

Тема 11. Агрегирующие функции. Группировки, выборки из нескольких таблиц.

Цель занятия:

Ознакомиться с концепциями работы с несколькими таблицами в базах данных, а также операциями объединения таблиц.

Учебные вопросы:

- 1. Агрегирующие функции**
- 2. Вложенные запросы**
- 3. Оператор GROUP BY для группировки данных**
- 4. Оператор HAVING для фильтрации сгруппированных данных**
- 5. Alias**
- 6. Первичные и внешние ключи**
- 7. Объединение таблиц при помощи JOIN**

1. Агрегирующие функции

Агрегирующие функции в SQL — это функции, которые выполняют вычисления над множеством значений из одного или нескольких столбцов и возвращают одно итоговое значение.

Они позволяют эффективно выполнять операции над группами данных, предоставляя возможность быстро извлекать ключевую информацию, такую как **общие суммы, количество записей, минимальные или максимальные значения и средние показатели.**

Существует 5 таких функций:

- **COUNT():** Подсчет количества строк.
- **SUM():** Суммирование значений.
- **AVG():** Вычисление среднего значения.
- **MIN():** Поиск минимального значения.
- **MAX():** Поиск максимального значения.

1. COUNT(): Подсчитывает количество строк в наборе данных.

Примеры: Подсчет количества сотрудников в таблице Employees

```
SELECT COUNT(*) FROM Employees;
```

Количество завершенных заказов в таблице Orders:

```
SELECT COUNT(*) FROM Orders WHERE Status = 'Completed';
```

2. SUM(): Вычисляет сумму значений в указанном столбце.

Примеры: Сумма всех зарплат сотрудников.

```
SELECT SUM(Salary) FROM Employees;
```

Запрос вычисляет общую сумму заказов клиента с идентификатором 101:

```
SELECT SUM(OrderAmount) FROM Orders WHERE CustomerID = 101;
```

3. AVG(): Вычисляет среднее значение для выбранного столбца.

Примеры: Средняя зарплата сотрудников:

```
SELECT AVG(Salary) FROM Employees;
```

Запрос возвращает средний возраст сотрудников в отделе кадров:

```
SELECT AVG(Age) FROM Employees WHERE Department = 'HR';
```


4. **MIN()**: Определяет минимальное значение в столбце.

Пример: Минимальная зарплата среди сотрудников:

```
SELECT MIN(Salary) FROM Employees;
```

5. MAX(): Определяет максимальное значение в столбце.

Примеры: Максимальная зарплата среди сотрудников:

```
SELECT MAX(Salary) FROM Employees;
```

Запрос возвращает минимальную и максимальную зарплату среди сотрудников IT-отдела:

```
SELECT MIN(Salary), MAX(Salary) FROM Employees WHERE Department = 'IT';
```

2. Вложенные запросы.

Вложенные запросы (subqueries) в SQL — это запросы, которые находятся внутри других запросов.

Они могут быть использованы в различных частях основного запроса, таких как **SELECT**, **FROM**, **WHERE**, **HAVING**, и могут возвращать одиночные значения, строки или целые таблицы данных.

Вложенные запросы позволяют выполнять более сложные и гибкие операции с данными.

1. Вложенные запросы в SELECT

Используются для вычисления значений в столбцах основного запроса.

Пример, вложенный запрос находит название отдела для каждого сотрудника:

```
SELECT EmployeeID,  
       (SELECT DepartmentName  
        FROM Departments  
        WHERE Employees.DepartmentID = Departments.DepartmentID) AS DepartmentName  
FROM Employees;
```

2. Вложенные запросы в WHERE

Используются для фильтрации данных в основном запросе на основе результатов подзапроса.

Пример, запрос находит сотрудников, работающих в отделе кадров:

```
SELECT EmployeeID, FirstName, LastName  
FROM Employees  
WHERE DepartmentID = (SELECT DepartmentID  
                      FROM Departments  
                      WHERE DepartmentName = 'HR');
```

3. Вложенные запросы в FROM

Используются для создания временных таблиц, которые затем могут быть использованы в основном запросе.

Пример, вложенный запрос создает таблицу зарплат для IT-отдела, на основе которой вычисляется средняя зарплата:

```
SELECT AVG(Salary)
FROM (SELECT Salary
      FROM Employees
      WHERE DepartmentID = 1) AS ITDepartmentSalaries;
```

4. Вложенные запросы в HAVING

Используются для фильтрации сгруппированных данных, основанных на агрегированных значениях.

Пример, запрос находит отделы, где количество сотрудников больше среднего по всем отделам:

```
SELECT DepartmentID, COUNT(EmployeeID) AS NumberOfEmployees
FROM Employees
GROUP BY DepartmentID
HAVING COUNT(EmployeeID) > (SELECT AVG(EmployeeCount)
                           FROM (SELECT COUNT(EmployeeID) AS EmployeeCount
                                FROM Employees
                                GROUP BY DepartmentID) AS DepartmentEmployeeCounts);
```

Примеры вложенных запросов:

Нахождение максимальной зарплаты в HR – отделе:

```
SELECT FirstName, LastName, Salary
FROM Employees
WHERE Salary = (SELECT MAX(Salary)
                FROM Employees
                WHERE DepartmentID = (SELECT DepartmentID
                                     FROM Departments
                                     WHERE DepartmentName = 'HR'));
```

Основной запрос выбирает столбцы FirstName, LastName и Salary из таблицы Employees

вложенный запрос вычисляет максимальную зарплату среди всех сотрудников, работающих в отделе кадров.

этот запрос выбирает идентификатор отдела (DepartmentID) из таблицы Departments.

3. Оператор GROUP BY.

Оператор GROUP BY используется для группировки строк в результирующем наборе данных по одному или нескольким столбцам.

Применяется совместно с агрегирующими функциями (например, COUNT, SUM, AVG, MAX, MIN), чтобы выполнять вычисления для каждой группы отдельно.

Синтаксис:

```
SELECT column1, column2, AGGREGATE_FUNCTION(column3)
FROM table_name
WHERE condition
GROUP BY column1, column2
HAVING aggregate_condition;
```

column1, column2: Столбцы, по которым будут группироваться строки.

AGGREGATE_FUNCTION(column3): Агрегирующая функция, выполняющая операцию над значениями столбца column3 для каждой группы.

HAVING aggregate_condition: Опциональная часть, которая позволяет фильтровать группы на основании агрегированных значений.

Группировочные столбцы должны обязательно присутствовать в SELECT!

Пример 1: Группировка по одному столбцу

Предположим, у нас есть таблица Sales, содержащая данные о продажах:

SalesID	ProductID	Quantity	SaleDate
1	101	5	2023-01-01
2	102	3	2023-01-02
3	101	2	2023-01-03
4	103	7	2023-01-04
5	102	6	2023-01-05

Чтобы узнать общее количество проданных единиц для каждого продукта, можно использовать следующий запрос:

```
SELECT ProductID, SUM(Quantity) AS TotalQuantity  
FROM Sales  
GROUP BY ProductID;
```

Результат:

ProductID	TotalQuantity
101	7
102	9
103	7

Этот запрос группирует данные по ProductID и вычисляет сумму количества проданных единиц (SUM(Quantity)) для каждого продукта.

Пример 2: Группировка по нескольким столбцам.

Предположим, что в таблицу Sales добавили еще один столбец Region, и нужно узнать количество проданных единиц для каждого продукта в каждом регионе:

```
SELECT ProductID, Region, SUM(Quantity) AS TotalQuantity  
FROM Sales  
GROUP BY ProductID, Region;
```

Таблица Sales:

SalesID	ProductID	Quantity	SaleDate	Region
1	101	5	2023-01-01	North
2	102	3	2023-01-02	South
3	101	2	2023-01-03	North
4	103	7	2023-01-04	West
5	102	6	2023-01-05	South
6	101	4	2023-01-06	East

Результат запроса:

ProductID	Region	TotalQuantity
101	North	7
101	East	4
102	South	9
103	West	7

4. Оператор **HAVING**.

Оператор **HAVING** работает аналогично **WHERE**, но в отличие от него, применяется не к отдельным строкам, а к группам строк, сформированным **после группировки**.
HAVING.

Синтаксис:

```
SELECT column1, column2, AGGREGATE_FUNCTION(column3)
FROM table_name
WHERE condition
GROUP BY column1, column2
HAVING aggregate_condition;
```

AGGREGATE_FUNCTION(column3): Агрегирующая функция (например, SUM, COUNT), которая применяется к сгруппированным данным.

HAVING aggregate_condition: Условие, накладываемое на результат агрегирующей функции.

Пример 1: Фильтрация групп по агрегированным данным

Предположим, у нас есть таблица Orders, содержащая информацию о заказах:

OrderID	CustomerID	Amount	OrderDate
1	101	150	2023-01-01
2	102	200	2023-01-02
3	101	250	2023-01-03
4	103	300	2023-01-04
5	102	400	2023-01-05

Задача: найти клиентов, у которых общая сумма заказов превышает 300.

Запрос:

```
SELECT CustomerID, SUM(Amount) AS TotalAmount  
FROM Orders  
GROUP BY CustomerID  
HAVING SUM(Amount) > 300;
```

Результат:

CustomerID	TotalAmount
101	400
102	600

Этот запрос группирует заказы по CustomerID, вычисляет общую сумму заказов (SUM(Amount)) для каждого клиента, а затем фильтрует только те группы, где общая сумма заказов превышает 300.

5. Псевдонимы (Alias).

Псевдонимы (Alias) в SQL используются для временного переименования столбцов или таблиц в запросах.

Это позволяет упростить работу с длинными именами, улучшить читаемость запросов или избежать конфликтов имен при объединении таблиц.

1. Псевдонимы для столбцов:

Псевдонимы для столбцов используются для переименования столбцов в результирующем наборе данных. Это может быть полезно, когда вы хотите, чтобы столбец имел более понятное или краткое имя.

Синтаксис:

```
SELECT column_name AS alias_name  
FROM table_name;
```

column_name: имя столбца в таблице.

AS – ключевое слово.

alias_name: имя, которое вы хотите присвоить этому столбцу в выводе.

Пример 1: Псевдоним для столбца

```
SELECT FirstName AS Name, LastName AS Surname  
FROM Employees;
```

В результате выполнения этого запроса, столбец FirstName будет отображаться как Name, а LastName — как Surname.

Результат:

Name	Surname
John	Doe
Alice	Smith

2. Псевдонимы для таблиц:

Псевдонимы для таблиц используются для упрощения работы с длинными именами таблиц или для сокращения запросов при объединении (JOIN) нескольких таблиц.

Синтаксис:

```
SELECT column_name  
FROM table_name AS alias_name;
```

table_name: имя таблицы.

alias_name: псевдоним для таблицы.

Пример 2: Псевдоним для таблицы

```
SELECT e.FirstName, e.LastName, d.DepartmentName  
FROM Employees AS e  
JOIN Departments AS d ON e.DepartmentID = d.DepartmentID;
```

В этом примере Employees обозначена как **e**, а Departments как **d**. Это позволяет использовать сокращенные имена в запросе.

3. Использование псевдонимов без ключевого слова AS:

В SQL допускается использование псевдонимов без ключевого слова AS. Это может сделать код чуть короче, но иногда ухудшает его читаемость.

Пример 3: Псевдонимы без AS

```
SELECT FirstName Name, LastName Surname  
FROM Employees e;
```

Этот запрос даст тот же результат, что и предыдущие примеры с AS.

4. Псевдонимы и вычисляемые поля:

Псевдонимы часто используются для названия вычисляемых полей.

Пример 4: Псевдонимы для вычисляемых столбцов

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

Здесь AnnualSalary — это псевдоним для столбца, который рассчитывает годовую зарплату на основе столбца Salary.

Результат:

FirstName	LastName	AnnualSalary
John	Doe	72000
Alice	Smith	84000

Ключевые моменты:

- Псевдонимы улучшают читаемость SQL-запросов, особенно в случаях длинных имен столбцов или таблиц.
- AS не является обязательным, но его использование повышает ясность кода.
- Псевдонимы полезны при объединении таблиц, создании вычисляемых полей и форматировании вывода данных.

6. 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)  
);
```



```
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.

Primary key

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

<u>id</u>	name	gpa
1	Egor	4.25
2	Egor	3.82
3	Egor	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).

```
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.

```
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.

7. 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

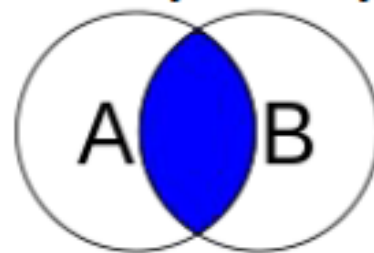
JOIN – оператор, предназначенный для объединения таблиц по определенному столбцу или связке столбцов (как правило, по первичному ключу).

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:

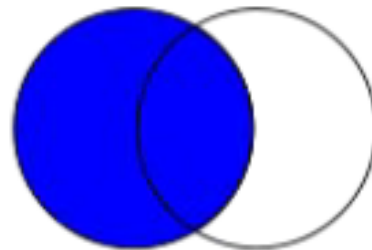
```
SELECT tables_columns FROM table_1  
JOIN table_2 ON table_1.column = table_2.column;
```

Типы JOIN

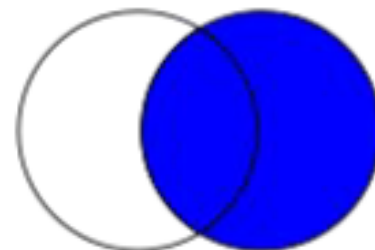
```
SELECT <fields>  
FROM TableA A  
INNER JOIN TableB B  
ON A.key = B.key
```



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

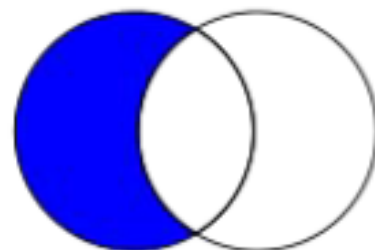


```
SELECT <fields>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.key = B.key
```

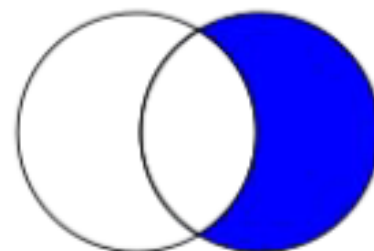


SQL JOINS

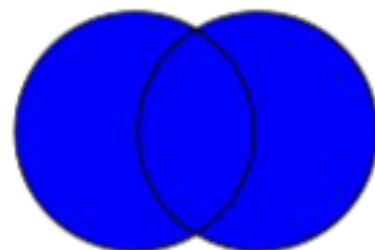
```
SELECT <fields>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.key = B.key  
WHERE B.key IS NULL
```



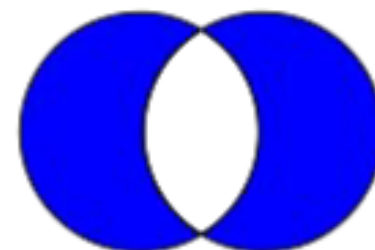
```
SELECT <fields>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.key = B.key  
WHERE A.key IS NULL
```



```
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

```
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.

```
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.


```
-- 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

Правая таблица

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

* Код 999 - для удаленщиков

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

```
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.

```
-- 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

Правая таблица

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

* Код 999 - для удаленщиков

Получаем только удаленщиков.

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

```
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.

```
-- 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

Правая таблица

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

* Код 999 - для удаленщиков

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

```
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

```
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 хранятся данные о бывших сотрудниках.

-- 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.

Домашнее задание:

Задание 1: Агрегирующие функции

Создайте таблицу sales с полями:

- id (INT, PRIMARY KEY, AUTO_INCREMENT)
- product_name (VARCHAR)
- quantity (INT)
- price (DECIMAL)
- sale_date (DATE)

Заполните таблицу данными (минимум 10 записей).

Напишите запросы:

- Найдите общее количество проданных товаров.
- Найдите среднюю цену товара.
- Найдите максимальную и минимальную цену товара.
- Найдите общую сумму продаж за все время.

Задание 2: Группировка данных

Используя таблицу `sales`, напишите запросы:

- Сгруппируйте данные по товарам (`product_name`) и найдите общее количество проданных единиц для каждого товара.
- Сгруппируйте данные по дате (`sale_date`) и найдите общую сумму продаж за каждый день.
- Найдите среднее количество проданных товаров в день.

Задание 3: Выборки из нескольких таблиц

Создайте две таблицы:

employees:

- id (INT, PRIMARY KEY, AUTO_INCREMENT)
- name (VARCHAR)
- department_id (INT)

departments:

- id (INT, PRIMARY KEY, AUTO_INCREMENT)
- department_name (VARCHAR)

Заполните таблицы данными (минимум 5 сотрудников и 3 отдела).

Напишите запросы:

- Выведите список всех сотрудников с указанием их отдела.
- Найдите количество сотрудников в каждом отделе.
- Выведите отделы, в которых работает больше 2 сотрудников.

Задание 4: Комбинированные запросы

Создайте таблицу orders:

- id (INT, PRIMARY KEY, AUTO_INCREMENT)
- customer_id (INT)
- order_date (DATE)
- total_amount (DECIMAL)

Создайте таблицу customers:

- id (INT, PRIMARY KEY, AUTO_INCREMENT)
- name (VARCHAR)
- city (VARCHAR)

Заполните таблицы данными (минимум 5 заказов и 3 клиента).

Напишите запросы:

- Выведите список всех заказов с указанием имени клиента и города.
- Найдите общую сумму заказов для каждого клиента.
- Найдите клиентов, у которых общая сумма заказов превышает 1000.

Задание 5: Практика с JOIN

Используя таблицы `orders` и `customers`, напишите запросы:

- Выведите список заказов, сделанных клиентами из определенного города (например, "Москва").
- Найдите среднюю сумму заказа для каждого клиента.
- Выведите клиентов, которые не сделали ни одного заказа (используйте `LEFT JOIN` и `IS NULL`).

Задание 6: Сложная группировка

Создайте таблицу products:

- id (INT, PRIMARY KEY, AUTO_INCREMENT)
- product_name (VARCHAR)
- category (VARCHAR)

Создайте таблицу sales:

- id (INT, PRIMARY KEY, AUTO_INCREMENT)
- product_id (INT)
- quantity (INT)
- sale_date (DATE)

Заполните таблицы данными (минимум 5 товаров и 10 продаж).

Напишите запросы:

- Сгруппируйте данные по категориям товаров и найдите общее количество проданных товаров в каждой категории.
- Найдите категорию с наибольшим количеством продаж.
- Выведите товары, которые не были проданы ни разу (используйте LEFT JOIN и IS NULL).

Список литературы:

1. [Руководство по MySQL.](#)
2. [Видеокурс.](#)

Материалы лекций:

<https://github.com/ShViktor72/Education>

Обратная связь:

colledge20education23@gmail.com