

كلية التربية للعلوم الصرفة/ ابن الهيثم

المرحلة الثالثة/قسم علوم الحاسبات

Database Design

ا.م. آمنة عبد الرزاق هاني الصفار

Database

Database is a collection of interrelated data stored together without harmful or unnecessary redundancy to serve multiple applications.

1. Database management system (DBMS)

(DBMSs) are specially designed applications that interact with the user, other applications, and the database itself to capture and analyze data. A general-purpose (DBMS) is a software system designed to allow the definition, creation, querying, update, and administration of DB.

Well-known DBMSs include:

- MySQL
- Microsoft SQL Server
- Oracle, dBASE
- FoxPro
- IBM DB2

A database is not generally portable across different DBMS, but different DBMSs can by using standards such as SQL (Structure Query Language) and ODBC (Open Database Connectivity) or JDBC (The Java Database Connectivity) to allow a single application to work with more than one database.

2. Classification of DBMS

Database management systems can be classified based on several criteria, such as the data model, user numbers and database distribution.

2.1 Classification Based on Data Model

A database model is a type of data model that determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized, and manipulated.

Types of DB models:

- Network
- Hierarchical
- Relational
- Entity-Relationship
- Extended Relational
- Object-oriented
- Object-relational
- Semi-structured (XML/ Extensible Markup Language)
- NoSQL

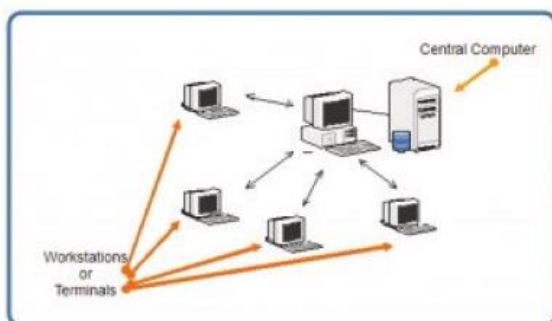
2.2 Classification Based on User Numbers

It can be a single-user database system, which supports one user at a time, or a multiuser database system, which supports multiple users concurrently.

2.3 Classification Based on Database Distribution

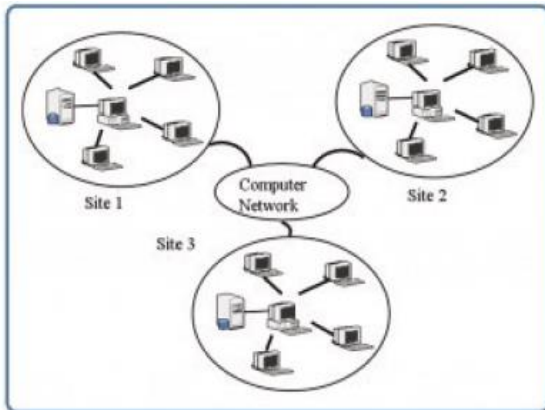
There are four main distribution systems for database systems and these, in turn, can be used to classify the DBMS.

2.3.1 Centralized systems: With a centralized database system, the DBMS and database are stored at a single site that is used by several other systems too. Fig(1)



Fig(1) centralized DB

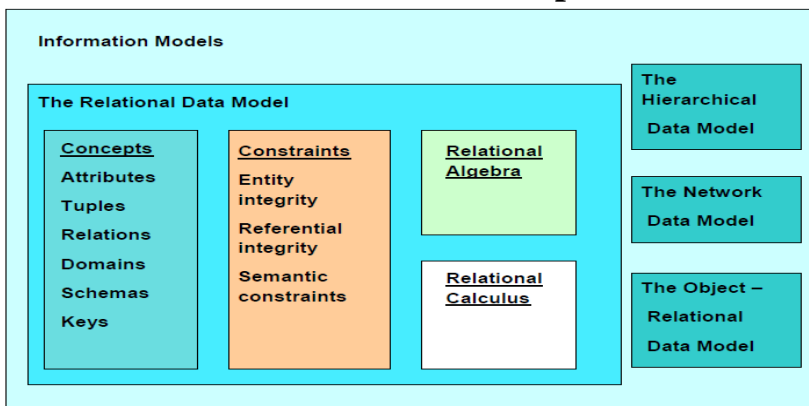
2.3.2 Distributed database system: In a distributed database system, the actual database and the DBMS software are distributed from various sites that are connected by a computer network, A distributed database system allows applications to access data from local and remote databases as shown in Fig(2)

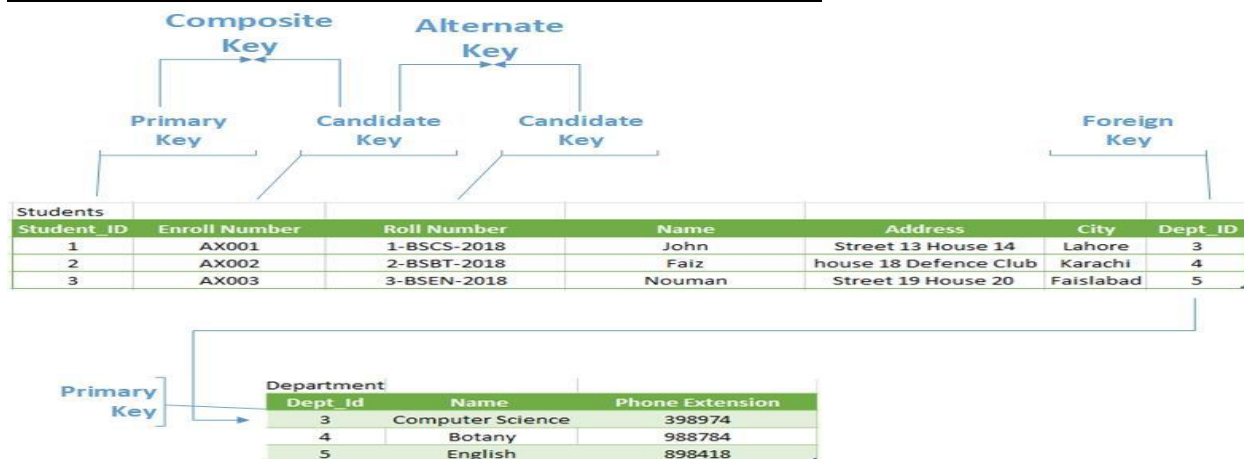
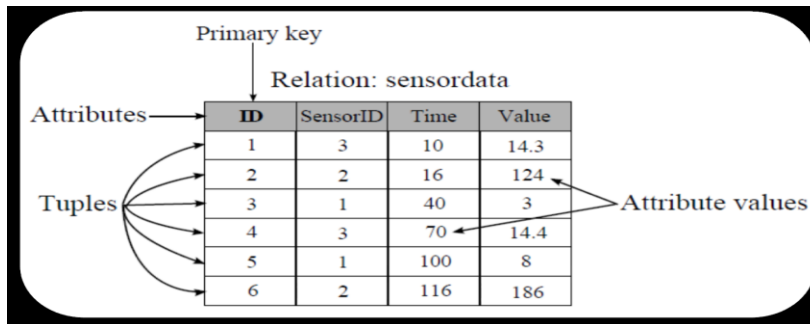


Fig(2) distributed DB

- **Homogeneous distributed database systems:** Homogeneous distributed database systems use the same DBMS software from multiple sites. Data exchange between these various sites can be handled easily.
- **Heterogeneous distributed database systems:** In a heterogeneous distributed database system, different sites might use different DBMS software, but there is additional common software to support data exchange between these sites.

3. RELATIONAL DB TABLE Concepts





A **database schema** is a formal description of all the database relations and all the relationships existing between them.

In a relational data model, **data integrity** can be achieved using **integrity rules** or **constraints**.

Those rules are general, specified at the database schema level, and they must be respected by each schema instance. If we want to have a correct relational database definition, we have to declare such constraints.

what the relational data model constraints are:

- Entity integrity constraint
- Referential integrity constraint
- Semantic integrity constraints

ENTITY INTEGRITY CONSTRAINT

The entity integrity constraint says that no attribute participating in the primary key of a relation is allowed to accept null values.

EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

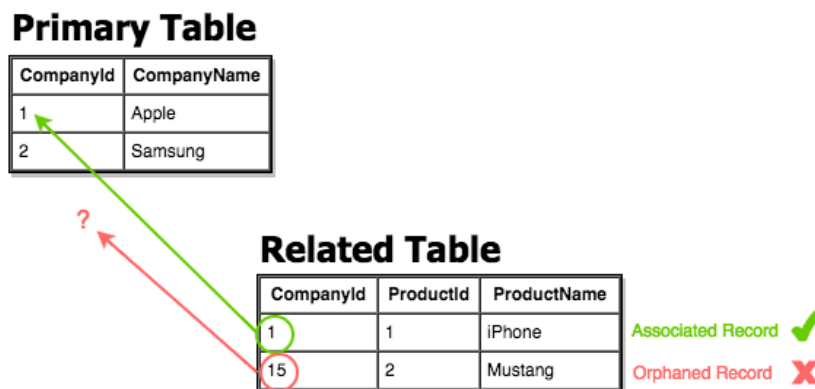
Not allowed as primary key can't contain a NULL value

NULL VALUE

- A Null represents property inapplicable information or unknown.
- Null is simply a marker indicating the absence of a value, an undefined value.

REFERENTIAL INTEGRITY CONSTRAINT

The referential integrity constraint says that if a relation R2 includes a foreign key FK matching the primary key PK of other relation R1, then every value of FK in R2 must either be equal to the value of PK in some tuple of R1 or be wholly null.



SEMANTIC INTEGRITY CONSTRAINTS

A semantic integrity constraint refers to the correctness of the meaning of the data. For example, the street number attribute value from the OWNERS relation must be positive, because the real-world street numbers are positive.

Semantic integrity constraints must be specified typically by a database administrator and must be maintained in the system catalog or dictionary.

Relational DBMS permits several types of semantic integrity constraints such as :

- domain constraint
- null constraint
- unique constraint
- check constraint

SEMANTIC INTEGRITY/ DOMAIN CONSTRAINT

A **domain constraint** implies that a particular attribute of a relation is defined on a particular domain.

For example, the Street attribute domain of OWNERS relation is CHAR(20), because streets have names in general and Number attribute domain is NUMERIC, because street numbers are numeric values.

There are some particular forms of domain constraints, namely **format constraints and range constraints**.

A **format constraint** might specify something like a data value pattern.

For example, the PhoneNumber attribute values must be of this type XXXX-XXXXXX where X represents an a digits first 4 X is the key and the last 6 is phone number .

A **range constraint** requires that values of the attribute lie within the range values. For example, the FabricationYear attribute values might range between 1950 and 2010.

SEMANTIC INTEGRITY /NULL CONSTRAINT

A null constraint specifies that attribute values cannot be null. On every tuple, from every relation instance, that attribute must have a value which exists in the underlying attribute domain.

For example, FirstName and LastName attributes values cannot be null, this means that a car owner must have a name.

SEMANTIC INTEGRITY /UNIQUE CONSTRAINT

A unique constraint specifies that attribute values must be different. It is not possible to have two tuples in a relation with the same values for that attribute.

For example, in the CARS relation the SerialNumber attribute values must be unique, because it is not possible to have two cars and only one engine.

A unique constraint is usually specified with an attribute name followed by the word Unique.

Note that NULL is a valid unique value.

4. Goals of database design

The features that a good database system should provide and explain to what degree they depend on good database design.

1. CRUD:

- CRUD stands for the four fundamental database operations that any database should provide: Create, Read, Update, and Delete.
- CRUD is more a feature of databases in general than it is a feature of good database design, but a good database design provides CRUD efficiently
- In general, however, if it doesn't have CRUD it's not a database.

2. Retrieval:

- Retrieval is another word for “read,” the R in CRUD. The database should allow you to find every piece of data. There’s no point putting something in the database if there’s no way to get it back later. (That would be a “data black hole,” not a database.)
- The database should allow you to structure the data so you can find particular pieces of data in one or more specific ways. For example, you should be able to find a customer’s billing record by searching for customer name or customer ID.

3. Consistency:

- Another aspect of the R in CRUD is consistency. The database should provide consistent results. If you perform the same search twice in a row, you should get the same results. Another user who performs the same search should also get the same results. Consistency means different parts of the database don’t hold contradictory views of the same information.
- A well-built database product can ensure that the exact same query returns the same result but design also plays an important role. If the database is poorly designed, you may be able to store conflicting data in different parts of the database.

4. Validity:

- Validity means *data is validated where possible against other pieces of data in the database*. In CRUD terms, data can be validated when a record is created, updated, or deleted.
- You can never protect a database from users who can’t spell or who just plain enter the wrong information, but a good database design can help prevent some kinds of errors that a physical database cannot prevent.
- For example, the database can easily verify that data has the correct type. A good database design also helps protect the database against incorrect changes

5. Error Correction:

- in a good design database It’s easy to update incorrect data. In bad design Simple and large-scale changes never happen. (Thousands of your customers’ bills are returned to you because their ZIP Code changed and the database didn’t get updated.)

6. Speed:

- in a good design you can quickly find customers by name, account number, or phone number.
- In bad design you can only find a customer’s record if he knows his 37-digit account number. Searching by name takes half an hour.

7. Security:

- good design :Users have access to the data they need and nothing else. bad design Hackers and disgruntled employees have access to everything.

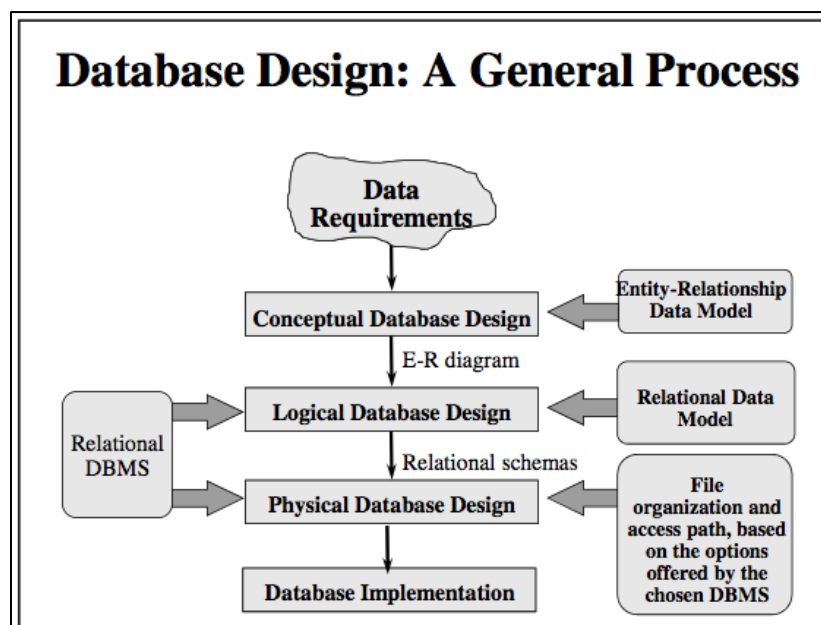
8. ACID:

- ACID is an acronym describing four features that an effective transaction system should provide. ACID stands for Atomicity, Consistency, Isolation, and Durability.

- *Transactions are a sequence of queries and updates that together carry out a task. Transactions can be committed, or rolled back; when a transaction is rolled back, the effects of all updates performed by the transaction are undone.*
- Atomicity means transactions are atomic. The operations in a transaction either all happen or none of them happen.
- Consistency means the transaction ensures that the database is in a consistent state before and after the transaction.
- In other words, if the operations within the transaction would violate the database's rules, the transaction is rolled back.
- Isolation means the transaction isolates the details of the transaction from everyone except the person making the transaction.
- Suppose you start a transaction, remove \$100 from Alice's account, and add \$100 to Bob's account. Another person cannot peek at the database while you're in the middle of this process and see a state where neither Alice nor Bob has the \$100. Anyone who looks in the database sees the \$100 somewhere, either in Alice's account before the transaction or in Bob's account afterwards.
- Durability means that once a transaction is committed, it will not disappear later. If the power fails, when the database restarts, the effects of this transaction will still be there.

5. Database design

Database Design and Application Development: How can a user describe a real-world enterprise (e.g., a university) in terms of the data stored in a DBMS? What factors must be considered in deciding how to organize the stored data?



5.1 Database Design Process

The database design process can be divided into six steps. The E-R model is most relevant to the first three steps.

1. Requirements Analysis:

The very first step in designing a DB application is to understand what data is to be stored in the DB, what applications must be built on top of it, and what operations are most frequent and subject to performance requirements.

2. Conceptual Database Design: The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the DB, along with the constraints known to hold over this data. This step is often carried out using the E-R model.

3. Logical Database Design: We must choose a DBMS to implement our DB design, and convert the conceptual database design into a DB schema in the data model of the chosen DBMS.

4. Schema Refinement: The fourth step with DB's design is to analyze the collection of relations in our relational DB schema to identify potential problems, and to refine it (Normalization).

5. Physical Database Design: It describes the details of how data is stored. This step may simply involve building indexes on some tables and clustering some tables

6. Application and Security Design: Any software project that involves a DBMS must consider aspects of the application that go beyond the database itself. We must identify the entities (e.g., users, user groups, departments) and processes involved in the




application. We must describe the role of each entity in every process that is reflected in some application task, as part of a complete workflow for that task.


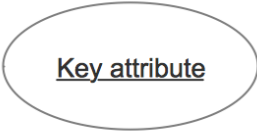



6. The Entity Relationship Mode

ER data model has existed for over 35 years. It is well suited to data modelling for use with DB because it is fairly abstract and is easy to discuss and explain. ER models are readily translated to relations. ER models, also called an ER schema, are represented by ER diagrams. ER modelling is based on two concepts:

1. Entities, defined as tables that hold specific information (data)
2. Relationships, defined as the associations or interactions between entities

Table(1) Entity Relationship Diagram Symbols

Symbol	Shape Name	Symbol Description
Entities		
	Entity	An entity is an object in the real world with an independent existence that can be differentiated from other objects. An entity is represented by a rectangle which contains the entity's name.
	Weak Entity	An entity that cannot be uniquely identified by its attributes alone. The existence of a weak entity is dependent upon another entity called the owner entity.
	Associative Entity	An entity used in a many-to-many relationship (represents an extra table). All relationships for the associative entity should be many
Attributes		

	Attribute	An attribute is a particular property that describes the entity. each attribute is represented by an oval containing attribute's name
	Key attribute	An attribute that uniquely identifies a particular entity. The name of a key attribute is underscored.
	Multivalued attribute	An attribute that can have many values (there are many distinct values entered for it in the same column of the table). Multivalued attribute is depicted by a dual oval.
	Derived attribute	An attribute whose value is calculated (derived) from other attributes. The derived attribute may not be physically stored in the database. This attribute is represented by dashed oval.
Relationships		
	Relationship	A relationship among two or more entities

6.1 Entity

An entity is an object in the real world with an independent existence that can be differentiated from other objects.

An entity might be

- An object with physical existence (e.g., a lecturer, a student, a car)
- An object with conceptual existence (e.g., a course, a job, a position)

Entities can be classified based on their strength into:-

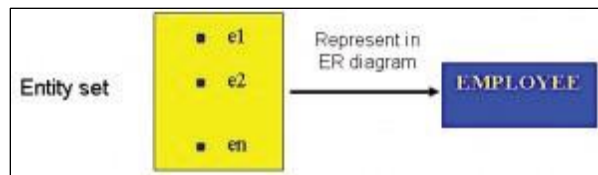
-Weak entity: an entity is considered **weak** if its tables are existence dependent.

- That is, it cannot exist without a relationship with another entity
- Its primary key is derived from the primary key of the parent entity

Strong entity: an entity is considered **strong** if it can exist a part from all of its related entities.

- A table without a foreign key or a table that contains a foreign key that can contain nulls is a strong entity

An **entity set** is a collection of entities of an entity type at a particular point of time. In an entity relationship diagram (ERD), an entity type is represented by a name in a box.



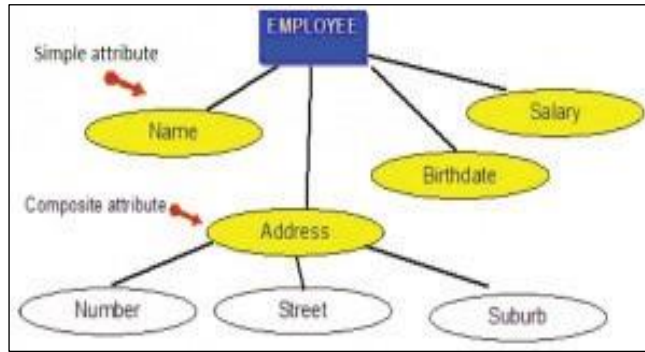
4.2 Entity and attributes

Each entity has attributes—the particular properties that describe it. For example, an EMPLOYEE entity may be described by the employee’s name, age, address, salary, and job. A particular entity may have values for each of its attributes.

Types Of Attributes

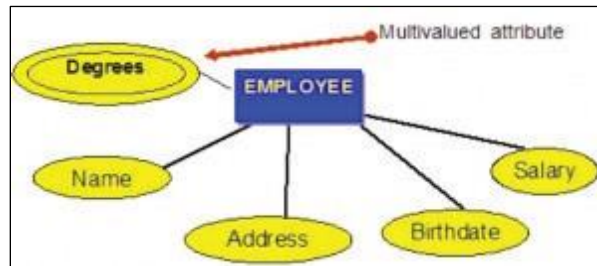
Several types of attributes occur in the ER model: **simple** versus **composite**, **single** valued versus **multivalued**, and **stored** versus **derived**.

1. **Simple attributes:** Are those drawn from the atomic value domains; they are also called single-valued attributes. In the COMPANY database, an example of this would be: Name = {John} ; Age = {23}
2. **Composite attributes:** Are those that consist of a hierarchy of attributes. Figure 3 Address may consist of Number, Street and Suburb. So this would be written as → Address = {59 + ‘Meek Street’ + ‘Kingsford’ }



Fig(3) Address is a Composite attribute

3. **Multivalued attributes:** Are attributes that have a set of values for each entity. See Figure 4, are the degrees of an employee: BSc, MIT, PhD.



Fig(4) Degree is a multi-valued attribute

4. **Derived attributes:** Are attributes that contain values calculated from other attributes. Figure 5 Age can be derived from the attribute Birthdate. In this situation, Birthdate is called a stored attribute, which is physically saved to the database

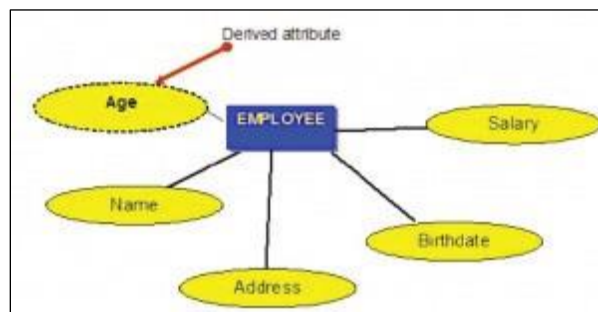


Fig (5) Age is a derived attribute, name, birthday, salary, address are stored attributes

4.3 NULL Values

A null is a special symbol, independent of data type, which means either unknown or inapplicable. It does not mean zero or blank. Features of null include:

- No data entry
- Not permitted in the primary key
- Should be avoided in other attributes

Null Can represent:-

- An unknown attribute value.
 - A known, but missing, attribute value.
 - A not “applicable” condition.
- Can create problems when functions such as COUNT, AVERAGE and SUM are used

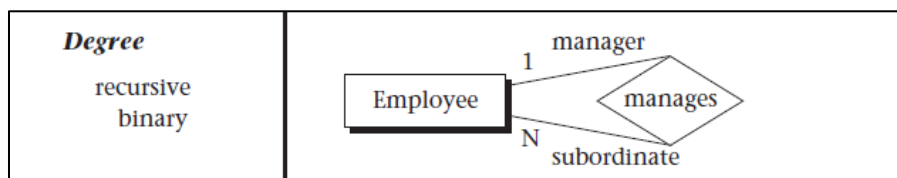
4.4 Relationship

Relationships are the glue that holds the tables together. They are used to connect related information between tables.

Degree of a relationship is the number of entities associated in the relationship. **Binary** and **ternary** relationships are special cases where the degrees are 2 and 3, respectively.

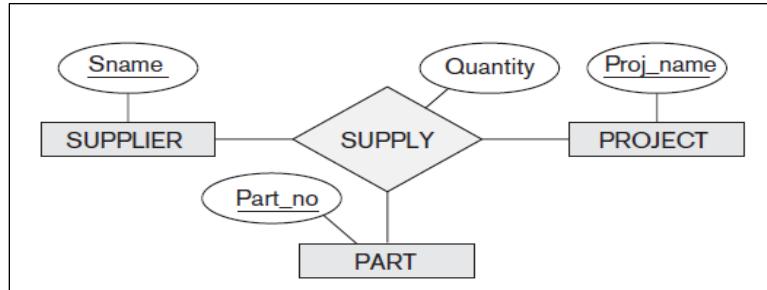
The **binary** relationship, an association between two entities, is by far the most common type in the natural world. In fact, many modeling systems use only this type.

Unary relationship (recursive): A unary relationship, also called recursive, is one in which a relationship exists between occurrences of the same entity set. In this relationship, the primary and foreign keys are the same, but they represent two entities with different roles fig(7).



Fig(7) recursive relationship

Ternary Relationships: A ternary relationship is a relationship type that involves many to many relationships between three tables. Refer to Figure (8) for an example of mapping a ternary relationship type. Note n-ary means multiple tables in relationship.

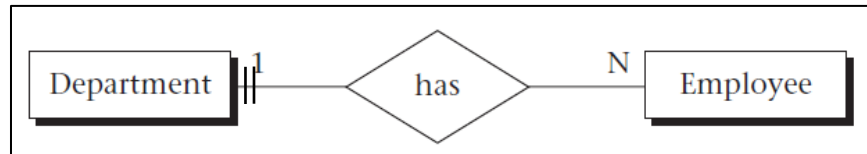


Fig(8) A ternary relationship

Connectivity of a Relationship

The connectivity of a relationship describes a constraint on the connection of the associated entity occurrences in the relationship. Values for connectivity are either “one” or “many.”

Example: for a relationship between the entities Department and Employee, a connectivity of one for Department and many for Employee means that there is at most one entity occurrence of Department associated with many occurrences of Employee.


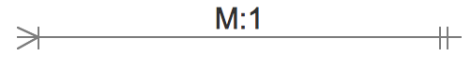
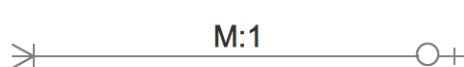




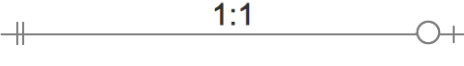
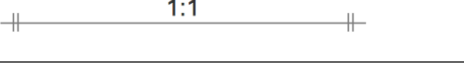


The actual count of elements associated with the connectivity is called the **cardinality of the relationship**.

Relationship Connectivity

Table (2) Relational Symbol and Meaning

Symbol	Meaning
Relationships (Cardinality and Modality)	
+○	Zero or One
⊎	One or More

	One and only One
	Zero or More
Many - to - One	
	a one through many notation on one side of a relationship and a one and only one on the other
	a zero through many notation on one side of a relationship and a one and only one on the other
	a one through many notation on one side of a relationship and a zero or one notation on the other
	a zero through many notation on one side of a relationship and a zero or one notation on the other
Many - to - Many	
	a zero through many on both sides of a relationship
	a zero through many on one side and a one through many on the other
	a one through many on both sides of a relationship
	a one and only one notation on one side of a relationship and a zero or one on the other
	a one and only one notation on both sides

4.5 How to Convert ER Diagram to Relational Database

We will be following the simple rules:

1. Entities and Simple Attributes:

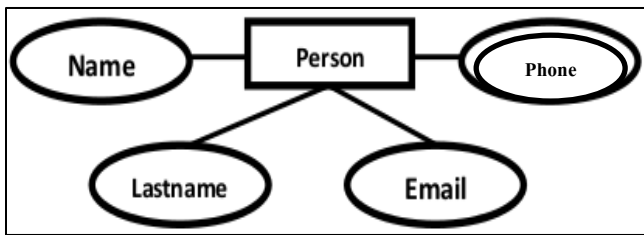
- An entity type within ER diagram is turned into a table. You may preferably keep the same name for the entity or give it a sensible name but avoid DBMS reserved words as well as avoid the use of special characters.

- Each attribute turns into a column (attribute) in the table. The key attribute of the entity is the primary key of the table which is usually underlined. It can be composite if required but can never be null.

Example:

Persons (personid, name, lastname, email)

Note the phone attribute not included.



2. Multi-Valued Attributes

A multi-valued attribute is usually represented with a double-line oval.

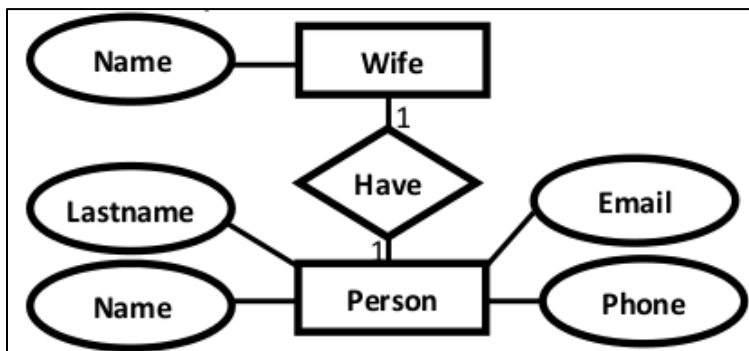
If you have a multi-valued attribute, take the attribute and turn it into a new entity or table of its own. Then make a 1: N relationship between the new entity and the existing one. In simple words:

- Create a table for the attribute.
- Add the primary (id) column of the parent entity as a foreign key within the new table

Persons (personid, name, lastname, email)

Phones (phoneid , **personid**, phone)

3. 1:1 Relationships



To keep it simple and even for better performances at data retrieval, I would personally recommend using attributes to represent such relationship. For instance, let us consider the case where the Person has or optionally has one wife. You can place the primary key of the wife within the table of the Persons which we call in this case Foreign key as shown below.

Persons (personid, name, lastname, email , **wifeid**)

Wife (wifeid , name)

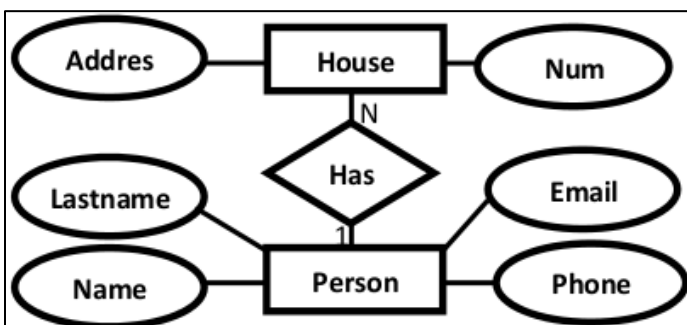
Or vice versa to put the personid as a foreign key within the Wife table as shown below:

Persons (personid , name, lastname, email)

Wife (wifeid , name , **personid**)

4. 1: N Relationships

This is the tricky part! For simplicity, use attributes in the same way as 1:1 relationship but we have only one choice as opposed to two choices. For instance, the Person can have a **House** from zero to many, but a **House** can have only one **Person**. To represent such relationship the **personid** as the Parent node must be placed within the Child table as a foreign key but not the other way around as shown next:



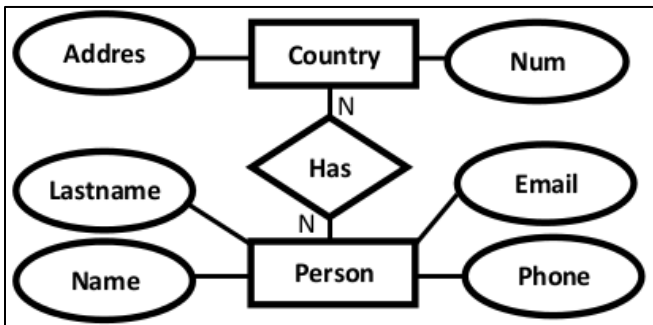
It should convert to:

Persons (personid , name, lastname, email)

House (houseid , num , address, **personid**)

5. N:N Relationships

We normally use tables to express such type of relationship. It is the same for N – ary relationship of ER diagrams. For instance, The Person can live or work in many countries. Also, a country can have many people. To express this relationship within a relational schema we use a separate table as shown below:



it should convert into :

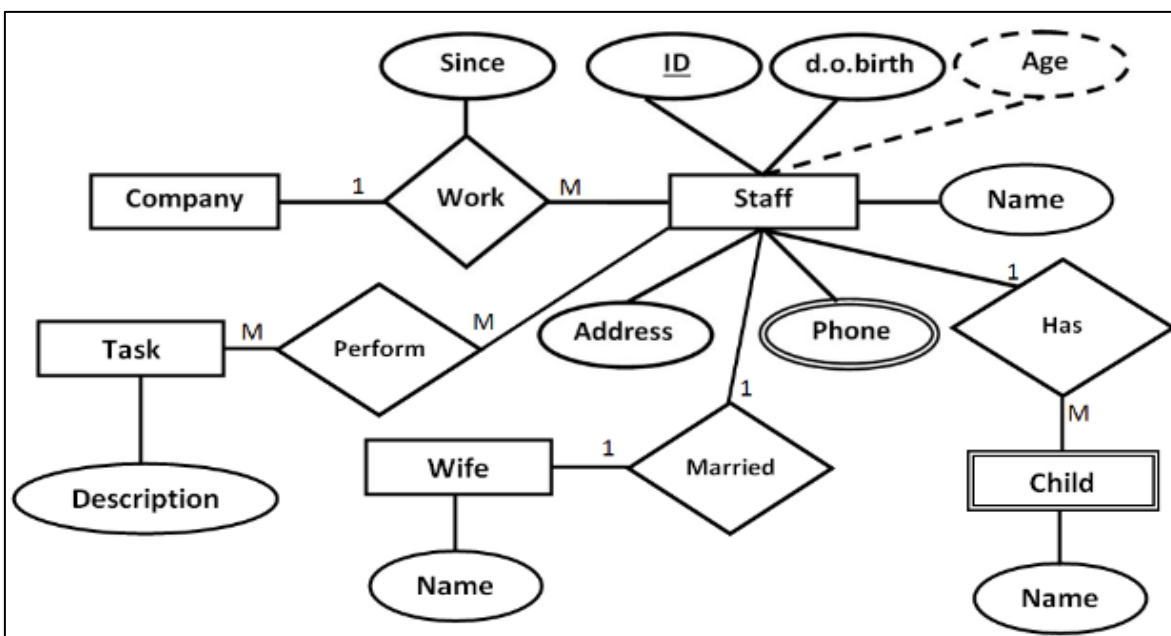
Persons(personid , name, lastname, email)

Countries (countryid , name, code)

HasRelat (personid + countryid)

Case Study

Convert the E-R diagram into relational database



7. Enhanced Entity Relationship Model (EER Model)

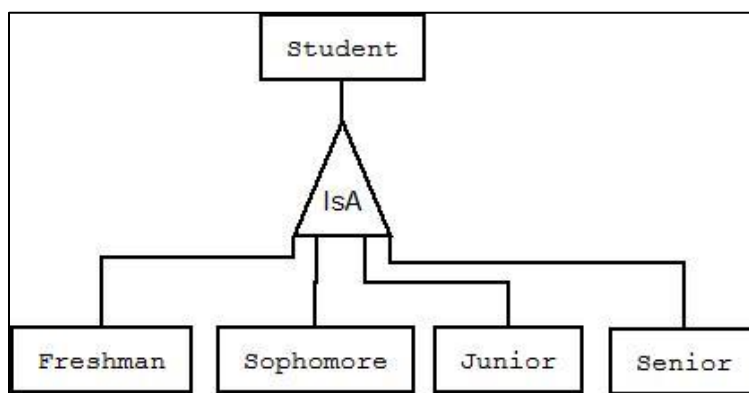
EER is a high-level data model that incorporates the extensions to the original ER model. It includes all modeling concepts of basic ER and additional:

- Sub Class and Super Class
- Specialization and Generalization
- Union or Category
- Aggregation

7.1 Sub class-super class

One entity type might be a subtype of another; very similar to subclasses in OO programming

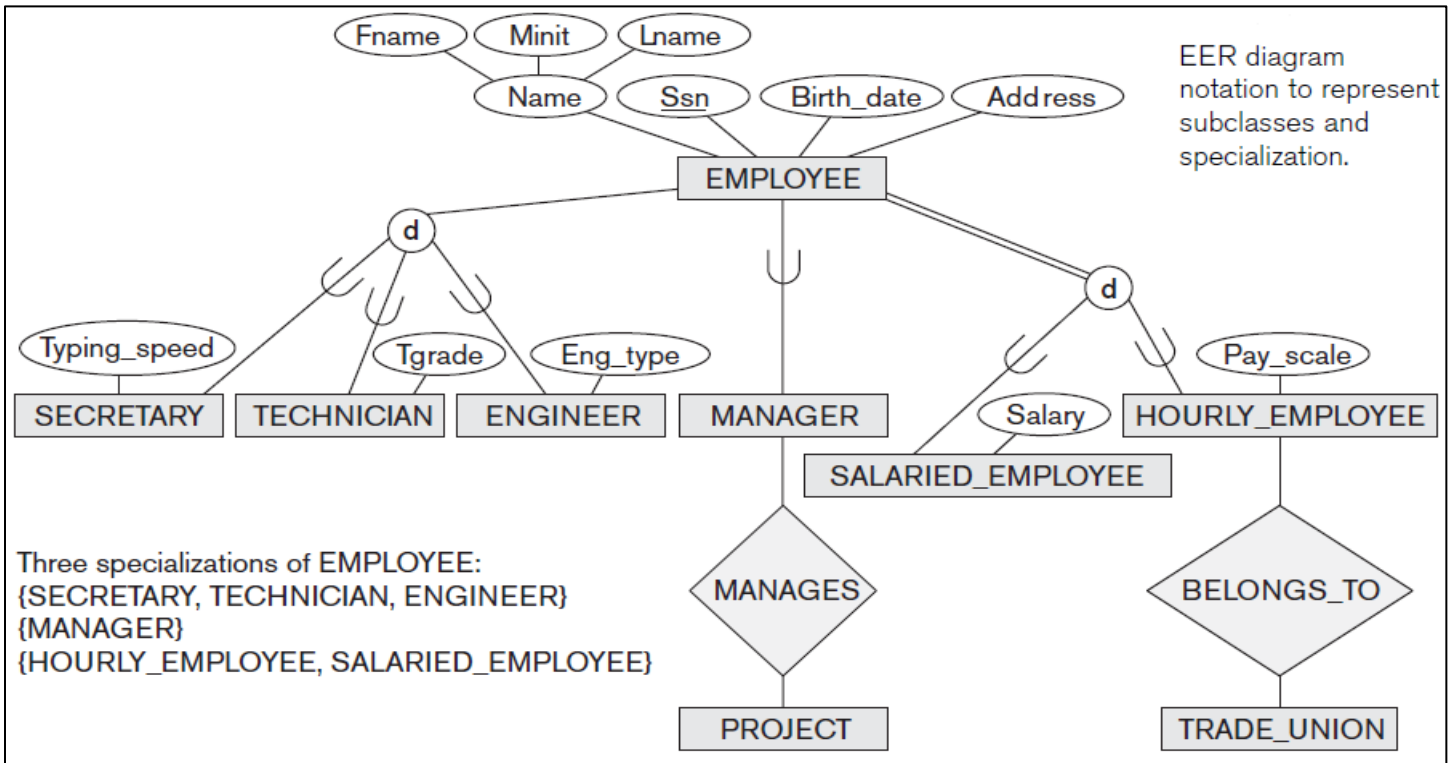
- Example:
 - EMPLOYEE may be further grouped into SECRETARY, ENGINEER, MANAGER, TECHNICIAN....
 - VEHICLE may be grouped into CAR, TRUCK, VAN, ...
 - Each of these groupings is a subset of EMPLOYEE entities and is called a **subclass** of EMPLOYEE
 - EMPLOYEE is the **superclass** for each of these subclasses
 - These are called **superclass/subclass relationships**.
 - These are also called IS-A relationships (SECRETARY IS-A EMPLOYEE, TECHNICIAN IS-A EMPLOYEE, ...). -



- An entity that is member of a subclass **inherits** all attributes of the entity as a member of the superclass
- It also inherits all relationships

7.2 Specialization

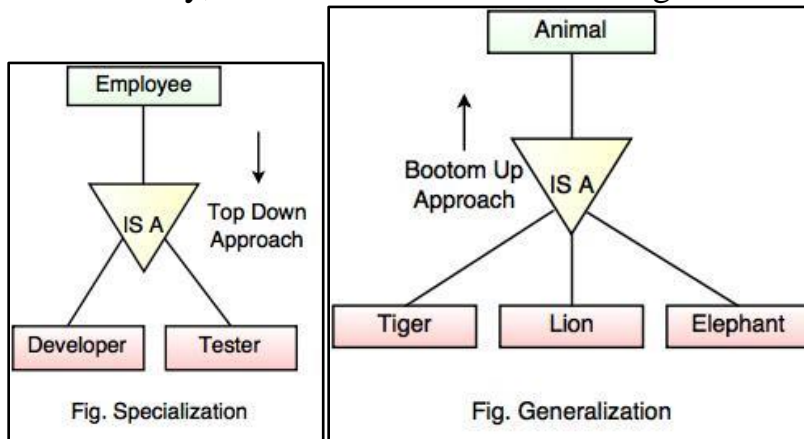
- Is the process of defining a set of subclasses of a superclass
- The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass
- Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon job type.
- May have several specializations of the same superclass
- Example: Another specialization of EMPLOYEE based in method of pay is {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.
- Superclass/subclass relationships and specialization can be diagrammatically represented in EER diagrams
- Attributes of a subclass are called specific attributes. For example, Typing Speed of SECRETARY
- The subclass can participate in specific relationship types. For example, BELONGS_TO of HOURLY_EMPLOYEE.



7.3 Generalization

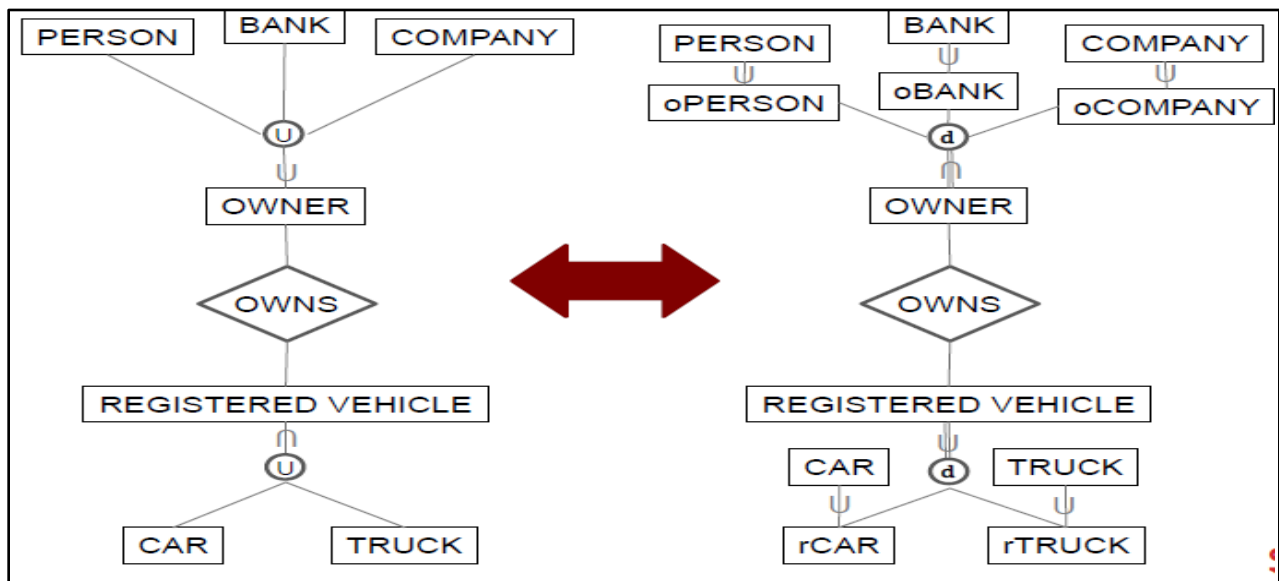
- The reverse of the specialization process
- Several classes with common features are generalized into a superclass; original classes become its subclasses
- Example: CAR, TRUCK generalized into VEHICLE; both CAR, TRUCK become subclasses of the superclass VEHICLE.
- We can view {CAR, TRUCK} as a specialization of VEHICLE

- Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK



7.4 Category or Union

- Category represents a single super class or sub class relationship with more than one super class.
- **For example** Car booking, Car owner can be a person, a bank (holds a possession on a Car) or a company. Category (sub class) → Owner is a subset of the union of the three super classes → Company, Bank, and Person. A Category member must exist in at least one of its super classes.



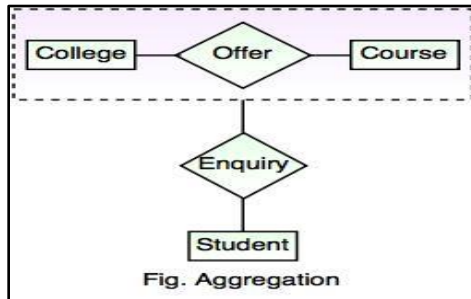
REWRITING UNION AS SPECIALIZATION

7.5 Aggregation

- Aggregation is a process that represent a relationship between a whole object and its component parts.

- It abstracts a relationship between objects and viewing the relationship as an object.
- It is a process when two entity is treated as a single entity.

In the following example, the relation between College and Course is acting as an Entity in Relation with Student.



7.6 EER to Relational Mapping

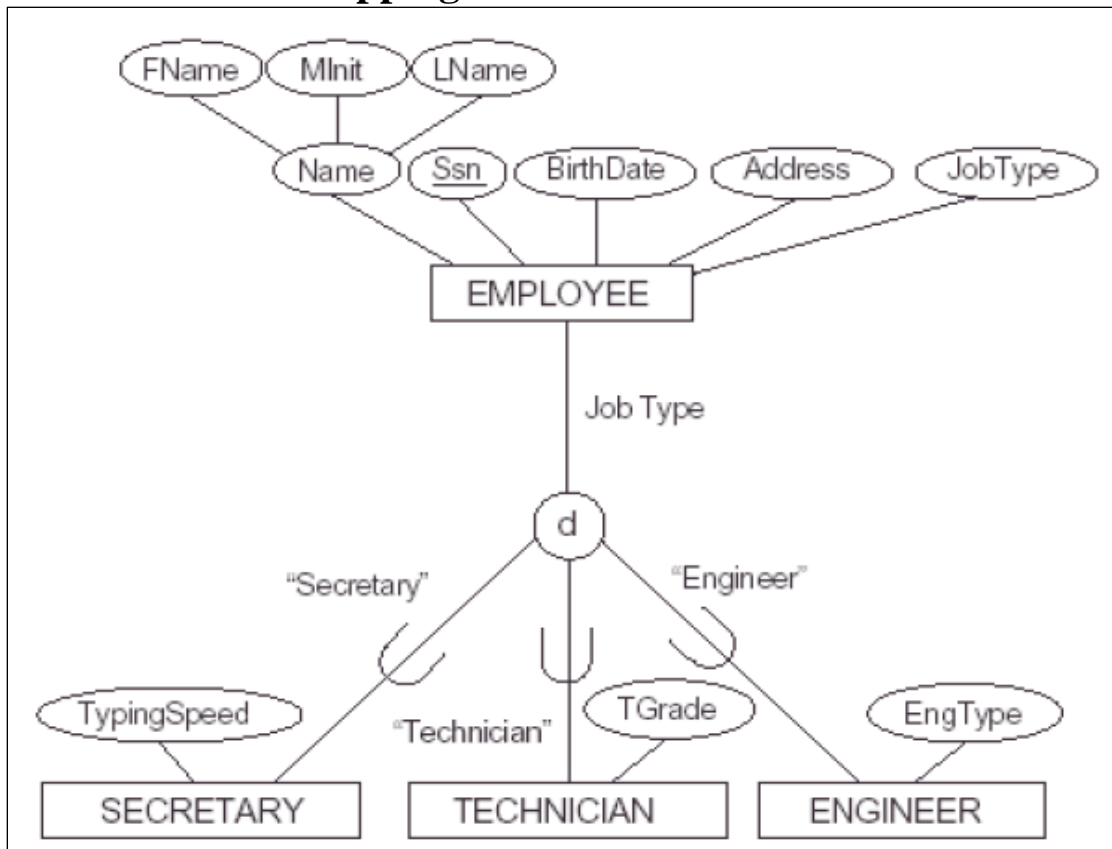


Fig (9) ER schema diagram specialization on job title

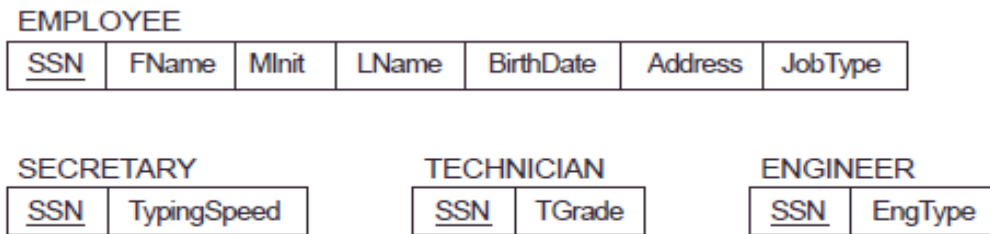
To convert each **super-class/subclass** relationship into a relational schema you must use one of the four options available.

Let C be the super-class, K its primary key and A1, A2, ..., An its remaining attributes and let S1, S2, ..., Sm be the sub-classes.

Option A (multiple relation option):

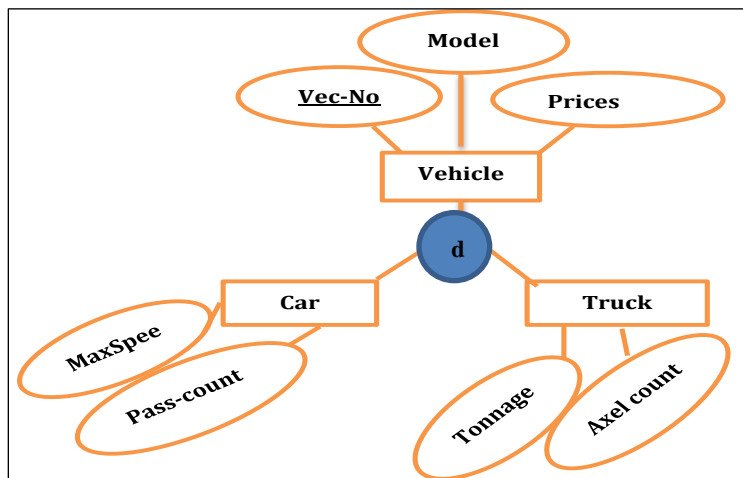
- Create a relation L for C with attributes $(L) = \{K, A1, A2, \dots, An\}$ and $PK(L) = K$.
- Create a relation Li for each subclass Si , $1 < i < m$, with the attributes $(Li) = \{K\} \cup \{\text{attributes of } Si\}$ and $PK(Li) = K$.
- This option works for any constraints: disjoint or overlapping; total or partial.

Mapping the EER diagram on fig(9) using option A



Option B (multiple relation option):

- Create a relation Li for each subclass Si , $1 < i < m$, with $(Li) = \{\text{attributes of } Si\} \cup \{K, A1, A2, \dots, An\}$ $PK(Li) = K$
- This option works well only for disjoint and total constraints.
- If not disjoint, redundant values for inherited attributes.



Fig(10)

Car

MaxSpeed	Pass-count	Model	Price	<u>Vec-no</u>
----------	------------	-------	-------	---------------

Truck

Tonnage	Axel count	Model	Price	<u>Vec-no</u>
---------	------------	-------	-------	---------------

Option c (Single Relation Option)

- Create a single relation L with attributes

$$(L) = \{K, A1, \dots, An\} \cup \{\text{attributes of } S1\} \cup \dots \cup \{\text{attributes of } Sm\} \cup \{T\} \text{ and } PK(L)=K$$

- This option is for specialization whose subclasses are DISJOINT, and T is a **type** attribute that indicates the subclass to which each tuple belongs, if any. This option may generate a large number of null values.
- Not recommended if many specific attributes are defined in subclasses (will result in many null values!)

Employee

<u>Ssn</u>	Fname	Minit	Lname	birthdate	Address	JobType	typpigSpeed	TGrad	EngType	Temp
------------	-------	-------	-------	-----------	---------	---------	-------------	-------	---------	------

Option d (Single Relation Option)

- Create a single relation schema L with attributes

$$(L) = \{K, A1, \dots, An\} \cup \{\text{attributes of } S1\} \cup \dots \cup \{\text{attributes of } Sm\} \cup \{T1, \dots, Tn\} \text{ and } PK(L)=K$$

- This option is for specialization whose subclasses are overlapping, and each $T_i, 1 < i < m$, is a Boolean attribute indicating whether a tuple belongs to subclass S_i .
- This option could be used for disjoint subclasses too.

Vehicle

Vec_no	Model	Price	Maxspee	Pass-count	carF	tonnage	Axel count	truckF
--------	-------	-------	---------	------------	------	---------	------------	--------

Relational algebra

- The traditional set operations: union, intersection, difference and Cartesian product
- The special relational operations: select, project, join and divide.

Relational algebra a formal query language for asking questions. Is a set of operators to manipulate relations. Each operator of the relational algebra takes either one or two relations as its input and produces a new relation as its output

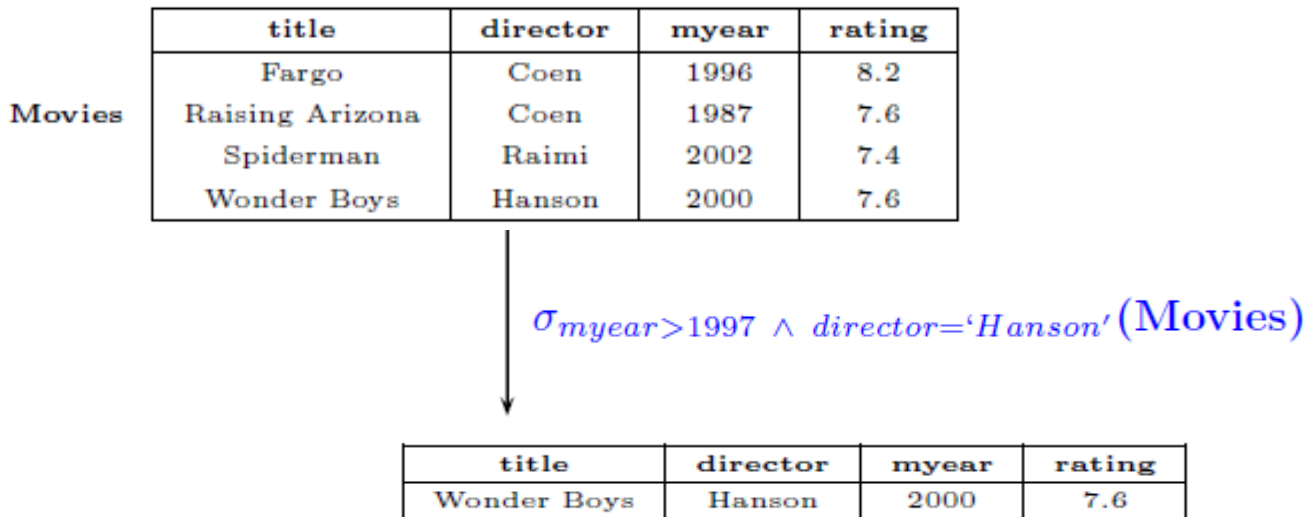
Codd defined 8 such operators, two groups of 4 each:

- The traditional set operations: union, intersection, difference and Cartesian product
- The special relational operations: select, project, join and divide.

Selection: σ

selects rows from relation R that satisfy selection condition c

- Example : find the movies made by Hanson after 1997



Selection Condition

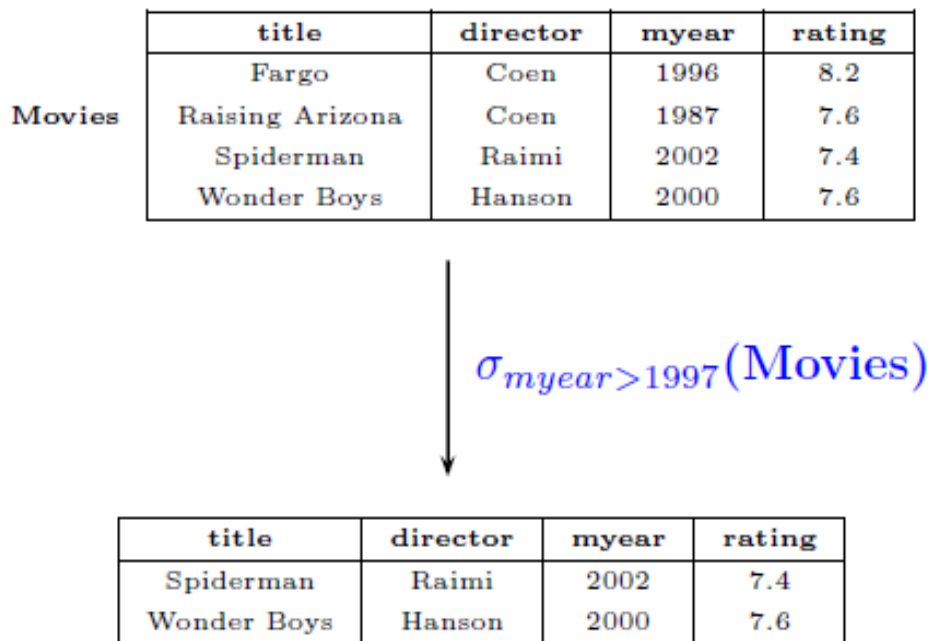
- Selection condition is a Boolean combination of terms
- A term is one of the following forms:

1. attribut op constant op $\in \{=, \neq, <, \leq, >, \geq\}$

2. attribute1 op attribute2
3. term1 \wedge term2
4. term1 \vee term2
5. \neg term1
6. (term1)

• Operator precedence: (), op, \neg , \wedge , \vee

Example: Find movies made after 1997



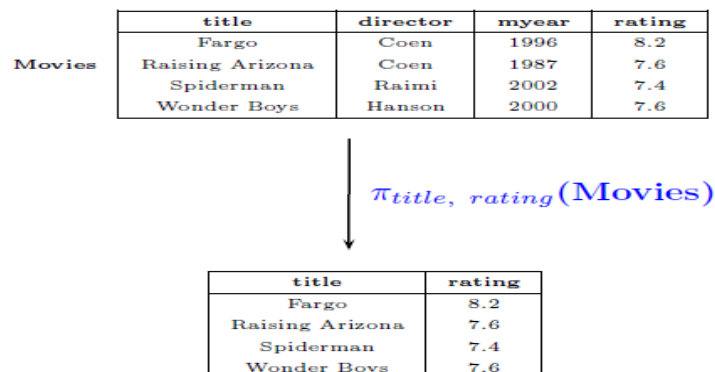
Projection π

$\pi_{fields}(\text{Input})$

Allows us to extract **columns** from a **relation**

Eliminate duplicate tuples, if any.

- Example: Find all movies and their ratings



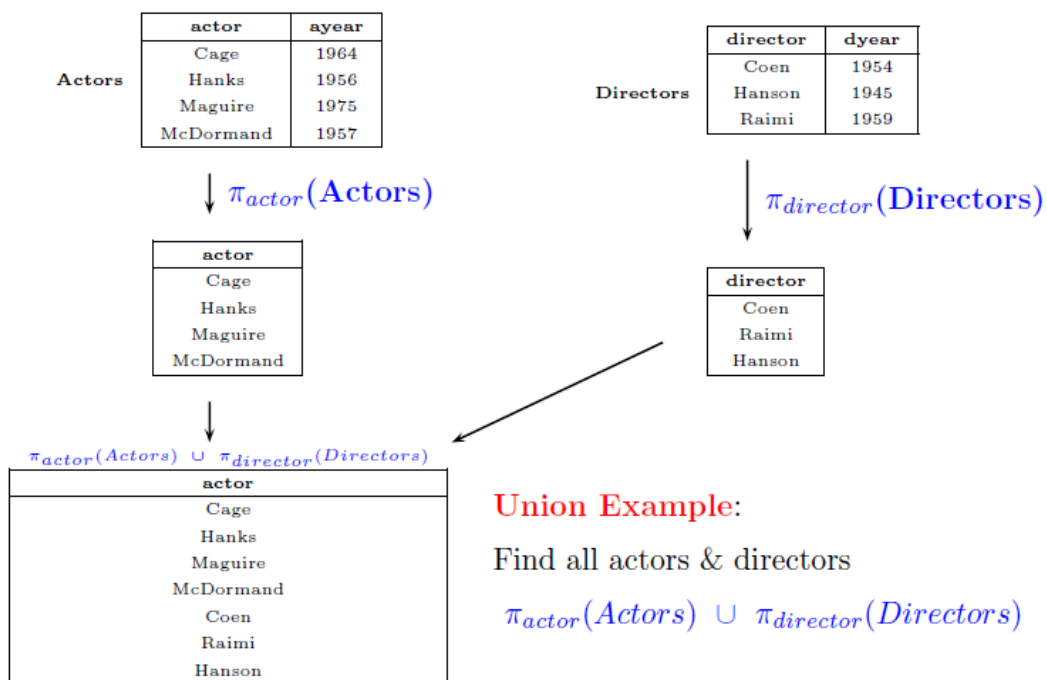
Set Operations

- **Union:** $R \cup S$ returns a relation containing all tuples that occur in R or S (or both). Remove the duplicated tuples
- **Intersection:** $R \cap S$ returns a relation containing all tuples that occur in both R and S
- **Set-difference:** $R - S$ returns a relation containing all tuples in R but not in S

Two relations are union compatible if

- They have the same number of fields
- corresponding fields, have the same domains

- union (\cup), intersection (\cap), and set-difference ($-$) operators require input relations to be union compatible



R1			R2		
Name	Age	Sex	Name	Age	Sex
A	20	M	D	20	F
C	21	M	A	20	M
B	21	F	E	21	F

$R3 = R1 \cap R2$

Name	Age	Sex
A	20	M

R1			R2		
Name	Age	Sex	Name	Age	Sex
A	20	M	D	20	F
C	21	M	A	20	M
B	21	F	E	21	F

$R3 = R1 - R2$

Name	Age	Sex
C	21	M
B	21	F

$R3 = R2 - R1$

Name	Age	Sex
D	20	F
E	21	F

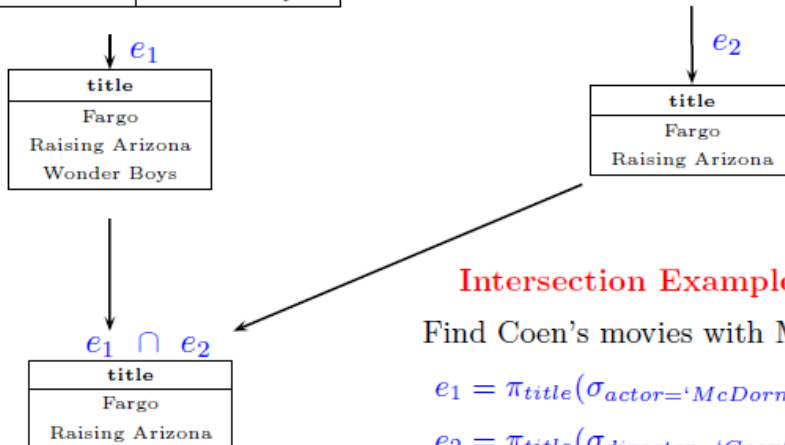
R1			R2		
Name	Age	Sex	Name	Age	Sex
A	20	M	D	20	F
C	21	M	A	20	M
B	21	F	E	21	F

$R3 = R1 \cup R2$

Name	Age	Sex
A	20	M
C	21	M
B	21	F
D	20	F
E	21	F

actor	title
Cage	Raising Arizona
Maguire	Spiderman
Maguire	Wonder Boys
McDormand	Fargo
McDormand	Raising Arizona
McDormand	Wonder Boys

title	director	myear	rating
Fargo	Coen	1996	8.2
Raising Arizona	Coen	1987	7.6
Spiderman	Raimi	2002	7.4
Wonder Boys	Hanson	2000	7.6



Intersection Example:

Find Coen's movies with McDormand

$$e_1 = \pi_{title}(\sigma_{actor='McDormand'}(Acts))$$

$$e_2 = \pi_{title}(\sigma_{director='Coen'}(Movies))$$

$$result = e_1 \cap e_2$$

Cross-Product

Consider $R(A,B,C)$ and $S(X, Y)$

- Cross-product: $R \times S$ returns a relation with attribute list (A,B,C,X, Y) defined as follows:

$$R \times S = \{(a, b, c, x, y) \mid (a, b, c) \in R, (x, y) \in S\}$$

- Cross-product operation is also known as Cartesian product
- Fields of the same name are may renamed

R1			R2		
Name	Age	Sex	Name	Age	Sex
A	20	M	D	20	F
C	21	M	E	21	F

R3= R1 X R2					
Name	Age	Sex	Name	Age	Sex
A	20	M	D	20	F
C	21	M	D	20	F
A	20	M	E	21	F
C	21	M	E	21	F

Join

Join can be defined as cross-product followed by selection and projection. We have

Several variants of join.

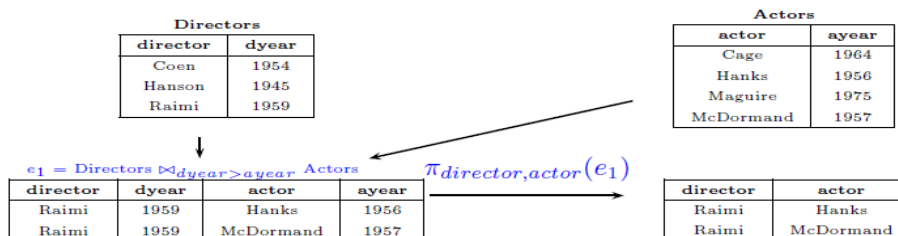
1. Condition joins

Condition Join: $R \bowtie_c S$

- **Condition join** = Cross-product followed by selection

$$R \bowtie_c S = \sigma_c(R \times S)$$

- **Example:** Find (director,actor) pairs where the director is younger than the actor
- **Answer:** $\pi_{director,actor}(\text{Directors} \bowtie_{dyear > ayear} \text{Actors})$



2. Equijoin

Condition consists only of equalities connected by \wedge

Redundancy in retaining both attributes in result. So, an additional projection is applied to remove the second attribute.

- **Example:** Find actors who have acted in some Coen's movie

- $\pi_{actor}(\sigma_{director='Coen'}(Acts \bowtie_{Acts.title = Movies.title} Movies))$

$e_1 = Acts \bowtie_{Acts.title = Movies.title} Movies$

actor	title	director	myear	rating
Cage	Raising Arizona	Coen	1987	7.6
Maguire	Spiderman	Raimi	2002	7.4
Maguire	Wonder Boys	Hanson	2000	7.6
McDormand	Fargo	Coen	1996	8.2
McDormand	Raising Arizona	Coen	1987	7.6
McDormand	Wonder Boys	Hanson	2000	7.6

Movies

title	director	myear	rating
Fargo	Coen	1996	8.2
Raising Arizona	Coen	1987	7.6
Spiderman	Raimi	2002	7.4
Wonder Boys	Hanson	2000	7.6

Acts

actor	title
Cage	Raising Arizona
Maguire	Spiderman
Maguire	Wonder Boys
McDormand	Fargo
McDormand	Raising Arizona
McDormand	Wonder Boys

$\pi_{actor}(\sigma_{director='Coen'}(e_1))$

actor
Cage
McDormand

3. Natural Join

It is an equijoin in which equalities are specified on all fields having the same name in R and S.

- We can then omit the join condition.
- Result is guaranteed not to have two fields with the same name.
- If no fields in common, then natural join is simply cross product

Example: Condition, Equi-, Natural Joins

A	B	X
0	6	x_1
1	9	x_2
2	7	x_3

A	B	Y
0	8	y_1
1	5	y_2
2	7	y_3

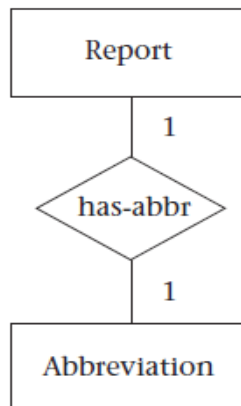
- $R \bowtie_{A=A' \wedge B < B'} \rho_{S'(A',B',Y)}(S)$

A	B	X	A'	B'	Y
0	6	x_1	0	8	y_1
- $R \bowtie_{A=A'} \rho_{S'(A',B',Y)}(S)$

A	B	X	B'	Y
0	6	x_1	8	y_1
1	9	x_2	5	y_2
2	7	x_3	7	y_3
- $R \bowtie S$

A	B	X	Y
2	7	x_3	y_3

Transformation Rules and SQL Constructs



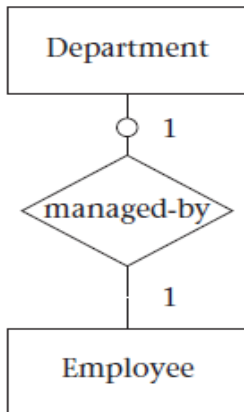
Every report has one abbreviation, and every abbreviation represents exactly one report.

```

create table report
(report_no integer,
 report_name varchar(256),
 primary key(report_no));

create table abbreviation
(abbr_no char(6),
 report_no integer not null unique,
 primary key (abbr_no),
 foreign key (report_no) references report
 on delete cascade on update cascade);
    
```

(a) One-to-one, both entities mandatory

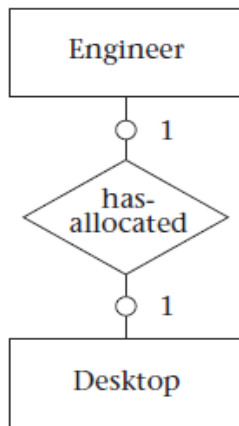


Every department must have a manager, but an employee can be a manager of at most one department.

```
create table department
(dept_no integer,
dept_name char(20),
mgr_id char(10) not null unique,
primary key (dept_no)
foreign key (mgr_id) references employee
on delete set default on update cascade);

create table employee
(emp_id char(10),
emp_name char(20)
primary key (emp_id));
```

(b) One-to-one, one entity optional, one mandatory

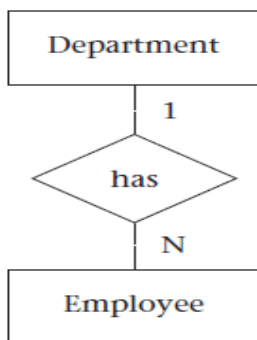


Some desktop computers are allocated to engineers, but not necessarily to all engineers.

```
create table engineer
(emp_id char(10),
desktop_no integer,
primary key (emp_id));

create table desktop
(desktop_no integer,
emp_id char(10)
primary key (desktop_no)
foreign key (emp_id) references engineer
on delete set null on update cascade);
```

(c) One-to-one, both entities optional

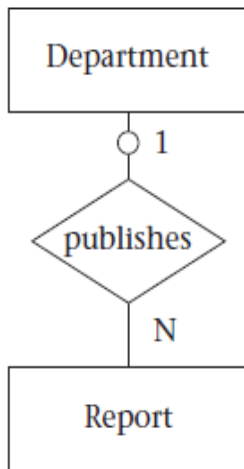


Every employee works in exactly one department, and each department has at least one employee.

```
create table department
(dept_no integer,
dept_name char(20),
primary key (dept_no));

create table employee
(emp_id char(10),
emp_name char(20),
dept_no integer not null,
primary key (emp_id),
foreign key (dept_no) references department
on delete set default on update cascade)
```

(d) One-to-many, both entities mandatory

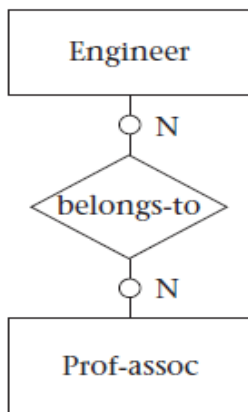


Each department publishes one or more reports. A given report may not necessarily be published by a department.

```
create table department
  (dept_no integer,
  dept_name char(20),
  primary key (dept_no));

create table report
  (report_no integer,
  dept_no integer,
  primary key (report_no),
  foreign key (dept_no) references department
  on delete set null on update cascade);
```

(e) One-to-many, one entity optional, one mandatory



Every professional association could have none, one, or many engineer members. Each engineer could be a member of none, one, or many professional associations.

```
create table engineer
  (emp_id char(10),
  primary key (emp_id));

create table prof_assoc
  (assoc_name varchar(256),
  primary key (assoc_name));

create table belongs_to
  (emp_id char(10),
  assoc_name varchar(256),
  primary key (emp_id, assoc_name),
  foreign key (emp_id) references engineer
  on delete cascade on update cascade,
  foreign key (assoc_name) references prof_assoc
  on delete cascade on update cascade);
```

(f) Many-to-many, both entities optional

Advanced SQL

The SQL language has several aspects to it.

1. The Data Manipulation Language (DML): This subset of SQL allows users to pose queries and to insert, delete, and modify rows.
2. The Data Definition Language (DDL): This subset of SQL supports the creation, deletion, and modification of definitions for tables and views.
3. Triggers and Advanced Integrity Constraints: The new SQL:1999 standard includes support for *triggers*.

4. Embedded SQL: Embedded SQL features allow SQL code to be called from a host language such as C or COBOL.
5. Dynamic SQL: Features allow a query to be constructed (and executed) at run-time.
6. Client-Server Execution and Remote Database Access: These commands control how a *client* application program can connect to an SQL database *server*, or access data from a database over a network.
7. Transaction Management: Various commands allow a user to explicitly control aspects of how a transaction is to be executed.
8. Security: SQL provides mechanisms to control users' access to data objects such as tables and views.

TRIGGERS AND ACTIVE DATABASES

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database.

A trigger description contains three parts:

- Event: A change to the database that activates the trigger.
- Condition: A query or test that is run when the trigger is activated.
- Action: A procedure that is executed when the trigger is activated and its condition is true.

A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the *event* specification.

An insert, delete, or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.

A *condition* in a trigger can be a true/false statement (e.g., all employee salaries are less than \$100,000) or a query. A query is interpreted as *true* if the answer set is

nonempty and false if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed.

A trigger action can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute the queries, and make changes to the database.

In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit) or call host-language procedures.

Triggers can be used in various applications, such as maintaining database consistency, monitoring database updates, and updating derived data automatically.

Trigger Syntax and Examples in MySQL

To create a trigger or drop a trigger, use the CREATE TRIGGER or DROP TRIGGER statement.

Syntax

```
CREATE
TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW
trigger_body
```

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema. Triggers in different schemas can have the same name.

trigger_time is the trigger action time. It can be BEFORE or AFTER to indicate that the trigger activates before or after each row to be modified.

Note : Basic column value checks occur prior to trigger activation, so you cannot use BEFORE triggers to convert values inappropriate for the column type to valid values.

trigger_event indicates the kind of operation that activates the trigger. These **trigger_event** values are permitted:

- **INSERT**: The trigger activates whenever a new row is inserted into the table; for example, through **INSERT** statements.
- **UPDATE**: The trigger activates whenever a row is modified; for example, through **UPDATE** statements.
- **DELETE**: The trigger activates whenever a row is deleted from the table; for example, through **DELETE** statements.

DROP TABLE and **TRUNCATE TABLE** statements on the table do *not* activate this trigger, because they do not use **DELETE**.

The **trigger_event** does not represent a literal type of SQL statement that activates the trigger so much as it represents a type of table operation. For example, an **INSERT** trigger activates not only for **INSERT** statements but also **LOAD DATA** statements because both statements insert rows into a table.

Notes :

1. Cascaded foreign key actions do not activate triggers.
2. There cannot be multiple triggers for a given table that have the same trigger event and action time. For example, you cannot have two BEFORE UPDATE triggers for a table. But you can have a BEFORE UPDATE and a BEFORE INSERT trigger, or a BEFORE UPDATE and an AFTER UPDATE trigger.

trigger_body is the statement to execute when the trigger activates. To execute multiple statements, use the **BEGIN ... END** compound statement construct.

Within the trigger body, you can refer to columns in the subject table (the table associated with the trigger) by using the aliases OLD and NEW. OLD.col_name refers to a column of an existing row before it is updated or deleted. NEW.col_name refers to the column of a new row to be inserted or an existing row after it is updated.

Here is a simple example that associates a trigger with a table, to activate for INSERT operations. The trigger acts as an accumulator, summing the values inserted into one of the columns of the table.

```
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
      FOR EACH ROW SET @sum = @sum + NEW.amount;
```

The CREATE TRIGGER statement creates a trigger named ins_sum that is associated with the account table. It also includes clauses that specify the trigger action time, the triggering event, and what to do when the trigger activates:

- The keyword BEFORE indicates the trigger action time. In this case, the trigger activates before each row inserted into the table. The other permitted keyword here is AFTER.
- The keyword INSERT indicates the trigger event; that is, the type of operation that activates the trigger. In the example, INSERT operations cause trigger activation. You can also create triggers for DELETE and UPDATE operations.
- The statement following FOR EACH ROW defines the trigger body; that is, the statement to execute each time the trigger activates, which occurs once for each row affected by the triggering event. In the example, the trigger body is a simple SET that accumulates into a user variable the values inserted into the amount column. The statement refers to the column as NEW.amount which means “the value of the amount column to be inserted into the new row.”

To use the trigger, set the accumulator variable to zero, execute an INSERT statement, and then see what value the variable has afterward:

```
mysql> SET @sum = 0;
mysql> INSERT INTO account VALUES(137,14.98),(141,1937.50),(97,-100.00);
mysql> SELECT @sum AS 'Total amount inserted';
+-----+
| Total amount inserted |
+-----+
|          1852.48 |
+-----+
```

In this case, the value of @sum after the INSERT statement has executed is 14.98 + 1937.50 - 100, or 1852.48.

To destroy the trigger, use a DROP TRIGGER statement. You must specify the schema name if the trigger is not in the default schema:

```
mysql> DROP TRIGGER test.ins_sum;
```

If you drop a table, any triggers for the table are also dropped.

Trigger names exist in the schema namespace, meaning that all triggers must have unique names within a schema. Triggers in different schemas can have the same name. In an INSERT trigger, only NEW.col_name can be used; there is no old row. In a DELETE trigger, only OLD.col_name can be used; there is no new row. In an UPDATE trigger, you can use OLD.col_name to refer to the columns of a row before it is updated and NEW.col_name to refer to the columns of the row after it is updated.

By using the BEGIN ... END construct, you can define a trigger that executes multiple statements. Within the BEGIN block, you also can use other syntax that is permitted within stored routines such as conditionals and loops. However, just as for stored routines, if you use the mysql program to define a trigger that executes multiple

statements, it is necessary to redefine the mysql statement delimiter so that you can use the ; statement delimiter within the trigger definition.

The DELIMITER statement changes the standard delimiter which is semicolon (;) to another. The delimiter is changed from the semicolon (;) to double-slashes //.

Why do we have to change the delimiter? To pass the trigger, stored procedure to the server as a whole rather than letting mysql tool to interpret each statement at a time.

The following example illustrates these points. It defines an UPDATE trigger that checks the new value to be used for updating each row, and modifies the value to be within the range from 0 to 100. This must be a BEFORE trigger because the value must be checked before it is used to update the row:

```
mysql> delimiter //
mysql> CREATE TRIGGER upd_check BEFORE UPDATE ON account
-> FOR EACH ROW
-> BEGIN
->   IF NEW.amount < 0 THEN
->     SET NEW.amount = 0;
->   ELSEIF NEW.amount > 100 THEN
->     SET NEW.amount = 100;
->   END IF;
-> END;//
mysql> delimiter ;
```

STORED PROCEDURES

A stored procedure is a set of Structured Query Language (SQL) statements with an assigned name, which are stored in a relational database management system as a group, so it can be reused and shared by multiple programs.

Why Stored Procedures?

- Stored procedures are fast. The main speed gain comes from reduction of network traffic. Running application logic directly at the database has the advantage that the amount of data that is transferred between the database server and the client issuing the SQL statement can be minimized.
- Stored procedures are beneficial for software engineering reasons. Once a stored procedure is registered with the database server, different users can re-use the stored procedure, eliminating duplication of efforts in writing SQL queries or application logic, and making code maintenance easy.
- Although they are called stored procedures, they do not have to be procedures in a programming language sense; they can be functions.

View

A view is tables whose rows are not explicitly stored in the database but are computed as needed from a view definition. A database view is a searchable object in a database that is defined by a query. Though a view doesn't store data, some refer to a views as "virtual tables," you can query a view like you can a table. A view can combine data from two or more table, and also just contain a subset of information. We create view using SQL commands.

What is the difference between table and view?

A view is a virtual table. A view consists of rows and columns just like a table. The difference between a view and a table is that views are definitions built on top of other tables (or views), and do not hold data themselves. If data is changing in the underlying table, the same change is reflected in the view. A view can be built on top of a single table or multiple tables. It can also be built on top of another view.

Views offer the following advantages:

1. Ease of use: A view hides the complexity of the database tables from end users. Essentially we can think of views as a layer of abstraction on top of the database tables.
2. Space savings: Views takes very little space to store, since they do not store actual data.
3. Additional data security: Views can include only certain columns in the table so that only the non-sensitive columns are included and exposed to the end user. In addition, some databases allow views to have different security settings, thus hiding sensitive data from prying eyes.

Create view in mysql

The basic syntax used to create a view in MySQL.

```
CREATE VIEW `view_name` AS SELECT statement;
```

WHERE

- "CREATE VIEW `view_name`" tells MySQL server to create a view object in the database named `view_name`
- "AS SELECT statement" is the SQL statements to be packed in the views. It can be a SELECT statement can contain data from one table or multiple tables.

Dropping views

The DROP command can be used to delete a view from the database that is no longer required. The basic syntax to drop a view is as follows.

```
DROP VIEW `view_name`;
```

Indexes

An index is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. An index allows us to efficiently retrieve all records that satisfy search conditions on the search key fields of the index. We can also create additional indexes on a given collection of data records, each with a different search key, to speed up search operations that are not efficiently supported by the file organization used to store the data records.

Basic Concepts

- Indexing mechanisms are used to optimize certain accesses to data (records) managed in files. For example, the author catalog in a library is a type of index.
- Search Key (definition): attribute or combination of attributes used to look up records in a file.
- An Index File consists of records (called **index entries**) of the form

search key value	pointer to block in data file
------------------	-------------------------------

- Index files are typically much smaller than the original file because only the values

for search key and pointer are stored.

- There are two basic types of indexes:
 - Ordered indices: Search keys are stored in a sorted order.
 - Hash indices: Search keys are distributed uniformly across "buckets" using a hash function.
- A file may have several indices on different search keys.

Create index in mysql

There are 2 ways to create an index. You can either create an index when you first create a table using the CREATE TABLE statement or you can use the CREATE INDEX statement after the table has been created.

Syntax

The syntax to create an index using the CREATE TABLE statement in MySQL is:

```
CREATE TABLE table_name (
  column1 datatype [ NULL | NOT NULL ],
  ...
  column_n datatype [ NULL | NOT NULL ],
  INDEX index_name [ USING BTREE | HASH ]
  (index_col1 [(length)] [ASC | DESC],
  ...
  index_col_n [(length)] [ASC | DESC]);
```

OR

The syntax to create an index using the CREATE INDEX statement in MySQL is:

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX index_name
 [ USING BTREE | HASH ]
 ON table_name
 (index_col1 [(length)] [ASC | DESC],
  index_col2 [(length)] [ASC | DESC],
  ...
  index_col_n [(length)] [ASC | DESC]);
```

UNIQUE

Optional. The *UNIQUE* modifier indicates that the combination of values in the indexed columns must be unique.

FULLTEXT

Optional. The *FULLTEXT* modifier indexes the entire column and does not allow prefixing.

SPATIAL

Optional. The *SPATIAL* modifier indexes the entire column and does not allow indexed columns to contain NULL values.

index_name

The name to assign to the index.

table_name

The name of the table in which to create the index.

index_col1, index_col2, ... index_col_n

col1...col_n The columns to use in the index.

Length

Optional. If specified, only a prefix of the column is indexed not the entire column.

For non-binary string columns, this value is the given number of characters of the column to index. For binary string columns, this value is the given number of bytes of the column to index.

ASC

Optional. The index is sorted in ascending order for that column.

DESC

Optional. The index is sorted in descending order for that column.

Example

Let's look at an example of how to create an index in MySQL using the CREATE TABLE statement. This statement would both create the table as well as the index at the same time.

For example:

```
CREATE TABLE contacts
( contact_id INT(11) NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(30) NOT NULL,
  first_name VARCHAR(25), birthday DATE,
  PRIMARY KEY (contact_id),
  INDEX contacts_idx (last_name, first_name));
```

In this example, we've created the *contacts* table as well as an index called *contacts_idx* which consists of the *last_name* and *first_name* columns.

Next, we will show you how to create the table first and then create the index using the CREATE INDEX statement.

For example:

```
CREATE TABLE contacts
( contact_id INT(11) NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(30) NOT NULL,
  first_name VARCHAR(25), birthday DATE,
  PRIMARY KEY (contact_id));
CREATE INDEX contacts_idx ON contacts (last_name, first_name);
```

In this example, the CREATE TABLE statement would create the *contacts* table. The CREATE INDEX statement would create an index called *contacts_idx* that consists of the *last_name* and the *first_name* fields.

Unique Index

To create a unique index on a table, you need to specify the UNIQUE keyword when creating the index. Again, this can be done with either a CREATE TABLE statement or a CREATE INDEX statement.

For example:

```
CREATE TABLE contacts
( contact_id INT(11) NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(30) NOT NULL,
  first_name VARCHAR(25),
  birthday DATE,
  CONSTRAINT contacts_pk PRIMARY KEY (contact_id),
  UNIQUE INDEX contacts_idx (last_name, first_name));
```

OR

```
CREATE TABLE contacts
( contact_id INT(11) NOT NULL AUTO_INCREMENT,
  last_name VARCHAR(30) NOT NULL,
  first_name VARCHAR(25),
  birthday DATE,
  CONSTRAINT contacts_pk PRIMARY KEY (contact_id));
CREATE UNIQUE INDEX contacts_idx
ON contacts (last_name, first_name);
```

Both of these examples would create a unique index on the *last_name* and *first_name* fields so that the combination of these fields must always contain a unique value with no duplicates. This is a great way to enforce integrity within your database if you require unique values in columns that are not part of your primary key.

Drop an Index

You can drop an index in MySQL using the DROP INDEX statement.

Syntax

The syntax to drop an index using the DROP INDEX statement in MySQL is:

```
DROP INDEX index_name ON table_name;
```

index_name

The name of the index to drop.

table_name

The name of the table where the index was created.

Example

Let's look at an example of how to drop an index in MySQL.

For example:


```
DROP INDEX contacts_idx ON contacts;
```

In this example, we've dropped an index called *contacts_idx* from the *contacts* table.

SQL Joins

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "person" table:

```
mysql> select * from person;
+----+-----+-----+
| id | name  | fk   |
+----+-----+-----+
|  1 | steve |  1   |
|  2 | aron  |  3   |
|  3 | mary  | NULL |
+----+-----+-----+
```

Then, look at a selection from the "color" table:

```
mysql> select *from color;
+----+-----+
| id | color |
+----+-----+
|  1 | read  |
|  2 | green |
|  3 | blue  |
+----+-----+
```

Notice that the "fk" column in the "person" table refers to the "id" in the "color" table.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

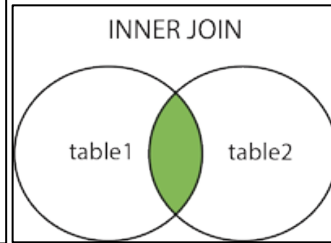
The MySQL INNER JOIN clause matches rows in one table with rows in other tables and allows you to query rows that contain columns from both tables.

Here are the different types of the JOINS in SQL:

- **(INNER) JOIN:** Returns records that have matching values in both tables

```
mysql> select * from person inner join color on person.fk=color.id;
```

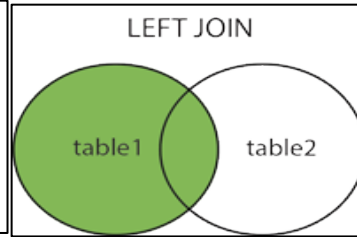
id	name	fk	id	color
1	steve	1	1	read
2	aron	3	3	blue



- **LEFT (OUTER) JOIN:** Return all records from the left table, and the matched records from the right table

```
mysql> select * from person left join color on person.fk=color.id;
```

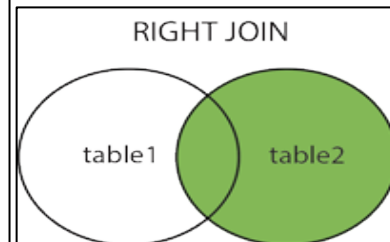
id	name	fk	id	color
1	steve	1	1	read
2	aron	3	3	blue
3	mary	NULL	NULL	NULL



- **RIGHT (OUTER) JOIN:** Return all records from the right table, and the matched records from the left table

```
mysql> select * from person right join color on person.fk=color.id;
```

id	name	fk	id	color
1	steve	1	1	read
2	aron	3	3	blue
NULL	NULL	NULL	2	green

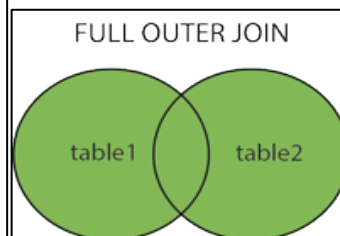


- **Cross join :** Return all records when there is a match in either left or right table

```
mysql> select * from person cross join color
```

id	name	fk	id	color
1	steve	1	1	read
2	aron	3	1	read
3	mary	NULL	1	read
1	steve	1	2	green
2	aron	3	2	green
3	mary	NULL	2	green
1	steve	1	3	blue
2	aron	3	3	blue
3	mary	NULL	3	blue

9 rows in set (0.00 sec)



Accessing SQL from a Programming Language

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a database programmer must have access to a general-purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.
2. Non declarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL.

Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, there must be a means to combine SQL with a general-purpose programming language.

There are two approaches to accessing SQL from a general-purpose programming language:

- **Dynamic SQL:** A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time.
- **Embedded SQL:** Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor.

The preprocessor submits the SQL statements to the database system for precompilation and optimization; then it replaces the SQL statements in the application program with appropriate code and function calls before invoking the programming-language compiler.

Two standards for connecting to an SQL database JDBC is an application program interface for the Java language. The other, ODBC is an application program interface originally developed for the C language C++, C#, and Visual Basic.

New DB Data Model Types

Several application areas for database systems are limited by the restrictions of the relational data model. As a result, researchers have developed several data models based on an object-oriented approach and NoSQL, to deal with these application domains.

Object-Oriented / Object-Relational Databases

In the 1980s, several database systems based on the object-oriented data model were developed. The major database vendors presently support the object-relational data model, a data model that combines features of the object-oriented data model and relational data model. It extends the traditional relational model with a variety of features such as structured and collection types, as well as object orientation.

An object typically has two components: state (value) and behavior (operations). It can have a complex data structure as well as specific operations defined by the programmer. Objects in an OOPL (OO Programming Language) exist only during program execution; therefore, they are called **transient objects**. An OO DB can extend the existence of objects so that they are stored permanently in a database, and hence the objects become persistent objects that exist beyond program termination and can be retrieved later and shared by other programs. In other words, OO databases store

persistent objects permanently in secondary storage, and allow the sharing of these objects among multiple programs and applications.

The Object-Relational (OR) model is very similar to the relational model; however, it treats every entity as an object (instance of a class), and a relationship as an inheritance.

Some features and benefits of an Object-Relational model are:

- Support for complex, user defined types
- Object inheritance
- Extensible objects

Object-Relational databases have the capability to store object relationships in relational form.

NOSQL

NoSQL is an approach to databases that represents a shift away from traditional relational database management systems (RDBMS) NoSQL can mean “not SQL” or “not only SQL.” NoSQL is particularly useful for storing unstructured data, which is growing far more rapidly than structured data and does not fit the relational schemas of RDBMS. Common types of unstructured data include: user and session data; chat, messaging, and log data; time series data such as IoT (Internet of Think) and device data; and large objects such as video and images.

Types of NOSQL DBS

Several different varieties of NoSQL databases have been created to support specific needs and use cases. These fall into four main categories:

- **Key-value data stores** : Key-value NoSQL databases emphasize simplicity and are very useful in accelerating an application to support high-speed read and write

processing of non-transactional data. Stored values can be any type of binary object (text, video, JSON document, etc.) and are accessed via a key. The application has complete control over what is stored in the value, making this the most flexible NoSQL model. Data is partitioned *and replicated* across a cluster to get scalability and availability. For this reason, key value stores often do not support transactions. However, they are highly effective at scaling applications that deal with high-speed, non-transactional data.

- Document stores : Document databases typically store self-describing JSON, XML, and BSON documents. They are similar to key-value stores, but in this case, a value is a single document that stores all data related to a specific key. Popular fields in the document can be indexed to provide fast retrieval without knowing the key. Each document can have the same or a different structure.

- Wide-column stores: Wide-column NoSQL databases store data in tables with rows and columns similar to RDBMS, but names and formats of columns can vary from row to row across the table. Wide-column databases group columns of related data together. A query can retrieve related data in a single operation because only the columns associated with the query are retrieved. In an RDBMS, the data would be in different rows stored in different places on disk, requiring multiple disk operations for retrieval.

- Graph stores: A graph database uses graph structures to store, map, and query relationships. They provide index-free adjacency, so that adjacent elements are linked together without using an index

- **Multi-modal databases** leverage some combination of the four types described above and therefore can support a wider range of applications.