Содержание

Аббревиатуры

- БД – база данных;
- СУБД – система управления базами данных;
- ПО – программное обеспечение;
- SQL – Structured Query Language (структурированный язык запросов);
- DBeaver – программа для работы с различными СУБД;
- DDL – Data Definition Language (язык определения данных);
- DML – Data Manipulation Language (язык манипулирования данными);
- DCL – Data Control Language (язык контроля данных);
- TCL – Transaction Control Language (язык управления транзакциями);
- ХП – хранимая процедура;
- НФ – нормальная форма.

Ссылки на документацию

- Документация PostgreSQL: https://postgrespro.ru/docs/postgresql/9.4/
- DBeaver: https://dbeaver.io/
- Установка pg Admin4:
    a. Установщик PostgreSQL: https://www.postgresql.org/download/
    b. Видеоинструкция: https://www.youtube.com/watch?v=oEi5IUgxaU0&list=PLJJA9GtpZ-8Tl5XzimM1RuB6s8xJMsANF&index=4
    c. Файл с данными для создания и наполнения БД: https://stepik.org/media/attachments/lesson/1072477/scripts_500_pg_admin_4_stepik.sql
- Установка MySQL:
    a. Установщик: https://dev.mysql.com/downloads/installer/
    b. Видеоинструкция: https://www.youtube.com/watch?v=xaPuXh8IFIU
- Описание БД: https://docs.yugabyte.com/latest/sample-data/northwind/
- Форматирование даты/времени: https://postgrespro.ru/docs/postgrespro/10/functions-formatting
- Агрегатные функции: https://postgrespro.ru/docs/postgresql/9.5/functions-aggregate
- Обобщенные табличные выражения: https://postgrespro.ru/docs/postgrespro/9.5/queries-with
- Триггеры:
    a. https://postgrespro.ru/docs/postgresql/15/trigger-definition
    b. https://www.postgresqltutorial.com/postgresql-triggers/
- Оконные функции:

a. https://learnsql.com/course/window-functions-practice-set/northwind/introduction/the-orders-table
b. http://thisisdata.ru/blog/uchimsya-primenyat-okonnyye-funktsii/

Термины для понимания БД

База данных (БД) – набор сведений об объектах, структурированный определенным образом.

Виды БД:

- Текстовые – информация собирается в простых по своей структуре файлах (.txt, .csv);
- Иерархические – похожи на текстовые, но добавляются связи, объекты делятся на родителей и потомков;
- Сетевые – развитый иерархический подход за счет моделирования сложных связей. В такой базе данные хранятся в виде узлов (вершин) и связей (ребра) между узлами;
- Реляционные – данные формируются в таблицы из строк и столбцов. В строках - значения, в столбцах - атрибуты (фамилия, имя, дата рождения).

Система управления базами данных (СУБД) – программное обеспечение (ПО), позволяющее создать базу данных, управлять ей и манипулировать данными.

Виды СУБД:

- MySQL;
- Microsoft SQL Server;
- PostgreSQL;
- ArangoDB и прочие.

Structured Query Language (SQL) – структурированный язык запросов, предназначенный для определения, управления и манипуляции данными.

DBeaver – программа для работы с различными СУБД (PostgreSQL, MySQL, Oracle и прочие).

Схема БД – набор объединенных и логически связанных объектов, включающий описание содержания и структуры БД.

Primary Key (PK, первичный ключ) – поле или комбинация полей однозначно определяющих каждую запись в таблице.

Foreign Key (FK, внешний ключ) – одно или несколько полей в таблице, содержащих ссылку на поле или поля **PK** в другой таблице.

Виды отношений в базе данных – связей между таблицами:

- один к одному (значению из таблицы A соответствует только одна запись из таблицы B;
- ***один ко многим*** (одной записи из таблицы A соответствуют несколько записей в таблице B);
- многие к одному (нескольким записям из таблицы B соответствует одна запись из таблицы A - один ко многим наоборот);
- многие ко многим (множественным записям из таблицы A соответствуют множественные записи из таблицы B).

Знакомство с БД «Northwind»

Данные для подключения (PostgreSQL):

- **Host:** 95.163.241.236
- **Port:** 5432
- **Data Base:** nordwind
- **User:** student
- **Password:** qweasd963

База данных содержит данные о продажах фиктивной компании «Nordwind Traders», которая импортирует и экспортирует фирменные продукты питания со всего мира.

База данных включает основные сведения:

- **Suppliers**: данные поставщиках компании;
- **Customers**: данные о клиентах;
- **Employees**: данные о сотрудниках;
- **Products**: данные о продуктах;
- **Shippers**: данные о грузоотправителях, которые отправляют продукцию от торговцев конечным покупателям

Orders and Order_Details: данные о заказах.

Группы операторов SQL

1. ***DDL (Data Definition Language)*** – используется для создания и изменения структуры БД и её составных частей - таблиц, представлений, индексов, триггеров и процедур.
    - Create – используется для создания объектов базы данных;
    - Alter – используется для изменения объектов базы данных;
    - Drop – используется для удаления объектов базы данных;
    - Truncate – очищает объект базы данных.
2. ***DML (Data Manipulation Language)*** – используется для манипуляций с данными (выборка, вставка, обновление и удаление).

- Select – осуществляет выборку данных;
- Insert – добавляет новые данные;
- Update – изменяет существующие данные;
- Delete – удаляет данные.

3. ***DCL (Data Control Language)*** – используется для управления правами доступа к данным и выполнением процедур в многопользовательской среде.
    - Grant – предоставляет разрешения пользователю на определённые операции с объектом;
    - Revoke – отзывает выданные разрешения;
    - Deny – задаёт запрет, имеющий приоритет над разрешением.
4. ***TCL (Transaction Control Language)*** – используется для осуществления транзакций в SQL.
    - Begin transaction – оператор для определения начала транзакции;
    - Commit transaction – оператор управления транзакциями для успешного завершения транзакции;
    - Rollback transaction – оператор для отмены всех изменений, которые были внесены в процессе выполнения транзакции, например, в случае ошибки, мы откатываем все назад;
    - Save transaction – оператор для установки точки сохранения в пределах транзакции.

Типы данных

Тип данных – атрибут, определяющий, какого рода данные могут храниться в объекте: целые числа, числа с дробной частью, текстовые данные, даты и так далее.

Целочисленные типы данных:

- smallint;
- integer;
- bigint.

Используются для хранения целых чисел, отличаются диапазоном и размером хранимых данных.

Числа с плавающей точностью:

- numeric.

Тип numeric позволяет хранить числа с очень большим количеством цифр и выполнять вычисления точно. Он рекомендуется для хранения денежных сумм и других величин, где важна точность.

Типы данных с плавающей точкой:

- real;
- double precision.

Типы данных real и double precision хранят приближённые числовые значения с переменной точностью.

Текстовые типы данных:

- char, character;
- varchar;
- text.

Используются для записи текстовой информации.

Логический (булевый) тип данных:

- true (истина);
- false (ложь).

Используются для записи состояния: истина или ложь (например, участвует ли товар в акции - да/нет).

Тип данных "Дата и время":

- timestamp;
- date;
- time;
- interval.

Используются для записи даты и времени.

NULL:

Поле со значением ***NULL*** является полем без значения (данные не заполнены). Важно понимать, что значение ***NULL*** отличается от нулевого значения или поля, которое содержит пробелы.

pg_typeof() - функция, предназначенная для определения типов данных.

Приведение типов с помощью оператора CAST:

Синтаксис оператора **CAST**:

```
CAST(expression AS target_type)
```

Пример:

```
SELECT CAST ('100' AS INTEGER)
```

Приведение с помощью оператора приведения (::):

Данный способ является специфичным для PostgreSQL и не соответствует стандарту SQL.

Синтаксис оператора **оператора приведения (::)**:

```
::numeric
```

Пример:

```
SELECT sum(unit_price * quantity)::numeric from order_details od
```

Выбор данных (select)

SELECT – оператор, предназначенный для выборки данных (выбирает указанные столбцы) из таблицы.

Синтаксис простого SQL-запроса:

```
SELECT <столбцы, которые требуется выбрать из БД>
```

```
FROM <наименование таблицы из которой берутся данные>
```

Выборка всех данных из таблицы:

- оператор SELECT;
- символ "*****";
- оператор FROM;
- наименование таблицы.

Результатом является таблица, в которую включены все строки и столбцы указанной в запросе таблицы.

Пример простого select-запроса:

```
SELECT * FROM order_details
```

Выборка определенных столбцов из таблицы:

- оператор SELECT;
- список столбцов таблицы через запятую;
- оператор FROM;
- наименование таблицы.

Результатом является таблица, в которую включены все данные из указанных после **SELECT** столбцов исходной таблицы.

Алиасы

Определение новых наименований для столбцов: для того чтобы задать новое наименование для столбца используется оператор **AS**;

Примечание:

- оператор **AS** можно не использовать и сразу после наименования столбца, которое используется в БД указывать новое наименование (например, **order_id** "id заказа");
- новое название столбца (можно русскими буквами), выводимое в результате запроса, но это должно быть одно слово, если название состоит из двух слов – соединяйте их подчеркиванием.

Результатом является таблица, в которую включены все данные из указанных после **SELECT** столбцов исходной таблицы. Каждому столбцу в результате запроса присваивается новое имя, заданное после **AS**, или столбец получает имя столбца исходной таблицы, если **AS** отсутствует.

Пример:

```
SELECT
order_id "id заказа",
customer_id as "id покупателя",
order_date as "Дата заказа",
ship_name as "Компания доставки",
ship_address as "Адрес доставки",
ship_city as "Город доставки"
FROM
orders
```

Вычисляемый столбец

Добавление вычисляемого столбца:

- после оператора **SELECT** указывается выражение и задается имя;
- выражение может включать имена столбцов, константы, знаки операций, встроенные функции.

Результатом является таблица, в которую включены все данные из указанных после **SELECT** столбцов, а также новый столбец, в каждой строке которого вычисляется заданное выражение.

Пример:

Вывести всю информацию о деталях заказов, а также для каждой позиции посчитать продажи (произведение цены на количество). Вычисляемому столбцу дать имя **amount**.

```
SELECT
order_id,
product_id,
unit_price,
quantity,
unit_price * quantity as amount
FROM
order_details od
```

Фильтрация (where)

Для того чтобы включить в итоговую выборку не все строки исходной таблицы, а только те, которые отвечают некоторому условию, используется оператор **WHERE** и логическое выражение, от результата которого зависит, будет ли включена строка в выборку или нет.

Если условие – **истина**, то строка (запись) включается в выборку,
если **ложь** – нет.

Логическое выражение может включать операторы сравнения.

Синтаксис оператора WHERE:

WHERE <наименование столбца фильтрации> <условие фильтрации строк>

Пример использования оператора WHERE: требуется вывести информацию по продуктам, которые продаются в бутылках по 24–12 унций (24 - 12 oz bottles).

```
SELECT
product_name,
supplier_id,
category_id,
quantity_per_unit,
unit_price
FROM
products p
WHERE quantity_per_unit = '24 - 12 oz bottles'
```

Операторы сравнения:

- = равно
- > - больше
- < - меньше
- <= - меньше или равно
- >= - больше или равно
- != или <> - не равно
- IN() - соответствие значению в списке
- NOT - отрицает условие
- BETWEEN - нахождение в пределах диапазона **(включительно)**
- IS NULL - равно значению NULL
- IS NOT NULL - не равно значению NULL
- LIKE - соответствие шаблону (% - неограниченное число символов, _ - только один символ)
- ILIKE - соответствие шаблону (регистронезависимо)
- EXISTS - условие выполнено, если подзапрос возвращает хотя бы одну строку

Логические операторы:

- AND;
- OR;

- NOT.

Очередность выполнения логических операторов:

1. NOT;
2. AND;
3. OR.

Distinct – оператор, который принимает набор значений и оставляет только уникальные.

Сортировка (order by)

Сортировка данных позволяет указывать, в каком порядке требуется выводить результат запроса. Например, таким образом можно легко найти наименьшие или наибольшие значения в столбце (продукт с самой высокой ценой или магазин с наибольшим количеством продаж).

Оператор ORDER BY – позволяет сортировать записи в вашем результирующем наборе.

Синтаксис ORDER BY:

```
ORDER BY <столбец(столбцы) сортировки> <вид сортировки>
```

Виды сортировки в ORDER BY:

- ASC (ascending, используется по умолчанию) - по возрастанию;
- DESC (descending) - по убыванию.

Оператор LIMIT – позволяет извлечь заданное количество строк.

Синтаксис оператора LIMIT:

```
LIMIT <ограничение на количество возвращаемых строк>
```

Оператор OFFSET позволяет пропустить заданное количество строк, прежде чем начать возвращать строки. OFFSET 0 это то же самое, что и не использовать OFFSET.

Синтаксис оператора OFFSET:

```
OFFSET <количество строк, которые следует пропустить при возвращении строк>
```

Пример использования операторов ORDER BY, LIMIT и OFFSET:

```
select product_id, product_name, unit_price
from products p
order by unit_price desc
limit 3
offset 1
```

Группировка (group by)

До этого момента мы получили информацию по каждой записи в БД.

Если мы хотим получить информацию не о каждой записи отдельно, а о группах, которые они образуют, тут будет полезна группировка данных по какому-либо признаку.

Группировка – данные, объединенные по некоторому признаку.

Группировка используется для того, чтобы:

1. Избавиться от избыточности данных.
2. Произвести вычисления.

Для группировки данных используется оператор **GROUP BY.**

Группировка в большинстве случаев используется совместно с агрегатными функциями.

Если столбец указан в SELECT **БЕЗ** применения функции, то он обязательно должен быть указан и в **GROUP BY**.

Синтаксис оператора группировки:

```
GROUP BY <группировка строк по столбцу(столбцам)>
```

Примеры группировки:

```
select product_id
from order_details od
group by product_id
order by product_id desc
select ship_country, ship_city
from orders o
```

```
group by ship_country, ship_city
order by ship_country, ship_city
```

Агрегатные функции

- MIN – находит наименьшее значение;
- MAX – находит наибольшее значение;
- AVG – находит среднее значение;
- SUM – находит сумму значений;
- COUNT(*) – находит количество строк в запросе;
- COUNT(expression) – находит количество строк в запросе, для которых expression не содержит значение NULL;
- STRING_AGG(expression, 'разделитель') – в рамках одной группы соединяет все указанные строковые значения по указанному разделителю.

Примечание: expression – наименование столбца.

Оператор DISTINCT используется для удаления дубликатов из набора результатов. DISTINCT может использоваться только с операторами SELECT.

Пример группировки с агрегатной функцией:

```
select order_id, sum(quantity)
from order_details od
group by order_id
order by order_id
```

Фильтрация групп – фильтрация уже агрегированных значений. Фильтрация не исходных значений строк, а уже сгруппированных значений.

Синтаксис оператора фильтрации групп:

```
HAVING <условие для фильтрации групп>
```

Примеры фильтрации групп:

```
select order_id, sum(quantity) as sum_quantity
from order_details od
group by order_id
having sum(quantity) > 100
order by order_id
select customer_id, count(order_id) as cnt
from orders o
```

```
group by customer_id
having count(order_id) > 15
order by customer_id
```

Соединения таблиц (join)

При работе с данными из одной таблицы работа со связующими столбцами не нужна, но если требуется получить данные из нескольких таблиц, тогда следует понимать по какому принципу связывать данные на основе значений связующих столбцов. Для этого существует оператор **Join.**

Join — оператор для объединения строк из двух или более таблиц на основе связующего столбца.

Синтаксис оператора join:

```
JOIN <название таблицы для присоединения>
```

```
ON <условие присоединения на основе связующих столбцов>
```

Основные виды соединений:

- ***INNER JOIN*** (возвращает только те записи, которые имеют совпадающие значения в обеих таблицах);
- ***LEFT JOIN*** (возвращает все записи из левой таблицы и совпадающие записи из правой таблицы);
- RIGHT JOIN (возвращает все записи из правой таблицы и совпадающие записи из левой таблицы);
- FULL OUTER JOIN (возвращает все записи, если есть совпадения в левой или правой таблицах);
- CROSS JOIN (возвращает результат объединения каждой записи из левой таблицы с записями из правой, такое действие называют декартовым произведением).

Примеры соединения:

```
SELECT o.order_id, o.customer_id, o.order_date, od.product_id,
od.unit_price, od.quantity, round(od.unit_price::numeric * od.quantity, 2) as
amount
FROM orders o
JOIN order_details od
ON o.order_id = od.order_id;
```

```
SELECT o.order_id, o.customer_id, o.order_date, e.first_name || ' ' || e.last_name
as fio
FROM orders o
JOIN employees e
ON o.employee_id = e.employee_id
JOIN customers c
ON o.customer_id = c.customer_id;
```

Объяснение логики работы соединений: https://stepik.org/media/attachments/lesson/1071867/joins-cheat-sheet-ledger.pdf

Структура SQL-запроса

1. SELECT <столбцы, которые требуется выбрать из БД>
2. FROM <наименование таблицы, из которой берутся данные>
3. ***JOIN <наименование таблицы, которую требуется присоединить к первой таблице> ON <условие объединения таблиц>***
4. WHERE <условие фильтрации строк>
5. GROUP BY <группировка строк по столбцу(столбцам)>
6. HAVING <условие для фильтрации групп>
7. ORDER BY <сортировка выбранных данных по столбцу(столбцам)>
8. LIMIT <ограничение на количество возвращаемых строк>
9. OFFSET <количество строк, которые следует пропустить при возвращении строк>

Порядок обработки операторов в SQL-запросе

1. FROM (определение таблиц, задействованных в запросе);
2. ***JOIN ON (выполнение операции объединения на основе условия объединения);***
3. WHERE (применение условий фильтрации к объединённой таблице);
4. GROUP BY (группировка строк по указанным столбцам);
5. HAVING (фильтрация групп по условию);
6. SELECT (выбор столбцов и агрегатных функций из каждой группы);
7. ORDER BY (сортировка строк по указанным столбцам);
8. LIMIT (пропустить определенное количество строк из отсортированного набора результатов);
9. OFFSET (выполнить смещение строк).

Важно!

Порядок **ОБРАБОТКИ** запросов – это не порядок **ЗАПИСИ** ключевых слов в запросе на выборку.

Порядок **ОБРАБОТКИ** нужен для того, чтобы понять, почему, например, в **WHERE** нельзя использовать имена выражений из **SELECT**, так как **SELECT** выполняется компилятором позже, чем **WHERE**, поэтому ему неизвестно, какое выражение написано в **SELECT**.

Объединение запросов

Оператор UNION: объединяет результаты запросов

Запрос №1

UNION

Запрос №2

Примечания:

- Результирующий состав столбцов в обоих запросах должен быть одинаковый (количество столбцов в запросах должно быть одинаковым);
- Если же необходимо при объединении сохранить все, в том числе повторяющиеся строки, то для этого необходимо использовать оператор **ALL**.

Оператор INTERSECT: выполняет запросы и возвращает те строки, которые есть и в 1-ом запросе и во 2-ом

Запрос №1

INTERSECT

Запрос №2

Примечания:

- Если же необходимо при объединении сохранить все, в том числе повторяющиеся строки, то для этого необходимо использовать оператор **ALL**.
- Можно смоделировать с помощью **INNER JOIN**.

Оператор EXCEPT: выполняет запросы и возвращает те записи, которые есть в 1-ом запросе и нет во 2-ом

Запрос №1

EXCEPT

Запрос №2

Примечания:

- Если же необходимо при объединении сохранить все, в том числе повторяющиеся строки, то для этого необходимо использовать оператор **ALL**.
- Можно смоделировать с помощью **LEFT JOIN** или подзапроса.

Пример применения: проверить активность пользователей на курсе.

- **1-ая таблица (start):** пользователи, которые запускают код.
- **2-ая таблица (send):** пользователи, которые отправляют код на проверку.

Пример UNION:

```
select user_id
from run_code
union
select user_id
from send_code
```

Пример INTERSECT:

```
select user_id
from run_code
INTERSECT
select user_id
from send_code
```

Моделирование INTERSECT:

```
select distinct run_code.user_id
from run_code
join send_code
on run_code.user_id = send_code.user_id
```

Пример EXCEPT:

```
select user_id
from run_code
EXCEPT
select user_id
from send_code
```

Моделирование EXCEPT:

```
select run_code.user_id
from run_code
left join send_code
on run_code.user_id = send_code.user_id
where send_code.user_id is null
```

ИЛИ

```
select run_code.user_id
from run_code
where run_code.user_id not in (select user_id from send_code)
```

Подзапросы

Подзапрос – оператор ***SELECT***, который может быть встроен в тело другого запроса.

Внешний (основной) запрос использует результат выполнения внутреннего запроса для определения окончательного результата.

Подзапрос используется для:

- сравнения выражения с результатом вложенного запроса;
- определения того, включено ли выражение в результаты вложенного запроса;
- проверки того, выбирает ли запрос определенные строки.

По количеству возвращаемых значений подзапросы разделяются на два типа:

- скалярные подзапросы, которые возвращают единственное значение;
- табличные подзапросы, которые возвращают множество значений.

По способу выполнения выделяют два типа подзапросов:

- простые подзапросы (может рассматриваться независимо от внешнего запроса, СУБД выполняет такой подзапрос один раз и затем помещает его результат во внешний запрос);
- сложные подзапросы (не может рассматриваться независимо от внешнего запроса, в этом случае выполнение оператора начинается с внешнего запроса, который отбирает каждую отдельную строку таблицы и для каждой выбранной строки СУБД выполняет подзапрос один раз).

Задача №1: из таблицы products вывести только те продукты, где поставщики из 'USA':

```
SELECT product_id, product_name, quantity_per_unit, unit_price

FROM products p

WHERE supplier_id in (select s.supplier_id from suppliers s where s.country = 'USA')
```

Задача №2: из таблицы order_details вывести только те заказы, где в таблице products есть такой же продукт и supplier_id = 1:

```
SELECT *

FROM order_details od

where exists (

select *

from products p

where od.product_id = p.product_id

and p.supplier_id = 1

)
```

Обобщенные табличные выражения (CTE)

Common Table Expression (CTE) – временный результирующий набор данных, к которому можно обращаться в последующих запросах. Для

написания обобщённого табличного выражения используется оператор **WITH.**

Синтаксис CTE:

```
with name_CTE as (
Тело_Запроса
)
```

Пример CTE:

```
with order_customers as (select o.order_id, od.product_id, c.company_name, c.address
from orders o
join customers c
on o.customer_id = c.customer_id
join order_details od
on o.order_id = od.order_id
)
select company_name, count(product_id) cnt
from order_customers
group by company_name
order by cnt desc
```

Оператор case

Оператор CASE – условный оператор языка SQL.
Данный оператор позволяет осуществить проверку условий и возвратить в зависимости от выполнения того или иного условия тот или иной результат.

Синтаксис CASE:

```
(CASE
WHEN <условие_1> THEN <возвращаемое_значение_1>
WHEN <условие_2> THEN <возвращаемое_значение_2>
WHEN <условие_n> THEN <возвращаемое_значение_n>
ELSE <возвращаемое_значение_по_умолчанию>
END
) AS case_name
```

Пояснения к структуре синтаксиса оператора CASE:

- WHEN-условия проверяются последовательно, сверху-вниз. При достижении первого удовлетворяющего условия дальнейшая проверка прерывается и возвращается значение, указанное после слова THEN, относящегося к данному блоку WHEN;
- Если ни одно из WHEN-условий не выполняется, то возвращается значение, указанное после слова ELSE;
- Если ELSE-блок не указан и не выполняется ни одно WHEN-условие, то возвращается NULL.

Пример использования оператора CASE:

```
select territory_id, territory_description, region_description,
(case
when region_description = 'Eastern' then 'Восточный'
when region_description = 'Western' then 'Западный'
when region_description = 'Northern' then 'Северный'
when region_description = 'Southern' then 'Южный'
else 'Другой регион'
end
) as rus_region
from region r
left join territories t
on r.region_id = t.region_id
```

Оператор coalesce

COALESCE – специальное выражение, которое вычисляет по порядку каждый из аргументов и на выходе возвращает значение первого аргумента, который не является NULL.

Функция принимает множество аргументов и вычисляет значения следующим образом: если первое значение не null, то возвращает его значение, если null, то переходит к следующему и тд. Если проверять нечего, тогда возвращается null.

Задача – требуется связаться с клиентами и приоритетный способ связи это fax, если fax – не указан, то нужен хотя бы телефон клиента. Создать вычисляемый столбец с факсом либо телефоном.

Решение оператором CASE:

```
select *, case
        when fax is not null then fax
        when phone is not null then phone
        else null
        end as contacts
from customers c
```

Решение оператором COALESCE:

```
select *, coalesce(fax, phone) as contacts
from customers c
```

Оператор nullif

Оператор NULLIF: служит для корректной отработки значения null.

Функция сравнивает два аргумента: если аргументы не равны, то возвращается первый аргумент, если аргументы равны, то возвращается null.

```
select nullif(1, 0)
```

```
select nullif(0, 1)
```

```
select nullif(1, 1)
```

Практический пример:

```
select product_name, units_in_stock, nullif(units_in_stock, 0)
from products p
```

Оператор exists

EXISTS: логический оператор, который проверяет наличие строк в подзапросе.

Синтаксис EXISTS:

```
EXISTS (подзапрос)
```

Аргументом **EXISTS** является обычный оператор **SELECT**, то есть подзапрос. Выполнив запрос, система проверяет, возвращает ли он строки в результате. Если он возвращает минимум одну строку, результатом **EXISTS** будет «true», а если не возвращает ни одной — «false».

Подзапрос может обращаться к переменным внешнего запроса, которые в рамках одного вычисления подзапроса считаются константами.

Пример использования оператора EXISTS:

Найдем все записи из таблицы customers, где есть хотя бы одна запись в таблице orders с тем же customer_id:

```
select *
FROM customers c
WHERE EXISTS (SELECT * FROM orders o WHERE c.customer_id =
o.customer_id)
```

Операторы SQL, использующие условие EXISTS, очень неэффективны, поскольку подзапрос повторно запускается для каждой строки

Функции по работе с текстом

- **concat (expression1, expression2, ... expression_n)** --> конкатенация строк;
- **concat_ws ('разделитель', expression1, expression2, ... expression_n)** --> конкатенация строк с одинаковым разделителем;

- **string || string** --> конкатенация строк ('Post' || 'greSQL');
- **length(string)** --> вычисление длины строки;
- **lower(string)** --> переводит символы строки в нижний регистр;
- **upper(string)** --> переводит символы строки в верхний регистр;
- **initcap(string)** --> переводит первую букву каждой строки в верхний регистр, остальный в нижний регистр;
- **substring (string [from int] [for int])** --> извлекает подстроку (substring('Thomas' from 2 for 3));
- **coalesce (expression1, expression2, ... expression_n)** --> возвращает первое не null выражение в списке;
- **trim ([leading | trailing | both] [trim_character] from string)** --> удаляет все указанные символы из начала или конца строки;
- **position(string in string)** --> возвращает число, означающее первое вхождение подстроки в строке;
- **left(string, количество символов)** --> возвращает указанное количество символов с начала строки;
- **right(string, количество символов)** --> возвращает указанное количество символов с конца строки.

Функции по работе с числами

- **round (number, [decimal_places])** --> возвращает число, округленное до определенного количества десятичных знаков;
- **abs (number)** --> возвращает абсолютное значение числа.
- **ceiling (number)** --> возвращает наименьшее целочисленное значение, которое больше или равно number;
- **floor (number)** --> возвращает наибольшее целочисленное значение, которое меньше или равно number;
- **trunc (number, [decimal_places])** --> возвращает number, усеченное до определенного количества десятичных знаков.

Функции даты/времени

- **current_date** --> текущая дата;
- **current_time** --> текущая время;
- **now()** --> текущая дата и время;
- **age(timestamp)** --> вычитает дату/время из `current_date` (полночь текущего дня);
- **extract (field from source)** --> получает из значений даты/времени поля, такие как год, месяц, неделя, час и др;
- **date_trunc ('поле', значение)** --> параметр *поле* определяет, до какой точности обрезать переданное значение.;
- **to_char(timestamp, text)** --> преобразует время в текст;
- **to_date(text, text)** --> преобразует текст в дату (to_date('05 Dec 2000', 'DD Mon YYYY'));

- **to_timestamp(text, text)** --> преобразует текст в дату со временем (to_timestamp('05 Dec 2000 22 04 55', 'DD Mon YYYY HH24 MI SS'));
- **make_date(expression1, expression2, ... expression_n)** --> создание даты из чисел.

Функции MySQL

- **datediff('2020-05-09', '2020-05-01')** --> возвращает разность в днях между двумя значениями даты (date1- date2). **Результат:** 8;
- **year('2020-04-12')** --> возвращает год для указанной даты. **Результат:** 2020;
- **month('2020-04-12')** --> возвращает месяц для указанной даты. **Результат:** 4;
- **monthname('2020-04-12')** --> возвращает название месяца на английском языке для указанной даты. **Результат:** April;
- **day('2020-04-12')** --> возвращает день для указанной даты. **Результат:** 12.

Операторы группы DDL

DDL (Data Definition Language) – используется для создания и изменения структуры БД и её составных частей - таблиц, представлений, индексов, триггеров и процедур.

- Create – используется для создания объектов базы данных;
- Alter – используется для изменения объектов базы данных;
- Drop – используется для удаления объектов базы данных;
- Truncate – очищает объект базы данных.

Создание таблицы (Create):

```
CREATE TABLE employee_territories (
    employee_id smallint NOT NULL,
    territory_id character varying(20) NOT NULL
);
```

Изменение таблицы (Alter):

```
ALTER TABLE ONLY categories
    ADD CONSTRAINT pk_categories PRIMARY KEY (category_id);
```

Удаление таблицы (Drop)**:**

```
DROP TABLE categories;
```

Удаление данных в таблице (Truncate):

```
TRUNCATE TABLE suppliers;
```

Операторы группы DML

DML (Data Manipulation Language) – используется для манипуляций с данными (выборка, вставка, обновление и удаление).

- Select – осуществляет выборку данных;
- Insert – добавляет новые данные;
- Update – изменяет существующие данные;
- Delete – удаляет данные.

Вставка данных (Insert):

```
INSERT INTO employee_territories VALUES (1, '06897');
INSERT INTO employee_territories VALUES (1, '19713');
```

Обновление данных (Update):

```
UPDATE employee_territories
SET territory_id = '001'
WHERE employee_id = 1
```

Удаление данных (Delete):

```
DELETE FROM employee_territories
WHERE territory_id = '001'
```

Ограничения в БД

Ограничения – это любое правило, применяемое к столбцу или таблице, которое определяет, какие данные можно в него вносить, а какие – нет.

Каждый раз, когда вы пытаетесь выполнить операцию, изменяющую данные в таблице, – такую как **INSERT, UPDATE или DELETE**, – СУБД проверяет, не нарушают ли эти данные какие-либо существующие ограничения. Если ограничения запрещают выполнять такую операцию, она возвращает ошибку.

Если клиент указывает, что каждая запись продукта должна иметь уникальный идентификационный номер, вы можете создать столбец с ограничением **UNIQUE**, которое гарантирует, что в одном столбце не может быть двух одинаковых записей.

Стандарт SQL формально определяет всего пять ограничений:

1. PRIMARY KEY (требует, чтобы каждая запись в данном столбце была уникальной и не равнялась NULL);
2. FOREIGN KEY (требует, чтобы каждая запись в данном столбце уже существовала в определенном столбце из другой таблицы);
3. UNIQUE (запрещает добавлять в заданный столбец любые повторяющиеся значения);
4. CHECK (задает ограничение для диапазона значений, которые могут храниться в столбце);
5. NOT NULL (запрещает присваивать значение NULL).

Пример ограничения PRIMARY KEY:

```
ALTER TABLE ONLY categories
    ADD CONSTRAINT pk_categories PRIMARY KEY (category_id);
```

Пример ограничения FOREIGN KEY:

```
ALTER TABLE ONLY orders
    ADD CONSTRAINT fk_orders_customers FOREIGN KEY
(customer_id) REFERENCES customers;
```

Пример ограничения UNIQUE:

```
CREATE TABLE customers (
    customer_id character varying(5) NOT NULL,
    company_name character varying(40) NOT NULL,
    contact_name character varying(30),
    contact_title character varying(30),
    address character varying(60),
    city character varying(15),
    region character varying(15),
    postal_code character varying(10),
    country character varying(15),
    phone character varying(24) UNIQUE,
    fax character varying(24) UNIQUE
);
```

Пример ограничения CHECK:

```
CREATE TABLE customers (
    customer_id character varying(5) NOT NULL,
    company_name character varying(40) NOT NULL,
    contact_name character varying(30),
```

```
    contact_title character varying(30),
    address character varying(60),
    city character varying(15),
    region character varying(15),
    postal_code character varying(10),
    country character varying(15),
    phone character varying(24) UNIQUE,
    fax character varying(24) CHECK(fax !='')
);
```

Пример ограничения NOT NULL:

```
CREATE TABLE region (
    region_id smallint NOT NULL,
    region_description character varying(60) NOT NULL);
```

Представления (view)

Предположим, что вы не хотите каждый раз писать полностью какой-либо запрос. Вы можете создать ***представление*** по данному запросу, фактически присвоить имя запросу, а затем обращаться к нему как к обычной таблице.

Представления позволяют вам скрыть внутреннее устройство ваших таблиц, их можно использовать практически везде, где можно использовать обычные таблицы.

View (представление) – особый объект, который содержит данные, полученные select-запросом из обычных таблиц. Это виртуальная таблица, к которой можно обратиться как к обычной таблице и получить хранимые данные.

Нужны для того, чтобы упростить работу с базой данных и ускорить время ответа сервера.

Синтаксис создания представления:

CREATE view <имя_представления> as <код запроса select, данные которого поместятся в представление>

Пример представления:

```
create view product as

(select * from orders o where customer_id = 'VINET')
```

К представлению также можно применять **select** и условия **where**.

```
SELECT order_id, employee_id FROM product
```

Модифицируемые представления – это означает, что при изменении данных в самом представлении, эти данные изменятся и в таблицах, которые эти данные хранят.

Хранимые процедуры

Хранимая процедура (ХП) – представляют набор инструкций, которые выполняются как единое целое. Хранимые процедуры могут выполнять действия над данными автоматически: вывод данных, удаление, изменение.

В хранимую процедуру можно передавать аргументы и выводить различные данные в зависимости от аргумента. Хранимая процедура является сущностью SQL, которую создают один раз, а затем взывают, передавая аргументы.

Особенности хранимых процедур:

1. Упрощение кода (позволяют упростить комплексные операции и вынести их в единый объект);
2. Безопасность (позволяют ограничить доступ к данным в таблицах и тем самым уменьшить вероятность нежелательных действий в отношении этих данных;
3. Производительность (процедуры обычно выполняются быстрее, чем обычные SQL-инструкции, так как код процедур компилируется один раз при первом её запуске, а затем сохраняется в скомпилированной форме).

Синтаксис ХП (СУБД - MySQL):

```
DELIMITER //
```

```
CREATE procedure <имя_процедуры> (arg_1 datatype_1, arg_2 datatype_2, .....)
```

```
BEGIN
```

- *ЗАПРОС*

```
END
```

```
DELIMITER //
```

Пример ХП (СУБД - MySQL):

```
DELIMITER //

CREATE procedure get_actors(lst_name varchar(50), fst_name varchar(50))

BEGIN

SELECT last_name, first_name, title, release_year, rating

   from sakila.film_actor

   join sakila.film

   on sakila.film_actor.film_id = sakila.film.film_id

   join sakila.actor

   on sakila.film_actor.actor_id = sakila.actor.actor_id

   where last_name = lst_name and first_name = fst_name;

END //
```

Синтаксис вызова ХП: CALL name_procedure(<Аргумент>)

```
CALL get_actors('ALLEN', 'CUBA')
```

Функции

Функция – это набор инструкций, написанных с использованием языка программирования, который может выполняться неоднократно.

Отличие процедур и функций состоит лишь в том, что **процедуры не возвращают никаких значений, в отличии от функций**. И процедуры и функции могут принимать входные параметры.

В PostgreSQL нет различий в синтаксисе создания функций и процедур, обе создаются с помощью ключевых слов **create function**.

Пример функции (Nordwind):

```
create or replace function count_orders(before_date date, after_date date)
returns table (
   count_total bigint
   )
```

```
language plpgsql
as $$
begin
    return query (
      select count(*) from orders
      where order_date between before_date and after_date
    );
end;
$$.
```

Пошаговый разбор синтаксиса функции:

- В первой строке мы создаем функцию и называем ее **count_app_users**.
- Команда **create or replace** используется для того, чтобы создать функцию, если она не существует, или обновить ее, если она уже есть.
- Дальше мы явно указываем тип результата. Она равен **table** для всех SELECT запросов.
- В скобках мы указываем столбцы итоговой таблицы и их тип.
- Команду **language plpgsql** и несколько следующих строк можно рассматривать как шаблон и не слишком вникать в детали.
- Сам запрос мы помещаем внутрь выражения **return query(...)** которое обернуто в **$$ begin** и **end $$**.

P.S. *PLpgSQL (Procedural Language PostGres Structured Query Language) – язык позволяющий расширить возможности обычного языка SQL до процедурного языка.*

P.S.2 $$ – функция, называемая долларовыми кавычками, позволяет включать основной текст, не заключая его в одинарные кавычки.

Вызов функции/процедуры:

```
SELECT * FROM count_orders('1996-07-04', '1996-08-04');
```

Посмотреть текст функции/процедуры:

```
SELECT prosrc
FROM pg_proc
WHERE proname = 'count_orders';
```

Удаление функции/процедуры:

```
DROP function count_orders(after_date date);
```

Триггеры

Триггеры – обработчики событий, они выполняются при наступлении какого-либо простого действия в SQL.

Такими действиями обычно являются:

- Вставка данных (insert);
- Обновление данных (update);
- Удаление данных (delete).

Варианты срабатывание триггера:

1. before insert;
2. before update;
3. before delete;
4. after insert;
5. after update;
6. after delete.

Необходимость триггеров: триггер помогает автоматизировать расчетные рутинные действия, является указанием, что база данных должна автоматически выполнить заданную функцию, всякий раз когда выполнен определённый тип операции.

Основным недостатком использования триггера является то, что необходимо знать о существовании триггера и понимать его логику, чтобы определить последствия изменения данных.

Триггер PostgreSQL поддерживает следующие операции:

- Create trigger – используется для создания триггера;
- Alter trigger – используется для изменения имени существующего триггера;
- Drop trigger – используется для удаления триггера;
- Enable trigger – используется для включения определенного или всех триггеров, связанных с таблицей;
- Disable trigger – используется для отключения определенного или всех триггеров, связанных с таблицей.

Триггер является указанием, что база данных должна автоматически выполнить заданную триггерную функцию, всякий раз когда выполнен определённый тип операции.

Триггерная функция должна быть создана до триггера, после создания триггерной функции создаётся триггер. Одна и та же триггерная функция может быть использована для нескольких триггеров.

Синтаксис триггерной функции:

```
CREATE FUNCTION trigger_function()
RETURNS TRIGGER
LANGUAGE PLPGSQL
AS $$
BEGIN
  -- trigger logic
END;
```

Пошаговый разбор синтаксиса:

1. После ключевых слов `CREATE FUNCTION` указывается имя триггерной функции;
2. Указывается возвращаемое значение с типом `TRIGGER`;
3. После ключевого слова `LANGUAGE` указывается ***PLpgSQL*** *(Procedural Language PostGres Structured Query Language) – язык позволяющий расширить возможности обычного языка SQL до процедурного языка;*
4. После ключевого слова `AS` указывается **$$** – функция, называемая долларовыми кавычками, позволяет включать основной текст, не заключая его в одинарные кавычки.
5. Логика триггерной функции помещается между ключевыми словами **$$ begin** и **end $$**.

Синтаксис создания триггера:

```
CREATE TRIGGER trigger_name
{BEFORE | AFTER} { event }
ON table_name
[FOR [EACH] { ROW | STATEMENT }]
EXECUTE PROCEDURE trigger_function
```

Пошаговый разбор синтаксиса:

1. После ключевых слов `CREATE TRIGGER` указывается имя триггера;
2. Указывается время срабатывания триггера. Это может быть `BEFORE` или `AFTER` событие;
    - Триггеры `BEFORE` срабатывают до того, как оператор начинает что-либо делать;
    - Триггеры `AFTER` срабатывают в самом конце работы оператора.

3. Указывается событие, которое вызывает триггер `(INSERT , DELETE, UPDATE или TRUNCATE)`;
4. После ключевого слова `ON` указывается имя таблицы, связанной с триггером;
5. Указывается тип триггера:
    - Триггер на уровне строки, который задается `FOR EACH ROW` предложением.
    - Триггер на уровне оператора, который задается `FOR EACH STATEMENT` предложением.
    - Триггер на уровне строки запускается для каждой строки, в то время как триггер на уровне оператора запускается для каждой транзакции.

      Предположим, что таблица содержит 100 строк и два триггера, которые будут запущены при возникновении `DELETE` события.

      Если `DELETE` инструкция удаляет 100 строк, триггер на уровне строк сработает 100 раз, по одному разу для каждой удаленной строки. С другой стороны, триггер уровня оператора будет срабатывать один раз независимо от того, сколько строк будет удалено.
6. После ключевых слов `EXECUTE PROCEDURE` указывается имя триггерной функции.

Пример использования триггера: допустим, требуется настроить логирование данных при изменении фамилии сотрудника и записывать изменения в отдельной таблице.

Алгоритм решения задачи:

1. Создать отдельную таблицу для записи изменений в фамилиях сотрудников;
2. Написать триггерную функцию;
3. Написать триггер;
4. Написать код для срабатывания триггера.
5. Проверить результат срабатывания триггера.

Пошаговый разбор синтаксиса:

1. Таблица для записи изменений в фамилиях:
    - ```
      CREATE TABLE employee_audits (
        id_audits INT GENERATED ALWAYS AS IDENTITY,
      ```
      ```

```
  employee_id INT NOT NULL,
  last_name VARCHAR(40) NOT NULL,
  changed_on TIMESTAMP(6) NOT NULL
);
```

2. Триггерная функция:
   - ```
     CREATE OR REPLACE FUNCTION log_last_name_changes()
      RETURNS TRIGGER
      LANGUAGE PLPGSQL
      AS
     $$
     BEGIN
        IF NEW.last_name <> OLD.last_name THEN
            INSERT INTO employee_audits(employee_id, last_name,
     changed_on)
            VALUES(OLD.employee_id, OLD.last_name, now());
        END IF;
        RETURN NEW;
     END;
     $$
     ```
     - Если фамилия сотрудника меняется, то функция вставляет в таблицу `employee_audits` идентификатор сотрудника, старую фамилию и время изменения.
     - `OLD` представляет строку перед обновлением, в то время как `NEW` представляет новую строку, которая будет обновлена.
     - `OLD.last_name` возвращает последнее имя перед обновлением и `NEW.last_name` возвращает новую фамилию.

Пошаговый разбор синтаксиса (продолжение):

3. Триггер:

- ```
  CREATE TRIGGER last_name_changes
    BEFORE UPDATE
    ON employees
    FOR EACH ROW
    EXECUTE PROCEDURE log_last_name_changes();
  ```

- Создаем триггер и привязываем функцию триггера к таблице `employees`.
- Имя триггера - `last_name_changes`.
- Перед обновлением значения `last_name` столбца автоматически вызывается функция trigger для регистрации изменений.

4. Код для срабатывания триггера:

- `UPDATE employees SET last_name = 'Stone' WHERE employee_id = 7`
    - Обновляем фамилию сотрудника Robert King на новую - `'Stone'`:

5. Результат срабатывания триггера:

- Проверяем была ли обновлена фамилия сотрудника King на `'Stone'`;
- Проверяем содержимое таблицы `employee_audits` : изменение в таблице `employee_audits` было зарегистрировано триггером.

Синтаксис изменение триггера:

```
ALTER TRIGGER trigger_name
ON table_name
RENAME TO new_trigger_name;
```

Пошаговый разбор синтаксиса:

1. После ключевых слов `ALTER TRIGGER` указывается имя триггера;
2. После ключевого слова `ON` указывается имя таблицы, связанной с триггером;
3. После ключевого слова `RENAME TO` указывается новое имя триггера.

Пример изменение триггера:

```
ALTER TRIGGER last_name_changes
ON employees
RENAME TO new_last_name_changes;
```

Синтаксис удаления триггера:

```
DROP TRIGGER [IF EXISTS] trigger_name
ON table_name [ CASCADE | RESTRICT ];
```

Пошаговый разбор синтаксиса:

1. После ключевых слов `DROP TRIGGER` указывается имя триггера;
2. Используйте `IF EXISTS` для удаления триггера, только если он существует. Попытка удалить несуществующий триггер инструкции приводит к ошибке.
3. После ключевого слова `ON` указывается имя таблицы, связанной с триггером;
4. Используйте `CASCADE` опцию, если вы хотите автоматически удалять объекты, зависящие от триггера. Обратите внимание, что `CASCADE` опция также удалит объекты, зависящие от объектов, зависящих от триггера.
5. Используйте `RESTRICT` опцию, чтобы отказаться от удаления триггера, если от него зависят какие-либо объекты. По умолчанию в `DROP TRIGGER` инструкции используется `RESTRICT`.

Пример удаления триггера:

```
DROP TRIGGER new_last_name_changes
ON employees;
```

Синтаксис включения триггера:

```
ALTER TABLE table_name
ENABLE TRIGGER trigger_name |  ALL;
```

Пошаговый разбор синтаксиса:

1. После ключевых слов `ALTER TRIGGER` указывается имя таблицы триггера, который вы хотите включить;
2. После ключевого слова `ENABLE TRIGGER` указывается имя триггера, который вы хотите включить;
3. Используйте `ALL` опцию, если вы хотите включить все триггеры, связанные с таблицей.

Пример включения триггера:

```
ALTER TABLE employees
ENABLE TRIGGER last_name_changes;
```

ИЛИ

```
ALTER TABLE employees
ENABLE TRIGGER ALL;
```

Синтаксис отключения триггера:

```
ALTER TABLE table_name
DISABLE TRIGGER trigger_name | ALL;
```

Пошаговый разбор синтаксиса:

1. После ключевых слов `ALTER TRIGGER` указывается имя таблицы триггера, который вы хотите отключить;
2. После ключевого слова `DISABLE TRIGGER` указывается имя триггера, который вы хотите отключить;
3. Используйте `ALL` опцию, если вы хотите отключить все триггеры, связанные с таблицей.

Пример отключения триггера:

```
ALTER TABLE employees
DISABLE TRIGGER last_name_changes;
```

Задача для применения триггера: в таблице необходимо поменять цену (amount), а новое значение, которое мы введем увеличить на 20%.

Решение (MySQL):

```
DELIMITER //
CREATE trigger before_update_amt
before update on payment
for each row
BEGIN
set New.amount = New.amount * 1.2;
END //
DELIMITER;
```

ИЛИ (вариант с before insert):

```
DELIMITER //
CREATE trigger before_insert_amt
before insert on payment
for each row
BEGIN
set New.amount = New.amount * 1.2;
```

```
END //
DELIMITER;
```

Оконные функции

Оконная функция – функция, которая работает с выделенным набором строк (окно или партиция) и выполняет вычисление для этого набора строк в отдельном столбце.

Партиция (окно из набора строк) – это набор строк, указанный для оконной функции по одному или группе столбцов таблицы.

Классы оконных функций:

- Агрегирующие;
- Ранжирующие;
- Функции смещения.

Синтаксис оконной функции:

function_name (column_name) --> имя оконной функции одного из классов

over (, --> ключевое слово определения оконной функции

partition by (column_name), --> определение партиций по колонкам

order by (column_name), --> сортировка вычисления для оконной функции

[frame_clause] --> указание фрейма для партиции

)

Правило обработки строк (указание фрейма для партиции):

- range between – диапазон строк, которые обрабатываются;
- unbounded preceding (безграничные предыдущие) – диапазон строк до начала окна;
- unbounded following (безграничные следующие) – диапазон строк до конца окна;
- current row (текущая строка) – диапазон строк до текущей строки в окне.

Агрегирующие оконные функции (возвращают значение, полученное путем арифметических вычислений):

- SUM();

- MAX();
- MIN();
- MAX();
- AVG();
- COUNT().

Ранжирующие оконные функции (позволяют получить порядковые номера записей в окне):

- RANK() – возвращает ранг каждой строки результирующего набора данных **с** промежутками в значениях ранжирования;
- DENSE_RANK() – возвращает ранг каждой строки результирующего набора **без** промежутков в значениях ранжирования;
- ROW_NUMBER() – возвращает номер строки по указанному окну;
- NTILE() – функция, которая делит результирующий набор на группы по определённому столбцу. Количество групп указывается в качестве параметра.

Функции смещения (возвращают значение из другой строки окна):

- FIRST_VALUE() – функция, которая позволяет выбрать первую строку в упорядоченном наборе строк;
- LAST_VALUE() – функция, которая позволяет выбрать последнюю строку в упорядоченном наборе строк;
- LAG() – возвращает значения из предыдущей строки в таблице;
- LEAD() – возвращает значения из следующей строки в таблице.

Примеры агрегирующих оконных функций:

Приведенный ниже код показывает id каждого продукта, цену за единицу, цену самого дорогого продукта, а также процентное отношение текущей цены продукта к цене самого дорогого продукта.

```
SELECT p.product_id,
p.unit_price,
MAX(p.unit_price) OVER() AS max_unit_price,
round(p.unit_price * 100.0 / MAX(p.unit_price) OVER()) AS percentage_of_max
FROM products
```

Приведенный ниже код показывает отклонение цены по каждому продукту в категории от средней цены в категории:

```
select *,
unit_price - avg(unit_price) over (partition by product_id),
avg(unit_price) over (partition by product_id)
from order_details od
```

Нормализация БД

Нормализация – исключение избыточности данных и лишних зависимостей при описании свойств объектов.

Всего нормальных форм более 6 (с учетом промежуточных), но особо важно понимать и помнить о первых 3-х:

1-ая нормальная форма (1 НФ):

- В каждой строке должно быть только одно значение для каждого столбца;
- Не должно быть повторяющихся строк.

2-ая нормальная форма (2 НФ):

- Таблица в 1-ой НФ;
- Каждый неключевой столбец (столбец не является PK) неприводимо зависит от каждой части составного ключа.

3-ая нормальная форма (3 НФ):

- Таблица во 2-ой НФ;
- Нет нетранзитивных зависимостей между неключевыми признаками.

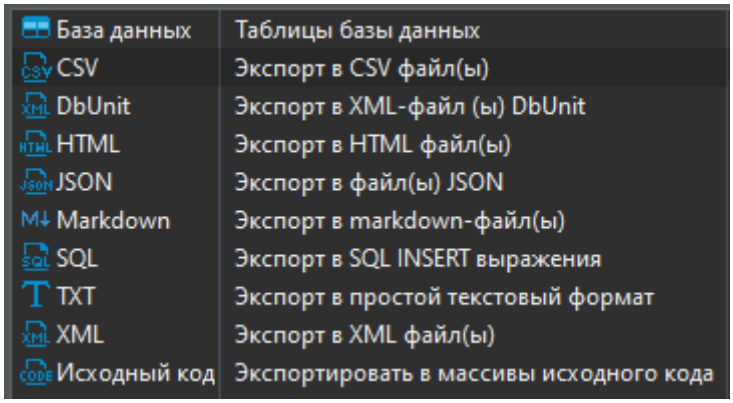
Пример нормализации: https://stepik.org/media/attachments/lesson/1073452/%D0%9D%D0%A4.xlsx

Денормализация — намеренное снижение или нарушение форм нормализации базы данных. Требуется, чтобы ускорить чтение из базы за счет добавления избыточных данных. Так происходит потому, что теория нормальных форм не всегда применима на практике.

Экспорт данных, создание дампов

При необходимости получения данных из базы данных в различных форматах (csv, txt, sql, xlsx и др.) необходимо выполнить экспорт данных.

Форматы экспорта данных в DBeaver:



Варианты экспорта данных:

- Написать запрос и в нижней части интерфейса нажать на "Экспорт данных";
- Выбрать таблицу, нажать правой кнопкой мыши и выбрать "Экспорт данных".

Дамп БД – файл, содержащий инструкции языка SQL, которые создают точную копию структуры и содержимого базы данных. Дамп может понадобиться для переноса содержимого на другой сервер, с локального компьютера на сервер, либо просто чтобы сделать резервную копию перед какими-либо изменениями на сайте.