Task 1.1: Superkey and Candidate Key Analysis

Given the following relations, identify all superkeys and candidate keys:

Relation A: Employee

```
Employee(EmpID, SSN, Email, Phone, Name, Department, Salary)
```

Sample Data:

EmpID	SSN	Email	Phone	Name	Department	Salary
101	123-45-6789	john@company.com	555-0101	John	IT	75000
102	987-65-4321	mary@company.com	555-0102	Mary	HR	68000
103	456-78-9123	bob@company.com	555-0103	Bob	IT	72000

Your Tasks: 1. List at least 6 different superkeys 2. Identify all candidate keys 3. Which candidate key would you choose as primary key and why? 4. Can two employees have the same phone number? Justify your answer based on the data shown.

Relation B: Course Registration

```
Registration(StudentID, CourseCode, Section, Semester, Year, Grade, Credits)

Business Rules:
- A student can take the same course in different semesters
- A student cannot register for the same course section in the same semester
- Each course section in a semester has a fixed credit value
```

Your Tasks: 1. Determine the minimum attributes needed for the primary key 2. Explain why each attribute in your primary key is necessary 3. Identify any additional candidate keys (if they exist)

Relation A — Employee(EmpID, SSN, Email, Phone, Name, Department, Salary)

1) Superkeys:

- {EmpID}
- {SSN}
- {Email}
- {EmpID, SSN}
- {EmpID, Email}
- {SSN, Email}

  (others also valid, e.g., {EmpID, Phone}, {SSN, Phone}.)

2) Candidate keys

- {EmpID}
- {SSN}
- {Email}

  (Phone is **not** assumed unique.)

3) Primary key choice

Choose **EmpID** as the PK.
Reasons: numeric/compact, internal (doesn't expose personal data), stable even if SSN or Email changes, and easy to index/join.
**(Не раскрывает персональные данные (в отличие от SSN/Email).
Стабильный и короткий, удобен для внешних ссылок и индексов.
Если сотруднику меняют Email, PK не нужно переписывать.)

4)

From the data shown we **cannot guarantee uniqueness**. Sample values are different, but there is no rule saying "Phone is unique"; shared office phones are common. Therefore, Phone should **not** be treated as a key.

Relation B — `Registration(StudentID, CourseCode, Section, Semester, Year, Grade, Credits).`

1) Minimum attributes for the (composite) primary key

(StudentID, CourseCode, Section, Semester, Year)

2) Why each attribute is necessary

- **StudentID** — identifies which student the row belongs to.
- **CourseCode** — sections are defined per course; "Section 1" can exist for many courses.
- **Section** — a course can have multiple sections in a term.
- **Semester** — same course/section can recur in different semesters.
- **Year** — distinguishes the same semester name across years

*(This composite key ensures a student cannot register for the **same course section in the same term**, but can take it again in a different term.)*

3) Additional candidate keys?

- For **Registration**, **no additional natural candidate key** exists in the given attributes.
- The combination **(CourseCode, Section, Semester, Year)** uniquely identifies a **course offering** (and determines Credits) but **not** a registration row (many students share that). If desired, you could normalize into:
    - `CourseOffering(CourseCode, Section, Semester, Year, Credits)` (PK = those four)
    - `Registration(StudentID, CourseCode, Section, Semester, Year, Grade)` (FK to CourseOffering)

Task 1.2: Foreign Key Design

Design the foreign key relationships for this university system:

Given Tables:

```
Student(StudentID, Name, Email, Major, AdvisorID)
Professor(ProfID, Name, Department, Salary)
Course(CourseID, Title, Credits, DepartmentCode)
Department(DeptCode, DeptName, Budget, ChairID)
Enrollment(StudentID, CourseID, Semester, Grade)
```

Your Tasks: 1. Identify all foreign key relationships

1) Minimal Primary Key
PK = (StudentID, CourseCode, Section, Semester, Year)

2) Why each attribute in the PK is necessary

- *StudentID — without it, many students could share the same offering (CourseCode, Section, Semester, Year) → duplicates.*
- *CourseCode — "Section 1" exists for many courses; removing CourseCode conflates different courses → duplicates.*
- *Section — a course can have multiple sections in the same term; removing Section merges them → duplicates.*
- *Semester — the same course/section can run in different semesters of the same year (Spring vs Fall); removing Semester merges them → duplicates.*
- *Year — the same semester name repeats across years (Fall 2024 vs Fall 2025); removing Year merges them → duplicates.*

Note: Grade and Credits are not part of the key; they depend on the registration/offering and do not ensure uniqueness.

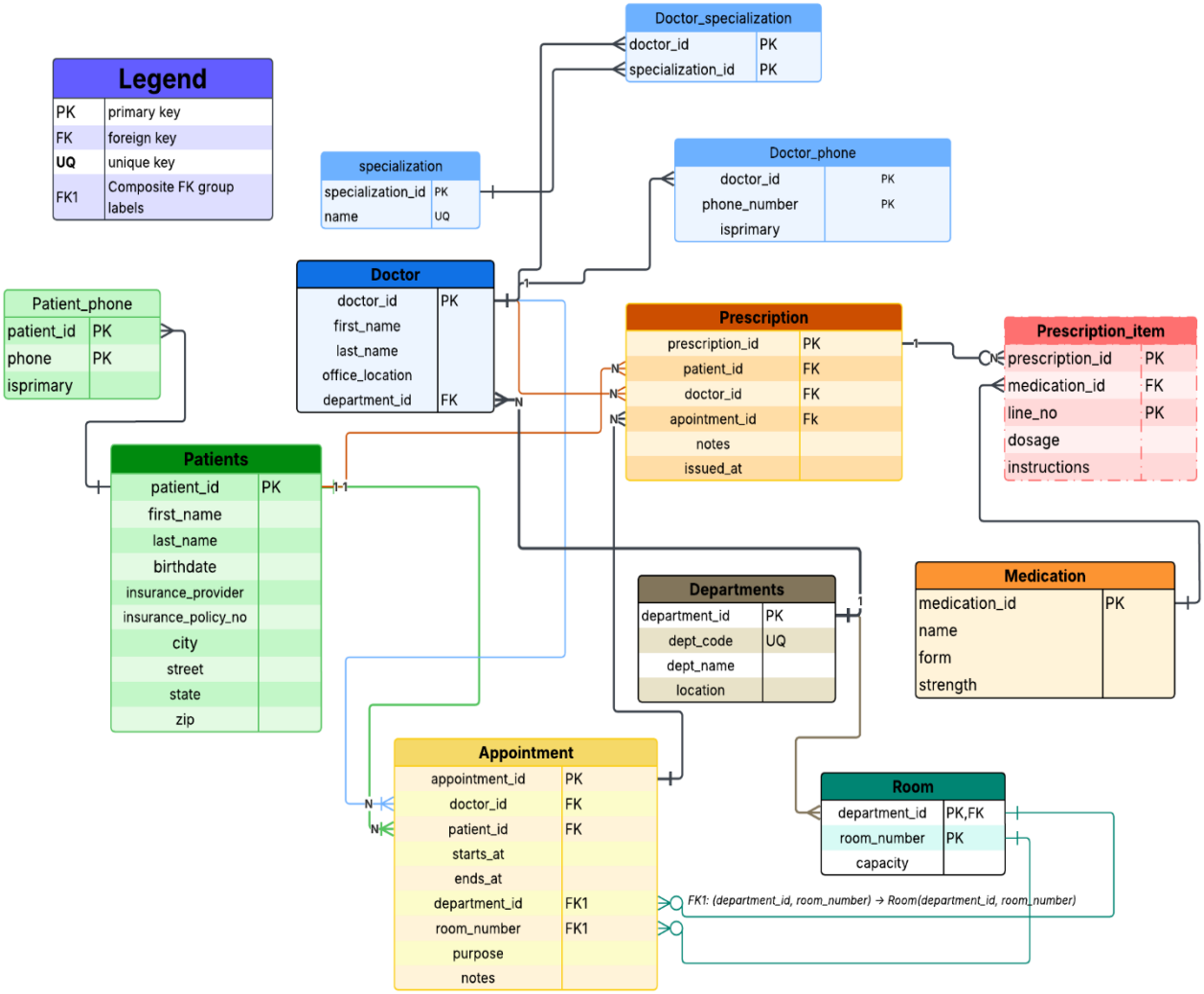
Task 2.1: Hospital Management System

Scenario: Design a database for a hospital management system.

Requirements:

- **Patients** have unique patient IDs, names, birthdates, addresses (street, city, state, zip), phone numbers (multiple allowed), and insurance information
- **Doctors** have unique doctor IDs, names, specializations (can have multiple), phone numbers, and office locations
- **Departments** have department codes, names, and locations
- **Appointments** track which patient sees which doctor at what date/time, the purpose of visit, and any notes
- **Prescriptions** track medications prescribed by doctors to patients, including dosage and instructions
- **Hospital Rooms** are numbered within departments (room 101 in Cardiology is different from room 101 in Neurology)

Your Tasks: 1. Identify all entities (specify which are strong and which are weak) 2. Identify all attributes for each entity (classify as simple, composite, multi-valued, or derived) 3. Identify all relationships with their cardinalities (1:1, 1:N, M:N) 4. Draw the complete ER diagram using proper notation 5. Mark primary keys



The schema models patients, doctors, departments, rooms, appointments, and prescriptions. **Patients** store demographics and insurance; multiple phone numbers are kept in **Patient_phone**. **Doctors** include basic info and a link to their department; multiple phones go to **Doctor_phone**, and multiple specialties are handled by the M:N table **Doctor_specialization** with the **specialization** lookup. **Departments** define hospital units. **Room** represents rooms numbered **within a department** and is identified by the composite key *(department_id, room_number)* (a weak entity relative to Department).

Appointment records a visit— which patient saw which doctor, start/end time, purpose/notes, and the room (FK to **Room**). **Medication** lists drugs. **Prescription** captures a doctor's prescription for a patient (optionally tied to an appointment), while **Prescription_item** stores the line items (medication, dosage, instructions).
This decomposition removes redundancy, correctly models M:N relationships, and satisfies 3NF/BCNF: every non-key attribute depends only on the key of its table.

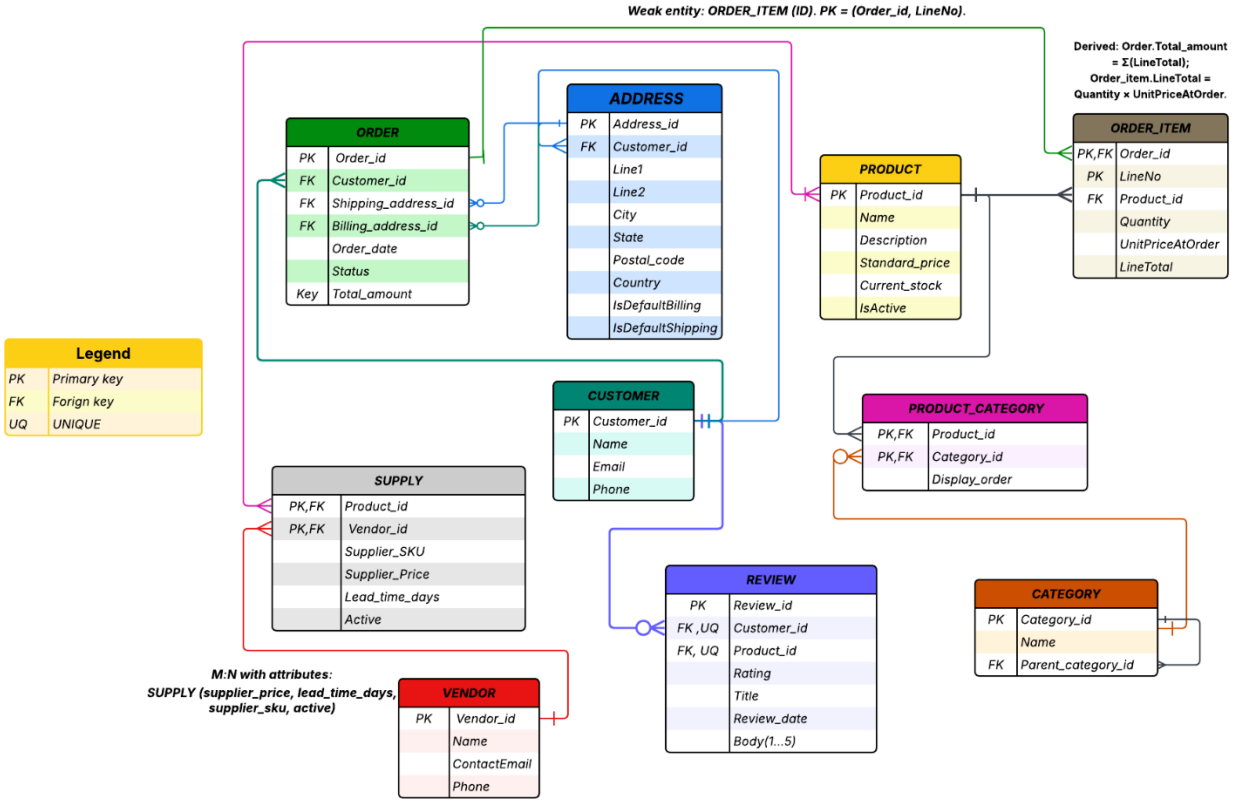
Task 2.2: E-commerce Platform

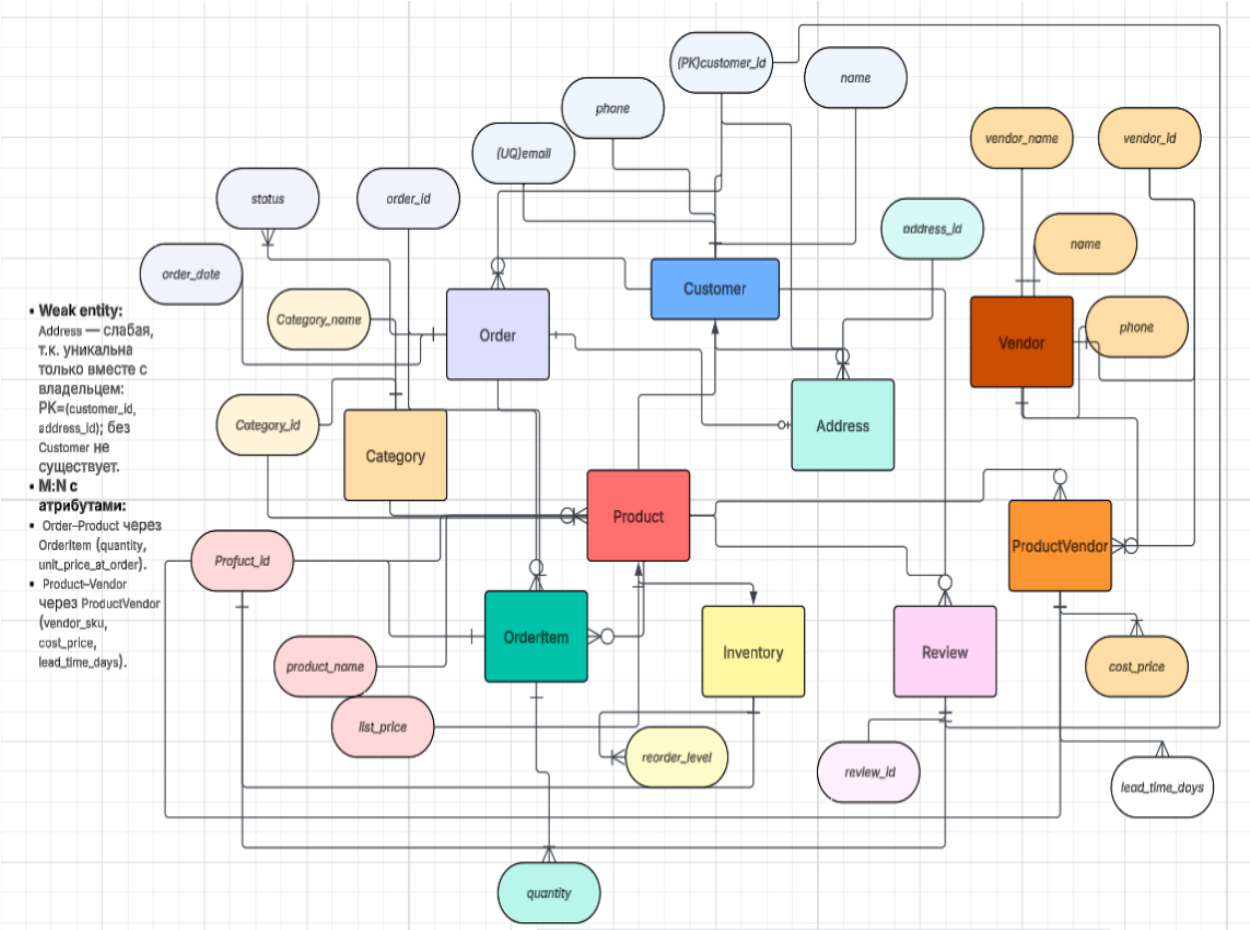
Scenario: Design a simplified e-commerce database.

Requirements:

- **Customers** place **Orders** for **Products**
- **Products** belong to **Categories** and are supplied by **Vendors**
- **Orders** contain multiple **Order Items** (quantity and price at time of order)
- **Products** have reviews and ratings from customers
- Track **Inventory** levels for each product
- **Shipping addresses** can be different from customer billing addresses

Your Tasks: 1. Create a complete ER diagram 2. Identify at least one weak entity and justify why it's weak 3. Identify at least one many-to-many relationship that needs attributes





Core entities

- Customer — shopper profile (name, email, phone).
- Address *(weak)* — a customer's billing/shipping addresses. Identified by *(customer_id, address_id)* and cannot exist without its owner.
- Order — a customer's purchase; holds order date, status, and pointers to billing and shipping addresses.
- Product — item for sale; belongs to one Category.
- Category — catalog taxonomy (optionally hierarchical via parent_category_id).
- Vendor — supplier of products.
- Inventory — current on-hand quantity and reorder level per product.
- OrderItem — line items inside an order (quantity and price captured at time of order).
- Review — customer's rating/text for a product.
- ProductVendor (or Supply) — relationship that says which vendors supply which products with purchasing details.

Relationships & cardinalities (Crow's Foot)

- Customer 1 < Address *(identifying; Address is weak)*
- Customer 1 < Order
- Order 1 < OrderItem >1 Product *(M:N via OrderItem; attributes: quantity, unit_price_at_order)*
- Product 1 < Review >1 Customer *(M:N overall; each review belongs to exactly one product and one customer; optionally limit to one review per customer per product)*
- Category 1 < Product *(simple single-category model)*
- Product 1 — 0..1 Inventory *(one inventory record per product)*
- *Product < ProductVendor >Vendor (M:N with attributes: vendor_sku, cost_price, lead_time_days)*

- Order — Address: two separate optional links from Order to Address labeled billing and shipping (each points to exactly one Address row).

2) Weak entity (and why)

Address is a weak entity because:

- It is uniquely identified only together with its owner: *(customer_id, address_id)*.
- It has existence dependence on Customer (if the customer is deleted, their addresses are meaningless).
- The identifying relationship Customer → Address is *identifying* (owner's key participates in the weak entity's primary key).

(Note: some diagrams call OrderItem "weak," but it is better described as an associative/bridge entity for the M:N Order–Product relationship.)

---

3) M:N relationships that need attributes

1. Order ↔ Product → OrderItem
    - Attributes stored on the relationship: quantity, unit_price_at_order (and a computed line_total = quantity × unit_price_at_order).
    - Composite PK: (order_id, product_id) (or (order_id, line_no) if you prefer line numbers).
2. Product ↔ Vendor → ProductVendor *(aka Supply)*
    - Attributes stored on the relationship: vendor_sku, cost_price, lead_time_days, active.
    - Composite PK: (product_id, vendor_id).

4) Normalized relational schema (3NF, concise)

Show PK/ FK/ UQ marks; types omitted for brevity.

- Customer(PK customer_id, UQ email, name, phone)
- Address(PK/FK customer_id, PK address_id, label, line1, line2, city, state, postal_code, country, is_default_billing, is_default_shipping)
  FK customer_id → Customer
- Order(PK order_id, FK customer_id → Customer, FK billing_customer_id, FK billing_address_id, FK shipping_customer_id, FK shipping_address_id, order_date, status)
  (billing_ , shipping_) together reference Address(customer_id, address_id)
- Category(PK category_id, UQ name, FK parent_category_id → Category NULL)
- Product(PK product_id, name, description, standard_price, FK category_id → Category, is_active)
- Inventory(PK/FK product_id → Product, qty_on_hand, reorder_level)
- OrderItem(PK/FK order_id → Order, PK/FK product_id → Product, quantity, unit_price_at_order)
  (Option: use line_no instead of product_id in the PK.)
- Review(PK review_id, FK customer_id → Customer, FK product_id → Product, rating, title, review_text, review_date, UQ(customer_id, product_id) optional)
- Vendor(PK vendor_id, name, contact_email, phone)
- ProductVendor(PK/FK product_id → Product, PK/FK vendor_id → Vendor, vendor_sku, cost_price, lead_time_days, active)

Derived values (not stored or clearly labeled as derived):

- Order.total_amount = Σ(OrderItem.quantity × unit_price_at_order)
- OrderItem.line_total = quantity × unit_price_at_order

5) notes

- Address as weak ensures addresses are scoped to their owner and avoids cross-customer collisions.

- OrderItem captures point-in-time pricing (doesn't change if the product price later changes).
- ProductVendor keeps purchasing data by supplier (realistic procurement).
- The schema is in 3NF: reference data split out; M:N relationships isolated; no partial or transitive dependencies inside base tables.

---

6) Example queries

1. "Show top 5 products by average review rating (minimum 10 reviews)."
2. "For each vendor, list the products they supply with cost_price and lead_time_days."
3. "Find customers whose billing and shipping addresses are in different countries."

Task 4.1: Denormalized Table Analysis

Given Table:

```
StudentProject(StudentID, StudentName, StudentMajor, ProjectID, ProjectTitle,
               ProjectType, SupervisorID, SupervisorName, SupervisorDept,
               Role, HoursWorked, StartDate, EndDate)
```

Your Tasks: 1. **Identify functional dependencies:** List all FDs in the format A → B 2. **Identify problems:** - What redundancy exists in this table? - Give specific examples of update, insert, and delete anomalies 3. **Apply 1NF:** Are there any 1NF violations? How would you fix them? 4. **Apply 2NF:** - What is the primary key of this table? - Identify any partial dependencies - Show the 2NF decomposition 5. **Apply 3NF:** - Identify any transitive dependencies - Show the final 3NF decomposition with all table schemas

Student
PK StudentID
StudentName
StudentMajor
Supervisor
PK SupervisorID
SupervisorName
SupervisorDept
Project
PK ProjectID
ProjectTitle
ProjectType
FK SupervisorID → Supervisor
StudentProject *(связь студент–проект)*
PK/FK StudentID → Student
PK/FK ProjectID → Project
Role, HoursWorked, StartDate, EndDate
Связи (Crow's Foot):
Student **1**—< StudentProject >—**1** Project
Supervisor **1**—< Project

Task 4.1 — Denormalized Table Analysis
Given table
StudentProject(StudentID, StudentName, StudentMajor, ProjectID, ProjectTitle, ProjectType, SupervisorID, SupervisorName, SupervisorDept, Role, HoursWorked, StartDate, EndDate)

1) Functional Dependencies (FDs)

- *StudentID → StudentName, StudentMajor*
- *ProjectID → ProjectTitle, ProjectType, SupervisorID*
- *SupervisorID → SupervisorName, SupervisorDept*
- *(StudentID, ProjectID) → Role, HoursWorked, StartDate, EndDate*

(By transitivity: ProjectID → SupervisorID → SupervisorName, SupervisorDept so ProjectID → SupervisorName, SupervisorDept.)

2) Problems (redundancy & anomalies)

- *Redundancy*
  - *Student info repeats across all their project rows (same StudentName, StudentMajor).*
  - *Project info repeats for every assigned student (ProjectTitle, ProjectType, SupervisorID…).*
  - *Supervisor info repeats across all projects/same supervisor (SupervisorName, SupervisorDept).*
- *Update anomaly*
  If supervisor changes department, you must update it in many rows; if one row is missed, data becomes inconsistent.
- *Insert anomaly*
  You can't insert a new project (its title/type/supervisor) until at least one student is assigned—otherwise you have no place to store project data.
- *Delete anomaly*
  If you delete the last student on a project, you also delete the only copy of that project's and supervisor's details.

3) Apply 1NF

- *Values are atomic as listed (no arrays or repeating groups).*
  1NF status: already in 1NF.
  (If the business allows multiple non-overlapping periods for the same student on the same project, then you'd need a separate AssignmentPeriod table. With the given attributes, we assume one period per (Student, Project).)

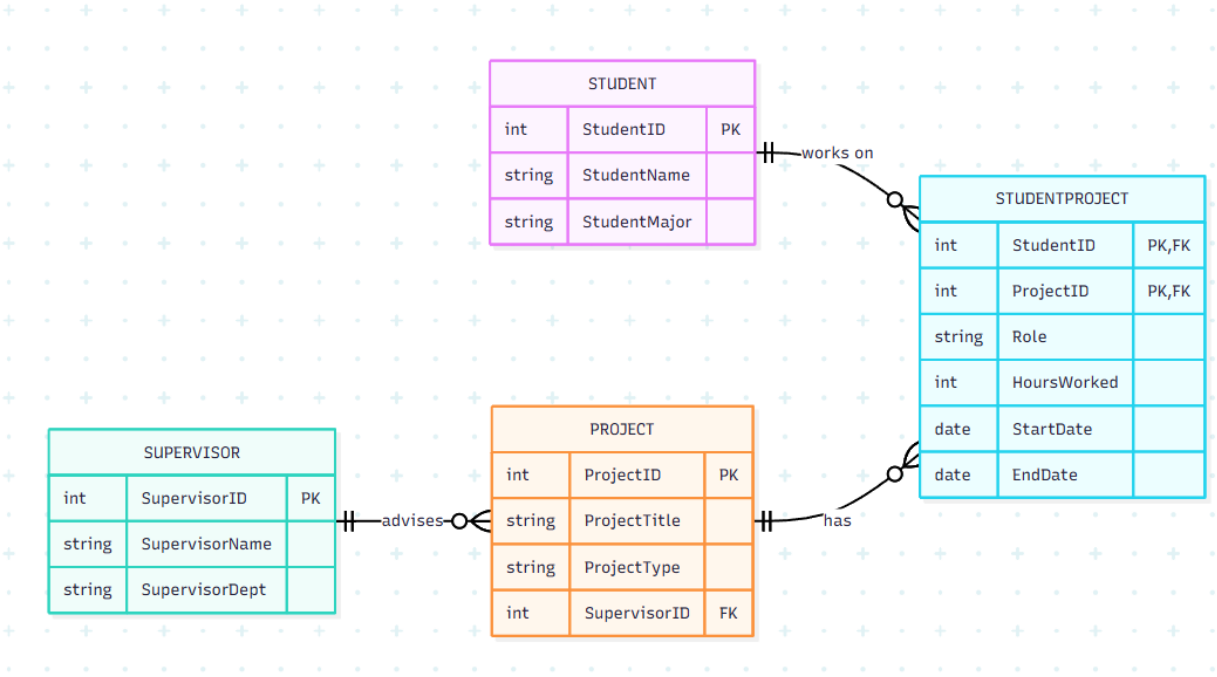
4) Apply 2NF

- *Natural PK (current table): (StudentID, ProjectID) (a student's assignment to a project).*
- *Partial dependencies (violate 2NF):*
  - *StudentID → StudentName, StudentMajor*
  - *ProjectID → ProjectTitle, ProjectType, SupervisorID*
- *2NF decomposition*
  - *Student(StudentID PK, StudentName, StudentMajor)*
  - *Project(ProjectID PK, ProjectTitle, ProjectType, SupervisorID FK)*
  - *StudentProject(StudentID PK/FK, ProjectID PK/FK, Role, HoursWorked, StartDate, EndDate)*
  - *(Keep supervisor attributes here for now; fixed in 3NF.)*

5) Apply 3NF

- *Transitive dependency: ProjectID → SupervisorID and SupervisorID → SupervisorName, SupervisorDept*
  → ProjectID → SupervisorName, SupervisorDept (via Supervisor)
  Move supervisor details to their own table.
- *Final 3NF schema (with keys & FKs)*
  - *Student(StudentID PK, StudentName, StudentMajor)*
  - *Supervisor(SupervisorID PK, SupervisorName, SupervisorDept)*
  - *Project(ProjectID PK, ProjectTitle, ProjectType, SupervisorID FK→Supervisor)*
  - *StudentProject(StudentID PK/FK→Student, ProjectID PK/FK→Project, Role, HoursWorked, StartDate, EndDate)*

This removes partial and transitive dependencies; all non-key attributes depend on a key, the whole key, and nothing but the key.



Task 4.2: Advanced Normalization

Given Table:

```
CourseSchedule(StudentID, StudentMajor, CourseID, CourseName,
               InstructorID, InstructorName, TimeSlot, Room, Building)
```

Business Rules:

- Each student has exactly one major
- Each course has a fixed name
- Each instructor has exactly one name
- Each time slot in a room determines the building (rooms are unique across campus)
- Each course section is taught by one instructor at one time in one room
- A student can be enrolled in multiple course sections

Your Tasks: 1. Determine the primary key of this table (hint: this is tricky!) 2. List all functional dependencies 3. Check if the table is in BCNF 4. If not in BCNF, decompose it to BCNF showing your work 5. Explain any potential loss of information in your decomposition

Student
PK StudentID
StudentMajor
Course
PK CourseID
CourseName
Instructor
PK InstructorID
InstructorName
Room
PK Room
Building
Section *(секция курса)*
PK CourseID
PK TimeSlot
PK Room
FK InstructorID → Instructor

Enrollment *(запись студента на секцию)*
PK/FK StudentID → Student
PK/FK CourseID → Section
PK/FK TimeSlot → Section
PK/FK Room → Section
Связи:
Course **1—<** Section
Room **1—<** Section
Instructor **1—<** Section
Section **1—<** Enrollment
Student **1—<** Enrollment
Ключи):

- **Section PK = (CourseID, TimeSlot, Room)**
- **Enrollment PK = (StudentID, CourseID, TimeSlot, Room)**

1) Primary Key

A row records a student enrolled in a specific course section. With no SectionID given, a section is identified by (CourseID, TimeSlot, Room).
So the row key is:
PK = (StudentID, CourseID, TimeSlot, Room)
(A student can take many sections; they should not have duplicate enrollment in the same section.)

2) Functional Dependencies

- StudentID → StudentMajor
- CourseID → CourseName
- InstructorID → InstructorName
- Room → Building
- (CourseID, TimeSlot, Room) → InstructorID *(each section has one instructor)*
- Key FD: (StudentID, CourseID, TimeSlot, Room) → *(all attributes)*

3) BCNF check

BCNF requires every non-trivial FD's determinant to be a superkey. Violations:

- StudentID → StudentMajor (StudentID is not a superkey of the whole table).
- CourseID → CourseName (CourseID not a superkey).
- InstructorID → InstructorName (InstructorID not a superkey).
- Room → Building (Room not a superkey).
- (CourseID, TimeSlot, Room) → InstructorID (this triple is not a superkey of the whole table, because many students can share the same section).

Therefore, the table is not in BCNF.

4) Decompose to BCNF (lossless, dependency-preserving)

Perform standard decomposition along the violating FDs:

- Student(StudentID PK, StudentMajor)
- Course(CourseID PK, CourseName)
- Instructor(InstructorID PK, InstructorName)
- Room(Room PK, Building)
- Section(CourseID, TimeSlot, Room, InstructorID;
  PK = (CourseID, TimeSlot, Room), FKs: CourseID→Course, Room→Room, InstructorID→Instructor)
- Enrollment(StudentID, CourseID, TimeSlot, Room;
  PK = (StudentID, CourseID, TimeSlot, Room), FK (CourseID, TimeSlot, Room)→Section, FK StudentID→Student)

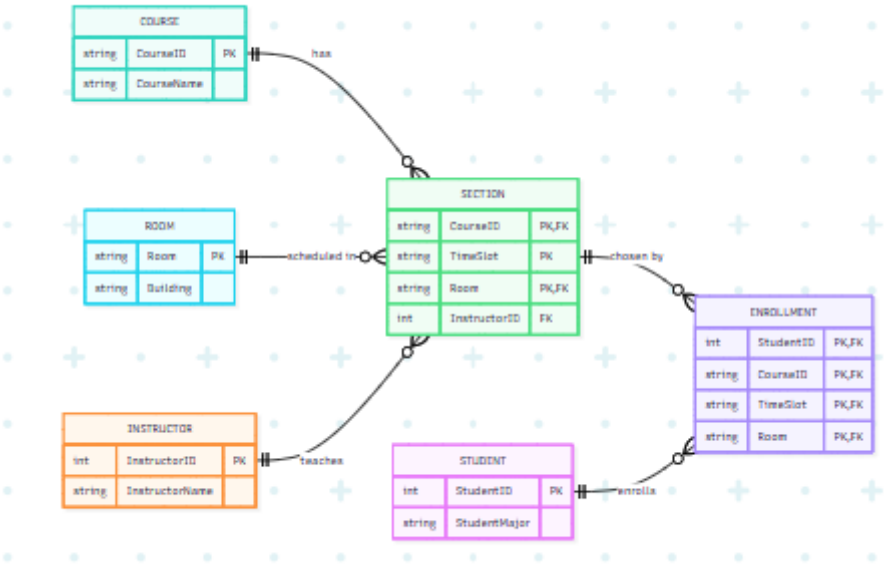
All relations above are in BCNF:

- In Student/Course/Instructor/Room, the only non-trivial FD has a key on the LHS.
- In Section, (CourseID, TimeSlot, Room) is the key and determines InstructorID.
- In Enrollment, the key is the entire tuple (StudentID, CourseID, TimeSlot, Room) and there are no other non-trivial FDs.

5) Information loss

- Lossless join: Yes—joining Enrollment ⋈ Section ⋈ Student ⋈ Course ⋈ Instructor ⋈ Room reconstructs all original rows (standard star-style schema).
- Dependency preservation: Each FD is preserved within one of the decomposed relations:
  - StudentID → StudentMajor (Student)
  - CourseID → CourseName (Course)
  - InstructorID → InstructorName (Instructor)
  - Room → Building (Room)
  - (CourseID, TimeSlot, Room) → InstructorID (Section)

No information is lost; we've removed anomalies and achieved BCNF.



Task 5.1: Real-World Application

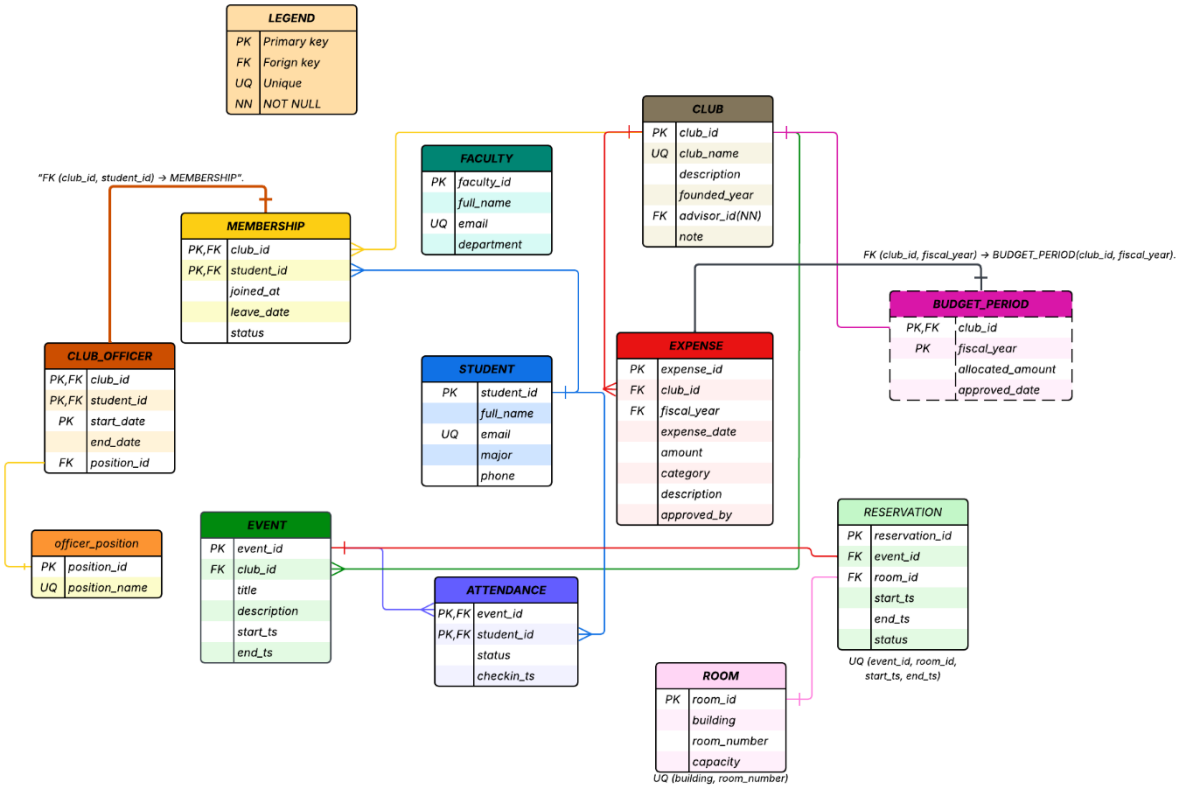
Scenario: Your university wants to track student clubs and organizations with the following requirements:

System Requirements:

- Student clubs and organizations information
- Club membership (students can join multiple clubs, clubs have multiple members)
- Club events and student attendance tracking
- Club officer positions (president, treasurer, secretary, etc.)
- Faculty advisors for clubs (each club has one advisor, faculty can advise multiple clubs)
- Room reservations for club events
- Club budget and expense tracking

Your Tasks: 1. Create a complete ER diagram for this system 2. Convert your ER diagram to a normalized relational schema 3. Identify at least one design decision where you had multiple valid options and explain your choice 4. Write 3 example queries that your database should support (in English, not SQL)

Example Query Format: - "Find all students who are officers in the Computer Science Club" - "List all events scheduled for next week with their room reservations"



System Description

This database stores university clubs, their members, officers, events, room reservations, budgets, and expenses. Each **Club** has exactly one **Faculty** advisor (mandatory). **Students** may join multiple clubs and each club has many members; this M:N relationship is modeled by **Membership**. Officer roles are tracked in **Club_Officer**, which references **Membership** (composite FK `(club_id, student_id)`) so that only members can hold officer positions. Officer titles come from the lookup table **Officer_Position**.

Clubs organize **Events**. Student attendance at events is recorded in **Attendance** (M:N between Event and Student). Events can reserve rooms via **Reservation**; a reservation links an **Event** to a **Room** with a time window, allowing multiple rooms per event and enforcing unique time slots.

*Financial tracking uses **Budget_Period** (a weak entity identified by `(club_id, fiscal_year)`) and **Expense**. Each expense references its club's budget period via a composite foreign key, enabling year-by-year budgeting.*

Key Entities

- **Student**(student_id, full_name, email, major, phone)
- **Faculty**(faculty_id, full_name, email, department)
- **Club**(club_id, club_name, description, founded_year, advisor_id→Faculty)
- **Membership**(club_id, student_id, joined_at, leave_date, status)
- **Officer_Position**(position_id, position_name)

- **Club_Officer**(club_id, student_id, start_date, end_date, position_id→Officer_Position; FK `(club_id, student_id)`→Membership)
- **Event**(event_id, club_id→Club, title, description, start_ts, end_ts)
- **Attendance**(event_id, student_id, status, checkin_ts)
- **Room**(room_id, building, room_number, capacity)
- **Reservation**(reservation_id, event_id→Event, room_id→Room, start_ts, end_ts, status)
- **Budget_Period**(club_id, fiscal_year, allocated_amount, approved_date) *(weak)*
- **Expense**(expense_id, club_id, fiscal_year, expense_date, amount, category, description, approved_by)

Integrity Rules (high level)

- Each club **must** have one advisor: `Club.advisor_id` (NOT NULL) → `Faculty.faculty_id`.
- An officer must be a member of that club: `Club_Officer (club_id, student_id)` → `Membership (club_id, student_id)`.
- Unique room identity: **Room** has `UQ(building, room_number)`.
- Unique reservation slot per event/room/time: **Reservation** has `UQ(event_id, room_id, start_ts, end_ts)`.
- Expenses attach to a specific budget year: **Expense (club_id, fiscal_year)** → **Budget_Period (club_id, fiscal_year)**.
- Time sanity: `end_ts > start_ts` for **Event** and **Reservation** (business rule).

Normalization Summary

The schema is in **3NF**. M:N relationships are separated into bridge tables (**Membership**, **Attendance**, **Reservation**). Reference data (Faculty, Rooms, Officer_Position) is stored once and referenced by FKs. Transitive dependencies are removed (e.g., officer titles in a lookup; budget periods separated from expenses). The weak entity **Budget_Period** is identified by the owner's key `(club_id)` plus `fiscal_year`.

Design Decision (example)

We link **Club_Officer** to **Membership** (not directly to Club/Student). This guarantees that only current members can be officers and supports role history via `(club_id, student_id, start_date)`.

Example Queries

1. "List students who joined any club within the last 14 days (recent members)."
2. "For the Debate Club this month, list each event with its attendees and their check-in timestamps."
3. "Find faculty advisors who advise more than three clubs (advisor workload)."