Лекция БД 3

Лекция 3: Реляционная модель данных

От чертежа архитектора к плану инженера

План на сегодня

1. **Вспомнить всё:** Где мы остановились?
2. **Новый язык:** Терминология реляционной модели (таблицы, строки, ключи).
3. **Главный инструмент:** Первичные и внешние ключи.
4. **Пошаговый алгоритм:** 7 правил трансформации ER-диаграммы в реляционную схему.
5. **Кейс:** Превращаем ER-диаграмму коворкинга в набор таблиц.
6. **Что дальше?** Зачем нужна нормализация.

---

1. Где мы остановились?

На прошлой лекции мы переводили хаос бизнес-требований в строгую и понятную **концептуальную модель** т.е. **ER-диаграмму**.

Задача на сегодня: Взять этот чертеж и превратить его в **логический план**, который поймет любая реляционная СУБД (PostgreSQL, MySQL и др.).

> **Концептуальная модель (ERD) ➡️ Логическая модель (Реляционная схема)**

---

2. Язык реляционной модели

Реляционная модель, предложенная Эдгаром Коддом, основана на простой и мощной идее: все данные представлены в виде таблиц.

Формальный термин	Аналог в мире СУБД	Описание
Отношение (Relation)	**Таблица (Table)**	Двумерная структура из строк и столбцов
Кортеж (Tuple)	**Строка (Row/Record)**	Один экземпляр данных (один студент, один заказ)
Атрибут (Attribute)	**Столбец (Column/Field)**	Характеристика данных (Имя, Дата рождения)
Домен (Domain)	**Тип данных (Data Type)**	Множество допустимых значений (TEXT, INT, DATE)

3. Ключи: "Скелет" и "клей" нашей схемы

Ключи - это не просто атрибуты. Это фундаментальные ограничения, которые обеспечивают целостность данных.

Первичный ключ (Primary Key, PK)

- **Что это?** Один или несколько атрибутов, которые **однозначно идентифицируют каждую строку** в таблице.
- **Правила:**
  1. Не может содержать `NULL` (пустых значений).
  2. Должен быть уникальным в пределах таблицы.
- **Пример:** `StudentID` в таблице `Students`.

Внешний ключ (Foreign Key, FK)

- **Что это?** Атрибут (или группа атрибутов) в одной таблице, который ссылается на **первичный ключ** в другой таблице.
- **Задача:** Установить и поддерживать связь между таблицами. Это и есть наш "клей".

- **Пример:** `DepartmentID` в таблице `Employees` ссылается на `DepartmentID` в таблице `Departments`.

---

4. Алгоритм трансформации

Это почти механический процесс. Следуйте правилам, и вы получите корректную схему.

Правило 1: Сильные сущности

Каждая сильная сущность на ER-диаграмме превращается в отдельную таблицу. Атрибуты сущности становятся столбцами этой таблицы.

Students(<u>StudentID</u>, Name, Semester)
Lectures(<u>LecID</u>, Title, CP)

Правило 2: Связь Один-ко-Многим (1:N)

Ключ со стороны "1" **мигрирует** в таблицу на стороне "N" в качестве внешнего ключа.

```
// Таблица на стороне "1"
Departments(DepartmentID, Name)

// Таблица на стороне "N"
// DepartmentID "мигрировал" сюда как внешний ключ
Employees(EmployeeID, FIO, DepartmentID)
```

Внешний ключ `Employees.DepartmentID` ссылается на `Departments.DepartmentID`.

Правило 3: Связь Многие-ко-Многим (M:N)

Такие связи нельзя реализовать напрямую. Мы создаем **третью, ассоциативную (связующую) таблицу**.

- Первичный ключ этой новой таблицы получается как **композиция** первичных ключей из двух исходных таблиц.
- Если у самой связи были атрибуты (например, "оценка" в связи "студент-изучает-курс"), они становятся столбцами в этой новой таблице.

```
Students(StudentID, FIO)
Courses(CourseID, Title)

// Новая ассоциативная таблица
Enrollments(StudentID, CourseID, Grade)
```

Первичный ключ таблицы `Enrollments` *(*`StudentID`*,* `CourseID`*).*
`Enrollments.StudentID` *ссылается на* `Students.StudentID`*.*
`Enrollments.CourseID` *ссылается на* `Courses.CourseID`*.*

Правило 4: Связь Один-к-Одному (1:1)

Есть два основных подхода:

1. **Слияние:** Если связь обязательна с обеих сторон (каждый сотрудник *обязан* иметь паспорт), можно объединить сущности в одну таблицу.
2. **Миграция ключа (как в 1:N):** Более гибкий вариант. PK одной таблицы становится FK в другой. Чтобы сохранить ограничение 1:1, на столбец с внешним ключом нужно добавить ограничение `UNIQUE`.

Правило 5: Слабые сущности

Слабая сущность также становится таблицей, но ее первичный ключ **всегда составной**.

- Он включает в себя **первичный ключ родительской (сильной) сущности** и **частичный ключ** самой слабой сущности.

```
Buildings(BuildingID, Address)

// PK родителя (BuildingID) становится частью PK дочерней таблицы
Rooms(BuildingID, RoomNumber, Capacity)
```

Первичный ключ таблицы `Rooms` *(*`BuildingID`*,* `RoomNumber`*).*
`Rooms.BuildingID` *также является внешним ключом к* `Buildings.BuildingID`*.*

Правило 6: Иерархии ISA (Наследование)

Есть три стратегии. Выбор зависит от специфики задачи (что важнее: скорость запросов, отсутствие `NULL` или целостность).

1. **Одна большая таблица:** Создать одну таблицу для родительской сущности, включив в нее атрибуты всех дочерних. Добавить столбец `EntityType` (`'Student'`, `'Professor'`).
   - **Плюс:** Нет `JOIN`'ов. Быстро.
   - **Минус:** Много `NULL`-значений.
2. **Таблица на каждый "лист":** Создать таблицы только для конечных дочерних сущностей (`Students`, `Assistants`). Общие атрибуты родителя дублируются в каждой.
   - **Плюс:** Нет `NULL`'ов.
   - **Минус:** Дублирование схемы. Сложно запросить всех "людей" сразу.
3. **Таблица на каждую сущность (самый "правильный"):** Создать таблицу для родителя и для каждого потомка. Дочерние таблицы содержат только свои уникальные атрибуты и внешний ключ, ссылающийся на PK родителя.
   - **Плюс:** Максимальная нормализация, нет дублирования и `NULL`'ов.
   - **Минус:** Требуются `JOIN`'ы для получения полной информации.

---

5. Кейс: Трансформация ER-схемы коворкинга

Задача: Спроектировать ER-диаграмму для системы бронирования коворкинга.

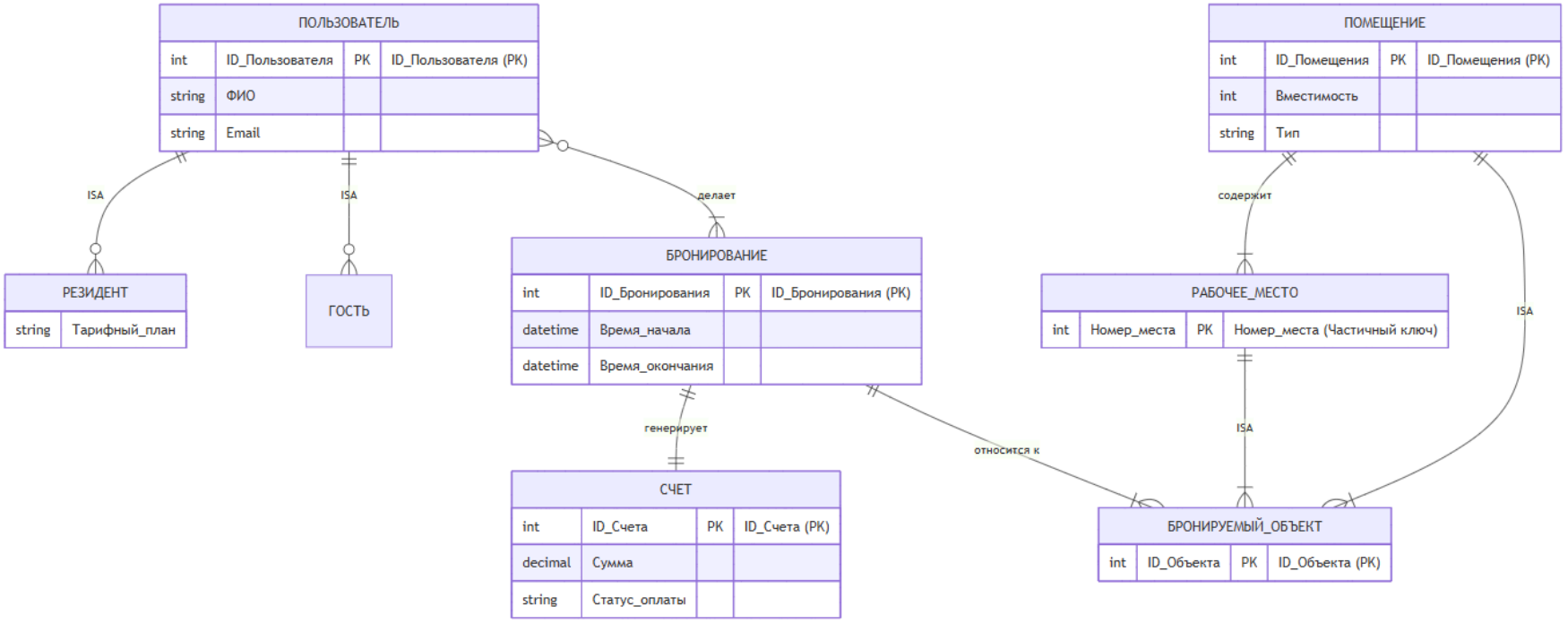
Требования:

1. Есть `Пользователи`. У всех есть ФИО и email. Пользователи делятся на `Резидентов` (с тарифным планом) и `Гостей`.
2. Есть `Помещения` для бронирования (переговорки, кабинеты). У них есть ID и вместимость.
3. В некоторых `Помещениях` есть `Рабочие_места` с номером, уникальным только в пределах этого помещения.
4. Пользователи делают `Бронирования`, указывая время. Бронировать можно как `Помещение` целиком, так и отдельное `Рабочее_место`.
5. Каждое бронирование генерирует `Счет` на оплату.

Процесс мышления и результат:

1. **Выделяем сущности (существительные):** `Пользователь`, `Резидент`, `Гость`, `Помещение`, `Рабочее_место`, `Бронирование`, `Счет`.
2. **Применяем ISA:** `Резидент` и `Гость` — это виды `Пользователя`. Создаем иерархию.
3. **Ищем слабую сущность:** `Рабочее_место` не может существовать без `Помещения`, и его номер уникален только внутри него. Это классический кандидат в слабую сущность.
4. **Определяем связи и кардинальность:**
   - `Пользователь` делает много `Бронирований` (1:N).
   - `Бронирование` относится к одному `Помещению` ИЛИ к одному `Рабочему_месту` (это эксклюзивная связь, сложный случай).
   - `Бронирование` порождает один `Счет` (1:1).

- `Помещение` содержит много `Рабочих_мест` (идентифицирующая связь 1:N).



Итоговая реляционная схема:

```
// Правило 6 (ISA): Таблица на каждую сущность
Users(UserID, FIO, Email)

Residents(UserID, TariffPlanID) // PK и FK одновременно
Guests(UserID, VisitDate)       // PK и FK одновременно
```

```
// Правило 1: Сильная сущность
Premises(PremiseID, Name, Capacity)

// Правило 5: Слабая сущность
Workplaces(PremiseID, WorkplaceNumber, Description) // PK = (PremiseID, WorkplaceNumber)

// Правило 1 + Правило 2
Bookings(BookingID, StartTime, EndTime, UserID, PremiseID, WorkplaceNumber)

// Правило 2: Связь 1:N
Invoices(InvoiceID, Amount, Status, BookingID)```
*Здесь также прописываются все внешние ключи, например, `Bookings.UserID` ->
`Users.UserID`.*
```

---

6. Что дальше?

Мы получили формально корректный набор таблиц. Но есть проблема. Наша схема может содержать **избыточность** и быть подвержена **аномалиям обновления, удаления и вставки**.

Пример аномалии:
В таблице `(StudentID, StudentFIO, CourseID, CourseTitle)` ФИО студента будет повторяться

для каждого курса, на который он записан. Если он сменит фамилию, нам придется обновить **много** строк, и мы рискуем сделать это не везде.

Решение? Процесс **нормализации**, который мы изучим на следующей лекции. Он позволяет привести схему к виду, минимизирующему избыточность.