1. From Idea to Database: The 'Big Picture' of Design

Welcome! Let's be honest: just looking at SQL code or a complex diagram can be confusing. The "why" is often missing. Why do we draw these diagrams? How do they become actual code?

This lecture is the "missing manual." We're going to walk through the **entire workflow**, from a simple idea in your head to a real, working database.

Think of it like building a house:

1. **The Sketch:** You talk with the owner and draw a simple sketch. ("A 3-bedroom house with a big kitchen.")
2. **The Blueprint:** An architect turns the sketch into a detailed, technical blueprint. ("The wall here is 3.5m, using 2x4 studs...")
3. **The Building:** The construction crew uses the blueprint to build the *actual house*.

Database design is the **exact same**.

From Idea to Database: The 'Big Picture' of Design

- Resources
- The 3-Step Design Workflow
- Step 1: The Sketch (Conceptual Model)
  - Tool #1: The ERD (Entity-Relationship Diagram)
    - The Most Important Part: Cardinality (Bản số)
  - Tool #2: The Class Diagram (UML)
    - The Class Diagram's Superpower: Inheritance (Kế thừa)
- Step 2: The "Magic" (Converting to a Logical Model)
  - Rule 1: Every Entity becomes a Table.
  - Rule 2: Relationships are handled by Cardinality.
    - Case 1: 1-to-Many (`1..n`)
    - Case 2: 1-to-1 (`1..1`)
    - Case 3: Many-to-Many (`n..n`)
- Step 3: The Building (Physical Model)
- Final Summary: Why Bother?
- PlantUML: From Sketch to Code
  - Example 1: The "1-to-Many" Relationship (Student-Class)
    - 1. Conceptual Sketch (ERD in PlantUML)
    - 2. Conceptual Sketch (Class Diagram in PlantUML)
    - 3. Logical Blueprint (Text)
    - 4. Physical Build (SQL)

Example 2: The "Many-to-Many" Relationship (Student-Subject)
1. Conceptual Sketch (ERD in PlantUML)
2. Conceptual Sketch (Class Diagram in PlantUML)
3. Logical Blueprint (Text)
4. Physical Build (SQL)

1.1. Resources

- Database Management Systems CS 370
  - Three Level Database Architecture
  - The Entity-Relationship Model

1.2. The 3-Step Design Workflow

Everything in database design fits into one of these three steps, and each step is for a different **audience**.

STEP	MODEL	ANALOGY	WHO IS IT FOR?
1	**Conceptual**	The Sketch	**Everyone** (Clients, Managers, Developers)
2	**Logical**	The Blueprint	**Designers** (DB Administrators, System Architects)
3	**Physical**	The Building	**Builders & Users** (DBAs and Developers)

1.3. Step 1: The Sketch (Conceptual Model)

Who is this for? This is the "**meeting room" diagram**. It's for **everyone**. Clients, managers, developers, and DBAs all gather around this sketch to agree on *what* to build.

Goal: To capture all the *business rules* in a simple picture. We must answer questions like:

- "Can one customer have multiple accounts?"
- "Does every employee *have* to be in a department?"
- "What information do we need to store for a 'product'?"

This is the most important step for avoiding misunderstandings. We use two main tools for this sketch:

1.3.1. Tool #1: The ERD (Entity-Relationship Diagram)

This is the best tool for designing a **database**. It is a *data-first* model, meaning it focuses *only* on the data we need to store. It has 3 key parts:

1. **Entity (Thực Thể):** A "thing" you want to store. A **noun**.
    - **Examples:** `SINHVIEN`, `MONHOC` (Subject), `GIAOVIEN` (Teacher).
    - **How we draw it:** A rectangle.
2. **Attribute (Thuộc Tính):** A property of an entity. An **adjective**.
    - **Examples:** `HOTEN` (Full Name), `NGAYSINH` (Date of Birth).
    - **How we draw it:** An oval connected to the entity. We underline the Primary Key (e.g., `MSSV`).
3. **Relationship (Mối Kết Hợp):** How two entities are connected. A **verb**.
    - **Examples:** A `SINHVIEN`... `THI` (takes)... a `MONHOC`.
    - **How we draw it:** A diamond connecting the

entities.

1.3.1.1. The Most Important Part: Cardinality (Bản số)

This tells us the *business rules* of the relationship, using `(min, max)` notation.

- `(1,n)`: "One-to-Many"
- `(n,n)`: "Many-to-Many"
- `(0,1)`: "Zero-or-One"

Example: A `LOP` (Class) must have exactly one `GIAOVIEN` (Teacher) as its head. A `GIAOVIEN` can be the head of *zero or one* `LOP`.

```
GIAOVIEN -- (0,1) -- (Homeroom) -- (1,1) -- LOP
```

1.3.2. Tool #2: The Class Diagram (UML)

This is the best tool for designing an **application**. It is a *code-first* model because it shows **data + behavior**.

Who is this for? This is the main diagram for **Application Developers** (people writing Java, C#, Python, etc.).

A class diagram has 3 parts:

1. **Name:** `SINHVIEN`

2. **Attributes:** `mssv`, `tenSV` (The data)
3. **Methods:** `ThemSV()`, `TimSV()` (The *behaviors* or *functions*)

This diagram helps a developer plan their actual *code objects*.

1.3.2.1. The Class Diagram's Superpower: Inheritance (Kế thừa)

- This is the key difference!

A Class Diagram can easily show "is-a" relationships, which a basic ERD (Chen notation) cannot.

- `NV_KYTHUAT` (Technical Staff) **is a** `NHANVIEN` (Employee).
- `NV_HANHCHANH` (Admin Staff) **is a** `NHANVIEN`.

You draw this with a hollow triangle arrow. This is **essential** for object-oriented code.

1.4. Step 2: The "Magic" (Converting to a Logical Model)

Who is this for? This is the primary "blueprint" for the **Database Administrator (DBA)** and **System Architect**.

This is the technical **blueprint**. We convert our "sketch" (the ERD) into a formal, text-based plan. This plan is **independent of any specific database** (it's not SQL, but it's very close).

A DBA uses this to plan:

- What tables are needed?
- What columns are in each table?
- How are the tables linked (what are the Foreign Keys)?
- What are the Primary Keys?

This is where you see that familiar format:

- `SINHVIEN (MSSV, HOTEN, DIACHI)`
- `LOP (MALOP, TENLOP, MAGV)`

So... how do you get from your *drawing* (ERD) to your *blueprint* (Logical Model)? You just follow the rules.

1.4.1. Rule 1: Every Entity becomes a Table.

This is the easy one.

- The `SINHVIEN` entity becomes `SINHVIEN (...)`.
- The `MONHOC` entity becomes `MONHOC (...)`.

1.4.2. Rule 2: Relationships are handled by Cardinality.

This is where the magic happens. You don't make tables for relationships... **you use Foreign Keys**.

1.4.2.1. Case 1: 1-to-Many (`1..n`)

*The "**Many**" side gets the Foreign Key.*

- **Example:** One `PHONGBAN` (Department) has *many* `NHANVIEN` (Employees).
- **Rule:** `NHANVIEN` is the "many" side. So, we add `MAPB` (Dept. ID) into the `NHANVIEN` table.
- **Blueprint:**
    - `PHONGBAN (MAPB, TenPB)`
    - `NHANVIEN (MANV, TenNV, **MAPB**)`

1.4.2.2. Case 2: 1-to-1 (`1..1`)

*You can put the Foreign Key in **either table**.*

- **Example:** One `NHANVIEN` has one `LYLICH` (Resume).
- **Rule:** You can put `MANV` in the `LYLICH` table.
- **Blueprint:**
    - `NHANVIEN (MANV, TenNV)`
    - `LYLICH (MALL, ChiTiet, **MANV**)`

1.4.2.3. Case 3: Many-to-Many (`n..n`)

You ***MUST*** *create a* ***NEW table*** *(a "linking table").*

- **Example:** *One* `SINHVIEN` takes *many* `MONHOC`. *One* `MONHOC` is taken by *many* `SINHVIEN`.
- **Rule:** You cannot put a foreign key in either! How would you store many keys in one field? You can't.
- **Solution:** Create a new table, often named after the relationship (e.g., `THI` or `KETQUA`).
- **Blueprint:**
    - `SINHVIEN (MASV, TenSV)`
    - `MONHOC (MAMH, TenMH)`
    - `THI (**MASV**, **MAMH**, DIEM)`
        - The primary key of this new table is the *combination* of both foreign keys.

1.5. Step 3: The Building (Physical Model)

Who is this for? This is where the **DBA** and the **Developer** meet. They both rely on this final, concrete model.

This is the **actual, physical code**. You take your "blueprint" (the Logical Model) and write the `CREATE TABLE` statements for your specific database (like SQL Server, MySQL, etc.).

- **The DBA's Job (The Builder):** The DBA takes the Logical Model and *builds* the database. They run the `CREATE TABLE` scripts, set up indexes for performance, and define security rules.
- **The Developer's Job (The User):** The Developer reads this physical schema to understand the *exact* table and column names so they can write their application code (e.g., `SELECT * FROM SINHVIEN WHERE MASV = '123'`).

Your Blueprint (Logical):

LOP (MALOP, TENLOP, MAGV)

Your Building (Physical):

```
CREATE TABLE LOP (
    MALOP VARCHAR(10) PRIMARY KEY,
    TENLOP NVARCHAR(100),
    MAGV VARCHAR(10),
    FOREIGN KEY (MAGV) REFERENCES GIAOVIEN(MAGV)
);
```

And you're done!

1.6. Final Summary: Why Bother?

Why not just skip to writing SQL?

You don't build a house without a blueprint.

1. **We "Sketch" (Conceptual):** We draw an **ERD** so *everyone* can agree on the business rules. (Prevents building the wrong thing).
2. **We "Blueprint" (Logical):** We create a **Logical Model** so the *DBA* can plan the database perfectly, without errors. (Prevents building a weak or broken thing).
3. **We "Build" (Physical):** The *DBA* writes the **SQL** based on the blueprint, and the *Developer* uses it. (This is the final, high-quality result).

Following this process saves you from **costly mistakes**. It's much, much easier to change a drawing (Step 1) than it is to rebuild a live database (Step 3)!

1.7. PlantUML: From Sketch to Code

Here are two practical examples showing the full workflow, with code for each step.

1.7.1. Example 1: The "1-to-Many" Relationship (Student-Class)

Business Rule: A student *must* belong to exactly one class. A class can have *one or many* students.

1.7.1.1. 1. Conceptual Sketch (ERD in PlantUML)

This is for the "meeting room" to confirm the rule.

```
@startchen
' A SINHVIEN must be in exactly 1 LOP
' A LOP must have at least 1, but can have many,
SINHVIEN

entity SINHVIEN {
  MSSV
  HOTEN
}

entity LOP {
  MALOP
  TENLOP
}

relationship R_BT {
  Name
}

' (min,max) notation
SINHVIEN -(1,1)- R_BT
LOP -(1,n)- R_BT

```

```
@endchen
```

1.7.1.2. 2. Conceptual Sketch (Class Diagram in PlantUML)

This is for the *developer* to plan their code.

```
@startuml
skinparam classAttributeIconSize 0
skinparam defaultFontName "Inter"

class Lop {
  - maLop: string
  - tenLop: string
  + themSinhVien(sv: SinhVien)
}

class SinhVien {
  - mssv: string
  - hoTen: string
  + layDiemTB()
}

' "1" Lop has "1..*" (one or many) SinhVien
' A SinhVien is in exactly "1" Lop
Lop "1" -- "1..*" SinhVien : has
@enduml
```

1.7.1.3. 3. Logical Blueprint (Text)

This is for the *DBA* to plan the tables. We follow the "1-to-Many" rule: the "Many" side (`SINHVIEN`) gets the foreign key.

```
LOP (MALOP, TENLOP)
SINHVIEN (MSSV, HOTEN, MALOP)
```

1.7.1.4. 4. Physical Build (SQL)

The *DBA* and *Developer* use this final code.

```sql
-- The '1' side is created first
CREATE TABLE LOP (
    MALOP VARCHAR(10) PRIMARY KEY,
    TENLOP NVARCHAR(100)
);

-- The 'Many' side is created next, referencing the '1' side
CREATE TABLE SINHVIEN (
    MSSV VARCHAR(10) PRIMARY KEY,
    HOTEN NVARCHAR(100),
    MALOP VARCHAR(10),
    FOREIGN KEY (MALOP) REFERENCES LOP(MALOP)
);
```

1.7.2. Example 2: The "Many-to-Many" Relationship (Student-Subject)

Business Rule: A student can take *many* subjects. A subject can be taken by *many* students. When a student takes a subject, they get a *grade*.

1.7.2.1. 1. Conceptual Sketch (ERD in PlantUML)

Note how the *relationship itself* (`THI`) has an attribute (`DIEM`).

```
@startchen
' A SINHVIEN can THI (take) 1 or more MONHOC
' A MONHOC can be taken by 1 or more SINHVIEN
skinparam defaultFontName Iosevka

entity SINHVIEN {
  MSSV<<key>>
  HOTEN
}

entity MONHOC {
  MAMH <<key>>
  TENMON
}

```

```
relationship R_THI {
  DIEM
}

SINHVIEN -(1,n)- R_THI
MONHOC -(1,n)- R_THI
@endchen
```

1.7.2.2. 2. Conceptual Sketch (Class Diagram in PlantUML)

For the developer, this "Many-to-Many" is modeled as an **Association Class**. The `KetQua` class links `SinhVien` and `MonHoc`.

```
@startuml
skinparam classAttributeIconSize 0
skinparam defaultFontName "Inter"

class SinhVien {
  - mssv: string
  - hoTen: string
  + layKetQua()
}

class MonHoc {
  - maMH: string
```

```
  - tenMH: string
}

' The "KetQua" (Result) class links the two
' It has its own attributes, like "diem"
class KetQua {
  - diem: float
}

SinhVien "1" -- "0..*" KetQua
MonHoc "1" -- "0..*" KetQua
@enduml
```

1.7.2.3. 3. Logical Blueprint (Text)

For the *DBA*. We follow the "Many-to-Many" rule: create a **new linking table** (`THI` or `KETQUA`).

```
SINHVIEN (MSSV, HOTEN)
MONHOC (MAMH, TENMON)
THI (MSSV, MAMH, DIEM)
```

1.7.2.4. 4. Physical Build (SQL)

The final code. Note the composite Primary Key in the `THI` table.

```sql
CREATE TABLE SINHVIEN (
    MSSV VARCHAR(10) PRIMARY KEY,
    HOTEN NVARCHAR(100)
);

CREATE TABLE MONHOC (
    MAMH VARCHAR(10) PRIMARY KEY,
    TENMON NVARCHAR(100)
);

-- This is the linking table
CREATE TABLE THI (
    MSSV VARCHAR(10),
    MAMH VARCHAR(10),
    DIEM FLOAT,
    -- The Primary Key is the *combination* of the two keys
    PRIMARY KEY (MSSV, MAMH),

    -- Set up both Foreign Keys
    FOREIGN KEY (MSSV) REFERENCES SINHVIEN(MSSV),
    FOREIGN KEY (MAMH) REFERENCES MONHOC(MAMH)
);
```