Министерство науки и образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н.Э. Баумана

(национальный исследовательский университет)»

(МГТУ им. Н.Э. Баумана)



**УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ К ЛАБОРАТОРНЫМ РАБОТАМ ПО**

**КУРСУ «БАЗЫ ДАННЫХ»**

# Лабораторная работа №2 «Создание БД»

Авторы:

Кудрявцев А.П., [kudryavtsevap@bmstu.ru](mailto:kudryavtsevap@bmstu.ru)

Фомин М.М.

Москва, 2025

# Общие сведения

## Сокращения

SQL– Structured Query Language («язык структурированных запросов»)

БД – База данных

СУБД – Система управления базами данных

РБД – Реляционная база данных

БП – Бизнес-процесс

ИС – информационная система

# 1. Проектирование схемы базы данных

## 1.1. Теоретическая часть

### 1.1.1. Модель «сущность – связь»

Модель «сущность–связь» предназначена для графического представления структуры данных предметной области. Существует несколько известных нотаций для изображения этой модели. Одна из таких, IDEF1 – методология функционального моделирования и графическая нотация, предназначенная для формализации и описания модели данных.

Для модели «сущность – связь» базовыми являются понятия:

* Сущность.
* Связь.
* Атрибут.

В настоящее время разработка БД неотделима от разработки ИС, с которой эта база будет работать, поэтому проектирование БД начинается вместе с проектированием ИС. В подавляющем большинстве проектов по разработке ИС осуществляется автоматизация бизнес-процессов (БП), поэтому основная работа по анализу и разработке требований к ПО сводится к изучению существующих и созданию новых БП.

В этом документе термин «бизнес-процесс» понимается как «совокупность взаимосвязанных действий пользователей и самой ИС, направленных на создание определённого продукта или услуги для потребителей».

При проектировании ИС создаются описания всех бизнес-процессов, которые задействованы в процессе функционирования автоматизируемого предприятия.

Описание бизнес-процесса – это описание последовательности действий сотрудников при выполнении определенных действий в графическом и текстовом виде с целью регламентации действий в коллективе, анализа и оптимизации их последовательности.

Процесс логического проектирования БД тоже может быть представлен как бизнес-процесс. Этот процесс в несколько упрощенном виде в нотации IDEF0 представлен на рисунке 1.

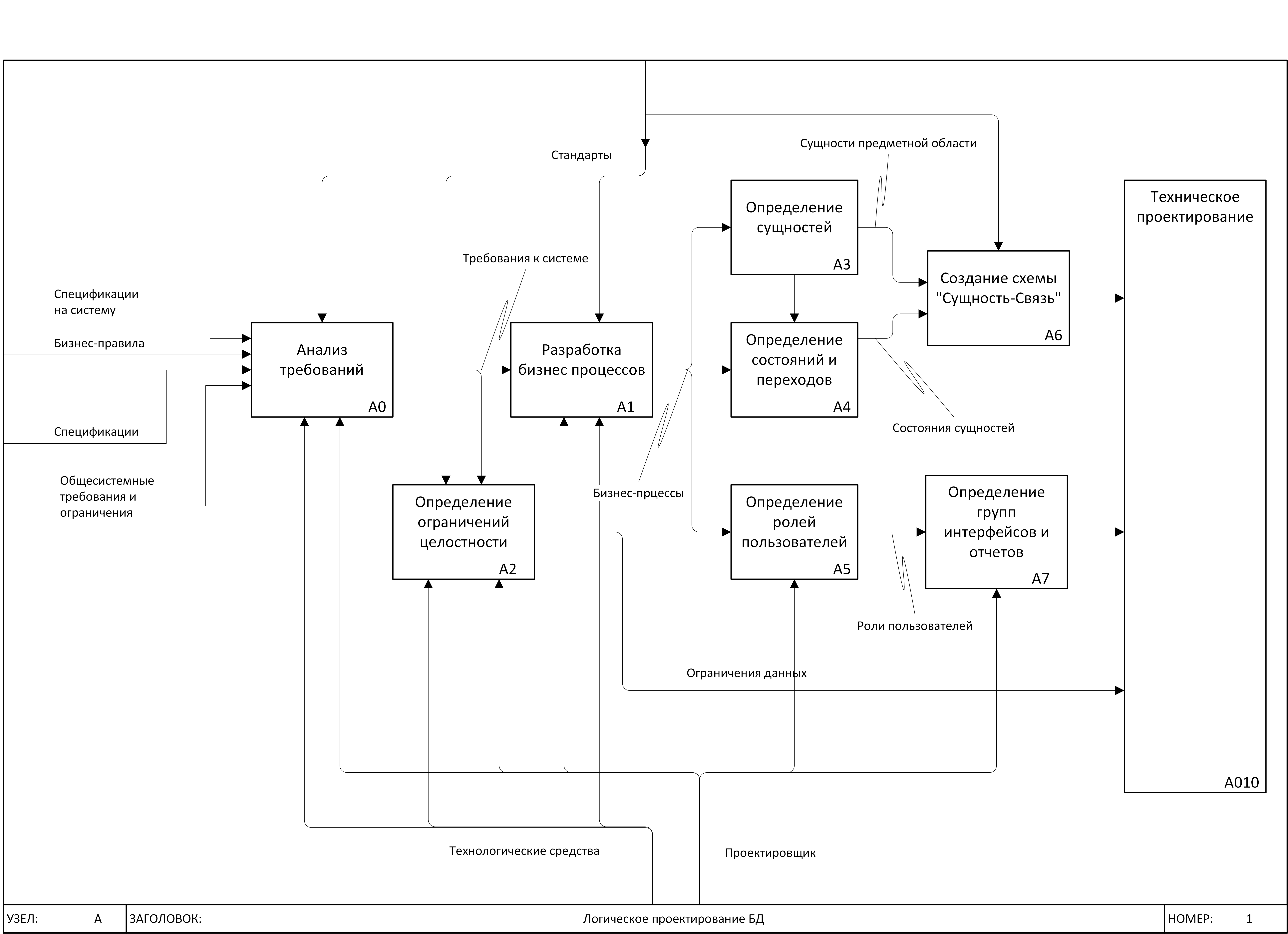


Рисунок 1 – Схема проектирования БП

### 1.1.2. Анализ бизнес-процессов

Поскольку БП — это действия пользователей в ИС и действия самой ИС, нашим первым вопросом будет – с какими объектами (или описаниями объектов) работает пользователь, какие документы поступают на вход БП и какие формируются на выходе? Это и будут сущности нашей ИС. (см. раздел «Определение сущностей»).

Второй, не менее важный вопрос, – какие пользователи выполняют те или другие действия в БП? (см. раздел «Создание ролей пользователей»).

Третий вопрос – каковы свойства и возможные состояния объектов, с которыми работает пользователь в этом бизнес-процессе (см. раздел «Создание атрибутов») и как эти свойства и состояния изменяются?

И, наконец, четвертый вопрос – какие связи существуют между сущностями предметной области?

С сущностями и атрибутами, мы ознакомились в первой лабораторной работе, а здесь рассмотрим связи между сущностями.

### 1.1.3. Виды связей между сущностями

Объекты реального мира находятся друг с другом в сложных взаимоотношениях. В БД такие взаимоотношения представляются как связи между сущностями. Под понятием «связь между сущностями А и В» мы будем понимать соответствие одному или нескольким экземплярам сущности А одного или нескольким экземплярам сущности В, например,

* Много студентов учатся в одной группе (Студенты – Группы);
* Студент изучает некоторые предметы (Студенты – Предметы);
* Врач является пациентом своей поликлиники (Врач – Пациент);
* Пациенты записываются на прием к врачам (Пациент – Врач).

Можно различать связи между двумя сущностями по количеству экземпляров, присутствующих в связи с разных сторон:

1. Связь много к одному - **много** **студентов** учатся в **одной группе;**
2. Связь один ко многим – **один студент** изучает **много предметов;**
3. Связь один к одному - **врач** является **пациентом**;
4. Связь много ко многим - **пациенты** ходят на прием к **врачам,**

т.е. один пациент может ходить на прием ко многим врачам, а к одному врачу может ходить на прием много пациентов.

Первый и второй варианты равнозначны (сравните *«****много******студентов*** *учатся в* ***одной группе****»* и *«в* ***одной группе*** *учатся* ***много******студентов****»*), а третий вариант является вырожденным случаем предыдущих. Рассмотрим, как осуществляются связи первых трех типов между двумя сущностями. В связанных сущностях присутствуют особые атрибуты, при помощи которых можно отыскать связанные экземпляры сущностей. Например, в сущности, «Студент» есть атрибут, содержащий значение, определяющее группу, к которой принадлежит студент (этот атрибут содержит номер группы). В общем случае связывающий атрибут первой сущности должен содержать значение, однозначно определяющее экземпляр второй сущности. Лучше всего для такого однозначного определения подходит первичный ключ.

### 1.1.4. Первичные и внешние ключи

Ключи, существующие в таблицах (а каждая таблица соответствует сущности), однозначно определяют экземпляры сущностей:

* первичный ключ — это поле или несколько полей таблицы (атрибут сущности), однозначно определяющий запись таблицы (например, в кинотеатре: номер зала, ряд и место), желательно, чтобы первичный ключ имел небольшой размер. В ряде случаях (пример ниже) целесообразно использовать суррогатный ключ. Поля, входящие в первичный ключ, не могут быть NULL. У таблицы может быть только один первичный ключ.
* индексы — индексы состоят из одного или нескольких полей. Индексы могут быть уникальными и неуникальными. Их активное использование играет важнейшую роль в повышении производительности СУБД. Уникальные индексы однозначно определяют запись и поэтому могут быть использованы в качестве первичных ключей. Например, студента можно однозначно определить по номеру паспорта, номеру зачетки, номеру студенческого билета, номеру пропуска.
* внешний ключ — это поле дочерней таблицы, содержащее значение первичного ключа внешней, родительской таблицы и, следовательно, однозначно определяющее запись связанной родительской таблицы. Таким образом поддерживается декларативная целостность БД. Отношение между родительской и дочерней таблицами имеет атрибуты, определяющие действия при удалении строки или изменении первичного ключа родительской таблицы, например: NO ACTION, CASCADE, SET NULL, SET DEFAULT.

Основное предназначение ключей - связывание сущностей (рис. 2):

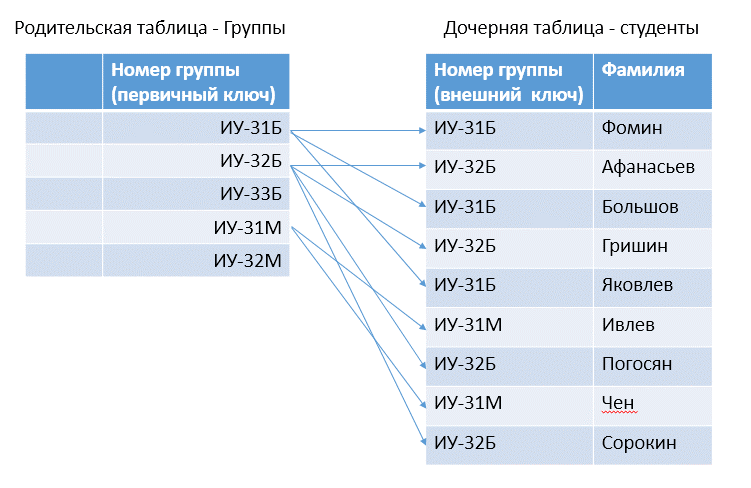


Рисунок 2 – Схема связей таблиц

В таблицах на рисунке присутствуют поле «Номер группы». В таблице «Группа» — это первичный ключ, а в таблице «Студент» поле «Номер группы» — это внешний ключ, определяющий группу, в которой учится студент.

Зная экземпляр группы и, следовательно, значение первичного ключа – «Номер группы», можно по значению внешнего ключа равному «Номер группы» выбрать всех студентов этой группы и, наоборот, по значению внешнего ключа сущности «Студент» можно найти нужный экземпляр сущности «Группа». Так работают связи «один ко многим» для таблиц (сущностей) в реляционной СУБД.

Но ведь номер группы меняется каждый семестр и, следовательно, связывать таблицы по этому атрибуту нежелательно. При проектировании БД полезно сохранять значения первичных ключей неизменными на протяжении жизненного цикла БД[[1]](#footnote-1). Выходом является использование в качестве первичного ключа суррогатного ключа. Суррогатный ключ — это целочисленный атрибут, значение которого генерируется автоматически при добавлении экземпляра сущности (записи в таблицу). В PostgreSQL для этих целей предназначены типы данных из семейства serial и IDENTITY. При использовании суррогатных ключей связи между таблицами получаются как на рисунке 3.

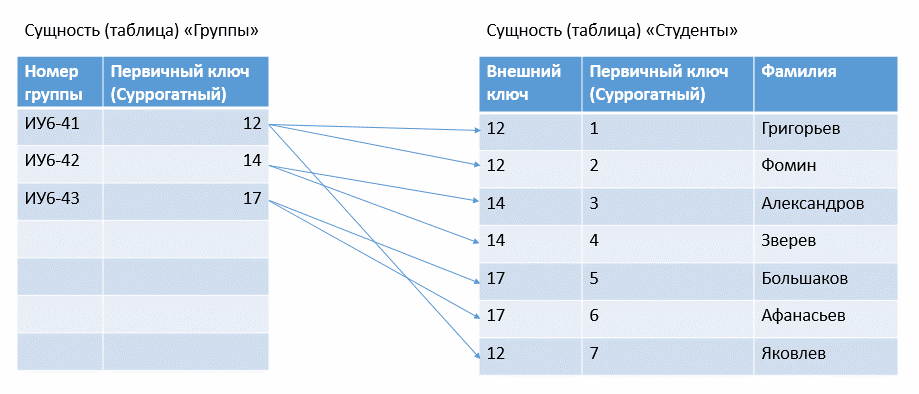


Рисунок 3 – Применение суррогатных ключей

Таким образом, номер группы меняется каждый семестр, но это нисколько не затрагивает связи в базе данных, поскольку суррогатный первичный ключ группы не меняется на протяжении всего жизненного цикла группы с момента ее создания и до перемещения данных о группе в архив. Этот суррогатный первичный номер группы уникален во всем вузе и на все времена.

### 1.1.5. Нормализация базы данных

Во многих источниках большое значение придается нормализации, она объявляется чуть ли не краеугольным камнем проектирования БД, однако в настоящее время разработчики БД обычно и не вспоминают про нормальные формы и сам процесс нормализации. Почему? Давайте разберем это подробно:

1. Понятия нормальных форм относятся к отношениям, переменным отношений, кортежам и другим понятиям реляционной алгебры, переложить их на язык СУБД весьма проблематично, например, первая нормальная форма запрещает в отношении одинаковые записи, но не одна СУБД этого не поддерживает (если мы говорим о таблицах и записях).
2. Мнение многих авторов-теоретиков расходятся по поводу «атомарности» полей таблиц, но тогда понятие первой нормальной формы становится неопределенным.
3. Для проведения нормализации необходимо выделить зависимости атрибутов, а это неформализованная и сложная процедура.

С другой стороны:

1. При использовании таких простых понятий, как суррогатный первичный ключ, мы гарантированно получаем таблицы, которые находятся во второй нормальной форме.
2. Если в качестве единственного потенциального ключа рассматривать первичный суррогатный ключ, то мы получим таблицу в усиленной третьей нормальной форме.

В настоящее время проектирование БД основывается на семантическом моделировании структуры данных, опираясь на смысл этих данных. В качестве инструмента семантического моделирования используются различные варианты диаграмм бизнес-процессов и диаграмм «сущность-связь» с использованием методологии IDEF.

При использовании правил и приемов, содержащемся в руководстве [1], проектируемая база данных автоматически получается в третьей нормальной форме, а даже теоретики утверждают, что дальнейшая нормализация излишня.

## 1.2. Проектирование схемы базы данных

### 1.2.1. Проектирование базы данных «Авиаперелеты»

Сначала необходимо построить схемы бизнес-процессов, которые используются в авиакомпании, для примера мы выберем только один процесс, наиболее известный – покупка авиабилетов online. На рисунке 4 вы видите упрощенную схему этого бизнес-процесса в нотации IDEF0.

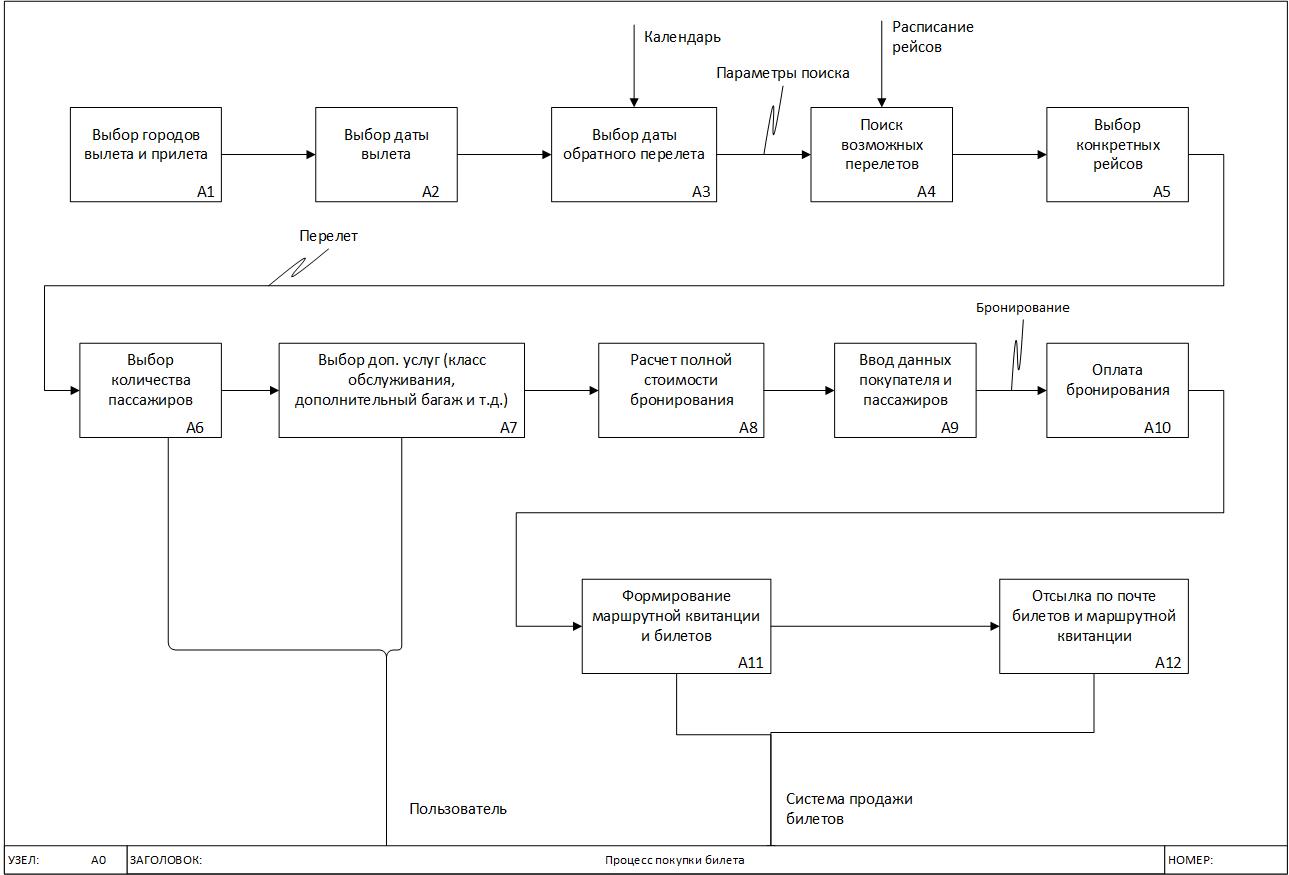


Рисунок 4 – Схема БД «Авиаперевозки»

Исходя из бизнес-процесса, показанного на рисунке, можно определить следующие сущности, которыми оперирует пользователь и система продажи билетов:

* Город
* Перелет
* Рейс
* Аэродром
* Билет
* Доп. Услуги
* Список доп. услуг
* Модель самолета
* Пассажир
* Покупатель

Конечно, чтобы определить параметры атрибутов, типы связей, необходимо знать огромное количество информации о предметной области.

Итак, некая российская авиакомпания выполняет пассажирские авиаперевозки. Она обладает своим парком самолетов различных моделей. Каждая модель самолета имеет определенный код, который присваивает Международная ассоциация авиаперевозчиков (IATA). При этом будем считать, что самолеты одной модели имеют одинаковые компоновки салонов, т. е. порядок размещения кресел и нумерацию мест в салонах бизнес-класса и экономического класса. Например, если это модель Sukhoi SuperJet-100, то место 2A относится к бизнес-классу, а место 20D – к экономическому классу. Бизнес-класс и экономический класс – это разновидности так называемого класса обслуживания. Наша авиакомпания выполняет полеты между аэропортами России. Каждому аэропорту присвоен уникальный трехбуквенный код, при этом используются только заглавные буквы латинского алфавита. Эти коды присваивает не сама авиакомпания, а специальные организации, управляющие пассажирскими авиаперевозками. Зачастую название аэропорта не совпадает с названием того города, которому этот аэропорт принадлежит. Например, в городе Новосибирске аэропорт называется Толмачево, в городе Екатеринбурге – Кольцово, а в Санкт-Петербурге – Пулково. К тому же некоторые города имеют более одного аэропорта. Сразу в качестве примера вспоминается Москва с ее аэропортами Домодедово, Шереметьево и Внуково. Формируются маршруты перелетов между городами, возможно с пересадками, т.к. прямых рейсов может и не быть. Каждый такой маршрут состоит из рейсов, которые связывают аэропорты, поскольку, как мы уже сказали, в городе может быть более одного аэропорта. Каждый рейс имеет шестизначный номер, включающий цифры и буквы латинского алфавита. На основе перечня рейсов формируется маршрутная квитанция, в которой указываются плановое время отправления и плановое время прибытия, а также тип самолета, выполняющего этот рейс. При фактическом выполнении рейса возникает необходимость в учете дополнительных сведений, а именно: фактического времени отправления и фактического времени прибытия, а также статуса рейса. Статус рейса может принимать ряд значений: 15 – Scheduled (за месяц открывается возможность бронирования); – On Time (за сутки открывается регистрация); – Delayed (рейс задержан); – Departed (вылетел); – Arrived (прибыл); – Cancelled (отменен). Полет начинается с бронирования авиабилета. В настоящее время общепринятой практикой является оформление электронных билетов. Каждый такой билет имеет уникальный номер, состоящий из 13 цифр. В рамках одной процедуры бронирования может быть оформлено несколько билетов, но каждая такая процедура имеет уникальный шестизначный номер (шифр) бронирования, состоящий из заглавных букв латинского алфавита и цифр. Кроме того, для каждой процедуры бронирования записывается дата бронирования и рассчитывается общая стоимость оформленных билетов. В каждый билет, кроме его тридцатизначного номера, записывается идентификатор пассажира, а также его имя и фамилия (в латинской транскрипции) и контактные данные. В качестве идентификатора пассажира используется номер документа, удостоверяющего личность. Конечно, пассажир может сменить свой документ, а иной раз даже фамилию и имя, за время, прошедшее между бронированием билетов в разные дни, поэтому невозможно наверняка сказать, что какие-то конкретные билеты были оформлены на одного и того же пассажира. В каждый электронный билет может быть вписано более одного рейса. Специалисты называют эти записи о перелетах сегментами. В качестве примера наличия нескольких сегментов можно привести такой: Красноярск – Москва, Москва – Анапа, Анапа – Москва, Москва – Красноярск. При этом возможно в рамках одного бронирования оформить несколько билетов на различных пассажиров. Для каждого перелета указывается номер рейса, аэропорты отправления и назначения, время вылета и время прибытия, а также стоимость перелета. Кроме того, указывается и так называемый класс обслуживания: экономический, бизнес и др. Когда пассажир прибывает в аэропорт отправления и проходит регистрацию билета, оформляется так называемый посадочный талон. Этот талон связан с авиабилетом: в талоне указывается такой же номер, который имеет электронный авиабилет данного пассажира. Кроме того, в талоне указывается номер рейса и номер места в самолете. Указывается также и номер посадочного талона – последовательный номер, присваиваемый в процессе регистрации билетов на данный рейс. Напомним, что каждому креслу в салоне самолета соответствует конкретный класс обслуживания. Данная информация учитывается при регистрации билетов и оформлении посадочных талонов. Если, например, пассажир приобрел билет с экономическим классом обслуживания, то в его посадочном талоне будет указан номер места в салоне экономического класса, но не в салоне бизнес-класса.

Следуя приведенному описанию предметной области, можно спроектировать схему данных. Но в нашем случае БД «Авиаперевозки» уже создана (demo), и нам остается воспользоваться ею, чтобы сформировать   
ER–модель. Сформируем с помощью pgAdmin ER–модель для БД demo.

Нажмем правой клавишей на узел demo и выберем команду меню *ERD For Database* (рис. 5):

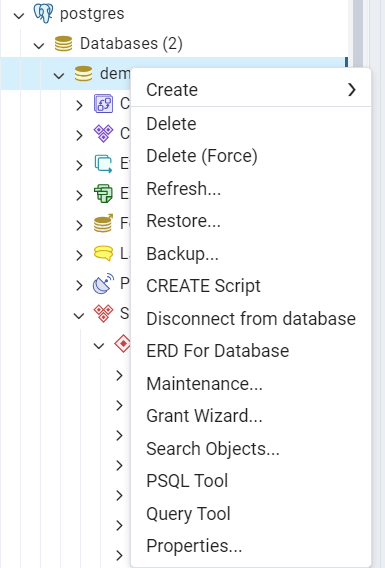


Рисунок 5 – Контекстное меню

После этого отобразится схема (рис. 6):

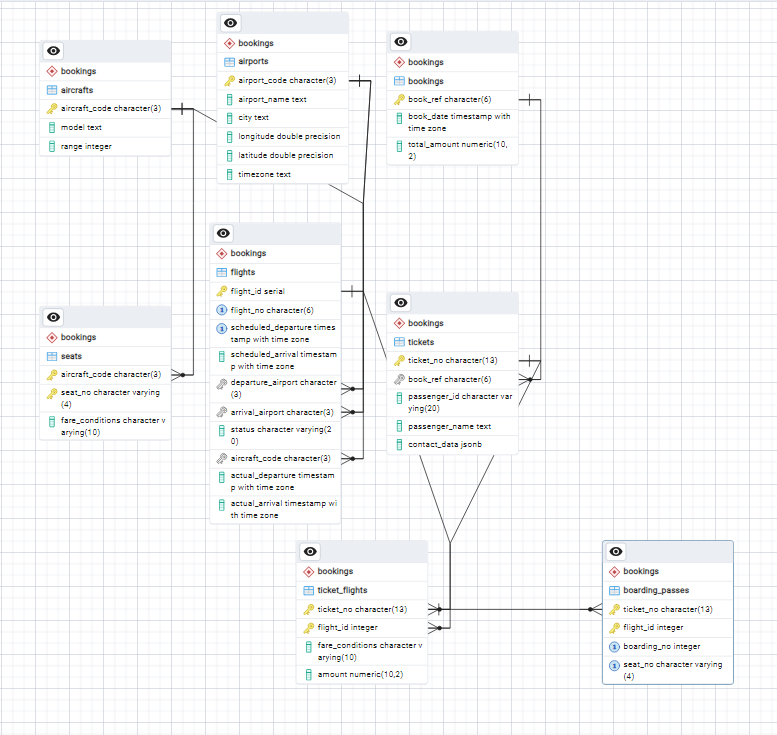


Рисунок 6 – ER-модель БД demo

Прямоугольники сущностей и связи можно перемещать, делая схему более читаемой.

Схема БД demo использует нотацию Мартина (Crow's Foot Notation). Связи изображаются линиями (рис. 7), соединяющими сущности; вид линии в месте соединения с сущностью определяет кардинальность связи:

|  |  |
| --- | --- |
| **Обозначение** | **Кардинальность** |
|  | один |
|  | один и только один |
|  | ноль или один |
|  | много |
|  | ноль или много |
|  | один или много |

Рисунок 7 – Кардинальность связи

### 1.2.2. Построение ER–модели базы данных

Для проектирования БД существует несколько средств разработки структуры БД: ERwin, MS Visio, Lucidchart и др. Но мы будем использовать инструмент, предоставляемый pgAdmin.

Будем разрабатывать упрощенную БД ВУЗ (university), которая должна обеспечить составления расписания занятий для групп. Очевидно, что в нее должны входить сущности «студент» (таблица 2) и «группа» (таблица 1). Отношение между двумя этими сущностями - один-ко-многим, так как группа состоит из нескольких студентов, и каждый студент принадлежит только одной группе. Этот факт отмечается наличием в сущности «студент» внешнего ключа (id\_group), ссылающегося на строку в сущности «группа».

**Таблица 1. Группа (group)**

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| id\_group | serial | Суррогатный первичный ключ |
| name | varchar(10) | Группа |

**Таблица 2. Студент (student)**

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| id\_student | serial | Суррогатный первичный ключ |
| id\_group | int | id группы, внешний ключ |
| name | varchar(50) | Фамилия, имя, отчество |
| … | ... | ... |

Сущность «аудитория»**[[2]](#footnote-2)** (таблица 3) тоже очевидна.

**Таблица 3. Аудитория (auditorium)**

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| id\_auditorium | serial | Суррогатный первичный ключ |
| room\_no | varchar(10) | Номер помещения |
| … | … | … |

Перейдем к сущности «расписание». В каждой аудитории в разное время могут проходить разные занятия, с другой стороны, строка сущности «расписание» однозначно ссылается на сущность «аудитория». Из чего можно заключить, что связь между сущностями «аудитория» и «расписание» один-ко-многим.

Как отмечалось выше, строка сущности «расписание» (таблица 4) должна однозначно определяться для каждой сущности «аудитории», т.е. в данный момент времени в аудитории не может проходить более одного занятия. Учитывая специфику ВУЗ’а, однозначность достигается следующими атрибутами: год, семестр, день недели, время начала занятий, продолжительность или время конца занятий и аудитория.

В «расписание» надлежит добавить атрибуты: тип занятия (лекции, семинар и т.п.), преподаватель и т.д., но на данный момент это для нас неважно и ограничимся атрибутом – тип занятия.

**Таблица 4. Расписание (schedule)**

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| id\_schedule | serial | Суррогатный первичный ключ |
| id\_auditorium | int | id аудитории, внешний ключ |
| year | smallint | Год |
| semester | smallint | 0 – осенний семестр, 1 – весенний |
| day\_week | smallint | День недели |
| time\_start | time | Время начала занятий |
| time\_end | time | Время конца занятий |
| type\_lesson | smallint | 0 – лекция, 1 – семинар и т.п. |
| … | … | … |

Нам осталось связать сущности «группа» и «расписание». С одной стороны то, что на занятиях может присутствовать несколько групп, в терминах БД сущности «расписание» соответствует несколько сущностей «группа». С другой, каждая группа в течения дня посещает несколько занятий, т.е. сущности «группа» соответствует несколько сущностей «расписание». Отсюда следует, что связь между группой и расписанием многие-ко-многим.

Чтобы создать связи многие-ко-многим, необходимо создать новую таблицу для подключения двух других. Эта новая таблица называется ассоциативной таблицей. Для этих связей необходимо просто соединить соответствующие поля с помощью строки.

В нашем случае, такой ассоциативной таблицей будет таблица group\_schedule (таблица 5). Одна строка посредством внешних ключей соединяет «группу» и «расписание».

**Таблица 5. Группа-расписание (group\_schedule)**

|  |  |  |
| --- | --- | --- |
| **Атрибут** | **Тип** | **Описание** |
| id\_group | int | id группы, внешний ключ |
| id\_schedule | int | id расписания, внешний ключ |

Таким образом, с помощью ассоциативной сущности связь многие-ко-многим мы представили двумя связями: один-ко-многим между сущностями «группа» - «группа-расписание» и «расписание» - «группа-расписание».

В качестве первичного ключа для сущности «группа-расписание» целесообразно использовать пару (id\_group, id\_schedule).

Займемся построением ER–модели. Запустим pgAdmin. Перед тем, как строить ER–модель, необходимо создать БД university.

Щелкните правой кнопкой по узлу Databeses и выберите пункт меню *Create/ Database...* (рис. 8):

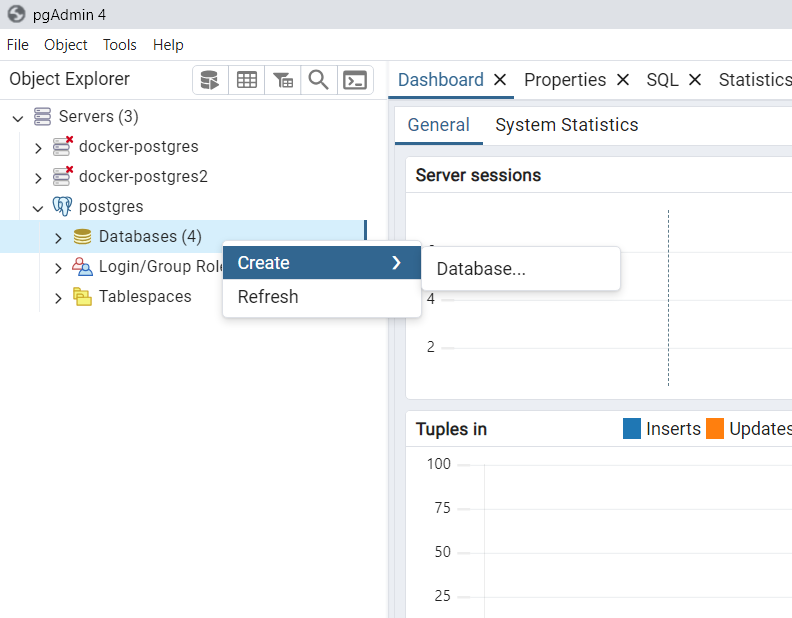


Рисунок 8 – Контекстное меню

Появится диалоговое окно для ввода подробной информации о новой базе данных (рис. 9).

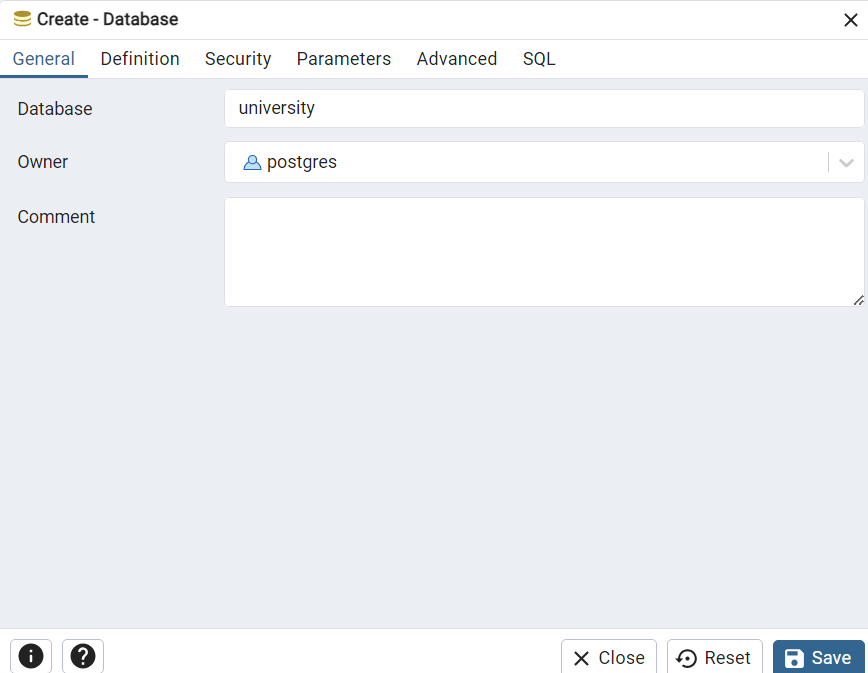


Рисунок 9 – Диалог создания БД

Введем название новой БД и сохраним. Щелкните правой кнопкой по узлу university и выберем команду меню *ERD For Database* (рис.10):

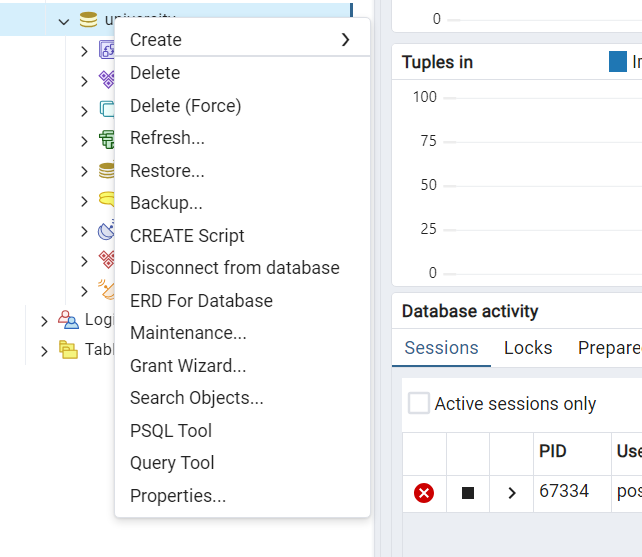


Рисунок 10 – Диалог создания БД

Откроется окно для проектирования БД (рис. 11):

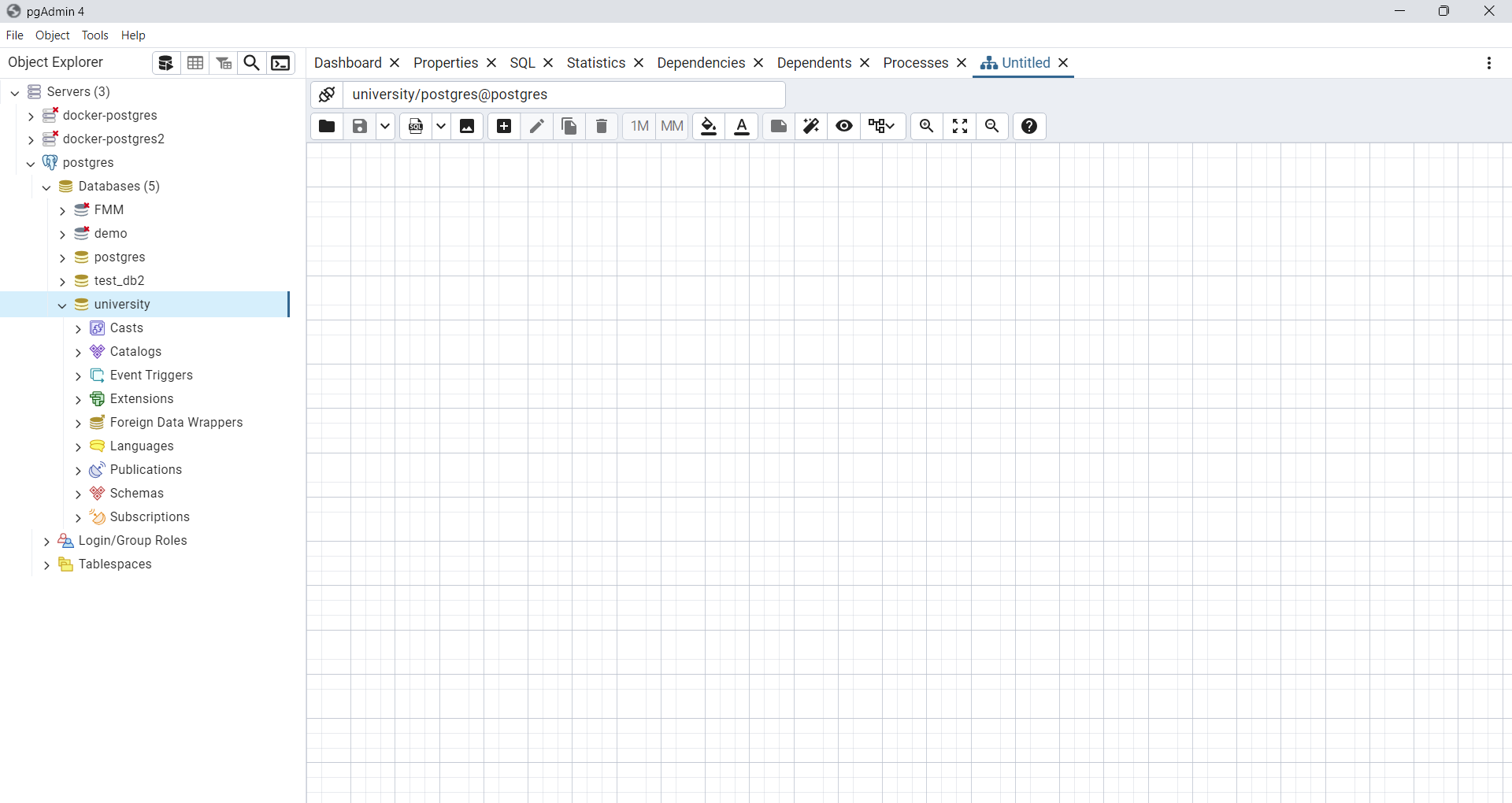


Рисунок 11 – ERD-панель

Панель инструментов ERD *Tool* (рис. 12) использует контекстно-зависимые значки, которые обеспечивают ярлыки для часто выполняемых задач. Опция включена для выделенного значка и отключена для серого значка.

Панель инструментов ERD

Рисунок 12 – ERD Tool

Нажмем на значок , чтобы приступить к созданию таблицы. Откроется диалоговое окно таблицы (рис. 13):

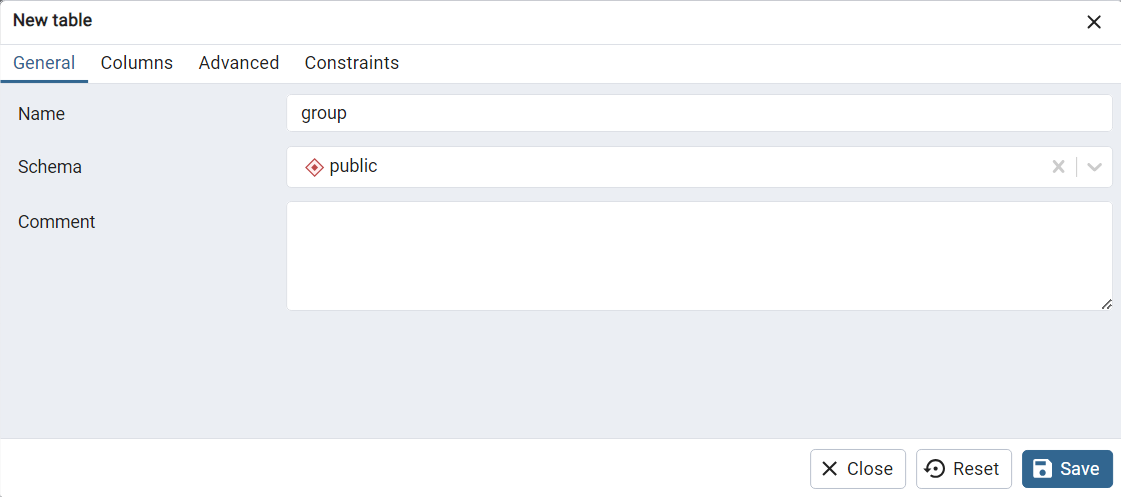


Рисунок 13 – Диалог таблицы

Введем название таблицы и перейдем к вводу атрибутов (полей), выбрав вкладку Columns (рис. 14):

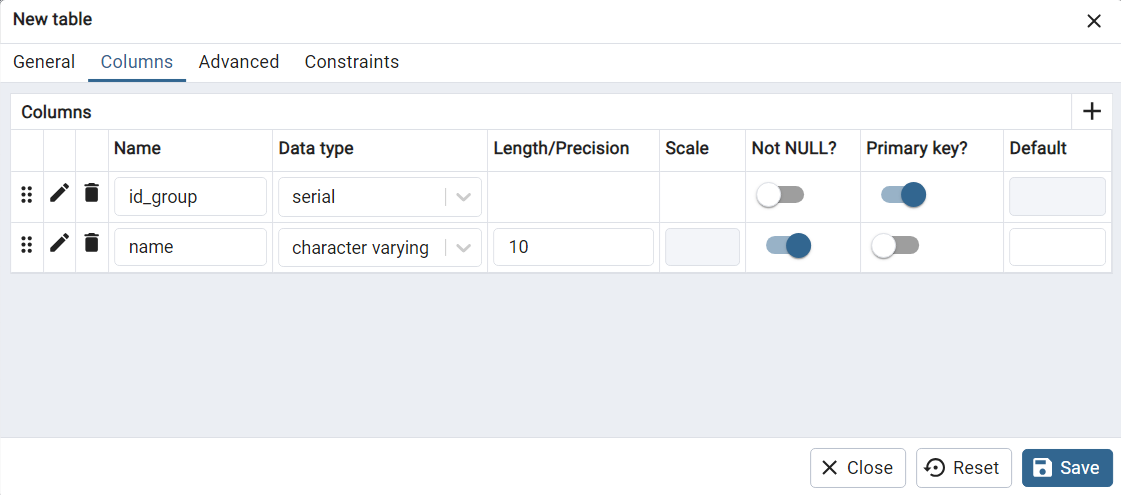


Рисунок 13 – Вкладка «колонки»

Перед тем, как вводить поле, нажмите на «+» в верхнем правом углу вкладки. Отметим, что id\_group будет первичным ключом, а название группы не может быть NULL (название группы должно быть введено, но это ограничение не предотвращает ввода пустой строки) и сохраним (рис. 14):

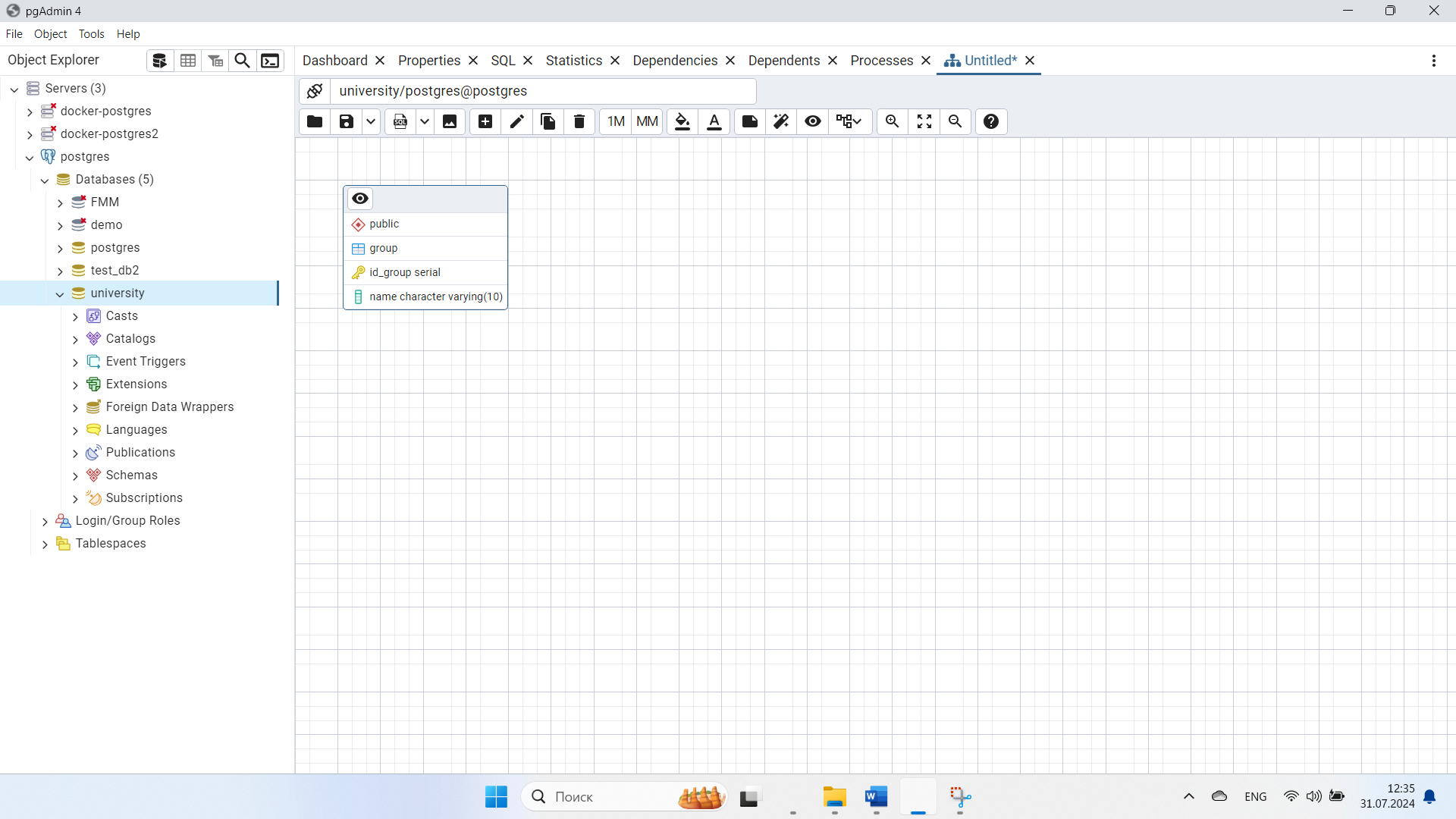


Рисунок 14 – Схема таблицы group

Аналогичным образом введем остальные таблицы, кроме group\_schedule (рис. 15):

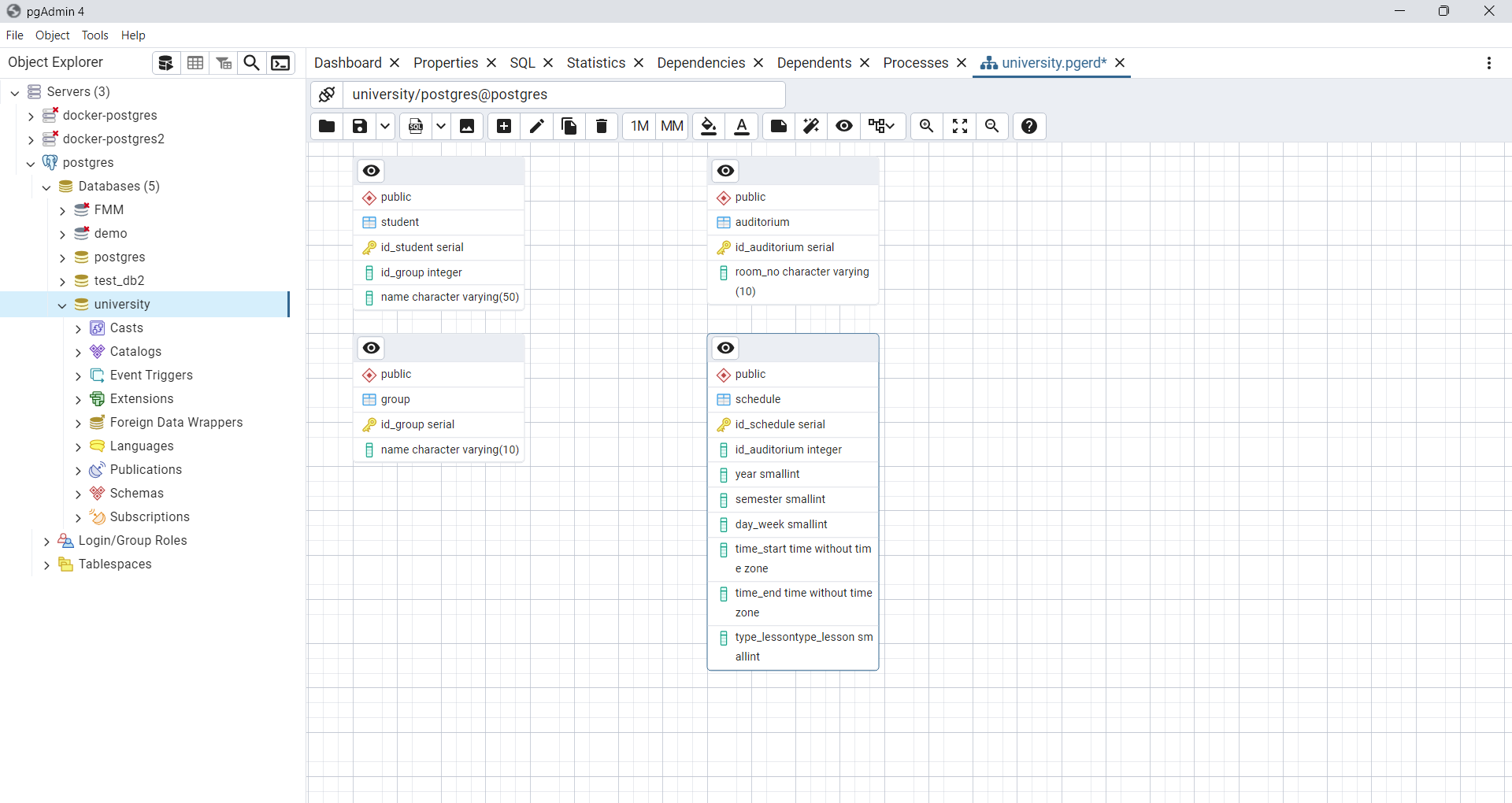


Рисунок 15 – Схема таблиц

Организуем связь, между таблицами student и group. Щелкнем по таблице student, а затем нажмем на панели инструментов значок 1M (один-ко-многим). Откроется диалог связей один-ко-многим (рис. 16):

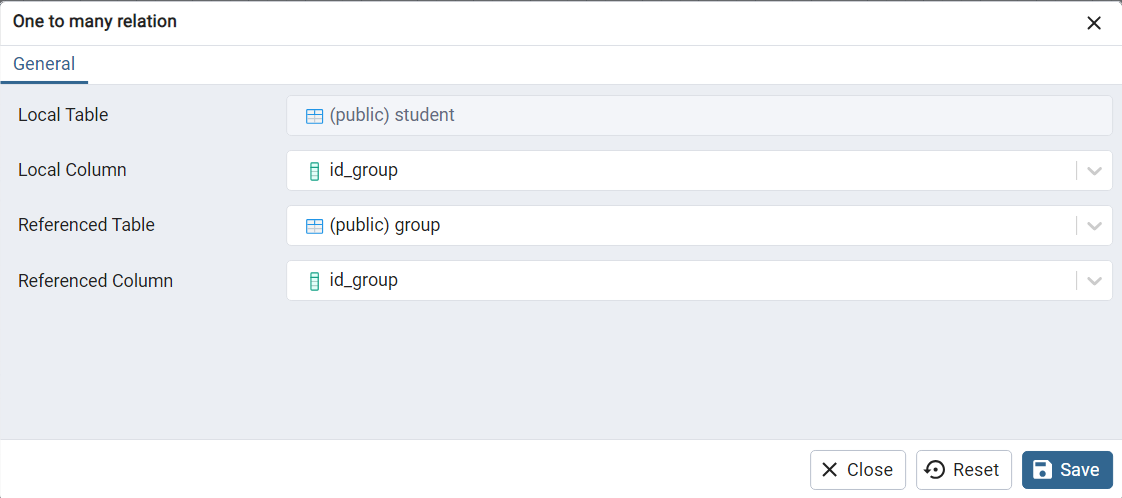


Рисунок 16 – Диалог связей один-ко-многим

Заполним поля, выбирая соответствующие значения из выпадающих списков и сохраним.

Так же организуем связь между schedule и auditorium (рис. 17):

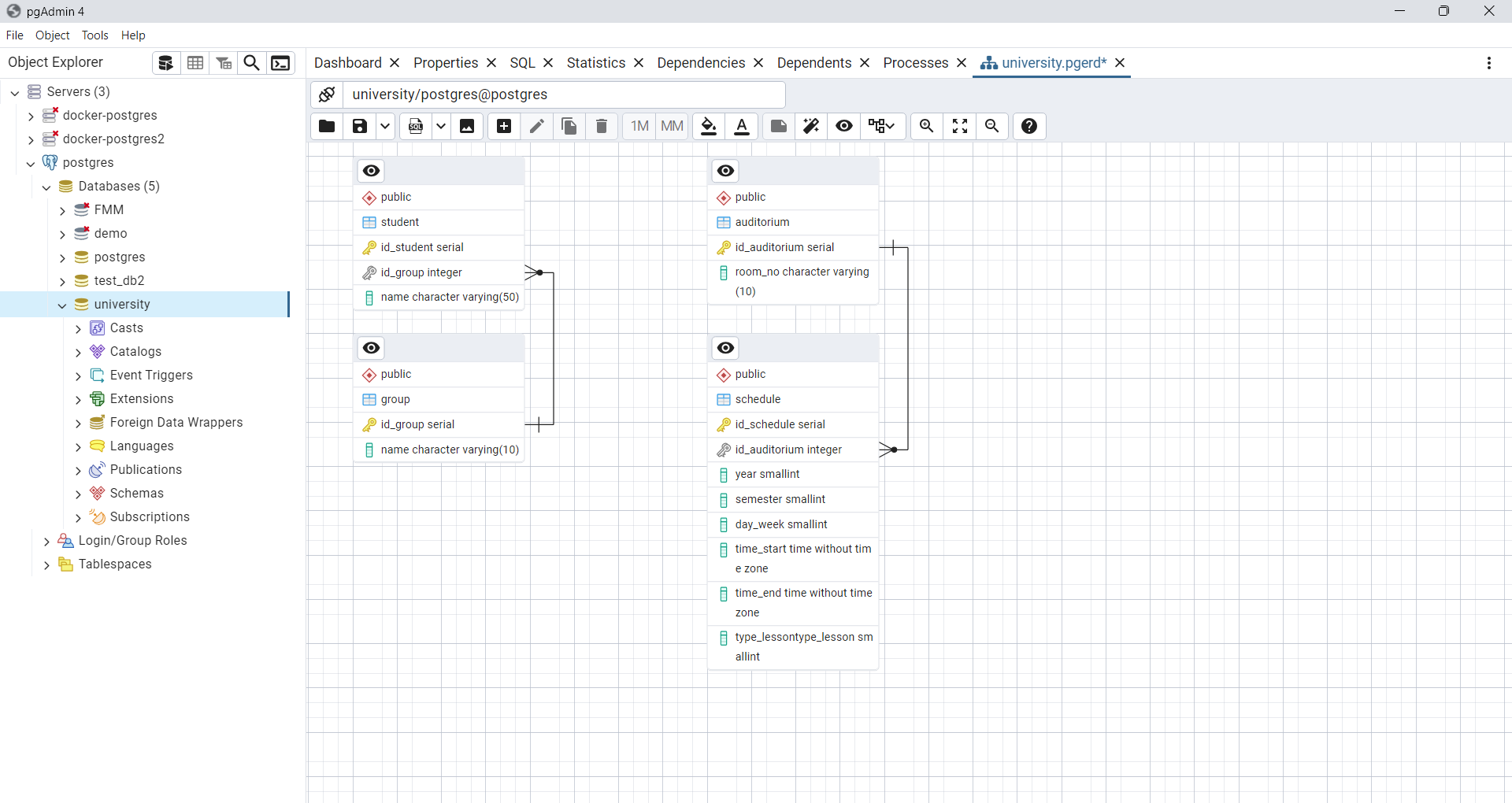


Рисунок 17 – ERD- модель со связями

Можно было ввести таблицу group\_schedule и организовать соответствующие связи, но воспользуемся возможностью pgAdmin. Щелкнем по таблице group, а потом нажмем на значок связи MM (многие-ко-многим) на панели инструментов. Откроется соответствующий диалог (рис. 18):

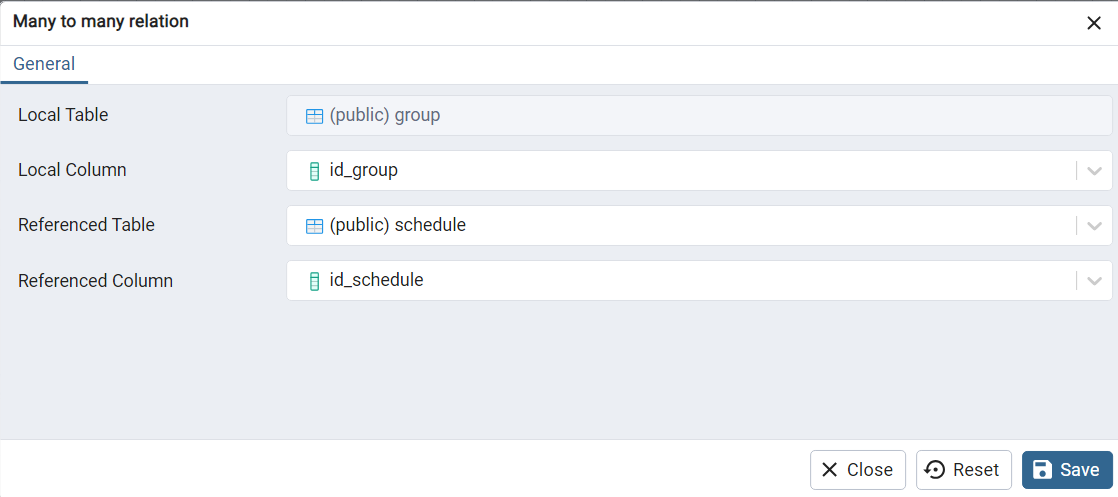


Рисунок 18 – Диалог связей многие-ко-многим

Заполним поля и сохраним (рис. 19):

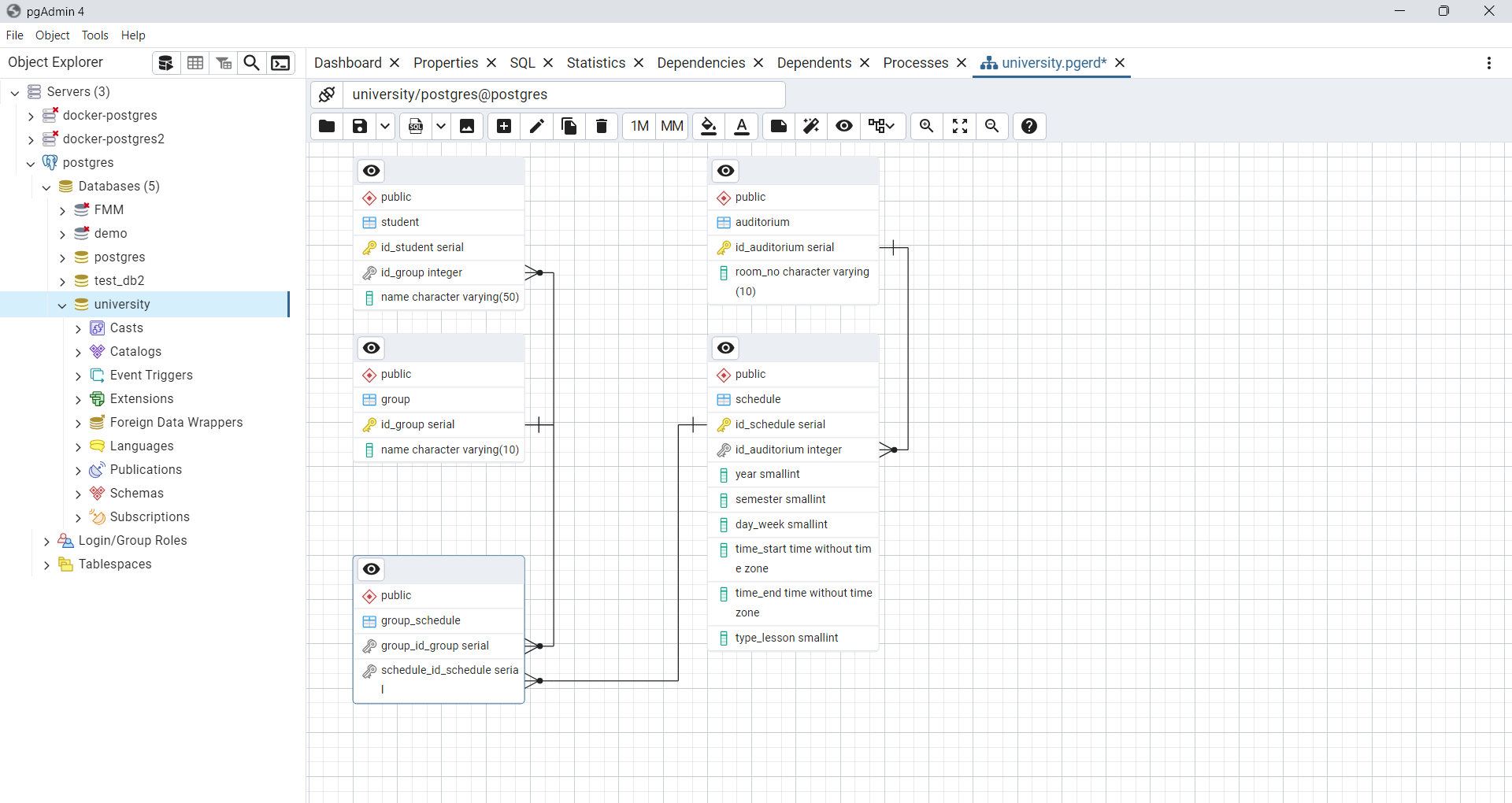


Рисунок 19 – ERD- модель

Как видим, автоматически создалась ассоциативная таблица и связи с таблицами group и schedule. Название таблицы нас устраивает (алгоритм формирования имени таблицы очевиден). Название полей можно оставить, но удобно, когда имя поля внешнего ключа совпадал с именем поля первичного ключа. А вот тип полей (serial) нас не устраивает. Отредактируем таблицу group\_schedule. Для этого дважды щелкнем по ней, отредактируем поля таблицы и создадим первичный ключ (рис. 20):

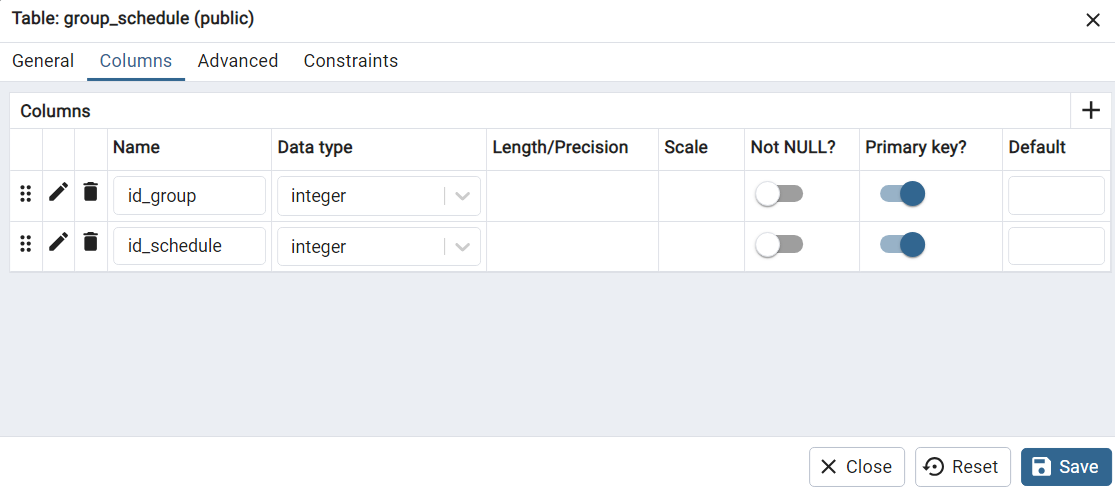


Рисунок 20 – Диалог таблицы group\_schedule

Сохраним и получаем:

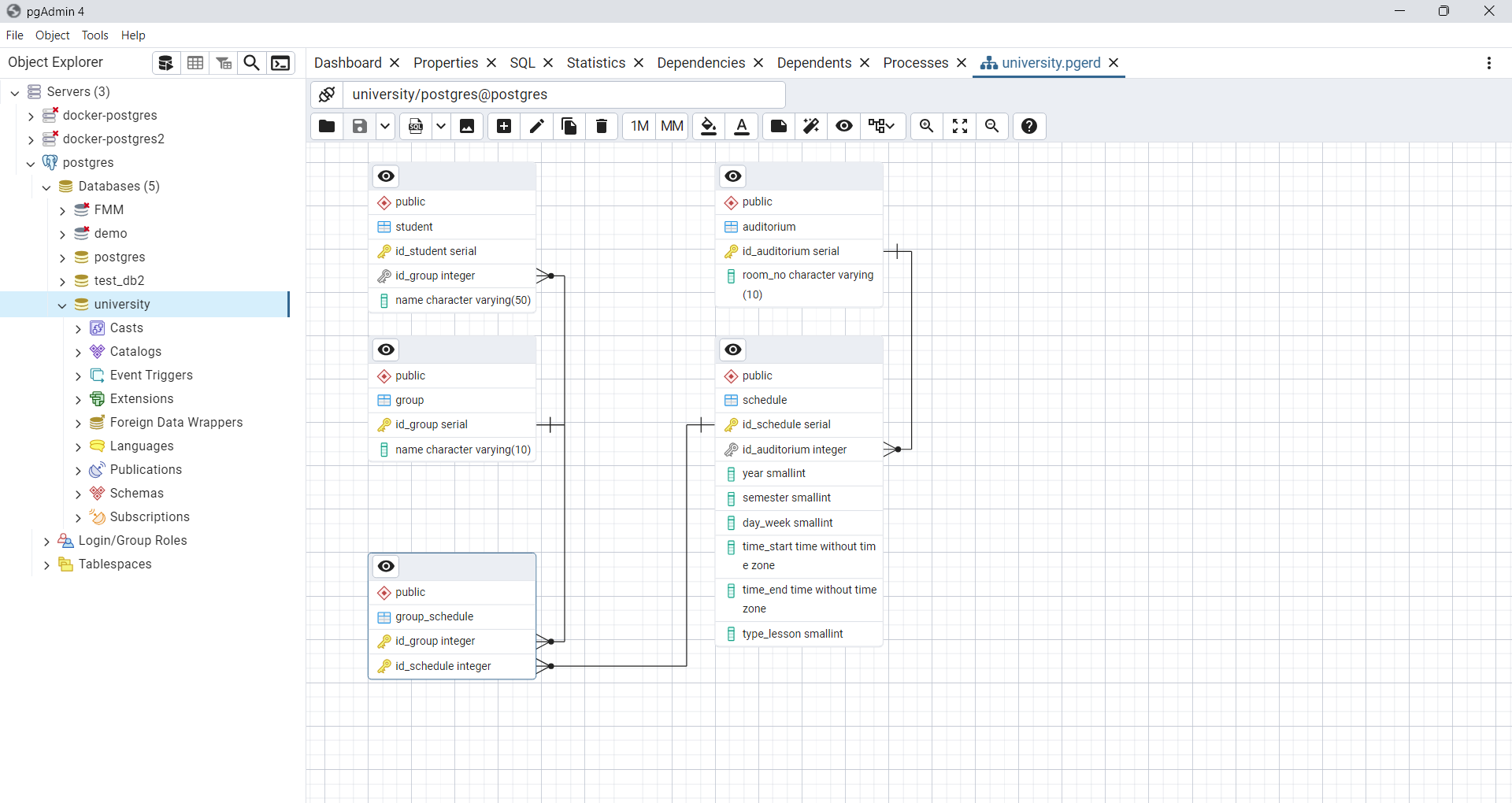


Рисунок 21 – ERD- модель завершенная

Сохраним ER–модель, нажав на значок дискета.

Все связи сформированы с опциями по умолчанию. Большинство опций можно оставить без изменения, только одну может потребоваться изменить. Рассмотрим следующею ситуацию. Клиент удаляет строку в таблице group, на эту строку ссылаются записи из таблицы student. В СУБД имеется возможность указать, что при удалении родительской записи (group) удалять дочерние записи (student), проставив опцию *CASCADE*, либо формировать ошибку, указав *NO ACTION* (поведение по умолчанию). В этом случае таблицы group и student редко изменяются, поэтому, скорее всего, это упущение при разработке программы и допускать каскадное удаление будет ошибкой.

Рассмотрим таблицу schedule и ассоциативную таблицу group\_schedule. При формировании расписания перед началом семестра (даже при использовании программы) не все факторы могут быть учтены (например, лопнула труба и затопило аудиторию), приходится активно редактировать, удалять строки расписания. И, если связь между таблицами schedule и group\_schedule имеет опцию *NO ACTION*, то перед тем, как удалить строку из group, надо удалить строки из group\_schedule, ссылающиеся на нее. Чтобы избежать дополнительной работы, достаточно выбрать опцию *CASCADE*.

*CASCADE* и *NO ACTION* наиболее часто используемые опции.

Отредактируем связь между таблицами schedule и group\_schedule. Дважды щелкнем по таблице group\_schedule. И в отрывшемся диалоге перейдем во вкладку *Constrains*, в ней выбираем вкладку *Foreign Key*, в ней нажимаем на значок редактирования связи schedule и group\_schedule (рис. 22):

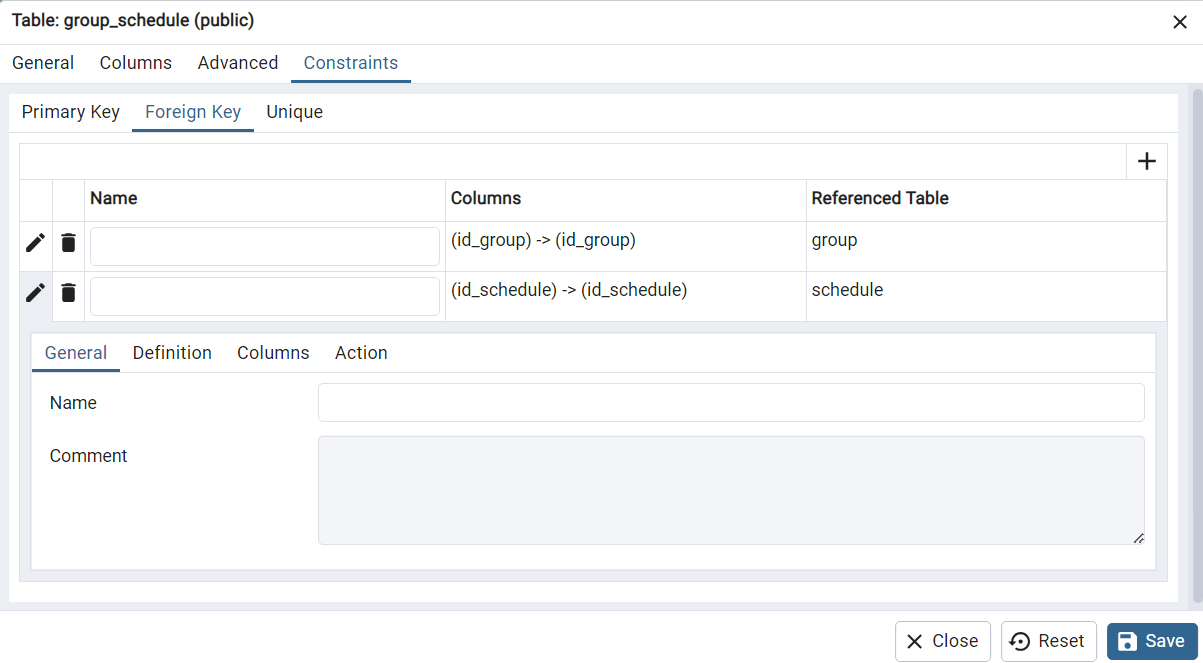


Рисунок 22 – Вкладка «внешние ключи»

Далее, *Action* и для операции удаления выбираем – *CASCADE* (рис. 23):

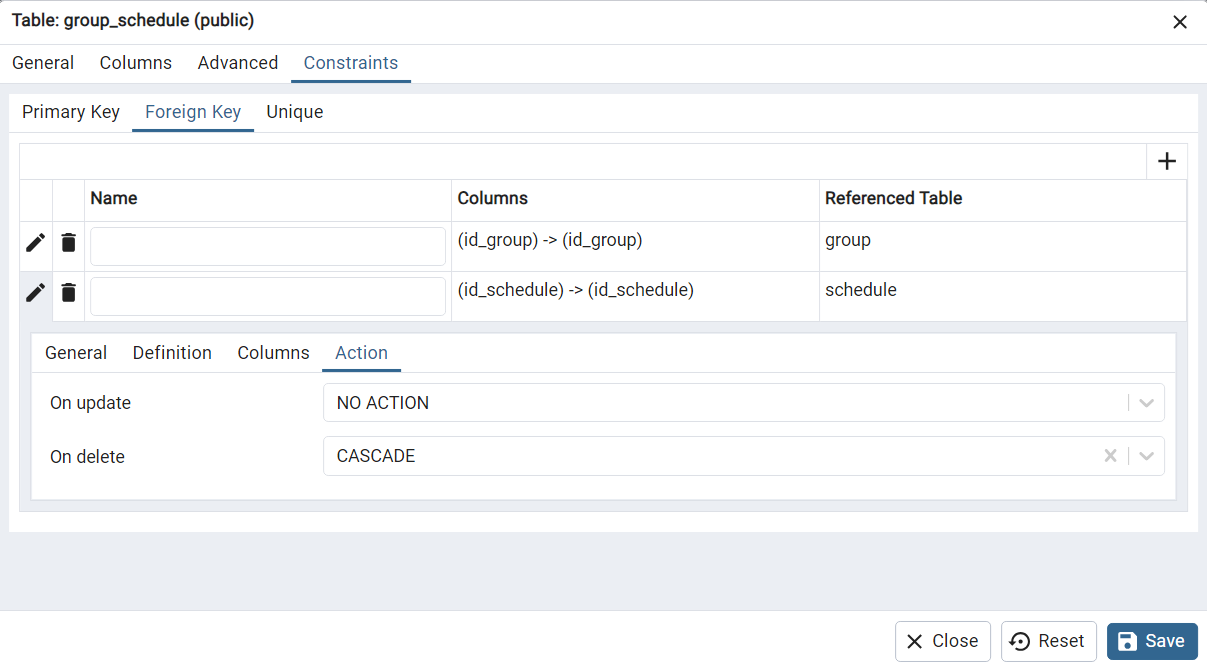


Рисунок 23 – Вкладка «действия»

Сохраним изменения.

Построив ER–модель, обсудив с коллегами и учтя их замечания, приступаем к созданию таблиц в PostgreSQL. Возникает вопрос, неужели по второму разу вводить таблицы и связи? Конечно нет. На панели инструментов имеется значок  (Generate SQL), нажав который сформируется SQL-скрипт, содержащий наши труды по разработке БД (рис. 24):

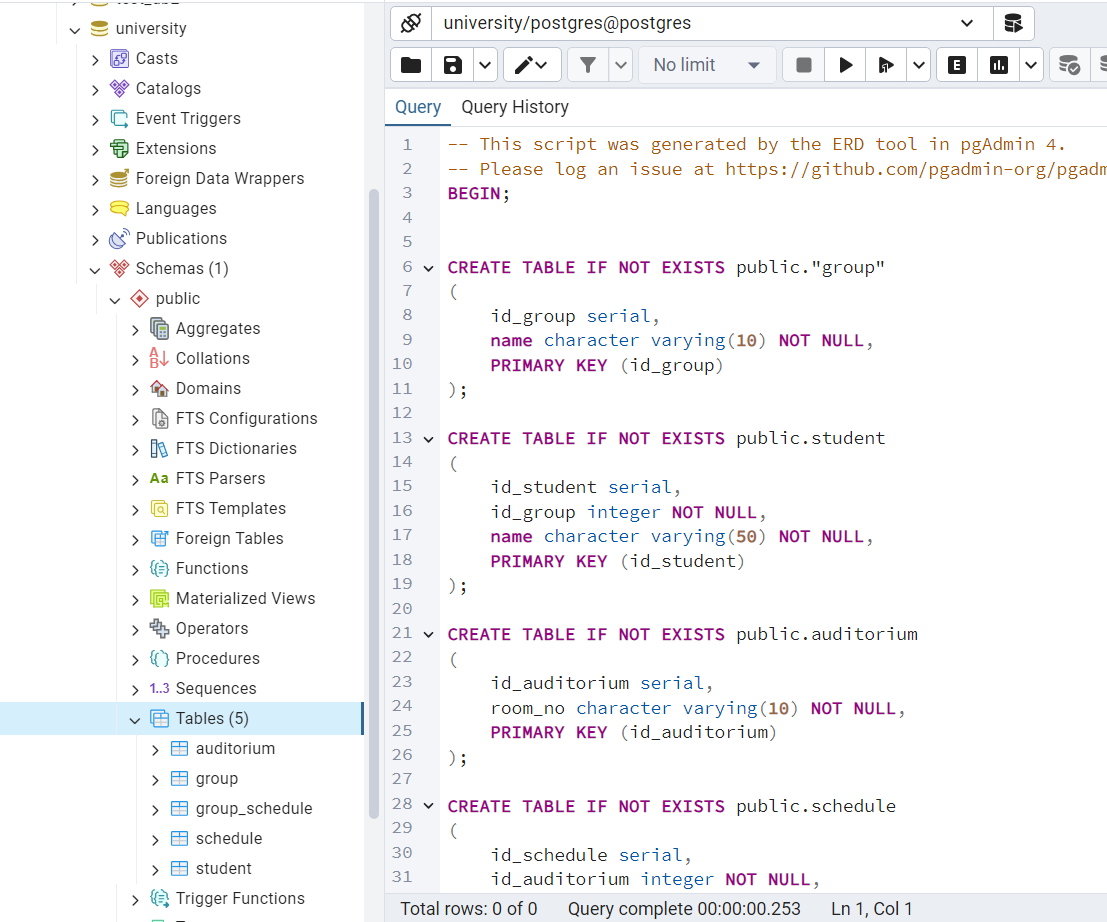


Рисунок 24 – SQL-скрипт создания БД

Запустим SQL-скрипт. И БД university сформируются таблицы, соответствующие нашей ER–модели.

# 2. Создание базы данных и таблиц

## 2.1. Создание базы данных

Чтобы создать базу данных, необходимо быть суперпользователем[[3]](#footnote-3) или иметь специальное право CREATEDB. Ниже приведен укороченный синтаксис для создания базы данных (полный синтаксис см. в документации <https://postgrespro.ru/docs/postgrespro/16/sql-createdatabase>):

CREATE DATABASE ***имя\_базы\_данных***

[ WITH ]

[ OWNER [=] ***имя\_пользователя*** ]

[ TEMPLATE [=] *шаблон* ]

[ ENCODING [=] ***кодировка*** ]

[ LC\_COLLATE [=] ***категория\_сортировки*** ]

[ LC\_CTYPE [=] ***категория\_типов\_символов*** ]

[ TABLESPACE [=] ***табл\_пространство*** ]

[ CONNECTION LIMIT [=] ***предел\_подключений*** ]

[ IS\_TEMPLATE [=] ***это\_шаблон*** ]

Параметры.

имя\_базы\_данных

Имя создаваемой базы данных.

***имя\_пользователя***

Имя пользователя (роли), назначаемого владельцем новой базы данных, либо DEFAULT, чтобы владельцем стал пользователь по умолчанию (а именно, пользователь, выполняющий команду).

шаблон

Имя шаблона, из которого будет создаваться новая база данных, либо DEFAULT, чтобы выбрать шаблон по умолчанию (template1).

***кодировка***

Этот параметр позволяет указать кодировку набора символов для новой базы данных. По умолчанию – UTF8.

***категория\_сортировки***

Порядок сортировки (LC\_COLLATE), который будет использоваться в новой базе данных. Этот параметр определяет порядок сортировки строк, например, в запросах с ORDER BY, а также порядок индексов по текстовым столбцам.

***категория\_типов\_символов***

Классификация символов (LC\_CTYPE), которая будет применяться в новой базе данных. Этот параметр определяет принадлежность символов категориям, например: строчные, заглавные, цифры и т. п.

***табл\_пространство***

Имя табличного пространства, связываемого с новой базой данных, или DEFAULT для использования табличного пространства шаблона. Это табличное пространство будет использоваться по умолчанию для объектов, создаваемых в этой базе.

***предел\_подключений***

Максимальное количество одновременных подключений к этой базе данных. Значение -1 (по умолчанию) снимает ограничение.

***это\_шаблон***

Если true, базу данных сможет клонировать любой пользователь с правами CREATEDB; в противном случае (по умолчанию), клонировать эту базу смогут только суперпользователи и её владелец.

При создании базы данных автоматически создается схема public.

### 2.1.1. Создание БД из командной строки

Сначала откройте командную строку в Windows или терминал в Unix-подобных системах и подключитесь к серверу PostgreSQL:

> psql.exe -U postgres**[[4]](#footnote-4)**

Password for user postgres:

psql (16.0, server 14.12 (Ubuntu 14.12-0ubuntu0.22.04.1))

WARNING: Console code page (866) differs from Windows code page (1251)

8-bit characters might not work correctly. See psql reference

page "Notes for Windows users" for details.

SSL connection (protocol: TLSv1.3, cipher: TLS\_AES\_256\_GCM\_SHA384, compression: off)

Type "help" for help.

postgres=# \! chcp 1251

Текущая кодовая страница: 1251

Посмотрим какие БД есть на сервере (рис. 25):

postgres=# \l

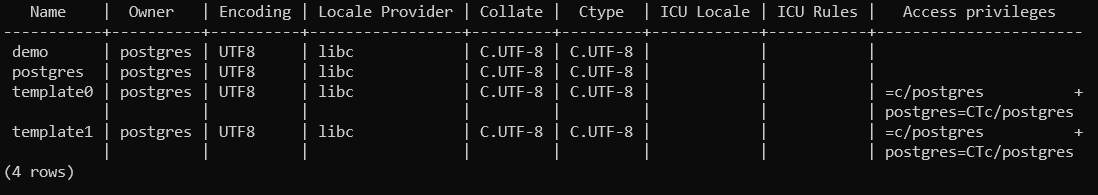


Рисунок 25 – Результат

Создайте базу данных с параметрами по умолчанию.

postgres=# CREATE DATABASE test\_db;

CREATE DATABASE

Проверим (рис. 26):

postgres=# \l

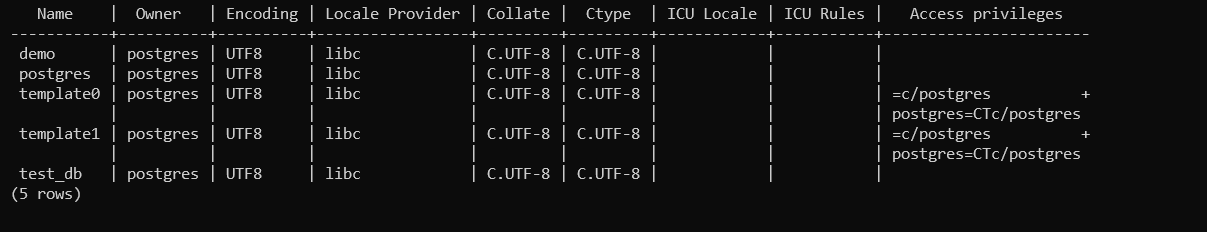


Рисунок 26 – Результат

Создать базу данных с поддержкой русской локали:

CREATE DATABASE test\_db2

WITH OWNER postgres

ENCODING 'UTF8'

LC\_COLLATE = 'ru\_RU.utf8'

LC\_CTYPE = 'ru\_RU.utf8'

TEMPLATE = template0;

В этом примере предложение TEMPLATE=template0 необходимо, только если указанная локаль отличается от локали в template1. (В противном случае явное указание локали является избыточным.)

Проверим (рис. 27):

postgres=# \l

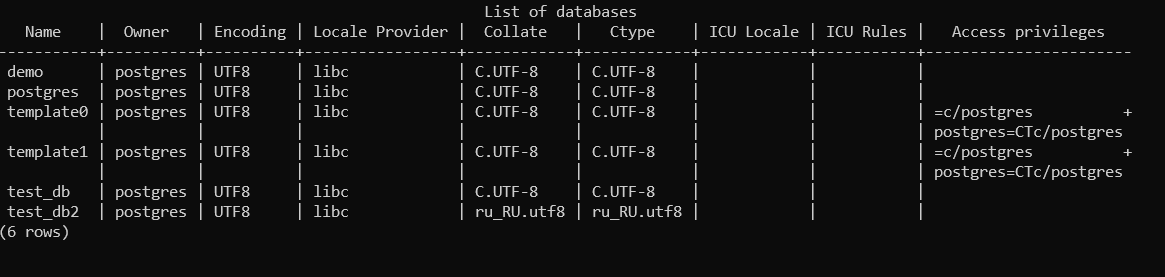


Рисунок 27 – Результат

### 2.1.2. Создание БД с помощью pgAdmin

Щелкните правой кнопкой по узлу Databeses и выберите пункт меню *Create/ Database...* (рис. 28):

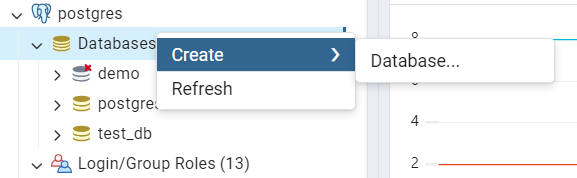


Рисунок 28 – Контекстное меню

Появится диалоговое окно для ввода подробной информации о новой базе данных.

В этом примере мы создаем новую базу данных с именем test\_db3 и владельцем postgres (рис. 29):

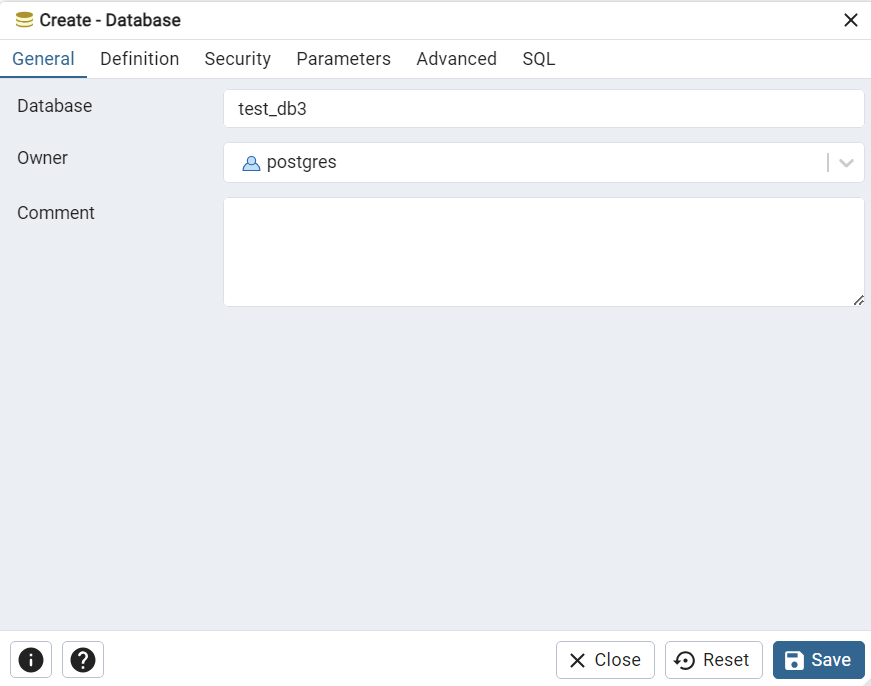


Рисунок 29 – Диалог создания БД

Выберите вкладку «Definition» (рис. 30), чтобы установить свойства базы данных:

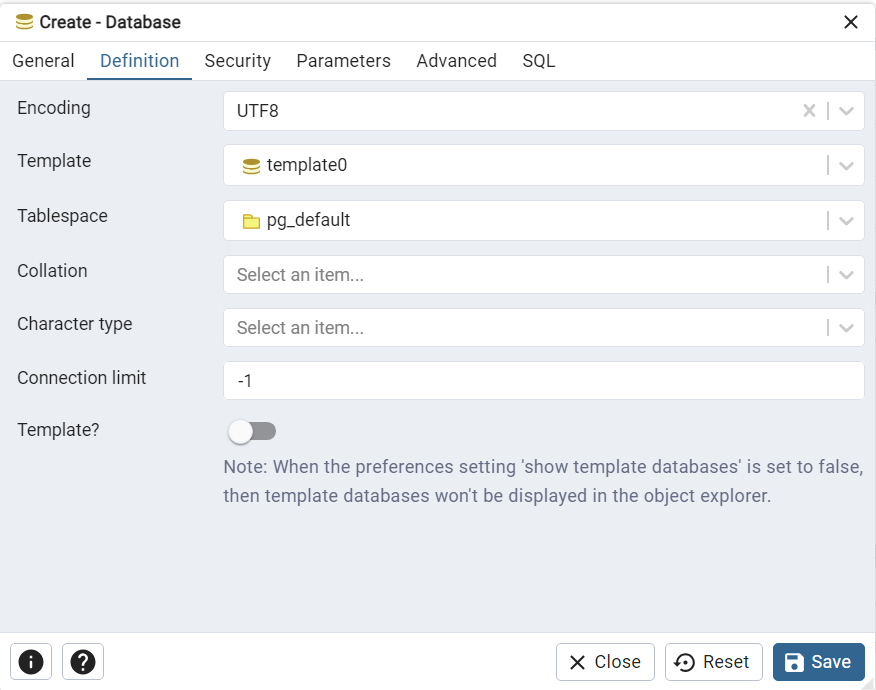


Рисунок 30 – Вкладка «определение»

На вкладке «Definition» вы можете выбрать кодировку, шаблон, табличное пространство, параметры сортировки, тип символов и ограничение на количество соединений.

Вкладка «Security» позволяет определять метки безопасности и назначать привилегии. Вкладка «Privileges» позволяет назначать привилегии роли.

Щелкните вкладку SQL (рис. 31), чтобы просмотреть сгенерированный оператор SQL, который будет выполнен.

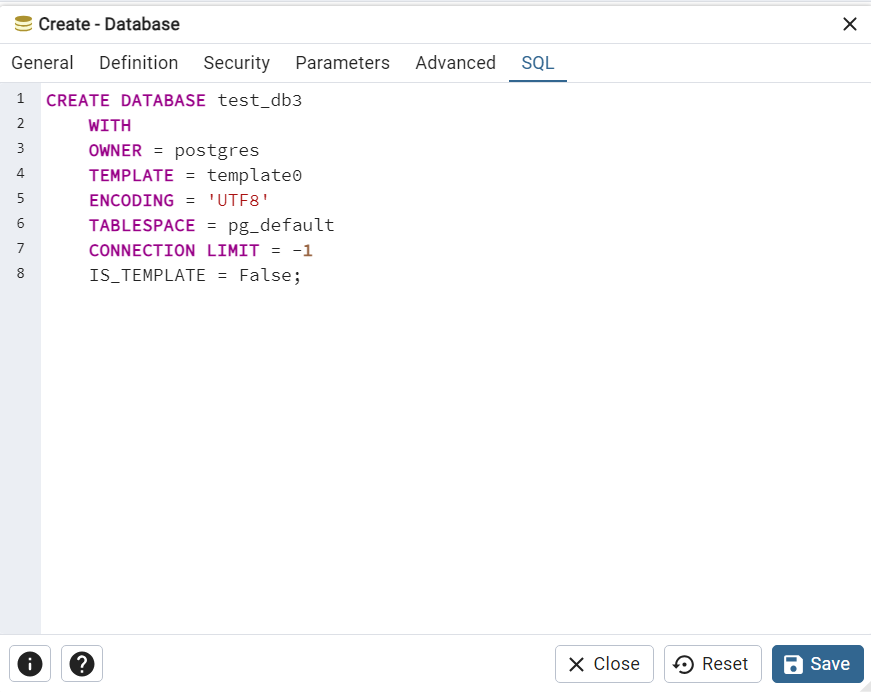


Рисунок 31 – Вкладка «SQL»

## 2.2. Удаление базы данных

Базу данных удаляют командой:

DROP DATABASE *имя*;

Лишь владелец базы данных или суперпользователь могут удалить базу. При удалении также удаляются все её объекты. Удаление базы данных — это необратимая операция.

Невозможно выполнить команду DROP DATABASE пока существует хоть одно подключение к заданной базе. Однако можно подключиться к любой другой, в том числе и template1. template1 может быть единственной возможностью при удалении последней пользовательской базы данных кластера.

2.2.1. Удаление БД из командной строки

Посмотрим, как в PostgreSQL выполнить удаление базы данных. Подключимся к PostgreSQL через терминал:

> psql.exe -U postgres

Password for user postgres:

psql (16.0, server 14.12 (Ubuntu 14.12-0ubuntu0.22.04.1))

WARNING: Console code page (866) differs from Windows code page (1251)

8-bit characters might not work correctly. See psql reference

page "Notes for Windows users" for details.

SSL connection (protocol: TLSv1.3, cipher: TLS\_AES\_256\_GCM\_SHA384, compression: off)

Type "help" for help.

demo=# \! chcp 1251

Текущая кодовая страница: 1251

Выполните запрос:

DROP DATABASE test\_db;

Чтобы убедиться, что новая БД успешно создана, выведите список всех баз данных (рис. 32):

postgres=# \l

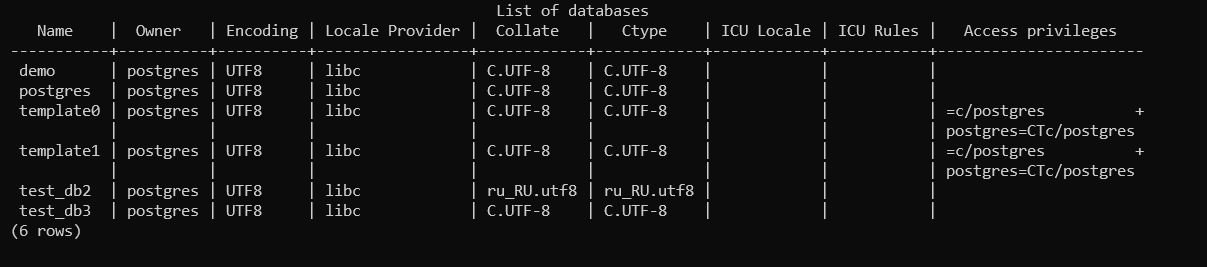


Рисунок 32 – Результат

WITH (FORCE)

Опция WITH (FORCE) нужна для «силового» удаления базы данных, которая используется в данный момент. WITH (FORCE) принудительно закрывает сеанс и удаляет БД.

DROP DATABASE *имя* WITH (FORCE);

2.2.2. Удаление БД с помощью pgAdmin

Удалим test\_db3. Щелкните правой кнопкой по узлу test\_db3 и выберите пункт меню *Delete* (рис. 33):

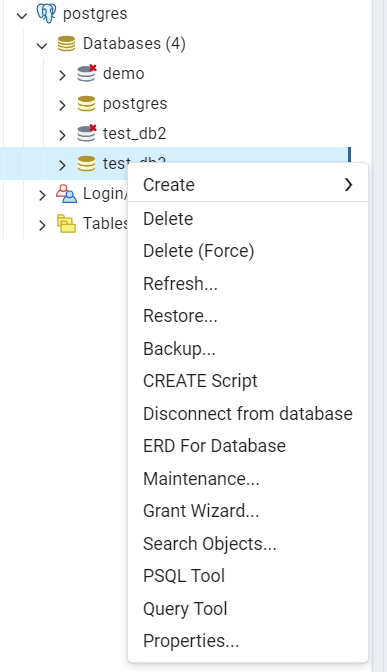


Рисунок 33 – Контекстное меню

После подтверждения test\_db3 будет удалена.

## 2.3. Создание схемы

Оператор CREATE SCHEMA позволяет создать новую схему в текущей базе данных:

CREATE SCHEMA ***имя\_схемы***;

Имя схемы должно быть уникальным в пределах текущей базы данных. Для выполнения инструкции CREATE SCHEMA у вас должны быть CREATE права доступа к текущей базе данных.

2.3.1. Создание схемы из командной строки

Создадим схему LR в БД test\_db2. Подключимся к БД test\_db2   
(рис. 34):

postgres=# \c test\_db2



Рисунок 34 – Сообщение о подключении к test\_db2

Перед созданием новой схемы отобразим список схем test\_db2 (рис. 35):

test\_db2=# \dn

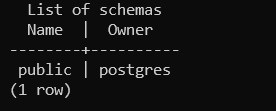


Рисунок 35 – Список схем

Создадим новую схему:

test\_db2=# CREATE SCHEMA LR;

Убедимся, что схема создана (рис. 36):

test\_db2=# \dn

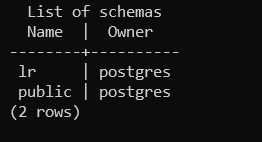


Рисунок 36 – Список схем

2.3.2. Создание схемы с помощью pgAdmin

Для создания схемы щелкните правой кнопкой по узлу Schemas нужной БД и выберите команду меню *Create/Schema…*. Далее откроется диалог, аналогичный при создании БД для ввода информации (рис. 37):

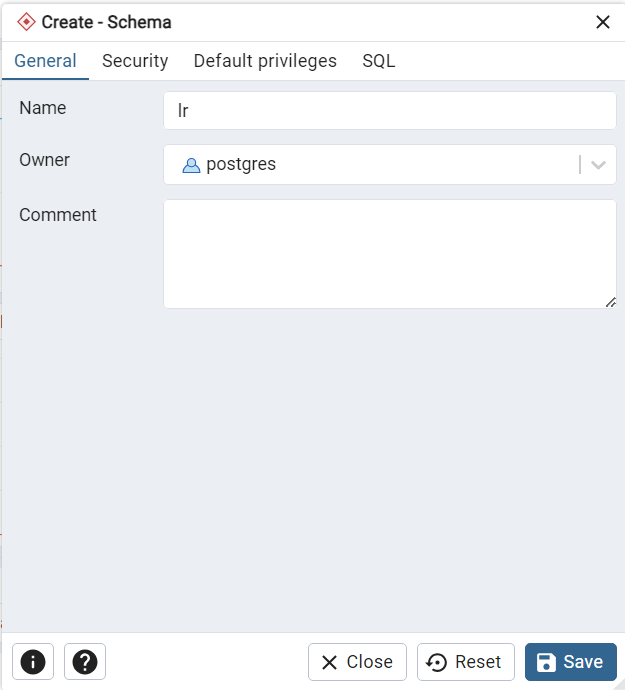


Рисунок 37 – Диалог создания схемы

## 2.4. Создание таблиц

Для создания таблиц нужно вспомнить основные типы данных в SQL, которые были рассмотрены в лабораторной работе №1.

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. Упрощенный синтаксис создания таблицы выглядит следующим образом:

CREATE TABLE название\_таблицы

(

имя\_столбца1 тип\_данных [ограничения\_целостности\_столбца],

имя\_столбца2 тип\_данных [ограничения\_целостности\_столбца],

…

имя\_столбцаN тип\_данных [ограничения\_целостности\_столбца],

[ограничения\_целостности\_таблицы],

[первичный\_ключ],

[внешний\_ключ],

);

Сначала укажите после ключевых слов имя таблицы, которую вы хотите создать CREATE TABLE. Имя таблицы должно быть уникальным в схеме. Если вы создадите таблицу с именем, которое уже существует, вы получите сообщение об ошибке.

Схема — это именованная коллекция объектов базы данных, включая таблицы. Если вы создаете таблицу без схемы (***схема.таблица***), по умолчанию она становится общедоступной, в схеме public.

После названия таблицы в скобках перечисляется спецификация для всех столбцов. Причем для каждого столбца надо указывать название и тип данных, который он будет представлять. Тип данных определяет, какие данные (числа, строки и т.д.) может содержать столбец.

Ограничение целостности столбца определяет правило, которое применяется к данным внутри столбца для обеспечения целостности данных. Ограничения столбца включают первичный ключ, внешний ключ, not null, unique, check и default.

Например, NOT NULL ограничение гарантирует, что значения в столбце не могут быть NULL.

Наконец, укажите ограничения для таблицы, включая ограничения первичного ключа, внешнего ключа и проверки.

Ограничение целостности таблицы — это правило, которое применяется к данным внутри таблицы для обеспечения целостности данных.

Обратите внимание, что некоторые ограничения столбцов могут быть определены как ограничения таблицы, такие как ограничения первичного ключа, внешнего ключа, уникальности и проверки.

PostgreSQL включает следующие ограничения столбцов:

* NOT NULL – гарантирует, что значения в столбце не могут быть равными NULL.
* UNIQUE – гарантирует уникальность значений в столбце во всех строках одной таблицы.
* PRIMARY KEY – столбец первичного ключа однозначно идентифицирует строки в таблице. Таблица может иметь один и только один первичный ключ. Ограничение первичного ключа позволяет определить первичный ключ таблицы.
* CHECK – гарантирует, что данные удовлетворяют логическому выражению. Например, значение в столбце цены должно быть нулевым или положительным.
* FOREIGN KEY — гарантирует, что значения в столбце или группе столбцов таблицы существуют в столбце или группе столбцов другой таблицы. В отличие от первичного ключа, таблица может иметь множество внешних ключей.

Ограничения таблиц аналогичны ограничениям столбцов, за исключением того, что в ограничение таблицы можно включить более одного столбца.

2.4.1. Внешний ключ

Внешний ключ устанавливает связь между данными в двух таблицах путем ссылки на первичный ключ или ограничение уникальности связанной таблицы.

Таблица, содержащая внешний ключ, называется ссылающейся таблицей или дочерней таблицей. И наоборот, таблица, на которую ссылается внешний ключ, называется ссылочной таблицей или родительской таблицей.

Основная цель внешних ключей — поддерживать ссылочную целостность в реляционной базе данных, гарантируя, что отношения между родительской и дочерней таблицами действительны. Например, внешний ключ предотвращает вставку значений, которые не имеют соответствующих значений в ссылочной таблице. Кроме того, внешний ключ обеспечивает согласованность, автоматически обновляя или удаляя связанные строки в дочерней таблице, когда происходят изменения в родительской таблице.

Таблица может иметь несколько внешних ключей в зависимости от ее отношений с другими таблицами.

Чтобы определить внешний ключ, вы можете использовать ограничение внешнего ключа.

Ниже показан синтаксис ограничения внешнего ключа:

[CONSTRAINT fk\_name]

FOREIGN KEY(fk\_columns)

REFERENCES parent\_table(parent\_key\_columns)

[ON DELETE delete\_action]

[ON UPDATE update\_action]

В этом синтаксисе:

* Сначала укажите имя ограничения внешнего ключа после CONSTRAINT ключевого слова. Это CONSTRAINT предложение является необязательным. Если вы его опустите, PostgreSQL присвоит автоматически сгенерированное имя.
* Во-вторых, укажите один или несколько столбцов внешнего ключа в круглых скобках после FOREIGN KEY ключевых слов.
* В-третьих, укажите родительскую таблицу и столбцы родительского ключа, на которые ссылаются столбцы внешнего ключа в REFERENCES предложении.
* Наконец, укажите желаемые действия по удалению и обновлению в предложениях ON DELETE и ON UPDATE.

Действия удаления и обновления определяют поведение при удалении и обновлении первичного ключа в родительской таблице.

Поскольку первичный ключ редко обновляется, update\_action на практике он используется нечасто. Рассмотрим delete\_action.

PostgreSQL поддерживает следующие действия:

* SET NULL – при удалении записи родительской таблицы устанавливаются NULL столбцы внешнего ключа в ссылающихся строках дочерней таблицы.
* SET DEFAULT – при удалении записи родительской таблицы устанавливает значение по умолчанию для столбца внешнего ключа ссылающихся строк.
* RESTRICT – Действие RESTRICT аналогично NO ACTION. Разница возникает только тогда, когда вы определяете ограничение внешнего ключа как DEFERRABLE режим INITIALLY DEFERRED или INITIALLY IMMEDIATE.
* NO ACTION – при удалении записи родительской таблицы никаких действий не предпринимается. И если имеется дочерняя запись, ссылающаяся на удаляемую строку, то PostgreSQL выдаст ошибку, так нарушается ограничение целостности.
* CASCADE - при удалении записи родительской таблицы удаляются ссылающиеся строки дочерней таблицы.

Добавление внешнего ключа в существующую таблицу

Чтобы добавить ограничение внешнего ключа в существующую таблицу, вы используете следующую форму оператора ALTER TABLE:

ALTER TABLE child\_table

ADD CONSTRAINT constraint\_name

FOREIGN KEY (fk\_columns)

REFERENCES parent\_table (parent\_key\_columns);

Если необходимо скорректировать существующий внешний ключ, например, исправить ON DELETE NO ACTION на ON DELETE CASCADE необходимо

выполнить следующие шаги:

* удалить существующее ограничение внешнего ключа:

ALTER TABLE child\_table

DROP CONSTRAINT constraint\_fkey;

* добавьте новое ограничение внешнего ключа с ON DELETE CASCADE

ALTER TABLE child\_table

ADD CONSTRAINT constraint\_fk

FOREIGN KEY (fk\_columns)

REFERENCES parent\_table(parent\_key\_columns)

ON DELETE CASCADE;

2.4.2. Ограничение CHECK

В PostgreSQL CHECK ограничение гарантирует, что значения в столбце или группе столбцов соответствуют определенному условию.

Ограничение проверки позволяет обеспечить соблюдение правил целостности данных на уровне базы данных. Проверочное ограничение использует логическое выражение для оценки значений, гарантируя, что в таблицу вставляются или обновляются только допустимые данные.

Обычно проверочное ограничение создается при создании таблицы с помощью CREATE TABLE оператора:

CREATE TABLE table\_name

(

column1 datatype,

...,

CONSTRAINT constraint\_name CHECK(condition)

);

Если CHECK ограничение включает только один столбец, вы можете определить его как ограничение столбца следующим образом:

CREATE TABLE table\_name

(

column1 datatype,

column2 datatype CHECK(condition),

...,

);

Добавление ограничений CHECK в таблицы

Чтобы добавить CHECK ограничение к существующей таблице, вы используете оператор ALTER TABLE ... ADD CONSTRAINT:

ALTER TABLE table\_name

ADD CONSTRAINT constraint\_name CHECK (condition);

Удаление ограничений CHECK

Чтобы удалить ограничение CHECK, вы используете ALTER TABLE ... DROP CONSTRAINT оператор:

ALTER TABLE table\_name

DROP CONSTRAINT constraint\_name;

2.4.3. Ограничение UNIQUE

Иногда требуется убедиться, что значения, хранящиеся в столбце или группе столбцов, являются уникальными во всей таблице, например адреса электронной почты или имена пользователей.

PostgreSQL предоставляет вам ограничение UNIQUE, которое корректно поддерживает уникальность данных.

Когда ограничение UNIQUE установлено, каждый раз, когда вы вставляете новую строку, он проверяет, есть ли уже значение в таблице. Он отклоняет изменение и выдает ошибку, если значение уже существует. Тот же процесс выполняется при изменении (UPDATE) существующих данных.

При добавлении ограничения UNIQUE к столбцу или группе столбцов PostgreSQL автоматически создаст уникальный индекс для столбца или группы столбцов.

Следующий оператор создает новую таблицу с именем person и ограничением UNIQUE для email столбца.

CREATE TABLE person

(

id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

name varchar (50),

email varchar (50) UNIQUE

);

2.4.4. Ограничение NOT NULL

В мире баз данных NULL представляет неизвестную или отсутствующую информацию. NULL — это не то же самое, что пустая строка или число ноль.

Предположим, вам нужно вставить адрес электронной почты абонента в таблицу. Вы можете запросить адрес электронной почты. Однако, если вы не знаете, есть ли у абонента адрес электронной почты или нет, вы следует вставить NULL в столбец адреса электронной почты. В этом случае NULL указывает на то, что адрес электронной почты неизвестен на момент записи.

NULL — очень особенный. Он не равен ничему, даже самому себе. Выражение NULL = NULL возвращает NULL, поскольку имеет смысл, что два неизвестных значения не должны быть равны.

Чтобы проверить, является ли значение NULL или нет, вы используете   
IS NULL логический оператор. Например, следующее выражение возвращает true, если значение в адресе электронной почты равно NULL:

email\_address IS NULL

Чтобы контролировать, может ли столбец принимать значение NULL, используйте ограничение NOT NULL:

CREATE TABLE table\_name

(

...

column\_name data\_type NOT NULL,

...

);

Если столбец имеет ограничение NOT NULL, любая попытка вставить или изменить на значение NULL в этом столбце приведет к ошибке.

Чтобы добавить ограничение NOT NULL к столбцу существующей таблицы, используйте следующую форму оператора ALTER TABLE:

ALTER TABLE table\_name

ALTER COLUMN column\_name\_1 SET NOT NULL,

ALTER COLUMN column\_name\_2 SET NOT NULL,

...;

Вместо ограничения NOT NULL, можно использовать ограничение CHECK:

CHECK(column\_name IS NOT NULL)

Это полезно, поскольку иногда может потребоваться, чтобы два столбца одновременно не были пустыми:

CREATE TABLE users

(

id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

name varchar(50),

email varchar(50),

CONSTRAINT name\_email\_notnull

CHECK

(

NOT (

IS NULL OR username = ''

AND

email IS NULL OR email = ''

)

)

);

Следующая команда успешно вставит строки:

INSERT INTO users (name, email)

VALUES ('name1', NULL),

(NULL, 'email1@example.com'),

('name2', 'email2@example.com'),

('name3', '');

2.4.5. Ограничение DEFAULT

При создании таблицы вы можете определить значение по умолчанию для столбца в таблице, используя ограничение DEFAULT. Вот базовый синтаксис:

CREATE TABLE table\_name

(

column1 type,

column2 type DEFAULT default\_value,

column3 type,

...

);

В этом синтаксисе column2 будет получаться default\_value при вставке новой строки в table\_name, если значение для столбца не задано.

Если вы хотите указать значение по умолчанию для столбца существующей таблицы, вы можете использовать следующий оператор   
ALTER TABLE:

ALTER TABLE table\_name

ALTER COLUMN column2 SET DEFAULT default\_value;

Чтобы позже удалить значение по умолчанию, вы также можете использовать оператор ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT:

ALTER TABLE table\_name

ALTER COLUMN column2 DROP DEFAULT;

2.4.6. Создание таблицы из командной строки

Мы создадим новую таблицу под названием accounts в схеме lr БД test\_db2. В accounts таблице имеются следующие столбцы:

user\_id – primary key

username – unique and not null

password – not null

email – unique and not null

created\_at – not null

last\_login – null

Команда для создания таблицы accounts выглядит следующим образом:

CREATE TABLE lr.accounts

(

user\_id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

username varchar (50) UNIQUE NOT NULL,

password varchar (50) NOT NULL,

email varchar (255) UNIQUE NOT NULL,

created\_at timestamp NOT NULL,

last\_login timestamp

);

Сначала откройте командную строку в Windows или терминал в Unix-подобных системах и подключитесь к PostgreSQL:

$ sudo su postgres

Вам будет предложено ввести пароль.

[sudo] password for user:

Подключимся к БД test\_db2:

$ psql test\_db2 -U postgres

Введите оператор CREATE TABLE и нажмите Enter (рис. 38):

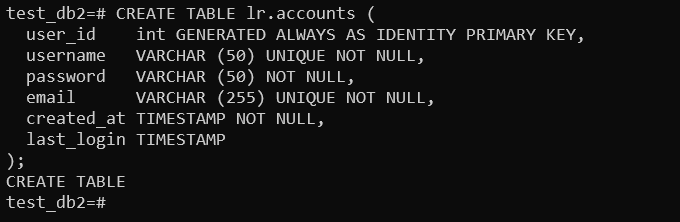


Рисунок 38 – SQL-скрипт создания таблицы lr.accounts

Вывод CREATE TABLE показывает, что таблица создана.

Для просмотра таблицы accounts можно использовать \d команду (рис. 39):

test\_db2=# \d lr.accounts

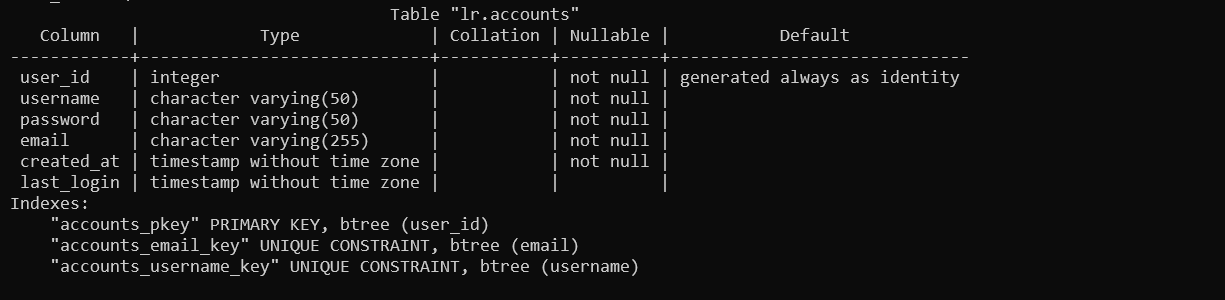


Рисунок 39 – Поля таблицы lr.accounts

2.4.7. Создание таблицы с помощью pgAdmin

Вы можете использовать pgAdmin для создания таблицы вручную или с помощью SQL-запросов. Чтобы создать таблицу с помощью запросов Postgres, откройте pgAdmin (рис. 40), выберите базу данных, запустите **Query Tool** и, наконец, выполните команду CREATE TABLE. Создадим таблицу role в схеме lr БД test\_db2:

CREATE TABLE lr.role

(

role\_id int GENERATED ALWAYS AS IDENTITY PRIMARY KEY,

role\_name varchar (255) UNIQUE NOT NULL

);

Выполним этот скрипт, и после обновления узла Tables увидим таблицу role.

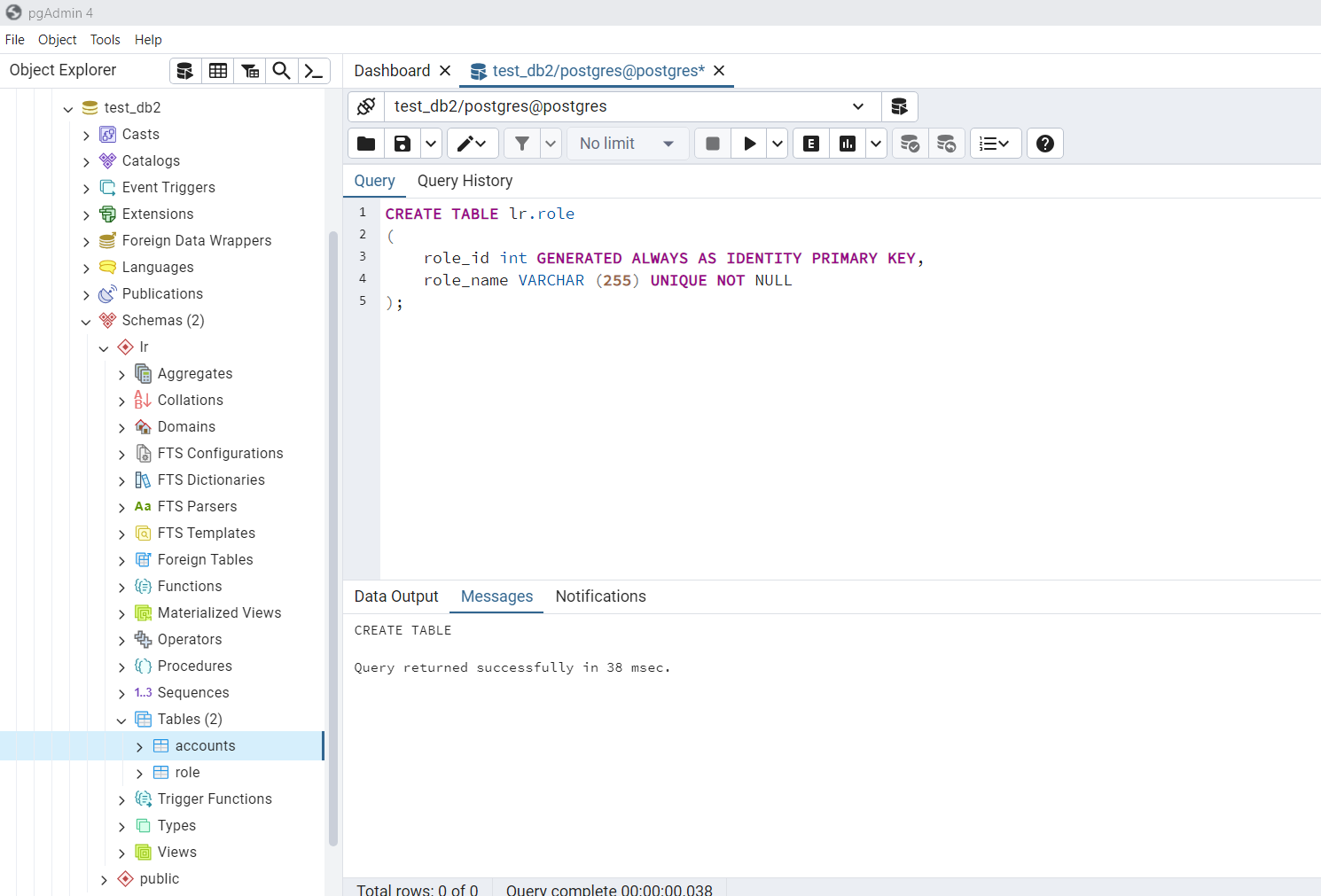


Рисунок 40 – Создание таблицы lr.role

Создадим еще одну таблицу account\_role, используя pgAdmin, состоящую из трех полей:

* user\_id – id пользователя, информация о котором находится в таблице accounts;
* role\_id – id роли (права) выданное пользователю, описание роли находится в таблице role;
* grant\_date – дата получения роли.

Таким образом, имеем три взаимосвязанные таблицы. В таблице accounts содержатся все пользователи некой информационной системы, которым выдаются роли (права). Пользователь может иметь несколько ролей.

Список ролей содержится в таблице role. В итоге, в таблице account\_role содержится информация о пользователях и их правах (рис. 41):

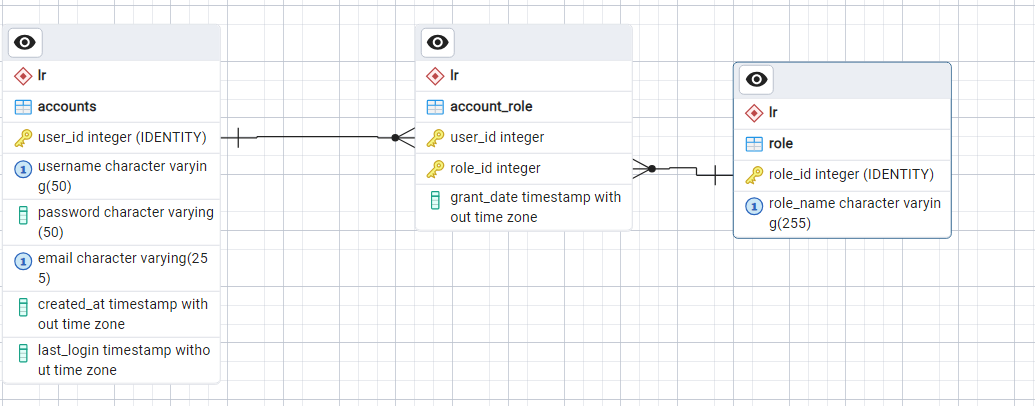


Рисунок 41 – ERD-модель test\_db2

Поскольку пользователю нет необходимости выделять одну и ту же роль, то получаем первичный ключ из пары (user\_id, role\_id), однозначно определяющий строку. Кроме того, имеем два внешних ключа, это user\_id и role\_id, связывающие соответствующие таблицы. Какие атрибуты внешнего ключа следует использовать при удалении строки из таблиц role и acounts?

Роли, как правило не меняются в течении длительного времени, то для внешнего ключа подходит опция NO ACTION. В этом случае PostgreSQL не даст удалить используемую роль.

Пользователей чаще приходится удалять, поэтому целесообразно использовать CASCADE, тем самым позволяя PostgreSQL удалять строки из таблицы account\_role, относящиеся к этому пользователю. В противном случае, пришлось самим программно или вручную удалять строки из account\_role (в PostgreSQL аналогичное отношение имеет атрибут NO ACTION).

Нажмем правой клавишей на узел Tables и *Create/Table…*. Откроется диалог (рис. 42):

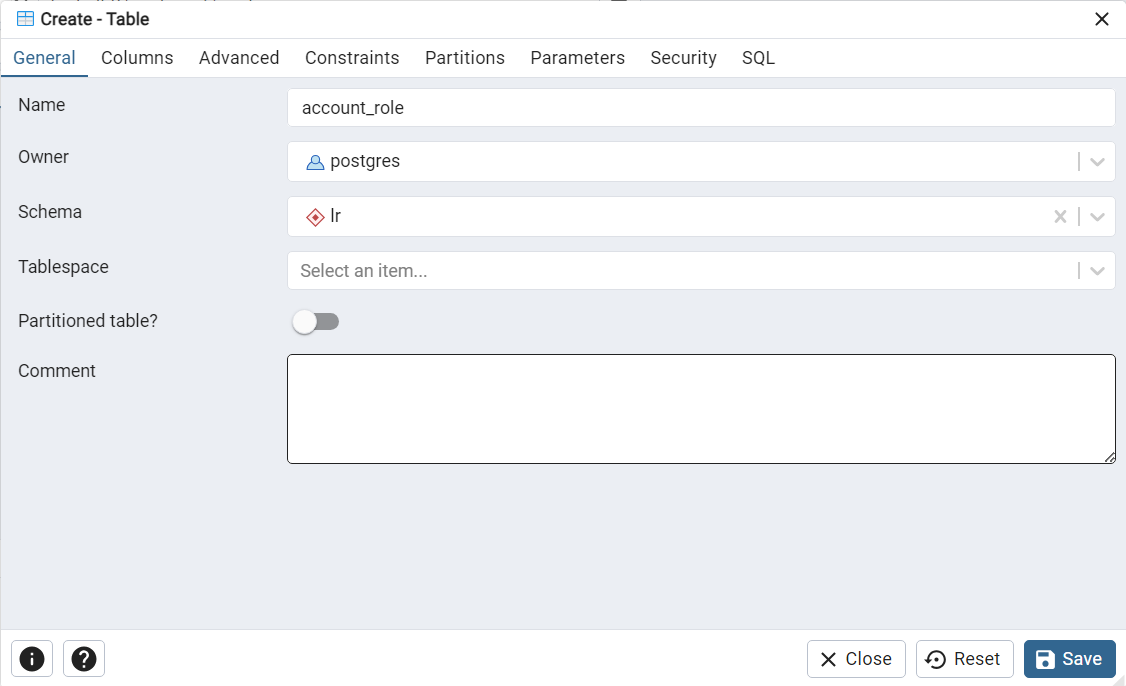


Рисунок 42 – Диалог создания таблицы

В поле *Name* вводим имя таблицы. Открываем вкладку *Columns* (рис. 43):

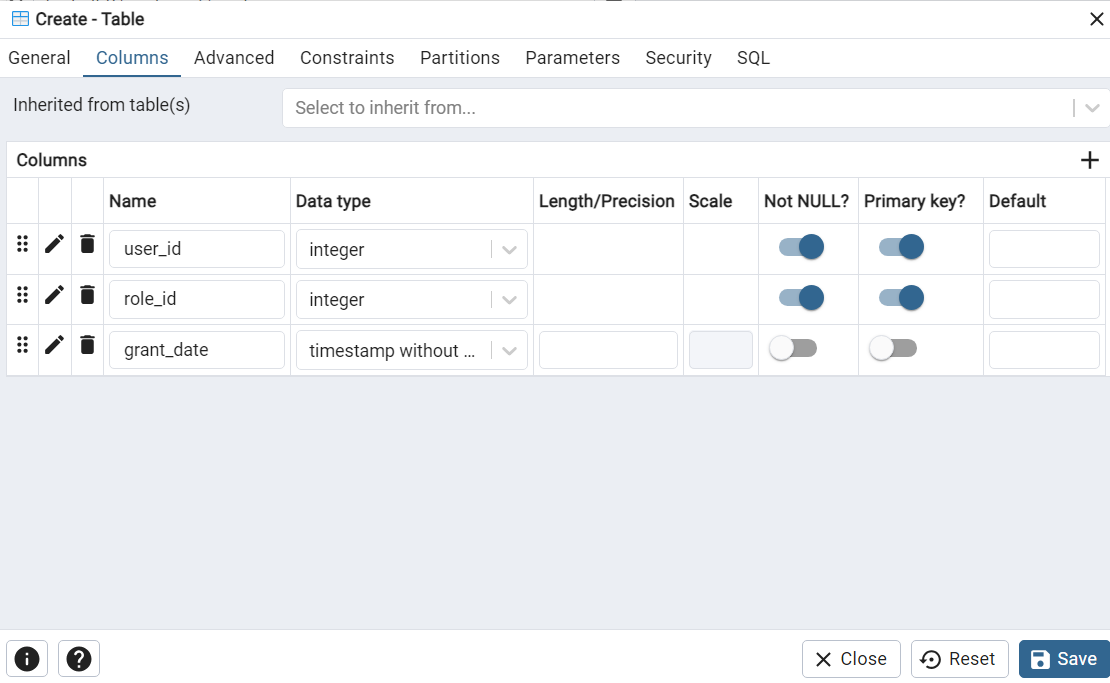


Рисунок 43 – Вкладка «колонки»

Заносим поля, выбирая их тип из выпадающего списка. Отмечаем первичный ключ.

Далее переходим на вкладку *Constraints*. Во вкладке *Constraints* имеются подвкладки, *General, Definition, Columns, Action*, используя их вводим (выбираем таблицы поля из выпадающих списков) два внешних ключа (рис. 44):

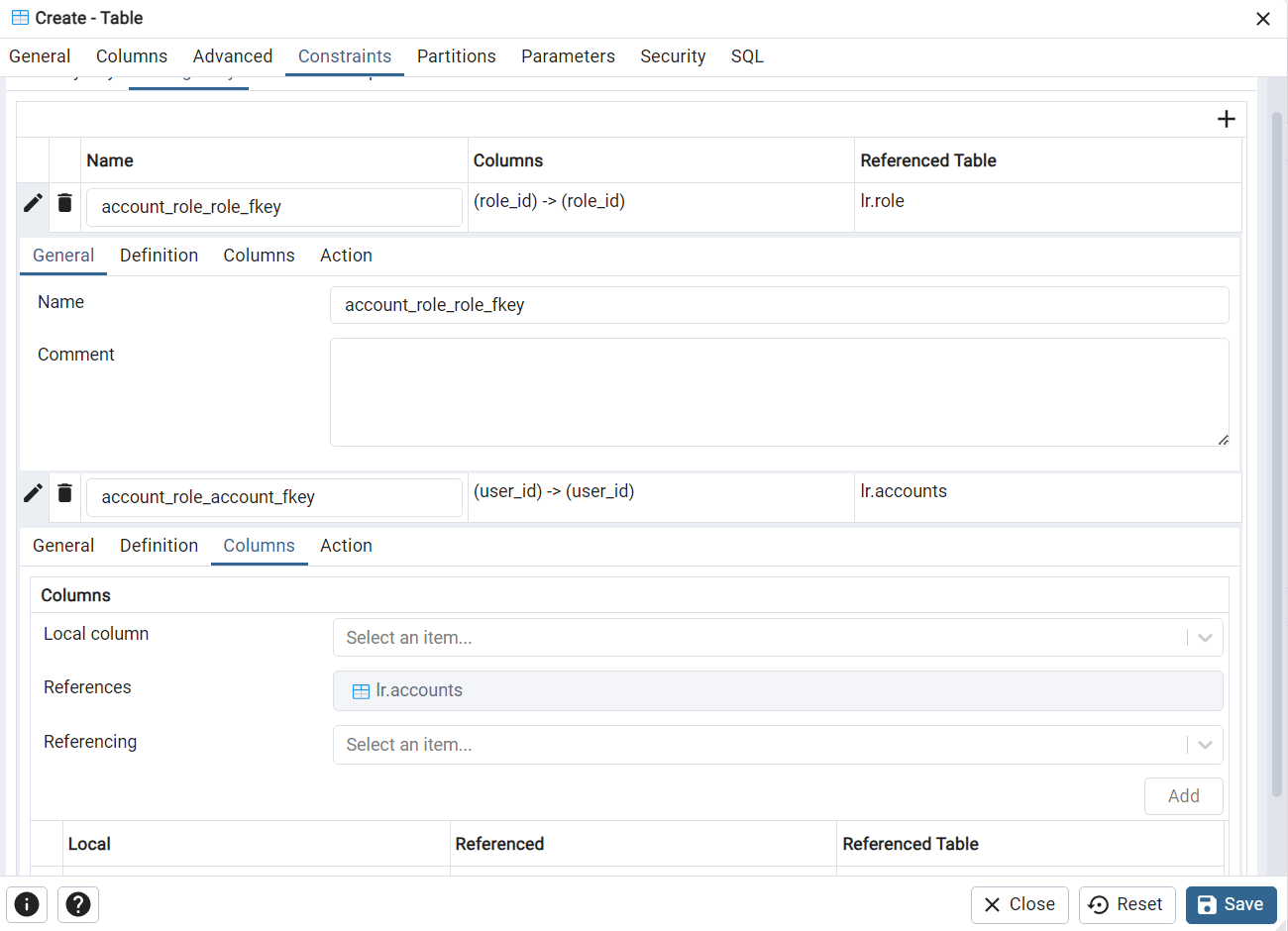


Рисунок 44 – Вкладка «ограничения»

Для проверки откроем вкладку *SQL* (рис. 45), чтобы увидеть скрипт, который будет использоваться при создании таблицы. Все устраивает. Нажимаем *Seve*.

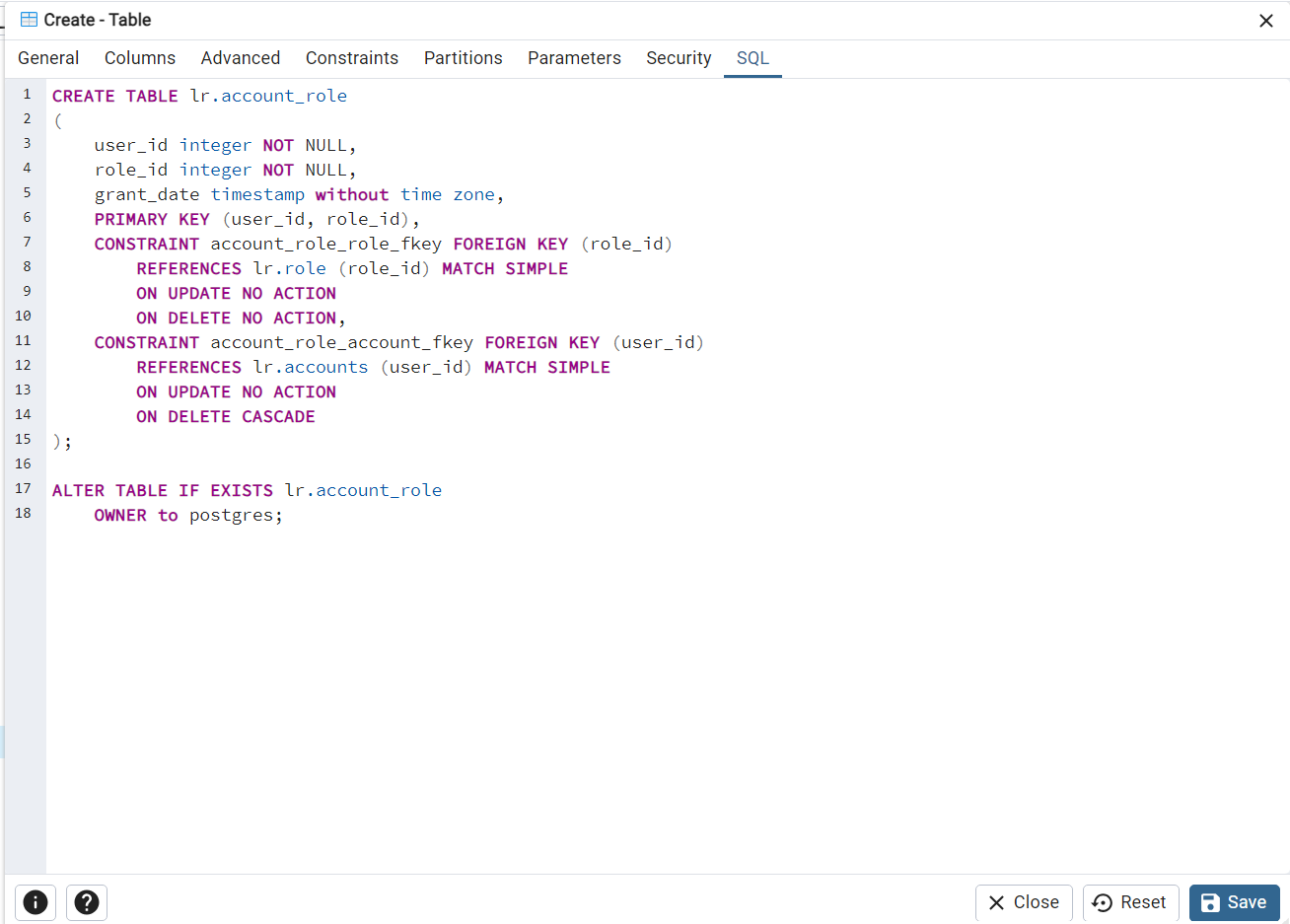


Рисунок 45 – Вкладка «SQL»

Таблица создана.

Посмотрим, как повлияло создание таблицы account\_role, например на таблицу accounts (рис. 46):

test\_db2=# \d lr.accounts

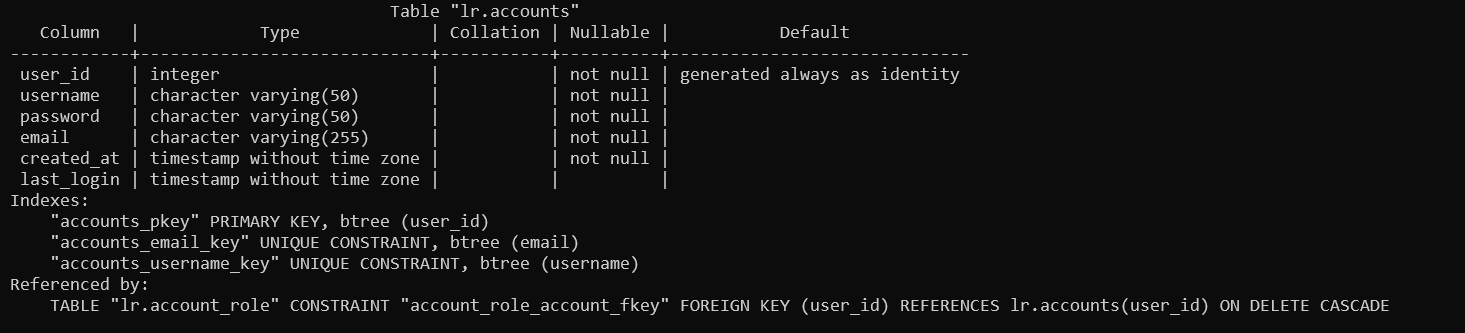


Рисунок 46 – Поля таблицы lr.accounts

Добавилась строка, информирующая, что с таблицей lr.account\_role организован внешний ключ с соответствующими параметрами:

Referenced by:

TABLE "lr.account\_role" CONSTRAINT "account\_role\_account\_fkey" FOREIGN KEY (user\_id) REFERENCES lr.accounts(user\_id) ON DELETE CASCADE

## 2.5. Изменение таблицы

### 2.5.1. Добавление столбца

Чтобы добавить новый столбец в таблицу, вы используете ALTER TABLE ADD COLUMN оператор:

ALTER TABLE table\_name

ADD COLUMN column\_name datatype column\_constraint;

### 2.5.2. Удаление столбца

Чтобы удалить столбец из таблицы, вы используете ALTER TABLE DROP COLUMN оператор:

ALTER TABLE table\_name

DROP COLUMN column\_name;

### 2.5.3. Переименование столбца

Чтобы переименовать столбец, вы используете ALTER TABLE RENAME COLUMN TO оператор:

ALTER TABLE table\_name

RENAME COLUMN column\_name TO new\_column\_name;

### 2.5.4. Установите значение по умолчанию для столбца

Чтобы изменить значение столбца по умолчанию, вы используете   
ALTER TABLE ALTER COLUMN SET DEFAULT или DROP DEFAULT:

ALTER TABLE table\_name

ALTER COLUMN column\_name

[SET DEFAULT value | DROP DEFAULT];

Чтобы изменить NOT NULL ограничение, вы используете ALTER TABLE ALTER COLUMN оператор:

ALTER TABLE table\_name

ALTER COLUMN column\_name

[SET NOT NULL| DROP NOT NULL];

### 2.5.5. Переименование таблицы

Чтобы переименовать таблицу, вы используете ALTER TABLE RENAME TO оператор:

ALTER TABLE table\_name

RENAME TO new\_table\_name;

## 2.6. Индексы

Индексы позволяют повысить производительность базы данных. PostgreSQL поддерживает различные типы индексов. Мы ограничимся рассмотрением только индексов на основе B-дерева. Индекс — специальная структура данных, которая связана с таблицей и создается на основе данных, содержащихся в ней. Основная цель создания индексов — повышение производительности функционирования базы данных.

Строки в таблицах хранятся в неупорядоченном виде. При выполнении операций выборки, обновления и удаления СУБД должна отыскать нужные строки. Для ускорения этого поиска и создается индекс. В принципе он организован таким образом: на основе данных, содержащихся в конкретной строке таблицы, формируется значение элемента (записи) индекса, соответствующего этой строке. Для поддержания соответствия между элементом индекса и строкой таблицы в каждый элемент помещается указатель на строку. Индекс является упорядоченной структурой. Элементы (записи) в нем хранятся в отсортированном виде, что значительно ускоряет поиск данных в индексе. После отыскания в нем требуемой записи СУБД переходит к соответствующей строке таблицы по прямой ссылке. Записи индекса могут формироваться на основе значений одного или нескольких полей соответствующих строк таблицы. Значения этих полей могут комбинироваться и преобразовываться различными способами. Все это определяет разработчик базы данных при создании индекса.

Для того чтобы увидеть индексы, созданные для данной таблицы, нужно воспользоваться командой \d (описание таблицы) утилиты psql, например, посмотрим описание таблицы bookings.flights (рис. 47):

demo=# \d bookings.flights

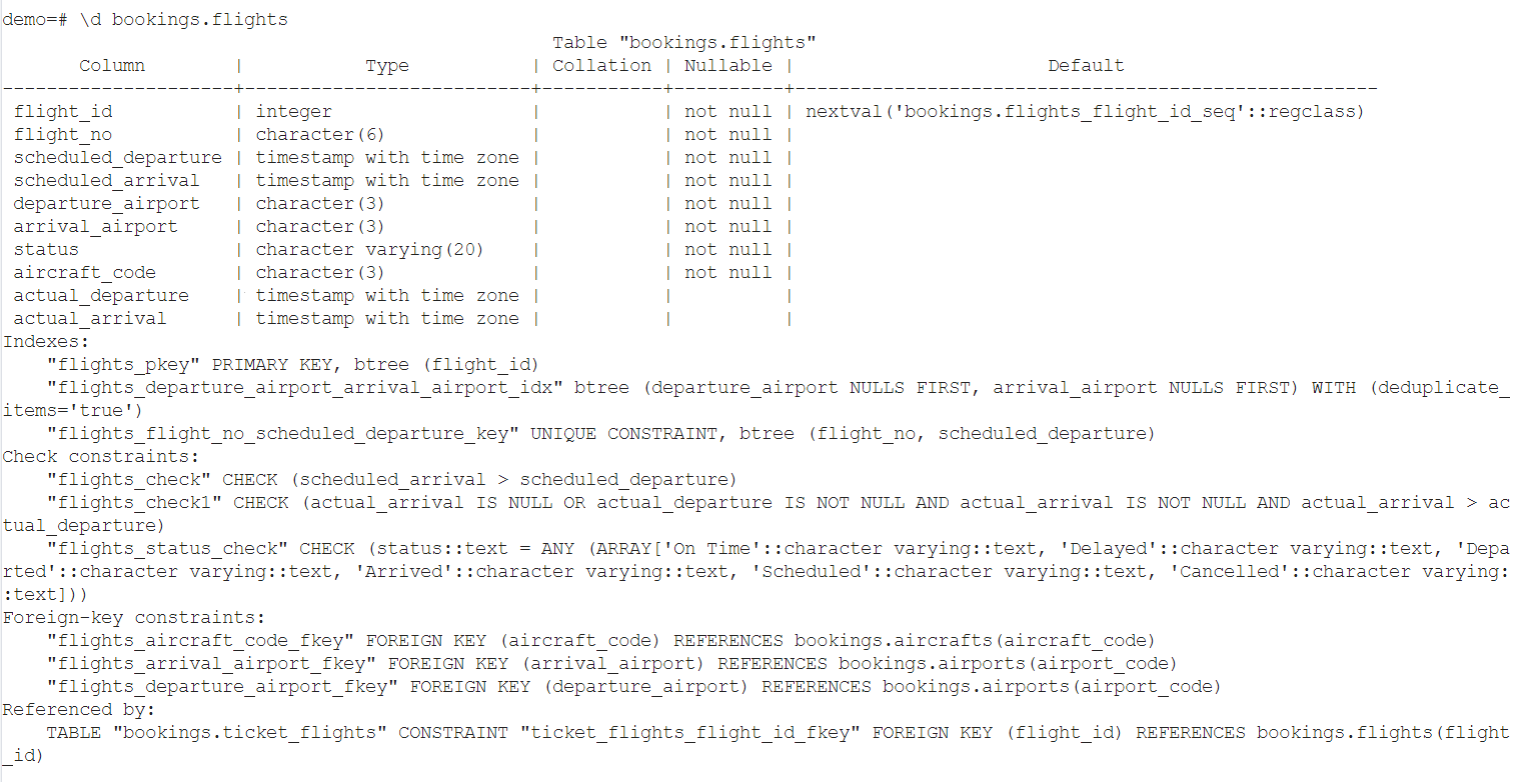


Рисунок 47 – Информация о таблице bookings.flights

Каждый индекс, который был создан самой СУБД, имеет типовое имя, состоящее из следующих компонентов: – имени таблицы и суффикса pkey — для первичного ключа; – имени таблицы, имен столбцов, по которым создан индекс, и суффикса key — для уникального ключа, а суффикс idx – для индекса. В описании также присутствует список столбцов, по которым создан индекс, и тип индекса — в данном случае это btree, т. е. B-дерево.

Создавать и изменять индексы можно с помощью pgAdmin. Но эти возможности оставим для самостоятельного изучения.

PostgreSQL может создавать индексы различных типов, но по умолчанию используется так называемое B-дерево. Такой индекс подходит для большинства типовых задач. Здесь мы будем рассматривать только индексы на основе B-дерева. Наличие индекса может ускорить выборку строк из таблицы, если он создан по столбцам, на основе значений которых и производится выборка. Поэтому, как правило, при разработке и эксплуатации баз данных не ограничиваются только индексами, которые автоматически создает СУБД, а создают дополнительные индексы с учетом наиболее часто выполняющихся выборок. Для создания индексов предназначена команда, на примере таблицы с аэропорта:

CREATE INDEX ON bookings.airports\_data ( airport\_name );

Посмотрим описание нового индекса (рис. 48):

demo=# \d bookings.airports

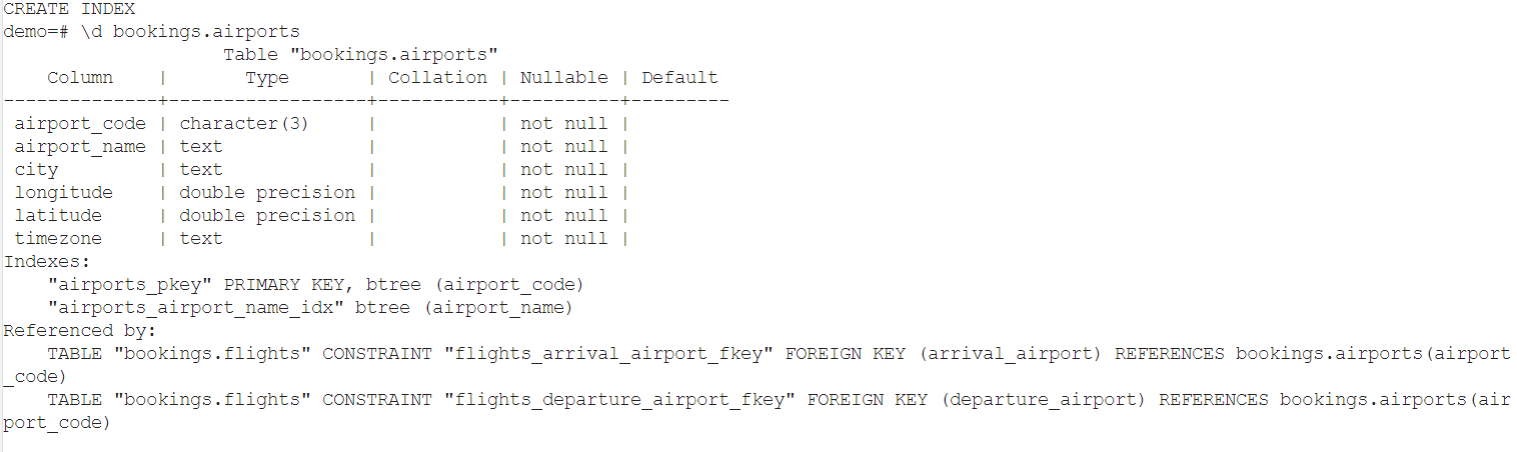


Рисунок 48 – Информация о таблице bookings.airports

Обратите внимание, что имя индекса, сформированное автоматически, включает имя таблицы, имя столбца и суффикс idx.

Прежде чем приступить к экспериментам с индексами, нужно включить в утилите psql секундомер с помощью следующей команды:

\timing on

Когда необходимость в использовании секундомера отпадет, для его отключения нужно будет сделать так:

\timing off

Теперь psql будет сообщать время, затраченное на выполнение всех команд. Для практической проверки влияния индекса на скорость выполнения выборок сначала выполним следующий запрос (рис. 49):

SELECT count(\*) FROM tickets WHERE passenger\_name = 'IVAN IVANOV';

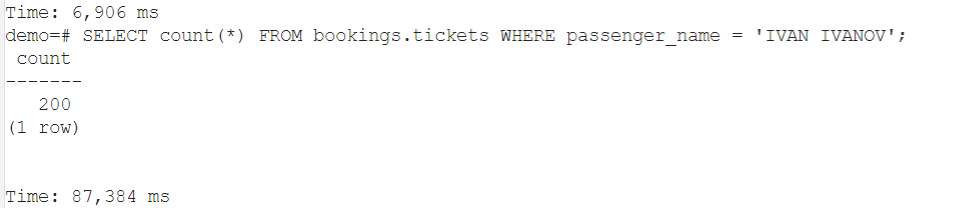


Рисунок 49 – Время выполнения запроса

Показатели времени, полученные на вашем компьютере, конечно, будут отличаться от приведенных значений, и — возможно — значительно. Эти показатели нужно рассматривать лишь как качественные ориентиры.

Создадим индекс по столбцу passenger\_name, при этом никакого суффикса в имени индекса использовать не будем, поскольку его наличие не является обязательным:

CREATE INDEX passenger\_name ON tickets (passenger\_name);

Теперь выполним ту же выборку из таблицы tickets:

SELECT count(\*) FROM tickets WHERE passenger\_name = 'IVAN IVANOV';

Теперь видно, что время выполнения выборки при наличии индекса оказалось значительно меньше (рис. 50):

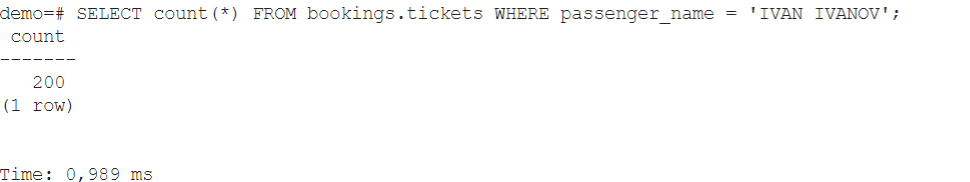


Рисунок 50 – Время выполнения запроса

Для удаления индекса используется команда:

DROP INDEX bookings.passenger\_name;

Хотя хорошо известно, что индексы повышают производительность базы данных, следует знать, как их заставить работать должным образом. Добавление ненужных или неподходящих индексов к таблице может даже привести к снижению производительности. Ниже предоставлены некоторые рекомендации по созданию эффективных индексов в БД.

* Индекс имеет смысл, если нужно обеспечить доступ одновременно не более чем к 4-5% данных таблицы. Альтернативной использования индекса для доступа к данным строки является полное последовательное чтение таблицы от начала до конца, что называется полным сканированием таблицы. Полное сканирование таблицы больше подходит для запросов, которые требуют извлечения большего процента данных таблицы. Помните, что применение индексов для извлечения строк требует двух операций чтения: индекса и затем таблицы.
* Избегайте создания индексов для сравнительно небольших таблиц. Для таких таблиц больше подходит полное сканирование. В случае маленьких таблиц нет необходимости в хранении данных и таблиц, и индексов.
* Создавайте первичные ключи для всех таблиц. При назначении столбца в качестве первичного колюча Postgres автоматически создаст индекс по этому столбцу.
* Индексируйте столбцы, участвующие в многотабличных операциях соединения (JOIN).
* Индексируйте столбцы, которые часто используются в конструкциях WHERE.
* Индексируйте столбцы, участвующие в операциях ORDER BY и GROUP BY или других операциях, таких как UNION и DISTINCT, включающих сортировку. Поскольку индексы уже отсортированы, объем работы по выполнению необходимой сортировки данных для упомянутых операций будет существенно сокращен.
* Столбцы, стоящие из длинно-символьных строк, обычно плохие кандидаты на индексацию.
* Столбцы, которые часто обновляются, в идеале не должны быть индексированы из-за связанных с этим накладных расходов.
* Индексируйте таблицы, в которых мало строк имеют одинаковые значения.
* Сохраняйте количество индексов небольшим.
* Составные индексы могут понадобиться там, где одностолбцовые значения сами по себе не уникальны. В составных индексах первым столбцом ключа должен быть столбец, в котором количество строк с одинаковым значением минимально.

Всегда помните золотое правило индексации таблиц: индекс таблицы должен быть основан на типах запросов, которые будут выполняться над столбцами этой таблицы. На таблице можно создавать более одного индекса: например, можно создать индекс на столбце X, или столбце Y, или обоих сразу, а также один составной индекс на обоих столбцах. Принимая правильное решение относительно того, какие индексы следует создавать, подумайте о наиболее часто используемых типах запросов данных таблицы.

# 3. Операторы манипулирования данными (DML)

## 3.1. Теоретическая часть и практические примеры

### 3.1.1. Оператор INSERT

Оператор INSERT позволяет вставить одну или несколько строк в таблицу. Вот основной синтаксис оператора INSERT:

INSERT INTO table\_name (column\_list)

VALUES

(value\_list\_1),

(value\_list\_2),

...

(value\_list\_n);

В этом синтаксисе:

* Сначала укажите имя таблицы, в которую вы хотите вставить данные, после ключевых слов INSERT INTO.
* Во-вторых, перечислите требуемые столбцы или все столбцы таблицы в скобках после имени таблицы.
* В-третьих, укажите список строк, разделенных запятыми, после ключевого слова VALUES.

Чтобы вставить несколько строк и вернуть вставленные строки, добавьте следующее предложение RETURNING:

INSERT INTO table\_name (column\_list)

VALUES

(value\_list\_1),

(value\_list\_2),

...

(value\_list\_n)

RETURNING \* | output\_expression;

Звездочка (\*) позволяет вернуть все столбцы вставленных строк. Если вы хотите вернуть значения в определенном столбце, вы также можете указать их после ключевого слова RETURNING.

Вставка нескольких строк одновременно имеет преимущества перед вставкой одной строки за раз:

* **Производительность:** Вставка нескольких строк в одном операторе часто более эффективна, чем несколько отдельных вставок, поскольку это сокращает количество циклов обмена между приложением и сервером PostgreSQL.
* **Атомарность:** Оператор INSERT является атомарным, что означает, что либо вставляются все строки, либо не вставляются ни одной. Это обеспечивает согласованность данных.

Команда INSERT используется для того, чтобы вставлять новые записи в таблицы. К следующему примеру добавим новые самолеты в таблицу aircrafts.

INSERT INTO bookings.aircrafts (aircraft\_code, model, range)

VALUES

('152', 'Ту-154М', 1111),

('153', 'МС-21', 1111)

RETURNING \*;

Результат (рис. 51):

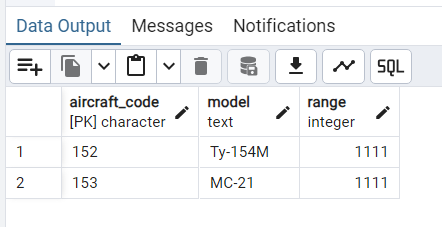


Рисунок 51 – Результат

На первый взгляд в предложении RETURNING нет особой необходимости, мы знаем какие строки вставили (или не вставили).

Вернемся к БД university. Вам поручили разработать SQL функцию, принимающую название группы, которую надо добавить в таблицу и вернуть id\_group для дальнейшего использования (чтобы использовать при добавлении студентов в группу). Поскольку id\_group значение формируется автоматически, то его можно получить после операции вставки. В этом случае целесообразно использовать RETURNING, вместо операции SELECT, для получения id\_group.

INSERT INTO public.group (name)

VALUES ('ИУ6-30Б')

RETURNING id\_group;

Результат (рис. 52):

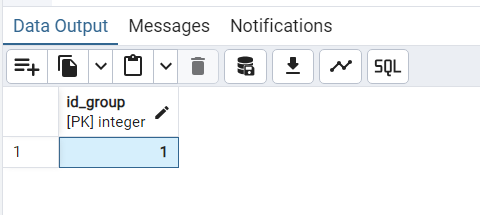


Рисунок 52 – Результат

Как видим, получили id\_group[[5]](#footnote-5).

### 3.1.2. Оператор DELETE

Оператор DELETE позволяет удалить одну или несколько строк из таблицы. Ниже показан основной синтаксис оператора DELETE:

DELETE FROM table\_name

WHERE condition;

В этом синтаксисе:

* Сначала укажите имя (table\_name) таблицы, из которой вы хотите удалить данные, после ключевых слов DELETE FROM.
* Во-вторых, укажите в предложении условие [WHERE](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-where/), определяющее, какие строки следует удалить.

Предложение WHERE необязательно. Если вы пропустите предложение WHERE, то оператор DELETE удалит все строки в таблице.

Оператор DELETE возвращает количество удаленных строк. Если он возвращает ноль, если оператор DELETE не удалил ни одной строки.

Чтобы вернуть удаленные строки клиенту, вы используете предложение RETURNING:

DELETE FROM table\_name

WHERE condition

RETURNING \* | output\_expression;

Звездочка (\*) позволяет вернуть все столбцы удаленной строки(строк). Если вы хотите вернуть значения в определенном столбце, вы также можете указать их после ключевого слова RETURNING.

Удалим одну добавленную строку в таблице aircrafts:

DELETE FROM bookings.aircrafts

WHERE aircraft\_code = '152'

RETURNING model;

Результат (рис. 53):

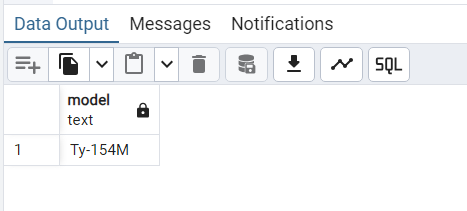


Рисунок 53 – Результат

### 3.1.2. Оператор UPDATE

Оператор UPDATE позволяет обновлять данные в одном или нескольких столбцах одной или нескольких строк таблицы. Вот основной синтаксис этого оператора UPDATE:

UPDATE table\_name

SET column1 = value1,

column2 = value2,

...

WHERE condition;

В этом синтаксисе:

* Сначала укажите имя таблицы, данные которой вы хотите обновить, после ключевого слова UPDATE.
* Во-вторых, укажите столбцы и их новые значения после ключевого слова SET. Столбцы, которые не появляются в предложении SET, сохраняют свои исходные значения.
* В-третьих, определите, какие строки в условии предложения следует обновить [WHERE](https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-where/).

Предложение WHERE необязательно. Если вы пропустите предложение WHERE, оператор UPDATE обновит все строки в таблице.

В операторе UPDATE есть необязательное предложение RETURNING, которое возвращает обновленные строки:

UPDATE table\_name

SET column1 = value1,

column2 = value2,

...

WHERE condition

RETURNING \* | output\_expression;

Исправим дальность полета у МС-21, в таблице aircrafts:

UPDATE bookings.aircrafts

SET range = 6500

WHERE aircraft\_code = '153'

RETURNING model, range;

Результат (рис. 54):

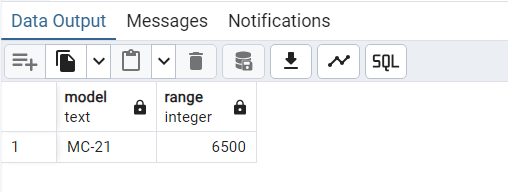


Рисунок 54 – Результат

# 4. Представления

## 4.1. Теоретическая часть

Представление — это именованный запрос, хранящийся на сервере базы данных PostgreSQL. Представление определяется на основе одной или нескольких таблиц, которые называются базовыми таблицами, а запрос, определяющий представление, называется определяющим запросом.

После создания представления вы можете запрашивать данные из него так же, как из обычной таблицы.

Представления не хранят данные, за исключением материализованных представлений. В PostgreSQL вы можете создавать специальные представления, называемые материализованными представлениями, которые физически хранят данные и периодически обновляют их из базовых таблиц.

Материализованные представления удобны в различных сценариях, обеспечивая более быстрый доступ к данным на удаленном сервере и выступая в качестве эффективного механизма кэширования.

Представления предлагают множество преимуществ:

* Представления помогают упростить сложные запросы. Вместо того, чтобы иметь дело с объединениями, агрегациями или условиями фильтрации, вы можете запрашивать представления, как если бы они были обычными таблицами. Обычно сначала вы создаете представления на основе сложных запросов и сохраняете их в базе данных. Затем вы можете использовать простые запросы на основе представлений вместо использования сложных запросов.
* Представления обеспечивают детальный контроль доступа к данным. Вы можете создавать представления, которые раскрывают подмножества данных в базовых таблицах, скрывая конфиденциальную информацию. Это особенно полезно, если у вас есть приложения, которым требуется доступ к отдельным частям данных.
* Если ваши приложения используют представления, вы можете свободно изменять структуру базовых таблиц. Другими словами, представления позволяют вам создать слой абстракции над базовыми таблицами.

Чтобы создать новое представление, можно использовать оператор CREATE VIEW.

Вот основной синтаксис этого оператора CREATE VIEW:

CREATE VIEW view\_name [(column\_list)]

AS

query;

В этом синтаксисе:

* Сначала укажите название представления после ключевых слов CREATE VIEW.
* Во-вторых, необязательный список имен столбцов представления. Если список столбцов не указан, то имена столбцов определяются на основании текста запроса.
* В-третьих, укажите оператор SELECT (query), который определяет представление. Запрос часто называют определяющим запросом представления.

Давайте создадим простое представление. За основу возьмем запрос для подсчета количества мест в салонах для всех моделей самолетов с учетом класса обслуживания:

SELECT aircraft\_code, fare\_conditions, count(\*)

FROM seats

GROUP BY aircraft\_code, fare\_conditions

ORDER BY aircraft\_code, fare\_conditions;

По данному запросу создадим представление и дадим ему имя, отражающее суть этого представления:

CREATE VIEW seats\_by\_fare\_cond

AS

SELECT aircraft\_code, fare\_conditions, count(\*)

FROM seats

GROUP BY aircraft\_code, fare\_conditions

ORDER BY aircraft\_code, fare\_conditions;

Теперь мы можем вместо написания сложного первоначального запроса обращаться непосредственно к представлению, как будто это обычная таблица:

SELECT \* FROM seats\_by\_fare\_cond;

СУБД PostgreSQL предлагает свое расширение команды CREATE VIEW, а именно – фразу OR REPLACE. Если представление уже существует, то можно его не удалять, а просто заменить новой версией. Однако нужно помнить о том, что при создании новой версии представления (без явного удаления старой с помощью команды DROP VIEW) должны оставаться неизменными имена столбцов представления. Если же вы хотите изменить имя хотя бы одного столбца, то сначала нужно удалить представление с помощью команды DROP VIEW, а уже затем создать его заново.

Имена столбцов можно явно указать в команде, но если они не указаны, то СУБД сама «вычислит» эти имена. В только что созданном нами представлении третий столбец получит имя count. Если мы захотим изменить это имя, то возможны два способа: первый заключается в том, чтобы создать псевдоним для этого столбца с помощью ключевого слова AS, а второй – в указании списка имен столбцов в начале команды CREATE VIEW.

Если захотим в представления изменить имя столбца, а не просто изменить логику работы в представлении, нам придется сначала удалить это представление, а затем создать его заново:

DROP VIEW seats\_by\_fare\_cond;

CREATE OR REPLACE VIEW seats\_by\_fare\_cond

AS

SELECT aircraft\_code, fare\_conditions, count(\*) AS num\_seats

FROM seats

GROUP BY aircraft\_code, fare\_conditions

ORDER BY aircraft\_code, fare\_conditions;

Есть второй способ задания имен столбцов в представлении — с помощью списка их имен, заключенного в скобки:

DROP VIEW seats\_by\_fare\_cond;

CREATE OR REPLACE VIEW seats\_by\_fare\_cond (code, fare\_cond, num\_seats ) AS

SELECT aircraft\_code, fare\_conditions, count(\*)

FROM seats

GROUP BY aircraft\_code, fare\_conditions

ORDER BY aircraft\_code, fare\_conditions;

Бывают ситуации, когда заранее известно, что возможна попытка удаления несуществующего представления. В таких случаях обычно стараются избежать ненужных сообщений об ошибке отсутствия представления. Для этого в команду DROP VIEW добавляют фразу IF EXISTS.

В базе данных «demo» создано представление «Рейсы» (flights\_v), сконструированное на основе таблицы «Рейсы» (flights), на нём и воспользуемся IF EXISTS:

DROP VIEW IF EXISTS flights\_v;

## 4.2. Вопросы для самостоятельного изучения

1. Модификация, вставка и удаление данных через представления.
2. Контроль над обновлением представлений (CHECK OPTION).
3. Материализованные представления.
4. Обновление материализованных представлений.

# 5. Транзакции

## 5.1. Транзакции и их свойства

**Транзакции** – это фундаментальное понятие во всех СУБД. Суть транзакции в том, что она объединяет последовательность действий в одну операцию «всё или ничего». Промежуточные состояния внутри последовательности не видны другим транзакциям, и, если что-то помешает успешно завершить транзакцию, ни один из результатов этих действий не сохранится в базе данных.

Например, рассмотрим базу данных банка, в которой содержится информация о счетах клиентов, а также общие суммы по отделениям банка. Предположим, что мы хотим перевести 100 долларов со счёта Алисы на счёт Боба. Простоты ради, соответствующие SQL-команды можно записать так:

* изменение состояния счета Алисы

UPDATE accounts SET balance = balance - 100.00

WHERE name = 'Alice';

* изменение баланса отделения банка, в котором ведется счет Алисы

UPDATE branches SET balance = balance - 100.00

WHERE name = (SELECT branch\_name FROM accounts   
 WHERE name = 'Aice');

* изменение состояния счета Боба

UPDATE accounts SET balance = balance + 100.00

WHERE name = 'Bob';

* изменение баланса отделения банка, в котором ведется счет Боба

UPDATE branches SET balance = balance + 100.00

WHERE name = (SELECT branch\_name FROM accounts   
 WHERE name = 'Bob');

Точное содержание команд здесь не важно, важно лишь то, что для выполнения этой довольно простой операции потребовалось несколько отдельных действий. При этом с точки зрения банка необходимо, чтобы все эти действия выполнились вместе, либо не выполнились совсем. Если Боб получит 100 долларов, но они не будут списаны со счёта Алисы, объяснить это сбоем системы определённо не удастся. И наоборот, Алиса вряд ли будет довольна, если она переведёт деньги, а до Боба они не дойдут. Нам нужна гарантия, что, если что-то помешает выполнить операцию до конца, ни одно из действий не оставит следа в базе данных. И мы получаем эту гарантию, объединяя действия в одну транзакцию. Говорят, что транзакция Atomicity (атомарная, неделимая*)* с точки зрения других транзакций она либо выполняется и фиксируется полностью, либо не фиксируется совсем. Помимо атомарности есть ещё три свойства транзакции.

**Consistency** (согласованность) – гарантирует, что по мере выполнения транзакций, данные переходят из одного согласованного состояния в другое, то есть транзакция не может разрушить взаимной согласованности данных.

**Isolation** (изолированность) – локализация пользовательских процессов означает, что конкурирующие за доступ к БД транзакции физически обрабатываются последовательно, изолированно друг от друга, но для пользователей это выглядит, как будто они выполняются параллельно. Или же по-другому, во время выполнения транзакции другие транзакции должны оказывать по возможности минимальное влияние на нее.

**Durability** (долговечность) – устойчивость к ошибкам – если транзакция завершена успешно, то те изменения в данных, которые были ею произведены, не могут быть потеряны ни при каких обстоятельствах.

Для обозначения всех этих четырех свойств используется аббревиатура ACID.

## 5.2. Аномалии

Ситуации, когда корректные транзакции некорректно работают вместе, называются аномалиями одновременного выполнения.

Простой пример: если приложение хочет получить из базы корректные данные, то оно, как минимум, не должно видеть изменения других незафиксированных транзакций. Иначе можно не просто получить несогласованные данные, но и увидеть что-то такое, чего в базе данных никогда не было (если транзакция будет отменена). Такая аномалия называется грязнымчтением. Если и другие, более сложные аномалии, с которыми мы разберемся чуть позже. Отказываться от одновременного выполнения, конечно, нельзя: иначе о какой производительности может идти речь? Но нельзя и работать с некорректными данными.

И снова на помощь приходит СУБД. Можно сделать так, чтобы транзакции выполнялись как будто последовательно*,* как будто одна за другой. Иными словами — изолированно друг от друга. В реальности СУБД может выполнять операции вперемешку, но гарантировать при этом, что результат одновременного выполнения будет совпадать с результатом какого-нибудь из возможных последовательных выполнений. А это устраняет любые возможные аномалии.

При параллельном выполнении транзакций теоретически возможны следующие аномалии.

**«Грязное» чтение** (dirty read). Транзакция читает данные, измененные параллельной транзакцией, которая еще не завершилась. Если эта параллельная транзакция в итоге будет отменена, тогда окажется, что первая транзакция прочитала данные, которых нет в системе.

**Неповторяющееся чтение** (non-repeatable read). При повторном чтении тех же самых данных в рамках одной транзакции оказывается, что другая транзакция успела изменить и зафиксировать эти данные. В результате тот же самый запрос выдает другой результат.

**Фантомное чтение** (phantom read). Транзакция выполняет повторную выборку множества строк в соответствии с одним и тем же критерием. В интервале времени между выполнением этих выборок другая транзакция добавляет новые строки и успешно фиксирует изменения. В результате при выполнении повторной выборки в первой транзакции может быть получено другое множество строк.

## 5.3. Транзакции и их представление в SQL

Транзакции в PostgreSQL определяются как набор SQL-команд, окружённым командами BEGIN и COMMIT.

BEGIN; --НАЧАЛО (начать транзакцию)

UPDATE accounts

SET balance = balance - 100.00

WHERE name = 'Alice'; ….

COMMIT; --КОНЕЦ (успешное завершение транзакции)

Если в процессе выполнения транзакции мы решим, что не хотим фиксировать её изменения (например, потому что оказалось, что баланс Алисы стал отрицательным), мы можем выполнить команду ROLLBACK вместо COMMIT, и все наши изменения будут отменены. Поставив к примеру «SAVEPOINT save1» перед тем, как пополнять счет Боба и с помощью «ROLLBACK TO save1» вернуться в точку сохранения.

## 5.4. Уровни изоляции и аномалии

Уровень изоляции транзакции определяет степень изоляции транзакции от действий других параллельно выполняющихся транзакций. Уровень изоляции определяет следующее:

* Будут ли строки, прочитанные и обновленные в текущей сессии, доступны другим одновременно выполняющимся сессиям базы данных.
* Будут ли обновления и записи строк другими транзакциями влиять на текущую транзакцию.

Таблица 1 показывает влияние уровня изоляции транзакции на возможность возникновения аномалий в соответствии со стандартом SQL2003.

**Таблица 1. Уровни изоляции транзакции**

|  |  |  |  |
| --- | --- | --- | --- |
| **Уровень изоляции** | **Грязное чтение** | **Неповторяющееся чтение** | **Фантомные записи** |
| READ UNCOMMITED | Возможно | Возможно | Возможно |
| READ COMMITED | Невозможно | Возможно | Возможно |
| REPEATABLE READ | Невозможно | Невозможно | Возможно[[6]](#footnote-6) |
| SERIALIZABLE | Невозможно | Невозможно | Невозможно |

**READ UNCOMMITTED.** Это самый низкий уровень изоляции. Согласно стандарту SQL на этом уровне допускается чтение «грязных» (незафиксированных) данных. Однако в PostgreSQL требования, предъявляемые к этому уровню, более строгие, чем в стандарте: чтение «грязных» данных на этом уровне не допускается.

Для организации выполнения параллельных транзакций с использованием утилиты psql будем запускать ее на двух терминалах.

На первом терминале выполним следующие команды (рис. 55):

BEGIN;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SHOW transaction\_isolation;

UPDATE bookings.aircrafts

SET range = range + 100

WHERE aircraft\_code = 'SU9';

SELECT \* FROM bookings.aircrafts WHERE aircraft\_code = 'SU9';

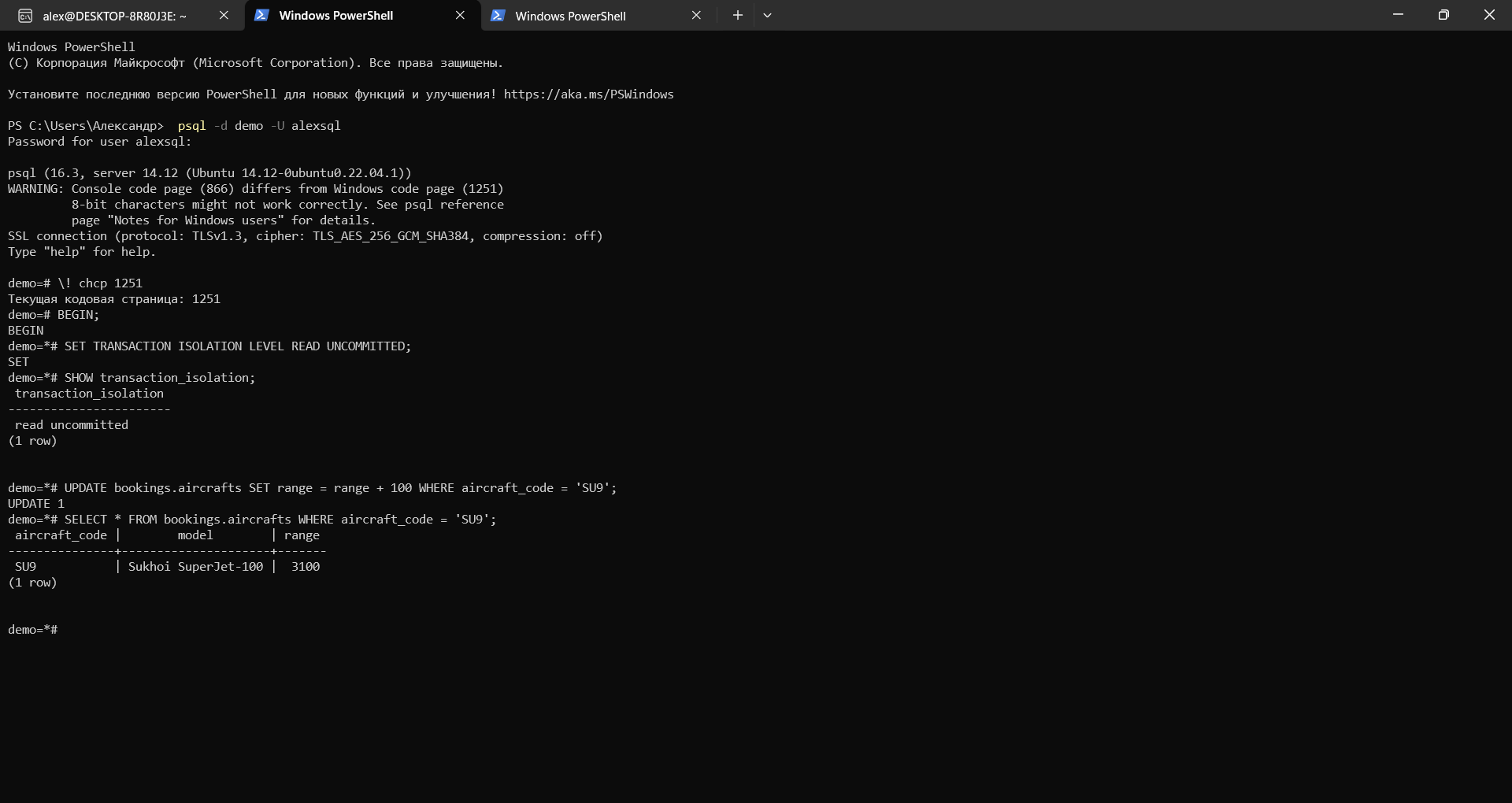


Рисунок 55 – Терминал 1

А во-втором терминале (рис. 56) начнём другую транзакцию.

BEGIN TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

SHOW transaction\_isolation;

SELECT \* FROM bookings.aircrafts WHERE aircraft\_code = 'SU9';

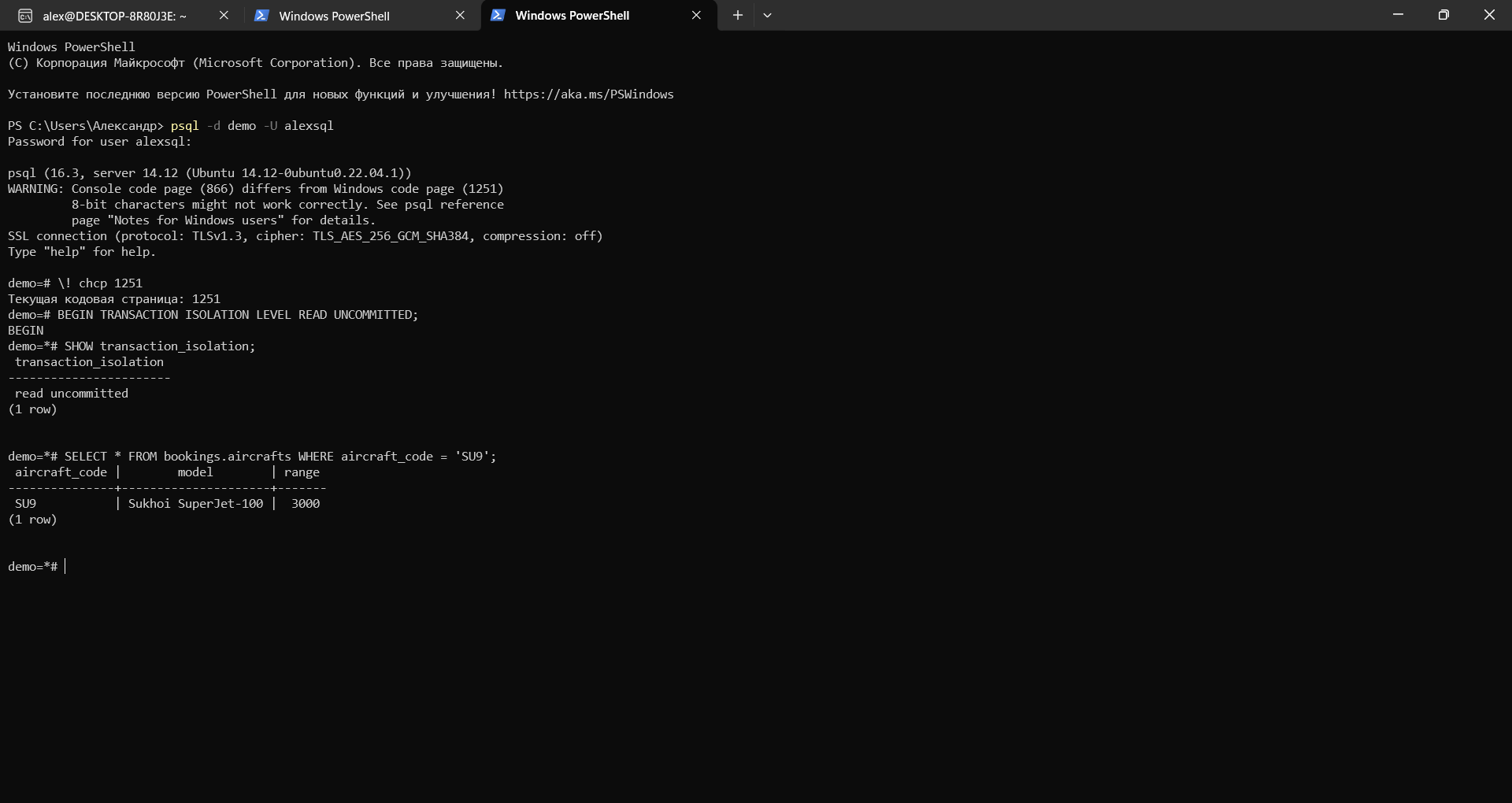


Рисунок 56 – Терминал 2

Можно увидеть, что в первой транзакции значение атрибута range было успешно изменено, хотя пока и не зафиксировано. Но транзакция видит изменения, выполненные в ней самой. Обратите внимание, что вместо использования команды SET TRANSACTION мы просто включили указание уровня изоляции непосредственно в команду BEGIN. Эти два подхода равносильны. Конечно, когда речь идет об использовании уровня изоляции READ COMMITTED, принимаемого по умолчанию, можно вообще ограничиться только командой BEGIN без дополнительных ключевых слов.

Таким образом, вторая транзакция не видит изменение значения атрибута range, произведенное в первой — незафиксированной — транзакции. Это объясняется тем, что в PostgreSQL реализация уровня изоляции READ UNCOMMITTED более строгая, чем того требует стандарт языка SQL.

Фактически этот уровень тождественен уровню изоляции READ COMMITTED. Поэтому будем считать эксперимент, проведенный для уровня изоляции READ UNCOMMITTED, выполненным и для уровня READ COMMITTED. Не будем фиксировать изменения в обоих консолях, для этого пишем команду ROLLBACK.

**READ COMMITTED.** Не допускается чтение «грязных» (незафиксированных) данных. Таким образом, в PostgreSQL уровень READ UNCOMMITTED совпадает с уровнем READ COMMITTED. Транзакция может видеть только те незафиксированные изменения данных, которые произведены в ходе выполнения ее самой.

В прошлых запросах уже было показано, что на этом уровне изоляции не допускается чтение незафиксированных данных. Далее покажем, что на этом уровне изоляции также гарантируется отсутствие потерянных обновлений, но возможно неповторяющееся чтение данных. Опять будем работать на двух терминалах. В первой транзакции увеличим значение атрибута range для самолета Sukhoi SuperJet-100 на 100 км, а во второй транзакции — на 200 км. Проверим, какое из этих двух изменений будет записано в базу данных.

SET search\_path = bookings;

BEGIN ISOLATION LEVEL READ COMMITTED;

SHOW transaction\_isolation;

UPDATE aircrafts

SET range = range + 100

WHERE aircraft\_code = 'SU9';

SELECT \* FROM aircrafts WHERE aircraft\_code = 'SU9';

На первом терминале (рис. 57) получаем:

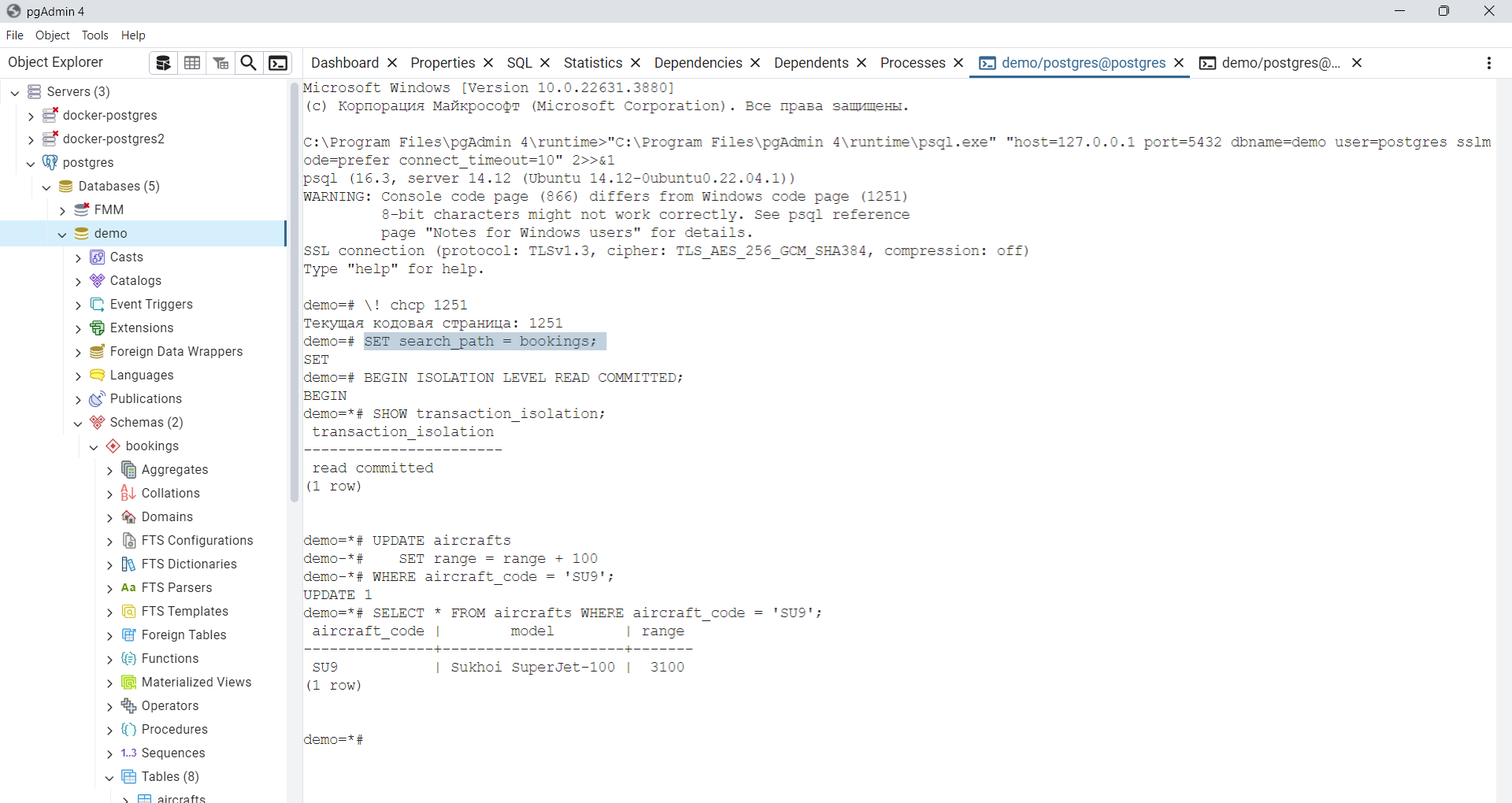


Рисунок 57 – Терминал 1

Во второй транзакции (рис. 58) попытаемся обновить эту же строку таблицы aircrafts, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута range не 100, а 200.

SET search\_path = bookings;

BEGIN ISOLATION LEVEL READ COMMITTED;

SHOW transaction\_isolation;

UPDATE aircrafts

SET range = range + 200

WHERE aircraft\_code = 'SU9';

SELECT \* FROM aircrafts WHERE aircraft\_code = 'SU9';

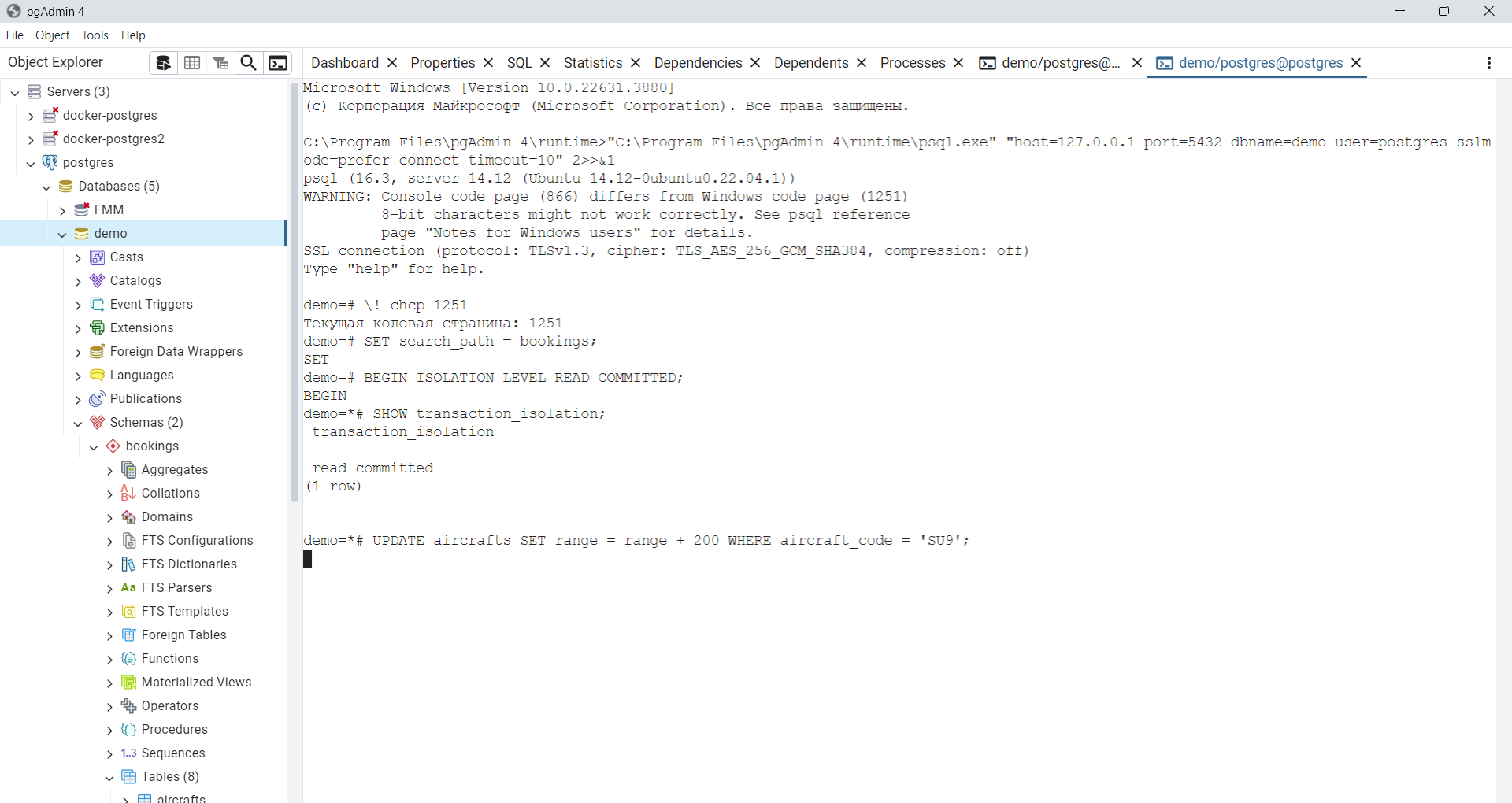


Рисунок 58 – Терминал 2

Наблюдаем, что команда UPDATE во второй транзакции не завершилась, а перешла в состояние ожидания. Это ожидание продлится до тех пор, пока не завершится первая транзакция. Дело в том, что команда UPDATE в первой транзакции заблокировала строку в таблице airctafts, и эта блокировка будет снята только при завершении транзакции либо с фиксацией изменений с помощью команды COMMIT, либо с отменой изменений по команде ROLLBACK.

Зафиксируем первую транзакцию – COMMIT. И перейдя обратно во второю консоль (рис. 59) уже видим завершенную команду UPDATE.

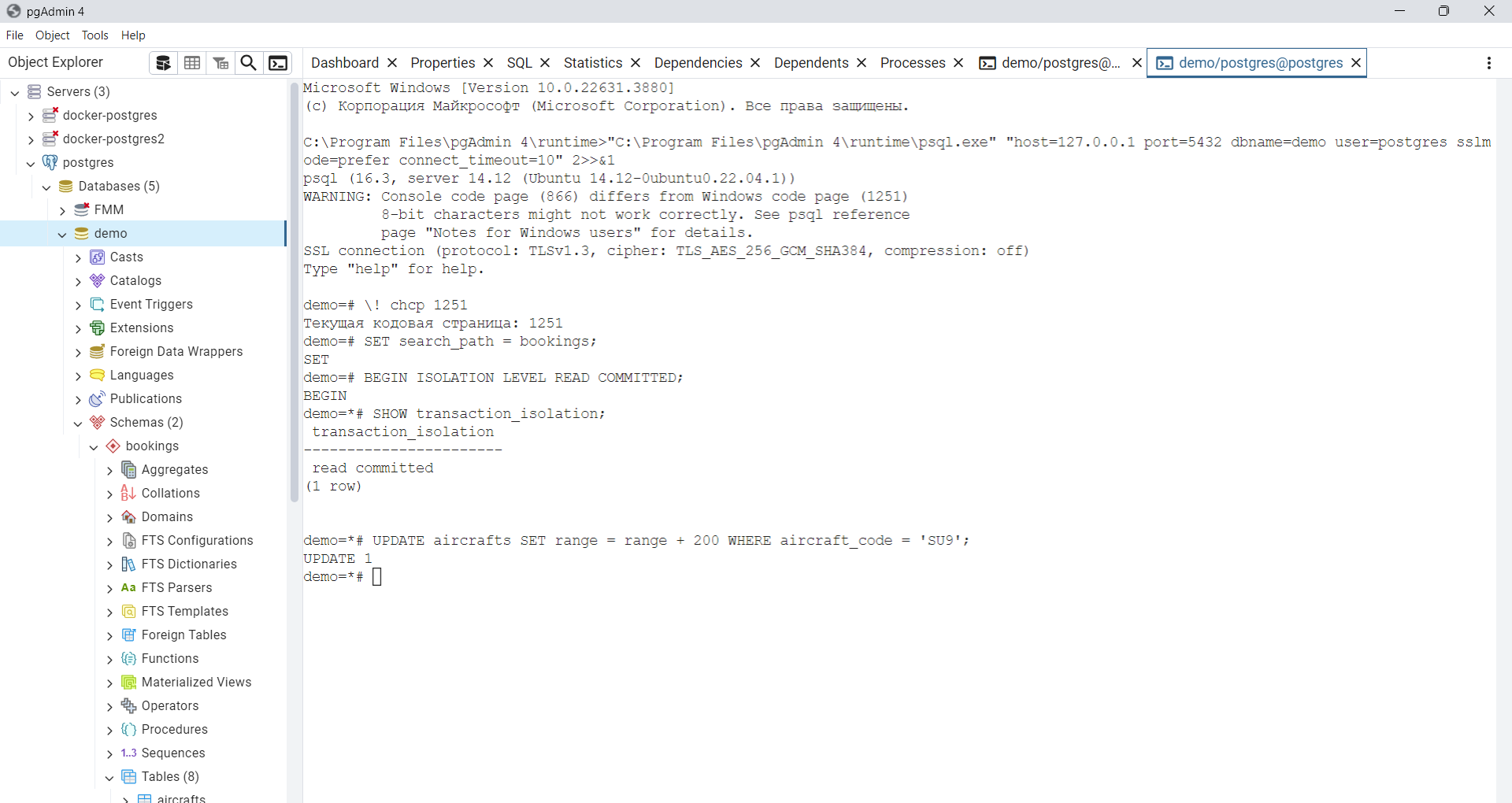


Рисунок 58 – Терминал 2

Теперь на втором терминале, не завершая транзакцию, посмотрим, что стало с нашей строкой в таблице aircrafts (рис. 59):

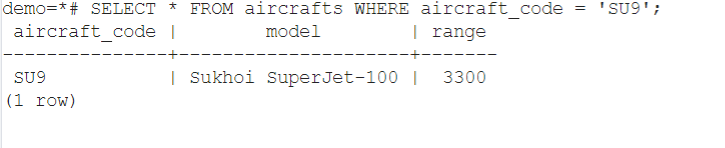


Рисунок 59 – Результат изменений

Как видно, были произведены оба изменения. Команда UPDATE во второй транзакции, получив возможность заблокировать строку после завершения первой транзакции и снятия ею блокировки с этой строки, перечитывает строку таблицы и потому обновляет строку, уже обновленную в только что зафиксированной транзакции. Таким образом, эффекта потерянных обновлений не возникает.

Завершим транзакцию на втором терминале, но при этом вместо команды COMMIT воспользуемся эквивалентной командой END, которая является расширением PostgreSQL.

**REPEATABLE READ.** Третий уровень изоляции — REPEATABLE READ. Само его название говорит о том, что он не допускает наличия феномена неповторяющегося чтения данных. А в PostgreSQL на этом уровне не допускается и чтение фантомных строк.

Приложения, использующие этот уровень изоляции, должны быть готовы к тому, что придется выполнять транзакции повторно. Это объясняется тем, что транзакция, использующая этот уровень изоляции, создает снимок данных не перед выполнением каждого запроса, а только однократно, перед выполнением первого запроса транзакции. Поэтому транзакции с этим уровнем изоляции не могут изменять строки, которые были изменены другими завершившимися транзакциями уже после создания снимка. Вследствие этого PostgreSQL не позволит зафиксировать транзакцию, которая попытается изменить уже измененную строку.

Важно помнить, что повторный запуск может потребоваться только для транзакций, которые вносят изменения в данные. Для транзакций, которые только читают данные, повторный запуск никогда не требуется. В первом терминале (рис. 60) пишем такую транзакцию:

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT \* FROM bookings.aircrafts;

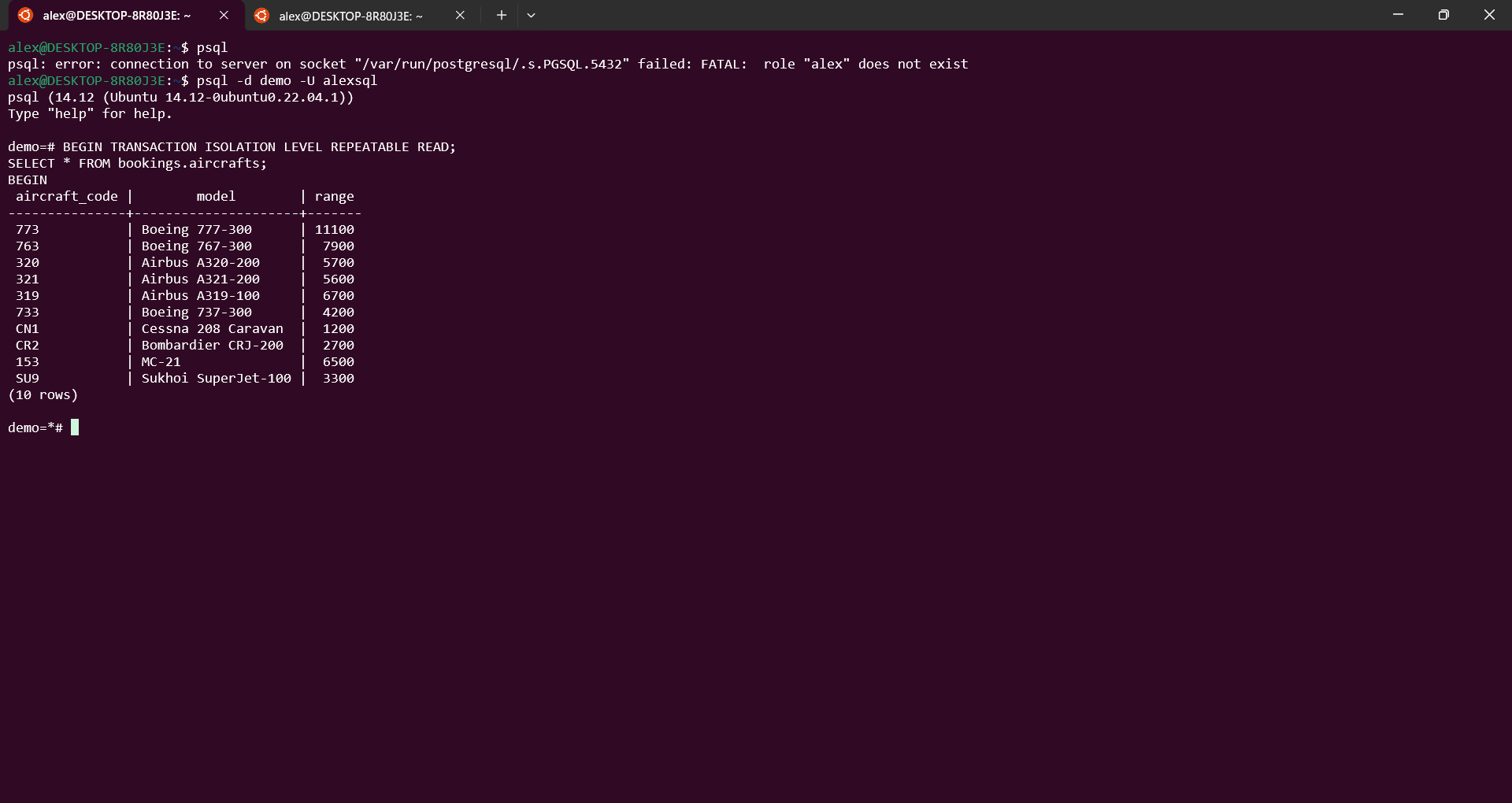


Рисунок 61 – Терминал 1

На втором терминале (рис. 62) проведем ряд изменений:

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

INSERT INTO bookings.aircrafts

VALUES ('IL9', 'Ильюшин ИЛ 96', 9800);

UPDATE bookings.aircrafts

SET range = range + 100

WHERE aircraft\_code = '320';

COMMIT;

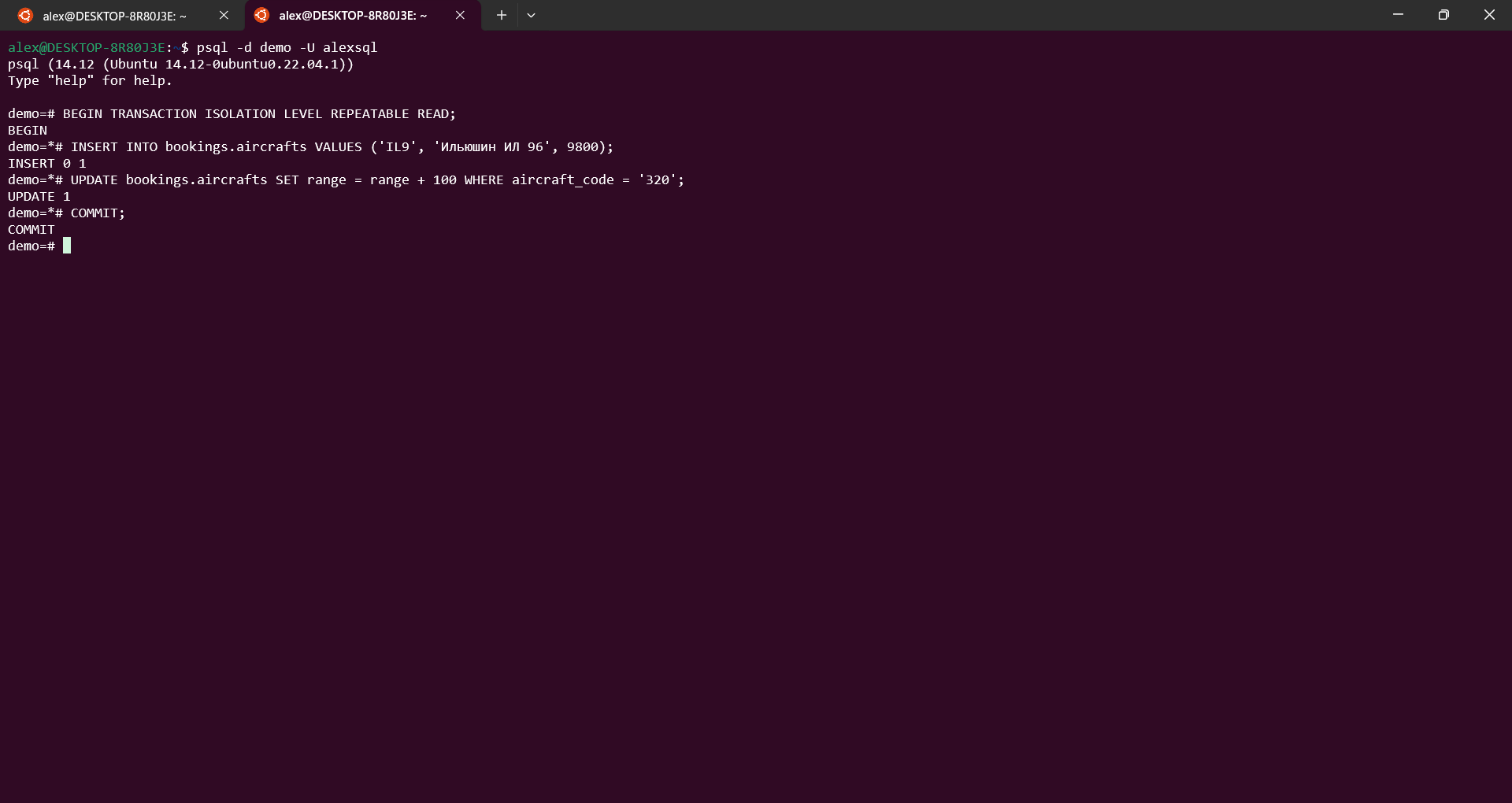


Рисунок 62 – Терминал 2

Переходим на первый терминал (рис. 63). И пишем запрос для вывода всей информации из таблицы aircrafts.

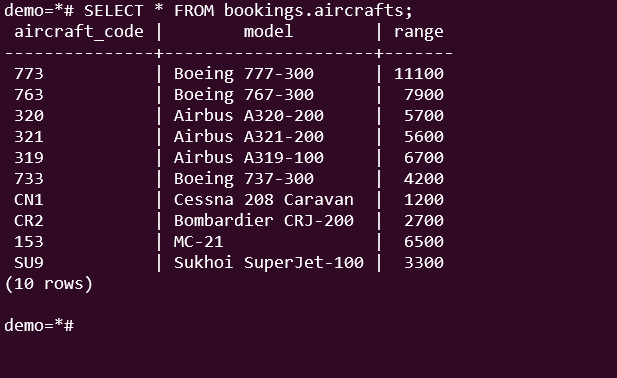


Рисунок 63 – Терминал 1

На первом терминале ничего не изменилось: фантомные строки не видны, и также не видны изменения в существующих строках. Это объясняется тем, что снимок данных выполняется на момент начала выполнения первого запроса транзакции. Зафиксируем транзакцию на первой консоли (рис. 64) и повторим запрос.

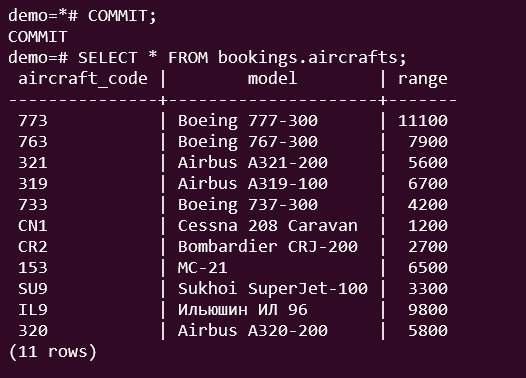


Рисунок 65 – Терминал 1

Как видим, одна строка добавлена, а значение атрибута range у самолета Airbus A320- 200 стало на 100 больше, чем было. Но до тех пор, пока мы на первом терминале находились в процессе выполнения первой транзакции, все эти изменения не были ей доступны, поскольку первая транзакция использовала снимок, сделанный до внесения изменений и их фиксации второй транзакцией.

**SERIALIZABLE** Самый высший уровень изоляции транзакций — SERIALIZABLE. Транзакции могут работать параллельно точно так же, как если бы они выполнялись последовательно одна за другой. Однако, как и при использовании уровня REPEATABLE READ, приложение должно быть готово к тому, что придется перезапускать транзакцию, которая была прервана системой из-за обнаружения зависимостей чтения/записи между транзакциями. Группа транзакций может быть параллельно выполнена и успешно зафиксирована в том случае, когда результат их параллельного выполнения был бы эквивалентен результату выполнения этих транзакций при выборе одного из возможных вариантов их упорядочения, если бы они выполнялись последовательно, одна за другой.

**Использование различных уровней изоляции транзакций**

Почему бы всегда не использовать уровень SERIALIZABLE, если он убирает все недостатки. Как нетрудно догадаться, чем выше уровень изоляции, тем большей производительностью вы жертвуете. Выбор того или иного уровня изоляции зависит от конкретного случая.

# 6. Темы для самостоятельной проработки

* **Изучить pgAdmin в части создания таблиц, первичных ключей, внешних ключей, ограничений и индексов**: <https://www.pgadmin.org/docs/pgadmin4/latest/index.html>.
* **Соединение таблиц**: PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова.

# 7. Контрольные вопросы

* Что такое первичный и внешний ключи.
* Чем отличается первичный ключ от уникального индекса?
* Что такое кластерный ключ?
* Какие типы связей между таблицами существуют и их атрибуты?
* Можно ли изменять первичные ключи?
* Какие существуют способы поддержания ссылочной целостности?
* Как создать уникальный индекс не чувствительный к регистру для поля типа text, char, varchar?
* Что такое тип данных serial?
* Что такое IDENTITY?
* Что такое ограничения для БД и их типы.
* Что такое JOIN, какие типы JOIN есть?
* В каких случаях имеет смысл создавать индексы? Какие колонки следует включать в индекс и почему?
* Имеет ли значение порядок указания колонок при создании индекса?
* Можно ли не указывать в операторе INSERT INTO список столбцов, а ограничиться только таблицей.
* Если в операторе INSERT INTO в списке столбцов указаны не все столбцы, то что произойдет.
* Если удалим в таблице group БД university все строки с помощью оператора DELETE, а потом вставим группу, например:

INSERT INTO "group"(name)

VALUES ('ИУ6-40Б');

Чему будет равен id\_group?

* Аномалии доступа к БД.
* Перечислить аномалии, возникающие на каждом из уровней изолированности.
* Рассказать о свойствах транзакций.
* Рассказать об управлении транзакциями.
* Что такое клинчи? Как бороться с клинчами?
* Как обеспечивается изолированность транзакций в СУБД?
* Как бороться с проблемой фантомов?
* Что такое журнал транзакций?
* Как обеспечивается постоянство хранения (durability) в СУБД?

# 6. Практическое задание

## Задание 1

1. Практическое задание связано с проектированием схемы базы данных. Каждый индивидуальный вариант (см. приложение «Схемы предметных областей») содержит предметную область, для которой должна быть разработана база данных. К первоначальной модели предметной области необходимо добавить не менее 2-х дополнительных сущностей (таблиц), необходимых для более полного решения поставленной задачи. Студент должен решить, для чего будет использоваться создаваемая база данных, построить бизнес-процессы её применения и, исходя из анализа бизнес-процессов, создать концептуальную схему БД. Результатом данной части лабораторной работы является схема базы данных (в виде ER-диаграммы, содержащей таблицы и связи между ними, с уточнением типов полей, с описанием внешних и первичных ключей). При сдаче задания студент должен обосновать соответствие созданной схемы поставленной задаче.
2. Необходимо определить первичные и внешние ключи (с атрибутами NO ACTION, CASCADE и др.).
3. На основе разработанной ER–модели, создать БД на сервере PostgreSQL.

Для проектирования схемы и построения диаграммы можно использовать любые предназначенные для этой цели средства.

### Требования к схеме

* Схема должна соответствовать поставленной задаче.
* Связи между сущностями должны быть правильно смоделированы.
* Желательно придерживаться какой-либо системы в именовании таблиц и полей.

## Задание 2

1. Подготовить SQL-скрипта для создания БД и таблиц согласно схеме, полученной в предыдущем задании. БД должна быть создана с локалью ru\_RU.utf8.
2. Необходимо определить первичные и внешние ключи.
3. Необходимо определить ограничения полей (возможность принимать неопределенное значение, уникальные индексы, проверочные ограничения и т.д.).
4. Необходимо определить индексы, позволяющие ускорить запросы.
5. Создать БД и таблицы. Проверить локаль БД, вызвав команду \l в psql или SHOW LC\_COLLATE, SHOW LC\_CTYPE.

## Задание 3

1. Заполнить данными созданные таблицы с использованием SQL-скриптов для вставки соответствующих строк в таблицы БД. Объем данных должен составлять не менее 10 экземпляров для каждой из стержневых сущностей и 1000 экземпляров для целевой сущности.
2. Необходимо подготовить два запроса:

* Запрос к одной таблице, содержащий фильтрацию по нескольким полям.
* Запрос к нескольким связанным таблицам (JOIN), содержащий фильтрацию по нескольким полям.

1. Для каждого из этих запросов необходимо провести следующие шаги:

* Получить время выполнения запроса без использования индексов.
* Создать нужные индексы, позволяющие ускорить запрос. Получить время выполнения запроса с использованием индексов и сравнить с первоначальной статистикой.

1. Сформулировать 3 запроса на изменение и удаление из базы данных. Запросы должны быть сформулированы в терминах предметной области.
2. Написать SQL-скрипты, для которые будет срабатывать ограничения целостности.

## Задание 4

1. Создайте три представления, которые позволяли бы упростить манипуляции с данными или позволяли бы ограничить доступ к данным, предоставляя только необходимую информацию.

## Задание 5

Необходимо подготовить SQL-скрипты для проверки наличия аномалий (потерянных изменений, грязных чтений, неповторяющихся чтений, фантомов) при параллельном исполнении транзакций на различных уровнях изолированности (READ COMMITTED, REPEATABLE READ, SERIALIZABLE). Подготовленные скрипты должны работать с одной из таблиц, созданных в практическом задании 2. Для проверки наличия аномалий потребуются два параллельных сеанса, операторы в которых выполняются пошагово:

1. 1. В первом терминале установите уровень изоляции READ COMMITTED.
   2. Во втором терминале также установить уровень изоляции READ COMMITTED.
   3. В первом терминале добавить строку в таблицу, изменить данные в какой-нибудь строке и удалить какую-нибудь строку.
   4. Во втором терминале выполнить SELECT и проверить наличие изменений в таблице, выполненных в первой транзакции.
   5. В первом терминале завершить транзакцию – COMMIT.
   6. Во втором терминале выполнить SELECT и проверить наличие изменений в таблице, выполненных в первой транзакции.
   7. Во втором терминале завершить транзакцию.
   8. Объяснить полученные результаты и указать на потенциальные ошибки.
2. 1. В первом терминале установите уровень изоляции READ COMMITTED.
   2. Во втором терминале также установить уровень изоляции REPEATABLE READ.
   3. В первом терминале добавить строку в таблицу, изменить данные в какой-нибудь строке и удалить какую-нибудь строку.
   4. Во втором терминале выполнить SELECT и проверить наличие изменений в таблице, выполненных в первой транзакции.
   5. В первом терминале завершить транзакцию – COMMIT.
   6. Во втором терминале выполнить SELECT и проверить наличие изменений в таблице, выполненных в первой транзакции.
   7. Во втором терминале завершить транзакцию.
   8. Объяснить полученные.

|  |  |
| --- | --- |
| id | amount |
| 1 | 100 |
| 2 | 200 |

1. Создать тестовую таблицу и добавит данные:  
   tab (id serial PRIMARY KEY, amount integer)

* 1. В первом терминале установите уровень изоляции REPEATABLE READ и

добавить строку, равную сумме amount:  
INSERT INTO tab (amount) VALUES((SELECT sum(amount) FROM tab));

* 1. Во втором терминале также установить уровень изоляции REPEATABLE READ и добавить строку, равную сумме amount:  
     INSERT INTO tab (amount) VALUES((SELECT sum(amount) FROM tab));
  2. В первом терминале завершить транзакцию – COMMIT.
  3. Во втором терминале завершить транзакцию – COMMIT.
  4. Вывести данные.
  5. Проделать то же самое для SERIALIZABLE.
  6. Объяснить результаты.

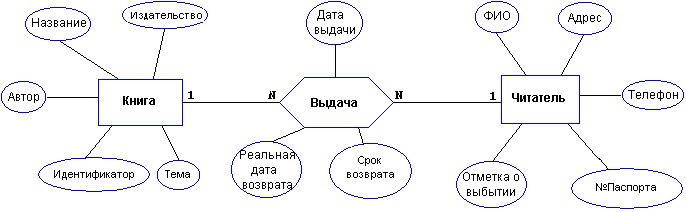
# 7. Список использованных источников

1. Фомин М.М. Реляционные базы данных. Учебное пособие для бакалавров <https://e-learning.bmstu.ru/iu6/mod/resource/view.php?id=6634>
2. Карпова И.П. Базы данных. Учебное пособие. – Московский государственный институт электроники и математики (Технический университет): учебное пособие– М., 2009. – 140-141 с, 102 c.
3. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. - СПб.: БХВ-Петербург, 2018. - 65-68 c, 68-72 с.

# 8. Приложение. Схемы предметных областей

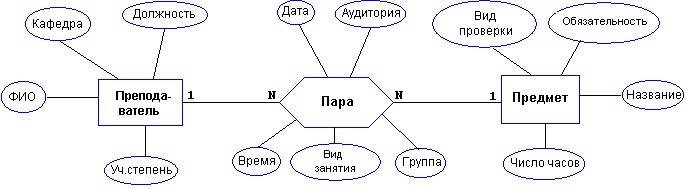
# Вариант №1

Предметная область для практических заданий: **Библиотека**



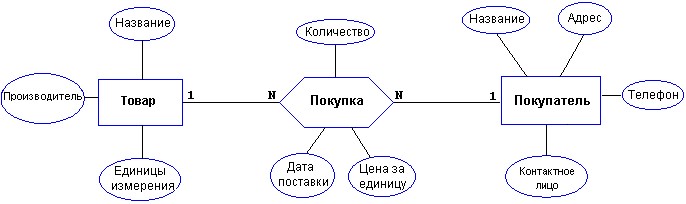
# Вариант №2

Предметная область для практических заданий: **Университет**



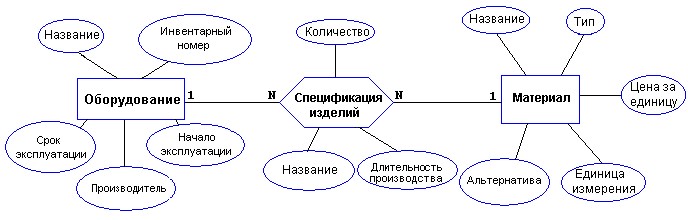
# Вариант №3

Предметная область для практических заданий: **Отдел продаж**



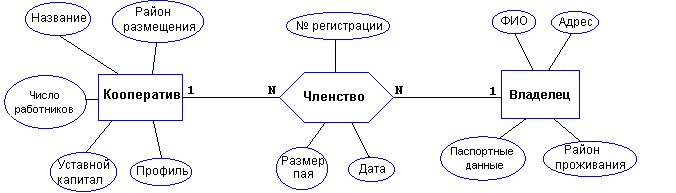
# Вариант №4

Предметная область для практических заданий: **Производство**



# Вариант №5

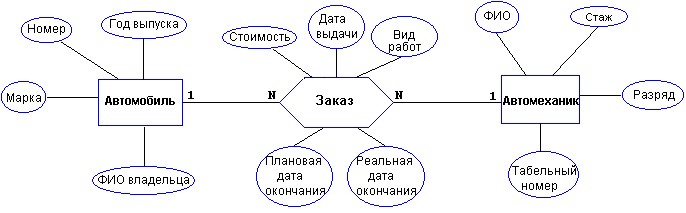
Предметная область для практических заданий: **Кооперативы**



Примечание: профиль - продуктовый, галантерейный, канцелярский и т.п.

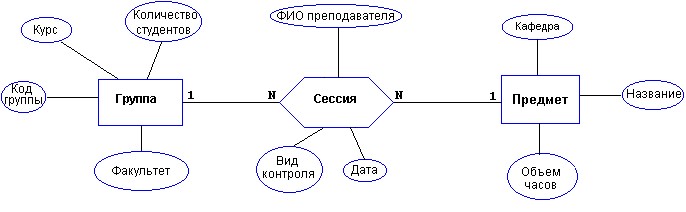
# Вариант №6

Предметная область для практических заданий: **Автомастерская**



# Вариант №7

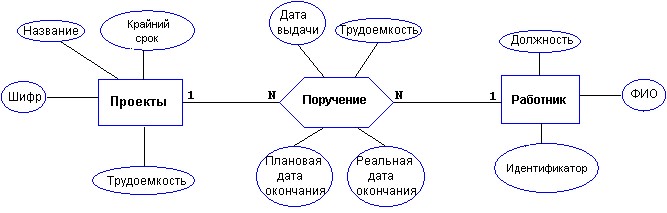
Предметная область для практических заданий: **Сессия**



Вид контроля - зачет, экзамен.

# Вариант №8

Предметная область для практических заданий: **Управление проектом**

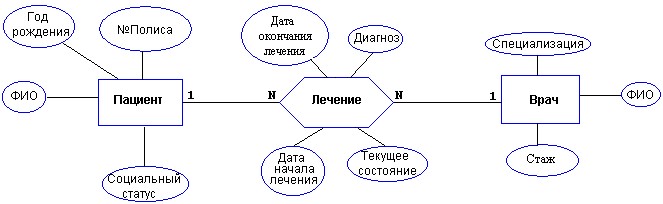


Категория дисциплины - гуманитарная, математическая, компьютерная и т.д.

Вид контроля - зачет, экзамен.

# Вариант №9

Предметная область для практических заданий: **Поликлиника**

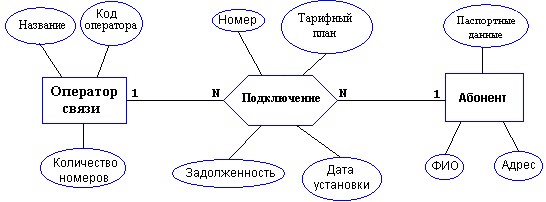


Текущее состояние - средней тяжести, тяжелое, направлен в стационар, умер. Социальный статус пациента - учащийся, работающий, врем. неработающий, инвалид, пенсионер

Специализация врача - терапевт, хирург и т.п.

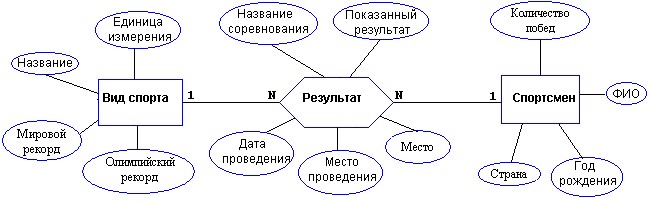
# Вариант №10

Предметная область для практических заданий: **Сотовая связь**



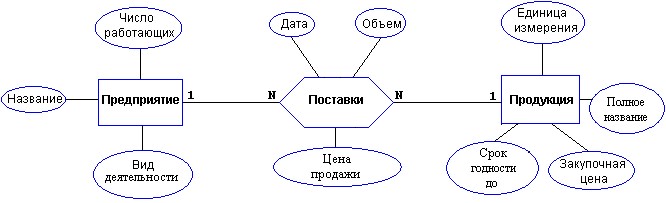
# Вариант № 11

Предметная область для практических заданий: **Спорт**



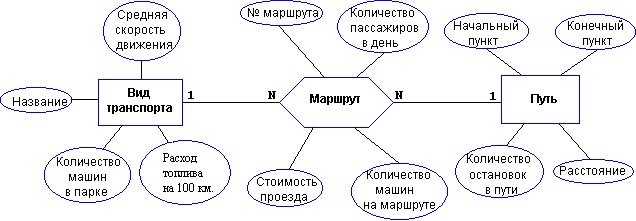
# Вариант №12

Предметная область для практических заданий: **Поставки**



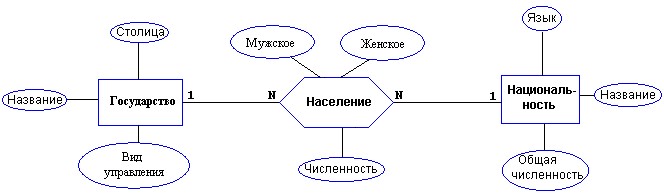
# Вариант №13

Предметная область для практических заданий: **Транспорт**



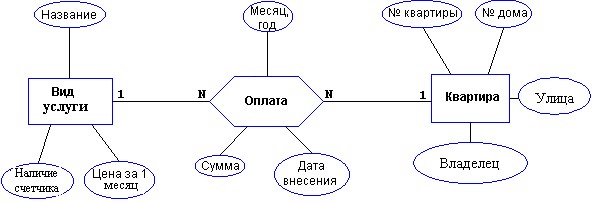
# Вариант № 14

Предметная область для практических заданий: **География**



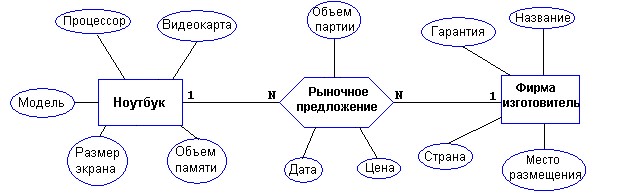
# Вариант №15

Предметная область для практических заданий: **Домоуправление**



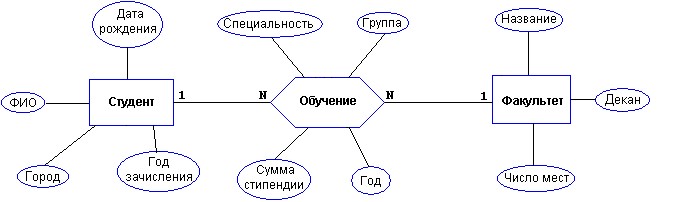
# Вариант №16

Предметная область для практических заданий: **Ноутбуки**



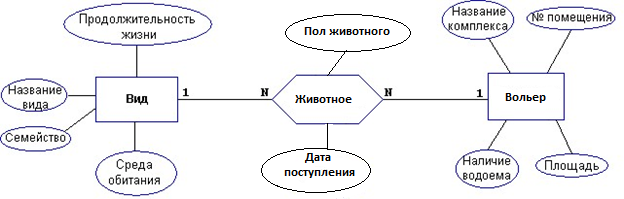
# Вариант №17

Предметная область для практических заданий: **Деканат**



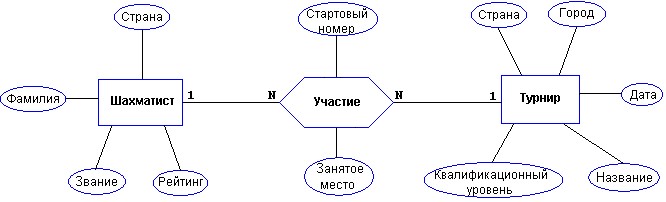
# Вариант №18

Предметная область для практических заданий: **Зоопарк**



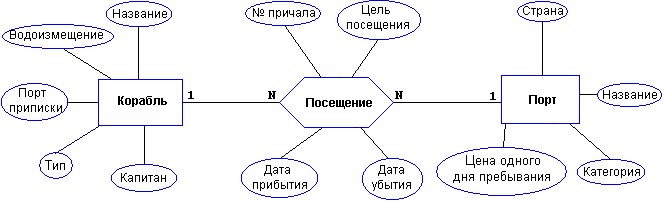
# Вариант №19

Предметная область для практических заданий: **Шахматы**



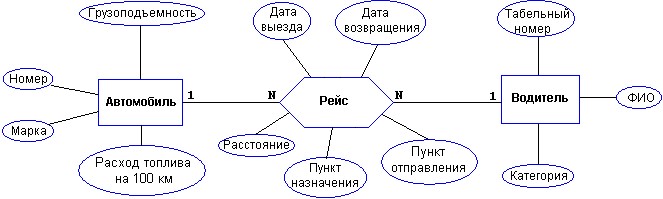
# Вариант №20

Предметная область для практических заданий: **Судоходство** 1



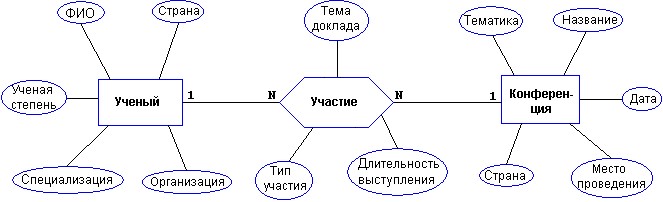
# Вариант №21

Предметная область для практических заданий: **Грузоперевозки**



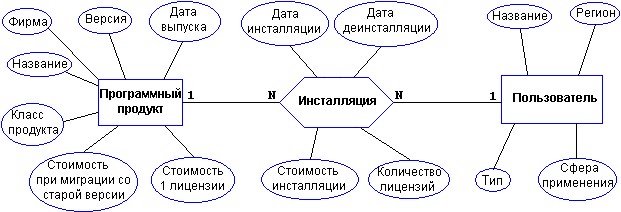
# Вариант №22

Предметная область для практических заданий: **Научные конференции**



# Вариант №23

Предметная область для практических заданий: **Программные продукты**

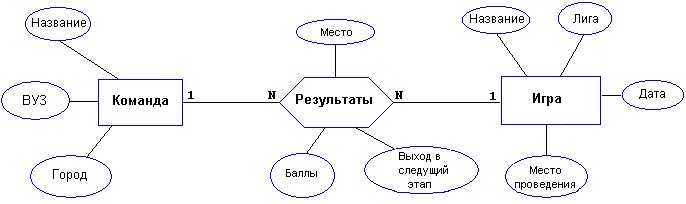


Класс: ОС, сервер приложений, СУБД, Web-сервер, система программирования и т.д.

Тип пользователя: индивидуальный, корпоративный, совместный, групповой и др.

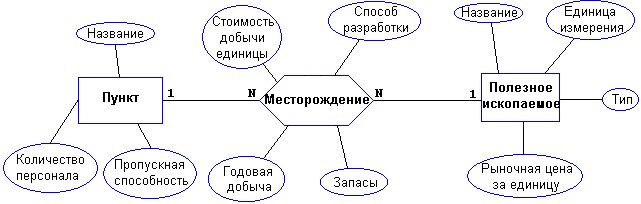
# Вариант №24

Предметная область для практических заданий: **КВН**



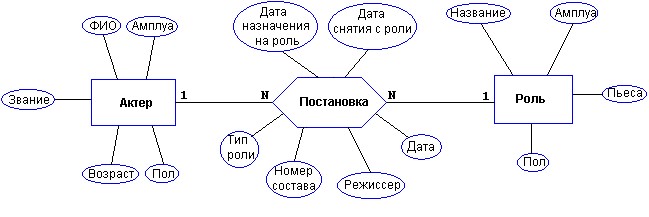
# Вариант №25

Предметная область для практических заданий: **Добыча ресурсов**



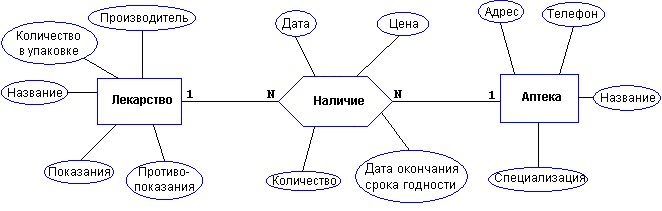
# Вариант №26

Предметная область для практических заданий: **Театр**



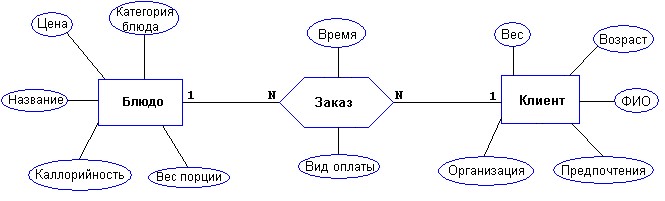
# Вариант №27

Предметная область для практических заданий: **Справочная аптек**



# Вариант №28

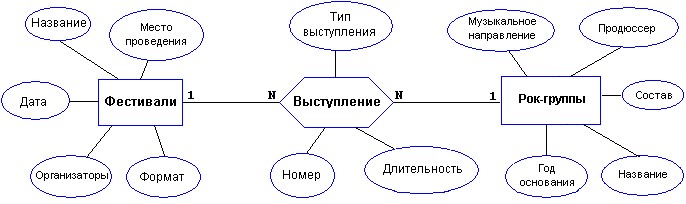
Предметная область для практических заданий: **Столовая**



Категория блюда: первое, второе, гарнир, десерт и т.д.

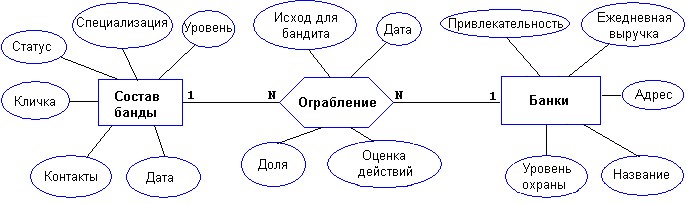
# Вариант №29

Предметная область для практических заданий: **Рок-группы**



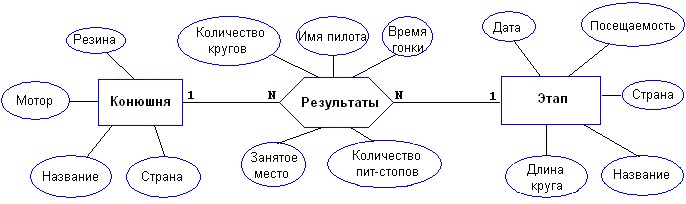
# Вариант №30

Предметная область для практических заданий: **ОПГ**



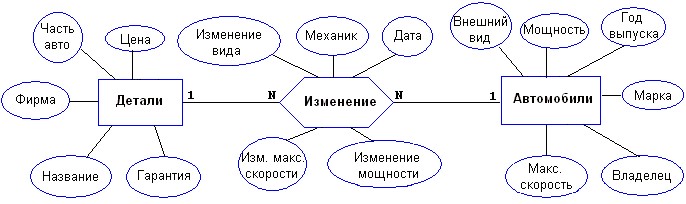
# Вариант №31

Предметная область для практических заданий: **Формула 1**



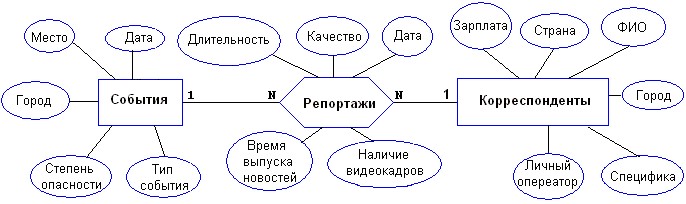
# Вариант №32

Предметная область для практических заданий: **Тюнинг**



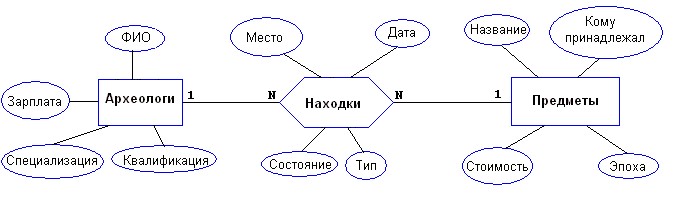
# Вариант №33

Предметная область для практических заданий: **Тележурналистика**



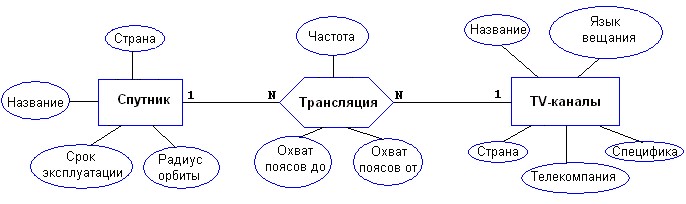
# Вариант №34

Предметная область для практических заданий: **Археология**



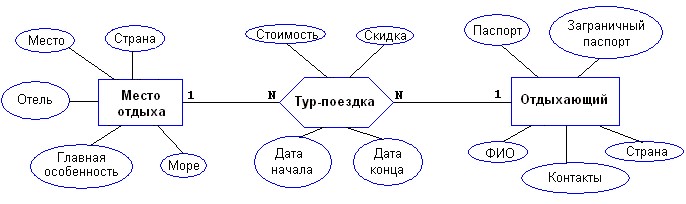
# Вариант №35

Предметная область для практических заданий: **Телевещание**



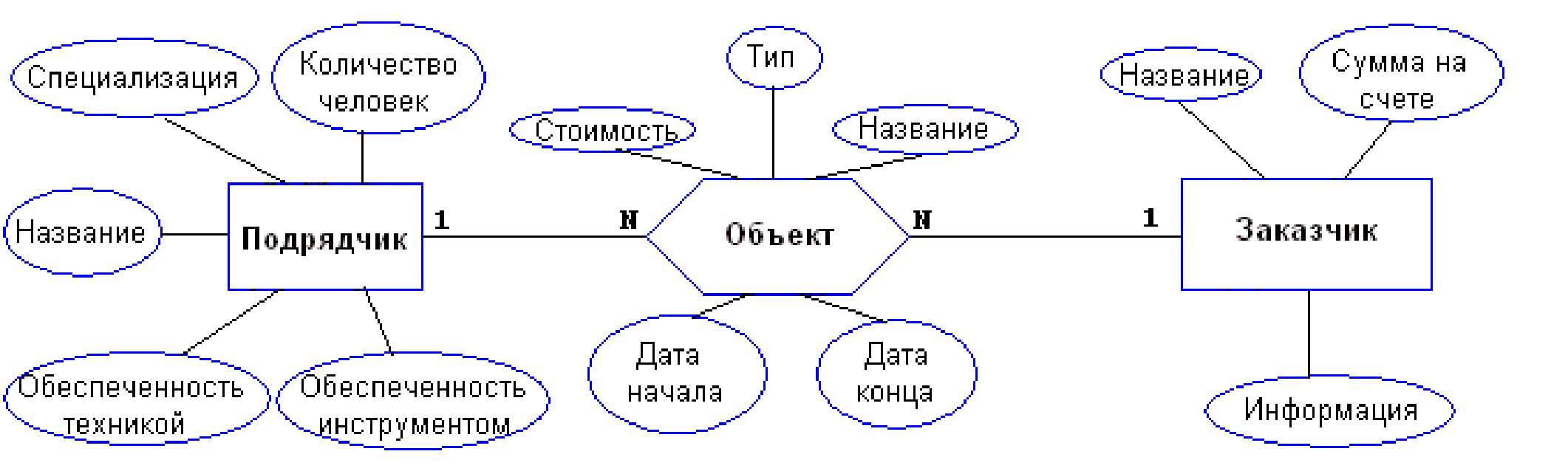
# Вариант №36

Предметная область для практических заданий: **Турфирма**



**Вариант №37**

Предметная область для практических заданий: **Строительство**



Тип объекта: промышленный, частный, специальный, хозяйственный и.т.д

# Вариант №38

Предметная область для практических заданий: **Искусство**

Тип произведения: скульптура, живопись, литье, графика и т.д.

Тип места хранения: частная коллекция, музей, галерея, неизвестно (тогда все остальные параметры пустые) и т.д.

1. В современных БД при соответствующих настройках связей между таблицами изменения первичного клича приводит к изменению внешнего ключа дочерней таблицы. Однако, пренебрегать временем, затраченным на выполнение этой операции тоже не следует. [↑](#footnote-ref-1)
2. Все занятия будут проходить в одном здании. [↑](#footnote-ref-2)
3. При установке PostgreSQL создается суперпользователь postgres. [↑](#footnote-ref-3)
4. Если при установке PostgreSQL был указан порт отличный от 5432, то в командной строке необходимо его указать – -**p nnnn.** [↑](#footnote-ref-4)
5. Как присвоить, полученное значение возвращаемому параметру мы рассмотрим при изучении функций. [↑](#footnote-ref-5)
6. Для PostgreSQL невозможно. [↑](#footnote-ref-6)