**Содержание**

[**Установка и настройка сервера БД PostgreSQL** 2](#_Toc37077035)

[**Для ОС Linux (Debian, Ubuntu)** 2](#_Toc37077036)

[**Для ОС Windows** 2](#_Toc37077037)

[**Проверка связи** 4](#_Toc37077038)

[**Тема №1 - Язык SQL. Проектирование структуры базы данных** 6](#_Toc37077039)

[**1.** **Основные теоретические сведения** 6](#_Toc37077040)

[**2.** **Создание базы данных** 7](#_Toc37077041)

[**3.** **Создание таблиц** 9](#_Toc37077042)

[**4.** **Исправление ошибок** 17](#_Toc37077043)

[**5.** **Создание связей между таблицами** 19](#_Toc37077044)

[**6.** **Контрольные вопросы** 23](#_Toc37077045)

[**Тема №2 - Модификация данных в PostgreSQL.** 24](#_Toc37077046)

[**1.** **Ввод данных** 24](#_Toc37077047)

[**2.** **Удаление данных** 25](#_Toc37077048)

[**3.** **Модификация данных** 25](#_Toc37077049)

[**4.** **Задания для самостоятельной работы** 26](#_Toc37077050)

[**5.** **Контрольные вопросы** 26](#_Toc37077051)

[**Тема №3 - Выборка данных (оператор SELECT).** 27](#_Toc37077052)

[**Использование агрегирующих функций.** 31](#_Toc37077053)

[**Использование числовых функций** 32](#_Toc37077054)

[**Вложенные подзапросы** 51](#_Toc37077055)

[**Использование оператора EXISTS** 58](#_Toc37077056)

**Установка и настройка сервера БД PostgreSQL**

Рассмотрим процесс установки комплекта ПО, необходимого для начала работы с СУБД PostgreSQL. В текущем руководстве используется 11 версия СУБД. Официальное руководство можно просмотреть в брошюре: <https://edu.postgrespro.ru/introbook_v5.pdf>.

**Для ОС Linux (Debian, Ubuntu)**

Для установки PostgreSQL не обходимо выполнить следующие шаги

Импортировать ключ репозитория из https://www.postgresql.org/media/keys/ACCC4CF8.asc:

sudo apt-get install curl ca-certificates gnupg

curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -

Создайте /etc/apt/sources.list.d/pgdg.list. Название вашего дистрибутива codename-pgdg. В примере замените слово buster на название вашего дистрибутива

deb http://apt.postgresql.org/pub/repos/apt buster-pgdg main

Например для ubuntu 18 заменяем buster на bionic, для 16й версии на xenia сооствественно. Ниже приведен пример для Ubuntu18.

deb [http://apt.postgresql.org/pub/repos/apt bionic-pgdg main](http://apt.postgresql.org/pub/repos/apt%20bionic-pgdg%20main\)

Наконец обновим список пакетов и начнем их установку.

sudo apt-get update

sudo apt-get install postgresql-11 pgadmin4

**Для ОС Windows**

Очень важно проверить имена в вашей операционной системе. Убедитесь, что имя компьютера не содержит кириллических символов, иначе сервер БД не сможет запуститься. Примечательно, что с кириллическими символами в имени домашней папки пользователя проблем не возникло, но если проблема останется, просто создайте нового пользователя с латинским названием и попробуйте запустить СУБД от него.

Первым делом, выберем из репозитория (<https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>) подходящую версию СУБД. Предварительно стоит проверить, поддерживает ли ваша операционная система данную версию. Узнать это можно на этой странице: <https://www.postgresql.org/download/windows/>.

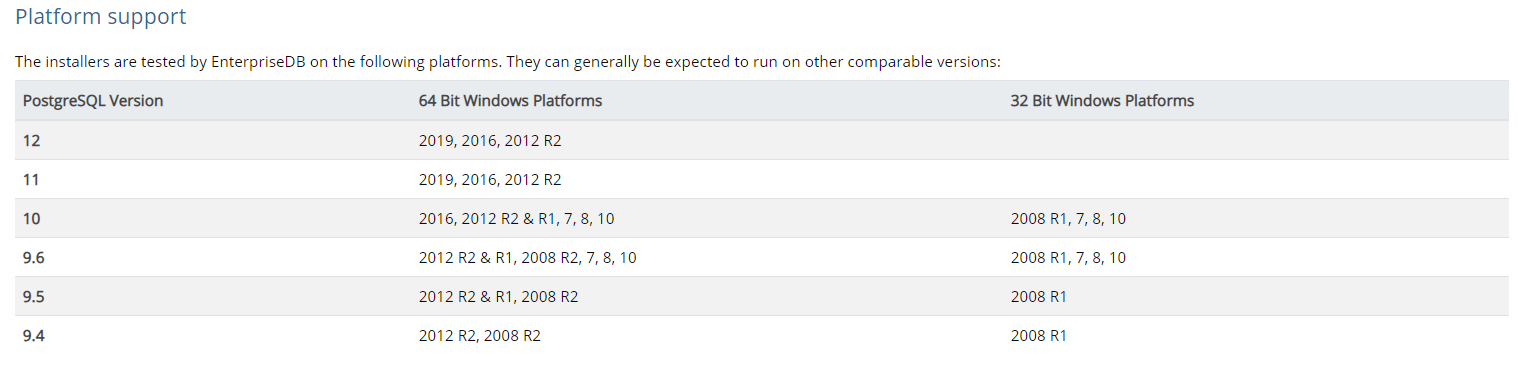


Рисунок 1 – Таблица совместимости ОС и дистрибутивов PostgreSQL

После загрузки установщика запустите его и следуйте инструкциям, путь установки по умолчанию. При установке, рекомендуется не отказываться от PGAdmin, дополнениями можно пренебречь.

После успешной установки СУБД, возможна работа в двух вариантах: через консоль и через pgAdmin. В первом случае необходимо добавить папку с файлами СУБД (например C:\Program Files\PostgreSQL\11\bin) в переменную PATH. После этого в cmd и PowerShell будет доступна команда psql, которая даст возможность работать с PostgreSQL.

Русскоязычные пользователи также должны ознакомиться со специальным разделом, связанным с отображением кириллических символов в консоли. Если возникает предупреждение о несовпадении клиентской и текстовой кодировок, можно решить проблему правкой реестра:

1. Запускаем regedit;
2. Переходим в System/CurrentControlSet/Control/Nls/CodePage;
3. Находим в конце списка элемент OEMCP и задаем ему значение 1251;
4. Перезапускаем ПК.

Теперь консоль запускается с кодировкой, совместимой с PostgreSql.

Если у Вас нет возможности править реестр, попробуйте следующее:

1. Запустите консоль;
2. Нажмите в верхний левый угол её окна;
3. Выберите пункт «Умолчания»;
4. Перейдите на вкладку «Шрифт»;
5. Укажите Lucida Console;
6. Выполните команду ‘chcp 1251’;
7. Выполните команду ‘set PGCLIENTENCODING=WIN1251’;

**Проверка связи**

Попробуйте запустить pgAdmin и авторизоваться под пользователем postgres (пароль пользователя postgres вы задаёте при установке СУБД):

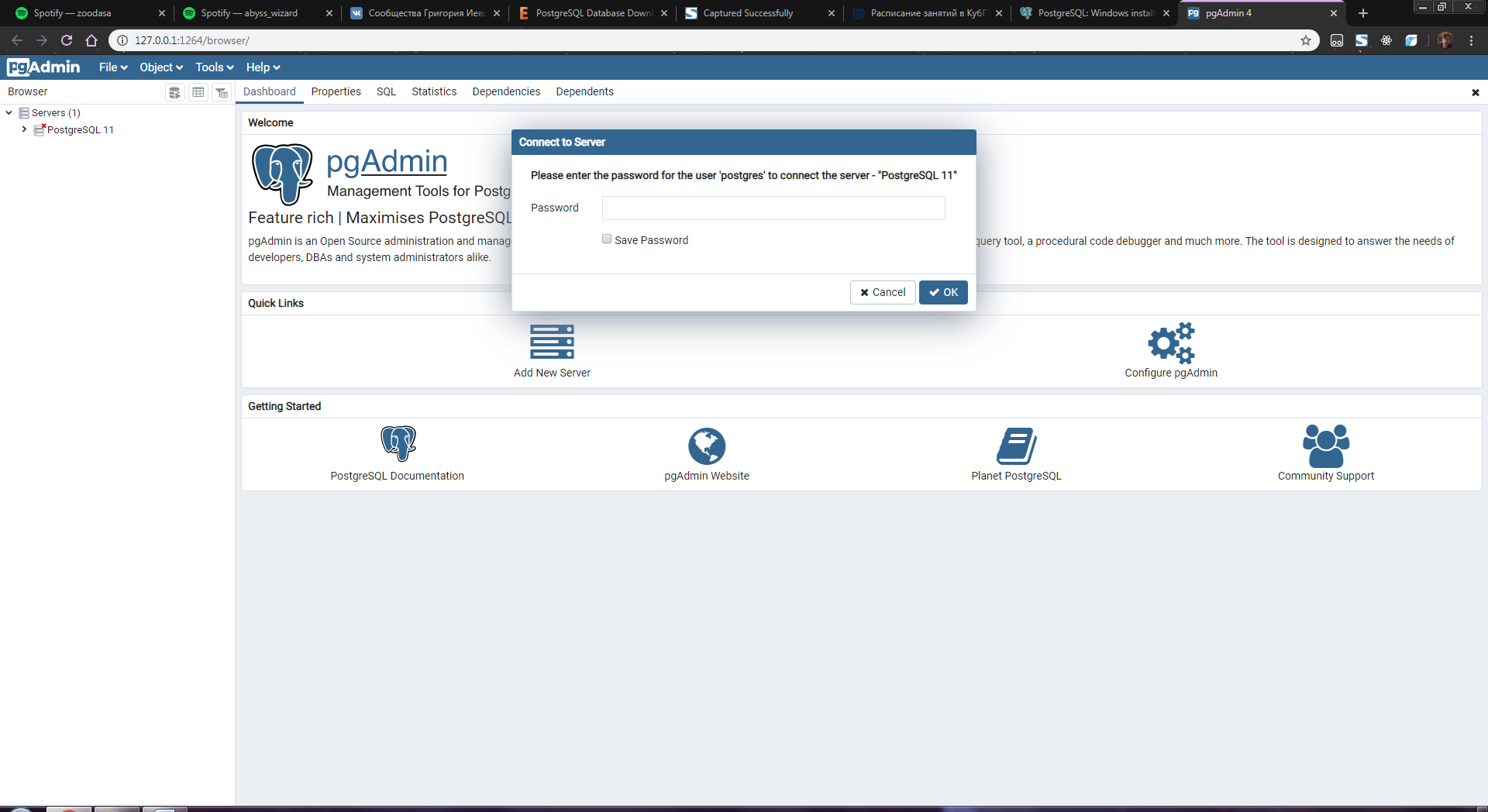


Рисунок 2 – Окно авторизации pgAdmin

Создайте нового пользователя, под которым рекомендуется проводить все действия. Его можно создать в разделе Login/Group Roles через ПКМ – Create – Login/Group Role…

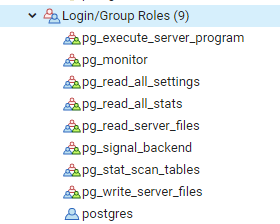


Рисунок 3 – Список групп пользователей в pgAdmin

Нужно в разделе General написать логин пользователя, и в разделе Definition пароль, дальше нажать на “Save”. Также стоит выставить все ползунки кроме “Superuser?” в разделе Privileges на “Yes”. Таким образом, пользователь будет создан.

**Для пользователей Ubuntu**

При работе на Ubuntu обычно возникает проблема, с подключением к серверу. Чтобы это исправить необходимо изменить пароль пользователя с именем 'postgres'. Мы входим в систему под ее именем и даем ей пароль.

$ sudo -u postgres psql

\password

Enter password: ...

...

Выйдите из системы psql, набрав \q или ctrl+d . Затем мы соединяемся как 'postgres'. Часть -h localhost **важна** : она сообщает клиенту psql, что мы хотим подключиться с помощью соединения TCP (которое настроено на использование аутентификации по паролю), а не с помощью соединения PEER (которое не заботится о пароле).

$ psql -U postgres -h localhost

После этого можно переходить к непосредственной работе с PostgreSql.

**Тема №1 - Язык SQL. Проектирование структуры базы данных**

1. **Основные теоретические сведения**

Язык SQL (Structured Query Language - структурированный язык запросов) представляет собой стандартный высокоуровневый язык описания данных и манипулирования ими в системах управления базами данных (СУБД), построенных на основе реляционной модели данных. SQL относится к классу непроцедурных языков программирования. Программа на языке SQL представляет собой простую линейную последовательность операторов языка SQL. Язык SQL в своем чистом виде операторов управления порядком выполнения запросов к БД (типа циклов, ветвлений, переходов) не имеет.

Операторы языка SQL строятся с применением:

* зарезервированных ключевых слов;
* идентификаторов таблиц и столбцов таблиц;
* логических, арифметических и строковых выражений, используемых для формирования критериев поиска информации в БД и для вычисления значений ячеек результирующих таблиц;
* идентификаторов (имен) операций и функций, используемых в выражениях.

В языке SQL не делается различия между прописными и строчными буквами, т.е., например, строки SELECT, Select, select представляют собой одно и то же ключевое слово. Для конструирования имен таблиц и их столбцов допустимо использовать буквы, цифры и знак подчеркивание, но первым символом имени обязательно должна быть буква. Запрещено использование ключевых слов и имен функций в качестве идентификаторов таблиц и имен столбцов. Оператор начинается с ключевого слова, например CREATE - создать, UPDATE - обновить, SELECT - выбрать, и заканчивается знаком точка с запятой. Оператор записывается в свободном формате и может занимать несколько строк. Допустимыми разделителями лексических единиц в операторе являются:

* один или несколько пробелов,
* один или несколько символов табуляции,
* один или несколько символов типа новая строка.

В приводимых в пособии примерах построения SQL-запросов используется база данных в основном одна и та же база данных «Университет». В ряде отдельных случаев запросы в целях наглядности формируются для таблиц, не входящих данную структуру. В этом случае перед выполнением запроса приводится таблица с исходной информацией.

1. **Создание базы данных**

Итак, спроектируем структуру базы данных «Университет». Для этого войдём в панель управления СУБД pgAdmin. Вводим пароль от ПУ (не от сервера БД) и раскрываем вкладку «Серверы», её расположение изображено на рисунке 3.

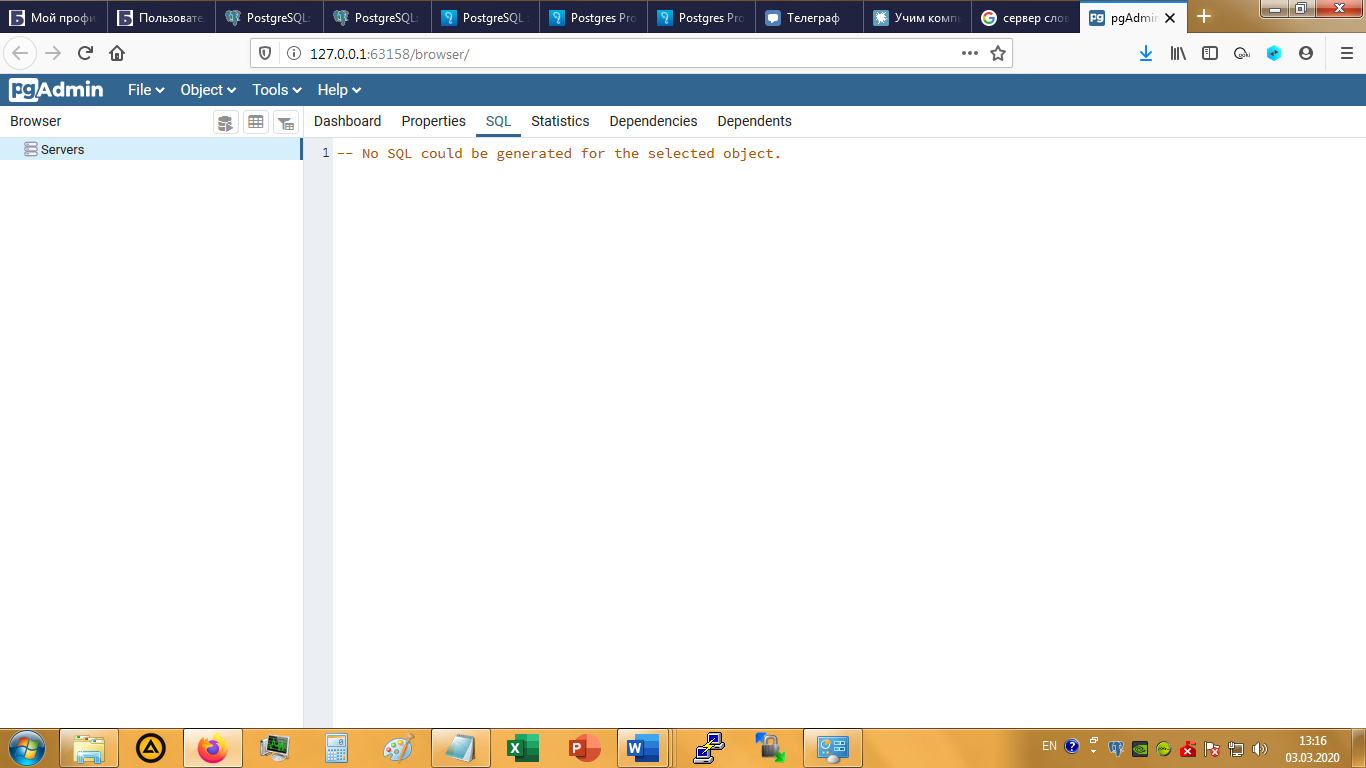


Рисунок 3 – Элемент «Серверы БД»

Кликаем правой кнопкой мыши и выбираем пункт «Создать −> Сервер» (рисунок 4).

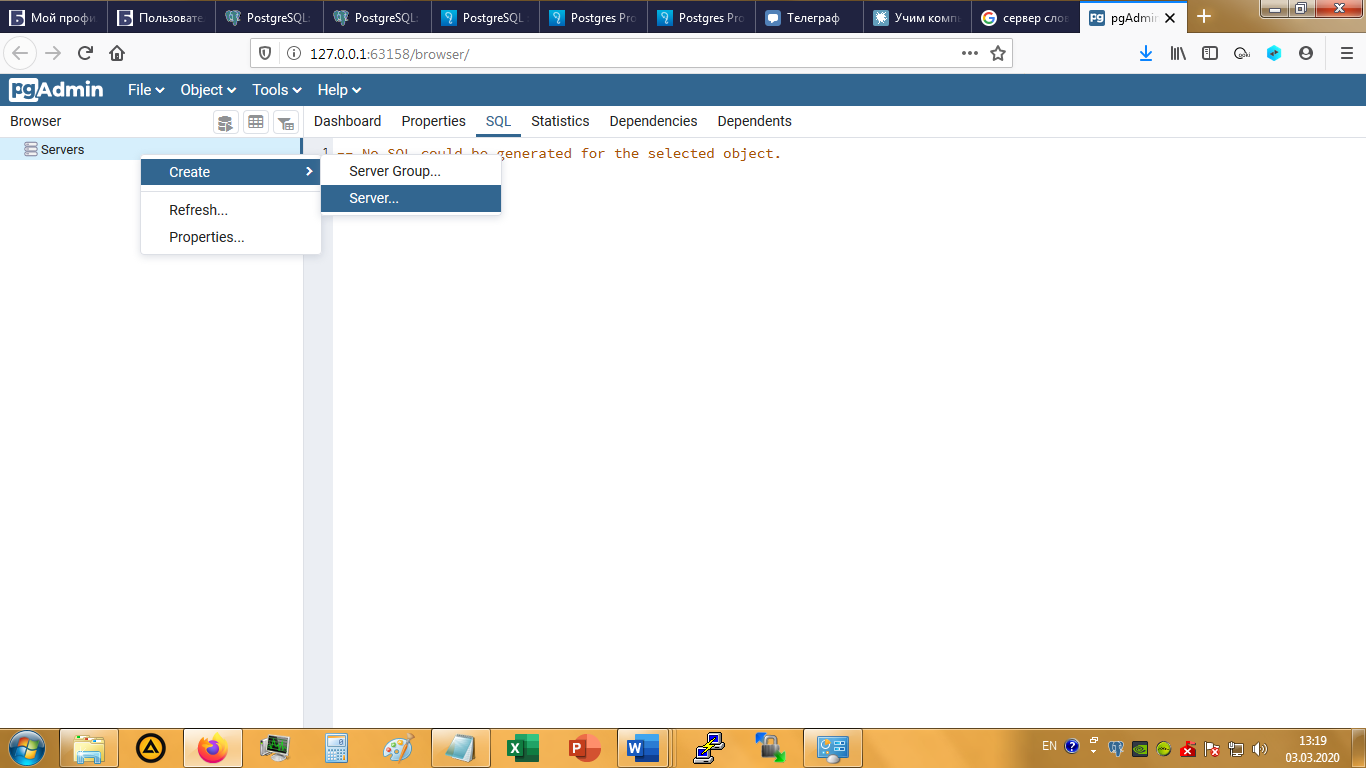


Рисунок 4 – Добавляем сервер БД

В открывшемся окне задаём желаемое имя, затем на вкладке «Подключение» задаём необходимые параметры:

• Хост: localhost;

• Порт, если не задан по умолчанию, 5432;

• Данные пользователя в соответствие с теми, что были заданы при установке

Этого будет достаточно, чтобы подключить наш сервер БД к pgAdmin. В случае успеха на вкладке серверов появится элемент, содержащий одну базу postgres, как показано на рисунке 5.

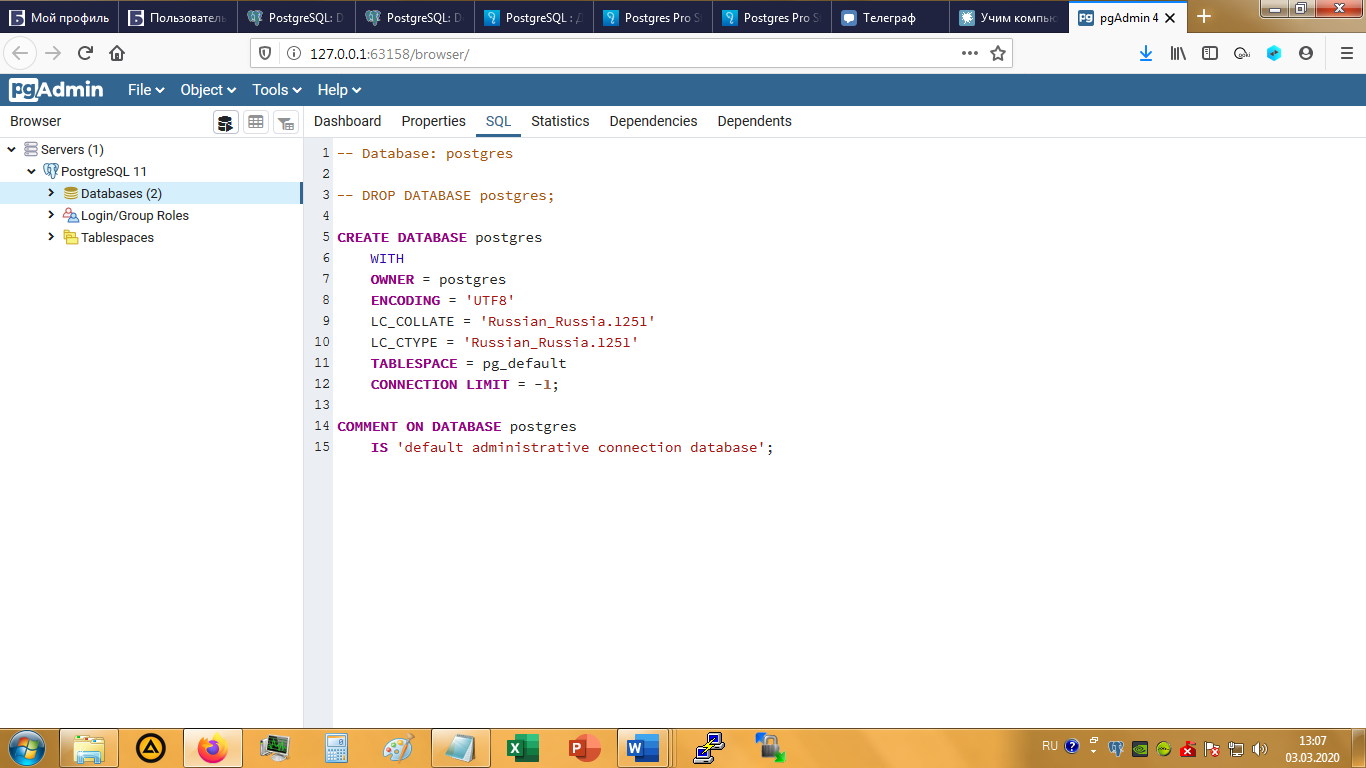


Рисунок 5 – Успешное подключение БД

Когда сервер БД подключен, мы можем создать новую базу данных. Для этого нужно кликнуть на элемент «Базы данных» и выбрать «Создать −> Базу данных». В открывшемся окне задаём имя в нижнем регистре, (аналогично для таблиц, иначе возможны ошибки с именами при работе в psql) владельца, комментарий, кодировку и прочие параметры по мере необходимости. Когда всё готово, нажмите «Сохранить». Все шаги проиллюстрированы на рисунке 6.

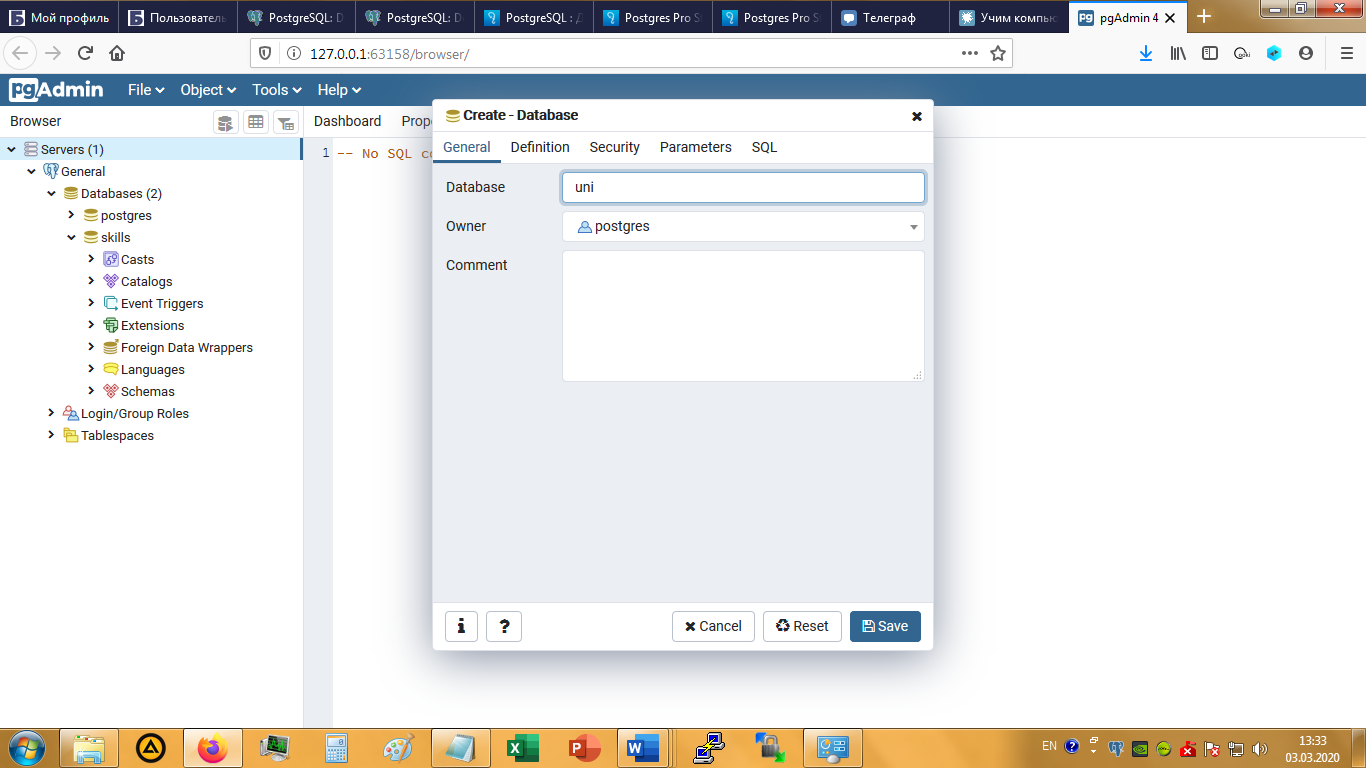


Рисунок 6 – Создание новой БД.

1. **Создание таблиц**

Добавим несколько таблиц в нашу БД. Развернем вкладку созданной БД, в ней находим вкладку «Schemas −> Public −> Tables». Кликаем ПКМ и добавляем таблицу. Задаём ей имя и необходимые поля на вкладке «Колонки» (см. рисунок 7).

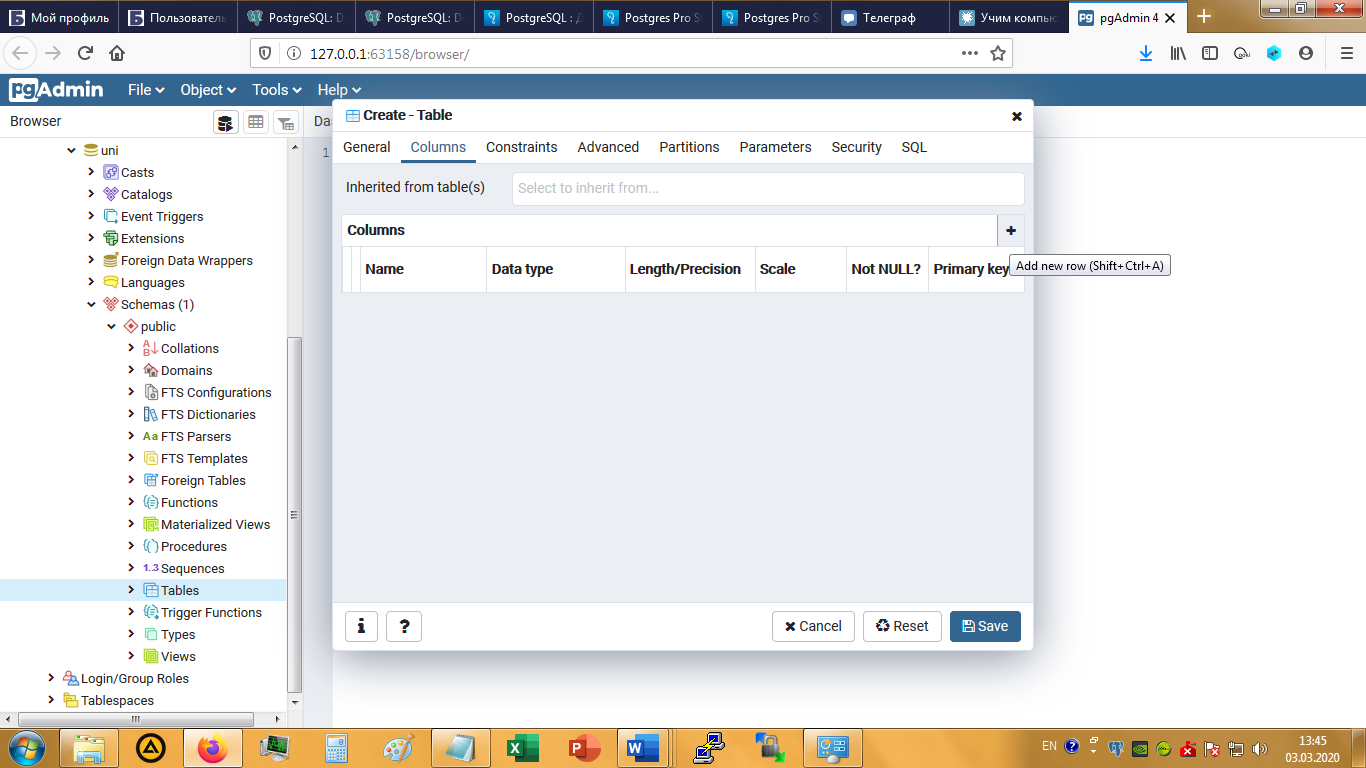


Рисунок 7 – Добавление полей

Чтобы добавить поле в таблицу, нажмите на иконку «Плюс». Задайте имя, тип и параметры поля. Например, создадим поле ID для таблицы «Product». Для этого поля подойдёт тип «Serial», который представляет собой четырёхбайтное целое с автоувеличением (каждый новый элемент будет иметь идентификатор на единицу больше, при удалении элементов освободившиеся диапазоны повторно не заполняются – это позволяет однозначно определять записи, в дальнейшем он будет отображаться по умолчанию как integer). В столбце «NOT NULL?» устанавливается флажок, определяющий возможность оставлять ячейку не заполненной (пустой). Для поля ID это недопустимо, оставим выключенным.

Ключевое поле создается при помощи флажка «Первичный ключ». Под первичным ключом понимают поле или набор полей, однозначно (уникально) идентифицирующих запись. Первичный ключ должен быть минимально достаточным: в нем не должно быть полей, удаление которых из первичного ключа не отразится на его уникальности. Правила хорошего тона при разработке структур баз данных, и чисто практические соображения должны побудить разработчика всегда определять первичный ключ для таблицы базы данных. В подавляющем большинстве случаев поле ID создаётся в каждой таблице и назначается первичным ключом. Поэтому мы зададим ей соответствующий атрибут. Итоговый набор заполненных параметров для поля ID изображен на рисунке 8.

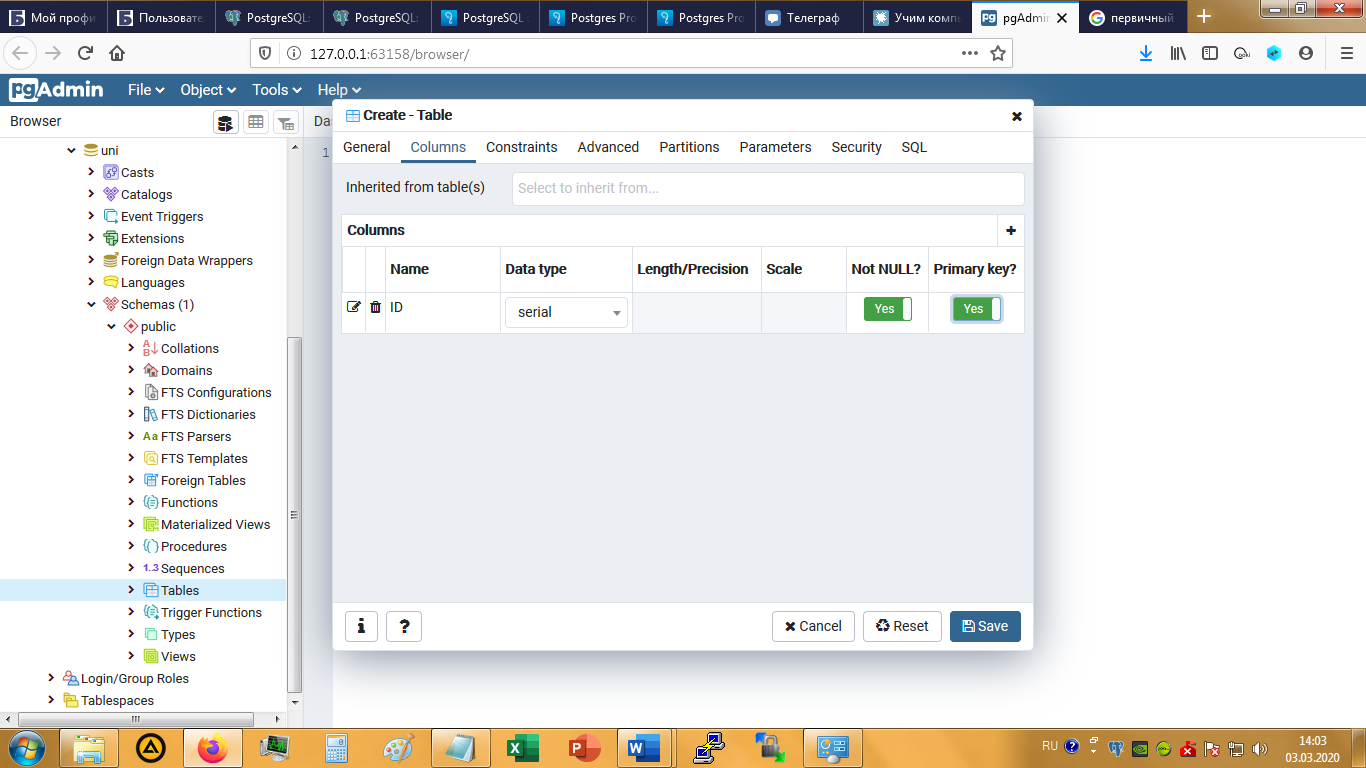


Рисунок 8 – Создание поля ID

Также создадим поля и для них выберем следующие типы данных:

* «ARTICUL» – character varying (длинны 21)
* «ID» - serial (он же integer)
* «NAME» - character varying (длинный 400)
* «WIDTH» - integer
* «LENGTH» - integer
* «IMAGE» - text
* «COMMENT» - text

Итого список полей будет выглядеть примерно таким образом, как на рисунке 9.

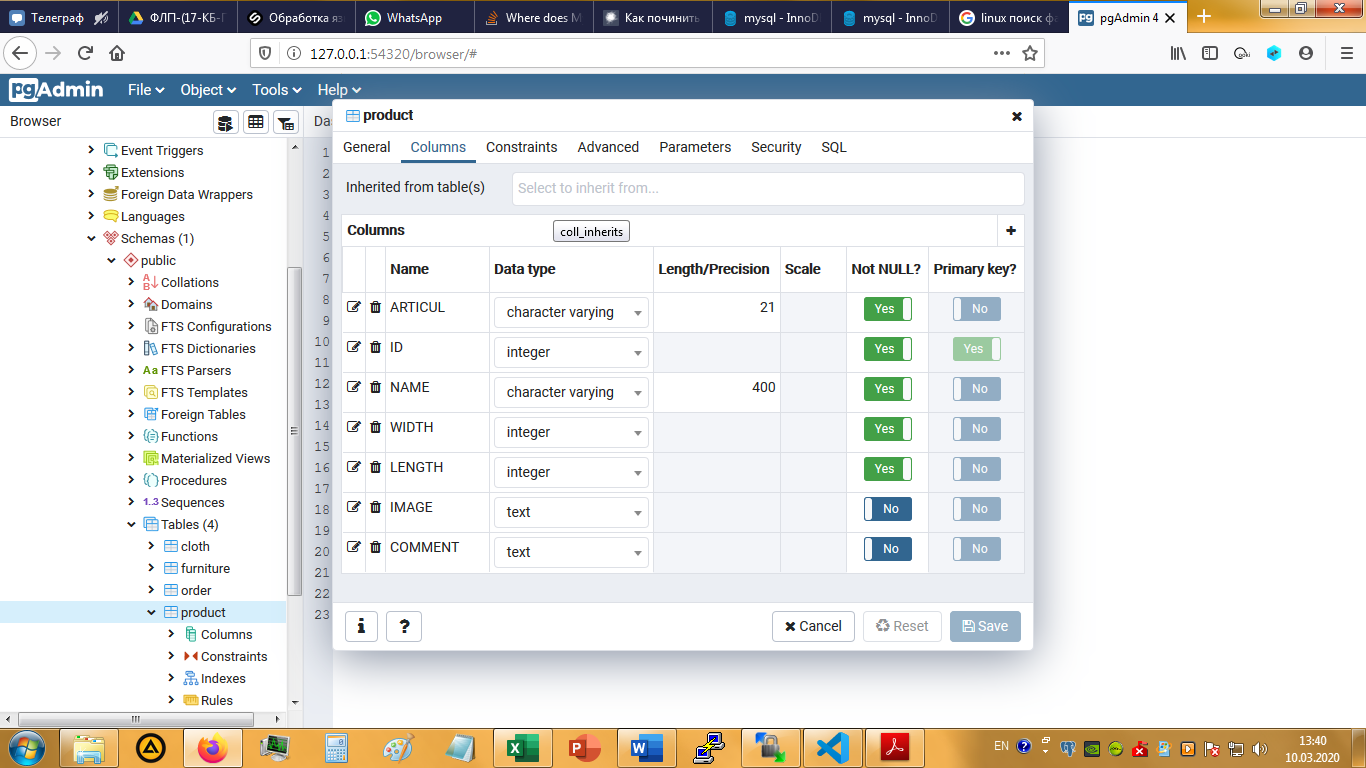


Рисунок 9 – Список полей для таблицы «product»

Когда все поля созданы, нажмём кнопку «Сохранить», если всё указано, таблица будет добавлена в нашу базу данных.

Аналогичным образом создаем остальные таблицы.

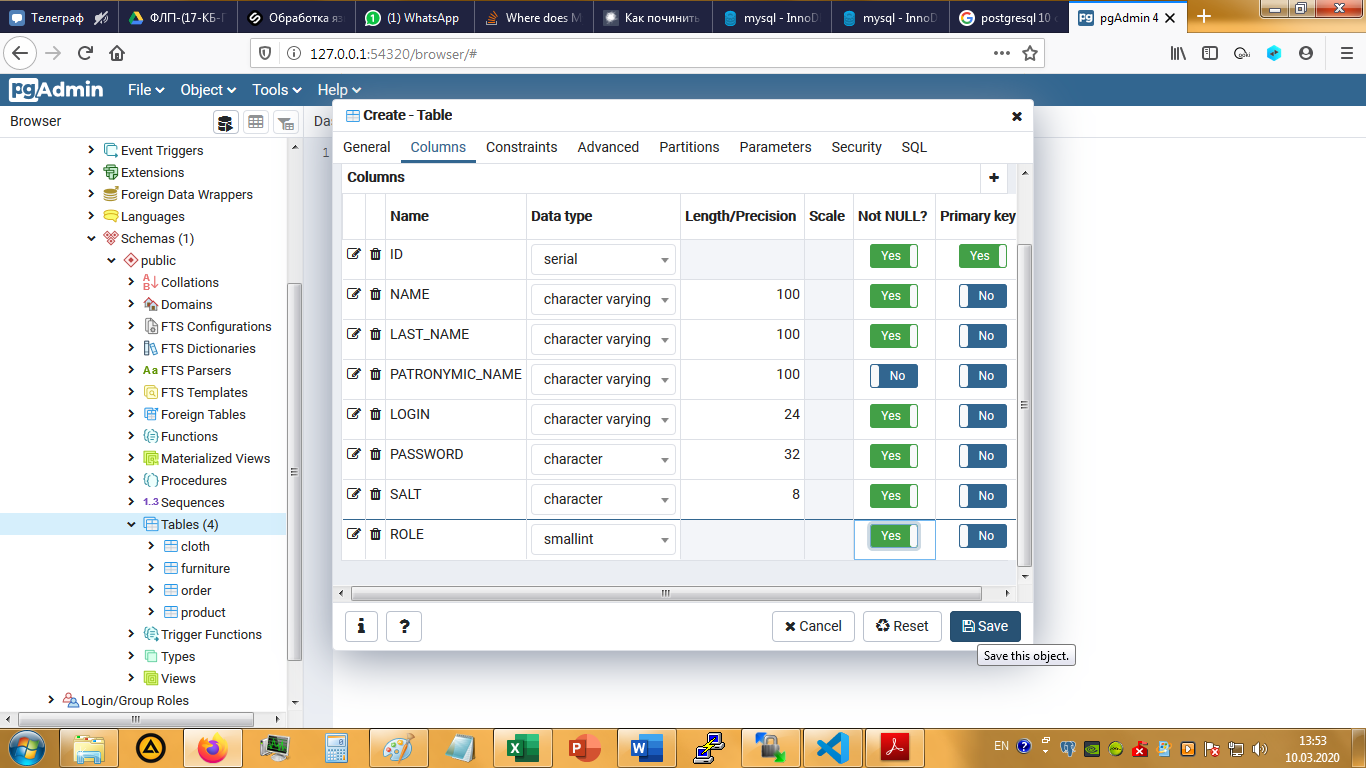


Рисунок 10 – Список полей для таблицы «users»

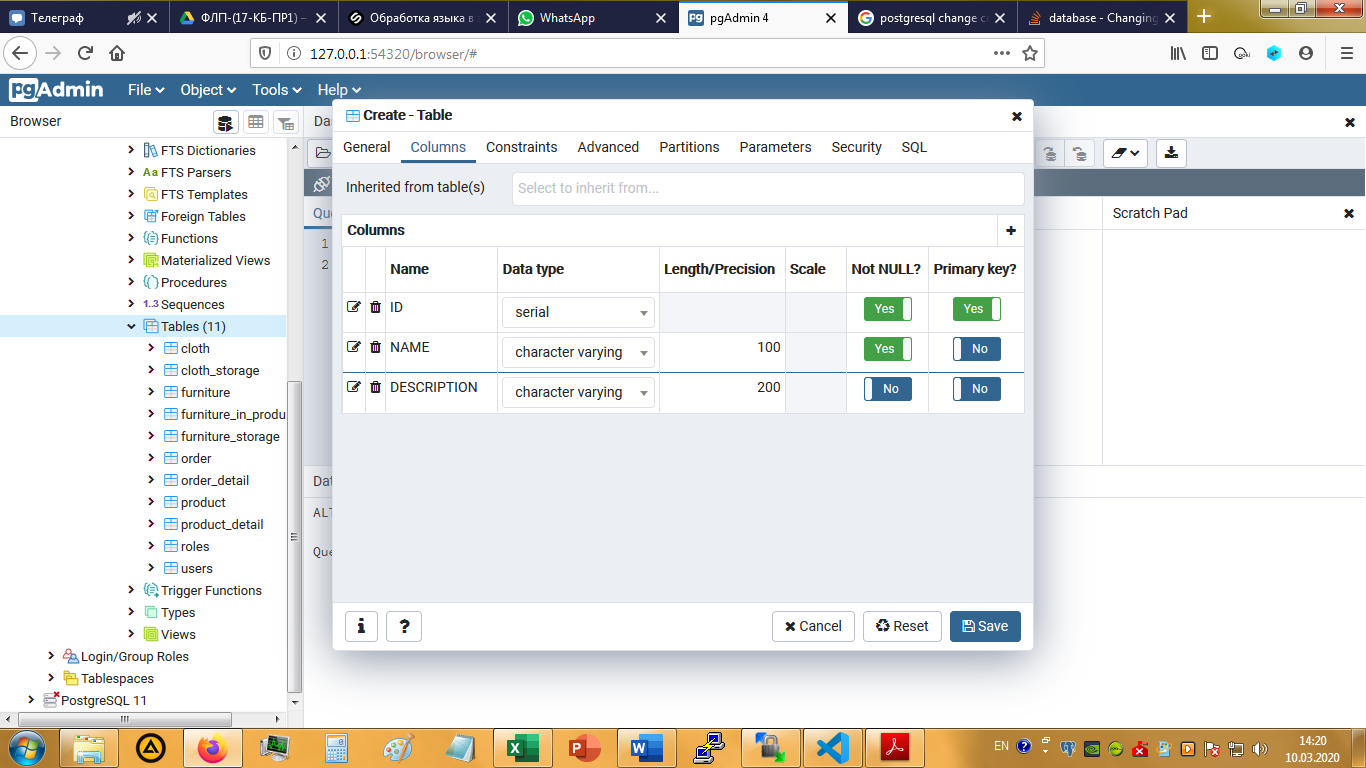


Рисунок 11 – Список полей для таблицы «colors»

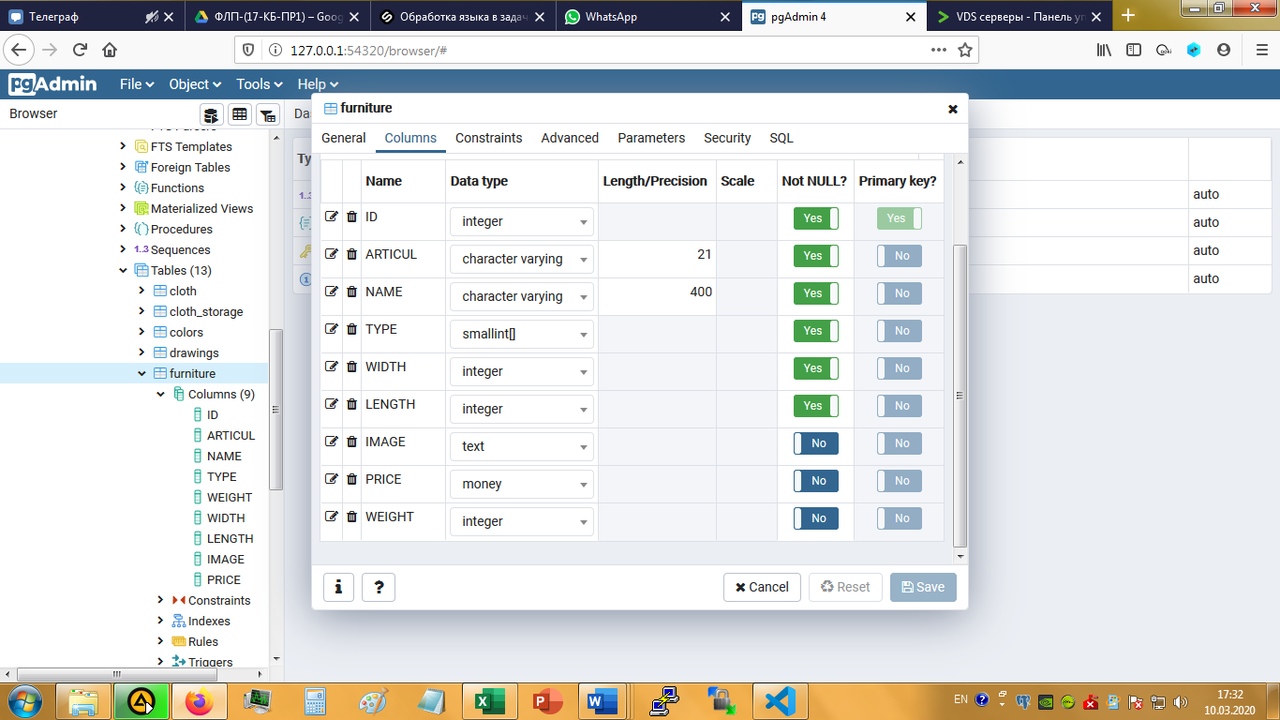


Рисунок 12 – Список полей для таблицы «furniture»

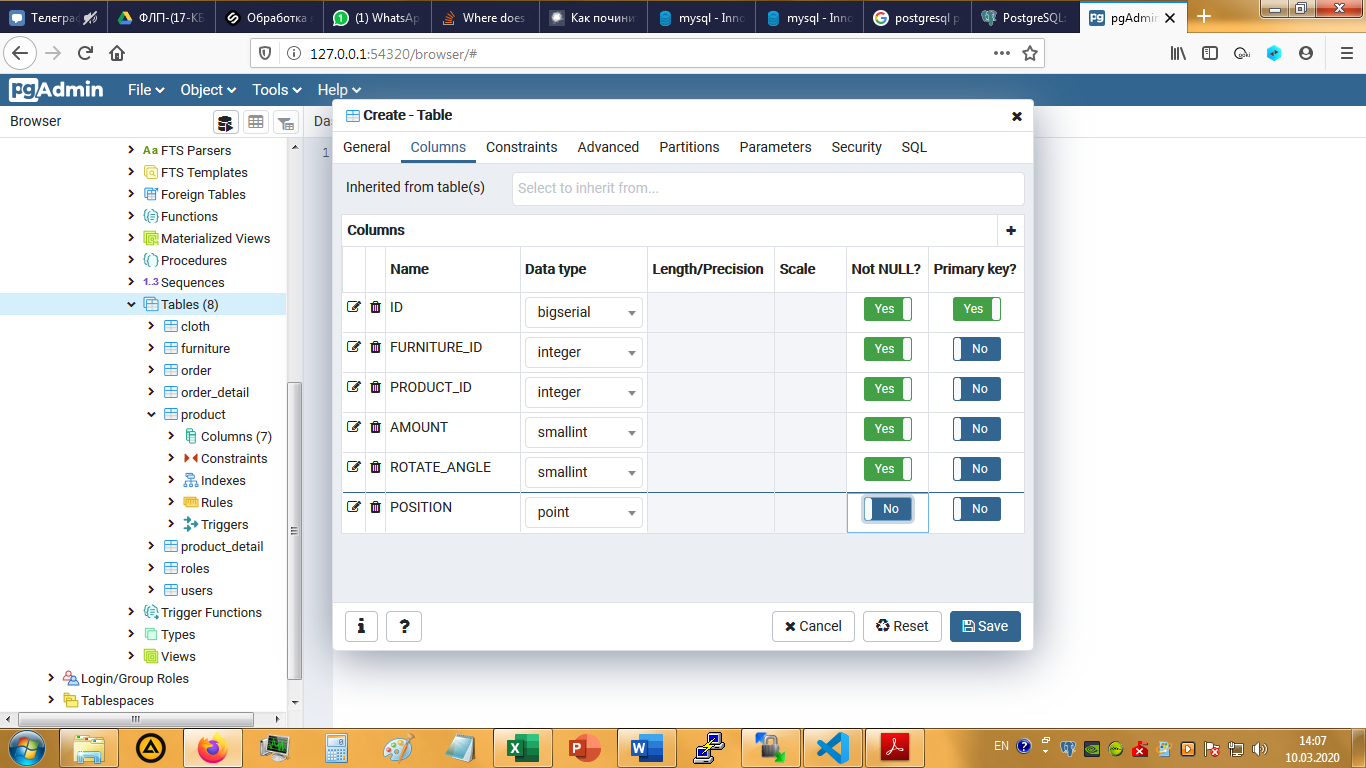


Рисунок 13 – Список полей для таблицы «furniture\_in\_product»

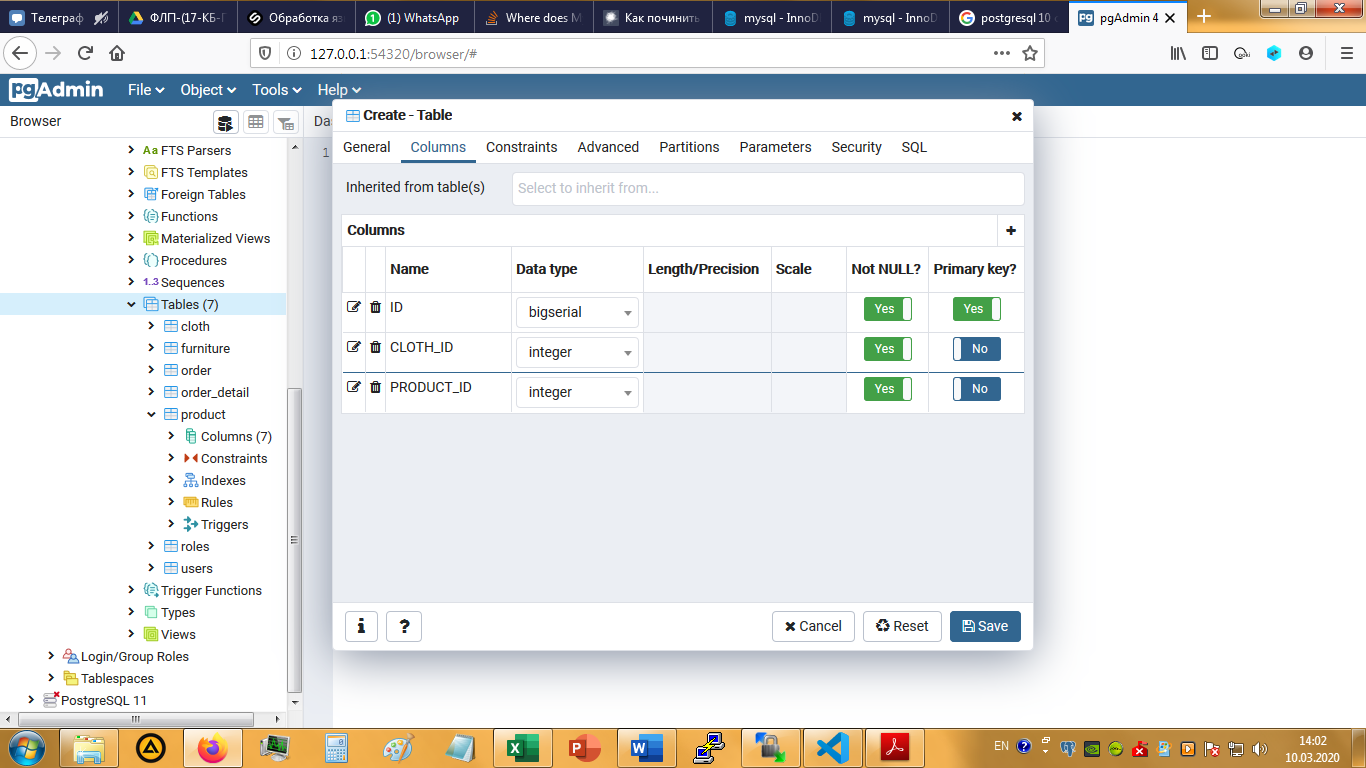


Рисунок 14 – Список полей для таблицы «product\_detail»

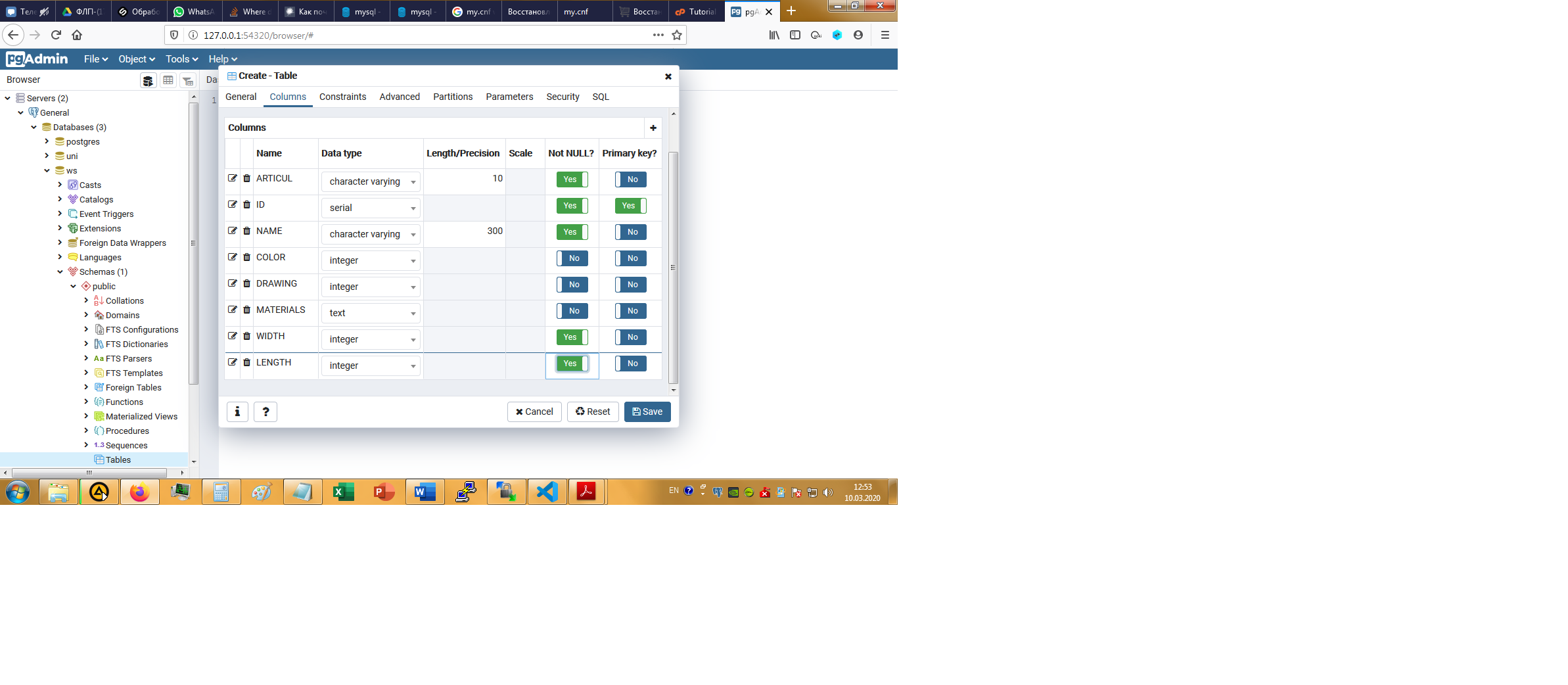


Рисунок 15 – Список полей для таблицы «cloth»

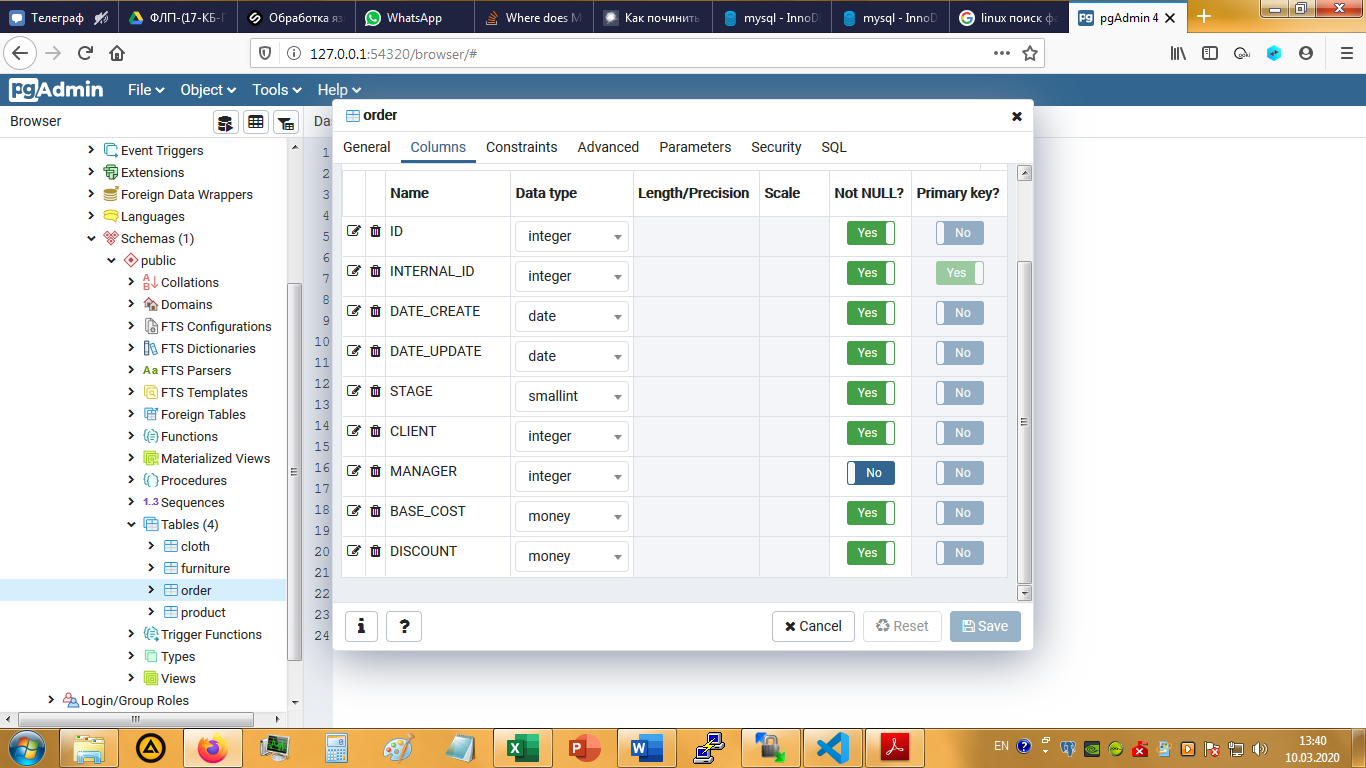


Рисунок 16 – Список полей для таблицы «order»

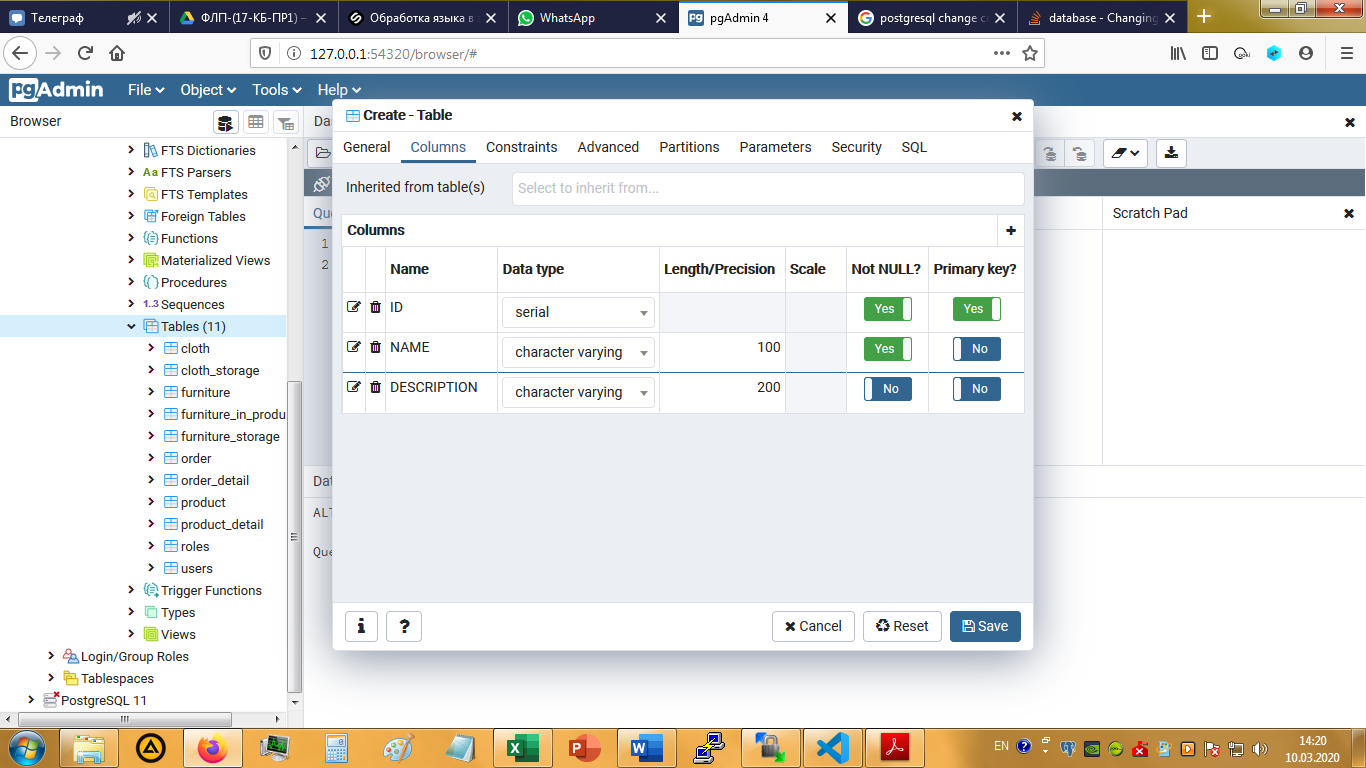


Рисунок 17 – Список полей для таблицы «drawings»

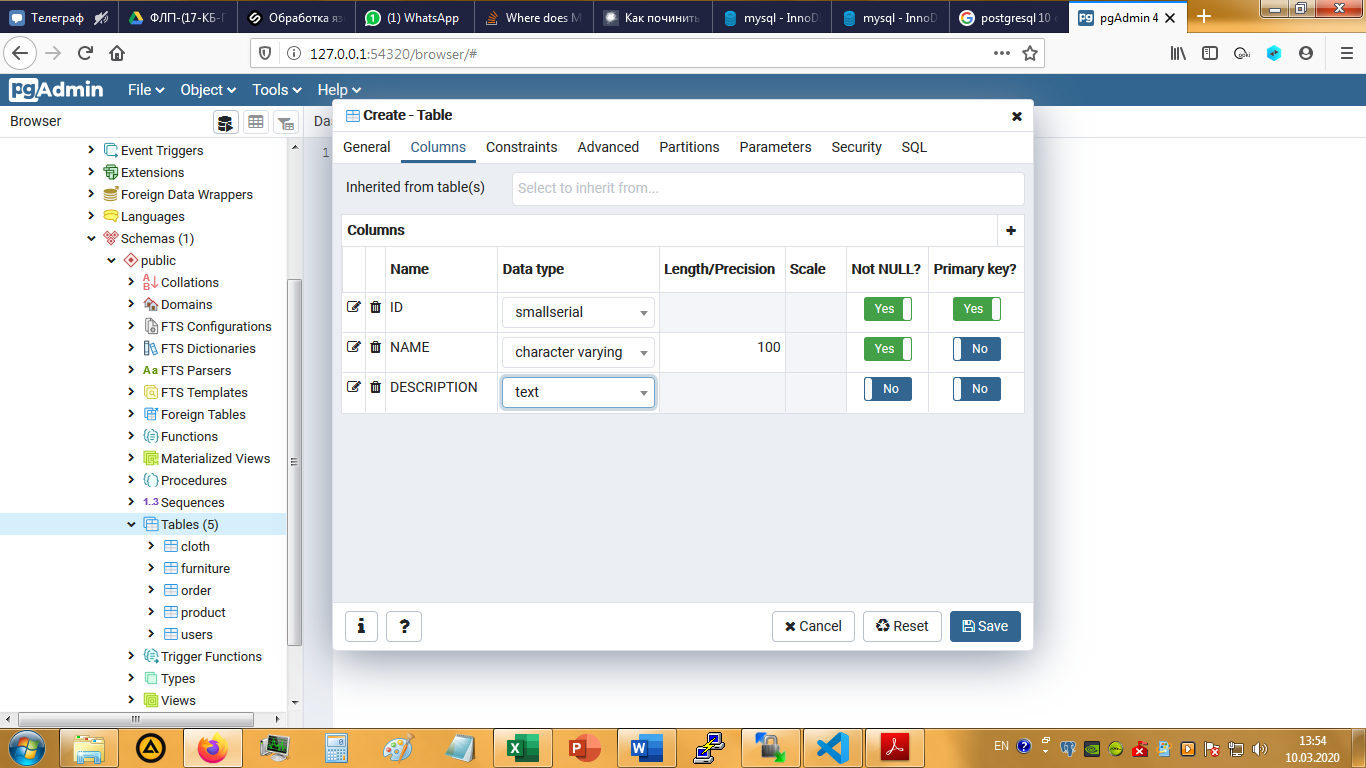


Рисунок 18 – Список полей для таблицы «roles»

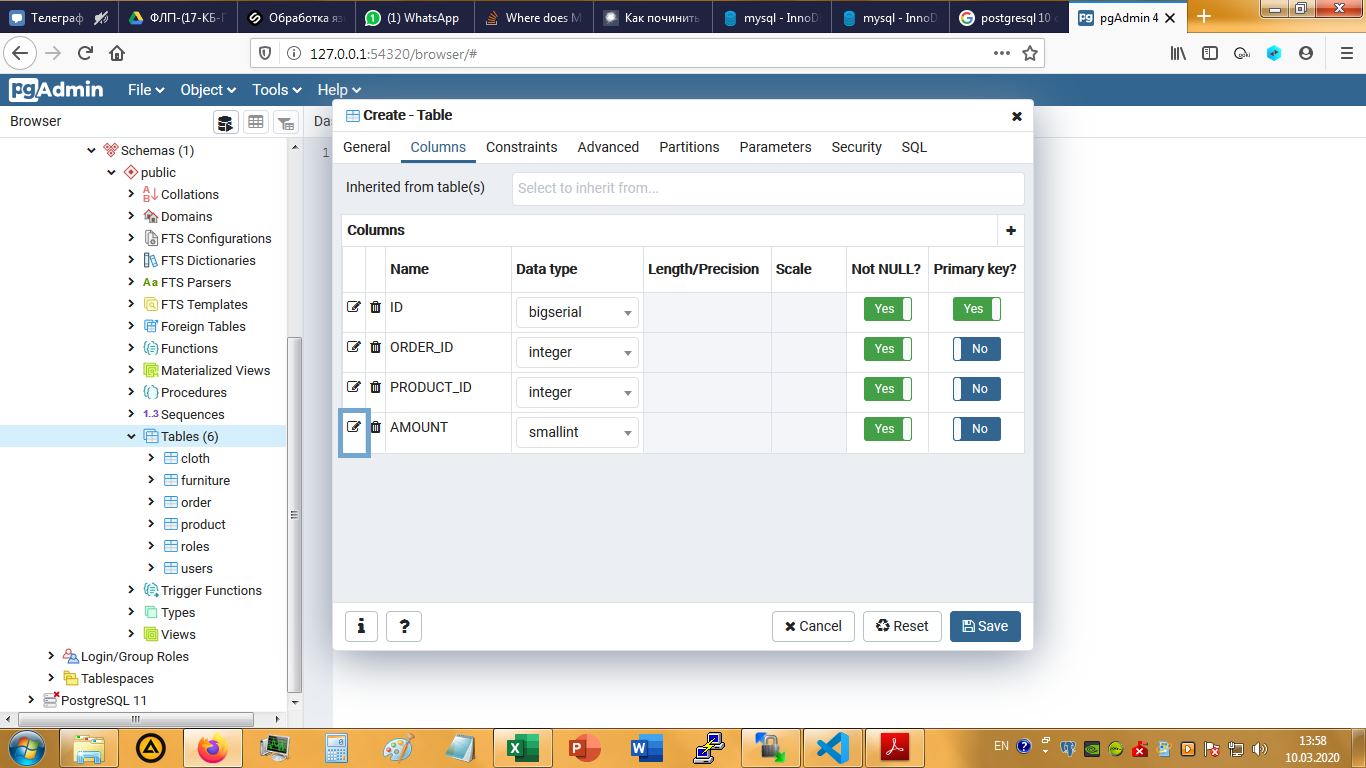


Рисунок 19 – Список полей для таблицы «order\_detail»

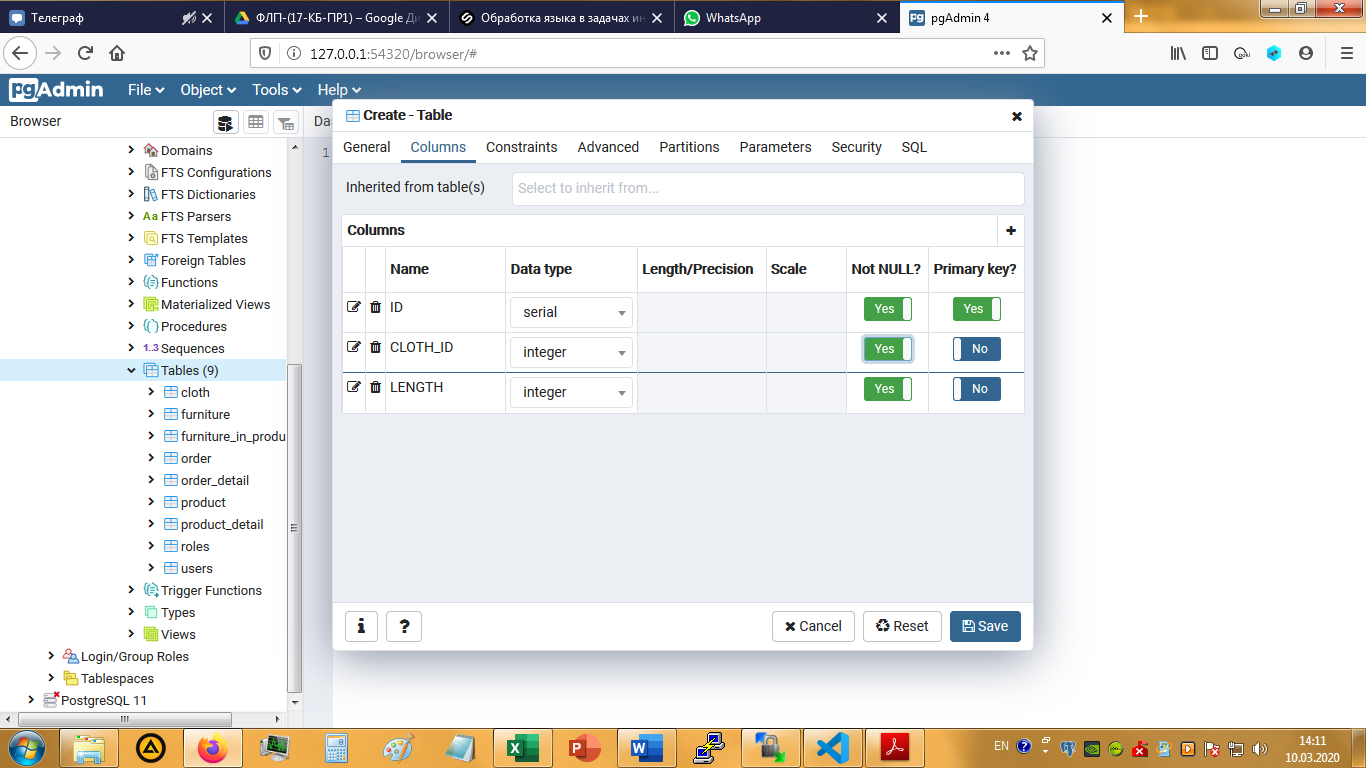


Рисунок 20 – Список полей для таблицы «cloth\_storage»

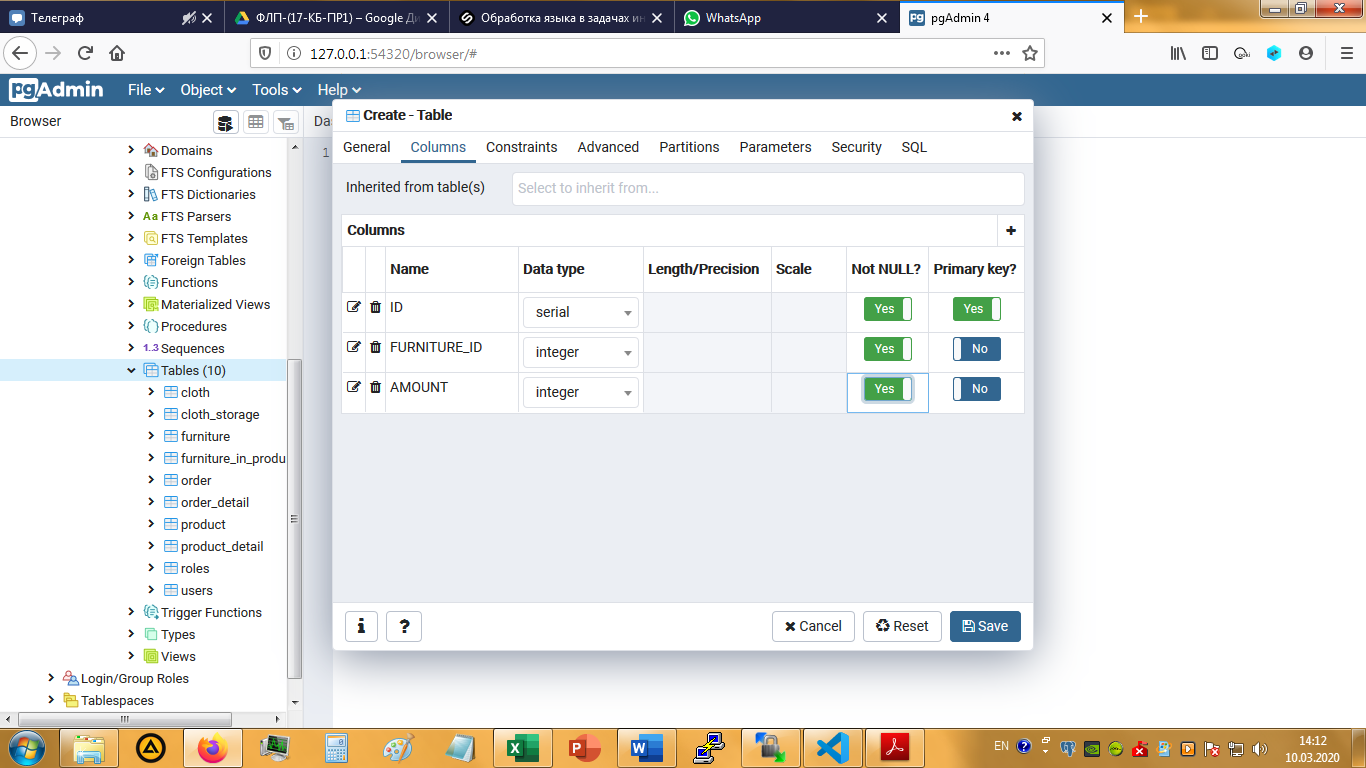


Рисунок 21 – Список полей для таблицы «furniture\_storage»

1. **Исправление ошибок**

Если при создании колонки (поля) произошла ошибка, то сменить тип может быть весьма затруднительно. Однако можно попытаться исправить ситуацию. Если необходимый тип совпадает по размерности с текущим, то его можно изменить на вкладке «Свойства» поля (рисунок 22).

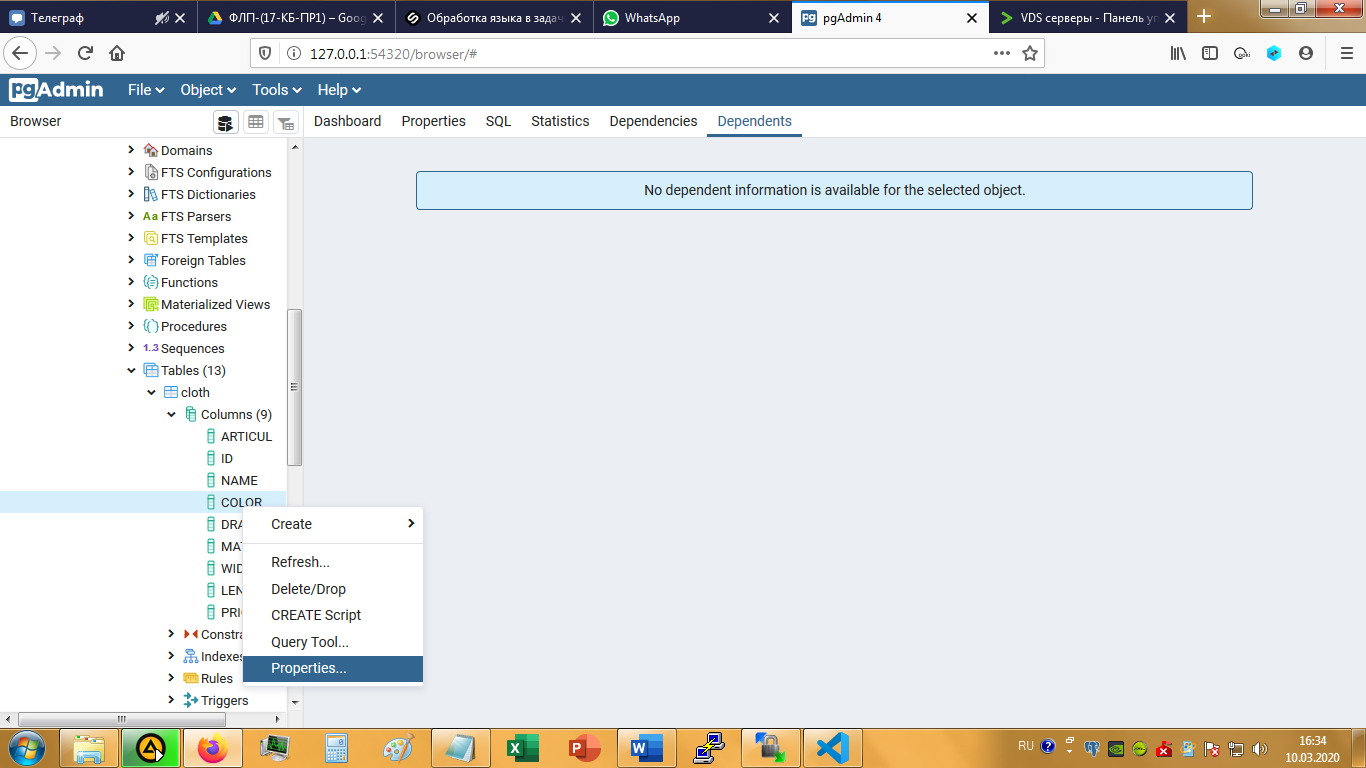


Рисунок 22 – Переход на вкладку свойств колонки

Иначе можно попробовать провести запрос ‘ALTER’. Для этого необходимо выбрать инструмент запросов. Делается это при помощи нажатия ПКМ по одной из таблиц в списке слева (рисунок 23), в открывшемся меню выбираем пункт «Query tool». В открывшемся поле пишем необходимый запрос, нажимаем на стрелочку вверху для выполнения. Пример смены типа integer на integer[] (массив целых 4-байтовых) показан на рисунке 24.

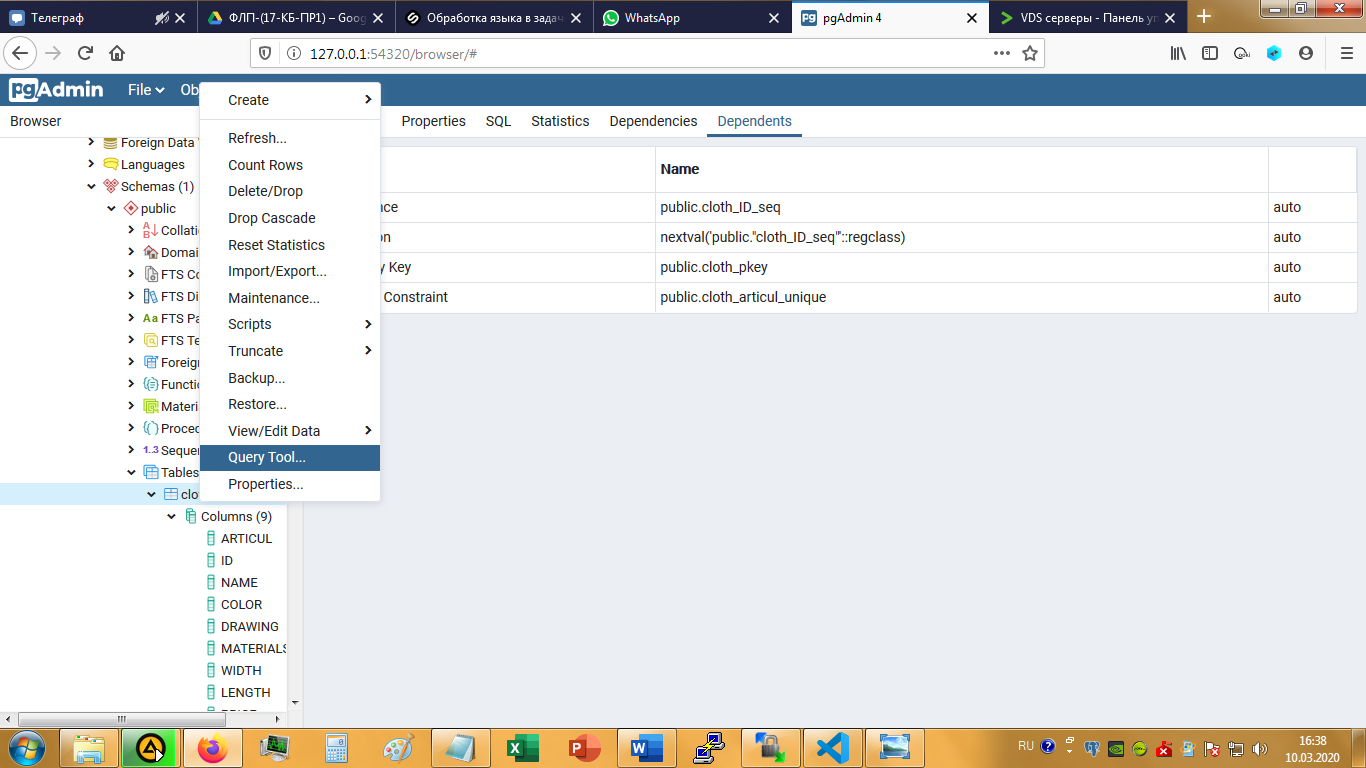


Рисунок 23 – Переход к написанию запросов

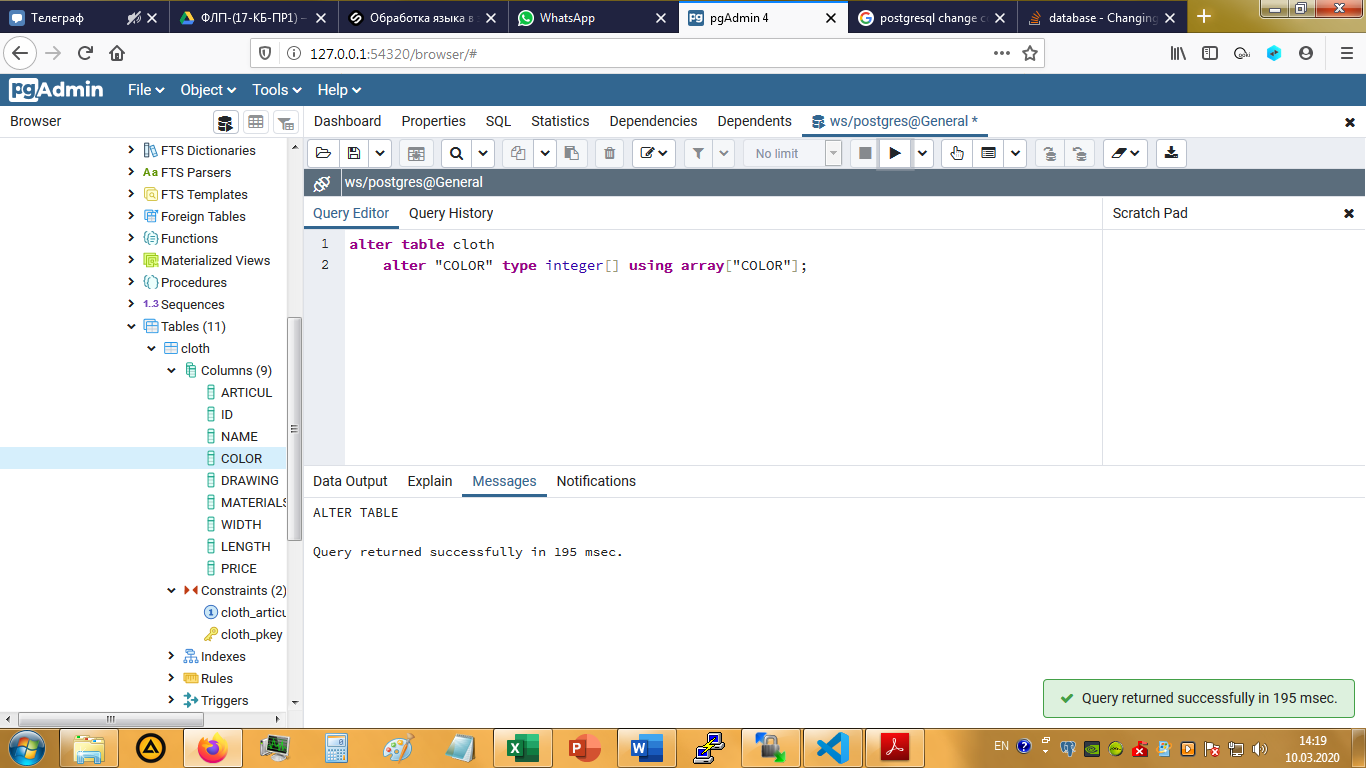


Рисунок 24 – Смена типа поля запросом

При создании таблицы «Пользователи» помимо колонки пароля мы указывали так называемую «соль» (колонка с именем SALT). Это необходимо для безопасного хранения паролей в таблицах. Подход следующий: мы храним не просто пароль, а результат хеш-функции (например md5), примененной к хеш-функции к указанному паролю с добавлением этой «соли», формируемой каждый раз случайным образом. Псевдокодом можно описать так

md5( md5(password) + salt )

Таким образом, мы значительно повышаем безопасность хранимых данных, так как данная хеш-функция необратима, а наличие «соли» и двойное шифрование уберегают данные от взлома простым перебором. Учтите, что пароль должен быть известен пользователю во время регистрации, в дальнейшем его можно будет только сбросить.

1. **Создание связей между таблицами**

После создания таблиц нужно создать связи между ними. Следует обратить ваше внимание на то, что для установления связей межу таблицами в каждой из них необходимо наличие однотипных полей. Например, таблица «Ткани» содержит поле «Цвет» в качестве зависимого от таблицы «Цвета», содержащей поле id цвета, в качестве ключевого. Таким образом, соединение этих двух таблиц по полю id цвета позволяет установить между ними тип связи один ко многим.

Перейдём непосредственно к созданию связей, называемых внешними ключами (foreign keys). Во вкладке с нужной таблицей найдём пункт «Constraints», нажмём ПКМ и выберем пункт «Создать −> Внешний ключ». Дадим ему имя, как правило для этого отношения имя формируется так: fk\_<имя\_таблицы\_зависимой>\_<имя\_таблицы\_главной>\_<имя\_связываемого\_поля>. Для примера создадим отношение между цветами по полю ID цвета – выберем описанный выше пункт в таблице тканей. Дадим связи имя fk\_cloth\_colors\_id. Во вкладке «Колонки» укажем поле из текущей таблицы, таблицу, с которой производим связь и поле в той таблице. Когда всё указано, нажмём иконку «Плюс» вверху окна, как показано на рисунке 25.

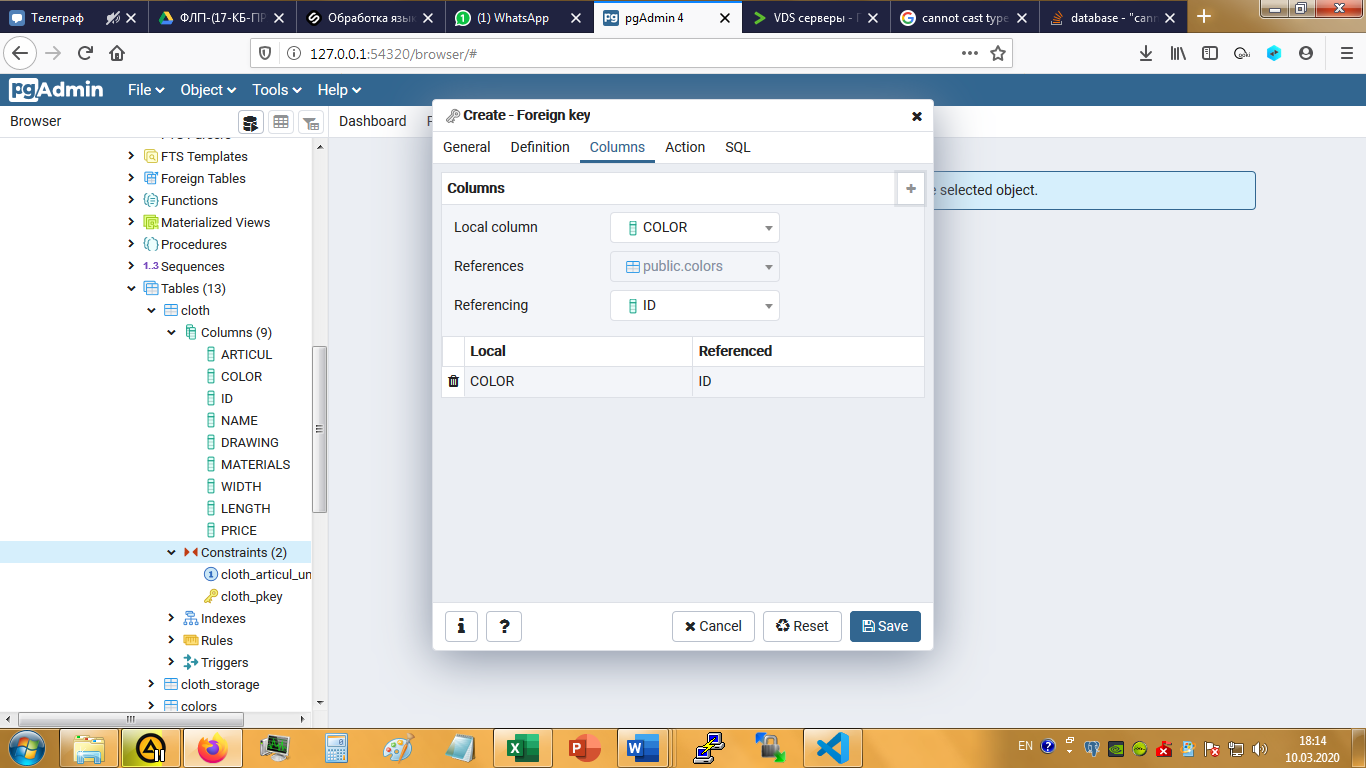


Рисунок 25 – Создание внешнего ключа

После этого сохраним связь. Аналогичным образом устанавливаем связи между остальными таблицами согласно ER-диаграмме, изображенной на рисунке 26.

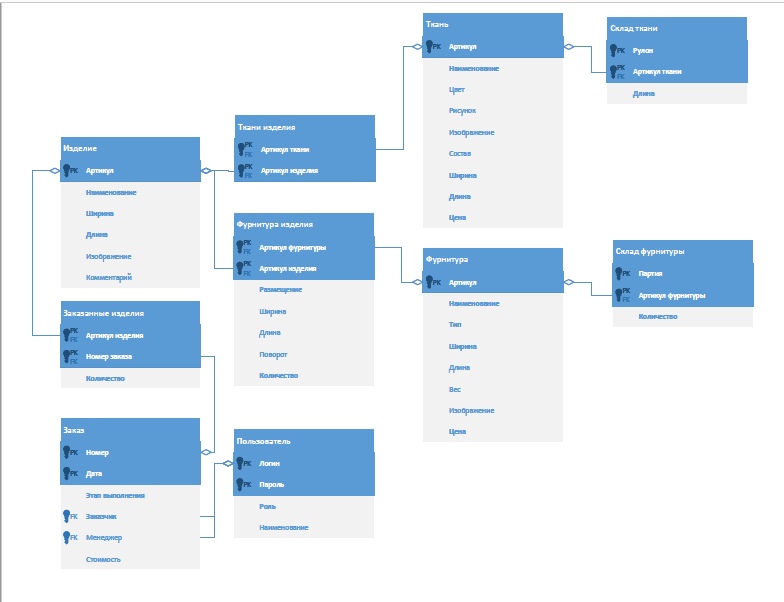


Рисунок 26 – ER-Диаграмма необходимой таблицы

В таблице №1 приведены основные типы данных, необходимые при проектировании полей таблиц базы данных.

Таблица №1 - Встроенные типы данных PostgreSQL 11

| **№** | **Тип** | **Псевдонимы** | **Описание** |
| --- | --- | --- | --- |
| 1 | bigint | int8 | знаковое целое из 8 байт |
| 2 | bigserial | serial8 | восьмибайтное целое с автоувеличением |
| 3 | bit [ (n) ] | varbit [ (*n*) ] | битовая строка фиксированной длины |
| 4 | bit varying [ (n) ] | bool | битовая строка переменной длины |
| 5 | boolean |  | логическое значение (true/false) |
| 6 | bytea |  | двоичные данные («массив байт») |
| 7 | character [ (n) ] | char [ (n) ] | символьная строка фиксированной длины |
| 8 | character varying [ (n) ] | varchar [ (n) ] | символьная строка переменной длины |
| 9 | date |  | календарная дата (год, месяц, день) |
| 10 | double precision | float8 | число двойной точности с плавающей точкой (8 байт) |
| 11 | integer | *Int*, *int4* | знаковое четырёхбайтное целое |
| 12 | Interval [ *поля* ] [ (*p*) ] |  | интервал времени |
| 13 | json |  | текстовые данные JSON |
| 14 | jsonb |  | двоичные данные JSON, разобранные |
| 15 | money |  | денежная сумма |
| 16 | numeric [ (*p*, *s*) ] | decimal [ (p, s) ] | вещественное число заданной точности |
| 17 | real | float4 | число одинарной точности с плавающей точкой (4 байта) |
| 18 | smallint | int2 | знаковое двухбайтное целое |
| 19 | smallserial | serial2 | двухбайтное целое с автоувеличением |
| 20 | serial | serial4 | четырёхбайтное целое с автоувеличением |
| 21 | text |  | символьная строка переменной длины |
| 22 | *timestamp [ (p) ] [ without time zone ]* |  | дата и время (без часового пояса) |
| 23 | *timestamp [ (p) ] with time zone* | timestamptz | дата и время с учётом часового пояса |
| 24 | uuid |  | универсальный уникальный идентификатор |
| 25 | xml |  | XML-данные |

Следует отметить, что одна из задач проектирования базы состоит в обеспечения способа идентификации различных объектов, другими словами система должна отличать друг от друга отдельные строки таблицы. Первичный ключ это столбец, однозначно определяющий строку в таблице. В случае, когда в таблице существует несколько одинаковых значений, уникальный первичный ключ поможет их различить.

Для связи данных между таблицами необходимо создать отношения. Отношения бывают:

* **Один к одному** - определение связи «один к одному» полностью соответствует ее названию. Связью «один к одному» называется такая связь, из наличия которой следует, что если имеется какая-то одна строка в одной таблице, то должна быть точно одна соответствующая ей строка в другой таблице.
* **Один ко многим** - Связью «один ко многим» называется такая связь, из наличия которой следует, что если имеется какая-то одна строка в одной таблице, то должны быть несколько соответствующих ей строк в другой таблице.
* **Многие ко многим** - Связь «многие ко многим» характеризуется тем, что на обеих сторонах связи может присутствовать несколько согласующихся строк, а не только одна.

Связь «Один ко многим» является типичной связью между таблицами, где одна таблица по вторичному ключу ссылается на вторую таблицу с индексным ключом.

Связь «Многие ко многим» характеризуется тем, что на обеих сторонах связи может присутствовать несколько согласующих строк.

1. **Контрольные вопросы**
2. Что такое ядро базы данных?
3. Что такое система управления баз данных?
4. Перечислите основные достоинства языка SQL?
5. Дайте определение логическому имени.
6. Приведите пример команды создания базы данных.
7. Какие системные и числовые типы данных PostgreSQL вам знакомы?
8. Приведите пример команды создания таблицы базы данных.
9. Дайте определения первичному и внешнему ключам.

**Тема №2 - Модификация данных в PostgreSQL.**

ВSQL для выполнения операций ввода данных в таблицу, ихизменения и удаления предназначены три команды языка манипулирования данными: INSERT (вставить), UPDATE (обновить), DELETE (удалить).

Для исполнения запросов, откройте «Query Tool» в pgAdmin.

1. **Ввод данных**

**INSERT** — оператор языка SQL, который позволяет добавить строки в таблицу, заполняя их значениями. Значения можно вставлять перечислением с помощью слова values и перечислив их в круглых скобках через запятую или оператором select, о котором будет сказано позже.

Команда INSERT осуществляет вставкув таблицу новой строки. Впростейшем случае она имеет вид:

INSERT

INTO

<имя таблицы>(<колонка1>, < колонка2> ...)

VALUES

(<значение>, <значение> ...),

...

(<значение>, <значение> ...);

При такой записи указанные в скобках после ключевого слова VALUES значения вводятся в поля соответствующие указанным после наименования таблицы. Можно не указывать имена полей, если заполнить данными нужно все - тогда добавление значений в новой строке будет в том порядке, вкотором соответствующие столбцы указаны при создании таблицы.

Например, ввод новой строки в таблицу «Фурнитура» может быть осуществлен следующим образом:

INSERT

INTO

furniture("ARTICUL", "NAME", "WIDTH", "LENGTH", "WEIGHT", "PRICE")

VALUES

('162164', 'БУСЫ ЖЕМЧУГ 5ММ', 50, 4200, 170, 839.00),

('GB1114', 'ПРЯЖКА МАГНИТНЫЙ ЗАМОК «MICRON»', 320, 170, 60, 143.00)

;

Обратите внимание, имена колонок заключаются в двойные кавычки, значения строковых переменных – в одинарные. Всегда проверяйте шрифт, в котором находятся кавычки – некоторые гарнитуры при копировании распознают кавычки как символы с другими кодами. Если вам нужно ввести пустое значение (NULL), вы вводите его точно так же, как и обычное значение.

Столбцам, наименования которых не указаны в приведенном в скобках списке, автоматически присваивается значение по умолчанию, если оно назначено при описании таблицы, либо значение NULL.

1. **Удаление данных**

Удалениестрок из таблицы осуществляется с помощью команды **DELETE FROM <имя таблицы> <условие>**. Условие определяется ключевым словом WHERE, например:

DELETE FROM

furniture

WHERE

"ARTICUL" = 'GB1114'

;

Данный пример удалит строку в таблице «Фурнитура», где «Артикул» будет равен «GB1114». Если не указывать условие, то произойдет полное очищение таблицы.

1. **Модификация данных**

**UPDATE** — оператор языка SQL, позволяющий обновить значения в заданных столбцах таблицы.

UPDATE <объект> SET <присваивание1 [, присваивание2, ...]> [WHERE <условие>] [OPTION <х1 [, х2, ...]>] <объект> — объект, над которым выполняется действие (таблица или представление(views – подробнее в следующих темах)) <присваивание> — присваивание, которое будет выполняться при каждом выполнении условия <условие>, или для каждой записи, если отсутствует раздел where <условие> — условие выполнения команды <х> — инструкция программе как исполнить запрос.

Например, в таблице цветов есть информация о красном, синем, зеленом и голубом. Но это было недостаточно конкретным – давайте поменяем имя «синий» на «Скай блю»:

UPDATE

colors

SET

"NAME" = 'Скай блю'

WHERE

"NAME" = 'синий'

;

Теперь произведем обновление по нескольким условиям – пользователь «Михаил Терентьев» попросил сменить логин, потому как свой он забыл. Запрос будет выглядеть следующим образом:

UPDATE

users

SET

"LOGIN" = 'chtotam'

WHERE

"NAME" = 'Михаил'

AND

"LAST\_NAME" = 'Терентьев'

;

1. **Задания для самостоятельной работы**

Заполните остальные таблицы небольшим количеством данных.

1. **Контрольные вопросы**
2. Какие команды для модификации данных существуют?
3. Попробуйте задать значение NULL полю, которое не поддерживает таковое. Что произойдёт?
4. Как очистить таблицу от данных?

**Тема №3 - Выборка данных (оператор SELECT).**

Оператор SELECT (выбрать) языка SQL является самым важным и самым часто используемым оператором. Он предназначен для выборкиинформации из таблиц базы данных. Синтаксис оператора SELECT выглядит следующим образом:

SELECT [DISTINCT] <список атрибутов>

FROM <список таблиц>

[WHERE <условие выборки>]

[ORDER BY <список атрибутов>]

[GROUP BY <список атрибутов>]

[HAVING <условие>]

[UNION <выражение с оператором SELECT>];

Вквадратных скобках указаны элементы, которые могут отсутствовать в запросе. Ключевое слово SELECT сообщает базе данных, что данное предложение является запросом на ***извлечение*** информации. После слова SELECT через запятую перечисляются ***наименования полей*** (список атрибутов), содержимое которых запрашивается. Обязательным ключевым словом в предложении-запросе SELECT является слово FROM. За ключевым словом FROM указывается список разделенных запятыми имен таблиц, из которых извлекается информация.

**Например:**

SELECT

"ARTICUL", "NAME", "PRICE"

FROM

product;

Приведенный запрос осуществляет выборку всех значений «ARTICUL», «NAME», «PRICE» из таблицы product.

Если необходимо вывести значения всех,столбцов таблицы, то можно вместо перечисления их имен использовать символ **«\*»** (звездочка).

**Например:**

SELECT \*

FROM

product;

В данном случае результатом выполнения запроса будет вся таблица product. Следует заметить, что результат данного запроса может содержать строки с одинаковыми значениями атрибутов. Для исключения из результата SELECT - запроса повторяющихся записей используется ключевое слово DISTINCT (отличный). Если запрос SELECT извлекает множество полей, то DISTINCT исключаетдубликаты строк, в которых значения всехвыбранных полей идентичны.

**Например:**

SELECT DISTINCT "NAME"

FROM product;

Ключевое слово ALL (все), в отличие от DISTINCT, оказывает противоположное действие, то есть при его использовании повторяющиеся строки включаютсяв выходные данные. Режим, задаваемый ключевым словом ALL, действует по умолчанию, поэтому в реальных запросах для этих целей оно практически не используется.

**Например:**

Напишите данный запрос выбирает все данные из таблицы PRODUCT, расположив столбцы таблицы в следующем порядке: ARTICUL, NAME, IMAGE, PRICE.

SELECT ALL

"ARTICUL", "NAME", "IMAGE", "PRICE"

FROM

product

;

Использование в операторе SELECT предложения, определяемого ключевым словом WHERE (где), позволяет задавать какие строкиуказанных таблиц должны быть выбраны*.* В таблицу, являющуюся результатом запроса, включаются только те строки, для которых условие, указанное в предложении WHERE, принимает значение истина.

**Например**: запрос выводит запрашиваемые поля определенного товара, артикул которого равен «12743303» из таблицы product базы данных рассмотренной в теме №1.

SELECT ALL

"ARTICUL", "NAME", "IMAGE", "PRICE"

FROM

product

WHERE

"ARTICUL" = '12743274'

;

Результатом запроса будет следующая информация:

Таблица №2 – вывод запроса

|  |  |  |  |
| --- | --- | --- | --- |
| ARTICUL | NAME | IMAGE | PRICE |
| 12743274 | Наволочка | [null] | $490.00 |

В задаваемых в предложении WHERE условиях могут использоваться операции сравнения, определяемые операторами = (равно), > (больше), < (меньше), >= (больше или равно), *<—* (меньше или равно), <> (не равно), а также логические операторы AND, OR И NOT.

**Например:**

SELECT ALL

"ARTICUL", "NAME", "IMAGE", "PRICE"

FROM

product

WHERE

"NAME" = 'Наволочка'

;

В результате мы получим все товары с наименованием “Наволочка”

Также рассмотрим пример с использованием логических операторов

**Например:**

SELECT ALL

"ARTICUL", "NAME", "IMAGE", "PRICE"

FROM

product

WHERE

"NAME" = 'Наволочка' AND "PRICE" < '500$'

;

В данном случае выведутся все товары с наименованием “Наволочка” и стоимостью меньше 500$.

При задании логического условия в предложении WHERE могут быть использованы операторы IN, BETWEEN, LIKE, is NULL. Операторы IN (равен любому из списка) и NOT IN (не равен ни одному из списка) используются для сравнения проверяемого значения поля с заданным списком. Этот список значений указывается в скобках справа от оператора IN. Построенное с использованием IN условие считается истинным, если значение поля, имя которого указано слева от IN, в точности совпадает с одним из значений, перечисленных в списке, указанном в скобках справа от IN. Условие, построенное с использованием NOT IN, считается истинным, если значение поля, имя которого указано слева от NOT IN, не совпадаетни с одним из значений, перечисленных в списке, указанном в скобках справа от NOT IN.

**Например:**

Получим из таблицы товары с стоимостью 130$ и 140$

SELECT

"ARTICUL", "NAME", "PRICE"

FROM

product

WHERE

"PRICE" IN ('130$','140$')

;

Оператор BETWEEN используется для проверки условия вхождения значения поля в заданный интервал, то есть вместо списка значений атрибута этот оператор задает границы его изменения.

**Например**:

Получим товары стоимость который находится в пределе от 100$ до 150$

SELECT

"ARTICUL", "NAME", "PRICE"

FROM

product

WHERE

"PRICE" BETWEEN '100$' AND '150$'

;

Оператор BETWEEN может использоваться как для числовых, так и для символьных типов полей.

Оператор LIKE применим только к символьным полям типа CHAR или VARCHAR. Этот оператор просматривает строковые значения полей с целью определения, входит ли заданная в операторе LIKE подстрока в символьную строку – значение проверяемого поля. Для выборки строковых значений по заданному образцу подстроки можно применять шаблон искомого образца строки, использующий следующие символы:

* символ подчеркивания «\_», указанный в шаблоне, определяет возможность наличия в указанном месте одного любого символа;
* символ «%» допускает присутствие в указанном месте проверяемой строки последовательности любых символов произвольной длины.

**Например:**

Осуществим выборку информации из таблицы product всех товаров, фамилии которых начинаются на букву «П»

SELECT \*

FROM

product

WHERE

"NAME" LIKE 'П%'

;

Следует отметить, что операторы сравнения «=, <, >, <=, >=, <>» и операторы IN, BETWEEN и LIKE нельзя использовать для проверки содержимого поля на наличие в нем пустого значения NULL. Для этих целей предназначены специальные операторы is NULL (является пустым) и IS NOT NULL (является не пустым).

**Например:**

Следующий запрос предназначен для выбора из таблицы product записей, в которых отсутствуют значения цен.

SELECT \*

FROM

product

WHERE

"PRICE" IS NULL

;

**Например:**

Следующий запрос предназначен для вывода из таблицы product записей, имеющих в поле «PRICE» значения оценок.

SELECT \*

FROM

product

WHERE

"PRICE" IS NOT NULL;

## Использование агрегирующих функций.

Агрегирующие функции позволяют получать из таблицы сводную информацию, выполняя операции над группой строк таблицы. Для задания в SELECT – запросе агрегирующих операций используются следующие ключевые слова:

COUNT определяет количество строк или значений поля, выбранных посредством запроса и не являющихся NULL-значениями;

SUM вычисляет арифметическую сумму всех выбранных значений данного поля;

AVG вычисляет среднее значение для всех выбранных значений данного поля;

МАХ вычисляет наибольшее из всех выбранных значений данного поля;

MIN вычисляет наименьшее из всех выбранных значений данного поля.

Аргументами агрегатных функций могут быть как столбцы таблицы, так и результаты выражений над ними. При этом выражение может быть сколько угодно сложным.

**Функция AVG ().**

Данные, содержащиеся в таблице должны иметь числовой тип, потому что функция AVG вначале суммирует все значения в столбце, а затем делит полученную сумму на количество этих значений.

**Функция SUM().**

Для вычисления суммы значений, содержащихся в столбце, используется функция SUM. При этом столбец должен иметь числовой тип данных, потому что результат, возвращаемый этой функцией, имеет тот же тип данных, что и столбец.

**Функции MAX и MIN.**

Для нахождения максимального и минимального значения в столбце, используются агрегатные функции MAX и MIN. При этом столбец может содержать числовые и строковые значения, либо значения даты/времени.

**Функция** **COUNT** возвращает количество записей в запросе.  
Синтаксис функции **COUNT**:

SELECT COUNT(expression) FROM table WHERE predicates;

**Функции MAXVALUE и MINVALUE** в чем-то похожи на агрегатные функции MAX и MIN, однако выбирают максимальное и минимальное значение не из множества значений строк в одном столбце, а из значений, заданных в списке аргументов.

**Функция CASE**

Выражение CASE позволяет возвращать разные значения в зависимости от условия.

**Функция COALESCE**

Возвращает первое выражение из списка аргументов, не равное NULL.

**Функция NULLIF**

Данная функция позволяет при выборке делать замену заданного значения на NULL.

Приведем примеры использования агрегирующих функций.

SELECT COUNT (\*)

FROM product;

Данный запрос выведет количество элементов в данной таблице.

SELECT COUNT (DISTINCT "NAME")

FROM product;

В данном случае происходит подсчет количества уникальных имен в таблице.

SELECT MIN("PRICE")

FROM product

Данный запрос находит наименьшую стоимость товара.

SELECT AVG(CAST ("PRICE" AS NUMERIC))

FROM product

Данный запрос находит среднюю стоимость товара, перед этим приводя данные к числовому типу, т.к. функция AVG в psql не определена для денежных типов.

Рассмотрим ряд примеров с использованием пустых значений (NULL) в агрегирующих функциях и упорядочивания выходных полей с использованием конструкции ORDER BY.

Предположим что в магазине проходит акция, и стоимость всех товаров по акции снижена на 20%. Мы хотим вывести данные о товарах, упорядочив их по тем или иным параметрам. Выходные данные упорядочить: а)по стоимости товара; б) в алфавитном порядке названий товаров.

а)

SELECT "NAME", "PRICE", "PRICE" \* 0.8

FROM product

ORDER BY "NAME";

б)

SELECT "NAME", "PRICE", "PRICE" \* 0.8

FROM product

ORDER BY "PRICE";

## Использование числовых функций

Числовые функции возвращают числовые значения на основании значений типа заданного в аргументе. Числовые функции используются для обработки данных, а также в условиях поиска. Основные типы числовых функций, а также основные математические операторы приведены в таблице №3 – Математические операторы

Таблица №3 – Математические операторы

|  |  |  |  |
| --- | --- | --- | --- |
| **Математические операторы** | | | |
| **Оператор** | **Описание** | **Пример** | **Результат** |
| + | сложение | 2 + 3 | 5 |
| - | вычитание | 2 – 3 | -1 |
| \* | умножение | 2 \* 3 | 6 |
| / | деление (при целочисленном делении остаток отбрасывается) | 4 / 2 | 2 |
| % | остаток от деления | 5 % 4 | 1 |
| ^ | возведение в степень (вычисляется слева направо) | 2.0 ^ 3.0 | 8 |
| |/ | квадратный корень | |/ 25.0 | 5 |
| ||/ | кубический корень | ||/ 27.0 | 3 |
| ! | факториал | 5 ! | 120 |
| !! | факториал (префиксная форма) | !! 5 | 120 |
| @ | модуль числа (абсолютное значение) | @ -5.0 | 5 |
| & | битовый AND | 91 & 15 | 11 |
| | | битовый OR | 32 | 3 | 35 |
| # | битовый XOR | 17 # 5 | 20 |
| ~ | битовый NOT | ~1 | -2 |
| << | битовый сдвиг влево | 1 << 4 | 16 |
| >> | битовый сдвиг вправо | 8 >> 2 | 2 |

Таблица №4 – Арифметические функции

|  |  |  |  |
| --- | --- | --- | --- |
| **Математические функции** | | | |
| **Функция** | **Описание** | **Пример** | **Результат** |
| abs(*x*) | модуль числа | abs(-17.4) | 17.4 |
| cbrt(dp) | кубический корень | cbrt(27.0) | 3 |
| ceil(dp или  numeric) | ближайшее целое, большее или равное аргументу | ceil(-42.8) | -42 |
| ceiling(dp или  numeric) | равнозначно ceil | ceiling(-95.3) | -95 |
| degrees(dp) | преобразование радианов в градусы | degrees(0.5) | 28.647889756541 |
| div(*y* numeric,  *x* numeric) | целочисленный результат *y*/*x* | div(9,4) | 2 |
| exp(dp или numeric) | экспонента | exp(1.0) | 2.7182818284590 |
| floor(dp или numeric) | ближайшее целое, меньшее или равное аргументу | floor(-42.8) | -43 |
| ln(dp или numeric) | натуральный логарифм | ln(2.0) | 0.6931471805599 |
| log(dp или numeric) | логарифм по основанию 10 | log(100.0) | 2 |
| log(*b* numeric,  *x* numeric) | логарифм по основанию *b* | log(2.0, 64.0) | 6.0000000000 |
| mod(*y*, *x*) | остаток от деления *y*/*x* | mod(9,4) | 1 |
| pi() | константа «π» | pi() | 3.1415926535897 |
| power(*a* dp, *b* dp) | *a* возводится в степень *b* | power(9.0, 3.0) | 729 |
| power(*a* numeric, *b* numeric) | *a* возводится в степень *b* | power(9.0, 3.0) | 729 |
| radians(dp) | преобразование градусов в радианы | radians(45.0) | 0.7853981633974 |
| round(dp или numeric) | округление до ближайшего целого | round(42.4) | 42 |
| round(*v* numeric, *s* int) | округление *v* до *s* десятичных знаков | round(42.4382, 2) | 42.44 |
| scale(numeric) | масштаб аргумента (число десятичных цифр в дробной части) | scale(8.41) | 2 |
| sign(dp или numeric) | знак аргумента (-1, 0, +1) | sign(-8.4) | -1 |
| sqrt(dp или numeric) | квадратный корень | sqrt(2.0) | 1.4142135623731 |
| trunc(dp или numeric) | округление к нулю | trunc(42.8) | 42 |
| trunc(*v* numeric, *s* int) | округление к 0 до *s* десятичных знаков | trunc(42.4382, 2) | 42.43 |

Таблица №5 – Случайные функции

| **Случайные функции** | | |
| --- | --- | --- |
| **Функция** | **Тип результата** | **Описание** |
| random() | dp | случайное число в диапазоне 0.0 <= x < 1.0 |
| setseed(dp) | void | задаёт отправную точку для последующих вызовов random() (значение между -1.0 и 1.0, включая границы) |

Таблица №6 – Тригонометрические функции

| **Функции (в радианах)** | **Функции (в градусах)** | **Описание** |
| --- | --- | --- |
| acos(*x*) | acosd(*x*) | арккосинус |
| asin(*x*) | asind(*x*) | арксинус |
| atan(*x*) | atand(*x*) | арктангенс |
| atan2(*y*, *x*) | atan2d(*y*, *x*) | арктангенс *y*/*x* |
| cos(*x*) | cosd(*x*) | косинус |
| cot(*x*) | cotd(*x*) | котангенс |
| sin(*x*) | sind(*x*) | синус |
| tan(*x*) | tand(*x*) | тангенс |

**Примечание**

Также можно работать с углами в градусах, применяя вышеупомянутые функции преобразования единиц radians() и degrees(). Однако предпочтительнее использовать тригонометрические функции с градусами, так как это позволяет избежать ошибок округления в особых случаях, например, при вычислении sind(30).

## Строковые функции.

Строковые функции используют в качестве аргумента строку символов, а в качестве результата возвращают символьную строку или числовое значение.

Таблица №7 – Строковые функции (основные)

|  |  |  |  |
| --- | --- | --- | --- |
| **Строковые функции** | | | |
| **Функция** | **Описание** | **Пример** | **Результат** |
| *string* || *string* | Конкатенация строк | 'Post' || 'greSQL' | PostgreSQL |
| *string* || *не string* или *не string* || *string* | Конкатенация строк с одним не строковым операндом | 'Value: ' || 42 | Value: 42 |
| bit\_length(*string*) | Число бит в строке | bit\_length('jose') | 32 |
| char\_length(*string*) или  character\_length(*string*) | Число символов в строке | char\_length('jose') | 4 |
| lower(*string*) | Переводит символы строки в нижний регистр | lower('TOM') | tom |
| octet\_length(*string*) | Число байт в строке | octet\_length  ('jose') | 4 |
| ovelay(*string* placing  *string* from int [for int]) | Заменяет подстроку | ovelay  ('Txxxxas' placing 'hom' from 2 for 4) | Thomas |
| position(*substring*  in *string*) | Положение указанной подстроки | position  ('om' in 'Thomas') | 3 |
| substring(*string* [from int] [for int]) | Извлекает подстроку | substring  ('Thomas' from 2 for 3) | hom |
| substring(*string*  from *шаблон*) | Извлекает подстроку, соответствующую регулярному выражению в стиле POSIX. | substring  ('Thomas' from '...$') | mas |
| substring  (*string* from *шаблон* for *спецсимвол*) | Извлекает подстроку, соответствующую регулярному выражению в стиле SQL. | substring  ('Thomas' from '%#"o\_a#"\_' for '#') | oma |
| trim([leading | trailing | both] [*characters*] from *string*) | Удаляет наибольшую подстроку, содержащую только символы *characters* (по умолчанию пробелы), с начала (leading), с конца (trailing) или с обеих сторон (both, (по умолчанию)) строки *string* | trim(both 'xyz' from 'yxTomxx') | Tom |
| trim([leading | trailing | both] [from] *string* [, *characters*] ) | Нестандартный синтаксис trim() | trim(both from 'yxTomxx', 'xyz') | Tom |
| upper(*string*) | Переводит символы строки в верхний регистр | upper('tom') | TOM |

Таблица №8 – Строковые функции (дополнительные)

|  |  |  |  |
| --- | --- | --- | --- |
| Другие функции работы со строками | | | |
| Функция | Описание | Пример | Результат |
| ascii(*string*) | Возвращает ASCII-код первого символа аргумента. Для UTF8 возвращает код символа в Unicode. Для других многобайтных кодировок аргумент должен быть ASCII-символом. | ascii('x') | 120 |
| btrim(*string* text  [, *characters* text]) | Удаляет наибольшую подстроку, состоящую только из символов *characters* (по умолчанию пробелов), с начала и с конца строки *string* | btrim('xyxtrimyyx', 'xyz') | trim |
| chr(int) | Возвращает символ с данным кодом. Для UTF8 аргумент воспринимается как код символа Unicode, а для других кодировок он должен указывать на ASCII-символ. Код 0 (NULL) не допускается, так как байты с нулевым кодом в текстовых строках сохранить нельзя. | chr(65) | A |
| concat  (*str* "any" [, *str* "any" [, ...] ]) | Соединяет текстовые представления всех аргументов, игнорируя NULL. | concat('abcde', 2, NULL, 22) | abcde222 |
| concat\_ws  (*sep* text, *str* "any"  [, *str* "any" [, ...] ]) | Соединяет все аргументы, кроме первого, через разделитель, игнорируя аргументы NULL. Разделитель указывается в первом аргументе. | concat\_ws(',', 'abcde', 2, NULL, 22) | abcde,2,22 |
| convert(*string* bytea,  *src\_encoding* name,  *dest\_encoding* name) | Преобразует строку string из кодировки src\_encoding в dest\_encoding. Переданная строка должна быть допустимой для исходной кодировки. | convert(  'text\_in\_utf8', 'UTF8', 'LATIN1') | строка text\_in\_utf8, представленная в кодировке Latin-1 (ISO 8859-1) |
| convert\_from  (*string* bytea,  *src\_encoding* name) | Преобразует строку *string* из кодировки *src\_encoding* в кодировку базы данных. Переданная строка должна быть допустимой для исходной кодировки. | convert\_from  ('text\_in\_utf8', 'UTF8') | строка text\_in\_utf8, представленная в кодировке текущей базы данных |
| convert\_to(*string* text,  *dest\_encoding* name) | Преобразует строку в кодировку *dest\_encoding*. | convert\_to  ('некоторый текст', 'UTF8') | некоторый текст, представленный в кодировке UTF8 |
| initcap(*string*) | Переводит первую букву каждого слова в строке в верхний регистр, а остальные — в нижний. Словами считаются последовательности алфавитно-цифровых символов, разделённые любыми другими символами. | initcap('hi THOMAS') | Hi Thomas |
| left(*str* text, *n* int) | Возвращает первые *n* символов в строке. Когда *n* меньше нуля, возвращаются все символы слева, кроме последних |*n*|. | left('abcde', 2) | ab |
| length(*string*) | Число символов в строке *string* | length('jose') | 4 |
| lpad(*string* text,  *length* int [, *fill* text]) | Дополняет строку *string* слева до длины *length* символами *fill* (по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается справа. | lpad('hi', 5, 'xy') | xyxhi |
| ltrim(*string* text [,  *characters* text]) | Удаляет наибольшую подстроку, содержащую только символы *characters* (по умолчанию пробелы), с начала строки *string* | ltrim('zzzytest', 'xyz') | test |
| md5(*string*) | Вычисляет MD5-хеш строки *string* и возвращает результат в 16-ричном виде | md5('abc') | 900150983cd24fb0d6963f7d28e17f72 |
| pg\_client\_encoding() | Возвращает имя текущей клиентской кодировки | pg\_client\_encoding() | SQL\_ASCII |
| quote\_literal  (*value* anyelement) | Переводит данное значение в текстовый вид и заключает в апострофы как текстовую строку. Символы апостроф и обратная косая черта при этом дублируются. | quote\_literal  (42.5) | '42.5' |
| repeat(*string* text,  *number* int) | Повторяет содержимое *string* указанное число (*number*) раз | repeat('Pg', 4) | PgPgPgPg |
| replace(*string* text,  *from* text, *to* text) | Заменяет все вхождения в *string* подстроки *from* подстрокой *to* | replace('abcdefabcdef',  'cd', 'XX') | abXXefabXXef |
| reverse(*str*) | Возвращает перевёрнутую строку | reverse('abcde') | edcba |
| right(*str* text, *n* int) | Возвращает последние *n* символов в строке. Когда *n* меньше нуля, возвращаются все символы справа, кроме первых |*n*|. | right('abcde', 2) | de |
| rpad(*string* text,  *length* int [, *fill* text]) | Дополняет строку *string* справа до длины *length* символами *fill* (по умолчанию пробелами). Если длина строки уже больше заданной, она обрезается. | rpad('hi', 5, 'xy') | hixyx |
| rtrim(*string* text [,  *characters* text]) | Удаляет наибольшую подстроку, содержащую только символы *characters* (по умолчанию пробелы), с конца строки *string* | rtrim  ('testxxzx', 'xyz') | test |
| split\_part(*string* text,  *delimiter* text,  *field* int) | Разделяет строку *string* по символу *delimiter* и возвращает элемент по заданному номеру (считая с 1) | split\_part  ('abc~@~def~@~ghi', '~@~', 2) | def |
| strpos(*string*,  *substring*) | Возвращает положение указанной подстроки (подобно position(*substring* in *string*), но с другим порядком аргументов) | strpos('high', 'ig') | 2 |
| substr(*string*, *from* [,  *count*]) | Извлекает подстроку (подобно substring(*string* from *from* for *count*)) | substr('alphabet', 3, 2) | ph |

## Предложение GROUP BY

GROUP BY собирает в одну строку все выбранные строки, выдающие одинаковые значения для выражений группировки. В качестве выражения внутри элемента\_группирования может выступать имя входного столбца, либо имя или порядковый номер выходного столбца (из списка элементов SELECT), либо произвольное значение, вычисляемое по значениям входных столбцов. В случае неоднозначности имя в GROUP BY будет восприниматься как имя входного, а не выходного столбца.

Элементы, указанные в предложении GROUP BY, называются элементами группировки*,* и именно они определяют, по какому признаку строки делятся нагруппы. При этом группой называется набор строк, имеющих одинаковоезначение в элементе (элементах) группировки.

Синтаксис предложения GROUP BY имеет следующий вид:

GROUP BY <элемент\_группирования> [, ...]

Если в элементе группирования задаётся GROUPING SETS, ROLLUP или CUBE, предложение GROUP BY в целом определяет некоторое число независимых наборов группирования. Это даёт тот же эффект, что и объединение подзапросов (с UNION ALL) с отдельными наборами группирования в их предложениях GROUP BY.

Агрегатные функции, если они используются, вычисляются по всем строкам, составляющим каждую группу, и в итоге выдают отдельное значение для каждой группы. (Если агрегатные функции используются без предложения GROUP BY, запрос выполняется как с одной группой, включающей все выбранные строки.) Набор строк, поступающих в каждую агрегатную функцию, можно дополнительно отфильтровать, добавив предложение FILTER к вызову агрегатной функции. С предложением FILTER на вход агрегатной функции поступают только те строки, которые соответствуют заданному фильтру.

Когда в запросе присутствует предложение GROUP BY или какая-либо агрегатная функция, выражения в списке SELECT не могут обращаться к негруппируемым столбцам, кроме как в агрегатных функциях или в случае функциональной зависимости, так как иначе в негруппируемом столбце нужно было бы вернуть более одного возможного значения. Функциональная зависимость образуется, если группируемые столбцы (или их подмножество) составляют первичный ключ таблицы, содержащей негруппируемый столбец.

Агрегатная функция берет столбец значений и возвращает одно значение. Предложение GROUP BY указывает, что результаты запроса следует разделить на группы, применить агрегатную функцию по отдельности к каждой группе и получить для каждой группы одну строку результатов. Имейте в виду, что все агрегатные функции вычисляются перед «скалярными» выражениями в предложении HAVING или списке SELECT. Это значит, что, например, с помощью выражения CASE нельзя обойти вычисление агрегатной функции. Например, если необходимо вычислить среднее значение цены для каждой заданной длины из таблицы product, то можно воспользоваться запросом, приведенном на рисунке 27.

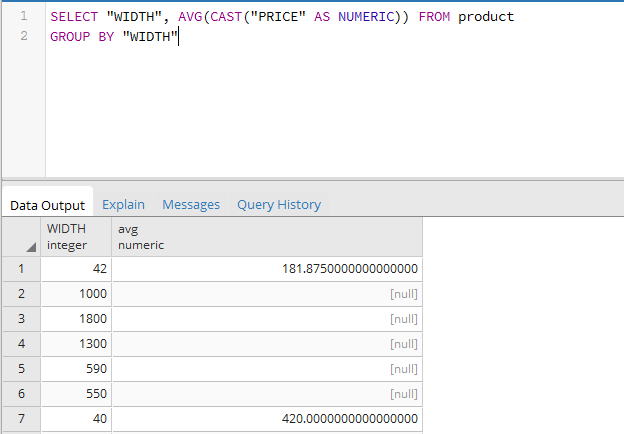


Рисунок 27 – Запрос и результаты его выполнения

Возможна группировка результатов запроса на основании порядкового номера возвращаемого элемента в предложении SELECT. Таким образом, запрос **SELECT “WIDTH”, AVG(CAST(“PRICE” AS NUMERIC)) FROM product GROUP BY 1** выдаст такой же результат, как и предыдущий запрос.

В SQL можно группировать результаты запроса на основании двух или более элементов. Например, необходимо для каждого товара (по артикулу товара) необходимо вывести минимальную цену за единицу продукции. Для этого нужно сгруппировать таблицу Товар по артикулу товара и ширине товара, применяя к значениям цен агрегатную функцию MIN (рисунок 28).

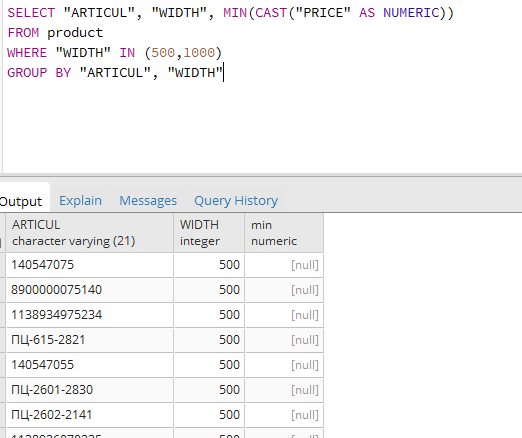


Рисунок 28 – Запрос с функцией MIN и результаты его выполнения

Как следует из предыдущих примеров, на запросы с группировкой накладываются следующие ограничения.

1. В настоящее время указания FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE и FOR KEY SHARE нельзя задать вместе с GROUP BY.

2. Все элементы списка возвращаемых столбцов должны иметь одно значение для каждой группы строк. Это означает, что возвращаемым элементом в предложении SELECT может быть:

* константа;
* агрегатная функция, возвращающая одно значение для всех строк, входящих в группу;
* элемент группировки, который по определению имеет одно и то же значение во всех строках группы;
* функция, которая используется в качестве элемента группировки;
* выражение, включающее в себя перечисленные выше элементы.

## Предложение HAVING

Предложение HAVING запроса SELECT применяется для наложения условий на строки, возвращаемые при использовании предложения GROUP BY. Оно состоит из ключевого слова HAVING, за которым следует <условие\_поиска>:

<условие\_поиска> ::= [NOT] <условие\_поиска1>

[[AND|OR][NOT] <условие\_поиска2>]…

где <условие\_поиска> позволяет исключить из результата группы, не удовлетворяющие заданным условиям. Условие поиска совпадает с условием поиска, рассмотренным выше для предложения WHERE. Однако в качестве значения часто используется значение, возвращаемое агрегатными функциями. Результат совместной работы HAVING с GROUP BY аналогичен результату работы запроса SELECT с предложением WHERE с той разницей, что HAVING выполняет те же функции над строками (группами) возвращаемого набора данных, а не над строками исходной таблицы. Из этого следует, что предложение HAVING начинает свою работу после того, как предложение GROUP BY разделит базовую таблицу на группы. В противоположность этому использование предложения WHERE приводит к тому, что сначала отбираются строки из базовой таблицы и только после этого отобранные строки начинают использоваться.

Рассмотрим следующий пример: в таблице хранится информация о заказах – его идентификатор, стоимость, клиент, менеджер и скидка. Мы при помощи агрегатной функции max() можем выбрать максимальную стоимость заказа по каждому из менеджеров, как показано на рисунке 29.

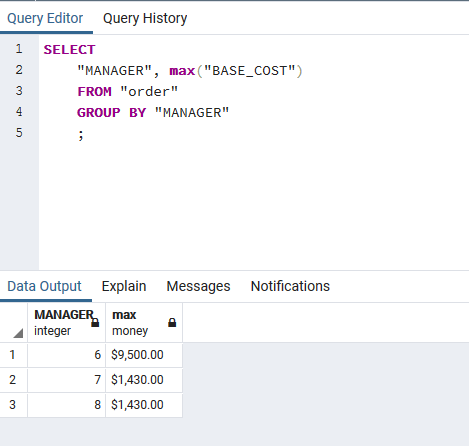


Рисунок 29 – Запрос без HAVING

Но если мы захотим выбрать менеджеров, которые оформляли особо большие заказы, скажем стоимостью более 2000, то нам нужно добавить ещё условие. Мы не можем это сделать внутри WHERE, так как агрегатная функция берет столбец значений и возвращает одно значение, в то время как WHERE использует предикаты при отборе записей в результат.

Поэтому реализовать наш замысел поможет предложение HAVING, как изображено на рисунке 30.

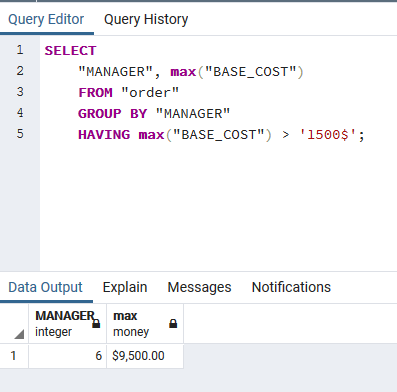


Рисунок 30 - Запрос с HAVING

Следует учесть, что предложение HAVING должно ссылаться только на агрегатные функции и элементы, выбранные GROUP BY.

Например, следующий запрос потерпит неудачу:

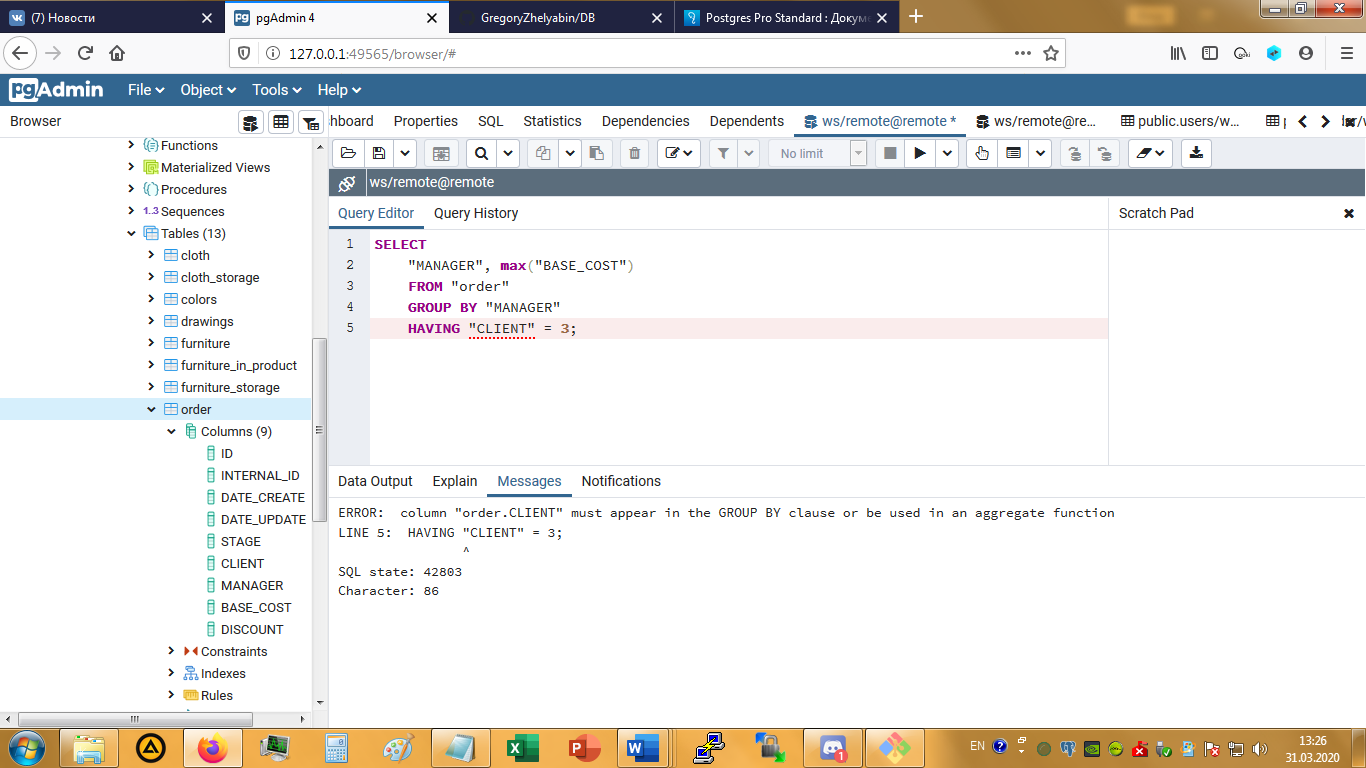


Рисунок 31 – Неправильное использование HAVING

Поле ИД клиента не может быть использовано в предложении HAVING, потому что оно может иметь (и действительно имеет) больше чем одно значение на группу вывода.

## Вложенные подзапросы

В предложении SELECT могут использоваться простые соотнесенные вложенные запросы. При использовании простого подзапроса возвращенный им результат вставляется во все строки, формируемые внешним запросом. В предложении SELECT может использоваться только <скалярный\_подзапрос>, то есть подзапрос, который возвращает только одно значение.

Выберем из таблицы заказов (рисунок 31) идентификатор, стоимость зазака и для наглядности – среднюю стоимость заказа.

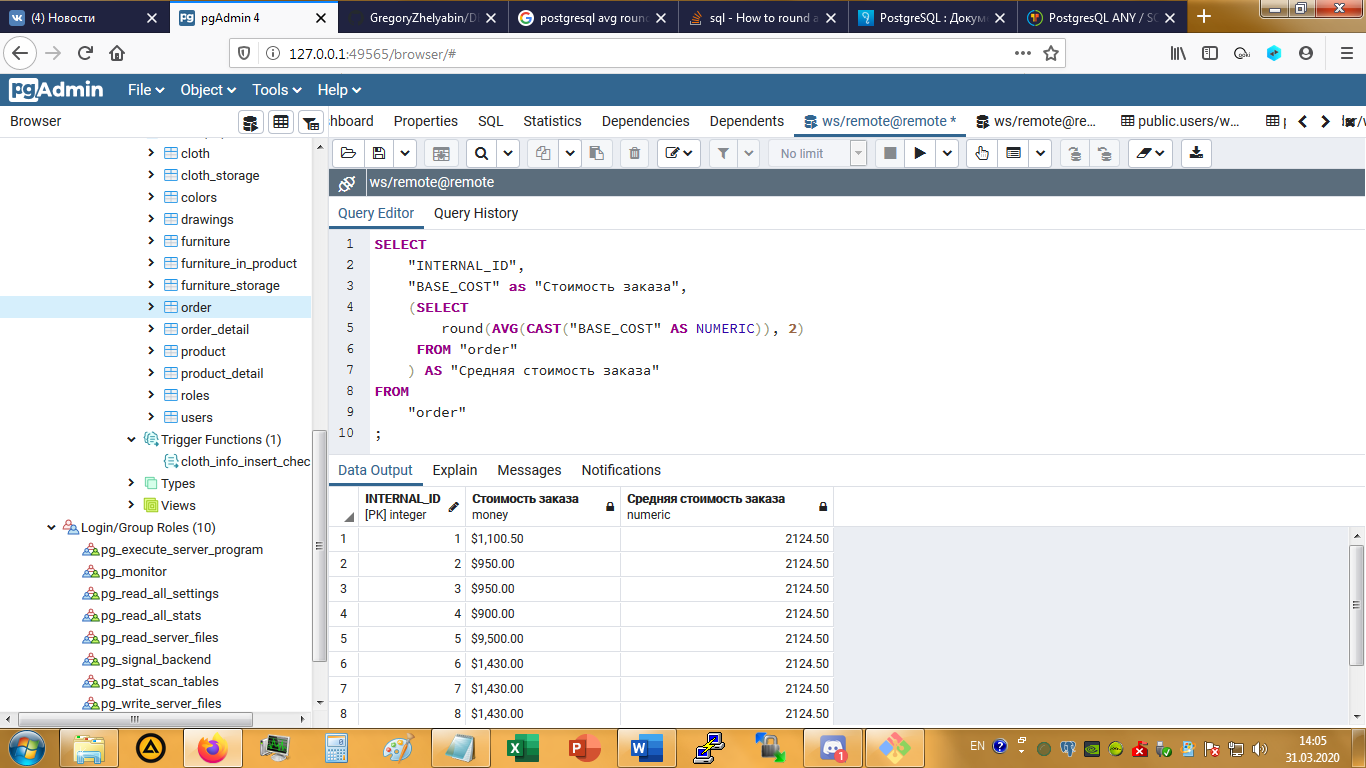


Рисунок 31 – Результат подзапроса в качестве столбца

Как следует из этого примера, связь между значением, возвращаемым простым вложенным запросом (средняя стоимость заказа), и значениями внешнего запроса фактически отсутствует.

В предложении FROM могут быть определены две и более производные таблицы. Например, требуется вывести среднее количество заказов зарегистрированных покупателей. Для этого нужно определить общее количество покупателей (роль = 3), общее число заказов и поделить полученное количество заказов на число клиентов (рисунок 32).

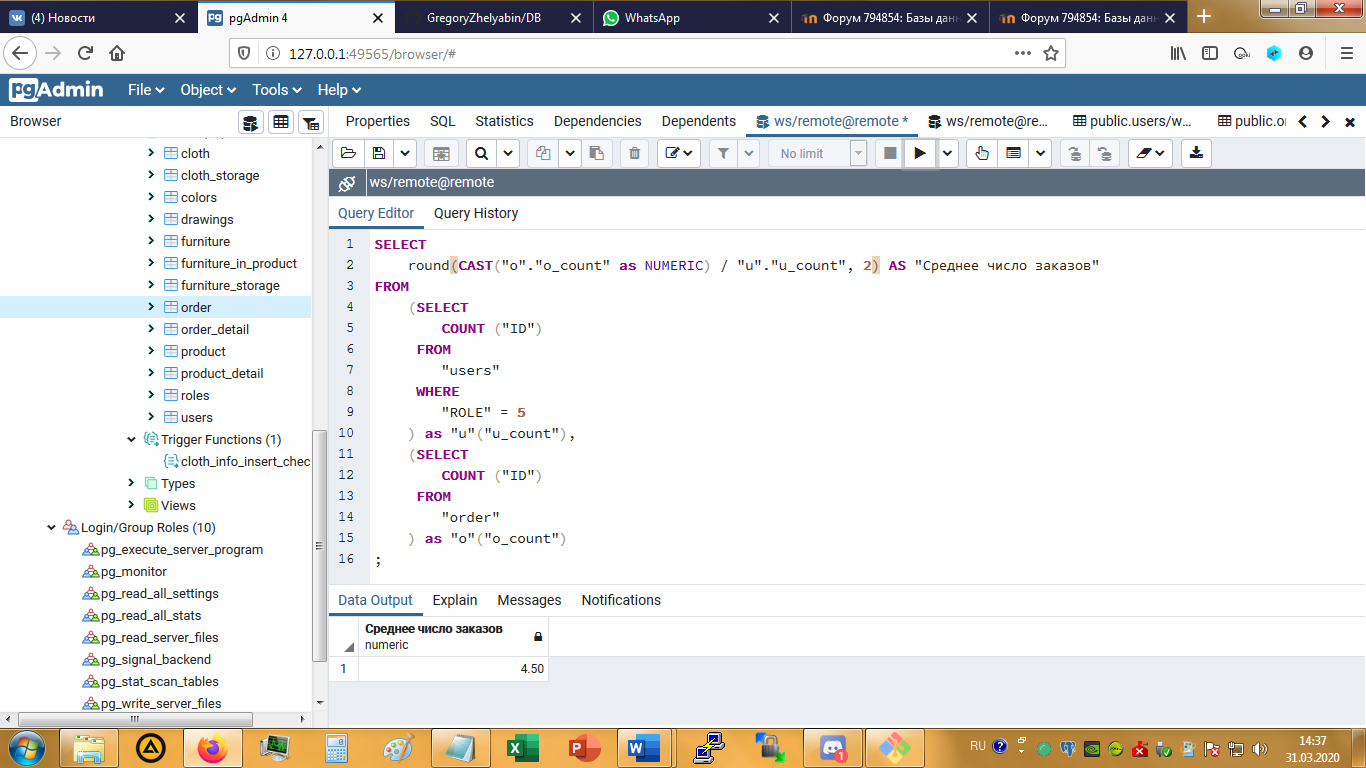


Рисунок 32 – Подсчёт среднего числа заказов пользователей

Во вложенном запросе можно использовать агрегатные функции. Допустим, необходимо вывести клиентов (рисунок 33), у которых средний чек выше, чем у тех, кто оформлял заказы через менеджера Терентьева (ID = 6).

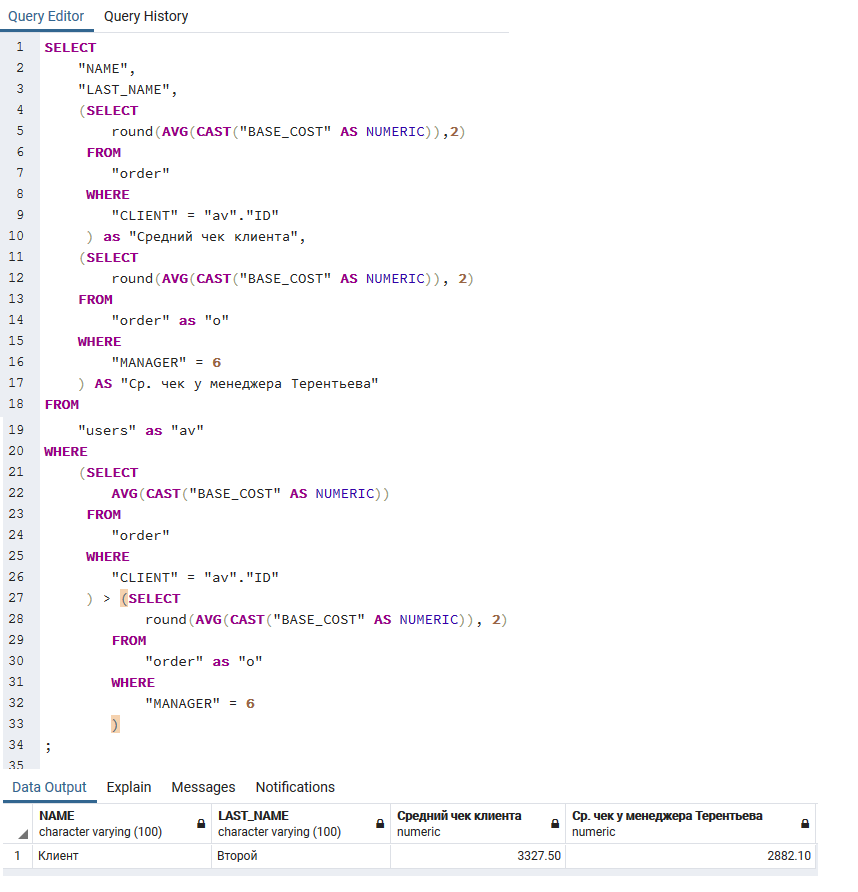


Рисунок 33 – Агрегатная функция в подзапросе

## Функции даты и времени.

Функции выполняют различные действия над входными значениями времени и даты и возвращают строковое, числовое значение или значение в формате даты и времени.

Для получения значений текущей даты и системного времени сервера используются следующие функции:

* Получение текущей системной даты и времени:

**SELECT CURRENT\_TIMESTAMP;**

Получение текущего системного времени:

**SELECT date\_part('hour', CURRENT\_TIMESTAMP);**

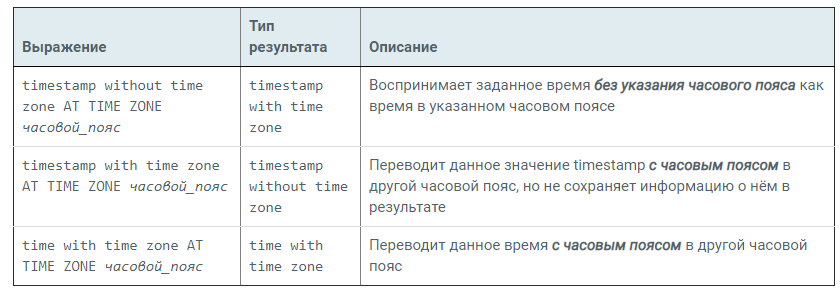
* Функция **EXTRACT** получает из значений даты/времени поля, такие как год или час. Здесь *источник* — значение типа timestamp, time или interval. (Выражения типа date приводятся к типу timestamp, так что допускается и этот тип.) Указанное *поле* представляет собой идентификатор, по которому из источника выбирается заданное поле. Синтаксис имеет следующий вид: EXTRACT(field FROM source).

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');

Функция **extract** в основном предназначена для вычислительных целей. В ней имеются допустимые поля:

* Век: SELECT EXTRACT(**CENTURY** FROM TIMESTAMP '2000-12-16 12:21:13');
* День: SELECT EXTRACT(**DAY** FROM TIMESTAMP '2001-02-16 20:38:40');
* Декада: SELECT EXTRACT(**DECADE** FROM TIMESTAMP '2001-02-16 20:38:40');
* Час: SELECT EXTRACT(**HOUR** FROM TIMESTAMP '2001-02-16 20:38:40')
* Минуты: SELECT EXTRACT(**MINUTE** FROM TIMESTAMP '2001-02-16 20:38:40');
* Месяц: SELECT EXTRACT(**MONTH** FROM TIMESTAMP '2001-02-16 20:38:40');
* Секунды: SELECT EXTRACT(**SECOND** FROM TIMESTAMP '2001-02-16 20:38:40');
* Неделя: SELECT EXTRACT(**WEEK** FROM TIMESTAMP '2001-02-16 20:38:40');
* Год: SELECT EXTRACT(**YEAR** FROM TIMESTAMP '2001-02-16 20:38:40');

Указание AT TIME ZONE позволяет переводить дату/время без часового пояса в дату/время с часовым поясом и обратно, а также пересчитывать значения времени для различных часовых поясов.

Таблица 9 – Использование выражений временных зон 

В этих выражениях желаемый часовой\_пояс можно задать либо в виде текстовой строки (например, 'America/Los\_Angeles'), либо как интервал (например, INTERVAL '-08:00'). Пример:

SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';

Результат: 2001-02-16 19:38:40-08

PostgreSQL предоставляет набор функций, результат которых зависит от текущей даты и времени. Все следующие функции соответствуют стандарту SQL и возвращают значения, отражающие время начала текущей транзакции:

CURRENT\_DATE

CURRENT\_TIME

CURRENT\_TIMESTAMP

CURRENT\_TIME(*точность*)

CURRENT\_TIMESTAMP(*точность*)

LOCALTIME

LOCALTIMESTAMP

LOCALTIME(*точность*)

LOCALTIMESTAMP(*точность*)

Пример:

SELECT CURRENT\_TIMESTAMP(2);

Результат: 2001-12-23 14:39:53.66-05

## Формирование связанных подзапросов

При использовании подзапросов во внутреннем запросе можно ссылаться на таблицу, имя которой указано в предложении FROM внешнего запроса. В этом случае такой связанныйподзапрос выполняется по одному разу для каждойстроки таблицы основного запроса. Предложение GROUP BY позволяет группировать выводимые SELECT – запросом записи по значению некоторого поля. Использование предложения HAVING позволяет при выводе осуществлять фильтрацию групп данных. Предикат предложения HAVING оценивается не для каждой строки результата, а для каждой группы выходных записей, сформированной предложением GROUP BY внешнего запроса.

Подзапрос представляет команду SELECT и заключается в скобки. В данном же случае при добавлении одного товара выполняется три подзапроса. Каждый подзапрос возвращает одно скалярное значение, например, идентификатор товара или покупателя.

Напишем запрос с подзапросом для получения товаров из таблицы cloth, которые имеют минимальную цену:

SELECT \*

FROM

cloth

WHERE

'PRICE' = (SELECT MIN('PRICE') FROM cloth)

;

Или найдем товары, цена которых выше средней:

SELECT \*

FROM

cloth

WHERE

(CAST("PRICE" AS NUMERIC)) > (SELECT AVG(CAST("PRICE" AS NUMERIC)) FROM cloth)

;

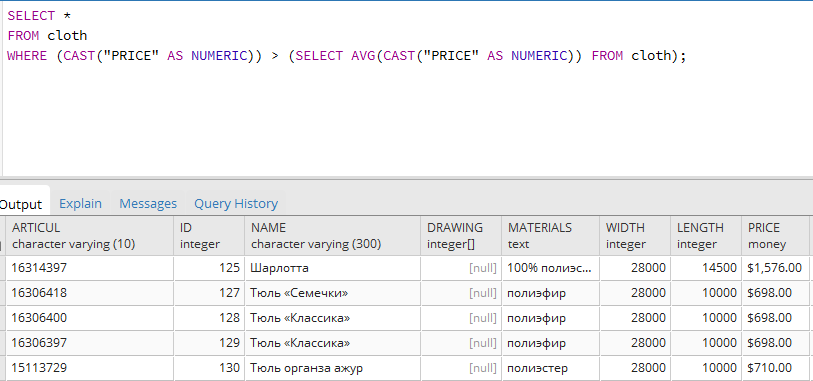


Рисунок 34 – Запрос товаров с ценой выше средней

Подзапросы бывают коррелирующими и некоррелирующими. В примерах выше команды SELECT выполняли фактически один подзапрос для всей команды, например, подзапрос возвращает минимальную или среднюю цену, которая не изменится, сколько бы мы строк не выбирали в основном запросе. Результат такого подзапроса не зависел от строк, которые выбираются в основном запросе. И такой подзапрос выполняется один раз для всего внешнего запроса.

Но кроме того есть **коррелирующие подзапросы** (correlated subquery), результаты которых зависят от строк, которые извлекаются в основном запросе.

## Использование оператора EXISTS

EXISTS (подзапрос)

Аргументом EXISTS является обычный оператор SELECT, т. е. 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.

Вообще говоря, подзапрос может выполняться не полностью, а завершаться, как только будет возвращена хотя бы одна строка. Поэтому в подзапросах следует избегать побочных эффектов (например, обращений к генераторам последовательностей); проявление побочного эффекта может быть непредсказуемым.

Так как результат этого выражения зависит только от того, возвращаются строки или нет, но не от их содержимого, список выходных значений подзапроса обычно не имеет значения. Как следствие, широко распространена практика, когда проверки EXISTS записываются в форме EXISTS(SELECT 1 WHERE ...). Однако из этого правила есть и исключения, например с подзапросами с предложением INTERSECT.

Для примера выполним выборку пользователей, совершавших покупки более чем на 1000 (рисунок 35).

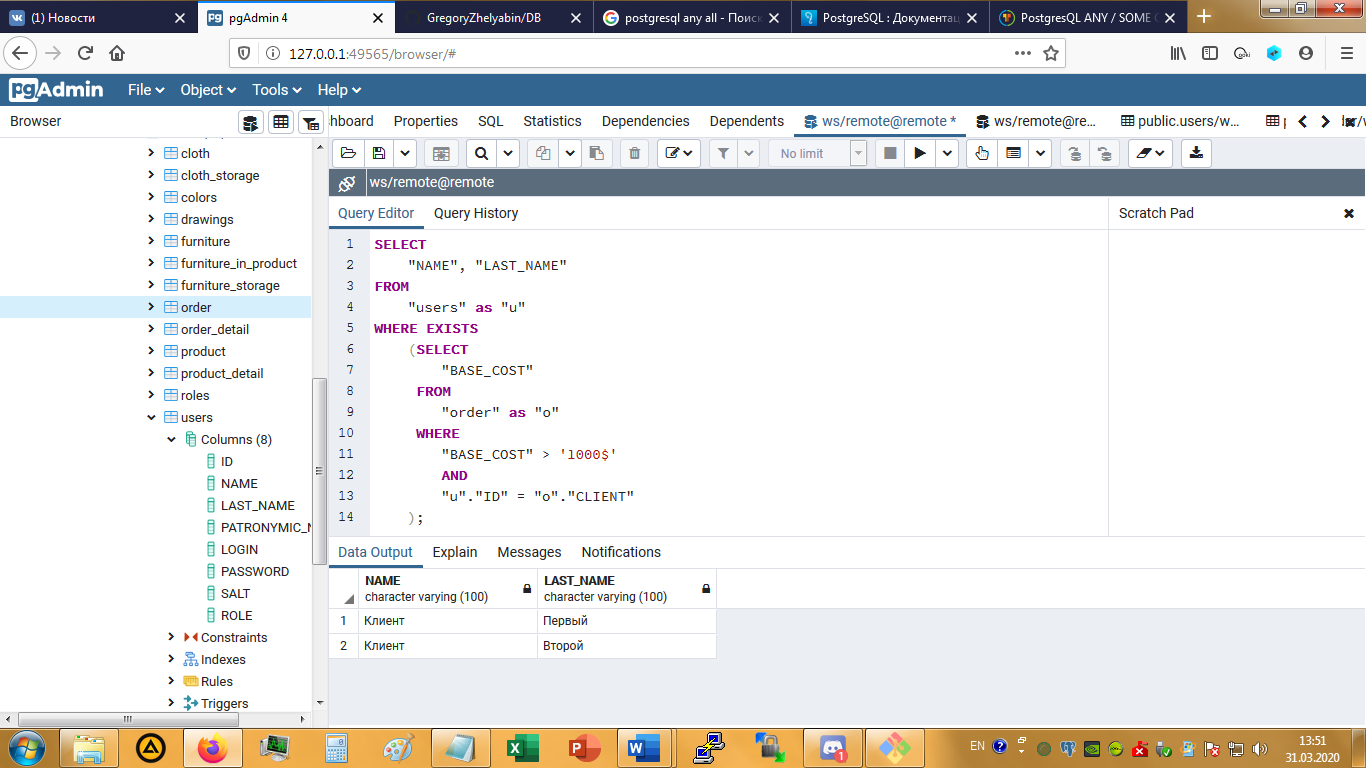


Рисунок 35 – Подзапрос с EXISTS

## Предикаты ANY и ALL

Операции сравнения можно расширить до многократного сравнения с использованием предикатов ANY и ALL. Это расширение используется при сравнении значения определенного столбца со значениями, возвращаемыми вложенным запросом. Предикат ANY, указанный после знака любой из операций сравнения, означает, что будет возвращено TRUE, если хотя бы для одного значения из подзапроса результат сравнения истинен. Предикат ALL требует, чтобы результат сравнения был бы истинен для всех значений, возвращаемых подзапросом.

Рассмотрим использование предиката **ANY**.

Например нам необходим запрос, который выведет скидки по заказам, размер которых превосходит стоимость какого-либо из товаров.

SELECT \*

FROM "order"

WHERE "DISCOUNT" > ANY (SELECT "PRICE" FROM "product")

Таким образом в данном запросе мы получаем стоимости всех товаров, и ищем по всем скидкам те, которые удовлетворяют условию что хотя бы одни из товаров по стоимости меньше скидки данного заказа.

Следует отметить, что использование сравнения «= ANY» эквивалентно использованию предиката IN.

Рассмотрим использование предиката **ALL**.

Например, требуется вывести информацию о ремонтных заявках абонентов, даты подачи заявок которых позднее, чем заявки любых абонентов с неисправностью с кодом, равным 2. Запрос будет выглядеть следующим образом:

Условие «> ALL» равносильно утверждению «больше, чем максимальное», а условие «< ALL» - «меньше, чем минимальное». Становится очевидным, что такие условия можно записать иначе, используя агрегатные функции MAX и MIN.

Результат выполнения будет таким же, как и в предыдущем примере. Следует отметить, что использование сравнения «<> ALL» эквивалентно использованию предиката NOT IN, независимо от того, простой или связанный подзапрос используется.

**Когда подзапрос возвращается пустым**

Одно значительное различие между ALL и ANY - способ действия в cитуации когда подзапрос не возвращает никаких значений. В принципе, всякий раз, ко- гда допустимый подзапрос не в состоянии сделать вывод, ALL - автоматически верен, а ANY автоматически неправилен. Это означает, что следующий запрос

SELECT \*

FROM Customers

WHERE rating > ANY

( SELECT rating

FROM Customers

WHERE city = Boston );

не произведет никакого вывода, в то врем как запрос -

SELECT

FROM Customers

WHERE rating > ALL

( SELECT rating

FROM Customers

WHERE city = 'Boston' );

выведет всю таблицу Заказчиков. Когда нет никаких заказчиков в Boston, естественно, ни одно из этих сравнений не имеет значения.