**7 На какие группы делятся операторы SQL**

<https://ru.wikipedia.org/wiki/SQL>

Операторы SQL делятся на:

* + операторы определения данных ([Data Definition Language](https://ru.wikipedia.org/wiki/Data_Definition_Language), DDL):
    - [CREATE](https://ru.wikipedia.org/w/index.php?title=Create_(SQL)&action=edit&redlink=1) создаёт объект базы данных (саму базу, таблицу, [представление](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)), пользователя и так далее),
    - [ALTER](https://ru.wikipedia.org/w/index.php?title=Alter_(SQL)&action=edit&redlink=1) изменяет объект,
    - [DROP](https://ru.wikipedia.org/w/index.php?title=Drop_(SQL)&action=edit&redlink=1) удаляет объект;
  + операторы манипуляции данными ([Data Manipulation Language](https://ru.wikipedia.org/wiki/Data_Manipulation_Language), DML):
    - [SELECT](https://ru.wikipedia.org/wiki/Select_(SQL)) выбирает данные, удовлетворяющие заданным условиям,
    - [INSERT](https://ru.wikipedia.org/wiki/Insert_(SQL)) добавляет новые данные,
    - [UPDATE](https://ru.wikipedia.org/wiki/Update_(SQL)) изменяет существующие данные,
    - [DELETE](https://ru.wikipedia.org/wiki/Delete_(SQL)) удаляет данные;
  + операторы определения доступа к данным ([Data Control Language](https://ru.wikipedia.org/wiki/Data_Control_Language), DCL):
    - [GRANT](https://ru.wikipedia.org/w/index.php?title=Grant_(SQL)&action=edit&redlink=1) предоставляет пользователю (группе) разрешения на определённые операции с объектом,
    - [REVOKE](https://ru.wikipedia.org/w/index.php?title=Revoke_(SQL)&action=edit&redlink=1) отзывает ранее выданные разрешения,
    - [DENY](https://ru.wikipedia.org/w/index.php?title=Deny_(SQL)&action=edit&redlink=1) задаёт запрет, имеющий приоритет над разрешением;
  + операторы управления [транзакциями](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D1%8F_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) ([Transaction Control Language](https://ru.wikipedia.org/wiki/Transaction_Control_Language), TCL):
    - [COMMIT](https://ru.wikipedia.org/wiki/Commit_(SQL)) применяет транзакцию,
    - [ROLLBACK](https://ru.wikipedia.org/wiki/Rollback_(SQL)) откатывает все изменения, сделанные в контексте текущей транзакции,
    - [SAVEPOINT](https://ru.wikipedia.org/wiki/Savepoint_(SQL)) делит транзакцию на более мелкие участки.

**8 В чем особенность типа данных serial?**

<https://metanit.com/sql/postgresql/2.3.php>

serial: представляет автоинкрементирующееся числовое значение, которое занимает 4 байта и может хранить числа от 1 до 2147483647. Значение данного типа образуется путем автоинкремента значения предыдущей строки. Поэтому, как правило, данный тип используется для определения идентификаторов строки.

**9 Как создать базу данных shop и таблицу products, содержащую атрибуты:**

* + ●  Id, автоинкрементирующееся значение
  + ●  Название продукта, строка переменной длины, уникальное, не NULL
  + ●  Цена, число с плавающей точкой, не NULL
  + ●  Количество продуктов на складе, целое число, по умолчанию 0, не NULL

Для создания базы данных используется команда CREATE DATABASE, после которой указывается название базы данных. Например:

*CREATE DATABASE usersdb;*

Для создания таблиц применяется команда CREATE TABLE, после которой указывается название таблицы. Также с этой командой можно использовать ряд операторов, которые определяют столбцы таблицы и их атрибуты. Общий синтаксис создания таблицы выглядит следующим образом:

CREATE TABLE название\_таблицы

(название\_столбца1 тип\_данных атрибуты\_столбца1,

название\_столбца2 тип\_данных атрибуты\_столбца2,

................................................

название\_столбцаN тип\_данных атрибуты\_столбцаN,

);

Например:

CREATE TABLE Customers

(

Id SERIAL PRIMARY KEY,

ProductName CHARACTER VARYING(30) UNIQUE NOT NULL,

Price real NOT NULL,

Count INTEGER NOT NULL DEFAULT 0

)

**10 Как из таблицы customers, содержащей атрибуты id, firstname, lastname, age с помощью запроса получить всех совершеннолетних пользователей?**

SELECT \* FROM customer WHERE age >= 18

**11 Как из таблицы customers, содержащей атрибуты id, firstname, lastname с помощью запроса получить всех пользователей, имя которых начинается на A?**

SELECT \* FROM customer WHERE firstname LIKE 'A%'

**12 Как найти число строк в таблице?**

SELECT COUNT(id) FROM customer WHERE firstname LIKE 'A%'

**13 Как в таблице users, содержащей атрибуты id, firstname, lastname, age   
отсортировать пользователей по возрасту в порядке убывания?**

SELECT \* FROM users ORDER BY age DESC

**14 Измените имя пользователя с Tom на Andy в таблице users, содержащей атрибуты   
id, firstname, lastname.**

UPDATE users

SET firstname = 'Andy'

WHERE firstname = 'Tom'

**15 Какие ограничения вы знаете, как они работают и указываются?**

<https://metanit.com/sql/postgresql/2.4.php>

**Ограничения столбцов и таблиц**

**PRIMARY KEY** - делает столбец первичным ключом.  Например, при создании таблицы:

CREATE TABLE Customers

(

Id SERIAL **PRIMARY KEY**,

FirstName CHARACTER VARYING(30) **NOT NULL**,

LastName CHARACTER VARYING(30) **NOT NULL**,

Email CHARACTER VARYING(30) **UNIQUE CHECK**(Email !=''),

Phone CHARACTER VARYING(20) **UNIQUE CHECK**(Phone !='')

Age INTEGER **DEFAULT** 18 **CHECK**(Age >0 AND Age < 100)

**UNIQUE**(Email)

)

**UNIQUE** - требует чтобы столбец имел уникальное значение. В примере выше столбцы, которые представляют электронный адрес и телефон, будут иметь уникальные значения. И мы не сможем добавить в таблицу две строки, у которых будут одинаковые номера телефонов или одинаковые email-адреса.

**NULL и NOT NULL** - Чтобы указать, может ли столбец принимать значение NULL, при определении столбца ему можно задать атрибут NULL или NOT NULL. Если этот атрибут явным образом не будет использован, то по умолчанию столбец будет допускать значение NULL. Исключением является тот случай, когда столбец выступает в роли первичного ключа - в этом случае по умолчанию столбец имеет значение NOT NULL.

**DEFAULT** - Атрибут DEFAULT определяет значение по умолчанию для столбца. Если при добавлении данных для столбца не будет предусмотрено значение, то для него будет использоваться значение по умолчанию.

**CHECK** - Ключевое слово CHECK задает ограничение для диапазона значений, которые могут храниться в столбце.

**16 Какое ограничение необходимо добавить в таблицу todos, чтобы установить   
отношение 1:N (один ко многим) между таблицами users (с первичным ключом id) и   
todos?**

<https://metanit.com/sql/postgresql/2.5.php>

The FOREIGN KEY (внешний ключ) является ограничением, которое направлено на предотвращения действий, которые могут разрушить связи между таблицами.

The FOREIGN KEY (внешний ключ) это поле (или коллекция полей), которое находится в так называемой зависимой\дочерней\подчинённой таблице (referencing table) и указывает на один из столбцов так называемой главной (master) таблицы (referenced table). Причём, как правило, внешний ключ указывает на первичный ключ в главной таблице.

CREATE TABLE users

(

Id SERIAL PRIMARY KEY,

FirstName VARCHAR(20) NOT NULL

);

CREATE TABLE todos

(

Id SERIAL PRIMARY KEY,

UserId INTEGER REFERENCES users (Id),

Quantity INTEGER

);

**17 В чем разница между inner join, outer join и cross join?**

<https://metanit.com/sql/postgresql/6.3.php>

-- Продукты

CREATE TABLE Products

(

Id SERIAL PRIMARY KEY,

ProductName VARCHAR(30) NOT NULL,

Company VARCHAR(20) NOT NULL,

ProductCount INTEGER DEFAULT 0,

Price NUMERIC NOT NULL

);

-- Клиенты

CREATE TABLE Customers

(

Id SERIAL PRIMARY KEY,

FirstName VARCHAR(30) NOT NULL

);

-- Заказы

CREATE TABLE Orders

(

Id SERIAL PRIMARY KEY,

ProductId INTEGER NOT NULL REFERENCES Products(Id) ON DELETE CASCADE,

CustomerId INTEGER NOT NULL REFERENCES Customers(Id) ON DELETE CASCADE,

CreatedAt DATE NOT NULL,

ProductCount INTEGER DEFAULT 1,

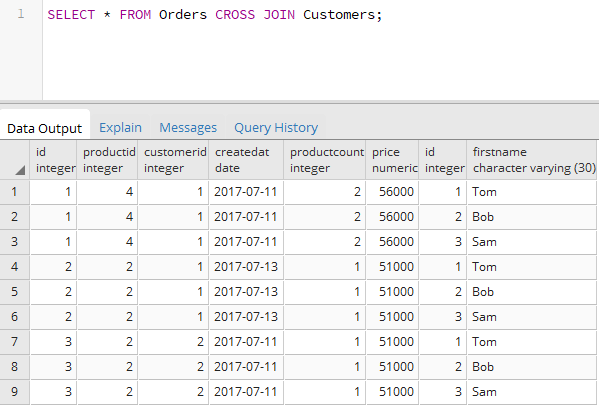
Price NUMERIC NOT NULL

);

**CROSS JOIN** или перекрестное соединение создает набор строк, где каждая строка из одной таблицы соединяется с каждой строкой из второй таблицы. Например, соединим таблицу заказов Orders и таблицу покупателей Customers:

-- CROSS JOIN

SELECT \* FROM Orders CROSS JOIN Customers;



Если мы добавим к CROSS JOIN запросу условие Where, то наш CROSS JOIN превратится в аналог INNER JOIN запроса.

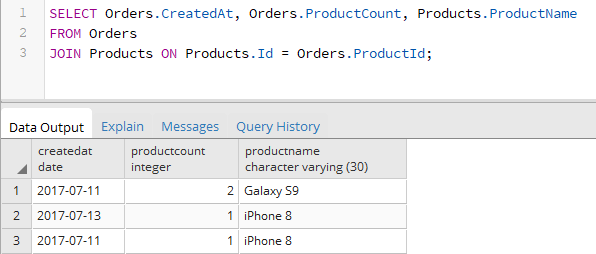
**Используя INNER JOIN** (при этом слово INNER можно опустить), выберем все Заказы и добавим к ним информацию о товарах:

-- INNER JOIN

SELECT Orders.CreatedAt, Orders.ProductCount, Products.ProductName

FROM Orders

JOIN Products ON Products.Id = Orders.ProductId;



**OUTER JOIN** или внешнее соединение позволяет возвратить все строки одной или двух таблиц, которые участвуют в соединении.

Перед оператором JOIN указывается одно из ключевых слов LEFT, RIGHT или FULL (а ключевое слово **OUTER** как правило пропускается), которые определяют тип соединения:

* + LEFT: выборка будет содержать все строки из первой или левой таблицы
  + RIGHT: выборка будет содержать все строки из второй или правой таблицы
  + FULL: выборка будет содержать все строки из обеих таблиц

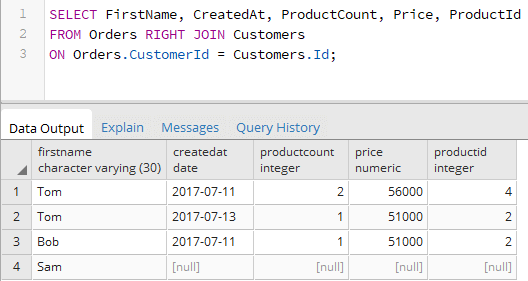
Например, нам надо получить всех Клиентов и их Заказы. Ну а если Клиент ни чего не заказывал, значит просто вывести строку с null-значениями в соответствующих полях, касающихся Заказа.

Воспользуемся для этого RIGHT JOIN

SELECT FirstName, CreatedAt, ProductCount, Price, ProductId

FROM Orders RIGHT JOIN Customers

ON Orders.CustomerId = Customers.Id;



Поскольку один из покупателей из таблицы Customers не имеет связанных заказов из Orders, то соответствующие столбцы, которые берутся из Orders, будут иметь значение NULL.

*Таким образом мы как бы выполняем* ***INNER JOIN****, а потом добавляем ему "выпавшие" строки из правой таблицы Customers*.

Аналогичного результата можно добиться если использовать LEFT JOIN, но просто таблицу Customers переносим из правой части выражения в левую.

**Т.е. операторы RIGHT JOIN и LEFT JOIN отличаются только порядком таблиц!!!**

**Полное соединение (FULL JOIN)** объединяет обе таблицы:

Например у нас есть таблицы

**ТАБЛИЦА City (Города)**

|  |  |
| --- | --- |
| **Id** | **Name** |
| 1 | Москва |
| 2 | Санкт-Петербург |
| 3 | Казань |

**ТАБЛИЦА Person (Люди)**

|  |  |
| --- | --- |
| **Name** | **CityId** |
| Андрей | 1 |
| Леонид | 2 |
| Сергей | 1 |
| Григорий | 4 |

Тогда такой **FULL OUTER JOIN** запрос

**SELECT** \*  
**FROM**  
 Person  
 **FULL OUTER JOIN**  
 City  
 **ON** Person.CityId = City.Id

вернёт результат:

|  |  |  |  |
| --- | --- | --- | --- |
| **Person.Name** | **Person.CityId** | **City.Id** | **City.Name** |
| Андрей | 1 | 1 | Москва |
| Сергей | 1 | 1 | Москва |
| Леонид | 2 | 2 | Санкт-Петербург |
| NULL | NULL | 3 | Казань |
| Григорий | 4 | NULL | NULL |

Т.е. к inner join мы добавим все "выпавшие" записи из правой и левой таблицы, для которых не нашлось соответствия, так что в соответствующих полях будет проставлено null.