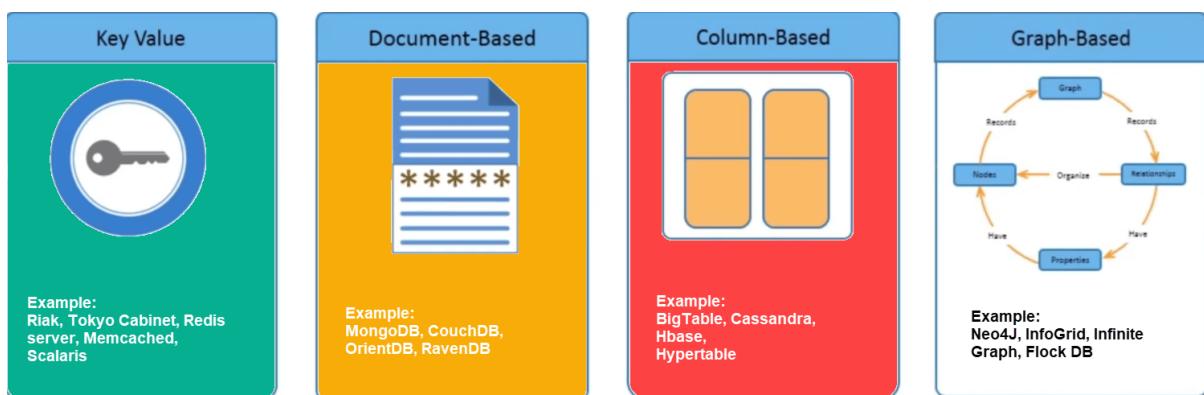
NoSQL is Shared Nothing.

Types of NoSQL Databases

NoSQL Databases are mainly categorized into four types: Key-value pair, Column-oriented, Graph-based and Document-oriented. Every category has its unique attributes and limitations. None of the above-specified database is better to solve all the problems. Users should select the database based on their product needs.

Types of NoSQL Databases:

- Key-value Pair Based
- Column-oriented Graph
- Graphs based
- Document-oriented



Key Value Pair Based

Data is stored in key/value pairs. It is designed in such a way to handle lots of data and heavy load.

Key-value pair storage databases store data as a hash table where each key is unique, and the value can be a JSON, BLOB(Binary Large Objects), string, etc.

For example, a key-value pair may contain a key like “Website” associated with a value like “Guru99”.

Key	Value
Name	Joe Bloggs
Age	42
Occupation	Stunt Double
Height	175cm
Weight	77kg

It is one of the most basic NoSQL database example. This kind of NoSQL database is used as a collection, dictionaries, associative arrays, etc. Key value stores help the developer to store schema-less data. They work best for shopping cart contents.

Redis, Dynamo, Riak are some NoSQL examples of key-value store DataBases. They are all based on Amazon’s Dynamo paper.

Column-based

Column-oriented databases work on columns and are based on BigTable paper by Google. Every column is treated separately. Values of single column databases are stored contiguously.

ColumnFamily			
Row Key	Column Name		
	Key	Key	Key
	Value	Value	Value
	Column Name		
	Key	Key	Key
	Value	Value	Value

Column based NoSQL database

They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.

Column-based NoSQL databases are widely used to manage data warehouses, [business intelligence](#), CRM, Library card catalogs,

HBase, Cassandra, HBase, Hypertable are NoSQL query examples of column based database.

Document-Oriented

Document-Oriented NoSQL DB stores and retrieves data as a key value pair but the value part is stored as a document. The document is stored in JSON or XML formats. The value is understood by the DB and can be queried.

Col1	Col2	Col3	Col4	Document 1	Document 2	Document 3
Data	Data	Data	Data	<pre>{ "prop1": data, "prop2": data, "prop3": data, "prop4": data }</pre>	<pre>{ "prop1": data, "prop2": data, "prop3": data, "prop4": data }</pre>	<pre>{ "prop1": data, "prop2": data, "prop3": data, "prop4": data }</pre>
Data	Data	Data	Data			
Data	Data	Data	Data			

Relational Vs. Document

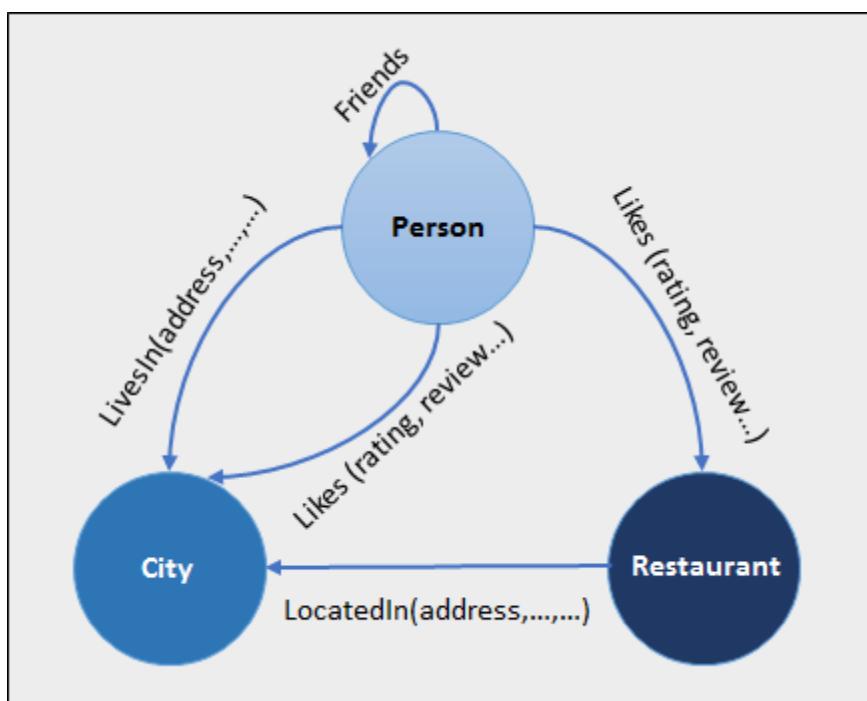
In this diagram on your left you can see we have rows and columns, and in the right, we have a document database which has a similar structure to JSON. Now for the relational database, you have to know what columns you have and so on. However, for a document database, you have data store like JSON object. You do not require to define which make it flexible.

The document type is mostly used for [CMS systems](#), blogging platforms, real-time analytics & e-commerce applications. It should not use for complex transactions which require multiple operations or queries against varying aggregate structures.

Amazon SimpleDB, CouchDB, MongoDB, Riak, Lotus Notes, MongoDB, are popular Document originated [DBMS systems](#).

Graph-Based

A graph type database stores entities as well the relations amongst those entities. The entity is stored as a node with the relationship as edges. An edge gives a relationship between nodes. Every node and edge has a unique identifier.



Compared to a relational database where tables are loosely connected, a Graph database is a multi-relational in nature. Traversing relationship is fast as they are already captured into the DB, and there is no need to calculate them.

Graph base database mostly used for social networks, logistics, spatial data.

Neo4J, Infinite Graph, OrientDB, FlockDB are some popular graph-based databases.

Query Mechanism tools for NoSQL

The most common data retrieval mechanism is the REST-based retrieval of a value based on its key/ID with GET resource

Document store Database offers more difficult queries as they understand the value in a key-value pair. For example, CouchDB allows defining views with MapReduce

What is the CAP Theorem?

CAP theorem is also called brewer's theorem. It states that is impossible for a distributed data store to offer more than two out of three guarantees

1. Consistency
2. Availability
3. Partition Tolerance

Consistency:

The data should remain consistent even after the execution of an operation. This means once data is written, any future read request should contain that data. For example, after updating the order status, all the clients should be able to see the same data.

Availability:

The database should always be available and responsive. It should not have any downtime.

Partition Tolerance:

Partition Tolerance means that the system should continue to function even if the communication among the servers is not stable. For example, the servers can be partitioned into multiple groups which may not communicate with each other. Here, if part of the database is unavailable, other parts are always unaffected.

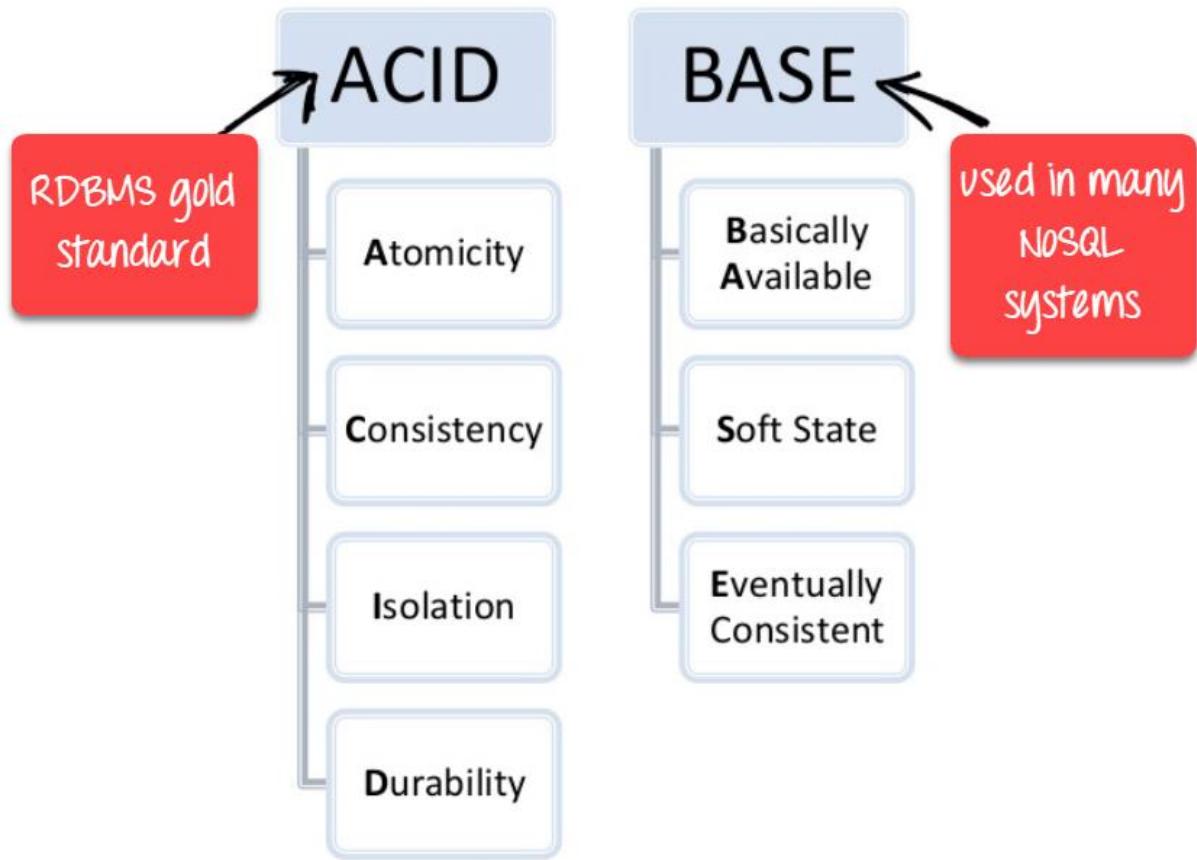
Eventual Consistency

The term “eventual consistency” means to have copies of data on multiple machines to get high availability and scalability. Thus, changes made to any data item on one machine has to be propagated to other replicas.

Data replication may not be instantaneous as some copies will be updated immediately while others in due course of time. These copies may be mutually, but in due course of time, they become consistent. Hence, the name eventual consistency.

BASE: Basically Available, Soft state, Eventual consistency

- Basically, available means DB is available all the time as per CAP theorem
- Soft state means even without an input; the system state may change
- Eventual consistency means that the system will become consistent over time



Advantages of NoSQL

- Can be used as Primary or Analytic Data Source
- Big Data Capability
- No Single Point of Failure
- Easy Replication
- No Need for Separate Caching Layer
- It provides fast performance and horizontal scalability.
- Can handle structured, semi-structured, and unstructured data with equal effect
- Object-oriented programming which is easy to use and flexible
- NoSQL databases don't need a dedicated high-performance server
- Support Key Developer Languages and Platforms
- Simple to implement than using RDBMS
- It can serve as the primary data source for online applications.
- Handles big data which manages data velocity, variety, volume, and complexity
- Excels at distributed database and multi-data center operations

- Eliminates the need for a specific caching layer to store data
- Offers a flexible schema design which can easily be altered without downtime or service disruption

Disadvantages of NoSQL

- No standardization rules
- Limited query capabilities
- RDBMS databases and tools are comparatively mature
- It does not offer any traditional database capabilities, like consistency when multiple transactions are performed simultaneously.
- When the volume of data increases it is difficult to maintain unique values as keys become difficult
- Doesn't work as well with relational data
- The learning curve is stiff for new developers
- Open source options so not so popular for enterprises.

Summary

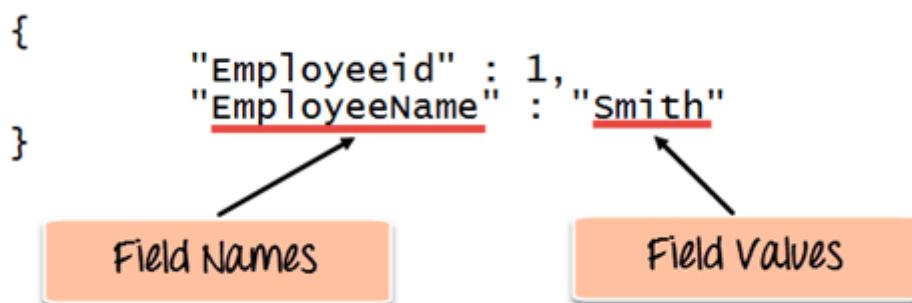
- NoSQL is a non-relational DMS, that does not require a fixed schema, avoids joins, and is easy to scale
- The concept of NoSQL databases became popular with Internet giants like Google, Facebook, Amazon, etc. who deal with huge volumes of data
- In the year 1998- Carlo Strozzi used the term NoSQL for his lightweight, open-source relational database
- NoSQL databases never follow the relational model it is either schema-free or has relaxed schemas
- Four types of NoSQL Database are 1). Key-value Pair Based 2). Column-oriented Graph 3). Graphs based 4). Document-oriented
- NoSQL can handle structured, semi-structured, and unstructured data with equal effect
- CAP theorem consists of three words Consistency, Availability, and Partition Tolerance

- BASE stands for **B**asically **A**vailable, **S**oft state, **E**ventual consistency
- The term “eventual consistency” means to have copies of data on multiple machines to get high availability and scalability
- **NOSQL** offer limited query capabilities

How to Create Database & Collection in MongoDB

In MongoDB, the first basic step is to have a database and collection in place. The database is used to store all of the collections, and the collection in turn is used to store all of the documents. The documents in turn will contain the relevant Field Name and Field values.

The snapshot below shows a basic example of how a document would look like.



The Field Names of the document are “Employeeid” and “EmployeeName” and the Field values are “1” and “Smith’ respectively. A bunch of documents would then make up a collection in MongoDB.

Creating a database using “use” command

Creating a database in MongoDB is as simple as issuing the “**using**” command. The following example shows how this can be done.

A screenshot of a terminal window titled "C:\Windows\system32\cmd.exe - mongo.exe". The command "use EmployeeDB" is being typed into the terminal. A green arrow points from an orange box labeled "Name of database" to the word "EmployeeDB" in the command line.

Code Explanation:

- The “**use**” command is used to create a database in [MongoDB](#). If the database does not exist a new one will be created.

If the command is executed successfully, the following Output will be shown:

Output:

```
C:\Windows\system32\cmd.exe - mongo.exe
> use EmployeeDB
switched to db EmployeeDB
>
```

Database will be created

MongoDB will automatically switch to the database once created.

Creating a Collection/Table using insert()

The easiest way to create a collection is to insert a record (which is nothing but a document consisting of Field names and Values) into a collection. If the collection does not exist a new one will be created.

The following example shows how this can be done.

```
db.Employee.insert
(
    {
        "Employeeid" : 1,
        "EmployeeName" : "Martin"
    }
)
```

Code Explanation:

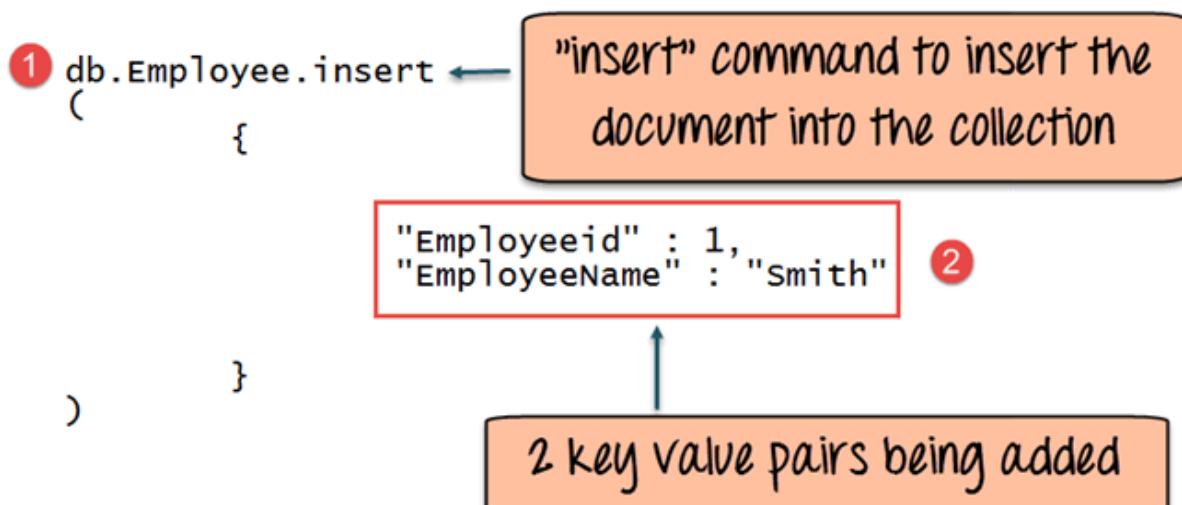
- As seen above, by using the “**insert**” command the collection will be created.

Adding documents using insert() command

MongoDB provides the **insert () command** to insert documents into a collection. The following example shows how this can be done.

Step 1) Write the “insert” command

Step 2) Within the “insert” command, add the required Field Name and Field Value for the document which needs to be created.



Code Explanation:

1. The first part of the command is the **“insert statement”** which is the statement used to insert a document into the collection.
2. The second part of the statement is to add the Field name and the Field value, in other words, what is the document in the collection going to contain.

If the command is executed successfully, the following Output will be shown

Output:

The screenshot shows a terminal window titled "C:\Windows\system32\cmd.exe - mongo.exe". The command entered is:

```
> db.Employee.insert({  
...   "  
...   "Employeeid" : 1,  
...   "EmployeeName" : "Smith"  
... }  
... );  
WriteResult({ "nInserted" : 1 })
```

An orange callout box with a black border and a white background is positioned below the terminal output. It contains the text: "The result shows that one document was added to the collection". A vertical arrow points from the word "WriteResult" in the terminal output up towards the callout box.

The output shows that the operation performed was an insert operation and that one record was inserted into the collection.

MongoDB Array of Objects using `insert()` with Example

The “insert” command can also be used to insert multiple documents into a collection at one time. The below code example can be used to insert multiple documents at a time.

The following example shows how this can be done,

Step 1) Create a [JavaScript](#) variable called myEmployee to hold the array of documents

Step 2) Add the required documents with the Field Name and values to the variable

Step 3) Use the insert command to insert the array of documents into the collection

```

var myEmployee=
[
    {
        "Employeeid" : 1,
        "EmployeeName" : "Smith"
    },
    {
        "Employeeid" : 2,
        "EmployeeName" : "Mohan"
    },
    {
        "Employeeid" : 3,
        "EmployeeName" : "Joe"
    }
];

db.Employee.insert(myEmployee);

```

If the command is executed successfully, the following Output will be shown

```

C:\Windows\system32\cmd.exe - mongo
> var myEmployee=
> [
>     {
>         "Employeeid" : 1,
>         "EmployeeName" : "Smith"
>     },
>     {
>         "Employeeid" : 2,
>         "EmployeeName" : "Mohan"
>     },
>     {
>         "Employeeid" : 3,
>         "EmployeeName" : "Joe"
>     }
> ];
> db.Employee.insert(myEmployee);
BulkwriteResult({
    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 3,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
}

```

The result shows that 3 documents were inserted into the collection

The output shows that those 3 documents were added to the collection.

Printing in JSON format

[JSON](#) is a format called **JavaScript Object Notation**, and is just a way to store information in an organized, easy-to-read manner. In our further

examples, we are going to use the JSON print functionality to see the output in a better format.

Let's look at an example of printing in JSON format

```
db.Employee.find().forEach(printjson)
```

Code Explanation:

1. The first change is to append the function called for Each() to the find() function. What this does is that it makes sure to explicitly go through each document in the collection. In this way, you have more control of what you can do with each of the documents in the collection.
2. The second change is to put the printjson command to the forEach statement. This will cause each document in the collection to be displayed in JSON format.

If the command is executed successfully, the following Output will be shown

Output:

```
> db.Employee.find().forEach(printjson);
{
  "_id" : ObjectId("563479cc8a8a4246bd27d784"),
  "Employeeid" : 1,
  "EmployeeName" : "Smith"
}
{
  "_id" : ObjectId("563479d48a8a4246bd27d785"),
  "Employeeid" : 2,
  "EmployeeName" : "Mohan"
}
{
  "_id" : ObjectId("563479df8a8a4246bd27d786"),
  "Employeeid" : 3,
  "EmployeeName" : "Joe"
}>
```

You will now see each document printed in json style

The output clearly shows that all of the documents are printed in JSON style.

Mongodb Primary Key: Example to set _id field with ObjectId()

What is Primary Key in MongoDB?

In MongoDB, _id field as the primary key for the collection so that each document can be uniquely identified in the collection. The _id field contains a unique ObjectId value.

By default when inserting documents in the collection, if you don't add a field name with the _id in the field name, then MongoDB will automatically add an Object id field as shown below

```
> db.Employee.find().forEach(printjson);
{
    "_id" : ObjectId("563479cc8a8a4246bd27d784"),
    "Employeeid" : 1,
    "EmployeeName" : "Smith"
}
{
    "_id" : ObjectId("563479d48a8a4246bd27d785"),
    "Employeeid" : 2,
    "EmployeeName" : "Mohan"
}
{
    "_id" : ObjectId("563479df8a8a4246bd27d786"),
    "Employeeid" : 3,
    "EmployeeName" : "Joe"
}>
```

every row has a unique object id

When you query the documents in a collection, you can see the ObjectId for each document in the collection.

If you want to ensure that [MongoDB](#) does not create the _id Field when the collection is created and if you want to specify your own id as the _id of the collection, then you need to explicitly define this while creating the collection.

When explicitly creating an id field, it needs to be created with _id in its name.

Let's look at an example on how we can achieve this.

```
db.Employee.insert({_id:10, "EmployeeName" : "Smith"})
```

Code Explanation:

1. We are assuming that we are creating the first document in the collection and hence in the above statement while creating the collection, we explicitly define the field `_id` and define a value for it.

If the command is executed successfully and now use the find command to display the documents in the collection, the following Output will be shown

Output:

```
C:\Windows\System32\cmd.exe - mongo.exe
> db.Employee.insert({ _id:10 , "EmployeeName" : "Smith" })
writeResult({ "nInserted" : 1 })
> db.Employee.find()
{ "_id" : 10, "EmployeeName" : "Smith" }
>
```

You will not see the ObjectId field value now
, but the value we specified is now the id
for the document

The output clearly shows that the `_id` field we defined while creating the collection is now used as the primary key for the collection.

MongoDB Query Document: **db.collection.find()** with Example

The method of fetching or getting data from a MongoDB database is carried out by using MongoDB queries. While performing a query operation, one can also use criteria's or conditions which can be used to retrieve specific data from the database.

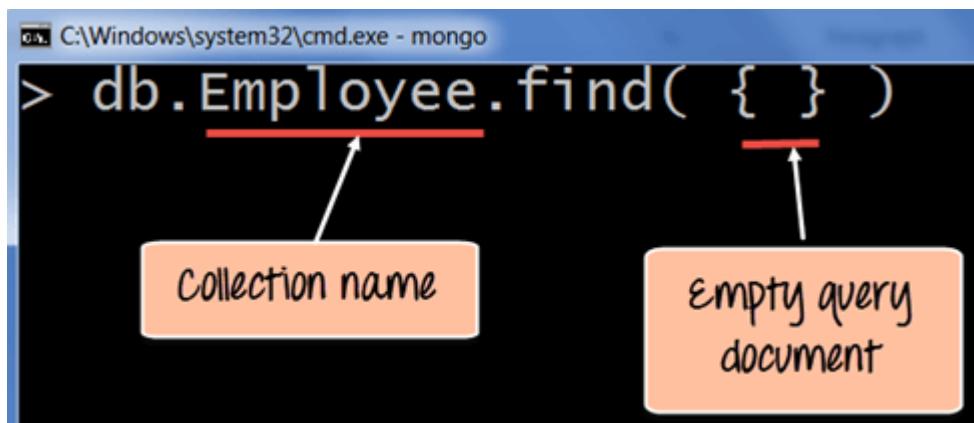
[MongoDB](#) provides a function called **db.collection.find()** which is used for retrieval of documents from a MongoDB database.

During the course of this MongoDB query tutorial, you will see how this function is used in various ways to achieve the purpose of document retrieval.

MongoDB Basic Query Operations

The basic MongoDB query operators cover the simple operations such as getting all of the documents in a MongoDB collection. Let's look at an db.collection.find example of how we can accomplish this.

All of our code will be run in the MongoDB [JavaScript](#) command shell. Consider that we have a collection named 'Employee' in our MongoDB database and we execute the below command.



C:\Windows\system32\cmd.exe - mongo
> db.Employee.find({})

Collection name Empty query document

MongoDB Basic query operation

Code Explanation:

1. Employee is the collection name in the MongoDB database
2. The MongoDB find query is an in-built function which is used to retrieve the documents in the collection.

If the command is executed successfully, the following Output will be shown for the MongoDB find example

Output:

```

> db.Employee.find().forEach(printjson);
{
    "_id" : ObjectId("563479cc8a8a4246bd27d784"),
    "Employeeid" : 1,
    "EmployeeName" : "Smith"
}

{
    "_id" : ObjectId("563479d48a8a4246bd27d785"),
    "Employeeid" : 2,
    "EmployeeName" : "Mohan"
}

{
    "_id" : ObjectId("563479df8a8a4246bd27d786"),
    "Employeeid" : 3,
    "EmployeeName" : "Joe"
}

```

Find command returns all of the documents in the collection

The output shows all the documents which are present in the collection.

We can also add criteria to our queries so that we can fetch documents based on certain conditions.

MongoDB Query Example – 1

Let's look at a couple of MongoDB query examples of how we can accomplish this.

```
db.Employee.find({EmployeeName : "Smith"}).forEach(printjson);
```

Code Explanation:

1. Here we want to find for an Employee whose name is “Smith” in the collection , hence we enter the filter criteria as EmployeeName : “Smith”

If the command is executed successfully, the following Output will be shown

Output:

```

C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.find({EmployeeName : "Smith"}).forEach(printjson);
{
    "_id" : ObjectId("563479cc8a8a4246bd27d784"),
    "Employeeid" : 1,
    "EmployeeName" : "Smith"
}

```

The document which contains EmployeeName as Smith is returned

The output shows that only the document which contains “Smith” as the Employee Name is returned.

MongoDB Query Example – 2

Now in this MongoDB queries tutorial, let's take a look at another code example which makes use of the greater than search criteria. When this criteria is included, it actually searches those documents where the value of the field is greater than the specified value.

```
db.Employee.find({Employeeid : {$gt:2}}).forEach(printjson);
```

Code Explanation:

1. Here we want to find for all Employee's whose id is greater than 2. The \$gt is called a query selection operator, and what it just means is to use the greater than expression.

If the MongoDB select fields command is executed successfully, the following Output will be shown

Output:

A screenshot of a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe - mongo.exe". The command entered is "db.Employee.find({Employeeid : { \$gt : 2}}).forEach(printjson);". The output shows one document: {_id: ObjectId("563479df8a8a4246bd27d786"), Employeeid: 3, EmployeeName: "Joe"}. A red box highlights this document. An orange callout box with a white border and black text points to the document, containing the text "All documents with employeeid greater than 2 are returned".

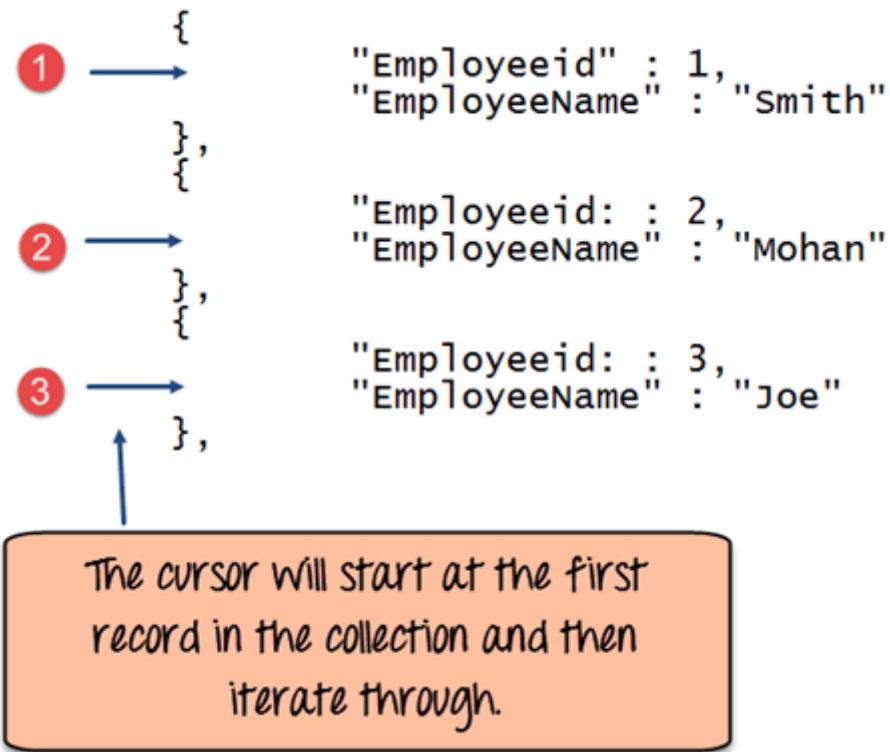
All of the documents wherein the Employee id is greater than 2 is returned.

Cursor in MongoDB

What is Cursor in MongoDB?

When the **db.collection.find ()** function is used to search for documents in the collection, the result returns a pointer to the collection of documents returned which is called a cursor.

By default, the cursor will be iterated automatically when the result of the query is returned. But one can also explicitly go through the items returned in the cursor one by one. If you see the below example, if we have 3 documents in our collection, the cursor object will point to the first document and then iterate through all of the documents of the collection.



The following example shows how this can be done.

```
var myEmployee = db.Employee.find( { Employeeid : { $gt:2 } }) ;  
while(myEmployee.hasNext())  
{  
    print(tojson(myEmployee.next()));  
}
```

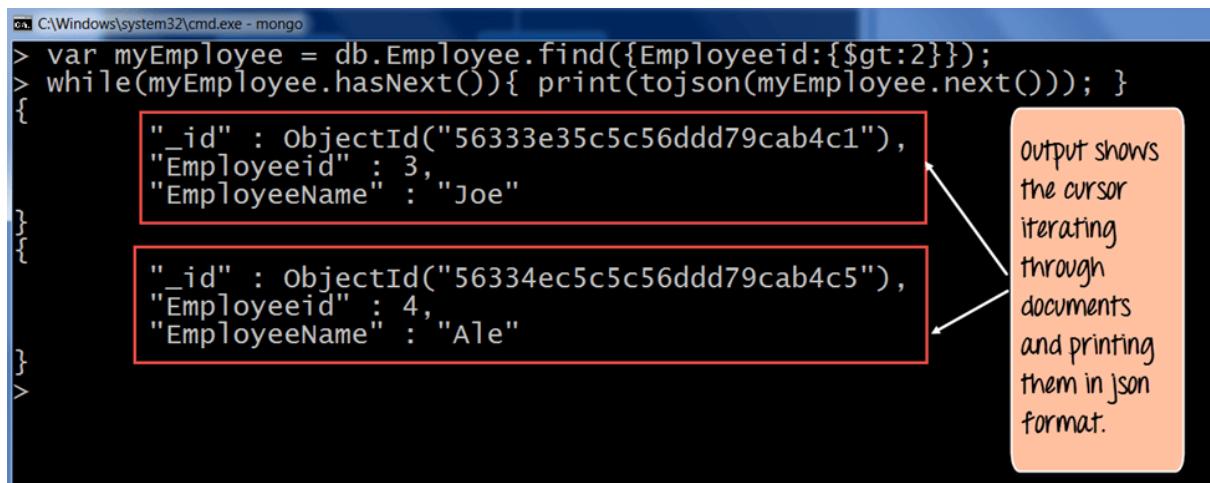
Code Explanation

1. First we take the result set of the query which finds the Employee's whose id is greater than 2 and assign it to the **JavaScript** variable 'myEmployee'
2. Next we use the while loop to iterate through all of the documents which are returned as part of the query.

- Finally for each document, we print the details of that document in **JSON** readable format.

If the command is executed successfully, the following Output will be shown

Output



```
C:\Windows\system32\cmd.exe - mongo
> var myEmployee = db.Employee.find({Employeeid:{$gt:2}});
> while(myEmployee.hasNext()){ print(tojson(myEmployee.next())); }
{
  "_id" : ObjectId("56333e35c5c56ddd79cab4c1"),
  "Employeeid" : 3,
  "EmployeeName" : "Joe"
}
{
  "_id" : ObjectId("56334ec5c5c56ddd79cab4c5"),
  "Employeeid" : 4,
  "EmployeeName" : "Ale"
}>
```

Output shows the cursor iterating through documents and printing them in json format.

MongoDB Sort() & Limit() Query with Order By Examples

What is Query Modifications?

[Mongo DB](#) provides query modifiers such as the ‘limit’ and ‘Orders’ clause to provide more flexibility when executing queries. We will take a look at the following query modifiers

MongoDB Limit Query Results

This modifier is used to limit the number of documents which are returned in the result set for a query. The following example shows how this can be done.

```
db.Employee.find().limit(2).forEach(printjson);
```

Code Explanation

- The above code takes the find function which returns all of the documents in the collection but then uses the limit clause to limit the number of documents being returned to just 2.

Output

If the command is executed successfully, the following Output will be shown

```
C:\Windows\System32\cmd.exe - mongo.exe
> db.Employee.find().limit(2).forEach(printjson);
{
  "_id" : ObjectId("563479cc8a8a4246bd27d784"),
  "Employeeid" : 1,
  "EmployeeName" : "Smith"
}
{
  "_id" : ObjectId("563479d48a8a4246bd27d785"),
  "Employeeid" : 2,
  "EmployeeName" : "Mohan"
}>
```

The output clearly shows that since there is a limit modifier, so at most just 2 records are returned as part of the result set based on the ObjectId in ascending order.

MongoDB Sort by Descending Order

One can specify the order of documents to be returned based on ascending or descending order of any key in the collection. The following example shows how this can be done.

```
db.Employee.find().sort({Employeeid:-1}).forEach(printjson)
```

Code Explanation

- The above code takes the sort function which returns all of the documents in the collection but then uses the modifier to change the order in which the records are returned. Here the -1 indicates that we want to return the documents based on the descending order of Employee id.

If the command is executed successfully, the following Output will be shown

Output



```
C:\Windows\System32\cmd.exe - mongo.exe
> db.Employee.find().sort({Employeeid : -1}).forEach(printjson);
{
  "_id" : ObjectId("563479df8a8a4246bd27d786"),
  "Employeeid" : 3,
  "EmployeeName" : "Joe"
}
{
  "_id" : ObjectId("563479d48a8a4246bd27d785"),
  "Employeeid" : 2,
  "EmployeeName" : "Mohan"
}
{
  "_id" : ObjectId("563479cc8a8a4246bd27d784"),
  "Employeeid" : 1,
  "EmployeeName" : "Smith"
}
```

A callout bubble points to the middle document in the results, containing the text: "can see that the documents are returned as Employeeid descending order".

The output clearly shows the documents being returned in descending order of the Employeeid.

Ascending order is defined by value 1.

Count() & Remove() Functions in MongoDB with Example

MongoDB Count() Function

The concept of aggregation is to carry out a computation on the results which are returned in a query. For example, suppose you wanted to know what is the count of documents in a collection as per the query fired, then MongoDB provides the count() function.

Example of MongoDB Count() Function

Let's look at an example of this.

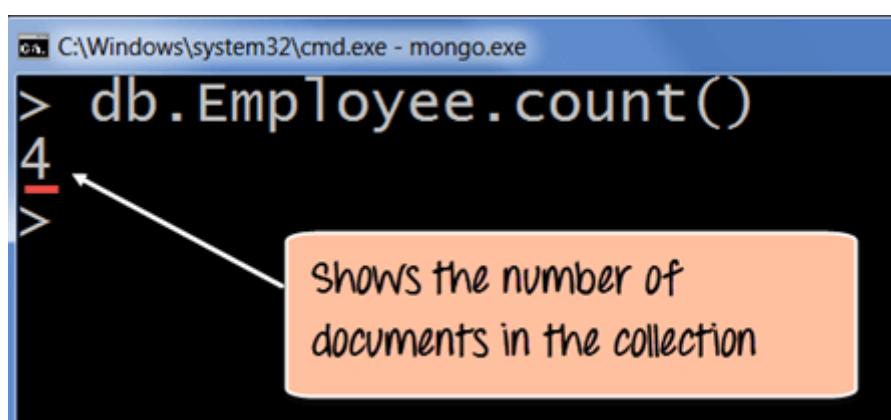
```
db.Employee.count()
```

Code Explanation:

- The above code executes the count function.

If the command is executed successfully, the following Output will be shown

Output:



A screenshot of a terminal window titled "C:\Windows\system32\cmd.exe - mongo.exe". The command "db.Employee.count()" is entered and the output "4" is displayed. A red arrow points from the number "4" to a callout box containing the text "Shows the number of documents in the collection".

The output clearly shows that 4 documents are there in the collection.

Performing Modifications

The other two classes of operations in MongoDB are the update and remove statements.

The update operations allow one to modify existing data, and the remove operations allow the deletion of data from a collection.

Remove() Function in MongoDB

In [MongoDB](#), the **db.collection.remove()** method is used to remove documents from a collection. Either all of the documents can be removed from a collection or only those which matches a specific condition.

If you just issue the remove command, all of the documents will be removed from the collection.

Example of MongoDB Remove() Function

The following code example demonstrate how to remove a specific document from the collection.

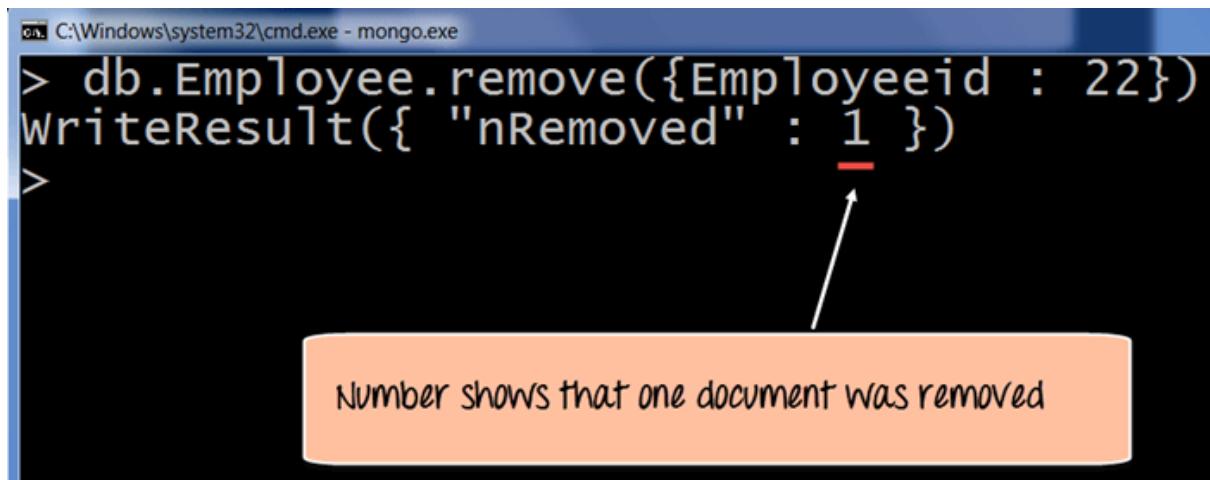
```
db.Employee.remove( {Employeeid:22} )
```

Code Explanation:

- The above code use the remove function and specifies the criteria which in this case is to remove the documents which have the Employee id as 22.

If the command is executed successfully, the following Output will be shown

Output:



```
C:\Windows\system32\cmd.exe - mongo.exe
> db.Employee.remove({Employeeid : 22})
WriteResult({ "nRemoved" : 1 })
>
```

Number shows that one document was removed

The output will show that 1 document was modified.

MongoDB Update() Document with Example

Basic document updates

MongoDB provides the update() command to update the documents of a collection. To update only the documents you want to update, you can add a criteria to the update statement so that only selected documents are updated.

The basic parameters in the command is a condition for which document needs to be updated, and the next is the modification which needs to be performed.

The following example shows how this can be done.

Step 1) Issue the update command

Step 2) Choose the condition which you want to use to decide which document needs to be updated. In our example, we want to update the document which has the Employee id 22.

Step 3) Use the set command to modify the Field Name

Step 4) Choose which Field Name you want to modify and enter the new value accordingly.

```
db.Employee.update(  
  { "Employeeid" : 1 },  
  { $set: { "EmployeeName" : "NewMartin" } } );
```

If the command is executed successfully, the following Output will be shown

Output:

```
C:\Windows\system32\cmd.exe - mongo.exe  
> db.Employee.update(  
... { "Employeeid" : 1 } ,  
... { $set: { "EmployeeName" : "NewMartin" } } );  
writeResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })  
>
```

Output shows that one record matched the filter criteria and hence one record was modified.

The output clearly shows that one record matched the condition and hence the relevant field value was modified.

Updating Multiple Values

To ensure that multiple/bulk documents are updated at the same time in [MongoDB](#) you need to use the multi option because otherwise by default only one document is modified at a time.

The following example shows how to update many documents.

In this example, we are going to first find the document which has the Employee id as “1” and change the Employee name from “Martin” to “NewMartin”

Step 1) Issue the update command

Step 2) Choose the condition which you want to use to decide which document needs to be updated. In our example, we want the document which has the Employee id of “1” to be updated.

Step 3) Choose which Field Name's you want to modify and enter their new value accordingly.

```
db.Employee.update
(
{
    Employeeid : 1
},
{
    $set :
    {
        "EmployeeName" : "NewMartin",
        "Employeeid" : 22
    }
}
)
```

If the command is executed successfully and if you run the “**find**” command to search for the document with Employee id as 22 you will see the following Output will be shown

Output:

```
C:\Windows\System32\cmd.exe - mongo.exe
> db.Employee.update(
... { "Employeeid" : 1 },
... { $set :
... { "EmployeeName" : "NewMartin" , "Employeeid" : 22
... }})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.Employee.find({ "Employeeid" : 22 }).forEach(printjson)
{
    "_id" : ObjectId("563479cc8a8a4246bd27d784"),
    "Employeeid" : 22,
    "EmployeeName" : "NewMartin"
}
>
```

You will see that both changes have been made.

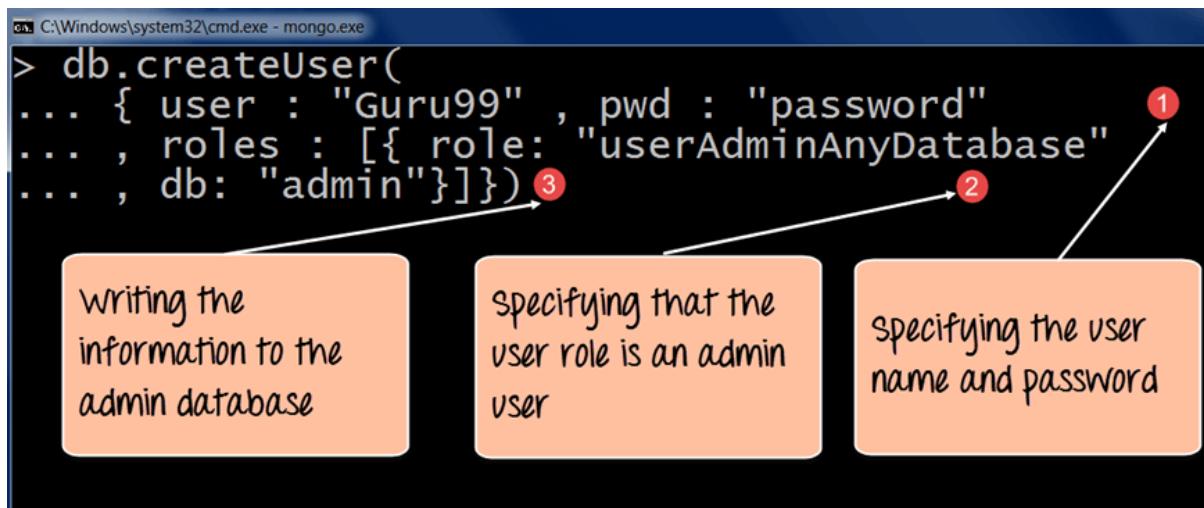
The output clearly shows that one record matched the condition and hence the relevant field value was modified.

How to Create User & add Role in MongoDB

MongoDB Create Administrator User

Creating a user administrator in MongoDB is done by using the `createUser` method. The following example shows how this can be done.

```
db.createUser(  
{  
    user: "Guru99",  
    pwd: "password",  
  
    roles:[{role: "userAdminAnyDatabase" , db:"admin"}] })
```



```
C:\Windows\system32\cmd.exe - mongo.exe  
> db.createUser(  
... { user : "Guru99" , pwd : "password"  
... , roles : [{ role: "userAdminAnyDatabase"  
... , db: "admin"}]} ) ③
```

1 Specifying the user name and password

2 Specifying that the user role is an admin user

3 Writing the information to the admin database

Code Explanation:

1. The first step is to specify the “username” and “password” which needs to be created.
2. The second step is to assign a role for the user. Since it needs to be a database administrator in which case we have assigned to the “userAdminAnyDatabase” role. This role allows the user to have administrative privileges to all databases in MongoDB.
3. The `db` parameter specifies the admin database which is a special Meta database within MongoDB which holds the information for this user.

If the command is executed successfully, the following Output will be shown:

Output:

```
> db.createUser(  
... { user : "Guru99" , pwd : "password"  
... , roles : [{ role: "userAdminAnyDatabase"  
... , db: "admin"}]})  
Successfully added user: {  
    "user" : "Guru99",  
    "roles" : [  
        {  
            "role" : "userAdminAnyDatabase",  
            "db" : "admin"  
        }  
    ]  
}  
>
```

shows the user
"guru99" created

shows the role
which was assigned

The output shows that a user called “Guru99” was created and that user has privileges over all the databases in MongoDB.

MongoDB Create User for Single Database

To create a user who will manage a single database, we can use the same command as mentioned above but we need to use the “userAdmin” option only.

The following example shows how this can be done;

```
C:\Windows\system32\cmd.exe - mongo.exe  
> db.createUser(  
... ① { user : "Employeeadmin" , pwd : "password"  
... , roles : [{role: "userAdmin" ②}  
... , db: "Employee"]}) ③
```

We are creating an administrator only in the Employee database

We are using the userAdmin role

```
db.createUser(  
{  
    user: "Employeeadmin",
```

```
pwd: "password",
roles:[{role: "userAdmin" , db:"Employee"}]})
```

Code Explanation:

1. The first step is to specify the “username” and “password” which needs to be created.
2. The second step is to assign a role for the user which in this case since it needs to be a database administrator is assigned to the “userAdmin” role. This role allows the user to have administrative privileges only to the database specified in the db option.
3. The db parameter specifies the database to which the user should have administrative privileges on.

If the command is executed successfully, the following Output will be shown:

Output:

```
C:\Windows\System32\cmd.exe - mongo.exe
> db.createUser(
... { user : "Employeeadmin" , pwd : "password"
... ,roles : [{role: "userAdmin"
... ,db: "Employee"}]})
```

Successfully added user: {
 "user" : "Employeeadmin", ← shows the user created
 "roles" : [
 {
 "role" : "userAdmin",
 "db" : "Employee"
 }
]
}

shows the role which was assigned to the user and to which database

The output shows that a user called “Employeeadmin” was created and that user has privileges only on the “Employee” database.

Managing users

First understand the roles which you need to define. There is a whole list of role available in [MongoDB](#). For example, there is a the “read role” which only allows read only access to databases and then there is the “readwrite” role which provides read and write access to the database , which means that the user can issue the insert, delete and update commands on collections in that database.

```
db.createUser(  
{  
    user: "Mohan",  
    pwd: "password",  
    roles:[  
        {  
            role: "read" , db:"Marketing"},  
            role: "readwrite" , db:"sales"}  
    ]  
})
```

Specifying the different roles for the user

```
db.createUser(  
{  
    user: "Mohan",  
    pwd: "password",  
    roles:[  
        {  
            role: "read" , db:"Marketing"},  
            {  
                role: "readWrite" , db:"Sales"}  
        ]  
    })
```

The above code snippet shows that a user called Mohan is created, and he is assigned multiple roles in multiple databases. In the above example, he is given read only permission to the “Marketing” database and readWrite permission to the “Sales” database.



Spark SQL

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

Relational Databases

SQL is the lingua franca for doing analytics.

Relational Databases

SQL is the lingua franca for doing analytics.

But it's a pain in the neck to connect big data processing pipelines like Spark or Hadoop to an SQL database.

Wouldn't it be nice...

- ▶ if it were possible to seamlessly intermix SQL queries with Scala?
- ▶ to get all of the optimizations we're used to in the databases community on Spark jobs?

Relational Databases

SQL is the lingua franca for doing analytics.

But it's a pain in the neck to connect big data processing pipelines like Spark or Hadoop to an SQL database.

Wouldn't it be nice...

- ▶ if it were possible to seamlessly intermix SQL queries with Scala?
- ▶ to get all of the optimizations we're used to in the databases community on Spark jobs?

Spark SQL delivers both!

Spark SQL: Goals

Three main goals:

1. Support relational processing both within Spark programs (on RDDs) and on external data sources with a friendly API.

Resilient Distributed Dataset

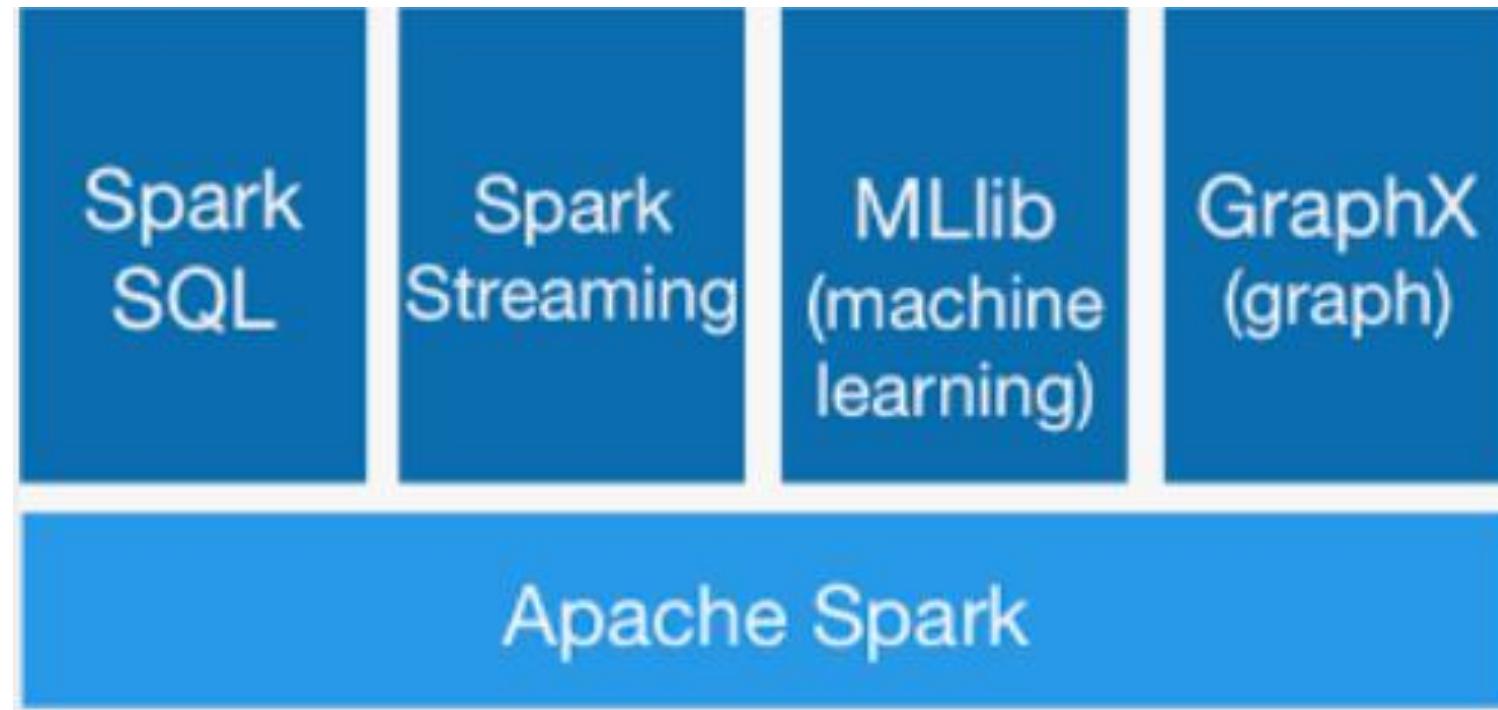
Sometimes it's more desirable to express a computation in SQL syntax than with functional APIs and vice versa.

Spark SQL: Goals

Three main goals:

1. Support **relational processing** both within Spark programs (on RDDs) and on external data sources with a friendly API.
2. High performance, achieved by using techniques from research in databases.
3. Easily support new data sources such as semi-structured data and external databases.
JSON

Spark Stack Revisited



Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Spark SQL

Spark SQL is a component of the Spark stack.

- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Three main APIs:

- ▶ **SQL literal syntax**
- ▶ **DataFrames**
- ▶ **Datasets**

Spark SQL

Two specialized backend components:

- ▶ **Catalyst**, query optimizer.
- ▶ **Tungsten**, off-heap serializer.

Spark SQL

Spark SQL is a component of the Spark stack.

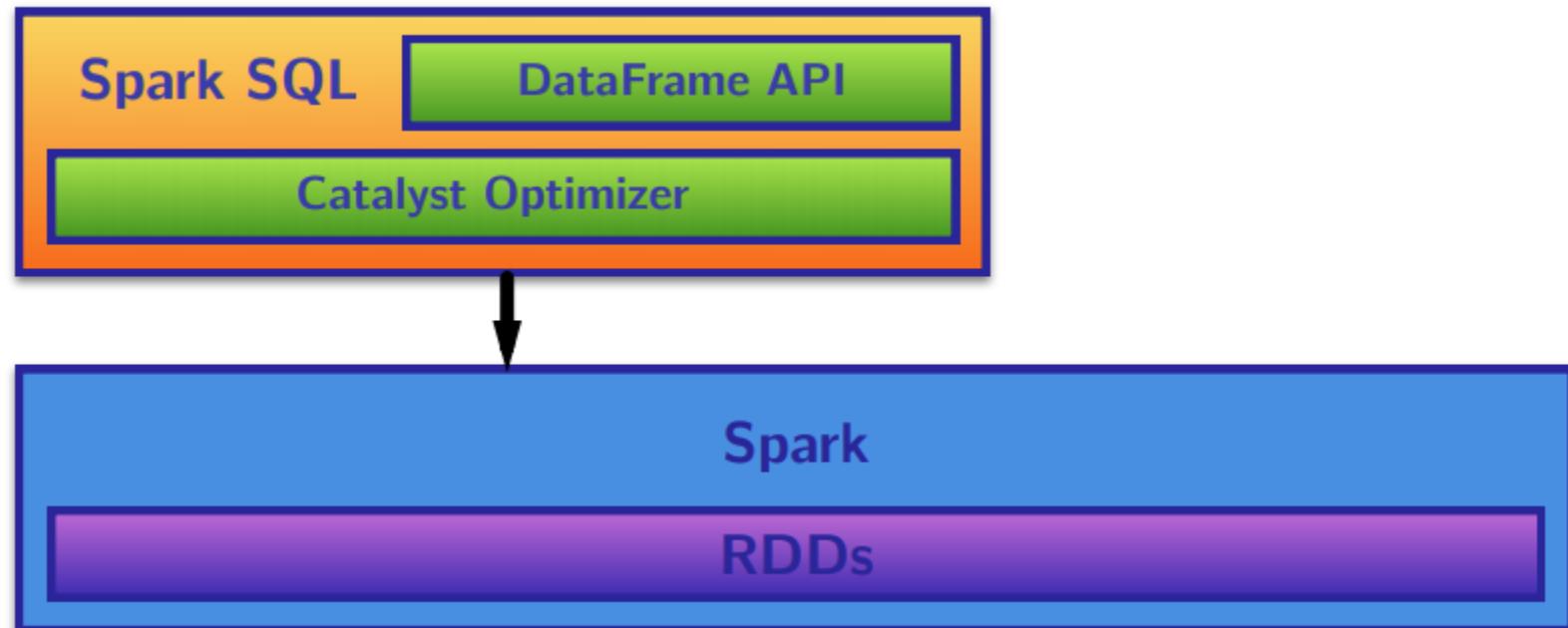
- ▶ It is a Spark module for structured data processing.
- ▶ It is implemented as a library on top of Spark.

Visually, Spark SQL relates to the rest of Spark like this:



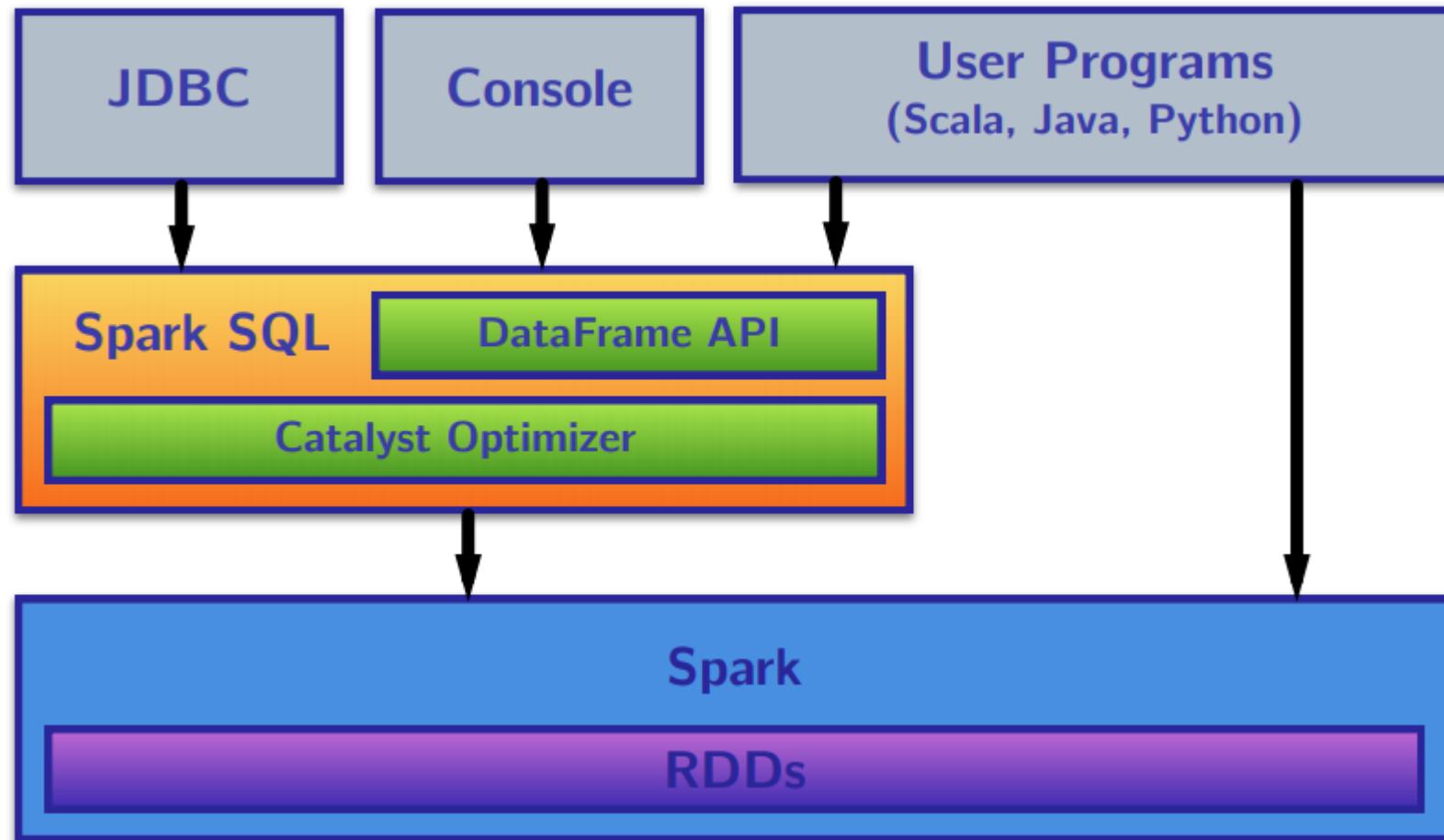
Spark SQL

Visually, Spark SQL relates to the rest of Spark like this:



Spark SQL

Visually, Spark SQL relates to the rest of Spark like this:



Relational Queries(SQL)

Everything about SQL is structured.

In fact, SQL stands for *structural query language*.

- ▶ There are a set of fixed data types. Int, Long, String, etc.
- ▶ There are fixed set of operations. SELECT, WHERE, GROUP BY, etc.

Research and industry surrounding relational databases has focused on exploiting this rigidness to get all kinds of performance speedups.

Relational Queries(SQL)

Everything about SQL is structured.

In fact, SQL stands for *structural query language*.

- ▶ There are a set of fixed data types. Int, Long, String, etc.
- ▶ There are fixed set of operations. SELECT, WHERE, GROUP BY, etc.

Research and industry surrounding relational databases has focused on exploiting this rigidness to get all kinds of performance speedups.

Let's quickly establish a common set of vocabulary and a baseline understanding of SQL.

Relational Queries(SQL)

Data organized into one or more **tables**

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries(SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.

columns

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries(SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.

rows

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries(SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.
- ▶ Tables typically represent a collection of objects of a certain type, such as **customers** or **products**

SBB customers dataset

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries(SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.
- ▶ Tables typically represent a collection of objects of a certain type, such as **customers** or **products**

A *relation* is just a table.

Attributes are columns.

attribute

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Relational Queries(SQL)

Data organized into one or more **tables**

- ▶ Tables contain **columns** and **rows**.
- ▶ Tables typically represent a collection of objects of a certain type, such as **customers** or **products**

A *relation* is just a table.

Attributes are columns.

Rows are *records* or *tuples*

record
/tuple

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records **with a known schema**

Unlike RDDs though, DataFrames **require** some kind of schema info!

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

DataFrames are **untyped!**

That is, the Scala compiler doesn't check the types in its schema!

DataFrames contain Rows which can contain any schema.

DataFrame, is a table, sort of.

Customer _ Name	Destination	Ticket _ Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

Spark SQL

DataFrame is Spark SQL's core abstraction.

Conceptually equivalent to a table in a relational database.

DataFrames are, *conceptually*, RDDs full of records with a known schema

DataFrames are **untyped!**

That is, the Scala compiler doesn't check the types in its schema!

Transformations on DataFrames are also known as **untyped transformations**

DataFrame, is a table, sort of.

Customer_Name	Destination	Ticket_Price
"Weitz"	"Luzern"	53.20
"Schinz"	"Zürich"	32.40
"Dubois"	"Neuchâtel"	12.50
"Hug"	"Basel"	32.10
"Strub"	"Winterthur"	9.60
"Chapuis"	"Lausanne"	6.60
"Smith"	"Genève"	12.70
"Weitz"	"Bern"	21.40

SparkSession

To get started using Spark SQL, everything starts with the `SparkSession`

SparkSession

To get started using Spark SQL, everything starts with the SparkSession

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("My App")
  // .config("spark.some.config.option", "some-value")
  .getOrCreate()
```

Creating DataFrames

DataFrames can be created in two ways:

1. From an existing RDD.

Either with schema inference, or with an explicit schema.

2. Reading in a specific **data source** from file.

Common structured or semi-structured formats such as JSON.

Creating DataFrames

(1a) Create DataFrame from RDD, schema reflectively inferred

Given pair RDD, RDD[(T1, T2, ... TN)], a DataFrame can be created with its schema automatically inferred by simply using the toDF method.

```
val tupleRDD = ... // Assume RDD[Int, String String, String]  
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use toDF without arguments, Spark will assign numbers as attributes (column names) to your DataFrame.

Creating DataFrames

(1a) Create DataFrame from RDD, schema reflectively inferred

Given pair RDD, RDD[(T1, T2, ... TN)], a DataFrame can be created with its schema automatically inferred by simply using the toDF method.

```
val tupleRDD = ... // Assume RDD[(Int, String, String, String)]
val tupleDF = tupleRDD.toDF("id", "name", "city", "country") // column names
```

Note: if you use toDF without arguments, Spark will assign numbers as attributes (column names) to your DataFrame.

If you already have an RDD containing some kind of case class instance, then Spark can infer the attributes from the case class's fields.

```
case class Person(id: Int, name: String, city: String)
val peopleRDD = ... // Assume RDD[Person]
val peopleDF = peopleRDD.toDF
```

Creating DataFrames

(1b) Create DataFrame from existing RDD, schema explicitly specified

Sometimes it's not possible to create a DataFrame with a pre-determined case class as its schema. For these cases, it's possible to explicitly specify a schema.

It takes three steps:

- ▶ Create an RDD of Rows from the original RDD.
- ▶ Create the schema represented by a StructType matching the structure of Rows in the RDD created in Step 1.
- ▶ Apply the schema to the RDD of Rows via `createDataFrame` method provided by `SparkSession`.

Given:

```
case class Person(name: String, age: Int)  
val peopleRdd = sc.textFile(...) // Assume RDD[Person]
```

Creating DataFrames

(1b) Create DataFrame from existing RDD, schema explicitly specified

```
// The schema is encoded in a string
val schemaString = "name age"

// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)

// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
  .map(_.split(","))
  .map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)
```

Creating DataFrames

(2) Create DataFrame by reading in a data source from file.

Using the SparkSession object, you can read in semi-structured/structured data by using the `read` method. For example, to read in data and infer a schema from a JSON file:

```
// 'spark' is the SparkSession object we created a few slides back  
val df = spark.read.json("examples/src/main/resources/people.json")
```

Creating DataFrames

(2) Create DataFrame by reading in a data source from file.

Using the SparkSession object, you can read in semi-structured/structured data by using the read method. For example, to read in data and infer a schema from a JSON file:

```
// 'spark' is the SparkSession object we created a few slides back  
val df = spark.read.json("examples/src/main/resources/people.json")
```

Semi-structured/Structured data sources Spark SQL can directly create DataFrames from:

- ▶ JSON
- ▶ CSV
- ▶ Parquet
- ▶ JDBC

*To see a list of all available methods for directly reading in semi-structured/structured data, see the latest API docs for DataFrameReader:
<http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.DataFrameReader>*

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

Given:

A DataFrame called peopleDF, we just have to register our DataFrame as a temporary SQL view first:

```
// Register the DataFrame as a SQL temporary view  
peopleDF.createOrReplaceTempView("people")  
// This essentially gives a name to our DataFrame in SQL  
// so we can refer to it in an SQL FROM statement
```

SQL Literals

Once you have a DataFrame to operate on, you can now freely write familiar SQL syntax to operate on your dataset!

Given:

A DataFrame called peopleDF, we just have to register our DataFrame as a temporary SQL view first:

```
// Register the DataFrame as a SQL temporary view
peopleDF.createOrReplaceTempView("people")
// This essentially gives a name to our DataFrame in SQL
// so we can refer to it in an SQL FROM statement

// SQL literals can be passed to Spark SQL's sql method
val adultsDF
= spark.sql("SELECT * FROM people WHERE age > 17")
```

SQL Literals

The SQL statements available to you are largely what's available in HiveQL. This includes standard SQL statements such as:

- ▶ SELECT
- ▶ FROM
- ▶ WHERE
- ▶ COUNT
- ▶ HAVING
- ▶ GROUP BY
- ▶ ORDER BY
- ▶ SORT BY
- ▶ DISTINCT
- ▶ JOIN
- ▶ (LEFT|RIGHT|FULL) OUTER JOIN
- ▶ Subqueries: `SELECT col FROM (SELECT a + b AS col from t1) t2`

Supported Spark SQL syntax:

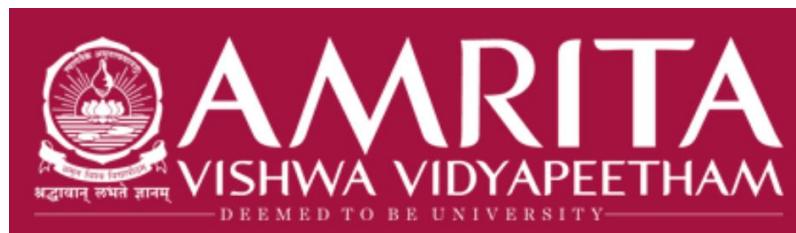
https://docs.datastax.com/en/archived/datastax_enterprise/4.6/datastax_enterprise/spark/sparkSqlSupportedSyntax.html

For a HiveQL cheatsheet:

For an updated list of supported Hive features in Spark SQL, the official Spark SQL docs enumerate:

<https://spark.apache.org/docs/latest/sql-programming-guide.html#supported-hive-features>

- <https://phoenixnap.com/kb/install-spark-on-windows-10>



Spark SQL

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

User-defined Function

- Spark SQL has language integrated User-Defined Functions (UDFs).
- UDF is a feature of Spark SQL to define new Column-based functions that extend the vocabulary of Spark SQL's DSL for transforming Datasets.

Q)Write the code to define a UDF to convert a given text to upper case.

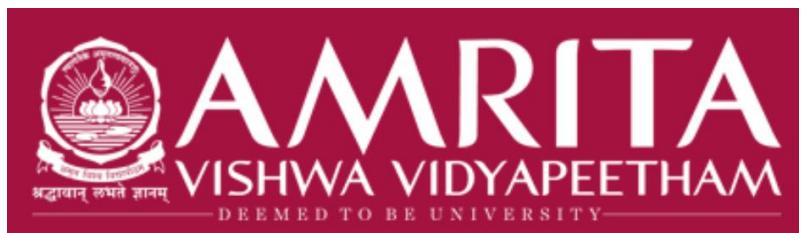
- Creating a dataset “hello world”
- val dataset = Seq((0, "hello"),(1, "world")).toDF("id","text")
- Defining a function ‘upper’ which converts a string into upper case.
- val upper: String => String =_.toUpperCase
- We now import the ‘udf’ package into Spark.
- import org.apache.spark.sql.functions.udf
- Defining our UDF, ‘upperUDF’ and importing our function ‘upper’.
- val upperUDF = udf(upper)
- Displaying the results of our User Defined Function in a new column ‘upper’.
- dataset.withColumn("upper", upperUDF('text)).show

```
scala> val upper: String => String = _.toUpperCase
upper: String => String = $Lambda$2362/933824459@15f8b713

scala> import org.apache.spark.sql.functions.udf
import org.apache.spark.sql.functions.udf

scala> val upperUDF = udf(upper)
upperUDF: org.apache.spark.sql.expressions.UserDefinedFunction = SparkUserDefinedFunction($Lambda$2362/933824459@15f8b713,
StringType,List(Some(class[value[0]: string])),Some(class[value[0]: string]),None,true,true)

scala> dataset.withColumn("upper", upperUDF('text)).show
+---+---+
| id| text|upper|
+---+---+
| 0|hello|HELLO|
| 1|world|WORLD|
+---+---+
```



DataFrame

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

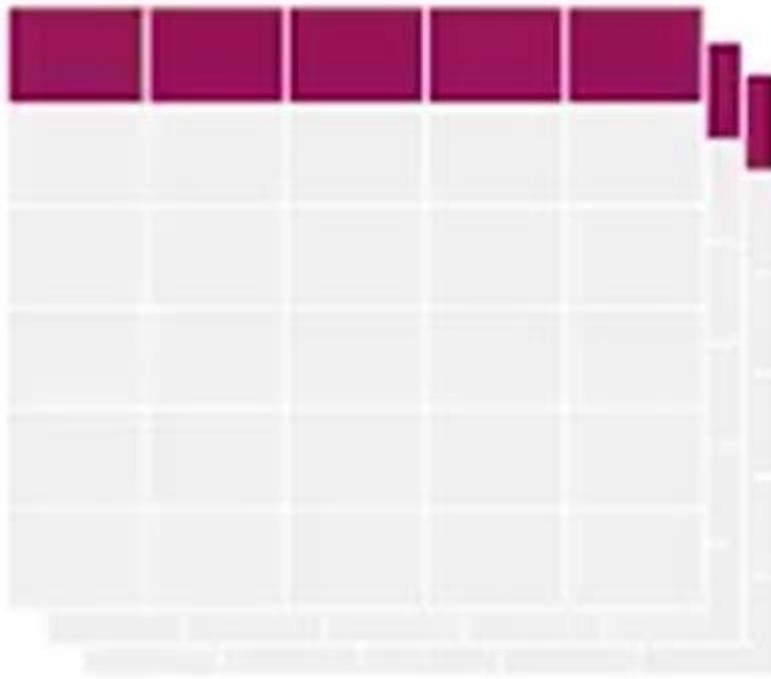
Assistant Professor(Selection Grade)

Department of CSE

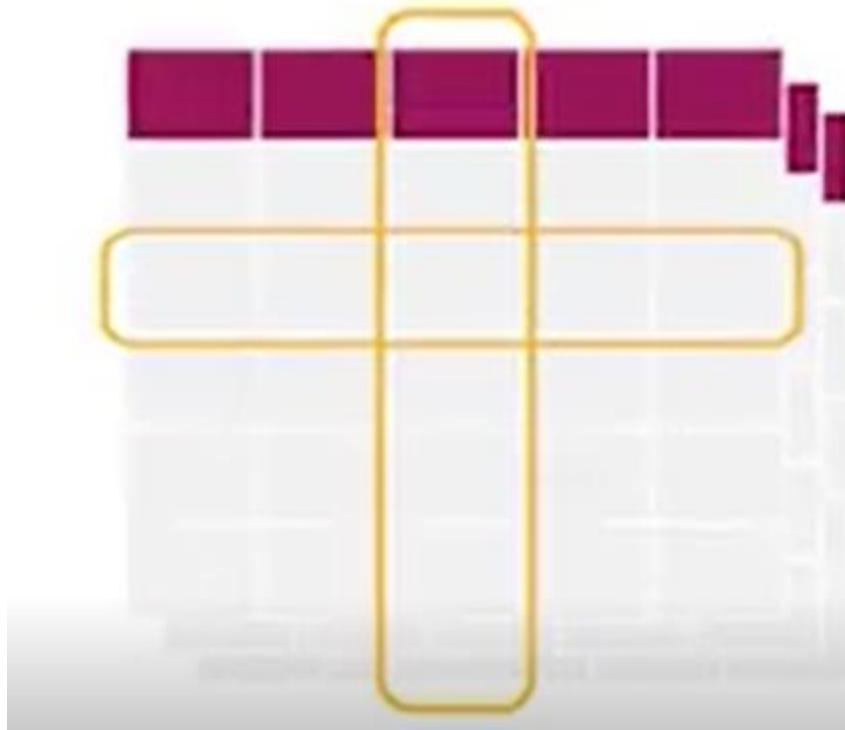
Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705



DataFrame



DataFrame = rdd + schema

When running SQL from within another programming language the results will be returned as a [Dataset/DataFrame](#). You can also interact with the SQL interface using the [command-line](#) or over [JDBC/ODBC](#).

Datasets and DataFrames

A Dataset is a distributed collection of data. Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine. A Dataset can be [constructed](#) from JVM objects and then manipulated using functional transformations (`map`, `flatMap`, `filter`, etc.). The Dataset API is available in [Scala](#) and [Java](#). Python does not have the support for the Dataset API. But due to Python's dynamic nature, many of the benefits of the Dataset API are already available (i.e. you can access the field of a row by name naturally `row.columnName`). The case for R is similar.

A DataFrame is a *Dataset* organized into named columns. It is conceptually equivalent to a table in a relational database or a data frame in R/Python, but with richer optimizations under the hood. DataFrames can be constructed from a wide array of [sources](#) such as: structured data files, tables in Hive, external databases, or existing RDDs. The DataFrame API is available in Scala, Java, [Python](#), and [R](#). In Scala and Java, a DataFrame is represented by a Dataset of Rows. In the [Scala API](#), DataFrame is simply a type alias of `Dataset[Row]`. While, in [Java API](#), users need to use `Dataset<Row>` to represent a DataFrame.

we will often refer to Scala/Java Datasets of Rows as DataFrames.

```
In: from pyspark.sql import SparkSession  
spark_session = SparkSession.builder\  
    .enableHiveSupport()\\  
    .appName("spark sql")\\  
    .master("local")\\  
    .getOrCreate()
```

```
In: geoip_rdd = spark_session\  
    .sparkContext\  
    .textFile("/user/pmezentsev/geoip")
```

```
In: geoip_rdd.take(3)
```

```
Out: [u'194.120.126.123, NL, Netherlands',  
      u'94.126.119.173, FR, France',  
      u'193.46.74.166, RU, Russian Federation']
```

Schema

ip **STRING**,
code **STRING**,
country **STRING**

```
In: from pyspark.sql.types import *
schema = StructType().add("ip", StringType())\
    .add("code", StringType())\
    .add("country", StringType())
```

```
In: geoip_df = spark_session\
    .createDataFrame(geoip_rdd1, schema)
```

```
In: geoip_df
```

Out: DataFrame[ip: string, code: string, country: string]

```
In: geoip_df.show(3)
```

	ip code	country
194.120.126.123 NL		Netherlands
94.126.119.173 FR		France
193.46.74.166 RU	Russian Federation	

only showing top 3 rows

In: geoip_df.rdd

Out: MapPartitionsRDD at javaToPython at
NativeMethodAccessorImpl.java:0

In: geoip_df.rdd.take(3)

Out: [Row(ip=u'194.120.126.123', code=u'NL',
country=u'Netherlands'),
Row(ip=u'94.126.119.173', code=u'FR',
country=u'France'),
Row(ip=u'193.46.74.166', code=u'RU',
country=u'Russian Federation')]

```
In: geoip_df.printSchema()
```

```
root
| -- ip: string (nullable = true)
| -- code: string (nullable = true)
| -- country: string (nullable = true)
```



DataSets

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

- Consider a DataFrame representing a data set of Listing of homes for sale; we have calculated average price for sale per zipcode.

```
case class Listing(street: String, zip: Int, price: Int)
val listingsDF = ... // DataFrame of Listings

import org.apache.spark.sql.functions._
val averagePricesDF = listingsDF.groupBy($"zip")
                           .avg("price")
```

```
val averagePrices = averagePricesDF.collect()  
// averagePrices: Array[org.apache.spark.sql.Row]
```

One method is ↓

```
val averagePricesAgain = averagePrices.map {  
    row => (row(0).asInstanceOf[String], row(1).asInstanceOf[Int])  
}
```

Nope.

```
// java.lang.ClassCastException
```

- Second method

```
averagePrices.head.schema.printTreeString()
// root
// |-- zip: integer (nullable = true)
// |-- avg(price): double (nullable = true)
```

Trying again...

```
val averagePricesAgain = averagePrices.map {
  row => (row(0).asInstanceOf[Int], row(1).asInstanceOf[Double]) // Ew...
}
// mostExpensiveAgain: Array[(Int, Double)]
```

yay! 🎉

Wouldn't it be nice if we could have both Spark SQL optimizations and typesafety?

DataFrames are Datasets!!!

DataFrames are actually Datasets.

```
type DataFrame = Dataset[Row]
```

- ▶ Datasets can be thought of as **typed** distributed collections of data.
- ▶ Dataset API unifies the DataFrame and RDD APIs. Mix and match!
- ▶ Datasets require structured/semi-structured data. Schemas and Encoders core part of Datasets.

Think of Datasets as a compromise between RDDs & DataFrames.

You get more type information on Datasets than on DataFrames, and you get more optimizations on Datasets than you get on RDDs.

Example:

Let's calculate the average home price per zipcode with Datasets.

Assuming `listingsDS` is of type `Dataset[Listing]`:

```
listingsDS.groupByKey(l => l.zip)      // looks like groupByKey on RDDs!
    .agg(avg($"price").as[Double])     // looks like our DataFrame operators!
```

We can freely mix APIs!

Observations

Datasets are something in the middle between DataFrames and RDDs

- ▶ You can still use relational DataFrame operations as we learned in previous sessions on Datasets.
- ▶ Datasets add more *typed* operations that can be used as well.
- ▶ Datasets let you use higher-order functions like `map`, `flatMap`, `filter` again!



DataFrames

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

Contents

- ▶ available DataFrame data types
- ▶ some basic operations on DataFrames
- ▶ aggregations on DataFrames

DataFrames in a NutShell

DataFrames are...

A relational API over Spark's RDDs

Because sometimes it is more convenient to use declarative relational APIs than functional APIs for analysis jobs.

Able to be automatically aggressively optimized

Spark SQL applies years of research on relational optimizations in the databases community to Spark.

Untyped!

The elements within DataFrames are Rows, which are not parameterized by a type. Therefore, the Scala compiler cannot type check Spark SQL schemas in DataFrames.

DataFrames DataTypes

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

Scala Type	SQL Type	Details
Byte	ByteType	1 byte signed integers (-128,127)
Short	ShortType	2 byte signed integers (-32768,32767)
Int	IntegerType	4 byte signed integers (-2147483648,2147483647)
Long	LongType	8 byte signed integers
java.math.BigDecimal	DecimalType	Arbitrary precision signed decimals
Float	FloatType	4 byte floating point number
Double	DoubleType	8 byte floating point number
Array[Byte]	BinaryType	Byte sequence values
Boolean	BooleanType	true/false
Boolean	BooleanType	true/false
java.sql.Timestamp	TimestampType	Date containing year, month, day, hour, minute, and second.
java.sql.Date	DateType	Date containing year, month, day.
String	StringType	Character string values (stored as UTF8)

DataFrames Data Types

To enable optimization opportunities, Spark SQL's DataFrames operate on a restricted set of data types.

Complex Spark SQL Data Types:

<i>Scala Type</i>	<i>SQL Type</i>
Array[T]	ArrayType(elementType, containsNull)
Map[K, V]	MapType(keyType, valueType, valueContainsNull)
case class	StructType(List[StructFields])

DataFrames Data Types

Arrays

Array of only one type of element (elementType), containsNull is set to true if the elements in ArrayType value can have null values.

Example:

```
// Scala type      // SQL type  
Array[String]     ArrayType(StringType, true)
```

DataFrames DataType

Complex Spark SQL Data Types:

Scala Type	SQL Type
Array[T]	ArrayType(elementType, containsNull)
Map[K, V]	MapType(keyType, valueType, valueContainsNull)
case class	StructType(List[StructFields])

Maps

Map of key/value pairs with two types of elements. valuecontainsNull is set to true if the elements in MapType value can have null values.

Example:

```
// Scala type          // SQL type
Map[Int, String]    MapType(IntegerType, StringType, true)
```

DataFrames DataType

Complex Spark SQL Data Types:

Scala Type	SQL Type
Array[T]	ArrayType(elementType, containsNull)
Map[K, V]	MapType(keyType, valueType, valueContainsNull)
case class	StructType(List[StructFields])

Structs

Struct type with list of possible fields of different types. containsNull is set to true if the elements in StructFields can have null values.

Example:

```
// Scala type                                // SQL type
case class Person(name: String, age: Int)    StructType(List(StructField("name", StringType, true),
                                                               StructField("age", IntegerType, true)))
```

Complex DataTypes can be combined

It's possible to arbitrarily nest complex data types! For example:

```
// Scala type
case class Account(
    balance: Double,
    employees:
    Array[Employee])

case class Employee(
    id: Int,
    name: String,
    jobTitle: String)

case class Project(
    title: String,
    team: Array[Employee],
    acct: Account)
```

Accessing Spark SQL Types

Important.

In order to access *any* of these data types, either basic or complex, you must first import Spark SQL types!

```
import org.apache.spark.sql.types._
```

DataFrame Operations

DataFrames Operations are more structured!!

When introduced, the `DataFrames API` introduced a number of relational operations.

The main difference between the `RDD API` and the `DataFrames API` was that `DataFrame APIs` accept `Spark SQL expressions`, instead of arbitrary user-defined function literals like we were used to on `RDDs`. This allows the optimizer to understand what the computation represents, and for example with `filter`, it can often be used to skip reading unnecessary records.

DataFrame Operations

DataFrames API: Similar-looking to SQL. Example methods include:

- ▶ select
- ▶ where
- ▶ limit
- ▶ orderBy
- ▶ groupBy
- ▶ join

Getting a look at your data

Before we get into transformations and actions on DataFrames, let's first look at the ways we can have a look at our data set.

show() pretty-prints DataFrame in tabular form. Shows first 20 elements.

Example:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.show()
// +---+---+---+---+
// | id|fname| lname|age| city|
// +---+---+---+---+
// | 12| Joel| Smith| 38|New York|
// |563|Sally| Owens| 48|New York|
// |645|Slate|Markham| 28| Sydney|
// |221|David| Walker| 21| Sydney|
// +---+---+---+---+
```

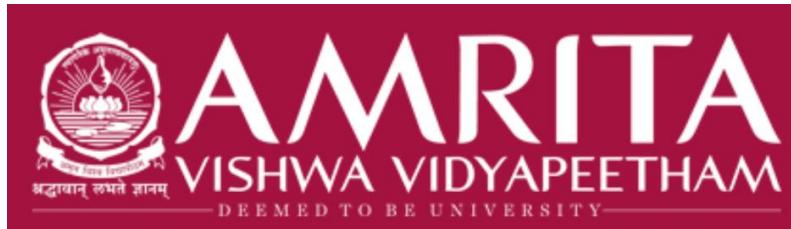
Getting a look at your data

printSchema() prints the schema of your DataFrame in a tree format.

Example:

```
case class Employee(id: Int, fname: String, lname: String, age: Int, city: String)
val employeeDF = sc.parallelize(...).toDF
employeeDF.printschema()
```

```
// root
//   |-- id: integer (nullable = true)
//   |-- fname: string (nullable = true)
//   |-- lname: string (nullable = true)
//   |-- age: integer (nullable = true)
//   |-- city: string (nullable = true)
```



Examples with SparkSQL

Dr. Divya Udayan J, Ph.D.(Konkuk University, S.Korea)

Assistant Professor(Selection Grade)

Department of CSE

Amrita School of Engineering, Amritapuri Campus,
Amrita Vishwa Vidyapeetham

Email: divyaudayanj@am.amrita.edu

Mobile: 9550797705

The entry point into all functionality in Spark is the [SparkSession](#) class. To create a basic `sparkSession`, just use `SparkSession.builder()`:

```
val spark = SparkSession.builder().appName("Spark SQL basic example").config("spark.some.config.option", "some-value").getOrCreate()
```

With a `SparkSession`, applications can create `DataFrames` from an [existing RDD](#), or from [Spark data sources](#).

As an example, the following creates a `DataFrame` based on the content of a JSON file:

```
scala> import spark.implicits._  
import spark.implicits._  
  
scala> val df = spark.read.json("C:/spark/examples/src/main/resources/people.json")  
df: org.apache.spark.sql.DataFrame = [age: bigint, name: string]  
  
scala> df.show()  
+---+-----+  
| age|    name|  
+---+-----+  
| null|Michael|  
|   30|    Andy|  
|   19| Justin|  
+---+-----+
```

Untyped Dataset Operations (aka DataFrame Operations)

Spark 2.0, DataFrames are just Dataset of Rows in Scala and Java API. These operations are also referred as "untyped transformations"

```
scala> df.printSchema()
root
|-- age: long (nullable = true)
|-- name: string (nullable = true)

scala> df.select("name").show()
+---+
|  name|
+---+
|Michael|
|  Andy|
| Justin|
+---+

scala> df.select($"name", $"age" + 1).show()
+---+---+
|  name| (age + 1)|
+---+---+
|Michael|    null|
|  Andy|      31|
| Justin|      20|
+---+---+
```

- // Select people older than 21
- df.filter(\$"age" > 21).show()
- // Count people by age

```
scala> df.groupBy("age").count().show()
+---+---+
| age|count|
+---+---+
| 19|    1|
| null|    1|
| 30|    1|
+---+---+
```

Creating Datasets

Datasets are similar to RDDs, however, they use a specialized [Encoder](#) to serialize the objects for processing or transmitting over the network. While both encoders and standard serialization are responsible for turning an object into bytes, encoders are code generated dynamically and use a format that allows Spark to perform many operations like filtering, sorting and hashing without deserializing the bytes back into an object.

```
scala> case class Person(name: String, age: Long)
defined class Person

scala> val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS: org.apache.spark.sql.Dataset[Person] = [name: string, age: bigint]

scala> caseClassDS.show()
+---+---+
|name|age|
+---+---+
|Andy| 32|
+---+---+
```

```
scala> val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS: org.apache.spark.sql.Dataset[Int] = [value: int]

scala> primitiveDS.map(_ + 1).collect()
res15: Array[Int] = Array(2, 3, 4)

scala> val path = "C:/spark/examples/src/main/resources/people.json"
path: String = C:/spark/examples/src/main/resources/people.json

scala> val peopleDS = spark.read.json(path).as[Person]
peopleDS: org.apache.spark.sql.Dataset[Person] = [age: bigint, name: string]

scala> peopleDS.show()
+---+-----+
| age|  name|
+---+-----+
| null|Michael|
|  30|   Andy|
|  19| Justin|
+---+-----+
```

Programmatically Specifying the Schema

When case classes cannot be defined ahead of time (for example, the structure of records is encoded in a string, or a text dataset will be parsed and fields will be projected differently for different users), a `DataFrame` can be created programmatically with three steps.

1. Create an RDD of `Rows` from the original RDD;
2. Create the schema represented by a `structType` matching the structure of `Rows` in the RDD created in Step 1.
3. Apply the schema to the RDD of `Rows` via `createDataFrame` method provided by `SparkSession`.

For example:

```
import org.apache.spark.sql.Row

import org.apache.spark.sql.types._

// Create an RDD
val peopleRDD = spark.sparkContext.textFile("examples/src/main/resources/people.txt")

// The schema is encoded in a string
val schemaString = "name age"
```

```
// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
  .map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)

// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
  .map(_.split(","))
  .map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)

// Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

// SQL can be run over a temporary view created using DataFrames
val results = spark.sql("SELECT name FROM people")
```

```
// The results of SQL queries are DataFrames and support all the normal RDD operations
// The columns of a row in the result can be accessed by field index or by field name
results.map(attributes => "Name: " + attributes(0)).show()
// +-----+
// |      value|
// +-----+
// |Name: Michael|
// |  Name: Andy|
// | Name: Justin|
// +-----+
```

Scalar Functions

Scalar functions are functions that return a single value per row, as opposed to aggregation functions, which return a value for a group of rows. Spark SQL supports a variety of [Built-in Scalar Functions](#). It also supports [User Defined Scalar Functions](#).

Aggregate Functions

Aggregate functions are functions that return a single value on a group of rows. The [Built-in Aggregation Functions](#) provide common aggregations such as `count()`, `countDistinct()`, `avg()`, `max()`, `min()`, etc. Users are not limited to the predefined aggregate functions and can create their own. For more details about user defined aggregate functions, please refer to the documentation of [User Defined Aggregate Functions](#).