Тема 5. Работа с несколькими таблицами. Связи между таблицами. Типы связей. Первичные и внешние ключи. Операции объединения таблиц.

Цель занятия:

Ознакомиться с концепциями работы с несколькими таблицами в базах данных, с видами связей между таблицами, понятиями первичных и внешних ключей, а также операциями объединения таблиц.

Учебные вопросы:

1. Введение.
2. Типы связей между таблицами.
3. Первичные и внешние ключи.
4. Операции объединения таблиц.

1. Введение.

Зачем нужны несколько таблиц в базе данных?

Использование нескольких таблиц в базе данных необходимо для организации данных в структурированную, логически упорядоченную и эффективную систему. Вот основные причины, почему важно разделять данные на несколько таблиц:

- **Нормализация данных.**

Устранение избыточности: Когда данные дублируются в одной таблице, это приводит к избыточности и увеличению объема базы данных. Разделение данных на несколько таблиц помогает устранить эту избыточность, сохраняя данные только в одном месте и ссылаясь на них из других таблиц.

Избежание аномалий: Аномалии могут возникать при вставке, обновлении или удалении данных, если структура таблицы не оптимальна. Нормализация данных (разделение данных по таблицам) помогает избежать таких проблем.

- **Удобство и логика хранения данных**

Группировка данных по логическим сущностям: В реальной жизни данные можно разделить на логические группы, такие как клиенты, заказы, товары и т.д. Разделение этих данных на соответствующие таблицы позволяет организовать данные более логично и эффективно.

Удобство работы с данными: Когда данные логически организованы в отдельные таблицы, их легче обновлять, удалять или извлекать. Это также упрощает чтение и понимание структуры базы данных.

- **Поддержка отношений между данными**

Отношения между сущностями: В реальном мире объекты могут быть связаны между собой. Например, каждый заказ связан с определенным клиентом и содержит один или несколько товаров. Использование нескольких таблиц позволяет правильно моделировать эти отношения с помощью связей типа один-к-одному, один-ко-многим или многие-ко-многим.

Поддержка целостности данных: Использование первичных и внешних ключей для связывания таблиц помогает обеспечить целостность данных, гарантируя, что ссылки между данными всегда остаются корректными.

- **Повышение производительности**

Более эффективные запросы: Разделение данных на отдельные таблицы позволяет оптимизировать SQL-запросы, так как меньше данных нужно обрабатывать в каждой таблице. Это особенно важно для больших баз данных.

Уменьшение объема данных в отдельных таблицах: Разделение данных помогает уменьшить количество записей в каждой таблице, что ускоряет операции выборки, вставки, обновления и удаления.

- **Облегчение управления данными**

Обновление данных: Если определенные данные изменяются (например, адрес клиента), их нужно обновить только в одном месте, что снижает вероятность ошибок и упрощает поддержку данных.

Управление правами доступа: Разделение данных на несколько таблиц позволяет более точно настроить права доступа, ограничивая доступ к определенным данным для разных пользователей.

- **Масштабируемость**

Рост данных: С течением времени количество данных в базе может существенно увеличиться. Разделение данных на несколько таблиц помогает справляться с увеличением объема данных и обеспечивает масштабируемость базы данных.

Расширение функциональности: Когда появляется необходимость добавить новые функции или сущности в базу данных, легче создать новые таблицы и установить связи между ними, чем пытаться вписать все в одну таблицу.

Пример: Система управления заказами в интернет-магазине

Задача: Управление заказами клиентов, включая информацию о покупателях, заказанных товарах и их доставке.

Необходимые таблицы:

Клиенты (Customers): Содержит информацию о клиентах (ID клиента, имя, контактная информация).

Заказы (Orders): Содержит информацию о заказах (ID заказа, дата заказа, ID клиента).

Товары (Products): Содержит информацию о товарах (ID товара, название, цена).

Заказанные товары (OrderDetails): Содержит детали каждого заказа (ID заказа, ID товара, количество, цена).

Таблица Orders связана с таблицей Customers через внешний ключ (ID клиента), а также с таблицей OrderDetails, которая, в свою очередь, связана с таблицей Products через внешний ключ (ID товара). Это позволяет получить полную информацию о заказе, включая данные о клиенте и товарах.

Управление библиотекой

Задача: Учёт книг, читателей и выданных книг в библиотеке.

Необходимые таблицы:

Книги (Books): Содержит информацию о книгах (ID книги, название, автор, год издания).

Читатели (Readers): Содержит информацию о читателях (ID читателя, имя, контактная информация).

Выдачи (Orders): Содержит информацию о выданных книгах (ID выдачи, ID книги, ID читателя, дата выдачи, дата возврата).

Таблица Orders связывает Books и Readers, что позволяет отслеживать, какие книги находятся на руках у каких читателей и когда они должны быть возвращены.

2. Типы связей между таблицами.

Типы связей между таблицами в базе данных описывают, как данные в одной таблице связаны с данными в другой таблице.

Понимание этих связей необходимо для правильного проектирования структуры базы данных и организации данных.

Основные типы связей:

- Связь "Один-к-одному" (One-to-One)
- Связь "Один-ко-многим" (One-to-Many)
- Связь "Многие-ко-многим" (Many-to-Many)

Связи между отношениями

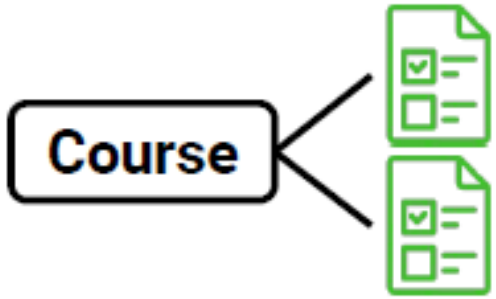
Типы связей

один к одному

пользователь и дополнительная информация о нём

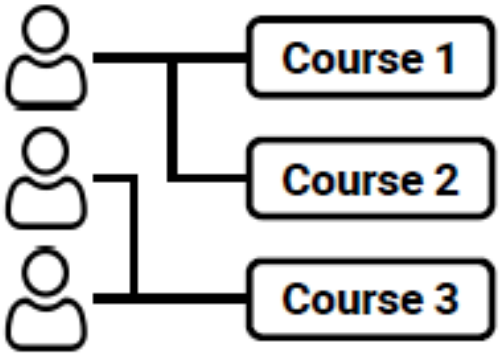
один ко многим

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**.

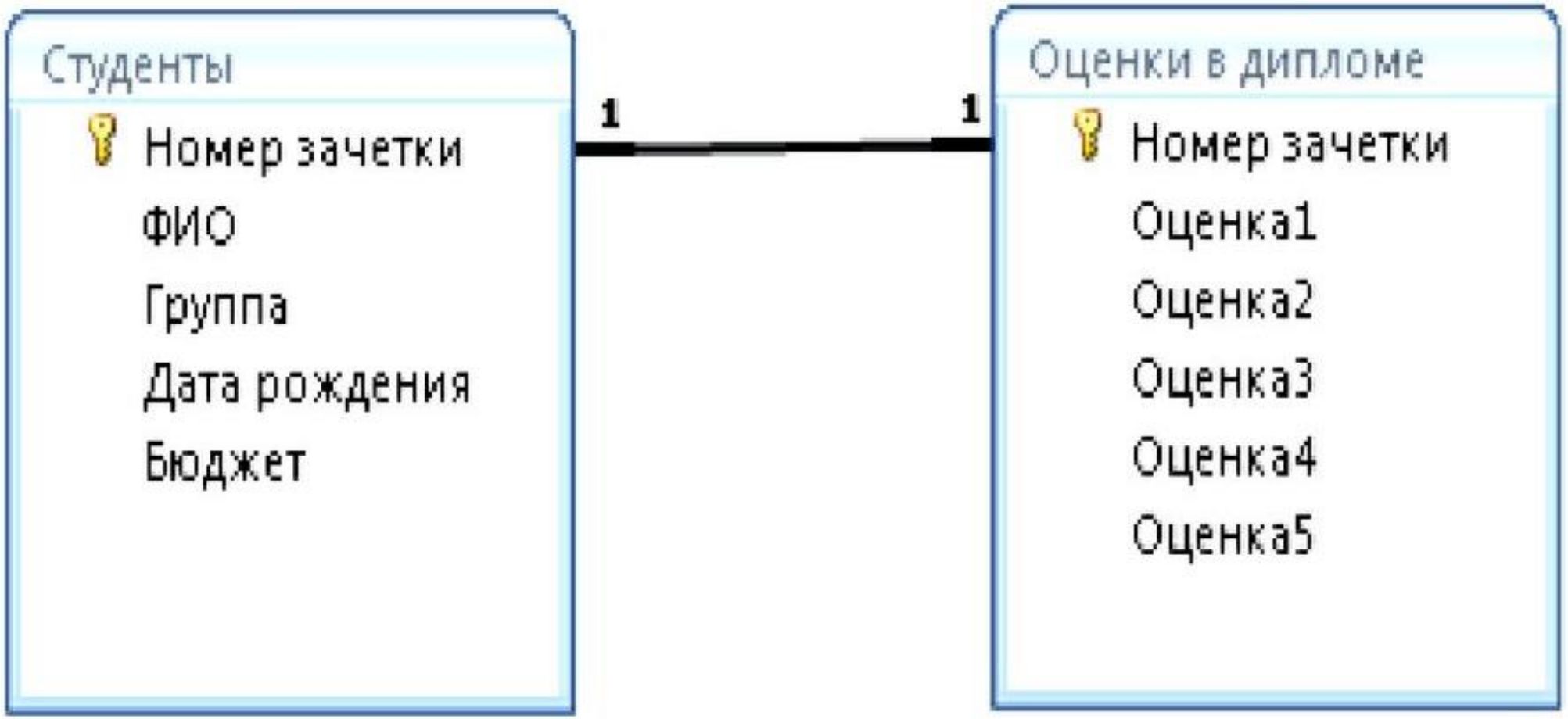
Связь "Один-к-одному" (One-to-One)

Определение: Каждый элемент в одной таблице связан не более чем с одним элементом в другой таблице, и наоборот.

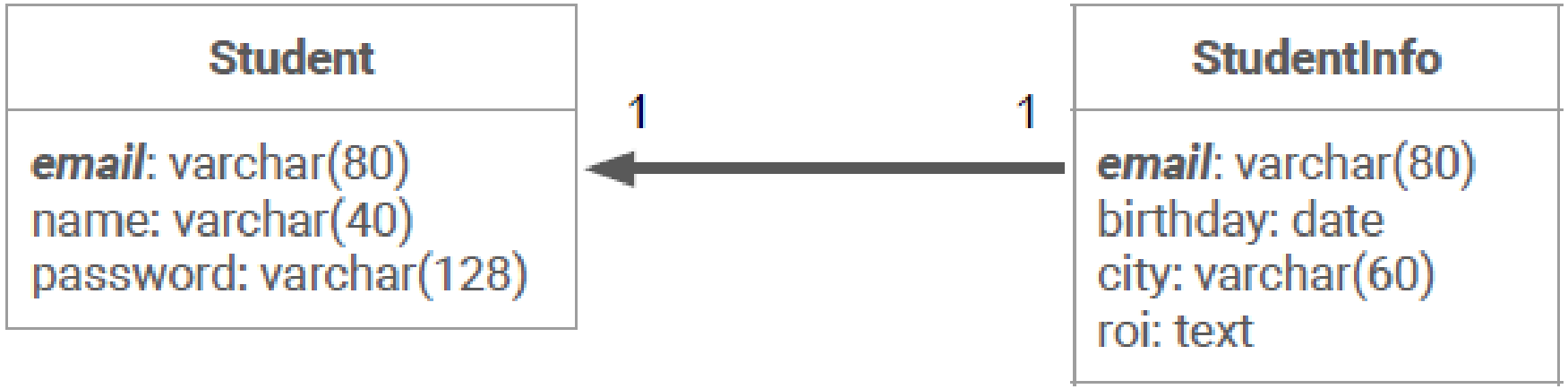
Пример: В базе данных о сотрудниках и их паспортных данных можно иметь таблицу Employees (Сотрудники) и таблицу Passports (Паспорта), где каждый сотрудник имеет один уникальный паспорт, а каждый паспорт относится к одному сотруднику.

Реализация: Связь может быть реализована через внешний ключ в одной из таблиц, который ссылается на первичный ключ другой таблицы. В некоторых случаях можно объединить обе таблицы в одну, если данные тесно связаны и не требуют разделения.

Связь один к одному

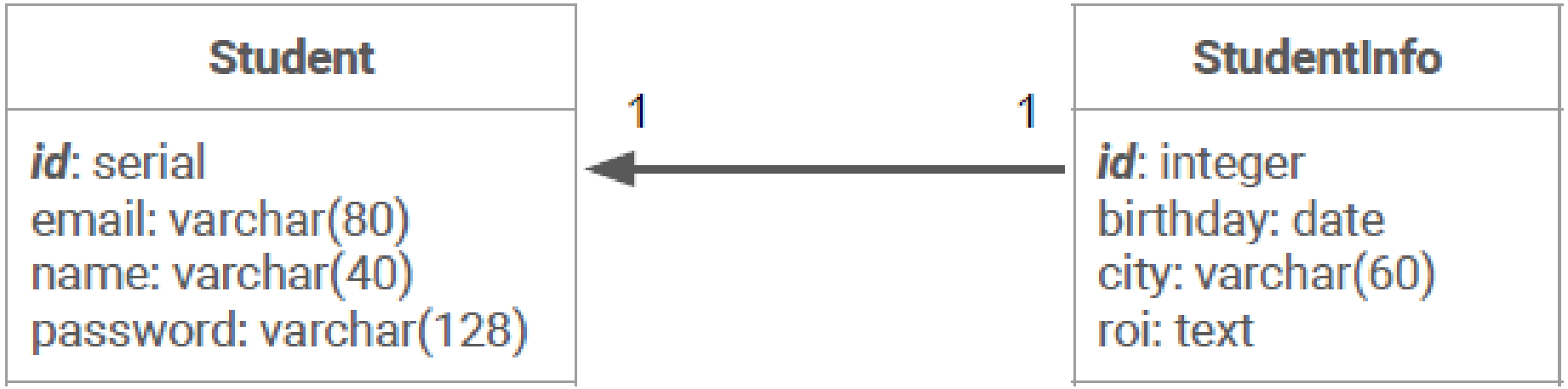


Связываем студента и дополнительную информацию о нём.




```
create table if not exists Student (
     email varchar(80) primary key,
     name varchar(40) not null,
     password varchar(128) not null
);

create table if not exists StudentInfo (
     email varchar(80) primary key references Student(email),
     birthday date,
     city varchar(60),
     roi text
);
```




```
create table if not exists Student (
     id serial primary key,
     email varchar(80) unique not null,
     name varchar(40) not null,
     password varchar(128) not null
);

create table if not exists StudentInfo (
     id integer primary key references Student(id),
     birthday date,
     city varchar(60),
     roi text
);
```

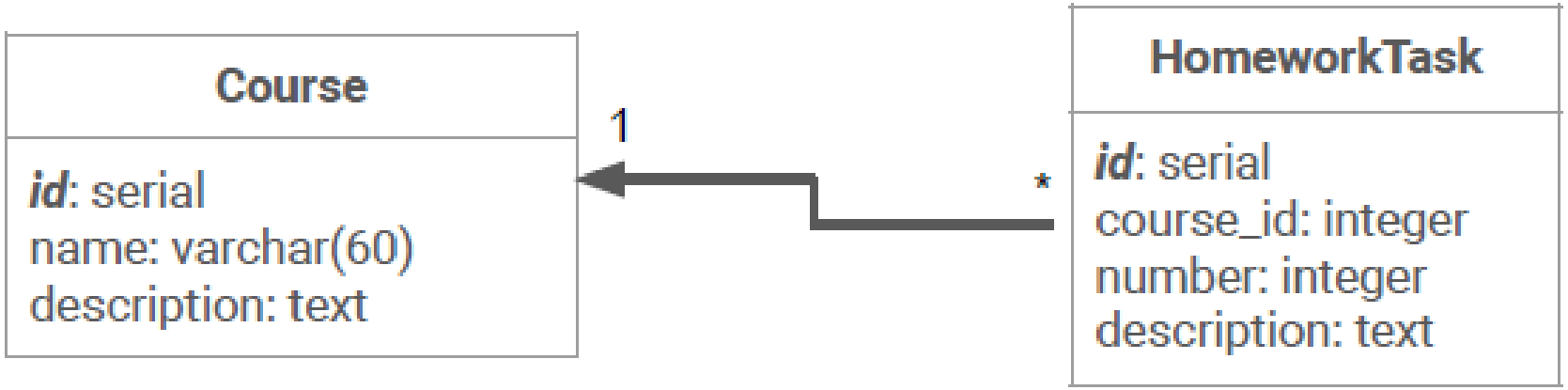
Связь "Один-ко-многим" (One-to-Many)

Определение: Один элемент в одной таблице может быть связан с несколькими элементами в другой таблице, но каждый элемент в другой таблице связан только с одним элементом в первой таблице.

Пример: В базе данных интернет-магазина одна таблица Customers (Клиенты) может быть связана с несколькими записями в таблице Orders (Заказы), так как один клиент может сделать много заказов, но каждый заказ связан с конкретным клиентом.

Реализация: В таблице, представляющей "многие" (например, Orders), используется внешний ключ, который ссылается на первичный ключ в таблице "один" (например, Customers).

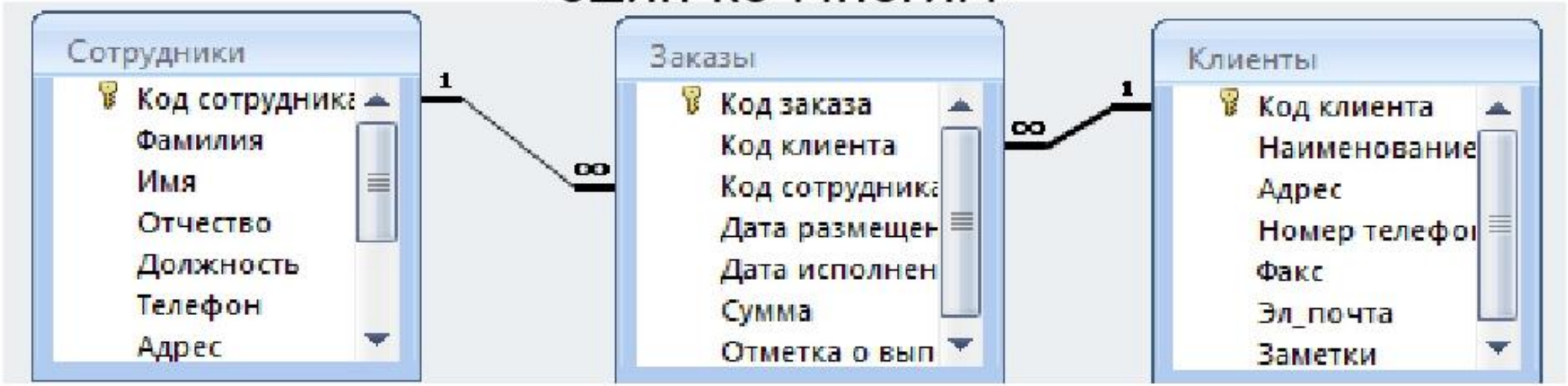
Связываем описание домашних заданий с курсами, к которым они относятся.




```
create table if not exists Course (
     id serial primary key,
     name varchar(60) not null,
     description text
);

create table if not exists HomeworkTask (
     id serial primary key,
     course_id integer not null references Course(id),
     number integer not null,
     description text not null
);
```

«один-ко-многим»



- Связь на схеме данных они отображаются в виде соединительных линий со специальными значками около таблиц: **«1»** вблизи главной таблицы (имеющей первичный ключ) и **«∞»** вблизи подчиненной таблицы (имеющей внешний ключ).

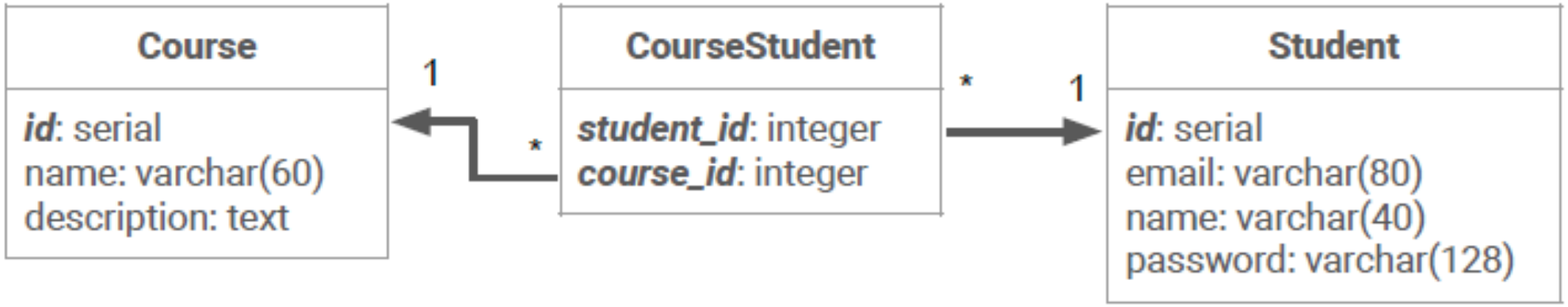
3. Связь "Многие-ко-многим" (Many-to-Many)

Определение: Один элемент в первой таблице может быть связан с несколькими элементами в другой таблице, и наоборот.

Пример: В учебной базе данных один студент может записаться на несколько курсов, а каждый курс может включать нескольких студентов. В этом случае связь между таблицами Students (Студенты) и Courses (Курсы) будет "многие-ко-многим".

Реализация: Для реализации такой связи требуется создание промежуточной таблицы (часто называемой таблицей связи или junction table), которая содержит внешние ключи, ссылающиеся на первичные ключи из обеих таблиц. Например, таблица StudentCourses может содержать два поля: StudentID и CourseID, которые будут ссылаться на соответствующие таблицы.

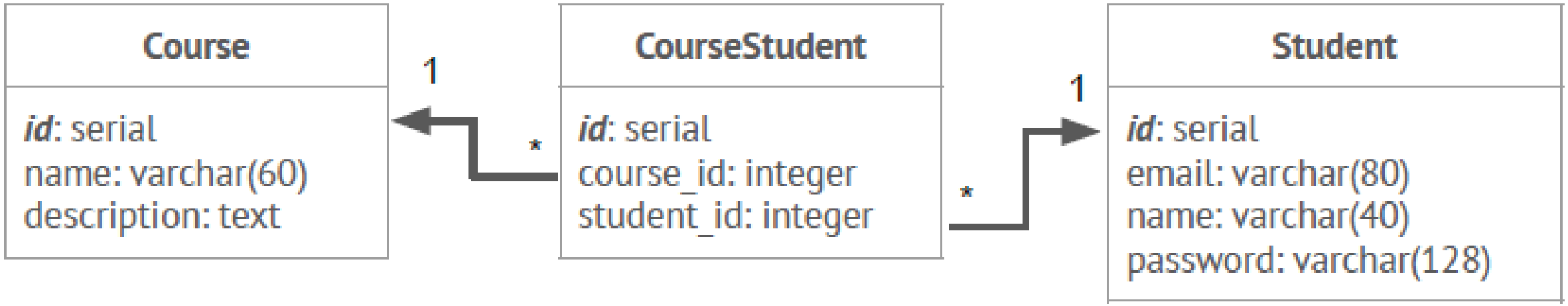
Связываем студентов и курсы, на которые они записаны.




```
create table if not exists CourseStudent (
     course id integer references Course(id),
     student id integer references Student(id),
     constraint pk primary key (course_id, student_id)
);
```

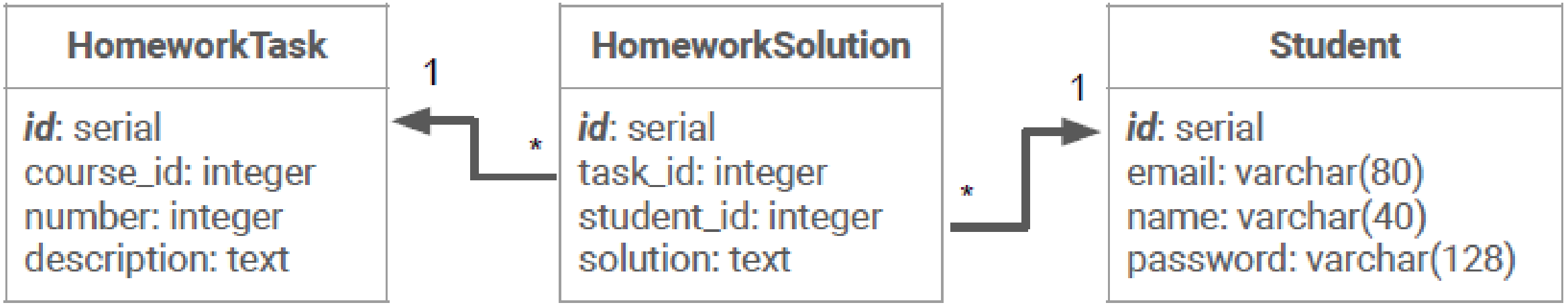
student_id	course_id
1 (Вова)	1 (Python)
1 (Вова)	2 (Java)
2 (Дима)	1 (Python)

Многие ко многим. Вариант 2




```
create table if not exists CourseStudent (
     id serial primary key,
     course_id integer not null references Course(id),
     student_id integer not null references Student(id)
);
```

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




```
create table if not exists HomeworkSolution (
      id serial primary key,
      task_id integer not null references HomeworkTask(id),
      student_id integer not null references Student(id),
      solution text not null
);
```

3. Первичные и внешние ключи.

Первичный ключ (Primary Key) — это **один** или **несколько** столбцов в таблице, которые однозначно идентифицируют каждую запись (строку) в этой таблице.

Каждый первичный ключ должен содержать уникальные значения, и ни одно из значений в столбце(ах) первичного ключа не может быть **NULL**.

Основные характеристики первичного ключа:

Уникальность: Значения первичного ключа должны быть уникальными для каждой строки в таблице, что позволяет однозначно идентифицировать каждую запись.

Не допускается NULL: Первичный ключ не может содержать значения NULL, так как это нарушает принцип уникальности.

Один первичный ключ на таблицу: В каждой таблице может быть только один первичный ключ, который может состоять из одного столбца (простой первичный ключ) или из нескольких столбцов (составной первичный ключ).

Первичные ключи в базе данных бывают двух типов: **простой** и **составной**. Оба типа выполняют одну и ту же основную функцию — однозначно идентифицируют записи в таблице, но различаются по структуре.

- **Простой первичный ключ** — это первичный ключ, состоящий из одного столбца таблицы. Значения в этом столбце уникальны для каждой строки, и они не могут быть NULL.
- **Составной первичный ключ** (Composite Primary Key) — это первичный ключ, который состоит из нескольких столбцов. В совокупности значения этих столбцов должны быть уникальными для каждой строки. Составной первичный ключ используется, когда уникальность строки должна определяться комбинацией нескольких атрибутов.

Простой первичный ключ.

В таблице Employees (Сотрудники) столбец EmployeeID (ID сотрудника) может быть объявлен первичным ключом, так как он содержит уникальный идентификатор для каждого сотрудника.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    HireDate DATE
);
```

Составной первичный ключ.

В таблице OrderDetails (Детали заказа), которая хранит информацию о товарах в заказе, можно использовать составной первичный ключ, состоящий из столбцов OrderID (ID заказа) и ProductID (ID товара), чтобы гарантировать уникальность записи о каждом товаре в каждом заказе.

```
CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    PRIMARY KEY (OrderID, ProductID)
);
```

```sql
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Projects (
    ProjectID INT PRIMARY KEY,
    ProjectName VARCHAR(100)
);

CREATE TABLE EmployeeProjects (
    EmployeeID INT,
    ProjectID INT,
    PRIMARY KEY (EmployeeID, ProjectID),
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID),
    FOREIGN KEY (ProjectID) REFERENCES Projects(ProjectID)
);
```

Пример: база данных системы управления проектами с таблицами Employees (Сотрудники) и Projects (Проекты).

Один сотрудник может участвовать в нескольких проектах, и один проект может включать нескольких сотрудников.

Для реализации такой связи создаётся промежуточная таблица EmployeeProjects.

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

```
create table if not exists Student (
     id serial primary key,
     name varchar(40) not null,
     gpa numeric(3, 2) check (gpa >= 0 and gpa <= 5)
);

create table if not exists Student (
     name varchar(40) primary key,
     gpa numeric(3, 2) check (gpa >= 0 and gpa <= 5)
);  # какой недостаток?

create table if not exists Student (
     name varchar(40),
     gpa numeric(3, 2) check (gpa >= 0 and gpa <= 5),
     constraint student_pk primary key (name, gpa)
);  # здесь все ок?
```

Primary key

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

Зачем нужен первичный ключ?

- **Идентификация**: Первичный ключ позволяет однозначно идентифицировать каждую запись в таблице, что необходимо для корректного выполнения операций вставки, обновления, удаления и поиска данных.
- **Обеспечение целостности данных**: Наличие первичного ключа предотвращает появление дубликатов в таблице и обеспечивает корректное построение связей между таблицами через внешние ключи.
- **Оптимизация**: Первичные ключи помогают оптимизировать работу базы данных, так как многие системы управления базами данных (СУБД) автоматически создают индексы на первичные ключи, что ускоряет операции поиска.

Внешний ключ (Foreign Key)

Внешний ключ (Foreign Key) — это столбец или набор столбцов в таблице, значения которых ссылаются на первичный ключ (или уникальный ключ) другой таблицы. Внешний ключ создаёт связь между двумя таблицами, обеспечивая целостность данных и предотвращая несовместимость данных.

Роль внешнего ключа в установлении связей между таблицами:

- **Создание связи между таблицами**: Внешний ключ связывает строки одной таблицы с соответствующими строками другой таблицы. Это позволяет структурировать данные так, чтобы информация, которая относится к одной сущности, была организована и связана с другой сущностью.
- **Поддержание ссылочной целостности**: Внешние ключи гарантируют, что значения в дочерней таблице (таблице, содержащей внешний ключ) всегда соответствуют существующим значениям в родительской таблице (таблице, на которую указывает внешний ключ). Это предотвращает создание записей, которые ссылаются на несуществующие данные, и защищает от ошибок при удалении или обновлении данных.
- **Обеспечение целостности данных**: Внешний ключ помогает поддерживать целостность базы данных. Например, при попытке удаления строки в родительской таблице, которая имеет связанные строки в дочерней таблице, СУБД может запретить это действие или автоматически удалить связанные строки (каскадное удаление).

```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Пример: база данных, где есть две таблицы: Customers (Клиенты) и Orders (Заказы).
Один клиент может сделать несколько заказов, но каждый заказ относится только к одному клиенту.

Связь "Один-ко-многим"

CustomerID в таблице Orders — это внешний ключ, который ссылается на столбец CustomerID в таблице Customers.

```sql
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(100)
);

CREATE TABLE StudentCourses (
    StudentID INT,
    CourseID INT,
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

Пример: система управления университетом, где есть таблицы Students (Студенты) и Courses (Курсы).

Один студент может быть записан на несколько курсов, и один курс может включать несколько студентов.

Для реализации такой связи используется промежуточная таблица StudentCourses.

Связь "Многие-ко-многим"

Таблица StudentCourses содержит два внешних ключа: StudentID, ссылающийся на Students, и CourseID, ссылающийся на Courses.

Ограничения и поддержка целостности данных

Принудительное выполнение ссылочной целостности

Ссылочная целостность — это свойство базы данных, обеспечивающее, что отношения между таблицами остаются логически связными. Она гарантирует, что каждый внешний ключ в дочерней таблице ссылается на существующую запись в родительской таблице.

Система управления базами данных (СУБД) обеспечивает принудительное выполнение ссылочной целостности за счёт использования внешних ключей и связанных с ними ограничений. Вот основные механизмы:

- **Ограничение внешнего ключа (Foreign Key Constraint):**

Это ограничение заставляет базу данных проверять, что любое значение внешнего ключа соответствует значению в первичном ключе (или уникальном ключе) в другой таблице.

Если при вставке или обновлении данных обнаруживается нарушение этого ограничения, операция будет отклонена, и данные не будут сохранены.

- **Обеспечение уникальности данных:**

СУБД гарантирует, что записи в родительской таблице, на которые ссылаются внешние ключи, будут уникальными и неповторимыми.

- **Запрет на "висячие" ссылки:**

Это предотвращает ситуации, когда запись в дочерней таблице ссылается на несуществующую запись в родительской таблице.

Поведение при удалении и обновлении данных (каскадные действия)

При удалении или обновлении записей в родительской таблице, на которые ссылаются записи в дочерней таблице, СУБД может применять различные стратегии для обеспечения целостности данных.

Эти стратегии задаются с помощью каскадных действий (ON DELETE и ON UPDATE) в определении внешнего ключа.

Каскадные действия при удалении:

CASCADE:

При удалении записи в родительской таблице автоматически удаляются все связанные записи в дочерней таблице.

Пример: Если удаляется заказ в таблице Orders, то все связанные позиции в таблице OrderDetails также будут удалены.

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE CASCADE
```

SET NULL:

При удалении записи в родительской таблице соответствующие внешние ключи в дочерней таблице устанавливаются в NULL.

Пример: Если удаляется клиент в таблице Customers, то поле CustomerID в связанных записях в таблице Orders будет установлено в NULL.

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE SET NULL
```

SET DEFAULT:

При удалении записи в родительской таблице соответствующие внешние ключи в дочерней таблице устанавливаются в значение по умолчанию, если оно определено.

Пример: Если удаляется запись из таблицы Projects, то связанные записи в таблице EmployeeProjects могут быть установлены в значение по умолчанию.

```
FOREIGN KEY (ProjectID) REFERENCES Projects(ProjectID) ON DELETE SET DEFAULT
```

RESTRICT / NO ACTION:

Удаление записи в родительской таблице запрещено, если существуют связанные записи в дочерней таблице.

Пример: Если попытаться удалить клиента, который имеет заказы, операция удаления будет отклонена.

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE RESTRICT
```

Каскадные действия при обновлении:

CASCADE:

При изменении значения первичного ключа в родительской таблице автоматически изменяются все соответствующие внешние ключи в дочерней таблице.

Пример: Если обновляется CustomerID в таблице Customers, то все связанные записи в таблице Orders также обновят своё значение CustomerID.

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON UPDATE CASCADE
```

SET NULL:

При изменении значения первичного ключа в родительской таблице соответствующие внешние ключи в дочерней таблице устанавливаются в NULL.

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON UPDATE SET NULL
```

SET DEFAULT:

При обновлении значения первичного ключа в родительской таблице внешние ключи в дочерней таблице могут быть установлены в значение по умолчанию.

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON UPDATE SET DEFAULT
```

RESTRICT / NO ACTION:

Обновление значения первичного ключа в родительской таблице запрещено, если существуют связанные записи в дочерней таблице.

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON UPDATE RESTRICT
```

Поддержка целостности данных с помощью внешних ключей и каскадных действий позволяет СУБД гарантировать, что все связи между таблицами остаются корректными и непротиворечивыми.

Это важно для предотвращения ошибок и сохранения целостности данных в реляционной базе данных.

4. Операции объединения таблиц.

Объединение таблиц (JOIN) — это одна из наиболее важных операций в SQL. Она позволяет комбинировать данные из нескольких таблиц, связанных между собой по определенным условиям.

Это особенно полезно, когда нужно получить комплексную информацию, которая разбросана по разным таблицам базы данных.

Объединение таблиц

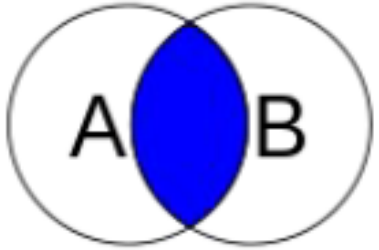
JOIN – оператор, предназначенный для объединения таблиц по определенному столбцу или связке столбцов (как правило, по первичному ключу).

JOIN записывается после FROM и до WHERE. Через оператор ON необходимо явно указывать столбцы, по которым происходит объединение. В общем виде структура запросы с JOIN выглядит так:

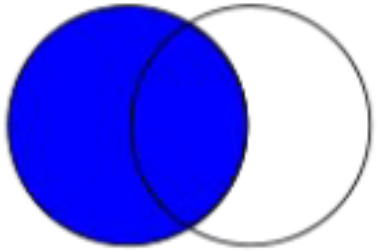
```
SELECT tables_columns FROM table_1
JOIN table_2 ON table_1.column = table_2.column;
```

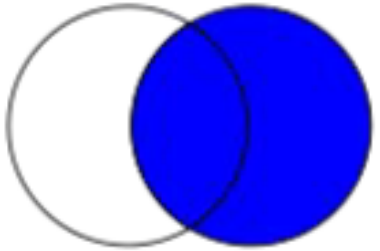
Типы JOIN

```
SELECT <fields>
FROM TableA A
INNER JOIN TableB B
ON A.key = B.key
```




```
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
```

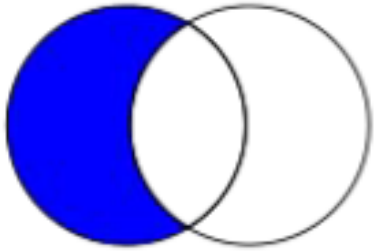


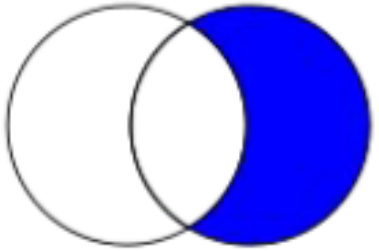


```
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
```

SQL JOINS

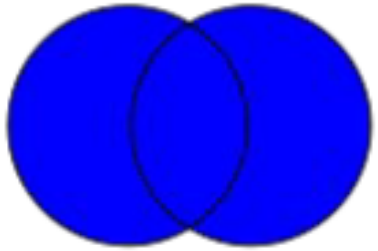
```
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
WHERE B.key IS NULL
```

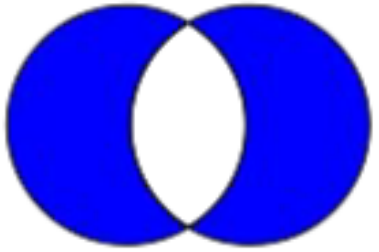




```
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
```

```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
```





```
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

Основные виды JOIN

[INNER] JOIN – в выборку попадут только те данные, по которым выполняются условия соединения;

LEFT [OUTER] JOIN – в выборку попадут все данные из таблицы **A** и только те данные из таблицы **B**, по которым выполнится условие соединения. Недостающие данные вместо строк таблицы B будут представлены **NULL**.

RIGHT [OUTER] JOIN – в выборку попадут все данные из таблицы **B** и только те данные из таблицы **A**, по которым выполнится условие соединения. Недостающие данные вместо строк таблицы **A** будут представлены **NULL**.

FULL [OUTER] JOIN – в выборку попадут все строки таблицы **A** и таблицы **B**. Если для строк таблицы **A** и таблицы **B** выполняются условия соединения, то они объединяются в одну строку. Если данных в какой-то таблице не имеется, то на соответствующие места вставляются **NULL**.

CROSS JOIN – объединение каждой строки таблицы **A** с каждой строкой таблицы **B.**

[INNER] JOIN

[INNER] JOIN – в выборку попадут только те данные, по которым выполняются условия соединения.

Левая таблица (к ней присоединяем)

employees		
id	name	office_id
1	John	1
2	James	2
3	Kate	3
4	Mary	999

*** Код 999 - для удаленщиков**

Правая таблица

offices	
id	office_name
1	Кабинет 42
2	Фиолетовый кабинет
3	Кабинет 126
4	Кабинет 22

Получаем только тех сотрудников, которые сидят в кабинетах и только те кабинеты, в которых сидят сотрудники.

employees.id	name	offices.id	office_name
1	John	1	Кабинет 42
2	James	2	Фиолетовый кабинет
3	Kate	3	Кабинет 126

```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- Пример INNER JOIN
SELECT Customers.FirstName, Customers.LastName, Orders.OrderID, Orders.OrderDate
FROM Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID;
```

Пример: Объединение таблиц клиентов и заказов.
Есть две таблицы: Customers (Клиенты) и Orders (Заказы).
Мы хотим получить информацию о клиентах и их заказах.

В этом запросе:
Мы выбираем имена клиентов и номера их заказов.
Объединяем таблицы Customers и Orders по столбцу CustomerID.
В результате мы получаем только те клиенты, которые сделали заказы, и информацию о каждом заказе.

```sql
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Projects (
    ProjectID INT PRIMARY KEY,
    ProjectName VARCHAR(100)
);

CREATE TABLE EmployeeProjects (
    EmployeeID INT,
    ProjectID INT,
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID),
    FOREIGN KEY (ProjectID) REFERENCES Projects(ProjectID)
);
```

Объединение таблиц сотрудников и проектов
Предположим, у нас есть таблицы Employees (Сотрудники), Projects (Проекты) и EmployeeProjects (Проекты сотрудников), которые содержат информацию о том, какие сотрудники работают над какими проектами.

```sql
-- Пример INNER JOIN
SELECT Employees.FirstName, Employees.LastName, Projects.ProjectName
FROM EmployeeProjects
INNER JOIN Employees
ON EmployeeProjects.EmployeeID = Employees.EmployeeID
INNER JOIN Projects
ON EmployeeProjects.ProjectID = Projects.ProjectID;
```

В этом запросе:

Мы выбираем имена сотрудников и названия проектов, над которыми они работают.

Объединяем три таблицы: EmployeeProjects, Employees, и Projects.

Используем два условия объединения для связывания данных между EmployeeProjects и двумя другими таблицами.

LEFT JOIN

LEFT JOIN – в выборку попадут все данные из левой таблицы и только те данные из правой, по которым выполняется условие соединения.

Левая таблица (к ней присоединяем)

employees		
id	name	office_id
1	John	1
2	James	2
3	Kate	3
4	Mary	999

*** Код 999 - для удаленщиков**

Правая таблица

offices	
id	office_name
1	Кабинет 42
2	Фиолетовый кабинет
3	Кабинет 126
4	Кабинет 22

Получаем всех сотрудников и только те кабинеты, в которых кто-то сидит.

employees.id	name	offices.id	office_name
1	John	1	Кабинет 42
2	James	2	Фиолетовый кабинет
3	Kate	3	Кабинет 126
4	Mary	NULL	NULL

```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- Пример LEFT JOIN
SELECT Customers.FirstName, Customers.LastName, Orders.OrderID, Orders.OrderDate
FROM Customers
LEFT JOIN Orders
ON Customers.CustomerID = Orders.CustomerID;
```

Пример использования LEFT JOIN. Объединение таблиц клиентов и заказов.
Предположим, у нас есть таблицы Customers (Клиенты) и Orders (Заказы). Мы хотим получить список всех клиентов, включая тех, у которых нет заказов.

В этом запросе мы выбираем все строки из таблицы Customers и соответствующие строки из таблицы Orders. Если у клиента нет заказов, в полях OrderID и OrderDate будут значения NULL.

```sql
-- Пример LEFT JOIN
SELECT Employees.FirstName, Employees.LastName, Projects.ProjectName
FROM Employees
LEFT JOIN EmployeeProjects
ON Employees.EmployeeID = EmployeeProjects.EmployeeID
LEFT JOIN Projects
ON EmployeeProjects.ProjectID = Projects.ProjectID;
```

В этом запросе мы выбираем всех сотрудников, включая тех, кто не участвует ни в одном проекте.

В полях ProjectName будут NULL для сотрудников, у которых нет назначенных проектов.

RIGHT JOIN

RIGHT JOIN – в выборку попадут все данные из правой таблицы и только те данные из левой, по которым выполняется условие соединения.

Левая таблица (к ней присоединяем)

employees		
id	name	office_id
1	John	1
2	James	2
3	Kate	3
4	Mary	999

*** Код 999 - для удаленщиков**

Правая таблица

offices	
id	office_name
1	Кабинет 42
2	Фиолетовый кабинет
3	Кабинет 126
4	Кабинет 22

Получаем все кабинеты и только тех сотрудников, которые сидят в них (не удаленщиков).

employees.id	name	offices.id	office_name
1	John	1	Кабинет 42
2	James	2	Фиолетовый кабинет
3	Kate	3	Кабинет 126
NULL	NULL	4	Кабинет 22

LEFT JOIN с фильтрацией по полю

LEFT JOIN с фильтрацией по полю – в выборку попадут только те данные из левой таблицы, по которым не выполняется условия соединения.

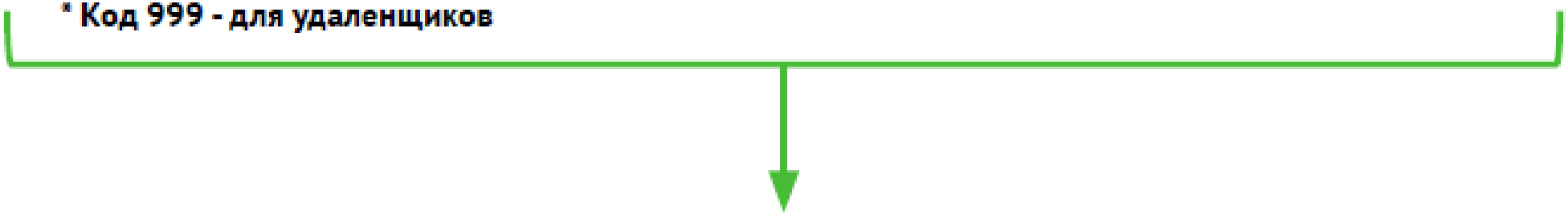
Левая таблица (к ней присоединяем)

employees		
id	name	office_id
1	John	1
2	James	2
3	Kate	3
4	Mary	999

*** Код 999 - для удаленщиков**

Правая таблица

offices	
id	office_name
1	Кабинет 42
2	Фиолетовый кабинет
3	Кабинет 126
4	Кабинет 22



Получаем только удаленщиков.

employees.id	name	offices.id	office_name
4	Mary	NULL	NULL

```sql
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(100)
);

CREATE TABLE StudentCourses (
    StudentID INT,
    CourseID INT,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

Пример использования LEFT JOIN с фильтрацией. Фильтрация студентов, не записанных на определённый курс. Рассмотрим таблицы Students (Студенты), Courses (Курсы)и StudentCourses (Курсы студентов). Мы хотим получить список всех студентов, которые не записаны на курс с конкретным названием.

```sql
-- Пример LEFT JOIN с фильтрацией
SELECT Students.FirstName, Students.LastName
FROM Students
LEFT JOIN StudentCourses
ON Students.StudentID = StudentCourses.StudentID
LEFT JOIN Courses
ON StudentCourses.CourseID = Courses.CourseID
WHERE Courses.CourseName <> 'Advanced Mathematics' OR Courses.CourseName IS NULL;
```

В этом запросе мы объединяем таблицы Students, StudentCourses, и Courses.

Фильтруем результат, чтобы получить студентов, которые не записаны на курс "Advanced Mathematics", или студентов, которые не записаны ни на один курс (где CourseName равен NULL).

RIGHT JOIN с фильтрацией по полю

RIGHT JOIN с фильтрацией по полю – в выборку попадут только те данные из правой таблицы, по которым не выполняется условия соединения.

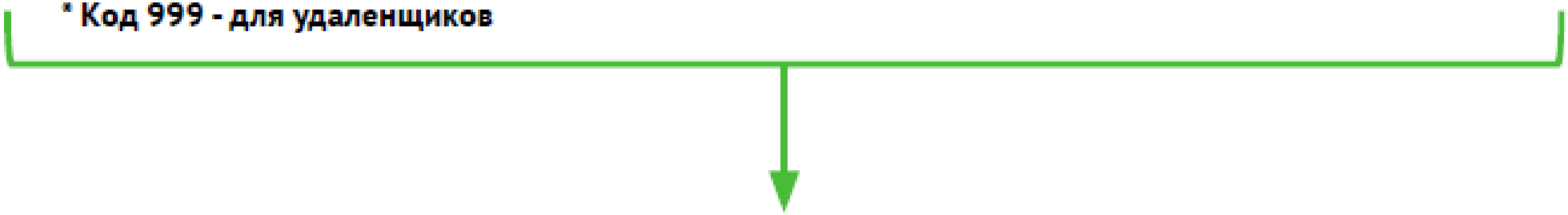
Левая таблица (к ней присоединяем)

employees		
id	name	office_id
1	John	1
2	James	2
3	Kate	3
4	Mary	999

*** Код 999 - для удаленщиков**

Правая таблица

offices	
id	office_name
1	Кабинет 42
2	Фиолетовый кабинет
3	Кабинет 126
4	Кабинет 22



Получаем только пустые кабинеты.

employees.id	name	offices.id	office_name
NULL	NULL	4	Кабинет 22

FULL OUTER JOIN

FULL OUTER JOIN – выборку попадут все строки исходных таблиц. Если по данным не выполняется условие соединения, то на соответствующие места вставляются **NULL**.

Левая таблица (к ней присоединяем)

employees		
id	name	office_id
1	John	1
2	James	2
3	Kate	3
4	Mary	999

*** Код 999 - для удаленщиков**

Правая таблица

offices	
id	office_name
1	Кабинет 42
2	Фиолетовый кабинет
3	Кабинет 126
4	Кабинет 22

Получаем абсолютно все кабинеты и абсолютно всех сотрудников.

employees.id	name	offices.id	office_name
1	John	1	Кабинет 42
2	James	2	Фиолетовый кабинет
3	Kate	3	Кабинет 126
4	Mary	NULL	NULL
NULL	NULL	4	Кабинет 22

```sql
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);

-- Пример FULL OUTER JOIN
SELECT Customers.FirstName, Customers.LastName, Orders.OrderID, Orders.OrderDate
FROM Customers
FULL OUTER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID;
```

Примеры использования FULL OUTER JOIN. Объединение таблиц клиентов и заказов.

Рассмотрим таблицы Customers (Клиенты) и Orders (Заказы). Мы хотим получить список всех клиентов и всех заказов, включая тех клиентов, у которых нет заказов, и те заказы, которые не привязаны к клиентам.

В этом запросе мы объединяем таблицы Customers и Orders по полю CustomerID.

В результате будут возвращены все строки из обеих таблиц. Если клиент сделал заказ, в строке будет информация и о клиенте, и о заказе. Если клиент не сделал заказ, поля из таблицы Orders будут заполнены значениями NULL. Если заказ не связан с клиентом, поля из таблицы Customers будут заполнены значениями NULL.

UNION и UNION ALL. Различие между UNION и JOIN.

UNION используется для объединения результатов двух или более SELECT-запросов в один результирующий набор данных. Он объединяет строки, при этом удаляет дублирующиеся строки.

Синтаксис:

```
SELECT столбцы1
FROM таблица1
UNION
SELECT столбцы2
FROM таблица2;
```

Особенности:

- Количество столбцов в обоих запросах должно совпадать.
- Типы данных соответствующих столбцов должны быть совместимы.
- Удаляет дубликаты из результирующего набора данных.

UNION ALL работает аналогично UNION, но включает все строки, включая дублирующиеся.

Синтаксис:

```
SELECT столбцы1
FROM таблица1
UNION ALL
SELECT столбцы2
FROM таблица2;
```

Особенности:

- Сохраняет все дубликаты, если они присутствуют.
- Быстрее, чем UNION, так как не требует дополнительной обработки для удаления дубликатов

```sql
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

CREATE TABLE FormerEmployees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);
```

В таблице Employees хранятся данные о текущих сотрудниках.
В таблице FormerEmployees храняться данные о бывших сотрудниках.

```sql
-- UNION: объединение без дубликатов
SELECT FirstName, LastName FROM Employees
UNION
SELECT FirstName, LastName FROM FormerEmployees;

-- UNION ALL: объединение с дубликатами
SELECT FirstName, LastName FROM Employees
UNION ALL
SELECT FirstName, LastName FROM FormerEmployees;
```

UNION объединяет эти результаты и возвращает уникальные записи, т.е. если одно и то же имя и фамилия есть и в текущих, и в бывших сотрудниках, в результатах они появятся только один раз.
UNION ALL объединяет эти результаты и возвращает все записи, включая дубликаты.

Различие между UNION и JOIN

UNION

Что делает:

- Объединяет результаты двух или более SELECT-запросов, возвращая все строки, соответствующие каждому запросу, либо с удалением (UNION), либо без удаления (UNION ALL) дубликатов.
- Используется, когда требуется объединить результаты нескольких запросов по горизонтали (добавить строки в результирующий набор).

Когда использовать:

- Когда нужно объединить два разных набора данных в один.
- Когда требуется получить полный набор данных из разных таблиц или запросов, которые не обязательно связаны между собой напрямую.

JOIN

Что делает:

- Объединяет строки из двух или более таблиц на основе связанного условия (например, совпадение значения в столбцах).
- Объединяет данные по вертикали, добавляя столбцы к результирующему набору.
- Типы JOIN (INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN) определяют, какие строки будут включены в результат на основе условий совпадения.

Когда использовать:

- Когда нужно получить связанные данные из нескольких таблиц на основе общего столбца.
- Когда требуется объединить таблицы по ключевым столбцам для анализа данных, находящихся в отношениях "один-к-одному", "один-ко-многим", или "многие-ко-многим".

Сравнение

Особенность	UNION	JOIN
Назначение	Объединение результатов запросов	Объединение строк на основе условий
Дублирование данных	Удаляет (UNION) или сохраняет (UNION ALL) дубликаты	Данные не дублируются, объединяются строки
Тип объединения	Горизонтальное (добавление строк)	Вертикальное (добавление столбцов)
Связь между таблицами	Не требуется	Требуется, используются ключи и условия
Скорость выполнения	UNION ALL быстрее, UNION медленнее	Зависит от типа JOIN и условий

Заключение

UNION и JOIN — это разные инструменты для объединения данных в SQL.

UNION объединяет результаты запросов по горизонтали, добавляя строки, в то время как JOIN объединяет строки из нескольких таблиц на основе условий совпадения, добавляя столбцы.

Выбор между ними зависит от задачи, которую нужно решить.

Контрольные вопросы:

- Зачем нужны несколько таблиц в базе данных?
- Перечислите основные типы связей между таблицами и приведите примеры для каждой из них.
- Что такое первичный ключ и какие требования к нему предъявляются?
- Какова роль внешнего ключа в реляционной базе данных?
- Объясните разницу между операциями JOIN и UNION. В каких случаях вы бы использовали каждую из них?
- Что происходит при удалении записи в родительской таблице, если существуют связанные записи в дочерней таблице?
- Как можно реализовать связь "многие-ко-многим" между таблицами?
- В чем отличие простого первичного ключа от составного? Приведите примеры.
- Что такое каскадные действия и как они влияют на целостность данных?
- Приведите пример системы, в которой используется несколько таблиц, и опишите их взаимосвязи.

Практика:

https://sql-academy.org/ru/guide