# *УПРАВЛЕНИЕ ДАННЫМИ В SELECT: ФИЛЬТРАЦИЯ, ГРУППИРОВКА, СОРТИРОВКА*

Оператор SELECT имеет следующую структуру:

```
SELECT
  [DISTINCT | DISTINCTROW | ALL]
  select_expression,...
FROM table_references
[WHERE where_definition]
[GROUP BY {unsigned_integer | col_name | formula}]
[HAVING where_definition]
[ORDER BY {unsigned_integer | col_name | formula}
[ASC | DESC], ...]
```

Самый простой запрос:

```
SELECT * FROM Table
```

## *Использование WHERE для фильтрации данных*

Оператор WHERE в SQL обеспечивает фильтрацию данных на основе заданных условий, что позволяет извлекать, обновлять или удалять именно те данные, которые соответствуют определенным критериям.

Если бы оператора WHERE не существовало, нужно было бы извлекать все данные из таблицы и затем вручную, например, в приложении, фильтровать их для выполнения определенных задач. Это было бы очень неэффективно, особенно для больших баз данных.

WHERE можно использовать с различными операторами сравнения, такими как равно (=), не равно (<>), больше (>), меньше (<), больше или равно (>=), меньше или равно (<=), а также более специализированными, вроде, BETWEEN, который позволяет указать диапазон значений, LIKE, предназначенный для поиска по шаблону, и IN, который дает возможность выбрать данные из определенного набора.

Рассмотрим несколько примеров использования WHERE для фильтрации данных.

Применение оператора WHERE с использованием условия равенства (=):

```
SELECT *
FROM Book WHERE PublicationYear = 2025
```

В этом случае оператор равенства используется для выбора записи, где идентификатор год выпуска книги точно соответствует 2025. Это простой эквивалент оператора равенства в математике.

Использование WHERE с операторами больше (>) или меньше (<):

```
SELECT *
FROM Book WHERE PublicationYear > 2000
```

Здесь используется оператор >, благодаря которому запрос отсеивает ненужное и возвращает нам данные о книгах, выпущенных после 2000 года. Этот оператор может быть полезным, если необходимо найти записи, которые обладают каким-то значением выше или ниже определенного порога.

Пример использования оператор WHERE с BETWEEN:

```
SELECT *
FROM Book WHERE PublicationYear BETWEEN 1900 AND 1950
```

BETWEEN позволяет выбрать записи, попадающие в определенный диапазон значений. В нашем случае, это все книги, год выпуска которых находится в интервале от 2000 до 2100 года включительно. Это полезно, когда нужен четкий диапазон значений, которые необходимо извлечь.

Оператор WHERE с использованием LIKE и символов подстановки:

```
SELECT *
FROM Book WHERE Title LIKE '%ть%'
```

Оператор LIKE используется для поиска данных по шаблонам. В базе данных SQL для обозначения шаблонов применяются два символа подстановки: % заменяет ноль или больше символов, символ нижнего подчеркивания _ — заменяет ровно один символ. Так, в нашем конкретном случае, запрос вернет все записи из таблицы Book, где в названии книги присутствуют символы «ть». Этот подход часто используется, когда требуется найти данные, точное значение которых не известно, или когда необходимо найти несколько совпадений.

## *Исключение повторов в атрибутах*

Можно использовать SELECT для выбора лишь уникальных значений определенного столбца, исключая повторяющиеся записи, что особенно полезно при анализе данных:

```
SELECT DISTINCT Nationality
FROM Author
```

## *Использование операторов AND, OR и NOT*

AND, OR и NOT являются ключевыми логическими операторами в SQL. Они используются для комбинирования или инвертирования условий в операторах SQL, таких как WHERE, HAVING и др.

Оператор **AND** используется для создания запроса, который возвращает true (истину), только когда оба сравниваемых условия являются истинными. Например:

```
SELECT * FROM Author WHERE  BirthYear > 1850 AND Nationality='русский'
```

В данном случае оператор AND связывает вместе два критерия отбора: год рождения и национальность автора. Результатом выполнения такого запроса станут записи из таблицы, которые удовлетворяют обоим условиям одновременно.

Оператор **OR** возвращает true, если хотя бы одно из условий оказывается истинным:

```
SELECT * FROM Book WHERE BETWEEN 1900 AND 1950 OR PublicationYear > 2000
```

Здесь оператор OR соединяет два условия отбора. Запрос выведет те записи из таблицы Book, в которых указано, что книга издана в период [1900..1950] годов или в 2000 году или позже.

Оператор **NOT** меняет логическое значение условия на противоположное, возвращая true, если условие неверно, и false (ложь), если условие верно.

```
SELECT * FROM Author WHERE NOT (Nationality='русский');
```

В этом запросе оператор NOT инвертирует условие Nationality='русский'. Запрос вернет все строки из таблицы Author с теми авторами, 'русский'. Это позволяет формировать запросы, исключающие определенные категории данных.

Эти логические операторы можно использовать в любых комбинациях для создания сложных условий.

## *Группировка данных с помощью GROUP BY*

Оператор GROUP BY используется для группировки строк в результирующем наборе по значениям определенного столбца или группе столбцов.

Пример:

```
SELECT Title, COUNT(BookID)
FROM Book
GROUP BY Book.Author;

# Получаем результаты
results = cursor.fetchall() # Выводим на экран for row in results:
print(f"Author: {row[0]}, Количество книг: {row[1]}")
```

В этом примере подсчитывается общее количество книг (BookID), написанных каждым отдельным автором (Book.Author).

Оператор GROUP BY особенно полезен в сочетании с агрегатными функциями, такими как COUNT(), SUM(), AVG(), MAX(), MIN().

Например:

```
SELECT
    -- Общая сумма и среднее
    (SELECT SUM(Amount) FROM Sales) AS TotalSum,
    (SELECT AVG(Amount) FROM Sales) AS TotalAvg,
    -- Сумма и среднее по клиентам
    ClientID,
    SUM(Amount) AS GroupSum,
    AVG(Amount) AS GroupAvg
FROM Sales
GROUP BY ClientID;
```

- *Подзапросы* (SELECT SUM(Amount) FROM Sales) и (SELECT AVG(Amount) FROM Sales) вычисляют агрегированные значения для всей таблицы.
- GROUP BY ClientID группирует данные по каждому клиенту и рассчитывает агрегаты (SUM() и AVG()) для каждой группы.
- В одном запросе вычисляются агрегированные данные для всей таблицы, и агрегированные данные по каждой группе.

И как это будет выглядеть на Питоне (желающие могут попробовать, хотя в базовые программы это задание не входит):

```
# Запрос с подзапросами для общих значений и агрегированных по группам
query = """
SELECT
    (SELECT SUM(Amount) FROM Sales) AS TotalSum,
    (SELECT AVG(Amount) FROM Sales) AS TotalAvg,
    ClientID,
    SUM(Amount) AS GroupSum,
    AVG(Amount) AS GroupAvg
FROM Sales
GROUP BY ClientID
"""

# Выполним запрос и получаем результат
cursor.execute(query)
results = cursor.fetchall()

# Извлекаем общие значения из первой строки
total_sum, total_avg = results[0][0], results[0][1]

# Выводим общие значения (они одинаковы для всех строк)
print(f"Общие значения: TotalSum = {total_sum}, TotalAvg = {total_avg}")

# Выводим результаты для каждой группы
print("\nАгрегированные значения по группам:")
```

```
for row in results:
    client_id, group_sum, group_avg = row[2], row[3], row[4]
    print(f"ClientID: {client_id}, GroupSum: {group_sum}, GroupAvg: {group_avg}")
```

1. Первоначальный запрос выполняется для получения всех необходимых данных (как общих, так и по группам).
2. Извлечение общих значений из первой строки результата (total_sum и total_avg), так как они одинаковы для всех групп.
3. После этого, мы выводим общие значения, а затем в цикле перебираем остальные строки результата, выводя данные по каждому клиенту.

Пример вывода:

```
Общие значения: TotalSum = 1000, TotalAvg = 200.0
Агрегированные значения по группам:
ClientID: 1, GroupSum: 300, GroupAvg: 150.0
ClientID: 2, GroupSum: 400, GroupAvg: 200.0
ClientID: 3, GroupSum: 300, GroupAvg: 300.0
```

Таким образом, общие значения для всей таблицы (`TotalSum` и `TotalAvg`) будут выведены только один раз, а результаты для каждой группы будут выводиться отдельно.

*Использование нескольких столбцов в GROUP BY*

В GROUP BY можно использовать не один столбец, а несколько. Это позволяет группировать данные по нескольким критериям.

Пример:

```
SELECT Country, Department, COUNT(EmployeeID)
FROM Employees
GROUP BY Country, Department
```

Здесь данные группируются сначала по стране, а затем по отделу.

*Группировка с вычислениями*

Иногда полезно использовать вычисления при группировке, например, группировка по диапазонам значений (например, возрастные группы).

Пример:

```
SELECT
    CASE
        WHEN Age < 18 THEN 'Under 18'
        WHEN Age BETWEEN 18 AND 35 THEN '18-35'
        WHEN Age BETWEEN 36 AND 50 THEN '36-50'
        ELSE 'Above 50'
    END AS AgeGroup,
    COUNT(*) AS Count
FROM People
GROUP BY AgeGroup
```

Здесь используется оператор CASE для создания возрастных групп.

– CASE проверяет значения в столбце Age и для каждого человека присваивает его возрастной группе.
– COUNT(*) подсчитывает количество людей, попавших в каждую возрастную группу.
– GROUP BY AgeGroup группирует результаты по созданным возрастным группам, таким образом, вы получаете один итог по каждой группе.

И как это будет выглядеть на Питоне (желающие могут попробовать, хотя в базовые программы это задание не входит):

```
# Запрос на группировку и подсчет
cursor.execute("
SELECT
    CASE
        WHEN Age < 18 THEN 'Under 18'
        WHEN Age BETWEEN 18 AND 35 THEN '18-35'
        WHEN Age BETWEEN 36 AND 50 THEN '36-50'
        ELSE 'Above 50'
    END AS AgeGroup,
    COUNT(*) AS Count
FROM People
GROUP BY AgeGroup")

# Получение и вывод результатов
results = cursor.fetchall()
print("Age Group Distribution:")
for row in results:
    print(f"AgeGroup: {row[0]}, Count: {row[1]}")
```

Пример вывода:

```
Age Group Distribution:
AgeGroup: Under 18, Count: 2
AgeGroup: 18-35, Count: 3
AgeGroup: 36-50, Count: 1
AgeGroup: Above 50, Count: 1
```

*Порядок результатов после группировки*

Хотя GROUP BY определяет, как будет производиться группировка, важно также знать, как можно упорядочить результаты после группировки с помощью ORDER BY. Это не обязательный элемент для работы с группировкой, но он полезен для сортировки итоговых агрегатов.

Пример группировки данных по ClientID:

```
SELECT ClientID, SUM(Amount) AS TotalAmount
FROM Sales
GROUP BY ClientID
ORDER BY TotalAmount DESC
```

– Для каждой группы (каждого клиента) подсчитывает общую сумму покупок SUM(Amount).
– Сортирует результаты по полученной общей сумме TotalAmount в порядке убывания (от наибольшей суммы к наименьшей).

И как это будет выглядеть на Питоне (желающие могут попробовать, хотя в базовые программы это задание не входит):

```
# SQL-запрос для подсчета общей суммы покупок по каждому клиенту и сортировки
cursor.execute("
SELECT ClientID, SUM(Amount) AS TotalAmount
FROM Sales
GROUP BY ClientID
ORDER BY TotalAmount DESC")

results = cursor.fetchall()

# Вывод результатов
print("ClientID и общая сумма покупок (TotalAmount):")
for row in results:
    print(f"ClientID: {row[0]}, TotalAmount: {row[1]}")
```

Пример вывода:

```
ClientID и общая сумма покупок (TotalAmount):
ClientID: 105, TotalAmount: 15,230.50
ClientID: 203, TotalAmount: 12,890.00
ClientID: 157, TotalAmount: 9,765.30
ClientID: 302, TotalAmount: 8,450.75
ClientID: 120, TotalAmount: 6,980.25
```

Этот список отсортирован по убыванию суммы покупок (TotalAmount).

## *Фильтрация агрегированных данных с помощью HAVING*

В SQL операторы **GROUP BY** и **HAVING** часто используются вместе для агрегации данных и вычисления разнообразных статистических показателей на основе группировки данных по заранее определенным критериям.

В отличие от WHERE, который применяется до агрегирования, HAVING применяется после того, как данные были сгруппированы. Например, если нужно вывести только те группы, где сумма покупок превышает определенный порог, можно использовать HAVING.

Пример:

```
SELECT ClientID, SUM(Amount) AS TotalAmount
FROM Sales
GROUP BY ClientID
HAVING SUM(Amount) > 500
```

Здесь:

- GROUP BY ClientID группирует данные по каждому клиенту.
- SUM(Amount) вычисляет сумму по каждому клиенту.
- HAVING SUM(Amount) > 500 фильтрует группы, оставляя только те, где сумма покупок больше 500.
- HAVING применяется после того, как данные были сгруппированы и агрегированы, в отличие от WHERE, который фильтрует данные до группировки.

Оператор HAVING позволяет ограничить результаты, основываясь на агрегированных показателях каждой группы – таких как сумма, среднее значение, максимальное или минимальное значение.. Это важно, потому что WHERE не может фильтровать агрегированные данные.

Пример с агрегатной функцией и HAVING. Предположим, мы хотим выбрать только тех клиентов, у которых средний чек превышает 200:

```
SELECT ClientID, AVG(Amount) AS AverageAmount
FROM Sales
GROUP BY ClientID
HAVING AVG(Amount) > 200
```

Здесь:

– AVG(Amount) вычисляет средний чек для каждого клиента.
– HAVING AVG(Amount) > 200 оставляет только те группы, где средний чек больше 200.

Заметьте, что HAVING применяется в SQL-запросах **исключительно после использования GROUP BY**. Нельзя использовать HAVING без предварительной группировки данных с помощью GROUP BY.

Операторы GROUP BY и HAVING являются неотъемлемыми инструментами для агрегации данных в SQL. Их использование дает возможности для широкого анализа данных, позволяя не только собирать статистические данные, но и выявлять в них определенные закономерности, тренды и паттерны.

## *Сортировка результатов запроса*

Оператор **ORDER BY** в SQL используется для сортировки результатов запроса. Это позволяет упорядочить строки в результирующем наборе данных в определенном порядке: по возрастанию (по умолчанию) или по убыванию.

– ASC (англ. ASCENDING) – сортировка по возрастанию (по умолчанию).
– DESC (англ. DESCENDING)– сортировка по убыванию.

Пример:

```
SELECT ClientID, Amount
FROM Sales
ORDER BY Amount DESC
```

Этот запрос сортирует результаты по столбцу Amount в порядке убывания (DESC – от большего к меньшему). В результате получается ь список клиентов и суммы их отдельных покупок. Самые крупные покупки будут в начале списка, а самые маленькие – в конце.

Пример с несколькими столбцами:

Если нужно отсортировать данные сначала по одному столбцу, а потом по другому столбцу, можно использовать несколько столбцов в ORDER BY.

```
SELECT ClientID, Amount, Date
FROM Sales
ORDER BY Date ASC, Amount DESC
```

Здесь сначала сортируются данные по дате Date в порядке возрастания (ASC можно и не указывать, т.к. по умолчанию сортировка и так производится по возрастанию), а затем, если даты совпадают, данные сортируются по сумме Amount в порядке убывания.

Пример с сортировкой строк по числовым и строковым значениям:

```
SELECT Name, Age
FROM People
ORDER BY Age ASC, Name DESC;
```

Сначала строки сортируются по возрасту Age по возрастанию. Если возраст одинаков, то по имени Name — по убыванию.

Этот оператор должен быть последним в SQL-запросе, после всех фильтров и группировок, т.е. ORDER BY всегда идет после GROUP BY, HAVING, WHERE.

## *Порядок выполнения SQL-запроса*

Обратите внимание, что ключевое слово SELECT указывается первым в запросе, но выборка происходит на предпоследнем этапе обработки запроса.

Вот порядок выполнения SQL-запроса:

FROM – указывается источник данных, то есть таблица или несколько таблиц, которые мы собираемся использовать.

JOIN (если используется) — если требуется объединить несколько таблиц, то это делается после указания источника.

WHERE – фильтрация строк перед группировкой.

GROUP BY – группировка строк по определенным столбцам.

HAVING – фильтрация групп после группировки. Это условие применяется к уже сгруппированным данным.

SELECT – выборка столбцов для отображения.

ORDER BY – сортировка окончательных результатов после выполнения всех других операций.

Пример:

```
SELECT ClientID, COUNT(OrderID)
FROM Orders
WHERE OrderDate >= '2023-01-01'  -- 1. WHERE
GROUP BY ClientID                -- 2. GROUP BY
HAVING COUNT(OrderID) > 5        -- 3. HAVING
ORDER BY COUNT(OrderID) DESC;    -- 4. ORDER BY
```

Пояснение:

- FROM: Указываем таблицу Orders, откуда выбираются данные.
- WHERE: Фильтруем данные, выбираем только заказы с датой позже '2023-01-01'.
- GROUP BY: Группируем данные по ClientID.
- HAVING: Применяем фильтрацию уже после группировки, показываем только те группы, где количество заказов больше 5.
- SELECT: Определяем, какие столбцы показывать — в данном случае это ClientID и количество заказов.
- ORDER BY: Сортируем результаты по убыванию количества заказов.

Применение условий фильтрации через WHERE происходит до группировки, что позволяет сократить объем обрабатываемых данных. Условия, определенные в HAVING, применяются к уже сформированным группам данных, что дает возможность провести более детальный анализ.

Собственно SELECT всегда исполняется после FROM, WHERE, GROUP BY, HAVING и ORDER BY. Этот порядок операций отражает логику обработки запросов в SQL.

SELECT не извлекает данные напрямую из исходной таблицы, а выбирает указанные столбцы из промежуточного результирующего набора данных, полученного после выполнения операторов FROM, JOIN, WHERE, GROUP BY и HAVING.

На момент выполнения SELECT промежуточный результат может содержать избыточные данные, но SELECT формирует окончательный набор, оставляя только нужные столбцы.