Мадока ВТ Вики

БД:Теория:Глава 4

В предыдущих главах мы рассмотрели, как спроектировать и создать структуру базы данных с помощью ER-моделей, нормализации и DDL-команд SQL (`CREATE TABLE`, `ALTER TABLE` и т.д.). Теперь перейдем к самой частой задаче при работе с БД — **извлечению данных** с помощью DML-команды `SELECT`.

Часть 1: Структура Запроса `SELECT`

Команда `SELECT` используется для выборки данных из одной или нескольких таблиц. Мы уже видели базовую структуру, но теперь рассмотрим ее полнее.

Логический порядок выполнения (как СУБД “думает” о запросе):

1. **`FROM`**: Определяет таблицы-источники данных. Если таблиц несколько, на этом этапе (логически) формируется их комбинация (например, декартово произведение или результат JOIN).
2. **`WHERE`**: Фильтрует строки, полученные на шаге `FROM`, оставляя только те, которые удовлетворяют заданному условию.
3. **`GROUP BY`**: Группирует строки, прошедшие фильтрацию `WHERE`, по одинаковым значениям в указанных столбцах. Все строки с одинаковыми значениями в этих столбцах “схлопываются” в одну строку группы.
4. **Агрегатные функции**: Вычисляются для каждой группы, созданной на шаге `GROUP BY` (или для всей таблицы, если `GROUP BY` отсутствует).
5. **`HAVING`**: Фильтрует *группы*, созданные на шаге `GROUP BY`, оставляя только те группы, которые удовлетворяют условию в `HAVING` (условие часто включает агрегатные функции).
6. **`SELECT`**: Выбирает столбцы (или вычисляет выражения), которые нужно вернуть в результате. Если использовался `GROUP BY`, здесь можно указывать только столбцы, по которым шла группировка, агрегатные функции или константы.
7. **`DISTINCT`**: (Если указано) Удаляет дублирующиеся строки из результирующего набора.
8. **`ORDER BY`**: Сортирует финальный результирующий набор строк по указанным столбцам.
9. **`LIMIT` / `OFFSET`**: (Нестандартные, но есть в PostgreSQL и многих других СУБД) Ограничивают количество возвращаемых строк и/или пропускают начальные строки.

Важно: Физический порядок выполнения СУБД может отличаться! Оптимизатор может переставлять операции (например, применять условия `WHERE` до полного формирования JOIN) для повышения эффективности, но *логический* результат должен оставаться таким же.

Пример полной структуры:

```
SELECT -- 6. Выбираем столбцы/выражения
    column1,
    AGG_FUNC(column2) AS aggregate_result
FROM -- 1. Указываем таблицу(ы)
    my_table
```

```
WHERE -- 2. Фильтруем строки
    column3 > 10
GROUP BY -- 3. Группируем строки
    column1
HAVING -- 5. Фильтруем группы
    AGG_FUNC(column2) > 100
ORDER BY -- 8. Сортируем результат
    aggregate_result DESC
LIMIT 10; -- 9. Ограничиваем вывод
```

Часть 2: Операторы и Предикаты в `WHERE` и `HAVING`

Условия в `WHERE` (фильтрация строк) и `HAVING` (фильтрация групп) строятся с использованием операторов и предикатов.

Логические операторы:

- `AND`: Логическое И.
- `OR`: Логическое ИЛИ.
- `NOT`: Логическое НЕ.

Работа с `NULL`: Вспомним таблицы истинности (из Лекции 5 / Начала Лекции 6):

- `TRUE AND NULL` -> `NULL`
- `FALSE AND NULL` -> `FALSE`
- `TRUE OR NULL` -> `TRUE`
- `FALSE OR NULL` -> `NULL`
- `NOT NULL` -> `NULL`
- `NULL AND NULL` -> `NULL`
- `NULL OR NULL` -> `NULL`

Помните, что в `WHERE` и `HAVING` условие, дающее `NULL` или `FALSE`, приводит к тому, что строка/группа **не** включается в результат.

Операторы сравнения:

- =, >, <, >=, <=, <> (или `!=`)
- При сравнении с `NULL` результат всегда `NULL`.

Предикаты сравнения:

- `expression BETWEEN x AND y`: Проверяет, находится ли значение `expression` в диапазоне `[x, y]` (включая границы). Эквивалентно `expression >= x AND expression <= y`.
- `expression NOT BETWEEN x AND y`: Обратная проверка.
- `expression IS NULL`: Проверяет, равно ли значение `NULL`. **Правильный способ проверки на `NULL`!**
- `expression IS NOT NULL`: Проверяет, не равно ли значение `NULL`.
- `expression IS DISTINCT FROM value`: Сравнивает `expression` и `value`. Равно `FALSE`, если оба равны *или оба* *`NULL`*. Равно `TRUE` в противном случае. Похоже на <>, но корректно работает с `NULL`.
- `expression IS NOT DISTINCT FROM value`: Сравнивает `expression` и `value`. Равно `TRUE`, если оба равны *или оба* *`NULL`*. Равно `FALSE` в противном случае. Похоже на =, но корректно работает с `NULL`.

Пример использования `IS NULL`:

```
-- Найти студентов без указанной группы
SELECT st_name FROM students WHERE gr_id IS NULL;
```

Поиск по шаблону:

- `string LIKE pattern`: Проверяет, соответствует ли строка `string` шаблону `pattern`.
  - `%`: Соответствует любой последовательности символов (включая пустую).
  - `_`: Соответствует ровно одному любому символу.
  - `ESCAPE 'char'`: Позволяет экранировать символы `%` и `_`, если их нужно искать как обычные символы.
  - **Пример:** `st_name LIKE 'Иван%'` (имена, начинающиеся на “Иван”).
  - **Пример:** `st_name LIKE '_ва%'` (имена, где второй символ “в”).
- `string ILIKE pattern`: То же, что `LIKE`, но без учета регистра (I - Insensitive). Это расширение PostgreSQL.
- `string SIMILAR TO pattern`: Более мощный стандартный SQL-вариант для поиска по шаблону, использующий синтаксис, похожий на регулярные выражения SQL (отличается от POSIX). Редко используется на практике по сравнению с `LIKE` или POSIX regex.
- **POSIX Регулярные выражения (PostgreSQL):** Предоставляют наиболее мощные средства поиска по шаблонам.
  - `string ~ pattern`: Соответствует регулярному выражению POSIX (с учетом регистра).
  - `string ~* pattern`: Соответствует регулярному выражению POSIX (без учета регистра).
  - `string !~ pattern`: Не соответствует (с учетом регистра).
  - `string !~* pattern`: Не соответствует (без учета регистра).
  - **Пример:** `st_name ~* '^иван'` (имена, начинающиеся на “иван” без учета регистра).
  - **Пример:** `email ~ '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'` (проверка формата email).

Условные выражения `CASE`:

Позволяют выполнять ветвление внутри SQL-запроса. Аналог `if/then/else` или `switch`.

Синтаксис 1 (Простой):

```
CASE выражение
    WHEN значение1 THEN результат1
    WHEN значение2 THEN результат2
    ...
    [ ELSE результат_по_умолчанию ]
END
```

Сравнивает `выражение` с `значение1`, `значение2` и т.д.

Синтаксис 2 (Общий):

```
CASE
    WHEN условие1 THEN результат1
```

```
    WHEN условие2 THEN результат2
    ...
    [ ELSE результат_по_умолчанию ]
END
```

Проверяет `условие1`, `условие2` и т.д. по порядку и возвращает результат для первого истинного условия.

- Если ни одно условие не истинно и нет `ELSE`, возвращается `NULL`.
- `CASE` можно использовать везде, где допустимо выражение (в `SELECT`, `WHERE`, `ORDER BY` и т.д.).

Пример (из Лекции 5, слайд 14): Преобразование оценки из баллов в текстовое представление.

```
-- Исходные данные (предположим)
SELECT Result FROM EXAM;
 Result
--------
     76
     95
     55

-- Запрос с CASE
SELECT Result,
    CASE
        WHEN Result >= 91 THEN 'Отл.'
        WHEN Result >= 75 THEN 'Хор.'
        WHEN Result >= 60 THEN 'Удовл.'
        ELSE 'Неуд.' -- Добавим вариант для < 60
    END AS Description
FROM EXAM;
```

Вывод:

```
 result | description
--------+-------------
     76 | Хор.
     95 | Отл.
     55 | Неуд.
(3 rows)
```

Преобразование типов:

Иногда нужно явно преобразовать значение из одного типа в другой.

- **`CAST ( expression AS type )`**: Стандартный SQL синтаксис.
- **`expression::type`**: Синтаксис PostgreSQL (более короткий и часто используемый).
- **`typename ( expression )`**: Исторический синтаксис через функцию (менее предпочтителен).

```
-- Преобразовать целое число 42 в число с плавающей точкой двойной точности
SELECT CAST(42 AS DOUBLE PRECISION);
SELECT 42::DOUBLE PRECISION;
SELECT float8(42); -- float8 это имя типа double precision в PostgreSQL
```

Вывод (для всех трех вариантов):

```
 float8
--------
```

```
      42
(1 row)
```

Преобразование возможно не всегда и зависит от совместимости типов.

Оператор `IN` (со списком значений):

Проверяет, присутствует ли значение выражения в заданном списке констант.

```
expression IN (value1, value2, ...)
```

Эквивалентно: `(expression = value1) OR (expression = value2) OR ...`

```
-- Найти студентов из групп P3100 или P3101
SELECT st_name FROM students WHERE group_name IN ('P3100', 'P3101');
```

Часть 3: Запросы с использованием Нескольких Таблиц (Соединения - `JOIN`)

Чаще всего нужная информация распределена по нескольким связанным таблицам. Чтобы получить ее в одном результате, таблицы нужно **соединить**.

Декартово Произведение (Cartesian Product):

- Комбинация *каждой* строки одной таблицы с *каждой* строкой другой таблицы.
- Число строк в результате = (число строк в табл. 1) * (число строк в табл. 2).
- В SQL получается, если перечислить таблицы в `FROM` через запятую без условия соединения в `WHERE` или использовать `CROSS JOIN`.
- **Обычно не является желаемым результатом**, так как создает много “бессмысленных” комбинаций строк и очень ресурсоемко.

```
-- Таблица lt
 id | name
----+------
  1 | aaa
  2 | bbb
  3 | ccc

-- Таблица rt
 id | value
----+-------
  1 | xxx
  3 | yyy
  7 | zzz

-- Декартово произведение
SELECT * FROM lt, rt;
-- или
SELECT * FROM lt CROSS JOIN rt;
```

Вывод (Декартово произведение):

```
 id | name | id | value
----+------+----+-------
  1 | aaa  |  1 | xxx
  1 | aaa  |  3 | yyy
  1 | aaa  |  7 | zzz
```

```
   2 | bbb  |  1 | xxx
   2 | bbb  |  3 | yyy
   2 | bbb  |  7 | zzz
   3 | ccc  |  1 | xxx
   3 | ccc  |  3 | yyy
   3 | ccc  |  7 | zzz
(9 rows)
```

Соединение (`JOIN`):

Правильный способ комбинировать строки из связанных таблиц на основе **условия соединения**.

Старый стиль (через `WHERE`) - НЕ РЕКОМЕНДУЕТСЯ:

```sql
-- Найти студентов из России (пример из Лекции 4)
SELECT s.Name, s.City
FROM STUDENT s, CITIES c -- Таблицы через запятую
WHERE c.CityName = s.City -- Условие соединения
  AND c.Country = 'Россия'; -- Условие фильтрации
```

Этот стиль смешивает условия соединения и условия фильтрации в `WHERE`, что ухудшает читаемость и может приводить к ошибкам (если забыть условие соединения, получится декартово произведение).

Современный стиль (явный `JOIN`) - РЕКОМЕНДУЕТСЯ:

Используются ключевые слова `JOIN`, `ON`, `USING`, `NATURAL JOIN`. Условия соединения отделены от условий фильтрации (`WHERE`).

Базовый синтаксис:

```sql
SELECT ...
FROM table1
[ join_type ] JOIN table2
  ON join_condition -- Условие для связи строк table1 и table2
[ WHERE filter_condition ]
...
```

Типы `JOIN`:

1. **`INNER JOIN` (или просто `JOIN`):**
   - Возвращает только те строки, для которых **найдено соответствие** в *обеих* таблицах согласно условию `ON`. Строки, не имеющие пары в другой таблице, в результат не попадают.
   - Это самый частый тип соединения.
   - Коммутативен (`A JOIN B` эквивалентно `B JOIN A`).
   - Ассоциативен (`(A JOIN B) JOIN C` эквивалентно `A JOIN (B JOIN C)`).

   Пример: Получить имена студентов и названия их групп.

```sql
SELECT s.st_name, g.gr_name
FROM students s
INNER JOIN groups g ON s.gr_id = g.gr_id; -- Условие соединения по внешнему ключу
```

Студенты, у которых `gr_id` равен `NULL` или ссылается на несуществующую группу, в результат не попадут.

2. **`LEFT OUTER JOIN` (или `LEFT JOIN`):**

   - Возвращает **все строки** из *левой* таблицы (`table1`).
   - Для каждой строки из левой таблицы ищется соответствие в *правой* таблице (`table2`) по условию `ON`.
   - Если соответствие найдено, возвращается комбинированная строка.
   - Если соответствие **не найдено**, все равно возвращается строка из левой таблицы, а столбцы из правой таблицы заполняются значениями **`NULL`**.
   - **Не коммутативен!** `A LEFT JOIN B` **не** эквивалентно `B LEFT JOIN A`.

   Пример: Получить всех студентов и названия их групп (даже если группа не указана).

```sql
SELECT s.st_name, g.gr_name
FROM students s
LEFT JOIN groups g ON s.gr_id = g.gr_id;
```

   В результате будут все студенты. Если у студента `gr_id` равен `NULL` или ссылается на несуществующую группу, в колонке `gr_name` для него будет `NULL`.

3. **`RIGHT OUTER JOIN` (или `RIGHT JOIN`):**

   - Симметричен `LEFT JOIN`. Возвращает **все строки** из *правой* таблицы (`table2`).
   - Если для строки из правой таблицы не найдено соответствие в левой, столбцы из левой таблицы заполняются `NULL`.
   - **Не коммутативен.**

   Пример: Получить все группы и имена студентов в них (даже если в группе нет студентов).

```sql
SELECT s.st_name, g.gr_name
FROM students s
RIGHT JOIN groups g ON s.gr_id = g.gr_id;
```

   В результате будут все группы. Если в группе нет студентов, в колонке `st_name` для такой группы будет `NULL`.

4. **`FULL OUTER JOIN` (или `FULL JOIN`):**

   - Комбинация `LEFT JOIN` и `RIGHT JOIN`. Возвращает **все строки** из *обеих* таблиц.
   - Если для строки из одной таблицы нет соответствия в другой, недостающие столбцы заполняются `NULL`.
   - Коммутативен.

   Пример: Получить всех студентов и все группы, сопоставив их.

```sql
SELECT s.st_name, g.gr_name
FROM students s
FULL OUTER JOIN groups g ON s.gr_id = g.gr_id;
```

Результат будет содержать: студентов с их группами; студентов без групп (с `NULL` в `gr_name`); группы без студентов (с `NULL` в `st_name`).

5. **`CROSS JOIN`:**
   - Возвращает **декартово произведение** двух таблиц. Условие `ON` не используется.
   - Используется редко, в основном для генерации комбинаций.

   Пример: Сгенерировать все возможные пары студент-группа.

```
SELECT s.st_name, g.gr_name
FROM students s
CROSS JOIN groups g;
```

Способы указания условия соединения:

- **`ON join_condition`**: Самый гибкий способ, позволяет указать любое логическое условие для соединения строк. Используется чаще всего.

```
FROM students s JOIN groups g ON s.gr_id = g.gr_id
```

- **`USING (column_list)`**: Упрощенная запись для эквисоединения, когда столбцы, по которым идет соединение, имеют **одинаковые имена** в обеих таблицах. Столбцы из списка `column_list` включаются в результат только **один раз**.

```
-- Предполагая, что в обеих таблицах есть колонка gr_id
FROM students JOIN groups USING (gr_id)
```

Эквивалентно `ON students.gr_id = groups.gr_id`, но в `SELECT` будет только одна колонка `gr_id`.

- **`NATURAL JOIN`**: Еще более короткая запись. Автоматически соединяет таблицы по *всем* столбцам с **одинаковыми именами**. Совпадающие столбцы включаются в результат только один раз.

```
-- Предполагая, что соединение нужно делать только по gr_id, и других колонок с одинаковыми именами
нет
FROM students NATURAL JOIN groups
```

Опасно! *Используйте `NATURAL JOIN` с осторожностью, так как случайное совпадение имен колонок может привести к неверному условию соединения или к `CROSS JOIN`, если совпадающих имен нет.*

Часть 4: Вложенные Подзапросы (Subqueries)

Подзапрос — это `SELECT`-запрос, вложенный внутрь другого SQL-запроса (`SELECT`, `INSERT`, `UPDATE`, `DELETE`).

Типы подзапросов:

1. **Простые (Некоррелированные):**
   - Внутренний запрос **не зависит** от внешнего.
   - Может быть выполнен **один раз** независимо от внешнего запроса.
   - Его результат (одно значение, список значений, таблица) подставляется во внешний запрос.
   - **Пример:** Найти студентов из городов России (как в Лекции 4, слайд 44).

```sql
SELECT Surname
FROM STUDENT
WHERE CityName IN (
    SELECT City -- Этот подзапрос выполняется один раз
    FROM CITIES
    WHERE Country = 'Россия'
);
```

   Сначала выполняется внутренний запрос, возвращает список городов (`'Москва', 'Санкт-Петербург', ...`). Затем внешний запрос ищет студентов, у которых `CityName` есть в этом списке.

2. **Коррелированные:**
   - Внутренний запрос **зависит** от данных из внешнего запроса (ссылается на его таблицы/колонки).
   - Внутренний запрос **перевыполняется для каждой строки**, обрабатываемой внешним запросом.
   - Часто бывают **менее эффективны**, чем некоррелированные запросы или соединения.
   - **Пример:** Найти студентов, участвовавших в олимпиадах (как в Лекции 4, слайд 48).

```sql
SELECT Surname
FROM STUDENT s -- Внешний запрос ссылается на таблицу STUDENT как 's'
WHERE EXISTS ( -- Оператор для подзапроса
    SELECT 1
    FROM STUDENT_OLYMPIAD so
    WHERE so.StID = s.StudentID -- Внутренний запрос ссылается на s.StudentID из внешнего
);
```

   Для каждой строки `s` из `STUDENT` выполняется внутренний `SELECT`. Если он находит хотя бы одну строку в `STUDENT_OLYMPIAD` с соответствующим `StID`, `EXISTS` возвращает `TRUE`, и фамилия студента `s` попадает в результат.

Сравнение Подзапросов и Соединений:

- Многие задачи можно решить и с помощью подзапросов, и с помощью соединений.
- Соединения (`JOIN`) часто **более читаемы** и **лучше оптимизируются** СУБД, особенно для эквисоединений.
- Коррелированные подзапросы часто можно переписать с использованием `JOIN`, что обычно повышает производительность.
- Подзапросы бывают удобны и необходимы в некоторых случаях, особенно с операторами `IN`, `ANY`, `ALL`, `EXISTS`, или когда нужно получить результат агрегации для использования во внешнем запросе.

Операторы для работы с Подзапросами:

- **`IN (подзапрос)`**:
  - Проверяет, равно ли выражение значению *хотя бы одной* строки, возвращаемой подзапросом.
  - Подзапрос должен возвращать **один столбец**.
  - `expression IN (SELECT column FROM ...)` эквивалентно `expression = ANY (SELECT column FROM ...)`
- **`NOT IN (подзапрос)`**:
  - Проверяет, что выражение *не равно ни одному* значению из подзапроса.
  - **Осторожно с `NULL`!** Если подзапрос возвращает хотя бы одно `NULL`-значение, `NOT IN` *всегда* вернет `FALSE` или `NULL`, но никогда `TRUE`, что часто приводит к неожиданным пустым результатам.
- **`EXISTS (подзапрос)`**:
  - Возвращает `TRUE`, если подзапрос возвращает **хотя бы одну строку**, иначе `FALSE`.
  - Содержимое строк подзапроса не имеет значения (часто пишут `SELECT 1` или `SELECT *`).
  - Часто используется с коррелированными подзапросами для проверки наличия связанных данных.
- **`ANY` / `SOME (подзапрос)`**: (`SOME` — синоним `ANY`)
  - Сравнивает выражение с *каждым* значением, возвращаемым подзапросом (который должен вернуть один столбец), используя заданный оператор (=, >, <, и т.д.).
  - Возвращает `TRUE`, если сравнение истинно **хотя бы для одного** значения из подзапроса.
  - **Пример:** `expression > ANY (SELECT column FROM ...)` - истинно, если выражение больше хотя бы одного значения из подзапроса.
- **`ALL (подзапрос)`**:
  - Сравнивает выражение с *каждым* значением, возвращаемым подзапросом (который должен вернуть один столбец), используя заданный оператор.
  - Возвращает `TRUE`, если сравнение истинно **для всех** значений из подзапроса (включая случай, когда подзапрос не вернул строк).
  - **Пример:** `expression > ALL (SELECT column FROM ...)` - истинно, если выражение больше *всех* значений из подзапроса (т.е. больше максимального).

Часть 5: Агрегатные Функции, Группировка и Фильтрация Групп

Часто нужно получить не сырые данные, а агрегированную информацию (итоги, средние значения, количество и т.д.).

Агрегатные функции:

Выполняют вычисления над набором строк и возвращают одно значение.

- `COUNT(*)`: Количество строк в группе (или во всей таблице).
- `COUNT(column)`: Количество не-`NULL` значений в указанном столбце в группе.
- `COUNT(DISTINCT column)`: Количество *уникальных* не-`NULL` значений в столбце.
- `SUM(column)`: Сумма значений в столбце (для числовых типов). Игнорирует `NULL`.
- `AVG(column)`: Среднее значение в столбце (для числовых типов). Игнорирует `NULL`.
- `MIN(column)`: Минимальное значение в столбце. Игнорирует `NULL`.
- `MAX(column)`: Максимальное значение в столбце. Игнорирует `NULL`.
- И другие (`STDDEV`, `VARIANCE` и т.д.).

Группировка (`GROUP BY`):

- Используется для разделения строк таблицы на группы на основе одинаковых значений в одной или нескольких колонках.
- Агрегатные функции затем применяются к каждой группе отдельно.
- **Важное правило:** Если используется `GROUP BY`, то в секции `SELECT` можно указывать только:
  1. Столбцы, перечисленные в `GROUP BY`.
  2. Агрегатные функции.
  3. Константы или выражения, не зависящие от конкретных строк внутри группы.

Пример: Посчитать количество студентов в каждой группе.

```sql
SELECT
    gr_id, -- Столбец из GROUP BY
    COUNT(*) AS student_count -- Агрегатная функция
FROM students
WHERE gr_id IS NOT NULL -- Исключим студентов без группы (опционально)
GROUP BY gr_id -- Группируем по ID группы
ORDER BY gr_id;
```

Вывод (примерный):

```
 gr_id | student_count
-------+---------------
    34 |             2
    37 |             1
(2 rows)
```

Фильтрация групп (`HAVING`):

- Применяется *после* группировки и вычисления агрегатных функций.
- Фильтрует *результаты группировки* (целые группы), а не отдельные строки (как `WHERE`).
- В условии `HAVING` можно использовать агрегатные функции.

Пример: Найти группы, в которых более одного студента.

```sql
SELECT
    gr_id,
    COUNT(*) AS student_count
FROM students
WHERE gr_id IS NOT NULL
GROUP BY gr_id
```

```
HAVING COUNT(*) > 1 -- Фильтруем группы по результату агрегатной функции
ORDER BY gr_id;
```

Вывод (примерный):

```
 gr_id | student_count
-------+---------------
    34 |             2 -- Группа 37 отфильтрована, т.к. там 1 студент
(1 row)
```

Источник — https://xn--b1amah.xn--80aalyho.xn--d1acj3b/mediawiki/index.php?title=БД:Теория:Глава_4&oldid=36