Extended ER Features

The basic ER features you learned in previous are enough to model most simple database requirements. But as the system grows and the relationships become more complex, these basic features start falling short. In such cases, it's better to use *Extended ER (EER) features* — like specialization, generalization, aggregation, and inheritance — to design a more accurate and structured database schema.

1.Specialisation

Specialisation is used when one broad entity set needs to be divided into more specific entity sets.

1. In the ER model, sometimes an entity set contains different types of entities that are not fully similar. To represent these differences clearly, we group them into smaller, distinct entity sets.
2. Specialisation means breaking one large entity set into sub-entity sets based on their functions, characteristics, or special features.
3. It follows a **top-down approach** — starting from a general entity and moving towards more specific entities.
4. Example:
   A **Person** entity can be specialised into **Customer**, **Student**, and **Employee**.
   - *Person* is the **superclass**.
   - The specialised entities are called **subclasses**.
5. Between superclass and subclass, the relationship is an **"is-a"** relationship (e.g., Student *is a* Person).
   ER diagrams show this using a **triangle symbol**.
6. **Why do we use Specialisation?**
   - Some attributes apply only to a few members of the main entity set.
     (Example: "salary" applies only to Employee, not to Student or Customer.)
   - It allows the database designer to represent the unique features of each sub-entity clearly.

Example of Specialisation

Consider an entity **Vehicle**.

A vehicle can be of different types, and each type has its own special features. So we specialise the **Vehicle** entity into more specific subclasses.

Superclass:

- **Vehicle**
  - Common attributes: *vehicle_id*, *brand*, *model*, *engine_no*

Subclasses (through specialisation):

1. **Car**
   - Special attributes: *number_of_doors*, *boot_space*
2. **Bike**
   - Special attributes: *is_geared*, *fuel_tank_capacity*
3. **Truck**
   - Special attributes: *load_capacity*, *axle_count*

Here,

- **Car is-a Vehicle**,
- **Bike is-a Vehicle**,
- **Truck is-a Vehicle**.

This is the **"is-a"** relationship.
All these subclasses inherit the common attributes of the Vehicle entity but also have their own unique attributes.

Thus, specialisation helps in explaining and storing the distinctive details of each type without mixing everything into one big entity.

2.Generalisation

Generalisation is used when several specific entity sets share common features, and we want to combine them into one broader entity.

1. It is simply the **reverse of Specialisation**.
2. When a database designer notices that two or more entity sets have overlapping or similar properties, a new **generalised entity set** is created.
   This newly formed entity becomes the **superclass**, while the original entities become subclasses.
3. The relationship between subclass and superclass remains an **"is-a"** relationship.
   (Example: Car *is a* Vehicle, Bus *is a* Vehicle.)
4. Example:
   Suppose we have three entities — **Car**, **Jeep**, and **Bus**.
   All of them share some common attributes such as *vehicle_no*, *brand*, *model*, *engine_capacity*.
   Instead of repeating these attributes in all three entities, they can be **generalised into a single entity set called "Vehicle".**
5. Generalisation follows a **Bottom-Up approach** — starting from specific entities and combining them into a more general entity.
6. **Why Generalisation?**
   - It makes the database structure cleaner and easier to manage.
   - Common attributes are written only once, avoiding duplication and saving storage.

Example of Generalisation

Imagine you have the following entity sets in your database:

1. Car

- Attributes: *car_id*, *brand*, *model*, *engine_no*, *seating_capacity*

2. Jeep

- Attributes: *jeep_id*, *brand*, *model*, *engine_no*, *drive_type* (4×4, etc.)

3. Bus

- Attributes: *bus_id*, *brand*, *model*, *engine_no*, *route_no*

You can see that **brand**, **model**, and **engine_no** are common attributes across all three.

Instead of repeating these attributes in each entity, the designer generalises them into a new superclass.

Generalised Superclass: Vehicle

- Common attributes:
  vehicle_id, *brand*, *model*, *engine_no*

Subclasses:

- **Car**
  - Special attributes: *seating_capacity*
- **Jeep**
  - Special attributes: *drive_type*
- **Bus**
  - Special attributes: *route_no*

Here:

- **Car is-a Vehicle**
- **Jeep is-a Vehicle**
- **Bus is-a Vehicle**

Generalisation avoids duplicate data and gives the database a cleaner and more organised structure.

3. Attribute Inheritance

In both **Specialisation** and **Generalisation**, the entities at the lower level automatically receive all the attributes of the higher-level entity.

Example

If **Person** has attributes like:

- *person_id*

- *name*
- *address*

Then the subclasses:

- **Customer**
- **Employee**

will automatically have these attributes, along with their own special attributes.

So,

- **Customer** = *person_id, name, address + customer_specific attributes*
- **Employee** = *person_id, name, address + employee_specific attributes*

This prevents attribute repetition and keeps the design consistent.

4. Participation Inheritance

Whenever a **superclass** participates in a relationship, all its **subclasses** also participate in that same relationship by default.

Example

Suppose **Person** is connected to a relationship **"has_account"** with **BankAccount**.

Then automatically:

- **Customer** has_account BankAccount
- **Employee** has_account BankAccount

There is no need to draw separate relationships for Customer and Employee.

This saves time and avoids duplication in the ER diagram.

5. Aggregation

Aggregation helps when you need to show a relationship **involving another relationship**.
Instead of drawing complex relationship-to-relationship links, you convert that relationship into a higher-level abstract entity.

Simple Example

Consider:

- **Project**
- **Employee**
- Relationship: **works_on** (Employee works on Project)

Now consider **Department** wants to **supervise** the *works_on* relationship.

Instead of connecting Department to both Employee and Project again, you treat **works_on** like an abstract entity.

Employee ---- works_on ---- Project

↑

|

(Abstract Entity)

↑

supervised_by

↑

Department

Why Aggregation?

- It prevents drawing multiple overlapping relationships.
- It reduces clutter.
- It represents higher-level concepts clearly.

Real-Life Example

A **Loan** is approved when:

- **Customer** applies for **Loan**
- **BankEmployee** verifies the Loan

If a **Manager** supervises this *verification*, we treat the **verification** relationship as an aggregated entity and then connect Manager to it.