Тема 10. ПРОЕКТИРОВАНИЕ БД, СВЯЗИ, НФ. SELECT-ЗАПРОСЫ, ВЫБОРКИ ИЗ ОДНОЙ ТАБЛИЦЫ.

Цель занятия:

Ознакомиться с концепциями работы с таблицами в базах данных, с видами связей между таблицами, понятиями первичных и внешних ключей.

Учебные вопросы:

1. Этапы проектирования базы данных.
2. ER-диаграммы.
3. Первичный ключ
4. Связи между отношениями
5. Нормализация данных. Нормальные формы.
6. DML-запросы. SELECT-запросы, выборки из одной таблицы.

2. Этапы проектирования базы данных.

Этапы проектирования базы данных:

1. Анализ требований
2. Проектирование концептуальной модели
3. Проектирование логической модели
4. Физическое проектирование

1. Анализ требований:

Цель: понять бизнес-процессы, для которых создается база данных, и определить, какие данные необходимо хранить и обрабатывать.

Действия:

- Общение с заказчиками и пользователями для сбора информации о том, какие задачи должна решать система.
- Определение объема данных, частоты их обновления и основных операций (чтение, запись, обновление).
- Формулирование целей системы и ключевых требований, например, объем данных, скорость обработки запросов, безопасность и масштабируемость.

Результат: четкое понимание, как база данных будет поддерживать бизнес-процессы, какие данные будут обрабатываться и какие требования к ней предъявляются.

2. Проектирование концептуальной модели:

Цель: создание наглядной модели данных, которая отражает ключевые сущности и их связи без учета специфики СУБД.

Действия:

- Определение сущностей (например, "Клиент", "Заказ", "Продукт"), атрибутов (свойства сущностей, такие как имя клиента, дата заказа) и связей между сущностями.
- Создание ER-диаграммы для отображения всех сущностей, их атрибутов и связей между ними.
- Определение типов связей (один к одному, один ко многим, многие ко многим).

Результат: ER-диаграмма, которая является концептуальной моделью базы данных, предоставляющей общее представление о структуре данных.

3. Проектирование логической модели:

Цель: преобразование концептуальной модели в логическую модель, которая уже учитывает требования нормализации и особенности реляционных баз данных.

Действия:

- Преобразование сущностей из ER-диаграммы в таблицы, где каждая сущность становится таблицей, а атрибуты — полями таблицы.
- Применение нормализации для устранения избыточности данных (приведение к 1NF, 2NF, 3NF, при необходимости — BCNF).
- Определение первичных и внешних ключей для таблиц.

Результат: логическая модель базы данных, включающая структуры таблиц, ключи и связи между таблицами.

4. Физическое проектирование:

Цель: реализация логической модели на уровне конкретной СУБД с учетом производительности, безопасности и будущих изменений.

Действия:

- Выбор конкретной СУБД (например, MySQL, PostgreSQL, SQL Server), учитывая требования системы (масштабируемость, стоимость, доступные функции).
- Проектирование таблиц, включая выбор типов данных для каждого поля, с учетом их объема и типа информации (например, INT для чисел, VARCHAR для текста).
- Создание индексов для ускорения поиска и выполнения сложных запросов.
- Определение стратегии управления транзакциями и обеспечения безопасности данных.

Результат: готовая к реализации физическая структура базы данных, которая оптимизирована для выбранной СУБД и учитывает требования по производительности и надежности.

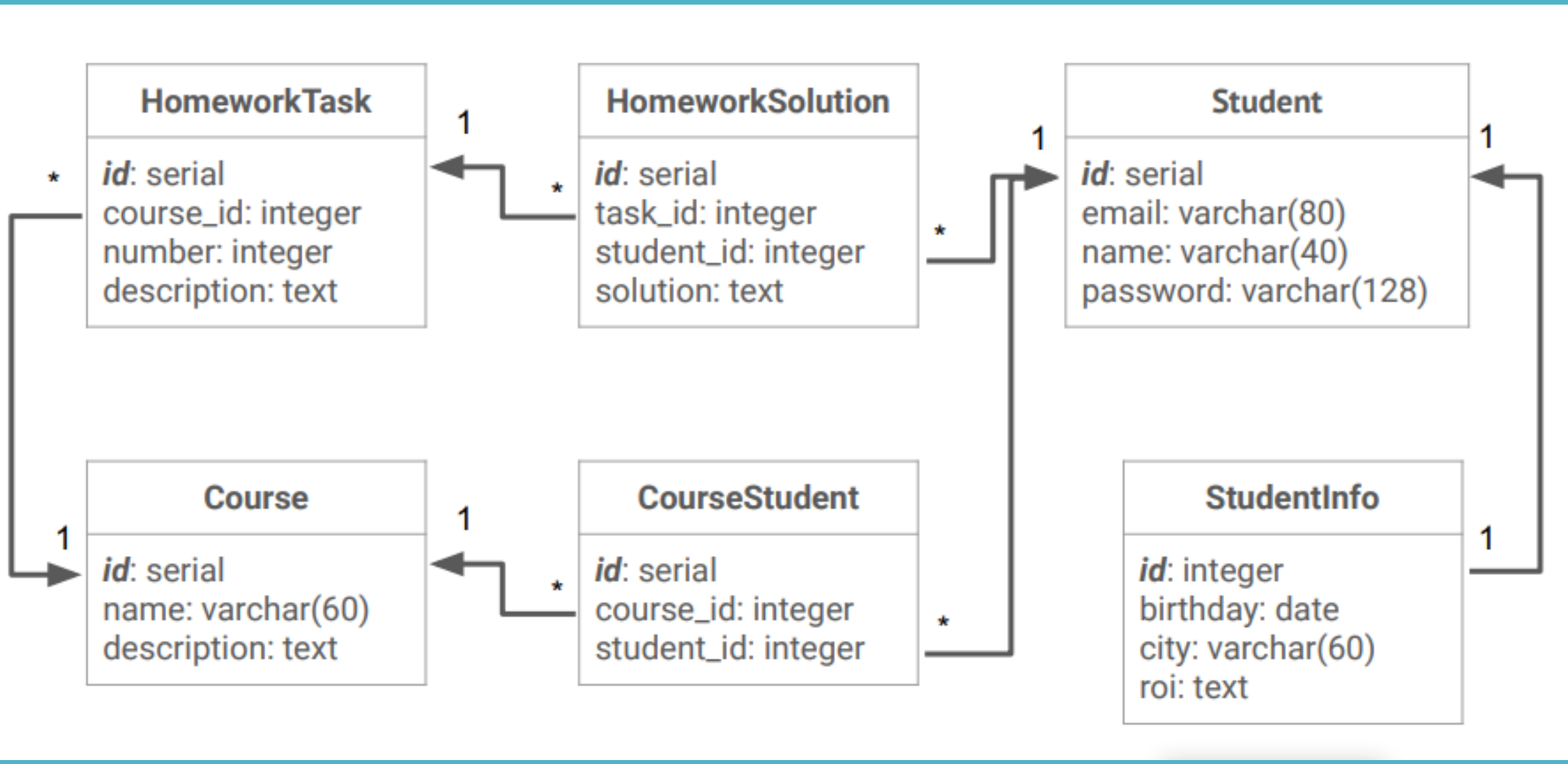
3. ER-диаграммы.

ER-диаграмма (Entity-Relationship диаграмма) — это графическая модель, которая используется для представления данных и взаимосвязей между ними.

Она помогает проектировщикам баз данных наглядно представить структуру данных, их взаимосвязь и основные атрибуты сущностей.

Описывает структуру данных в базе данных, фокусируясь на сущностях (таблицах), их атрибутах (полях) и связях между ними.

Пример. ER-диаграмма.



Основные элементы ER-диаграммы:

Сущности (таблицы): Объекты, для которых необходимо хранить данные. Сущности обычно представлены прямоугольниками. Примеры сущностей: "Студент", "Курс", "Преподаватель".

Атрибуты (поля): Свойства или характеристики сущности, которые содержат данные. Атрибуты находятся внутри прямоугольника класса. Примеры атрибутов для сущности "Студент": "Имя", "Дата рождения", "Номер студенческого билета".

Связи (отношения): Отражают взаимодействие или ассоциации между сущностями. Связи изображаются линиями, которые соединяют сущности. Например, связь между сущностями "Клиент" и "Заказ" может означать, что один клиент может разместить несколько заказов.

Популярные инструменты для построения ER-диаграмм:

- **MySQL Workbench**: Интегрированная среда разработки для MySQL, которая поддерживает создание ER-диаграмм и автоматическое преобразование их в схемы базы данных. Поддерживает реверс-инжиниринг для анализа существующих баз данных.
- **Microsoft Visio**: Мощный инструмент для создания диаграмм, включая ER-диаграммы. Предлагает шаблоны для моделирования баз данных с возможностью настройки сущностей, атрибутов и связей.
- **Lucidchart**: Онлайн-инструмент для создания различных диаграмм, включая ER-диаграммы. Поддерживает совместную работу в реальном времени, шаблоны и удобный интерфейс для моделирования баз данных.
- **Draw.io** (diagrams.net): Бесплатный онлайн-инструмент для создания диаграмм, включая ER-диаграммы. Поддерживает интеграцию с Google Drive, OneDrive и другими облачными сервисами для хранения проектов.
- **dbdiagram.io**: Простое и удобное онлайн-решение для создания ER-диаграмм с поддержкой синтаксиса для описания таблиц и связей. Предлагает экспорт моделей в различные форматы, такие как SQL и PDF.
- **и другие**.

4. Первичный ключ.

Первичный ключ — это отдельное поле или комбинация полей, которые однозначно определяют запись — кортеж.

Зачем лишний атрибут?

name	gpa
Егор	4.82
Егор	4.11
Егор	3.88

Как отличить одного Егора от другого?

Primary key (первичный ключ) на схемах-таблицах обозначается с помощью подчеркивания. На схеме ниже атрибут **id** — это первичный ключ.

id	**name**	**gpa**
1	Егор	4.25
2	Егор	3.82
3	Егор	4.25

5. Связи между отношениями.

В реляционных базах данных связи между таблицами реализуются через внешние ключи (FOREIGN KEY).

В зависимости от логики данных, связи могут быть следующих типов:

Один-к-одному (1:1)

Один-ко-многим (1:M)

Многие-ко-многим (M:N)

Типы связей:

- **Один к одному**. Каждая сущность A может быть связана только с одной сущностью B, и наоборот. Пример: каждый человек имеет один паспорт, и каждый паспорт принадлежит только одному человеку.
- **Один ко многим.** Одна сущность A может быть связана с несколькими сущностями B, но каждая сущность B может быть связана только с одной сущностью A. Пример: один преподаватель ведет несколько курсов, но каждый курс преподается только одним преподавателем.
- **Многие ко многим.** Несколько сущностей A могут быть связаны с несколькими сущностями B. Пример: студенты могут посещать несколько курсов, и каждый курс может быть посещён несколькими студентами.

Смоделируем ситуацию:

Есть система онлайн обучения.

В ней есть пользователи — студенты. У каждого пользователя есть почта (она же является логином), пароль и имя.

Также у пользователя есть возможность указать дополнительную информацию: дату рождения, город проживания, свои интересы.

Пользователи могут записываться на курсы.

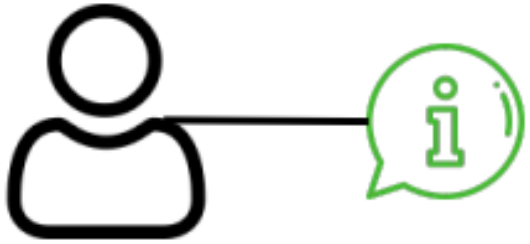
В рамках курса пользователи должны выполнять домашние задания и загружать их в систему.

Связи между отношениями

Типы связей

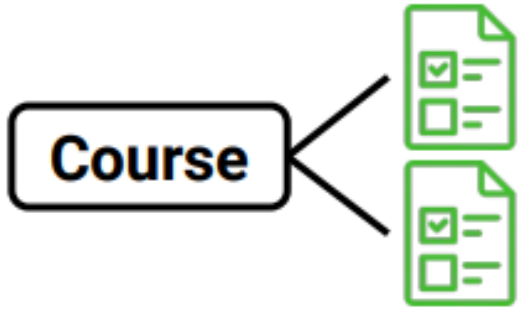
один к одному

пользователь и дополнительная информация о нём



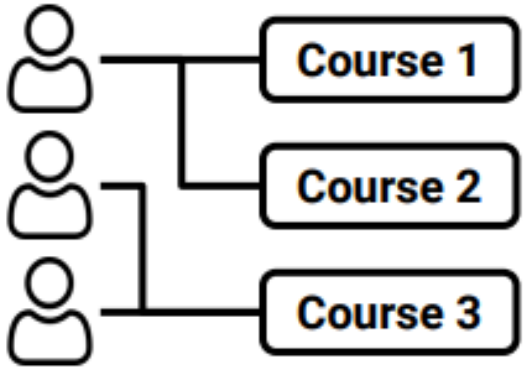
один ко многим

домашние задания на курсе



многие ко многим

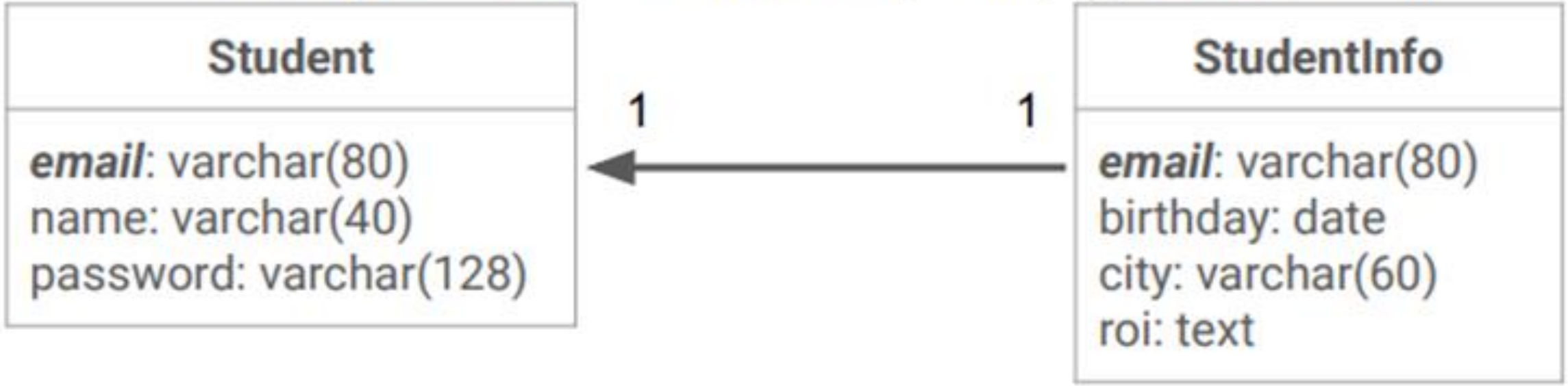
пользователи и курсы



> Связи, как и первичный ключ, можно попросить контролировать СУБД.
> Для этого используется ограничение **foreign key**.

Один к одному. Вариант 1

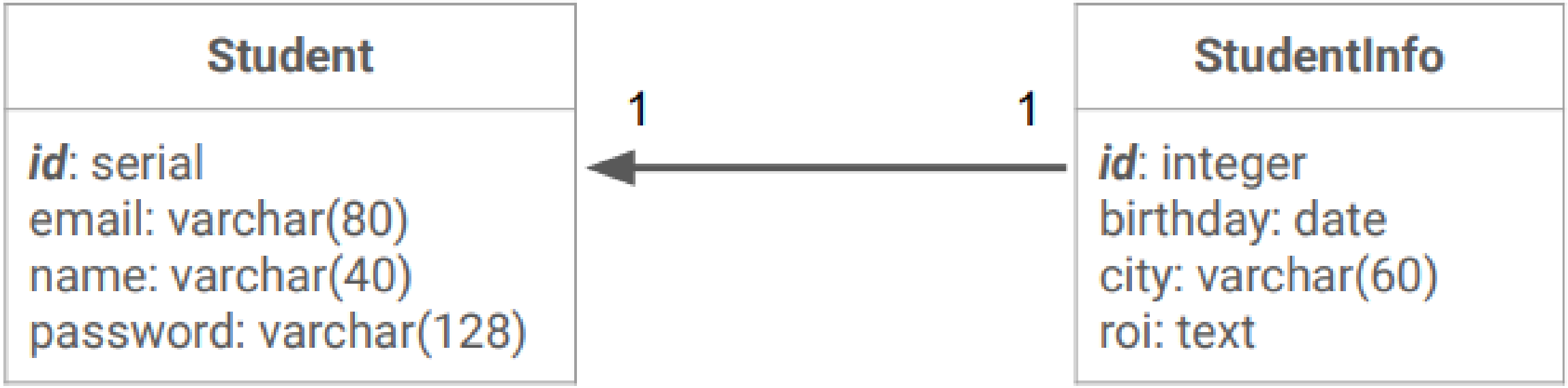
Связываем студента и дополнительную информацию о нём.



```
mysql> CREATE TABLE IF NOT EXISTS StudentInfo (
    ->     email VARCHAR(80) PRIMARY KEY,
    ->     birthday DATE,
    ->     city VARCHAR(60),
    ->     roi TEXT,
    ->     CONSTRAINT fk_student_email FOREIGN KEY (email) REFERENCES Student(email)
    -> );
Query OK, 0 rows affected (0.06 sec)
```

```
mysql> CREATE TABLE IF NOT EXISTS StudentInfo (
    ->     email VARCHAR(80) PRIMARY KEY,
    ->     birthday DATE,
    ->     city VARCHAR(60),
    ->     roi TEXT,
    ->     CONSTRAINT fk_student_email FOREIGN KEY (email) REFERENCES Student(email)
    -> );
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

Один к одному. Вариант 2

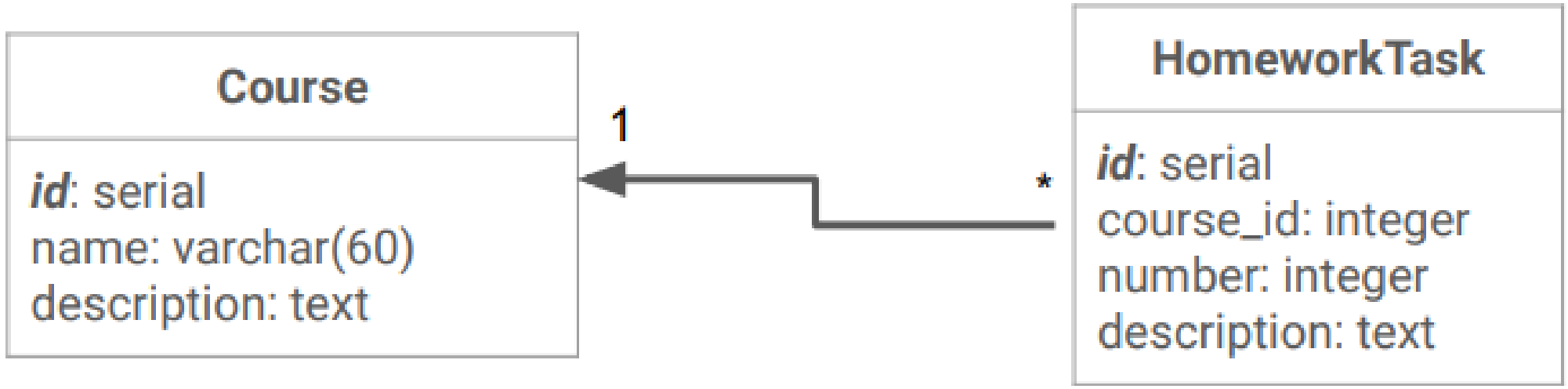


```sql
CREATE TABLE IF NOT EXISTS Student (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(80) UNIQUE NOT NULL,
    name VARCHAR(40) NOT NULL,
    password VARCHAR(128) NOT NULL
);
```

```sql
CREATE TABLE IF NOT EXISTS StudentInfo (
    id INT PRIMARY KEY,
    birthday DATE,
    city VARCHAR(60),
    roi TEXT,
    FOREIGN KEY (id) REFERENCES Student(id) ON DELETE CASCADE
);
```

Один ко многим

Связываем описание домашних заданий с курсами, к которым они относятся.

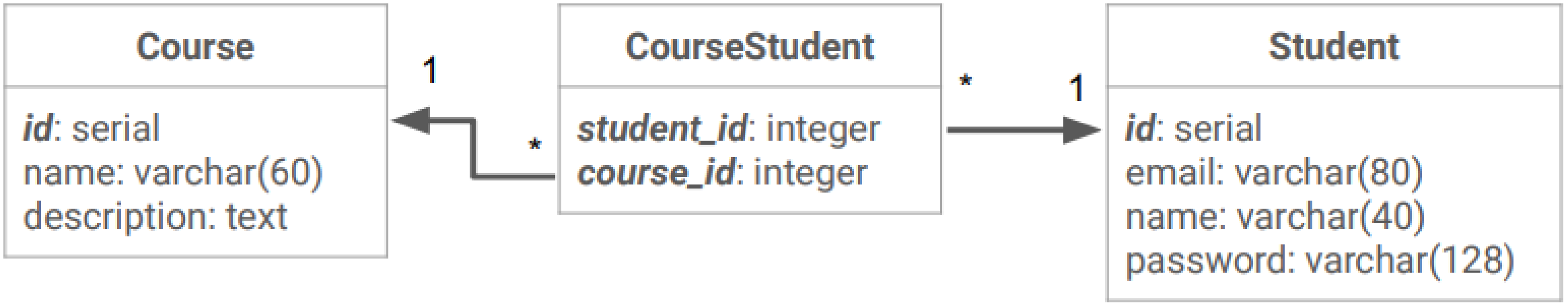


```sql
CREATE TABLE IF NOT EXISTS Course (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(60) NOT NULL,
    description TEXT
);

CREATE TABLE IF NOT EXISTS HomeworkTask (
    id INT AUTO_INCREMENT PRIMARY KEY,
    course_id INT NOT NULL,
    number INT NOT NULL,
    description TEXT NOT NULL,
    FOREIGN KEY (course_id) REFERENCES Course(id) ON DELETE CASCADE
);
```

Многие ко многим. Вариант 1

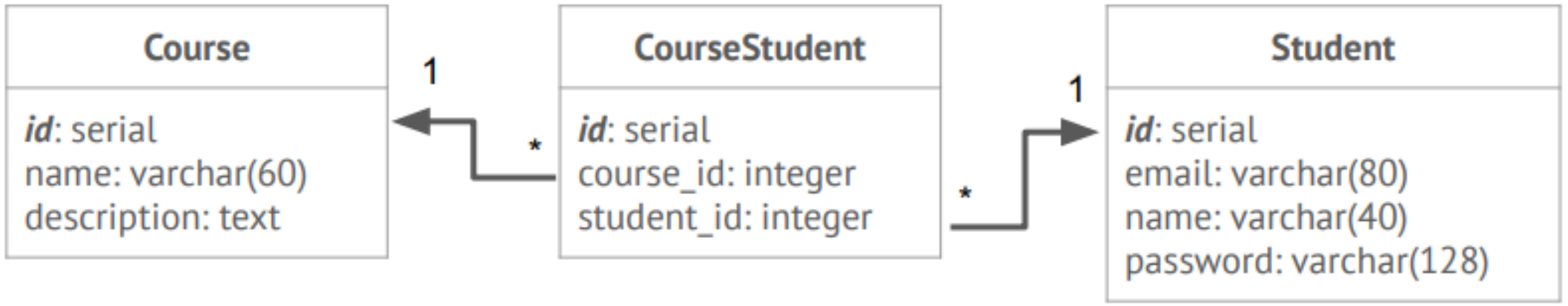
Связываем студентов и курсы, на которые они записаны.



```sql
CREATE TABLE IF NOT EXISTS CourseStudent (
    course_id INT NOT NULL,
    student_id INT NOT NULL,
    PRIMARY KEY (course_id, student_id),
    FOREIGN KEY (course_id) REFERENCES Course(id) ON DELETE CASCADE,
    FOREIGN KEY (student_id) REFERENCES Student(id) ON DELETE CASCADE
);
```

student_id	course_id
1 (Вова)	1 (Python)
1 (Вова)	2 (Java)
2 (Дима)	1 (Python)

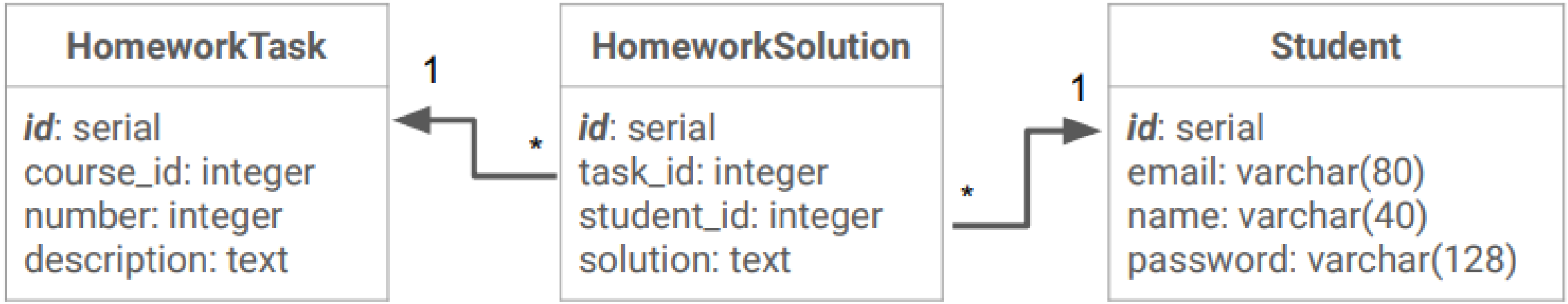
Многие ко многим. Вариант 2




```
CREATE TABLE IF NOT EXISTS CourseStudent (
    id INT AUTO_INCREMENT PRIMARY KEY,
    course_id INT NOT NULL,
    student_id INT NOT NULL,
    FOREIGN KEY (course_id) REFERENCES Course(id) ON DELETE CASCADE,
    FOREIGN KEY (student_id) REFERENCES Student(id) ON DELETE CASCADE
);
```

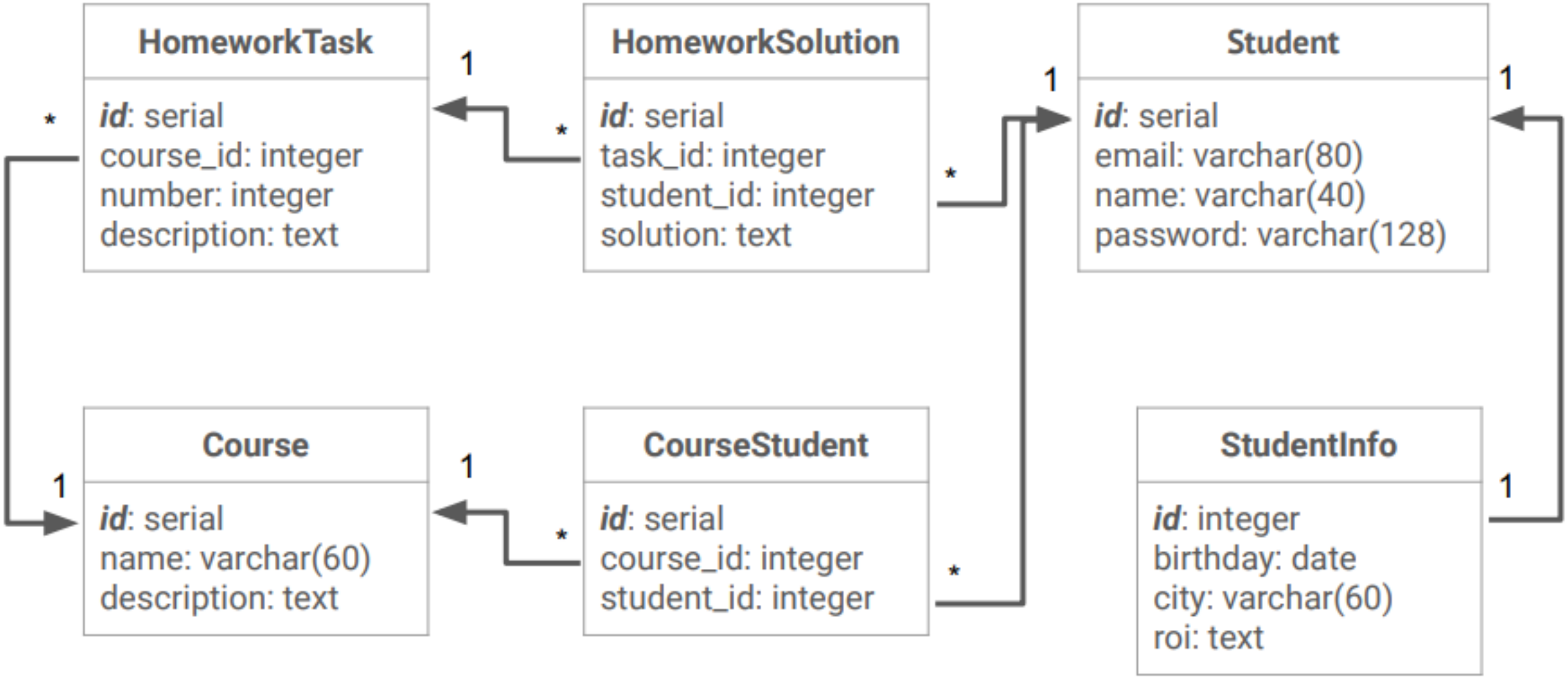
Многие ко многим

Связываем студента и домашние работы, которые он отправил в систему.



```
CREATE TABLE IF NOT EXISTS HomeworkSolution (
    id INT AUTO_INCREMENT PRIMARY KEY,
    task_id INT NOT NULL,
    student_id INT NOT NULL,
    solution TEXT NOT NULL,
    FOREIGN KEY (task_id) REFERENCES HomeworkTask(id) ON DELETE CASCADE,
    FOREIGN KEY (student_id) REFERENCES Student(id) ON DELETE CASCADE
);
```

Итоговая схема



5. Нормализация данных.

Нормализация — это процесс организации структуры базы данных, направленный на минимизацию избыточности данных и предотвращение аномалий при обновлении, удалении или добавлении данных.

Нормализация разбивает большие, дублирующиеся таблицы на более маленькие связанные таблицы, что делает базу данных более логичной и управляемой.

Роль нормализации: Нормализация играет ключевую роль в проектировании базы данных, обеспечивая, что структура данных будет соответствовать принципам правильного хранения и организации. Она помогает предотвратить избыточное хранение данных и ошибки при обновлении информации, делая базы данных гибкими и согласованными.

Цели нормализации:

- **Устранение избыточности данных.** Избыточные данные занимают лишнее место и могут привести к несогласованным записям. Нормализация помогает избежать дублирования, распределяя данные по нескольким связанным таблицам. Например, вместо хранения информации о клиенте в каждой записи о заказе, создается отдельная таблица клиентов, на которую ссылаются заказы.
- **Обеспечение целостности данных.** Нормализация позволяет поддерживать целостность данных, то есть гарантирует, что данные всегда будут корректными и непротиворечивыми. Это достигается за счет четкой структуры и разделения данных по смысловым категориям. Например, если номер клиента изменяется, достаточно обновить информацию в одной таблице, а не во всех связанных таблицах.
- **Улучшение производительности запросов.** Хотя нормализация изначально направлена на структурную целостность, в некоторых случаях она также может способствовать улучшению производительности запросов, особенно при работе с обновлениями и удалением данных. Меньшие таблицы с четкими связями позволяют быстрее выполнять выборки данных и избегать лишних операций.

Пример избыточных и несогласованных данных:

Customers			
id	**email**	**name**	**city**
1	ivanoff@bk.ru	Иванов	Алматы
2	sidorov@gmail.com	Сидоров	Алма-Ата
3	dtepanof@ya.ru	Степанова	Алмата
4	egorova@list.ru	Егорова	Almaty
5	petrovv@gmail.com	Петров	Астана
6	zuev@bk.com	Зуев	Актау

Пример без избыточных и несогласованных данных:

Customers			
id	**email**	**name**	**city_id**
1	ivanoff@bk.ru	Иванов	1
2	sidorov@gmail.com	Сидоров	1
3	dtepanof@ya.ru	Степанова	1
4	egorova@list.ru	Егорова	1
5	petrovv@gmail.com	Петров	2
6	zuev@bk.com	Зуев	3

Cities	
id	**title**
1	Алматы
2	Астана
3	Актау

Нормальные формы.

1. Первая нормальная форма (1NF) — Гарантирует, что все данные в таблице атомарны (неделимы).
2. Вторая нормальная форма (2NF) — Устраняет частичные зависимости от первичного ключа, обеспечивая, что каждый неключевой атрибут полностью зависит от всего первичного ключа.
3. Третья нормальная форма (3NF) — Устраняет транзитивные зависимости, обеспечивая, что все неключевые атрибуты зависят только от первичного ключа и не зависят друг от друга.
4. Бойс-Кодд нормальная форма (BCNF) — Дополняет 3NF, устраняя случаи, когда определение функциональных зависимостей нарушает целостность данных, хотя она уже находится в 3NF.
5. Четвертая нормальная форма (4NF) — Устраняет многозначные зависимости, обеспечивая, что каждая независимая многозначная зависимость должна находиться в отдельной таблице.
6. Пятая нормальная форма (5NF) — Устраняет случаи, когда таблица может быть разделена на несколько таблиц, не теряя информацию о связях между данными, и предотвращает проблемы с проекцией данных.
7. Шестая нормальная форма (6NF) — Используется для обработки временных зависимостей и изменения данных, которые могут быть разделены по времени.

На практике чаще всего рассматривают первые три нормальные формы. Почему?

- **Практическая применимость.** 1NF, 2NF и 3NF покрывают большинство случаев в проектировании баз данных и решают основные проблемы избыточности и целостности данных, что делает их достаточными для большинства бизнес-приложений.
- **Сложность и избыточность.** BCNF и 4NF и последующие нормальные формы предназначены для более специализированных случаев и могут вводить дополнительную сложность, которая не всегда оправдана в реальных приложениях. Например, 4NF и 5NF часто применяются в теоретических исследованиях или очень сложных системах, где многозначные зависимости и проектирование данных играют ключевую роль.

Первая нормальная форма (1NF).

Определение: Таблица находится в первой нормальной форме, если все её поля содержат только **атомарные** (неделимые) значения и каждая ячейка содержит только одно значение.

Требования:

- Каждое поле таблицы должно содержать только **одно значение** (например, не должно быть списков или наборов значений в одной ячейке).
- Каждая строка таблицы должна быть **уникальной**, что достигается использованием первичного ключа.

Пример. Таблица, содержащая информацию о студентах и их курсах:

Students_Courses		
id	**name**	**courses**
1	Иванов	Python, Java
2	Сидоров	Python, CSharp

Students_Courses		
id	**name**	**courses**
1	Иванов	Python
2	Иванов	Java
3	Сидоров	Python
4	Сидоров	CSharp

Вторая нормальная форма (2NF).

Определение: Таблица находится во второй нормальной форме, если она уже находится в 1NF и все неключевые атрибуты полностью функционально зависят от всего первичного ключа.

Требования:

- Таблица должна быть в 1NF.
- Все атрибуты, не входящие в первичный ключ, должны **зависеть** от всего первичного ключа, а не от его части.

Пример. Таблица с данными о студентах и их факультетах:

Students_faculties			
Student_ID	StudentName	faculty	Dean_of_th_Faculty
1	Иванов	программирование	ДТН Егорова
2	Сидоров	кибербезопасность.	КТН Петров
3	Степанова	робототехника	КТН Зуев

Поле **Dean_of_th_Faculty** (декан факультета) зависит только от **Faculty** (Факультет), а не от первичного ключа. Это означает, что знание **Faculty** позволяет определить **Dean_of_th_Faculty**, независимо от значения StudentID.

Преобразование в 2NF. Для приведения таблицы в 2NF нужно устранить частичные зависимости и разделить данные на несколько таблиц:

Students_faculties		
Student_ID	StudentName	faculty
1	Иванов	программирование
2	Сидоров	кибербезопасность.
3	Степанова	робототехника

faculties	
faculty	Dean_of_th_Faculty
программирова	ДТН Егорова
кибербезопасно	КТН Петров
робототехника	КТН Зуев

Пример. Таблица с данными о заказах:

Orders			
OrderID	**ProductID**	**ProductName**	**Quantity**
1	101	Widget	10
1	102	Gadget	5
2	101	Widget	20

В данной таблице первичный ключ составной, состоящий из двух полей: OrderID и ProductID.

Поле ProductName зависит только от ProductID, а не от всего первичного ключа (OrderID и ProductID).

Это означает, что знание ProductID позволяет определить ProductName, независимо от значения OrderID.

Из-за этой частичной зависимости данные о продукте (например, ProductName) повторяются для каждого заказа, в котором этот продукт появляется.

Если название продукта изменится, его нужно будет обновить в каждой строке таблицы, что может привести к ошибкам и несогласованности.

Чтобы привести таблицу в 2NF, нужно устранить частичные зависимости и разделить данные на несколько таблиц:

Orders		
OrderID	ProductID	Quantity
1	101	10
1	102	5
2	101	20

Products	
ProductID	ProductName
101	Widget
102	Gadget

Третья нормальная форма (3NF).

Определение: Таблица находится в третьей нормальной форме, если она уже находится во 2NF и все неключевые атрибуты не зависят транзитивно от первичного ключа.

Требования:

- Таблица должна быть в 2NF.
- Никакой неключевой атрибут не должен зависеть от другого неключевого атрибута.

Предположим, у нас есть таблица с информацией о сотрудниках и их проектах:

Projects				
EmployeeID	EmployeeName	ProjectID	ProjectName	ProjectManager
1	John Doe	101	Alpha	Sarah Lee
1	John Doe	102	Beta	Mike Brown
2	Jane Smith	101	Alpha	Sarah Lee
3	Alice Johnson	103	Gamma	Emma Davis

Нарушение 3NF: Третья нормальная форма требует, чтобы все неключевые атрибуты зависели только от первичного ключа и не имели транзитивных зависимостей.

В этом примере, ProjectManager транзитивно зависит от EmployeeID, через ProjectID и ProjectName.

Чтобы привести таблицу в 3NF, нужно устранить транзитивные зависимости и разделить данные на несколько таблиц:

EmployeeProjects	
EmployeeID	**ProjectID**
1	101
1	102
2	101
3	103

Projects		
ProjectID	**ProjectName**	**ProjectManager**
101	Alpha	Sarah Lee
102	Beta	Mike Brown
103	Gamma	Emma Davis

Пошаговое преобразование базы данных от 1NF до 3NF на конкретном примере базы данных магазина:

Orders					
OrderID	**CustomerName**	**Product**	**Quantity**	**Price**	**CustomerAddress**
1	Иван Иванов	Товар1	2	200	Адрес1
1	Иван Иванов	Товар2	1	100	Адрес1
2	Петр Петров	Товар1	1	200	Адрес2

Приведение к 2NF. 2NF требует устранения частичных зависимостей. Мы разделяем таблицу на две:

Orders		
OrderID	CustomerName	CustomerAddress
1	Иван Иванов	Адрес1
2	Петр Петров	Адрес2

OrderDetails			
OrderID	Product	Quantity	Price
1	Товар1	2	200
1	Товар2	1	100
2	Товар1	1	200

3NF требует устранения транзитивных зависимостей. Добавляем таблицы для клиентов и продуктов:

Customers		
CustomerID	CustomerName	CustomerAddress
1	Иван Иванов	Адрес1
2	Петр Петров	Адрес2

Products		
ProductID	ProductName	Price
1	Товар1	200
2	Товар2	100

Orders	
OrderID	CustomerID
1	1
2	2

OrderDetails		
OrderID	ProductID	Quantity
1	1	2
1	2	1
2	1	1

6. DML-запросы. SELECT-запросы, выборки из одной таблицы.

DML (Data Manipulation Language) — это подмножество SQL, которое включает команды для работы с данными в базах данных.

В MySQL DML-запросы используются для добавления, обновления, удаления и выборки данных из таблиц.

Далее рассмотрим основные команды DML и примеры их использования.

1. INSERT — Вставка данных в таблицу

Команда INSERT используется для добавления новых строк в таблицу.

Пример:

```
INSERT INTO Employees (FirstName, LastName, Age, Department)
VALUES ('John', 'Doe', 30, 'HR');
```

INSERT INTO Employees: Указывает, что вы добавляете новую запись в таблицу Employees.

(FirstName, LastName, Age, Department): Это список столбцов, в которые будут добавлены значения.

VALUES ('John', 'Doe', 30, 'HR'): Указывает значения, которые будут вставлены в соответствующие столбцы.

2. **UPDATE** — Обновление данных в таблице

Команда UPDATE используется для изменения существующих данных в таблице.

Пример:

```sql
UPDATE Employees
SET Age = 31
WHERE FirstName = 'John' AND LastName = 'Doe';
```

Этот запрос обновляет возраст сотрудника с именем John Doe на 31 год.

3. DELETE — Удаление данных из таблицы

Команда DELETE используется для удаления строк из таблицы.

Пример:

```
DELETE FROM Employees
WHERE LastName = 'Doe';
```

Этот запрос удаляет всех сотрудников с фамилией Doe из таблицы Employees.

4. SELECT — Выборка данных из таблицы

Команда SELECT используется для извлечения данных из таблицы в MySQL. Она позволяет выбирать все или определённые столбцы, фильтровать записи, сортировать и агрегировать данные.

Пример. Простая выборка всех данных Выбирает все записи из таблицы.:

```
SELECT * FROM Students;
```

Выборка определённых столбцов. Извлекает только указанные поля.

Пример:

```
SELECT id, name, email FROM Students;
```

Этот запрос возвращает только id, name и email из Students.

Общий вид структуры SELECT-запроса:

```
SELECT [DISTINCT] столбцы
FROM таблица
[JOIN другая_таблица ON условие_связи]
[WHERE условие_фильтрации]
[GROUP BY столбец_или_столбцы]
[HAVING условие_для_группы]
[ORDER BY столбец_или_столбцы [ASC | DESC]]
[LIMIT количество_строк [OFFSET смещение]];
```

Компоненты запроса:

SELECT: Обязательная часть запроса, указывающая, какие столбцы или выражения нужно выбрать.

Опционально можно использовать ключевое слово DISTINCT, чтобы удалить дубликаты из результата.

FROM: Обязательная часть, указывающая таблицу (или таблицы), из которых нужно выбирать данные.

JOIN: Опциональная часть, позволяющая объединять данные из нескольких таблиц на основе заданного условия.

Виды JOIN: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN.

WHERE: Опциональная часть, задающая условия фильтрации строк, которые должны быть включены в результат.

GROUP BY: Опциональная часть, группирующая строки, имеющие одинаковые значения в указанных столбцах.

Обычно используется с агрегатными функциями (например, COUNT, SUM, AVG).

HAVING: Опциональная часть, задающая условие фильтрации для групп, образованных с помощью GROUP BY.

ORDER BY: Опциональная часть, определяющая порядок сортировки результирующих строк.

Можно указать сортировку по возрастанию (ASC, по умолчанию) или по убыванию (DESC).

LIMIT: Опциональная часть, ограничивающая количество возвращаемых строк.

Параметр OFFSET позволяет пропустить указанное количество строк перед выборкой.

Пример полного SELECT-запроса:

```sql
SELECT DISTINCT FirstName, LastName, COUNT(*) AS NumOrders
FROM Customers
INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID
WHERE Orders.OrderDate >= '2024-01-01'
GROUP BY FirstName, LastName
HAVING COUNT(*) > 5
ORDER BY NumOrders DESC
LIMIT 10 OFFSET 5;
```

Этот запрос выбирает уникальные имена и фамилии клиентов, которые сделали более 5 заказов с 1 января 2024 года, сортирует результаты по количеству заказов в убывающем порядке и возвращает 10 строк, начиная с 6-й строки.

Пример с фильтрацией:

```
SELECT * FROM Students WHERE age > 18;
```

WHERE — условие, ограничивающее выборку.

Арифметические операции в SELECT-запросах.

в SELECT-запросах можно использовать различные арифметические операторы для данных, хранящихся в таблицах:

- + – сложение;
- - – вычитание;
- / – деление;
- * – умножение;
- % – взятие остатка от деления.

Пример:

```
SELECT FirstName, Salary, Salary * 12 AS AnnualSalary
FROM Employees;
```

Этот запрос выбирает имя и зарплату сотрудника, а также вычисляет годовую зарплату, умножая месячную зарплату на 12.

```
SELECT FirstName, Salary, Salary + 1000 AS IncreasedSalary
FROM Employees;
```

Этот запрос выбирает имя и зарплату сотрудника, а также вычисляет зарплату с увеличением на 1000 единиц.

Оператор WHERE и фильтрация по условиям.

Оператор WHERE в SQL используется для фильтрации строк в результате запроса, чтобы вернуть только те записи, которые соответствуют заданным условиям.

Он позволяет ограничить выборку данных на основе различных критериев.

В качестве условий можно использовать:

- сравнения (=, >, <, >=, <=, !=);
- оператор IN;
- оператор BETWEEN;
- оператор LIKE.

С условиями можно применять логические операторы and, or и not:

- Оператор AND отображает запись, если оба операнда истинны;
- Оператор OR отображает запись, если хотя бы один операнд истинен;
- Оператор NOT инвертирует исходный операнд.

Примеры:

```
SELECT * FROM Employees
WHERE Department = 'HR';
```

Этот запрос выбирает все строки из таблицы Employees, где столбец Department равен 'HR’.

```
SELECT * FROM Employees
WHERE Age > 30 AND Department = 'HR';
```

Этот запрос выбирает все строки, где возраст больше 30 и отдел равен 'HR'.

```sql
SELECT title, release_year
FROM film
WHERE release_year >= 2000;
```

Этот запрос выбирает данные из таблицы film, отображая названия и года выпуска фильмов, которые были выпущены в 2000 году или позже.

```sql
SELECT first_name, last_name, active
FROM staff
WHERE NOT active = true;
```

Этот запрос выбирает данные из таблицы staff, отображая имена, фамилии и статус активности сотрудников, которые не активны.

Оператор **IN** в SQL используется для проверки, находится ли значение в заданном списке значений.

Он позволяет упростить запросы, которые требуют проверки на соответствие нескольким возможным значениям.

Синтаксис оператора IN

```
SELECT столбцы
FROM таблица
WHERE столбец IN (значение1, значение2, ..., значениеN);
```

Пример:

```sql
SELECT first_name, last_name, department
FROM staff
WHERE department IN ('HR', 'IT', 'Finance');
```

Этот запрос выбирает имена, фамилии и отделы сотрудников, которые работают в отделах HR, IT или Finance

Использование с подзапросом :

```sql
SELECT title, release_year
FROM film
WHERE film_id IN (
    SELECT film_id
    FROM rentals
    WHERE rental_date >= '2024-01-01'
);
```

Этот запрос выбирает названия и года выпуска фильмов, которые были арендованы начиная с 1 января 2024 года. Подзапрос возвращает идентификаторы фильмов, соответствующих условию.

Использование с отрицанием:

```sql
SELECT first_name, last_name
FROM staff
WHERE department NOT IN ('HR', 'IT');
```

Этот запрос выбирает имена и фамилии сотрудников, которые не работают в отделах HR или IT.

Оператор **BETWEEN** в SQL используется для фильтрации данных по диапазону значений.

Он позволяет выбрать строки, значения в которых находятся в заданном диапазоне, включая граничные значения.

Этот оператор может применяться как к числовым, так и к дата-временным данным.

Синтаксис оператора BETWEEN

```
SELECT столбцы
FROM таблица
WHERE столбец BETWEEN значение1 AND значение2;
```

Фильтрация по числовому диапазону:

```sql
SELECT first_name, salary
FROM employees
WHERE salary BETWEEN 30000 AND 50000;
```

Этот запрос выбирает имена и зарплаты сотрудников, чьи зарплаты находятся в диапазоне от 30,000 до 50,000 включительно.

Фильтрация по дате:

```sql
SELECT order_id, order_date
FROM orders
WHERE order_date BETWEEN '2024-01-01' AND '2024-12-31';
```

Этот запрос выбирает идентификаторы заказов и даты заказов, которые были сделаны в течение 2024 года, включая оба крайних дня.

Фильтрация с отрицанием:

```sql
SELECT first_name, last_name
FROM employees
WHERE salary NOT BETWEEN 30000 AND 50000;
```

Этот запрос выбирает имена и фамилии сотрудников, чьи зарплаты не находятся в диапазоне от 30,000 до 50,000.

Оператор LIKE в SQL используется для поиска строк, которые соответствуют определенному шаблону.

Это полезно для фильтрации данных, когда нужно найти строки, содержащие определенные подстроки или соответствующие определенным условиям.

Синтаксис оператора LIKE:

```
SELECT столбцы
FROM таблица
WHERE столбец LIKE шаблон;
```

Специальные символы шаблона в операторе LIKE:

%: Заменяет ноль или более символов.

_: Заменяет ровно один символ.

Поиск по начальной части строки:

```sql
SELECT first_name, last_name
FROM employees
WHERE first_name LIKE 'A%';
```

Этот запрос выбирает имена и фамилии сотрудников, чьи имена начинаются с буквы 'A'. Символ % заменяет любые символы после 'A'.

Поиск по содержимому строки:

```sql
SELECT email
FROM users
WHERE email LIKE '%@example.com';
```

Этот запрос выбирает электронные адреса пользователей, которые заканчиваются на '@example.com’.

Поиск по части строки с фиксированной длиной:

```sql
SELECT product_name
FROM products
WHERE product_code LIKE 'AB_123';
```

Этот запрос выбирает названия продуктов, где код продукта начинается с 'AB', за которым следует любой один символ (обозначенный _), затем '123'.

Использование NOT LIKE:

```sql
SELECT first_name, last_name
FROM employees
WHERE first_name NOT LIKE 'A%';
```

Этот запрос выбирает имена и фамилии сотрудников, чьи имена не начинаются с буквы 'A'.

Поиск по строкам, содержащим определенный набор символов:

```
SELECT address
FROM contacts
WHERE address LIKE '%Main St%';
```

Этот запрос выбирает данные из таблицы film, отображая названия и описания фильмов, в которых в поле description содержится слово "Scientist".

Сортировка данных при помощи ORDER BY.

Оператор ORDER BY в SQL используется для сортировки результатов запроса по одному или нескольким столбцам.

Он позволяет упорядочить строки в результирующем наборе данных в порядке возрастания или убывания.

Синтаксис оператора ORDER BY:

```
SELECT столбцы
FROM таблица
ORDER BY столбец1 [ASC|DESC], столбец2 [ASC|DESC], ...;
```

столбцы: Список столбцов, которые должны быть выбраны.

таблица: Имя таблицы, из которой будут извлечены данные.

столбец1, столбец2, ...: Столбцы, по которым будет производиться сортировка.

ASC: Сортировка по возрастанию (по умолчанию).

DESC: Сортировка по убыванию.

Сортировка по одному столбцу:

```sql
SELECT first_name, last_name
FROM employees
ORDER BY last_name ASC;
```

Этот запрос выбирает имена и фамилии сотрудников и сортирует их по фамилии в порядке возрастания (по умолчанию ASC).

Сортировка по нескольким столбцам:

```sql
SELECT first_name, last_name, hire_date
FROM employees
ORDER BY hire_date DESC, last_name ASC;
```

Этот запрос выбирает имена, фамилии и даты найма сотрудников. Сначала сортирует по дате найма в порядке убывания, а затем по фамилии в порядке возрастания для сотрудников, нанятых в один и тот же день.

Сортировка числовых значений:

```sql
SELECT product_name, price
FROM products
ORDER BY price DESC;
```

Этот запрос выбирает названия продуктов и их цены, сортируя их по цене в порядке убывания.

Оператор LIMIT.

Оператор LIMIT в SQL используется для ограничения числа строк, возвращаемых запросом. Это полезно для оптимизации запросов и управления объемом данных, возвращаемых клиенту. Особенно актуально это при работе с большими таблицами или при реализации постраничного отображения данных.

Синтаксис оператора LIMIT:

```
SELECT столбцы
FROM таблица
ORDER BY столбец
LIMIT количество_строк;
```

Возвращение первых N строк:

```sql
SELECT first_name, last_name
FROM employees
ORDER BY hire_date DESC
LIMIT 5;
```

Этот запрос выбирает имена и фамилии пяти самых последних нанятых сотрудников, упорядоченных по дате найма в порядке убывания.

Возвращение строк с определенного номера:

```sql
SELECT first_name, last_name
FROM employees
ORDER BY hire_date DESC
LIMIT 10 OFFSET 5;
```

Этот запрос выбирает 10 сотрудников, начиная с 6-го (пропустив первые 5), упорядоченных по дате найма в порядке убывания.

Домашнее задание:

1. Задача: Построить ER-диаграмму для простой системы управления библиотекой.

Сущности:

- Книга: id (первичный ключ), название, автор, год издания, жанр, количество экземпляров, статус (доступна/выдана).
- Читатель: id (первичный ключ), имя, фамилия, номер телефона, адрес, дата рождения.
- Заказ: id (первичный ключ), дата выдачи, дата возврата, читатель_id (внешний ключ), книга_id (внешний ключ).

Связи:

- Книга и Заказ связаны отношением "один ко многим": одна книга может быть выдана многим читателям.
- Читатель и Заказ связаны отношением "один ко многим": один читатель может взять много книг.

2. Дана таблица Students со следующей структурой:

- StudentID (INT) — уникальный идентификатор студента
- Name (VARCHAR) — имя студента
- Age (INT) — возраст студента
- City (VARCHAR) — город проживания
- Score (DECIMAL) — средний балл

Напишите запрос, который выведет все данные из таблицы Students.
Выведите только имена и возраст студентов.
Найдите всех студентов старше 20 лет.
Найдите студентов, проживающих в городе "Moscow".
Выведите студентов, чей средний балл больше или равен 4.5.
Отсортируйте студентов по возрастанию их среднего балла.
Выведите первые 3 записи из таблицы.
Найдите студентов, чьи имена начинаются на букву "A".
Выведите студентов, чей возраст находится в диапазоне от 18 до 25 лет.

3. Дана таблица "Employees" со следующей структурой:

- EmployeeID (INT)
- FirstName (VARCHAR)
- LastName (VARCHAR)
- Age (INT)
- Department (VARCHAR)
- Salary (DECIMAL)
- HireDate (DATE)

Необходимо написать следующие SELECT-запросы:

- Вывести полный список всех сотрудников, отсортированный по фамилии.
- Найти всех сотрудников старше 30 лет из отдела "HR".
- Вывести информацию о сотрудниках с зарплатой в диапазоне от 30000 до 50000.
- Найти всех сотрудников, чьи фамилии начинаются на букву "A".
- Вывести топ-5 самых высокооплачиваемых сотрудников.
- Найти среднюю зарплату по каждому отделу.
- Вывести список сотрудников, нанятых в 2023 году.
- Найти всех сотрудников с именем "John" или "Jane".
- Вывести список сотрудников, чьи фамилии содержат подстроку "ов".

Список литературы:

1. Руководство по MySQL.
2. Видеокурс.

Материалы лекций:

https://github.com/ShViktor72/Education

Обратная связь:

colledge20education23@gmail.com