

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)



**МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ  
ПО КУРСУ «БАЗЫ ДАННЫХ»**

**Лабораторная работа №2  
«Создание БД для приложения»**

Авторы:  
Скворцова М.А., [magavrilova@bmstu.ru](mailto:magavrilova@bmstu.ru)  
Лапшин А.В.

Москва, 2022

## Общие сведения

### Сокращения

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

БД – База данных.

СУБД – Система управления базами данных.

РБД – Реляционная база данных.

РСУБД – Реляционная система управления базами данных.

НФ – нормальная форма.

ПК – первичный ключ.

ВК – внешний ключ.

Одним из основных применений РСУБД является хранение и обработка данных небольших пользовательских приложений. Специфика такого использования заключается в том, что данные постоянно добавляются/изменяются/удаляются. При этом данных сравнительно немного. Схема данных достаточно устойчива и редко изменяется. В этом случае непосредственно с СУБД взаимодействует не человек, а программа, что уменьшает требования к способу взаимодействия с базой данных. Однако разработчику необходимо уметь проектировать гибкую и эффективную схему данных, использовать ограничения целостности, манипулировать данными и понимать механизмы поддержки согласованности базы данных, такие как транзакции и триггеры.

Данная лабораторная работа призвана сформировать у студента понимание особенностей хранения данных приложения в РСУБД и умение – это хранение настраивать и поддерживать. Для более последовательного изучения этих знаний работа разделена на четыре части.

## **Лабораторная работа №2. Создание БД для приложения**

Данная лабораторная состоит из 4 взаимосвязанных частей. В каждой части есть теоретическая и практическая части лабораторной работы. В конце каждой части есть задания для выполнения в рамках лабораторной работы.

### **Цель:**

Данная лабораторная работа призвана сформировать у студента понимание особенностей хранения данных приложения в РСУБД, а также настройка и поддержка хранения данных.

### **Задачи:**

- Получить теоретические знания по концептуальным картам.
- Ознакомится с нормализацией в БД.
- Изучение типов связей.
- Ознакомится с DDL операторами.
- Изучение типов данных.
- Научится добавлять записи в таблицы.
- Научиться удалять и изменять записи в таблице.
- Научиться контролю целостности данных.
- Ознакомиться с механизмами контроля, транзакциями и триггерами.

# Часть 1. Проектирование схемы базы данных

## Теоретическая часть

### Модель «сущность – связь»

Модель «сущность – связь» основана на диаграммной технике. Для представления различных аспектов структуры данных (объектов, свойств объектов, связей между объектами, свойств связей и других) используются графические средства. Что бы их представлять были созданы нотации бизнес-процессов, представляющий собой совокупность графических объектов, используемых при моделировании, а также правил моделирования. Одна из таких, IDEF0 – методология функционального моделирования и графическая нотация, предназначенная для формализации и описания бизнес-процессов.

Для модели «сущность – связь» базовыми являются понятия:

- Сущность.
- Связь.
- Атрибут.

С первым и последним уже знакомы, а вот что такое связь разберем далее.

### Ключи

Перед тем как приступить к понятию «связей» нужно узнать о потенциальных и альтернативных, а также первичных и внешних ключах, так как они дадут четкое понимание как строятся связи в БД. **Потенциальный ключ** – это комбинация атрибутов таблицы, позволяющая уникальным образом идентифицировать строки в ней. Ключи нужны для адресации на уровне строк (записей). При наличии в таблице более одного потенциального ключа один из них выбирается в качестве так называемого **первичного ключа**, а остальные будут являться **альтернативными ключами**. Для внешнего ключа рассмотрим, к примеру таблицы «Студенты» и «Успеваемость». Предположим, что в таблице «Студенты» нет строки с

номером зачетной книжки 55900, тогда включать строку с таким номером зачетной книжки в таблицу «Успеваемость» не имеет смысла. Таким образом, значения столбца «Номер зачетной книжки» в таблице «Успеваемость» должны быть согласованы со значениями такого же столбца в таблице «Студенты». Атрибут «Номер зачетной книжки» в таблице «Успеваемость» является примером того, что называется **внешним ключом**.

### **Связи и их типы**

Связи осуществляют более важную роль, чем просто информация размещения данных по таблицам. Прежде они требуются разработчикам для сопровождения цельности баз данных. Корректно настроив связи, можно быть уверенным, что ничего не потеряется.

Связь не позволит удалить группу, пока она имеется во внешних ключах других таблиц. Для начала прежде, чем удалить ненужную запись, следовало установить сотрудников в другие новые группы или имеющиеся. Поэтому связи называют еще ограничениями. Представьте, что Вы решили удалить одну из групп в таблице учебной базы данных. Если бы связи не было, то для тех сотрудников, которые к ней были определены, остался идентификатор несуществующей группы.

При проектировании информационной системы данные обычно размещают в нескольких таблицах. Таблицы при этом связывают с семантикой информации. В реляционной СУБД для указания связей в таблице производят операции их связывания. Рассмотрим наиболее часто встречаемые бинарные связи:

- **Связи вида (1:1)** образуется в случае, когда все поля записи основной таблицы и дополнительной таблицы являются ключевыми.
- **Связь (1:M)** в случае, когда одной информации главной таблицы соответствует несколько записей дополнительной таблицы.
- **Связь (M:1)** может быть, когда нескольким атрибутам основной таблицы ставится в соответствии одна запись дополнительной.

- **Связь (М:М)** возникает в том случае, когда нескольким записям основной таблицы соответствует несколько записей вспомогательной. В реляционной базе данных связь (М:М) реализуется через вспомогательные таблицы.

### **Нормальные формы и нормализация**

Процесс проектирования БД с использованием метода НФ является итерационным и заключается в последовательном переводе отношения из 1НФ в НФ более высокого порядка по определенным правилам. Каждая следующая НФ ограничивается определенным типом функциональных зависимостей и устранением соответствующих аномалий при выполнении операций над отношениями БД, а также сохранении свойств предшествующих НФ. Чтобы рассмотреть их стоит ознакомиться ещё с парочкой терминов. Домен атрибута – множество допустимых значений, которые может принимать атрибут. Скалярное значение в БД – это одно значение (одна строка, один столбец) необходимого типа данных. Существуют семь нормальных форм. Строго говоря, база данных считается нормализованной, если к ней применяется третья нормальная форма и выше. Собственно, и рассмотрим три первых формы.

#### **1НФ**

Отношение находится в 1НФ, если все его атрибуты являются простыми, все используемые домены должны содержать только скалярные значения. Не должно быть повторений строк в таблице. Например, есть таблица «Автомобили»:

Таблица 1 – Нарушение 1НФ

Фирма	Модели
BMW	M5, X5M, M1
Nissan	GT-R

Нарушение нормализации 1НФ происходит в моделях BMW, т.к. в одной ячейке содержится список из 3 элементов: M5, X5M, M1, т.е. он не является атомарным. Преобразуем таблицу к 1НФ:

Таблица 2 – Преобразование к 1НФ

Фирма	Модели
BMW	M5
BMW	X5M
BMW	M1
Nissan	GT-R

## 2НФ

Отношение находится во 2НФ, если оно находится в 1НФ и каждый не ключевой атрибут неприводимо зависит от Первичного Ключа (ПК). Неприводимость означает, что в составе потенциального ключа отсутствует меньшее подмножество атрибутов, от которого можно также вывести данную функциональную зависимость. Например, дана таблица:

Таблица 3 – Нарушение 2НФ

Модель	Фирма	Цена	Скидка
M5	BMW	5500000	5%
X5M	BMW	6000000	5%
M1	BMW	2500000	5%
GT-R	Nissan	5000000	10%

Таблица находится в первой нормальной форме, но не во второй. Цена машины зависит от модели и фирмы. Скидка зависят от фирмы, то есть зависимость от первичного ключа неполная. Исправляется это путем

декомпозиции на два отношения (таблицы 4 и 5), в которых не ключевые атрибуты зависят от ПК.

Таблица 4 – Преобразование к 2НФ (1)

Модель	Фирма	Цена
M5	BMW	5500000
X5M	BMW	6000000
M1	BMW	2500000
GT-R	Nissan	5000000

Таблица 5 – Преобразование к 2НФ (2)

Фирма	Скидка
BMW	5%
Nissan	10%

### **3НФ**

Отношение находится в 3НФ, когда находится во 2НФ и каждый не ключевой атрибут не транзитивно зависит от первичного ключа. Проще говоря, второе правило требует выносить все не ключевые поля, содержимое которых может относиться к нескольким записям таблицы в отдельные таблицы. Рассмотрим таблицу:

Таблица 6 – Нарушение 3НФ

Модель	Магазин	Телефон
BMW	Риал-авто	87-33-98
Audi	Риал-авто	87-33-98
Nissan	Некст-Авто	94-54-12



Таблица 6 находится во 2НФ, но не в 3НФ. В отношении атрибут «Модель» является первичным ключом. Личных телефонов у автомобилей нет, и телефон зависит исключительно от магазина. Таким образом, в отношении существуют следующие функциональные зависимости:

- Модель → Магазин.
- Магазин → Телефон.
- Модель → Телефон.

Зависимость Модель → Телефон является транзитивной, следовательно, отношение не находится в 3НФ. В результате разделения исходного отношения получаются два отношения, находящиеся в 3НФ:

Таблица 7 – Преобразование к 3НФ (2)

Магазин	Телефон
Риал-авто	87-33-98
Некст-Авто	94-54-12

Таблица 8 – Преобразование к 3НФ (2)

Модель	Магазин
BMW	Риал-авто
Audi	Риал-авто
Nissan	Некст-Авто

## Разбор базы данных «Аэропорт»

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. Добавим еще одну важную деталь: каждый аэропорт характеризуется географическими координатами – долготой и широтой, а также часовым поясом. Формируются маршруты перелетов между городами. Конечно, каждый такой маршрут требует указания не только города, но и аэропорта, поскольку, как мы уже сказали, в городе может быть и более одного аэропорта. В качестве упрощения реальности мы решим, что маршруты не будут иметь промежуточных посадок, т. е. у них будет только аэропорт отправления и аэропорт назначения. Каждый маршрут имеет шестизначный номер, включающий цифры и буквы латинского алфавита. На основе перечня маршрутов формируется расписание полетов (или рейсов). В расписании указывается 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). Теперь обратимся к пассажирам. 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 тринадцатизначного 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 перелета. 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. Где связи обозначены в виде стрелок, которые означают виды как на рисунке 1.







НОТАЦИЯ	ОБОЗНАЧЕНИЕ СВЯЗИ
IDEF1	 Много
	 Один
IE	 Много
	 Один
Любая	 Необязательная
	 Обязательная

Рисунок 1 – Обозначение связей

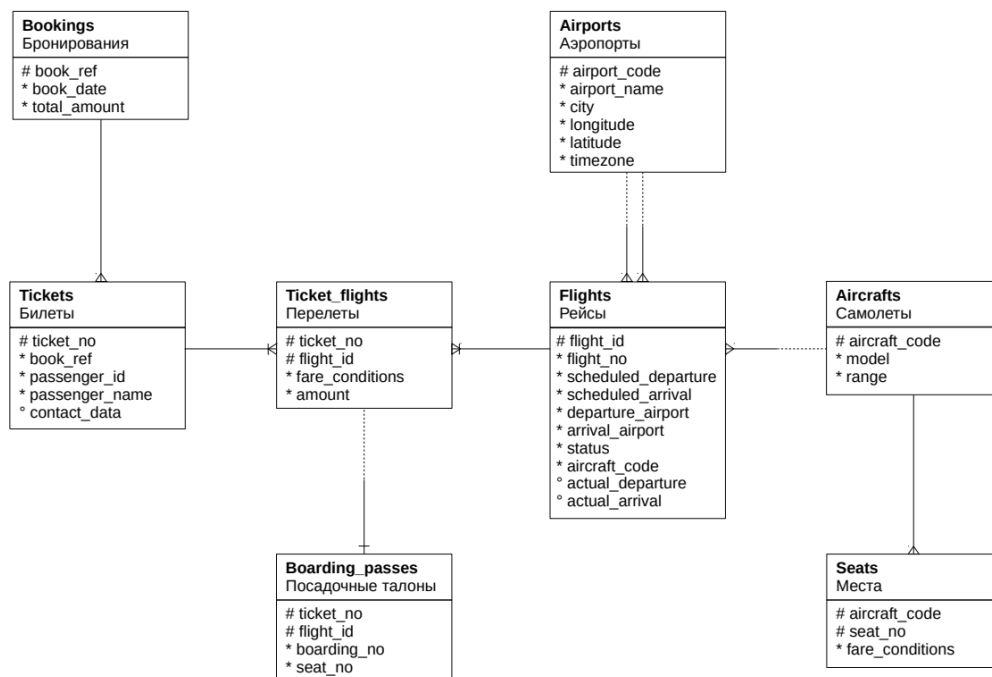


Рисунок 2 – Схема данных

## **Практическое задание для первой части ЛР**

Второе практическое задание связано с проектированием схемы базы данных для работы приложения (WEB/Mobile/Desktop). Каждый индивидуальный вариант содержит предметную область, из которой должна быть проектируемая база данных. К данной предметной области необходимо добавить не менее 2-х дополнительных таблиц (сущностей), необходимых для детального решения поставленной задачи. Задачей студента является решить, для чего будет использоваться создаваемая база данных, и, исходя из этого, построить её концептуальную схему. Результатом данной части лабораторной работы является схема базы данных (в виде ER-диаграммы, содержащей таблицы и связи между ними, с уточнением типов столбцов, с описанием внешних и первичных ключей). При сдаче задания студент должен обосновать соответствие созданной схемы поставленной задаче.

Для проектирования схемы и построения диаграммы можно использовать любые средства, один из вариантов использовать сайт:

<https://www.lucidchart.com/pages/examples/er-diagram-tool>

### **Требования к схеме**

- Схема должна соответствовать поставленной задаче.
- Связи между сущностями должны быть правильно смоделированы.
- Таблицы должны удовлетворять, по крайней мере, третьей нормальной форме.
- Желательно придерживаться какой-либо системы в именовании таблиц и столбцов.

### **Темы для самостоятельной проработки**

- Модель «сущность-связь» (ER-модель).
- Первичные и внешние ключи.
- Типы связей и их моделирование.
- Нормальные формы и нормализация.

## **Вопросы для самостоятельной проработки**

- Понятие первичного и вторичного ключей.
- Какие типы связей существуют?
- Что такое нормализация? Каковы её цели?
- Сколько известно нормальных форм? По какому принципу они строятся?
- В чем суть первой нормальной формы?
- В чем суть второй нормальной формы?
- В чем суть третьей нормальной формы?

## Часть 2. Создание и заполнения таблиц.

### Теоретическая часть и практические примеры

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

#### CREATE TABLE

SQL оператор CREATE TABLE позволяет создавать и определять таблицу. Чтобы сократить последующие изменения, стоит заранее продумать структуру таблицы и ее содержимое. Наиболее важные пункты:

- Названия таблиц и столбцов.
- Типы данных столбцов.
- Атрибуты и ограничения.

Реляционные базы данных хранят данные в таблицах, и каждая таблица содержит набор столбцов. У столбца есть название и тип данных, так же можно указать атрибуты и ограничения, такие параметры как NOT NULL (то есть обязательное поле для заполнения) или же PRIMARY KEY (первичный ключ), так же к примеру можно поставить ограничение с помощью ключевого слова CHECK (amount < 3000). Более понятно становится на примерах, поэтому для демонстрации сделаем копию таблицы airports с помощью запроса SQL. При написании запроса стоит учитывать правила имен, они звучат так:

- Начинаться с буквы
- Иметь длину 1–30 символов
- Содержать только A–Z, a–z, 0–9, \_, \$ и #
- Не дублировать имя другого объекта, принадлежащего тому же самому пользователю

## Запрос:

```
CREATE TABLE airports_copy (  
    airport_code char( 3 ) NOT NULL, -- Код аэропорта  
    airport_name text NOT NULL, -- Название аэропорта  
    city text NOT NULL, -- Город  
    longitude float NOT NULL, -- Координаты аэропорта: долгота  
    latitude float NOT NULL, -- Координаты аэропорта: широта  
    timezone text NOT NULL, -- Часовой пояс аэропорта  
    PRIMARY KEY ( airport_code )  
);
```

### Последовательность запроса такова:

- С начало нужно объявить команду для создания таблицы с помощью ключевого слова CREATE TABLE, а после пишем название будущей таблицы и открываем круглые скобки, в которых будем описывать столбцы.

- В скобках для каждого столбца нужно указать два обязательных параметра, это название и тип данных. В нашем случае ещё дописывается команда NOT NULL (её предназначение описывалось выше), так как все поля аэропорта должны быть заполнены.

- После тогда, когда наполнили таблицу будущими столбцами последним значением указываем первичный ключ с помощью ключевого слова PRIMARY KEY и указанием в скобках столбца для назначения его ключом.

- С помощью двух тире «--» можно оставлять комментарии (то есть часть кода, которая не будет прочитана транслятором)

Как это выглядит в самой среде можно посмотреть рисунке 3.



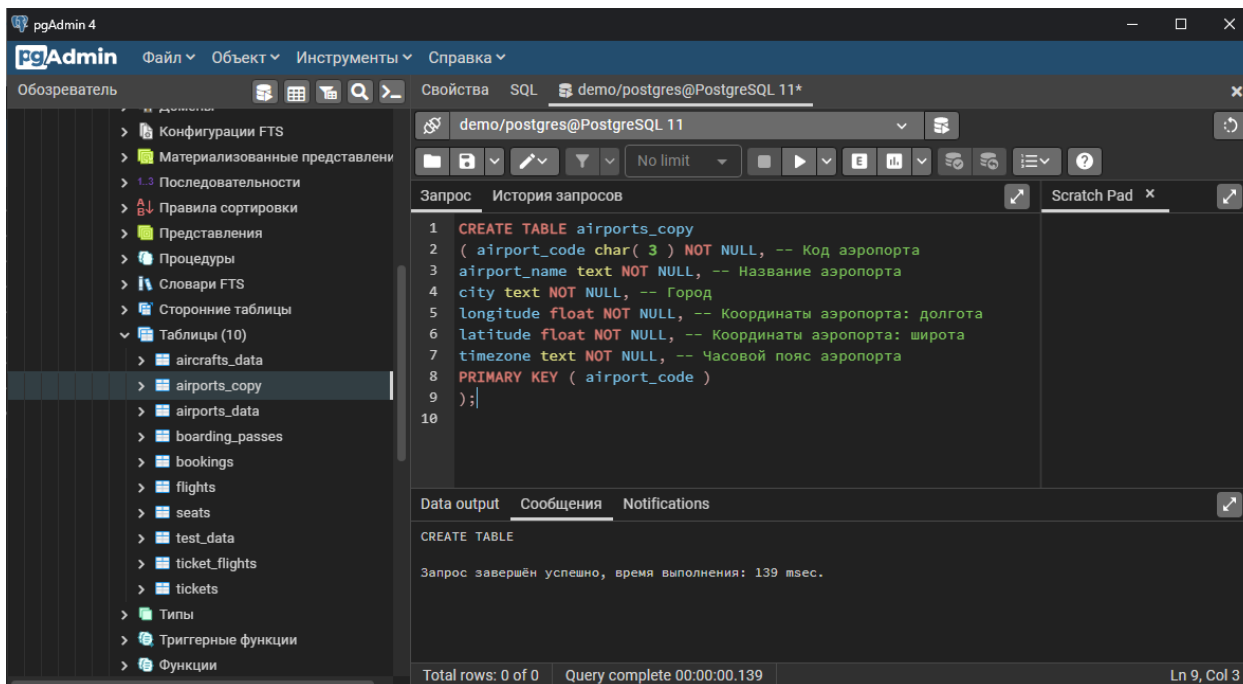


Рисунок 3 – Запрос №1

## ALTER TABLE

Если же все-таки произошло так что нужно исправить структуру таблицы, то для этого нужно будет воспользоваться специальным оператором ALTER TABLE. Используется, собственно, для добавления, изменения или удаления столбца в таблице. Оператор ALTER TABLE также используется для переименования таблиц.

Эта команда очень многообразна и логична. Она предусматривает, наверное, все ситуации, которые могут возникнуть в реальной работе. Например, может возникнуть необходимость добавить новый столбец в таблицу – команда ALTER TABLE имеет для этого фразу ADD COLUMN. Возможна и обратная ситуация, когда нужно удалить столбец из таблицы – для этого есть фраза DROP COLUMN. Если нужно добавить ограничение, то помогут фразы ADD CHECK и ADD CONSTRAINT. Если потребовался внешний ключ, то можно добавить и его.

Предположим, что нам понадобилось иметь в базе данных сведения о крейсерской скорости полета всех моделей самолетов, которые эксплуатируются в нашей авиакомпании. Следовательно, необходимо

добавить столбец в таблицу «Самолеты» (aircrafts). Дадим ему имя speed (наверное, можно предложить и более длинное имя — cruise\_speed). Тип данных для этого столбца выберем integer, добавим ограничение NOT NULL. Наложим и ограничение на минимальное значение крейсерской скорости, выраженное в километрах в час: CHECK( speed >= 300 ). В результате у каждой записи должен появиться столбец speed. Так же нам нужно чтобы в будущем нельзя было оставить это поля пустым, то есть надо добавить команду NOT NULL, но из этого образуется проблема «ОШИБКА: столбец "speed" содержит значения NULL» это происходит из-за того, что добавленные до этого записи в таблицу не соответствуют новому столбцу.

### **Запрос:**

```
ALTER TABLE airports_data ADD COLUMN speed integer CHECK (
speed >= 300 );
```

### **Последовательность запроса такова:**

- Сначала пишется ключевое слово ALTER TABLE и после название таблицы, в которой планируются изменения.
- Далее пишется операция с таблицей ADD COLUMN и после описывается столбец, который нужно добавить, синтаксис как при создании таблицы (название и тип данных)
- После главных параметров столбца в данном запросе пишется ограничитель с помощью CHECK и в скобках пишется условие.

Как запрос в среде pgAdmin можно посмотреть рисунке 4.

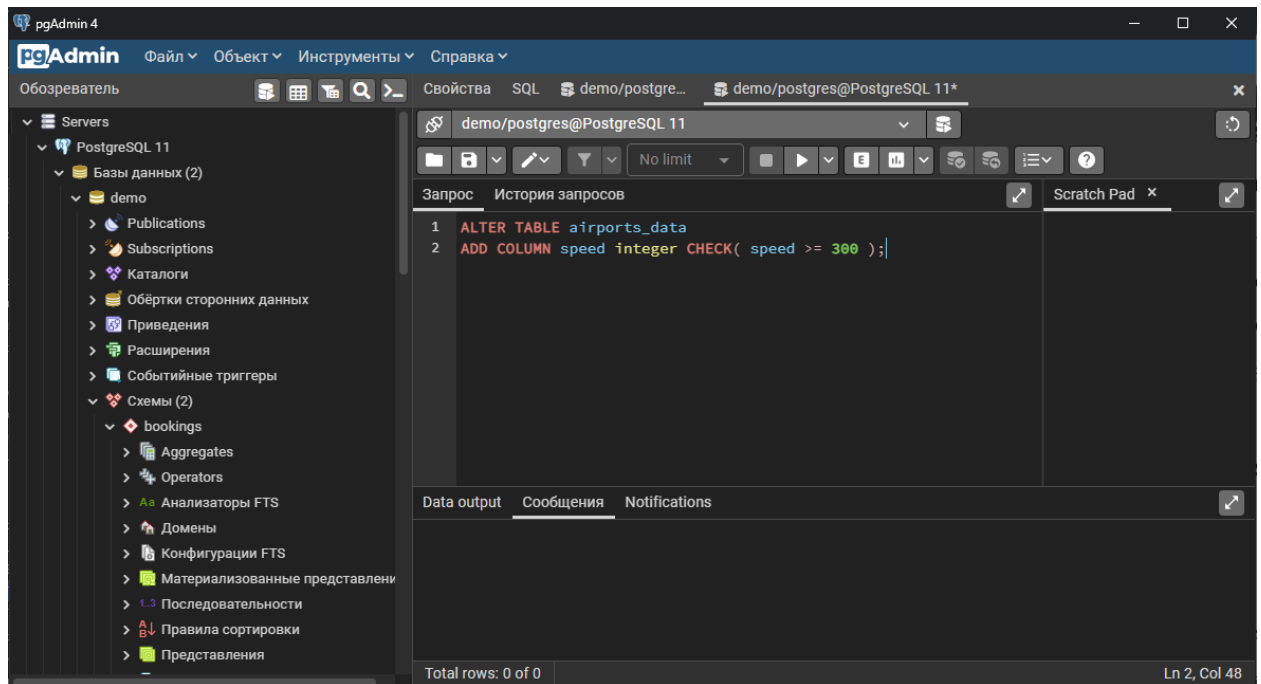


Рисунок 4 – Запрос №2

## INSERT

Как и описывалось выше команда INSERT используется для того, чтобы вставлять новые записи в таблицы. К примеру, у нас есть задача добавить новый самолет в таблицу `aircrafts_data`. В запросе «случайно» добавим опечатку в первый запрос, и так же «случайно» добавим тот же самолет, но с другим кодом. Это пригодится для будущих запросов.

### Запрос:

```
INSERT INTO aircrafts_data
VALUES
('152', '{ "en": "Ti2-153N", "ru": "Ту-154М" }', 6601),
('153', '{ "en": "Ti2-153N", "ru": "Ту-154М" }', 6601);
```

### Последовательность запроса такова:

- С помощью INSERT INTO выбираем таблицу, в которую будем помещать данные
- С помощью команды VALUES и используя синтаксис языка вводим данные полей записи (номер самолета, json объект с русской версией названия

самолета и английской, дальность полёта). JSON объект стоит описывать в одинарных кавычках и внутри них ещё уровень из скобок.

Как запрос в среде pgAdmin можно посмотреть рисунке 5.

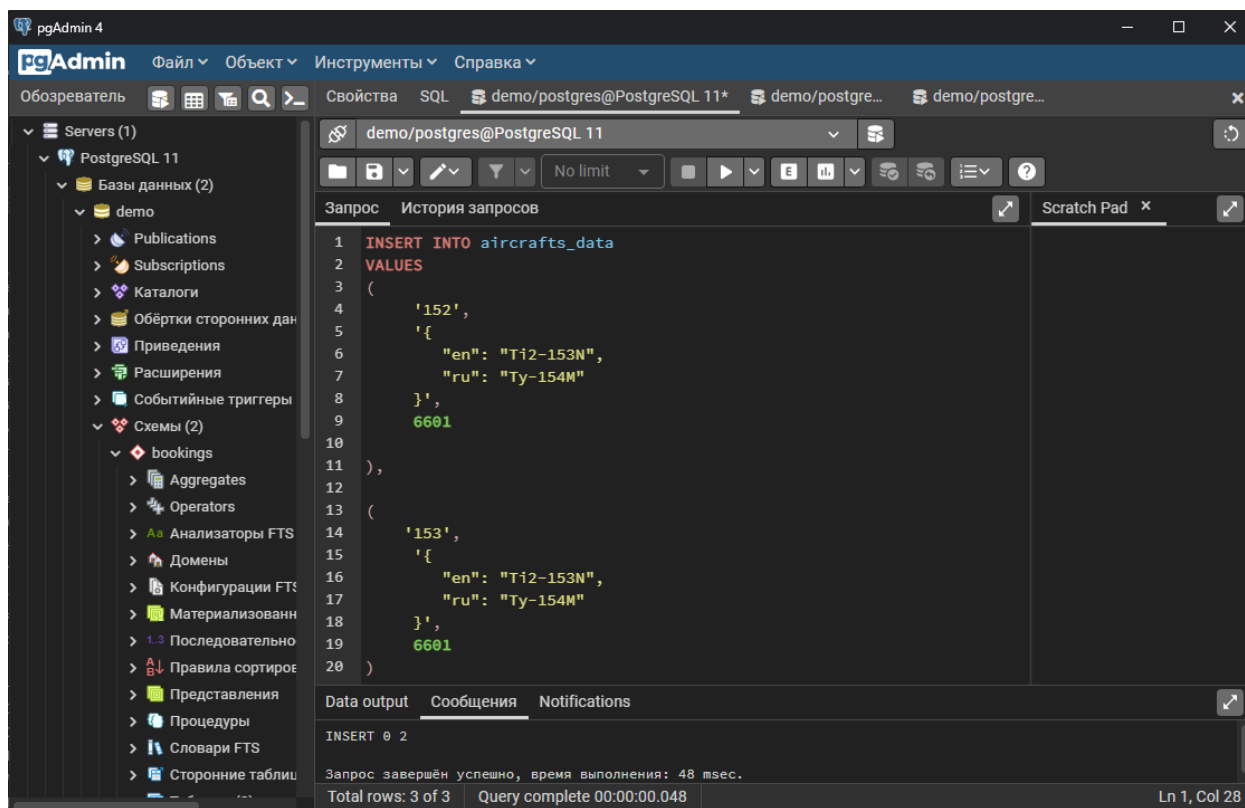


Рисунок 5 – Запрос №3

## ОГРАНИЧЕНИЯ

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Однако для многих приложений такие ограничения слишком грубые. Например, столбец, содержащий цену продукта, должен, вероятно, принимать только положительные значения. Но такого стандартного типа данных нет. Возможно, вы также захотите ограничить данные столбца по отношению к другим столбцам или строкам. Например, в таблице с информацией о товаре должна быть только одна строка с определённым кодом товара.

Для решения подобных задач SQL позволяет вам определять ограничения для столбцов и таблиц. Ограничения дают вам возможность управлять данными в таблицах так, как вы захотите. Если пользователь

попытается сохранить в столбце значение, нарушающее ограничения, возникнет ошибка. Ограничения будут действовать, даже если это значение по умолчанию.

### **Ограничения-проверки (CHECK)**

Ограничение-проверка состоит из ключевого слова CHECK, за которым идёт выражение в скобках. Это выражение должно включать столбец, для которого задаётся ограничение, иначе оно не имеет большого смысла.

### **Ограничения NOT NULL**

Ограничение NOT NULL просто указывает, что столбцу нельзя присваивать значение NULL. (По умолчанию к столбцам применяется просто NULL, то есть значение, может быть, не заполнено)

### **Ограничения уникальности (UNIQUE)**

Ограничения уникальности гарантируют, что данные в определённом столбце или группе столбцов уникальны среди всех строк таблицы.

### **Первичные ключи (PRIMARY KEY)**

Ограничение первичного ключа означает, что образующий его столбец или группа столбцов может быть уникальным идентификатором строк в таблице.

### **Внешние ключи (REFERENCES)**

Ограничение внешнего ключа указывает, что значения столбца (или группы столбцов) должны соответствовать значениям в некоторой строке другой таблицы.

### **Ограничения-исключения (EXCLUDE USING)**

Ограничения-исключения гарантируют, что при сравнении любых двух строк по указанным столбцам или выражениям с помощью заданных операторов, минимум одно из этих сравнений возвратит false или NULL.

## Практическое задание для второй части ЛР

Третье практическое задание заключается в подготовке SQL-скрипта для создания таблиц согласно схеме, полученной в предыдущем задании (с уточнением типов столбцов). Необходимо определить первичные и внешние ключи, а также декларативные ограничения целостности (возможность принимать неопределенное значение, уникальные ключи, проверочные ограничения и т. д.). Таблицы следует создавать в отдельной базе данных. Кроме того, нужно подготовить данные для заполнения созданных таблиц. Кроме того, нужно подготовить данные для заполнения созданных таблиц. Объем подготовленных данных должен составлять не менее 10 экземпляров для каждой из стержневых сущностей и 1000 экземпляров для целевой сущности. На основе этих данных необходимо создать SQL-скрипт для вставки соответствующих строк в таблицы БД.

### Темы для самостоятельной проработки

- Язык DDL, операторы CREATE TABLE и ALTER TABLE.

<https://postgrespro.ru/docs/postgrespro/11/ddl-basics>

<https://postgrespro.ru/docs/postgrespro/11/ddl-default>

<https://postgrespro.ru/docs/postgrespro/11/ddl-alter>

- Типы данных.

<https://postgrespro.ru/docs/postgrespro/11/datatype>

- Декларативные ограничения целостности.

<https://postgrespro.ru/docs/postgrespro/11/ddl-constraints>

- Оператор INSERT.

<https://postgrespro.ru/docs/postgrespro/11/dml-insert>

<https://postgrespro.ru/docs/postgrespro/11/dml-returning>

- Полное описание синтаксиса встретившихся команд

<https://postgrespro.ru/docs/postgrespro/11/sql-commands>

## Вопросы для самостоятельного изучения

- Объяснить, что делают написанные запросы.
- В чем различие типов CHAR и VARCHAR? VARCHAR и TEXT?
- Что такое внешний ключ?
- Какие существуют способы поддержания ссылочной целостности?
- Что такое уникальный ключ?
- Что такое SERIAL?
- Рассказать о значениях по умолчанию и неопределенных значениях.
- Как можно хранить даты и время?
- Рассказать о числовых типах данных.
- Каким образом можно вставить несколько строк с помощью одного оператора INSERT?

## Часть 3. Операторы манипулирования

### Теоретическая часть и практические примеры

#### DELETE

Так как в процессе предыдущего запроса с INSERT у нас «случайно» появилась лишняя запись, то её нужно удалить.

#### Запрос:

```
DELETE FROM aircrafts_data WHERE aircraft_code = '153';
```

#### Последовательность запроса такова:

- С помощью DELETE указываем последующие действия (удаление)
- Далее указываем с каким элементом нужно взаимодействовать (таблица aircraft\_code)
- Конструкция WHERE здесь нужна для того, чтобы удалить конкретный элемент из таблицы (самолет с номером = 153), иначе удалятся все записи в таблице

Как запрос в среде pgAdmin можно посмотреть рисунке 6.

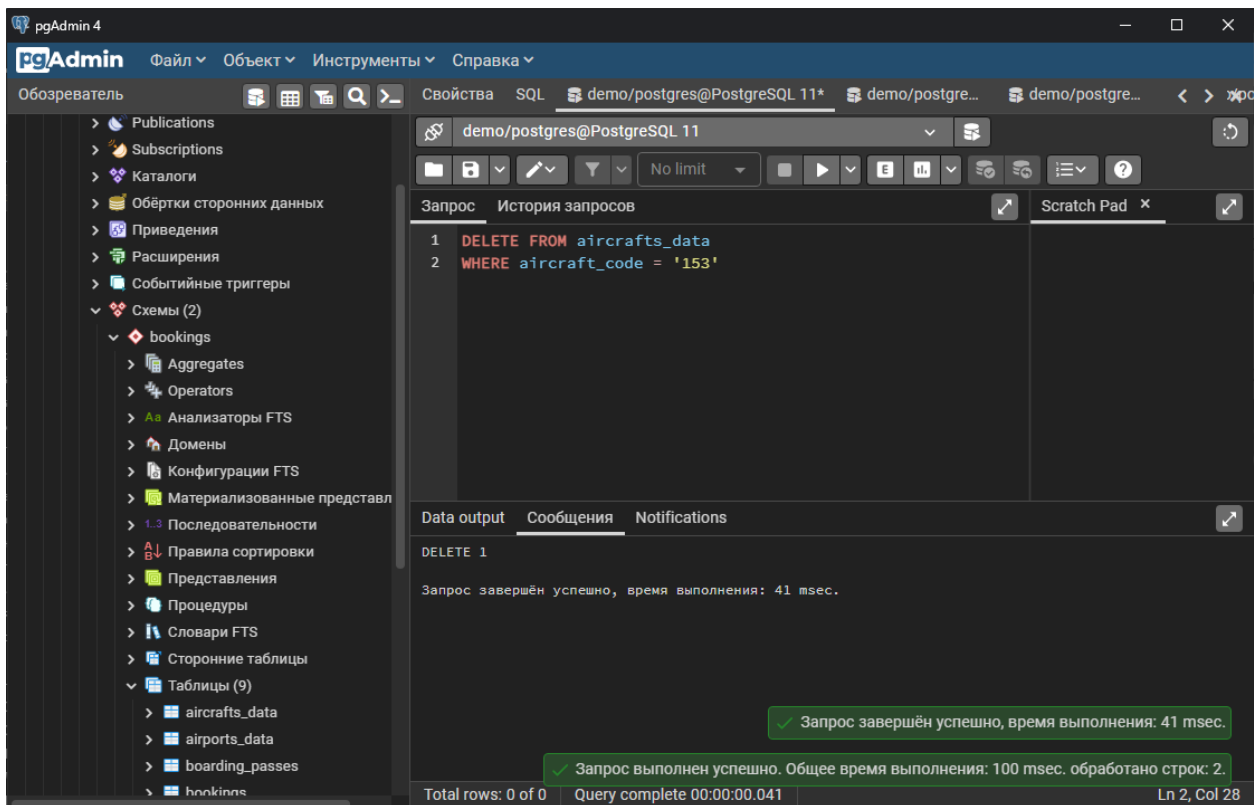


Рисунок 6 – Запрос №3

## UPDATE

SQL оператор UPDATE используется для обновления существующих записей в таблицах. Воспользуемся его возможностями чтобы исправить английское название и дальность полета самолета под номер «152».

### Запрос:

```
UPDATE aircrafts_data SET "range" = 6600, model = '{ "en": "Tu-154M", "ru": "Ту-154M" }' WHERE aircraft_code = '152';
```

### Последовательность запроса такова:

- Сначала пишется ключевое слово UPDATE и указывается имя таблицы, в которой нужно изменить записи
- Далее пишется операция SET и после через запятую название столбца и его новое значение
- В конце нужно описать фильтр с помощью WHERE для того, чтобы изменить значение у конкретной записи, а именно у самолета с номером «152»



Как запрос в среде pgAdmin можно посмотреть рисунке 7.

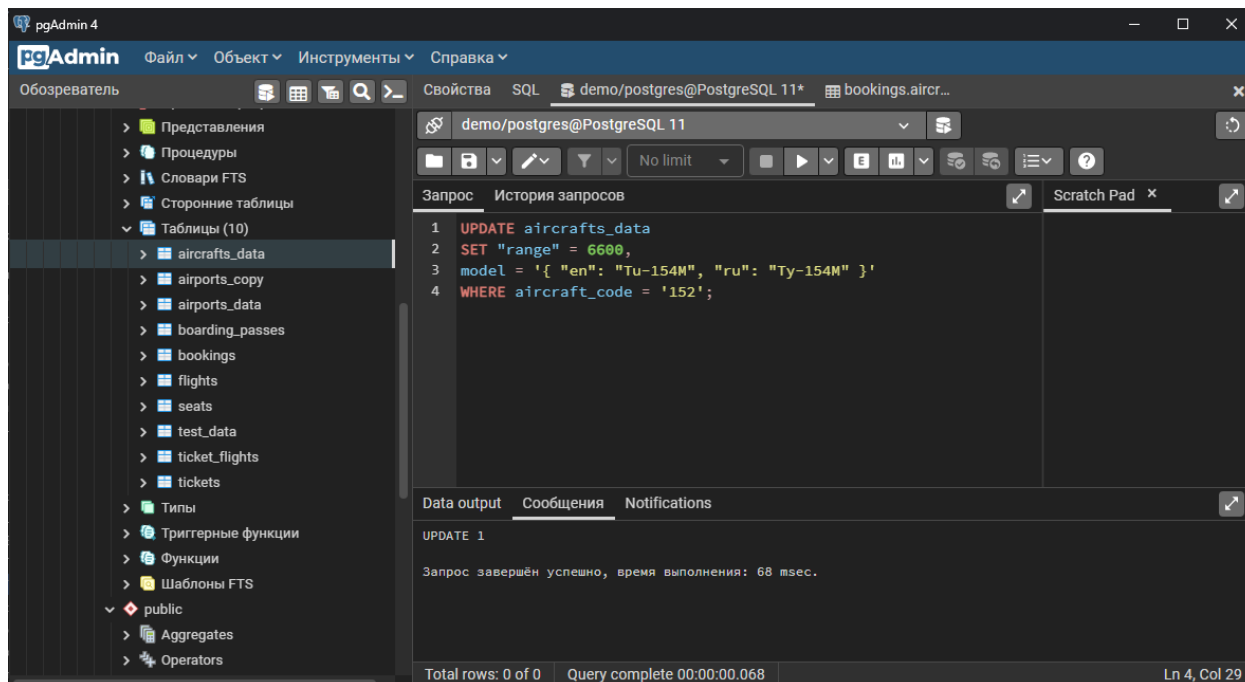


Рисунок 6 – Запрос №4

## Практическое задание для третьей части ЛР

. В ходе выполнения задания необходимо:

- Подготовить 3-4 выборки, которые имеют осмысленное значение для предметной области, и также составить для них SQL-скрипты.
- Сформулировать 3-4 запроса на изменение и удаление из базы данных. Запросы должны быть сформулированы в терминах предметной области. Среди запросов обязательно должны быть такие, которые будут вызывать срабатывание ограничений целостности. Составить SQL-скрипты для выполнения этих запросов.

## Темы для самостоятельной проработки

- Оператор SELECT.  
<https://postgrespro.ru/docs/postgrespro/11/queries>
- Оператор UPDATE.  
<https://postgrespro.ru/docs/postgrespro/11/dml-update>
- Оператор DELETE.

<https://postgrespro.ru/docs/postgrespro/11/dml-delete>

- Декларативные ограничения целостности.

<https://postgrespro.ru/docs/postgrespro/11/ddl-constraints>

- Полное описание синтаксиса встретившихся команд

<https://postgrespro.ru/docs/postgrespro/11/sql-commands>

### **Вопросы для самостоятельной проработки**

- Объяснить, как работают написанные запросы.
- Примеры вопросов по оператору SELECT см. в задании №1.
- Исправить неверно работающий запрос (запросы).
- Упростить один или несколько запросов.
- Написать или модифицировать запрос по сформулированному заданию.

## Часть 4. Контроль целостности данных

### Теоретическая часть и практические примеры

#### 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 на аккаунте (-)

```
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
```

-- изменение состояния счета Алисы в банке (-)

```
UPDATE branches SET balance = balance - 100.00
WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
```

-- изменение состояния счета Боба на аккаунте (+)

```
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
```

-- изменение состояния счета Боба в банке (+)

```
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 выполнены вместе, либо не выполнены 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.

При параллельном выполнении транзакций теоретически возможны следующие феномены.

**Потерянное обновление** (lost update). Когда разные транзакции одновременно изменяют одни и те же данные, то после фиксации изменений может оказаться, что одна транзакция перезаписала данные, обновленные и зафиксированные другой транзакцией.

**«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.

**Аномалия сериализации (serialization anomaly).** Результат успешной фиксации группы транзакций, выполняющихся параллельно, не совпадает с результатом ни одного из возможных вариантов упорядочения этих транзакций, если бы они выполнялись последовательно.

Транзакции в PostgreSQL определяются набором 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.

## Уровни изолированности

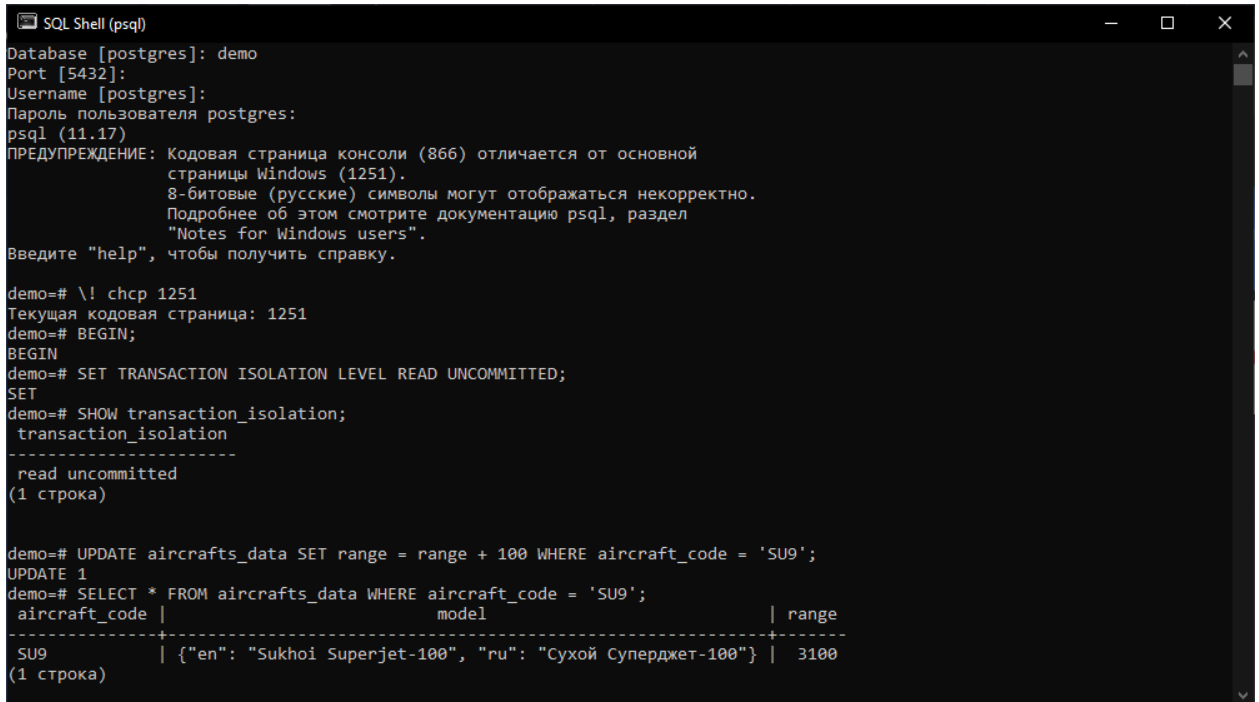
Сериализация двух транзакций при их параллельном выполнении означает, что полученный результат будет соответствовать одному из двух возможных вариантов упорядочения транзакций при их последовательном выполнении. При этом нельзя сказать точно, какой из вариантов будет реализован. Если распространить эти рассуждения на случай, когда параллельно выполняется более двух транзакций, тогда результат их параллельного выполнения также должен быть таким, каким он был бы в случае выбора некоторого варианта упорядочения транзакций, если бы они выполнялись последовательно, одна за другой. Конечно, чем больше транзакций, тем больше вариантов их упорядочения.

Концепция сериализации не предписывает выбора какого-то определенного варианта. Речь идет лишь об одном из них. В том случае, если СУБД не сможет гарантировать успешную сериализацию группы параллельных транзакций, тогда некоторые из них могут быть завершены с ошибкой. Эти транзакции придется выполнить повторно. Для конкретизации степени независимости параллельных транзакций вводится понятие уровня изоляции транзакций. Каждый уровень характеризуется перечнем тех феноменов, которые на данном уровне не допускаются. Всего в стандарте SQL предусмотрено четыре уровня. Каждый более высокий уровень включает в себя все возможности предыдущего.

**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 как на рисунке 7.



```
SQL Shell (psql)
Database [postgres]: demo
Port [5432]:
Username [postgres]:
Пароль пользователя postgres:
psql (11.17)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной
страницы Windows (1251).
8-битовые (русские) символы могут отображаться некорректно.
Подробнее об этом смотрите документацию psql, раздел
"Notes for Windows users".
Введите "help", чтобы получить справку.

demo=# \! chcp 1251
Текущая кодовая страница: 1251
demo=# BEGIN;
BEGIN
demo=# SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET
demo=# SHOW transaction_isolation;
transaction_isolation
-----
read uncommitted
(1 строка)

demo=# UPDATE aircrafts_data SET range = range + 100 WHERE aircraft_code = 'SU9';
UPDATE 1
demo=# SELECT * FROM aircrafts_data WHERE aircraft_code = 'SU9';
 aircraft_code | model | range
-----
SU9 | {"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"} | 3100
(1 строка)
```

Рисунок 7 – Транзакция №1 (Терминал №1)

### Запрос:

```
\! chcp 1251
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET SHOW transaction_isolation;
UPDATE aircrafts_data SET range = range + 100 WHERE aircraft_code = 'SU9';
SELECT * FROM aircrafts_data WHERE aircraft_code = 'SU9';
```

А во-втором терминале начнём другую транзакцию на рисунке 8.

```
SQL Shell (psql)
Server [localhost]:
Database [postgres]: demo
Port [5432]:
Username [postgres]:
Пароль пользователя postgres:
psql (11.17)
ПРЕДУПРЕЖДЕНИЕ: Кодовая страница консоли (866) отличается от основной
                  страницы Windows (1251).
                  8-битовые (русские) символы могут отображаться некорректно.
                  Подробнее об этом смотрите документацию psql, раздел
                  "Notes for Windows users".
Введите "help", чтобы получить справку.

demo=# \! chcp 1251
Текущая кодовая страница: 1251
demo=# BEGIN
demo=# ;
BEGIN
demo=# SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SET
demo=# SELECT * FROM aircrafts_data WHERE aircraft_code = 'SU9';
 aircraft_code | model | range
-----+-----+-----
 SU9           | {"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"} | 3000
(1 строка)

demo=#
```

Рисунок 8 – Транзакция №1 (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. На втором терминале так и сделаем. Во второй транзакции попытаемся обновить эту же строку таблицы aircrafts\_data, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута range не 100, а 200.

**Запрос:**

```
\! chcp 1251
BEGIN
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM aircrafts_data WHERE aircraft_code = 'SU9';
```



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 обеих консолях, для этого пишем команду «`ROLLBACK;`».

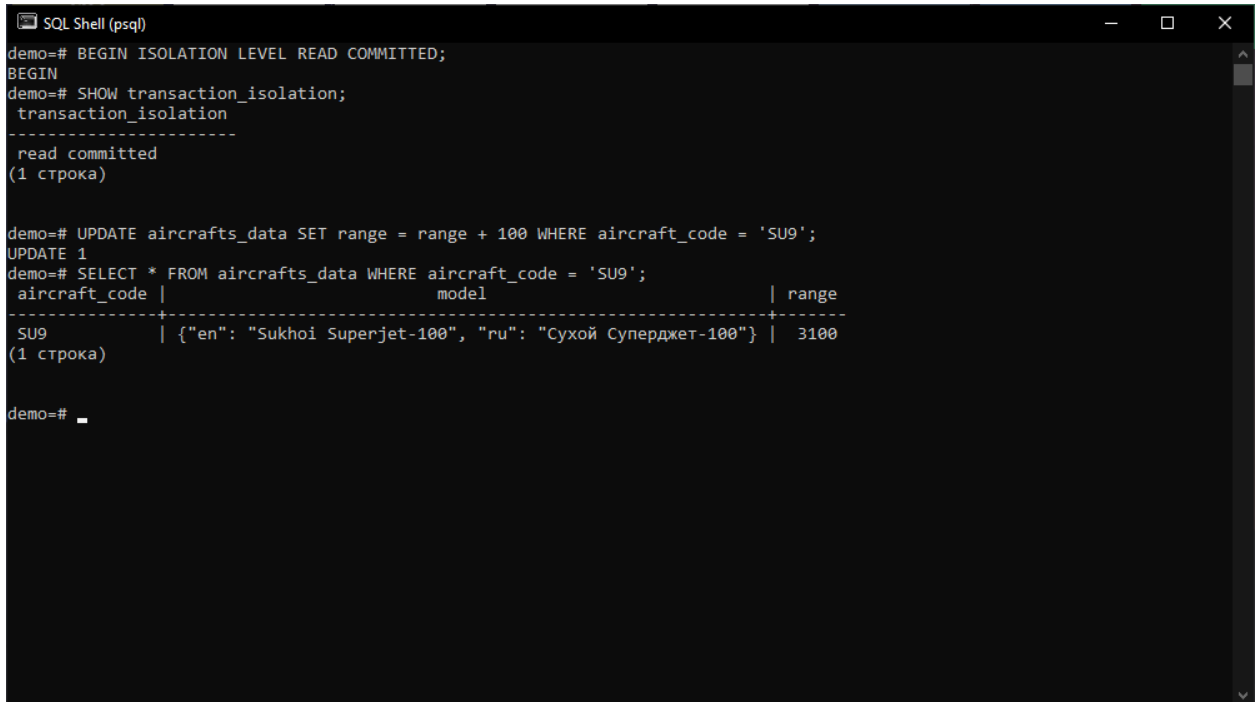
**READ COMMITTED.** Не допускается чтение «грязных» (незафиксированных) данных. Таким образом, в PostgreSQL уровень `READ UNCOMMITTED` совпадает с уровнем `READ COMMITTED`. Транзакция может видеть только те незафиксированные изменения данных, которые произведены в ходе выполнения ее самой.

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.

**Запрос:**

```
BEGIN ISOLATION LEVEL READ COMMITTED;  
SHOW transaction_isolation;  
UPDATE aircrafts_data SET range = range + 100 WHERE aircraft_code = 'SU9';  
SELECT * FROM aircrafts_data WHERE aircraft_code = 'SU9';
```

На первом терминале выполним следующие команды как на рисунке 9.



```
SQL Shell (psql)
demo=# BEGIN ISOLATION LEVEL READ COMMITTED;
BEGIN
demo=# SHOW transaction_isolation;
transaction_isolation
-----
read committed
(1 строка)

demo=# UPDATE aircrafts_data SET range = range + 100 WHERE aircraft_code = 'SU9';
UPDATE 1
demo=# SELECT * FROM aircrafts_data WHERE aircraft_code = 'SU9';
 aircraft_code | model | range
-----
SU9 | {"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"} | 3100
(1 строка)

demo=#
```

Рисунок 8 – Транзакция №2 (Терминал №1)

Видно, 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 в этот раз просто включено через 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.

На втором терминале так и сделаем, рисунок 9. Во второй транзакции попытаемся обновить эту же строку таблицы aircrafts\_tmp, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута range не 100, а 200.

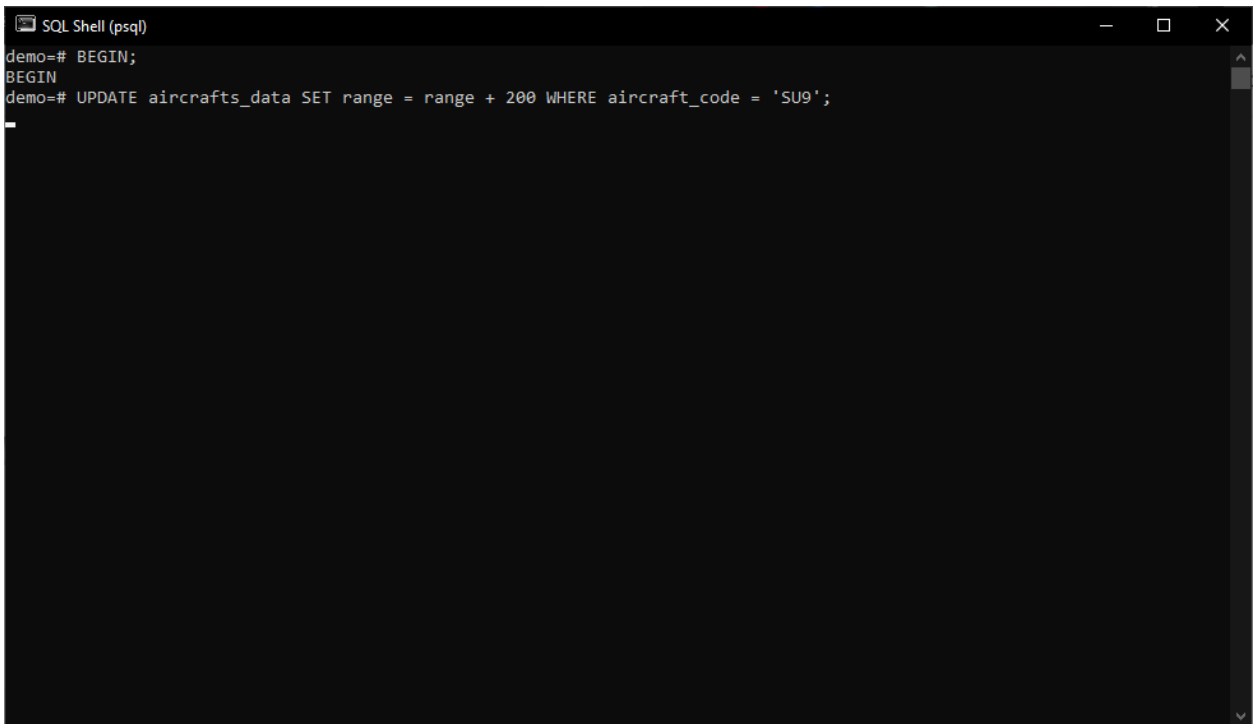
A screenshot of a terminal window titled "SQL Shell (psql)". The terminal shows a sequence of SQL commands: "demo=# BEGIN;", "BEGIN", and "demo=# UPDATE aircrafts\_data SET range = range + 200 WHERE aircraft\_code = 'SU9';". The cursor is positioned at the end of the last command, indicating it has been entered but not yet executed. The terminal background is black with white text.

Рисунок 9 – Транзакция №2 (Терминал №2)

И 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 aircrafts\_tmp, и эта блокировка будет снята только при завершении транзакции либо с фиксацией изменений с помощью команды COMMIT, либо с отменой изменений по команде ROLLBACK. И перейдя обратно во вторую консоль уже видим завершенную команду UPDATE, рисунок 10.

```
SQL Shell (psql)
demo=# BEGIN;
BEGIN
demo=# UPDATE aircrafts_data SET range = range + 200 WHERE aircraft_code = 'SU9';
UPDATE 1
demo=#
```

Рисунок 10 – UPDATE (Терминал №2)

Завершим транзакцию во первой консоли с помощью COMMIT.

Теперь на втором терминале, не завершая транзакцию, посмотрим, что стало с нашей строкой в таблице aircrafts\_data, рисунок 11.

```
SQL Shell (psql)
demo=# BEGIN;
BEGIN
demo=# UPDATE aircrafts_data SET range = range + 200 WHERE aircraft_code = 'SU9';
UPDATE 1
demo=# SELECT * FROM aircrafts_data WHERE aircraft_code = 'SU9';
 aircraft_code | model | range
-----+-----+-----
SU9            | {"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"} | 3300
(1 строка)

demo=#
```

Рисунок 11 – Результат операций в таблице (Терминал №1)

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 пишем такую транзакцию как на рисунке 12.

**Запрос:**

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;  
SELECT * FROM aircrafts_data;
```

```
SQL Shell (psql)
demo=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
demo=# SELECT * FROM aircrafts_data;
 aircraft_code | model | range
-----|-----|-----
773 | {"en": "Boeing 777-300", "ru": "Боинг 777-300"} | 11100
763 | {"en": "Boeing 767-300", "ru": "Боинг 767-300"} | 7000
320 | {"en": "Airbus A320-200", "ru": "Аэробус А320-200"} | 5700
321 | {"en": "Airbus A321-200", "ru": "Аэробус А321-200"} | 5600
319 | {"en": "Airbus A319-100", "ru": "Аэробус А319-100"} | 6700
733 | {"en": "Boeing 737-300", "ru": "Боинг 737-300"} | 4200
CN1 | {"en": "Cessna 208 Caravan", "ru": "Сессна 208 Караван"} | 1200
CR2 | {"en": "Bombardier CRJ-200", "ru": "Бомбардье CRJ-200"} | 2700
152 | {"en": "Tu-154M", "ru": "Ту-154М"} | 3222
SU9 | {"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"} | 3100
(10 строк)

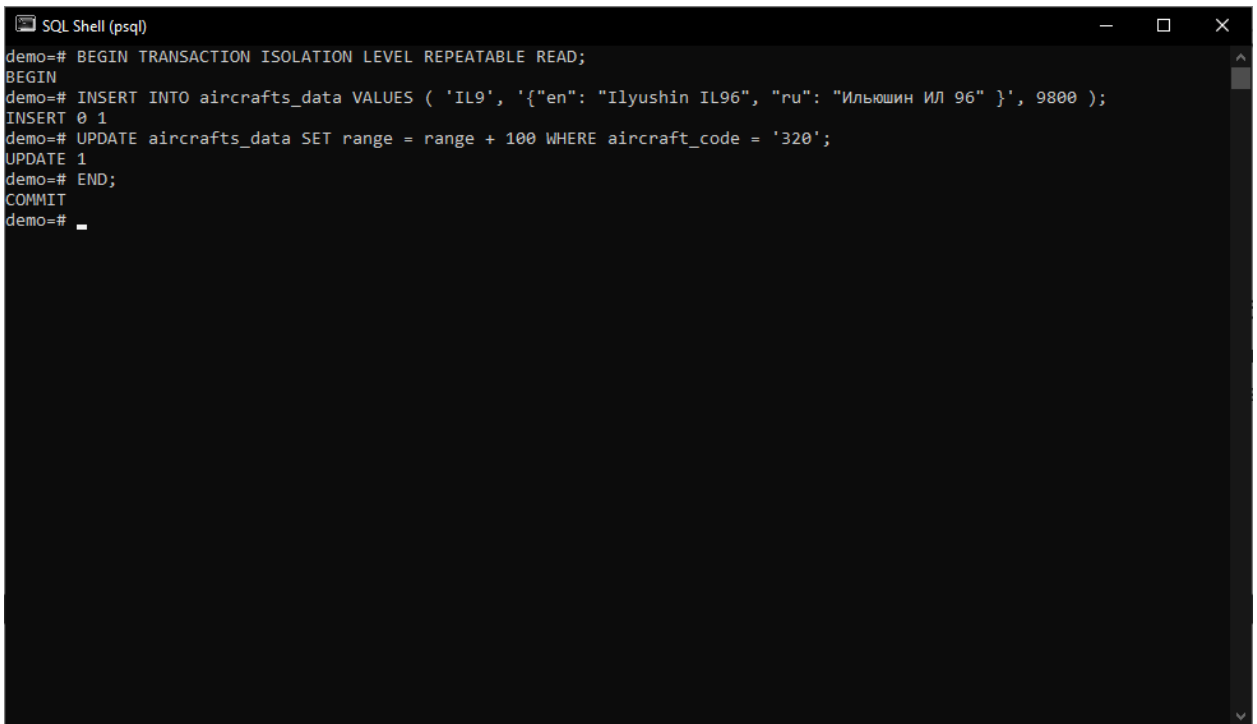
demo=#
```

Рисунок 12 – Транзакция №3 (Терминал №1)

На втором терминале проведем ряд изменений, рисунок 13.

**Запрос:**

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
INSERT INTO aircrafts_data VALUES ( 'IL9', '{"en": "Ilyushin IL96", "ru":
"Ильюшин ИЛ 96" }', 9800 );
UPDATE aircrafts_data SET range = range + 100 WHERE aircraft_code = '320';
END;
```



```
SQL Shell (psql)
demo=# BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN
demo=# INSERT INTO aircrafts_data VALUES ( 'IL9', '{"en": "Ilyushin IL96", "ru": "Ильюшин ИЛ 96" }', 9800 );
INSERT 0 1
demo=# UPDATE aircrafts_data SET range = range + 100 WHERE aircraft_code = '320';
UPDATE 1
demo=# END;
COMMIT
demo=#
```

Рисунок 13 – Транзакция №3 (Терминал №2)

Переходим на первый терминал. И пишем запрос для вывода всей информации из таблицы «aircrafts\_data». 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. Завершим транзакцию на в первой консоли и опять смотрим данные в таблице, рисунок 14.

```
SQL Shell (psql)
demo=# END;
COMMIT
demo=# SELECT * FROM aircrafts_data;
 aircraft_code | model | range
-----|-----|-----
773 | [{"en": "Boeing 777-300", "ru": "Боинг 777-300"}] | 11100
763 | [{"en": "Boeing 767-300", "ru": "Боинг 767-300"}] | 7900
321 | [{"en": "Airbus A321-200", "ru": "Аэробус А321-200"}] | 5600
319 | [{"en": "Airbus A319-100", "ru": "Аэробус А319-100"}] | 6700
733 | [{"en": "Boeing 737-300", "ru": "Боинг 737-300"}] | 4200
CN1 | [{"en": "Cessna 208 Caravan", "ru": "Сессна 208 Караван"}] | 1200
CR2 | [{"en": "Bombardier CRJ-200", "ru": "Бомбардье CRJ-200"}] | 2700
152 | [{"en": "Tu-154M", "ru": "Ту-154М"}] | 3222
SU9 | [{"en": "Sukhoi Superjet-100", "ru": "Сухой Суперджет-100"}] | 3300
IL9 | [{"en": "Ilyushin IL96", "ru": "Ильюшин ИЛ 96"}] | 9800
320 | [{"en": "Airbus A320-200", "ru": "Аэробус А320-200"}] | 5800
(11 строк)
demo=#
```

Рисунок 14 – Транзакция №3 (Терминал №1)

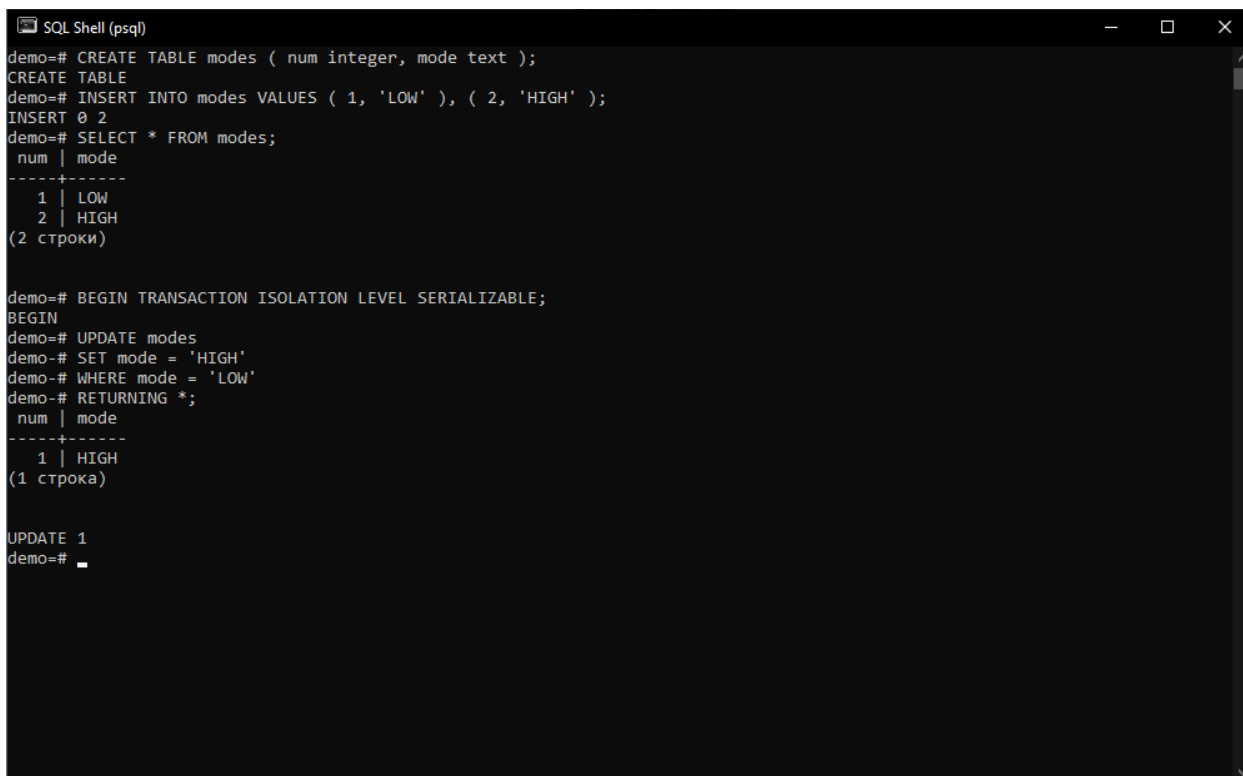
Как видим, одна строка добавлена, а значение атрибута range у самолета Airbus A320- 200 стало на 100 больше, чем было. Но до тех пор, пока мы на первом терминале находились в процессе выполнения первой транзакции, все эти изменения не были ей доступны, поскольку первая транзакция использовала снимок,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

Для проведения эксперимента создадим специальную таблицу, в которой будет всего два столбца: один — числовой, а второй — текстовый. Назовем эту таблицу — `modes`, рисунок 15. На первом терминале начнем транзакцию и обновим одну строку из тех двух строк, которые были показаны в предыдущем запросе.



```
SQL Shell (psql)
demo=# CREATE TABLE modes ( num integer, mode text );
CREATE TABLE
demo=# INSERT INTO modes VALUES ( 1, 'LOW' ), ( 2, 'HIGH' );
INSERT 0 2
demo=# SELECT * FROM modes;
 num | mode
-----+-----
  1  | LOW
  2  | HIGH
(2 строки)

demo=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
demo=# UPDATE modes
demo=# SET mode = 'HIGH'
demo=# WHERE mode = 'LOW'
demo=# RETURNING *;
 num | mode
-----+-----
  1  | HIGH
(1 строка)

UPDATE 1
demo=#
```

Рисунок 15 – Транзакция №4 (Терминал №1)

В команде обновления строки будем использовать предложение `RETURNING` (Возврат данных из изменённых строк). Поскольку значение поля `num` не изменяется, то будет видно, какая строка была обновлена. Это особенно пригодится во второй транзакции.

На втором терминале тоже начнем транзакцию и обновим другую строку из тех двух строк, которые были показаны выше, рисунок 16.

```
SQL Shell (psql)
demo=# BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN
demo=# UPDATE modes
demo-# SET mode = 'LOW'
demo-# WHERE mode = 'HIGH'
demo-# RETURNING *;
 num | mode
-----+-----
    2 | LOW
(1 строка)

UPDATE 1
```

Рисунок 16 – Транзакция №4 (Терминал №2)

Изменение, произведенное в первой транзакции, вторая транзакция не видит, поскольку на уровне изоляции SERIALIZABLE каждая транзакция работает с тем снимком базы данных, которых был сделан в ее начале, т. е. непосредственно перед выполнением ее первого оператора. Поэтому обновляется только одна строка, та, в которой значение поля mode было равно «HIGH» изначально.

Обратите внимание, что обе команды UPDATE были выполнены, ни одна из них не ожидает завершения другой транзакции.

Заканчиваем эксперимент. Сначала завершим транзакцию на первом терминале, рисунок 17. А потом на втором терминале, рисунок 18. Таким образом, параллельное выполнение двух транзакций сериализовать не удалось. Почему? Если обратиться к определению концепции сериализации, то нужно рассуждать так. Если бы была зафиксирована и вторая транзакция, тогда в таблице modes содержались бы такие строки ( num | mode -----+----- 1 | HIGH 2 | LOW).

```
SQL Shell (psql)
demo=# COMMIT;
COMMIT
demo=#
```

Рисунок 17 – Транзакция №4 (Терминал №1)

```
SQL Shell (psql)
demo=# COMMIT;
ОШИБКА: не удалось сериализовать доступ из-за зависимостей чтения/записи между транзакциями
ПОДРОБНОСТИ: Reason code: Canceled on identification as a pivot, during commit attempt.
ПОДСКАЗКА: Транзакция может завершиться успешно при следующей попытке.
demo=# SELECT * FROM modes;
 num | mode
-----+-----
   2 | HIGH
   1 | HIGH
(2 строки)

demo=#
```

Рисунок 18 – Транзакция №4 (Терминал №2)

Но результат из терминала 2 не соответствует результату выполнения транзакций ни при одном из двух возможных вариантов их упорядочения, если бы они выполнялись последовательно. Следовательно, с точки зрения концепции сериализации, эти транзакции невозможно сериализовать. Покажем это, выполнив транзакции последовательно. Предварительно необходимо пересоздать таблицу modes или с помощью команды UPDATE вернуть ее измененным строкам исходное состояние.

## **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 грязным чтением. 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.

## **CREATE TRIGGER**

Триггер является указанием, что база данных должна автоматически выполнить заданную функцию, всякий раз когда выполнен определённый тип операции. Триггеры можно использовать с таблицами (секционированными и обычными), с представлениями и с внешними таблицами.

Для обычных и сторонних таблиц можно определять триггеры, которые будут срабатывать до или после любой из команд INSERT, UPDATE или DELETE; либо один раз для каждой модифицируемой строки, либо один раз для оператора SQL. Триггеры на UPDATE можно установить так, чтобы они срабатывали, только когда в предложении SET оператора UPDATE упоминаются определённые столбцы. Также триггеры могут срабатывать для

операторов TRUNCATE. Если происходит событие триггера, для обработки этого события в установленный момент времени вызывается функция триггера.

Формальное определение триггера:

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER имя { BEFORE | AFTER | INSTEAD  
OF } { событие [ OR ... ] }  
ON имя_таблицы  
[ FROM ссылающаяся_таблица ]  
[ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY  
DEFERRED ] ]  
[ REFERENCING { { OLD | NEW } TABLE [ AS ] имя_переходного_отношения } [ ... ]  
]  
[ FOR [ EACH ] { ROW | STATEMENT } ]  
[ WHEN ( условие ) ]  
EXECUTE { FUNCTION | PROCEDURE } имя_функции ( аргументы )
```

К примеру, напишем триггер, который будет добавлять налог на цену билета после добавления записи. Для этого нам понадобится ещё и написать процедуру, которая, собственно, и будет вызываться в нашем триггере, рисунок 19.

**Процедура:**

```
CREATE OR REPLACE FUNCTION add_tax()  
RETURNS TRIGGER AS  
$$  
BEGIN  
    UPDATE ticket_flights SET amount = amount + (amount * 0.36)  
    WHERE flight_id = NEW.flight_id;  
    RETURN NEW;  
END;  
$$  
LANGUAGE 'plpgsql';
```

Рисунок 19 – Процедура

У процедура имеет свое тело, которое обозначается «\$\$» в начале и конце, а само действие BEGIN и END. Сама процедура будет изменять цену билета, а для получения того товара, который был добавлен или изменен, находим этот товар по Id. Но как найти нужный билет? Воспользуемся ключевым словом NEW, который хранит в себе параметры нашего билета.

Таким образом, триггер будет срабатывать при любой операции INSERT над таблицей ticket\_flights после записи, рисунок 20.

### Триггер:

```
CREATE TRIGGER check_update
    AFTER INSERT ON ticket_flights
    FOR EACH ROW
    EXECUTE FUNCTION add_tax();
```

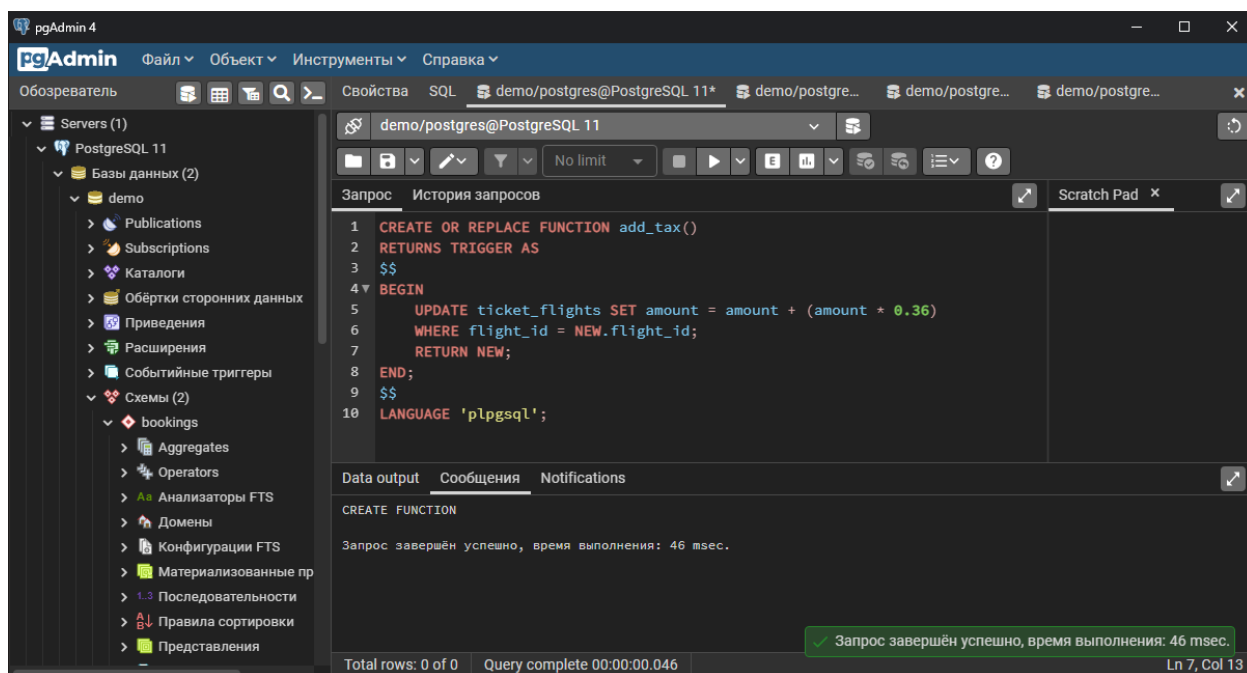


Рисунок 20 – Триггер

В результате при добавлении новой записи к ней применяется триггер изменения цены, что можно пронаблюдать на рисунках 21, 22.

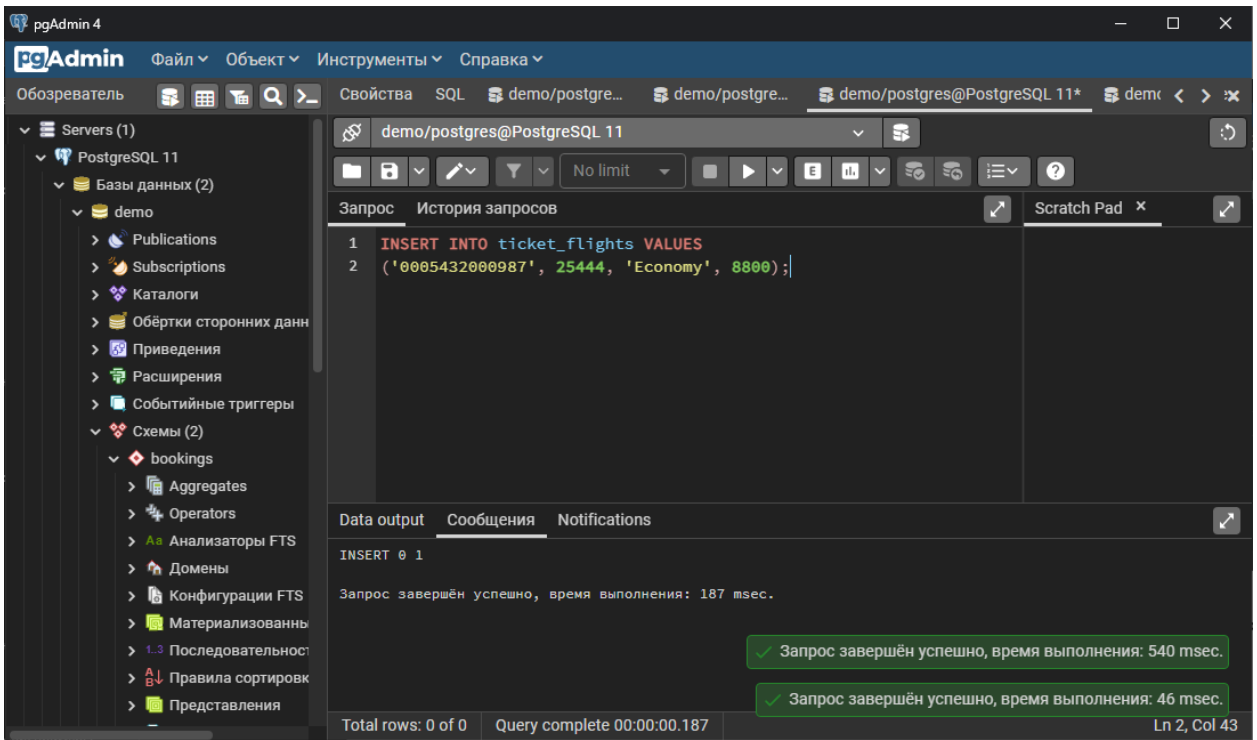


Рисунок 21 – Новая запись

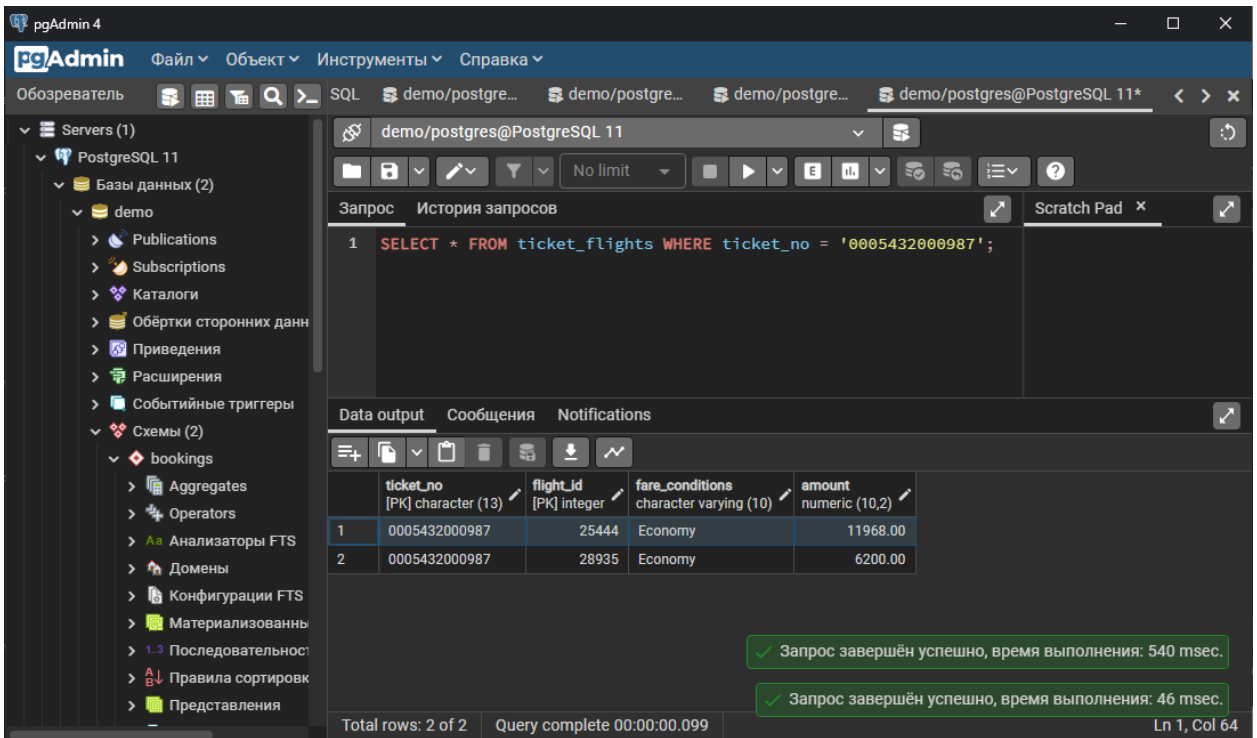


Рисунок 22 – Результат

## Практическое задание для четвертой части ЛР

Пятое практическое задание посвящено контролю целостности данных, который производится с помощью механизма транзакций и триггеров. Транзакции позволяют рассматривать группу операций как единое целое, либо обрабатывают все операции, либо ни одной. Это позволяет избежать несогласованности данных. Триггеры позволяют проверять целостность данных в момент выполнения транзакций, поддерживать целостность, внося изменения, и откатывать транзакции, приводящие к потере целостности. Необходимо подготовить SQL-скрипты для проверки наличия аномалий (потерянных изменений, грязных чтений, неповторяющихся чтений, фантомов) при параллельном исполнении транзакций на различных уровнях изолированности SQL/92 (READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE). Подготовленные скрипты должны работать с одной из таблиц, созданных в практическом задании №2.1. Для проверки наличия аномалий потребуются два параллельных сеанса, операторы в которых выполняются пошагово:

- Установить в обоих сеансах уровень изоляции READ UNCOMMITTED. Выполнить сценарии

- проверки наличия аномалий потерянных изменений и грязных чтений.

- Установить в обоих сеансах уровень изоляции READ COMMITTED.

Выполнить сценарии

- проверки наличия аномалий грязных чтений и неповторяющихся чтений.

- Установить в обоих сеансах уровень изоляции REPEATABLE READ.

Выполнить сценарии проверки наличия аномалий неповторяющихся чтений и фантомов.

- Установить в обоих сеансах уровень изоляции SERIALIZABLE.

Выполнить сценарий проверки наличия фантомов.



Необходимо составить скрипт для создания триггера, а также подготовить несколько запросов для проверки и демонстрации его полезных свойств:

- Изменение данных для сохранения целостности.
- Проверка транзакций и их откат в случае нарушения целостности.

### **Темы для самостоятельной проработки**

- Понятие транзакции, свойства транзакций.  
<https://postgrespro.ru/docs/postgrespro/11/tutorial-transactions>
- Уровни изолированности и аномалии  
<https://postgrespro.ru/docs/postgrespro/11/transaction-iso>
- Триггеры и триггерные функции  
<https://postgrespro.ru/docs/postgrespro/11/trigger-definition>  
<https://postgrespro.ru/docs/postgrespro/11/plpgsql-trigger>
- Сообщения и ошибки  
<https://postgrespro.ru/docs/postgrespro/11/plpgsql-errors-and-messages>
- Полное описание синтаксиса встретившихся команд  
<https://postgrespro.ru/docs/postgrespro/11/sql-commands>

### **Примеры вопросов для самостоятельной проработки**

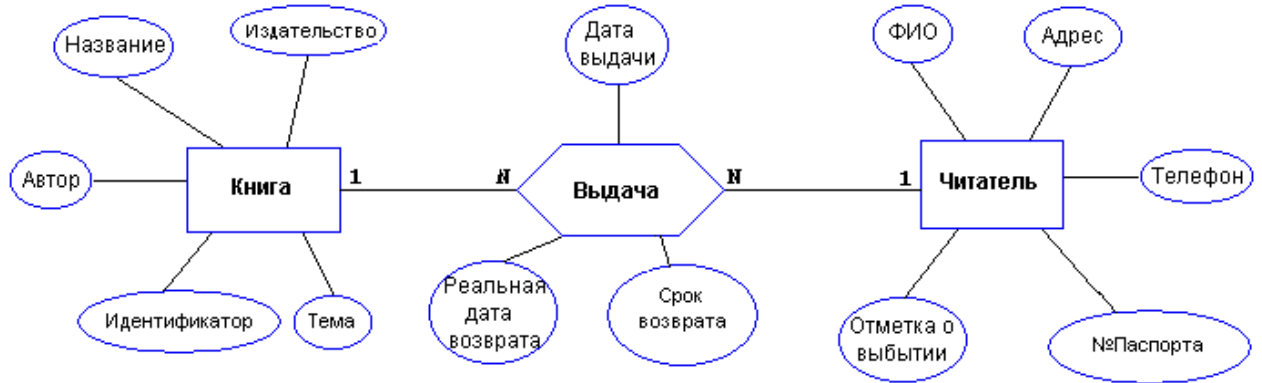
- Рассказать об аномалиях доступа к БД.
- Перечислить аномалии, возникающие на каждом из уровней изолированности.
- Рассказать о свойствах транзакций.
- Рассказать об управлении транзакциями.
- Что такое тупики? Как бороться с тупиками?
- На каком уровне изолированности возможны тупики?
- Как обеспечивается изолированность транзакций в СУБД?
- Как бороться с проблемой фантомов?
- Что такое журнал транзакций?

- Как обеспечивается постоянство хранения (durability) в СУБД?
- Объяснить принцип работы написанного триггера.
- Какие бывают типы триггеров?
- Когда может срабатывать триггер?
- В каком порядке срабатывают триггеры?
- Можно ли менять порядок срабатывания триггеров?
- Срабатывает ли триггер, если оператор, выполненный пользователем, не затрагивает ни одну строку таблицы?
- Продемонстрировать откат транзакции при возникновении ошибок.
- Продемонстрировать возникновение тупика.
- Исправить неверные сценарии проверки аномалий
- Исправить ошибки в работе триггера.
- Модифицировать триггер каким-либо образом.

## Вариант №1

Предметная область для практических заданий №2.\* №3.\*: **Библиотека**

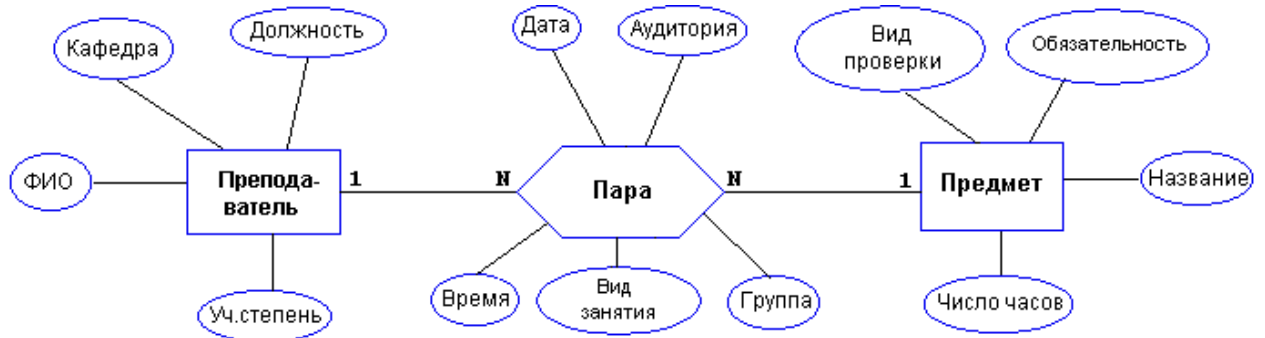
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №2

Предметная область для практических заданий №2.\* №3.\*: **Университет**

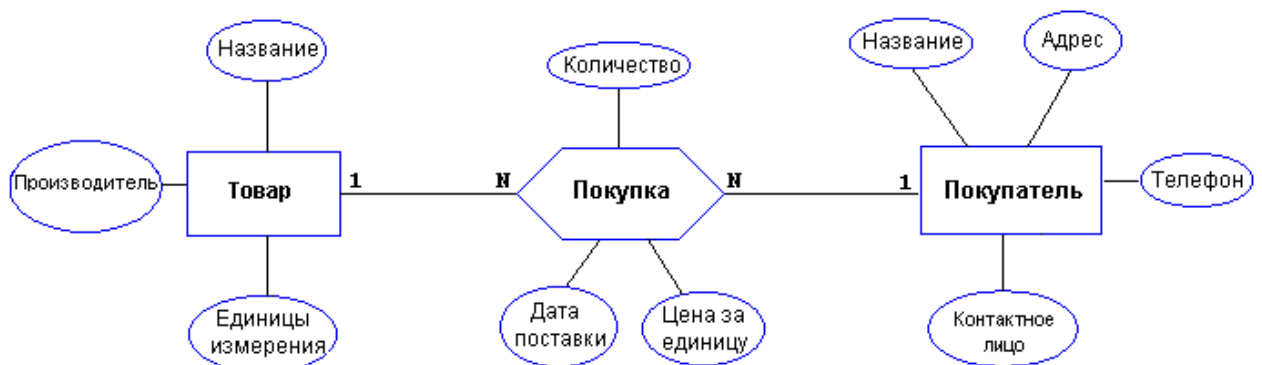
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №3

Предметная область для практических заданий №2.\* №3.\*: **Отдел продаж**

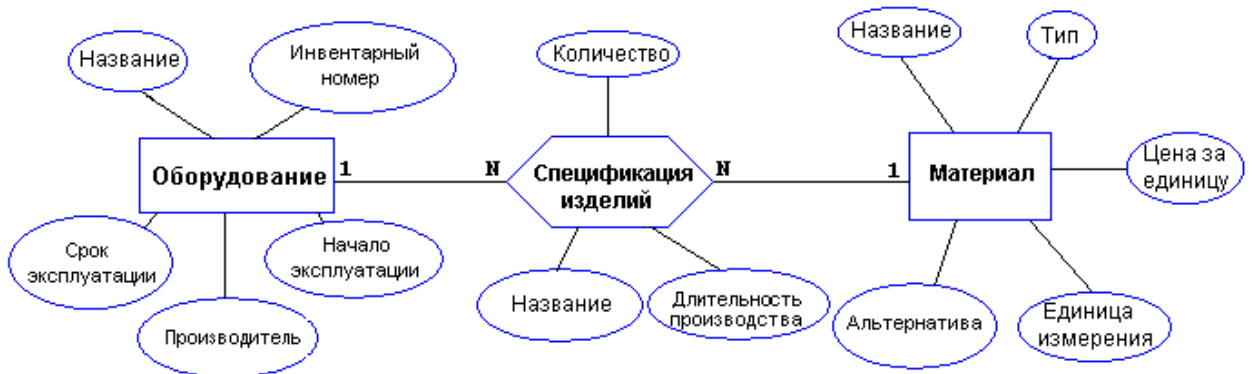
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №4

Предметная область для практических заданий №2.\* №3.\*: **Производство**

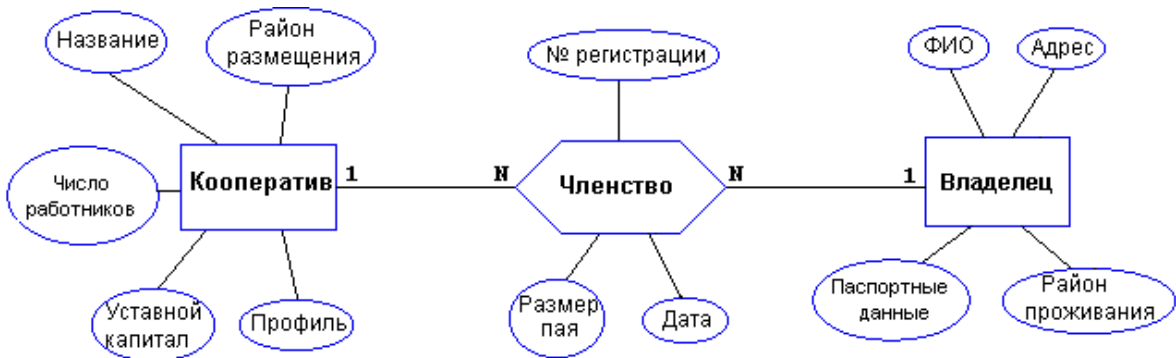
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №5

Предметная область для практических заданий №2.\* №3.\*: **Кооперативы**

Пример схемы для задания №2.1, от которого можно отталкиваться:



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

## Вариант №6

Предметная область для практических заданий №2.\* №3.\*: **Автомастерская**

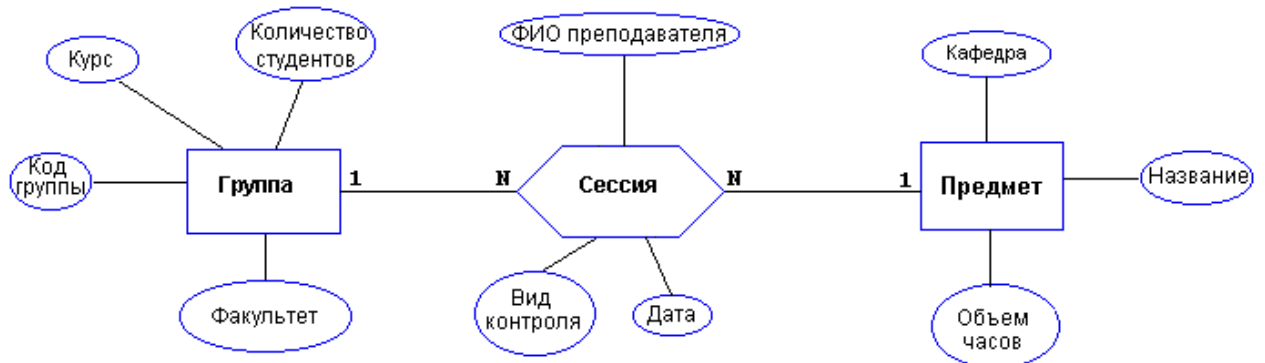
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №7

Предметная область для практических заданий №2.\* №3.\*: **Сессия**

Пример схемы для задания №2.1, от которого можно отталкиваться:



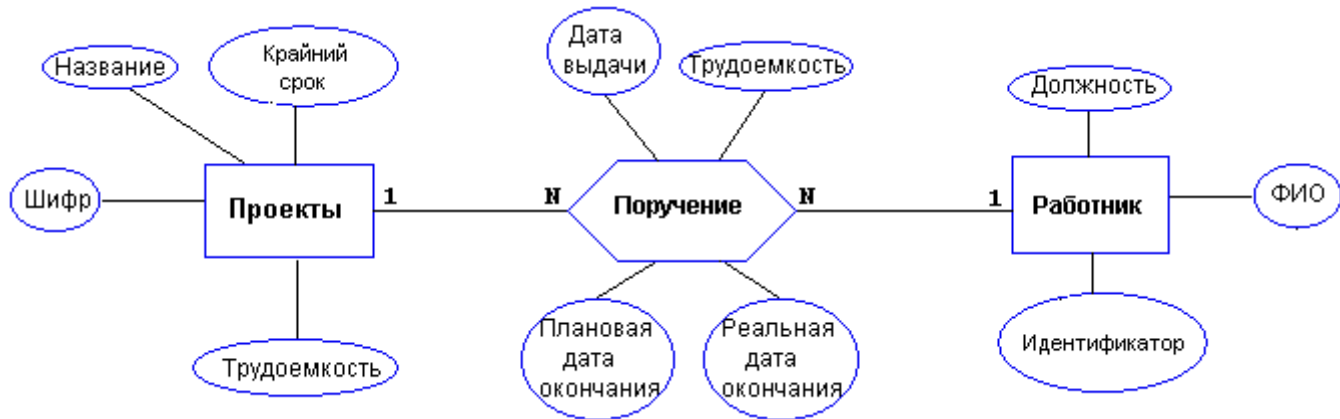
Вид контроля - зачет, экзамен.

## Вариант №8

Предметная область для практических заданий №2.\* №3.\*:

### Управление проектом

Пример схемы для задания №2.1, от которого можно отталкиваться:



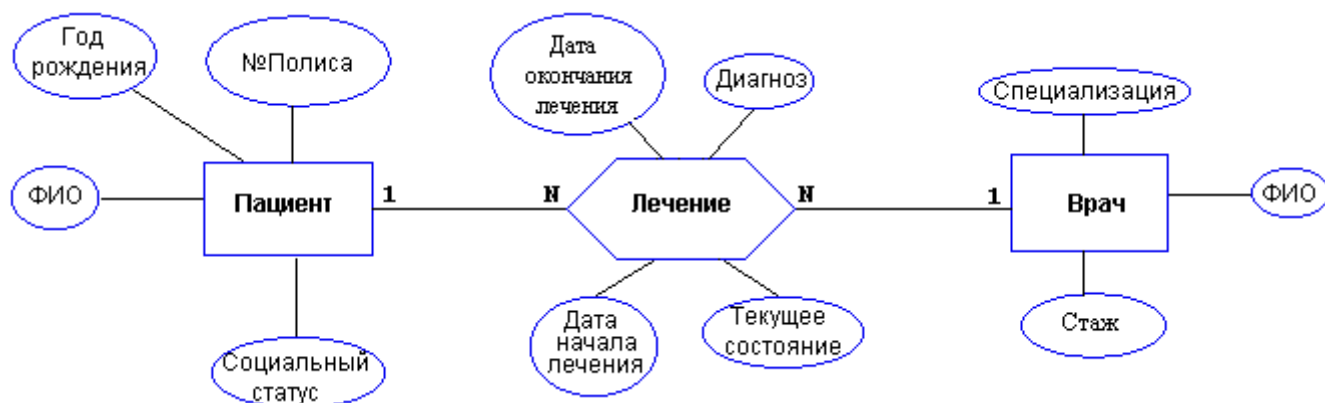
Категория дисциплины - гуманитарная, математическая, компьютерная и т.д.

Вид контроля - зачет, экзамен.

## Вариант №9

Предметная область для практических заданий №2.\* №3.\*: **Поликлиника**

Пример схемы для задания №2.1, от которого можно отталкиваться:



Текущее состояние - средней тяжести, тяжелое, направлен в стационар, умер.

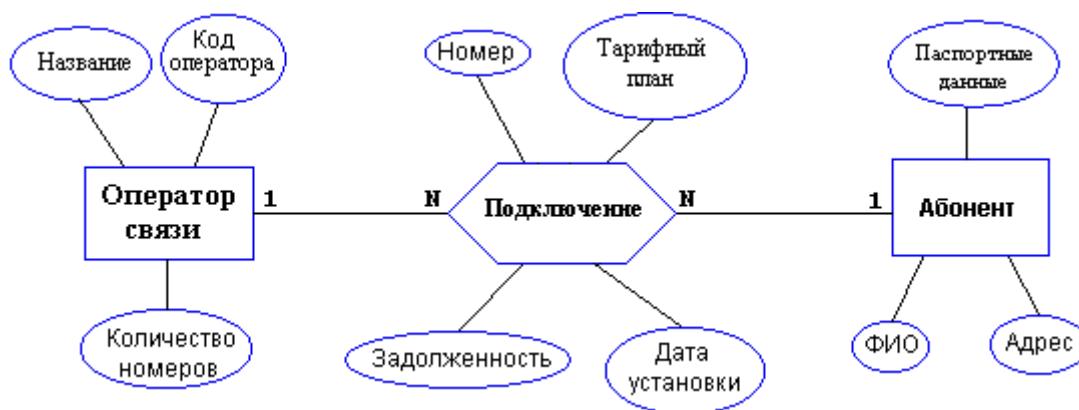
Социальный статус пациента - учащийся, работающий, врем. неработающий, инвалид, пенсионер

Специализация врача - терапевт, хирург и т.п.

## Вариант №10

Предметная область для практических заданий №2.\* №3.\*: **Сотовая связь**

Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №11

Предметная область для практических заданий №2.\* №3.\*: **Спорт**

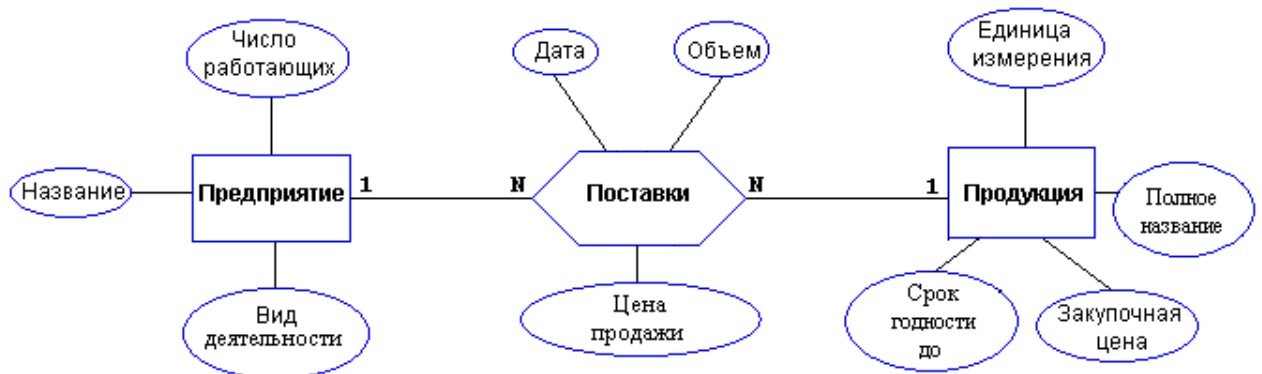
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №12

Предметная область для практических заданий №2.\* №3.\*: **Поставки**

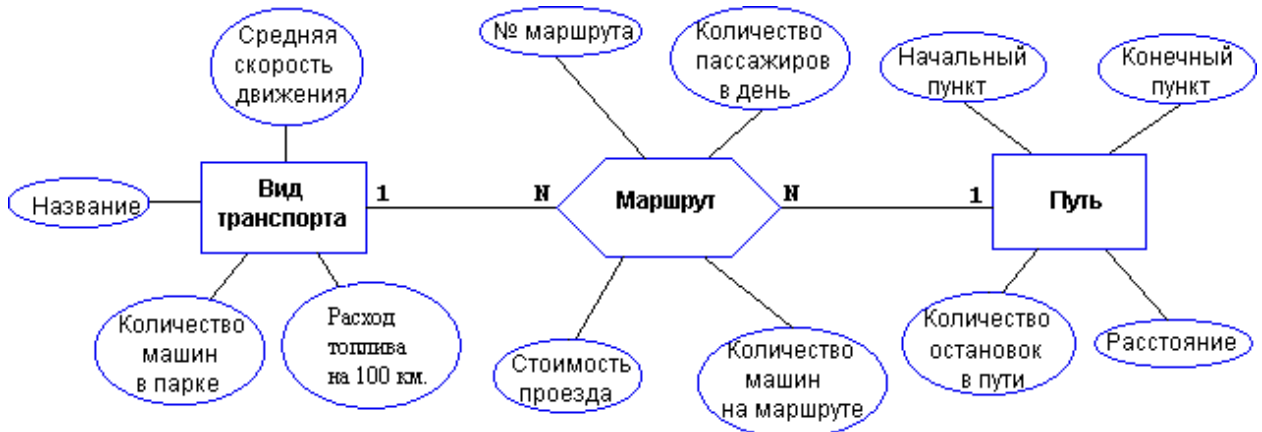
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №13

Предметная область для практических заданий №2.\* №3.\*: **Транспорт**

Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №14

Предметная область для практических заданий №2.\* №3.\*: **География**

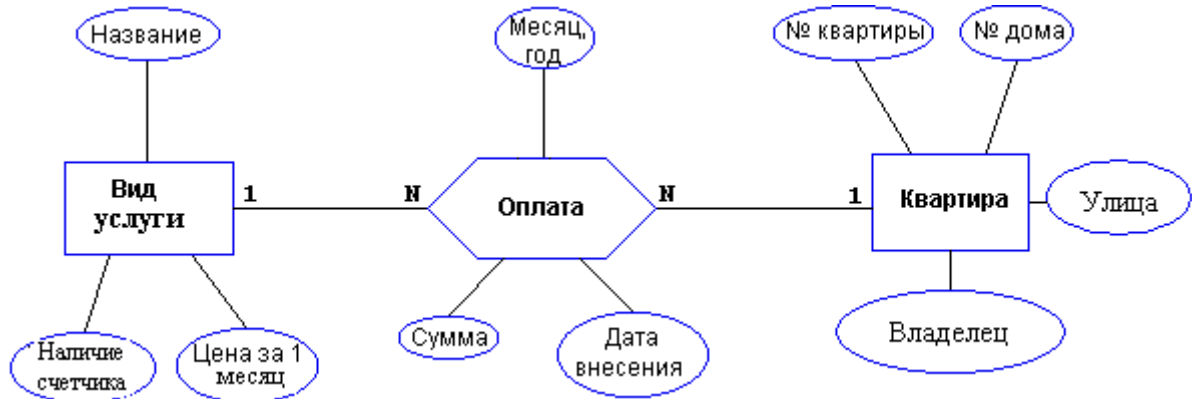
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №15

Предметная область для практических заданий №2.\* №3.\*: **Домоуправление**

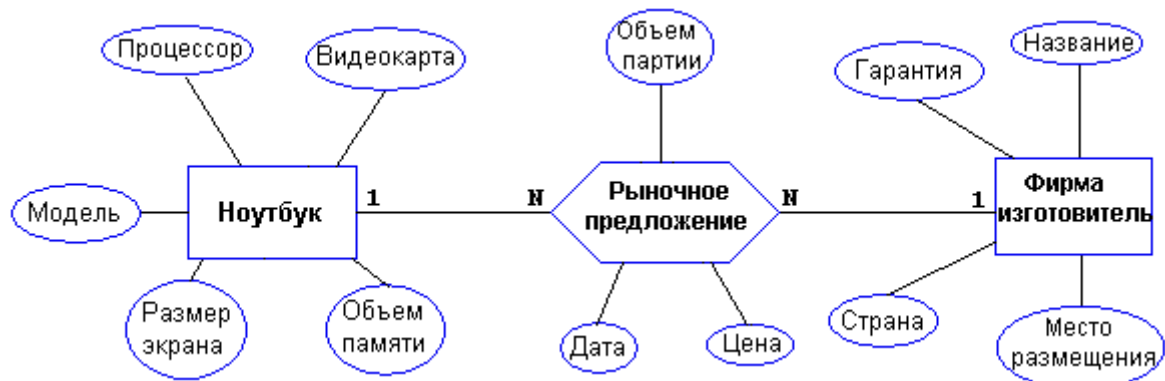
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №16

Предметная область для практических заданий №2.\* №3.\*: **Ноутбуки**

Пример схемы для задания №2.1, от которого можно отталкиваться:





## Вариант №17

Предметная область для практических заданий №2.\* №3.\*: **Деканат**

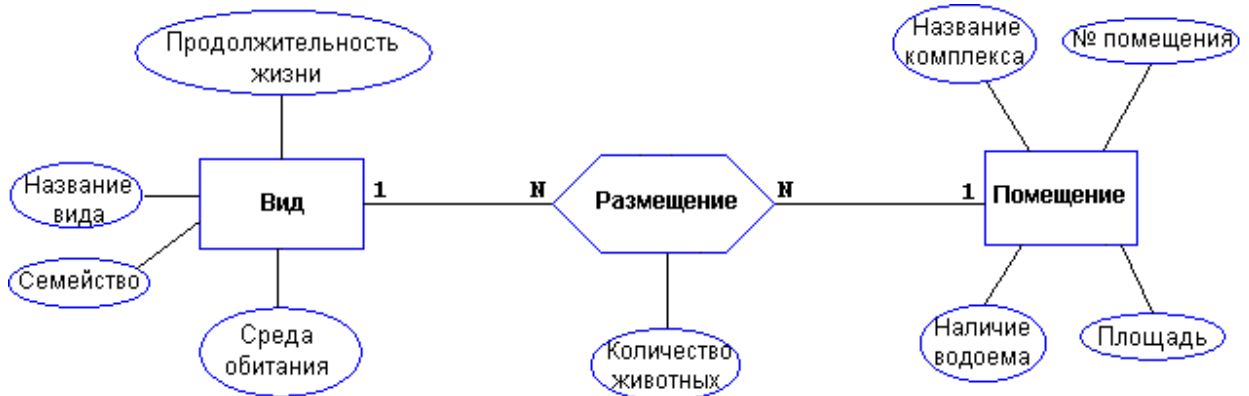
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №18

Предметная область для практических заданий №2.\* №3.\*: **Зоопарк**

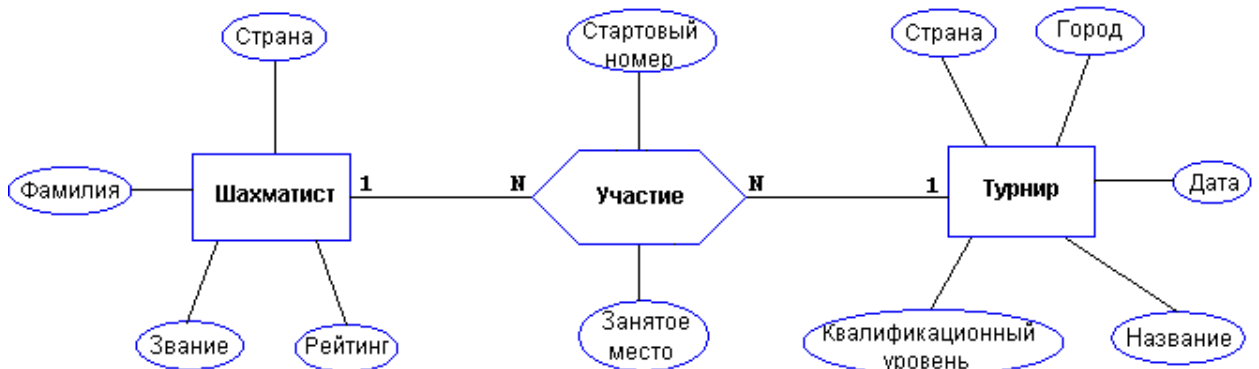
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №19

Предметная область для практических заданий №2.\* №3.\*: **Шахматы**

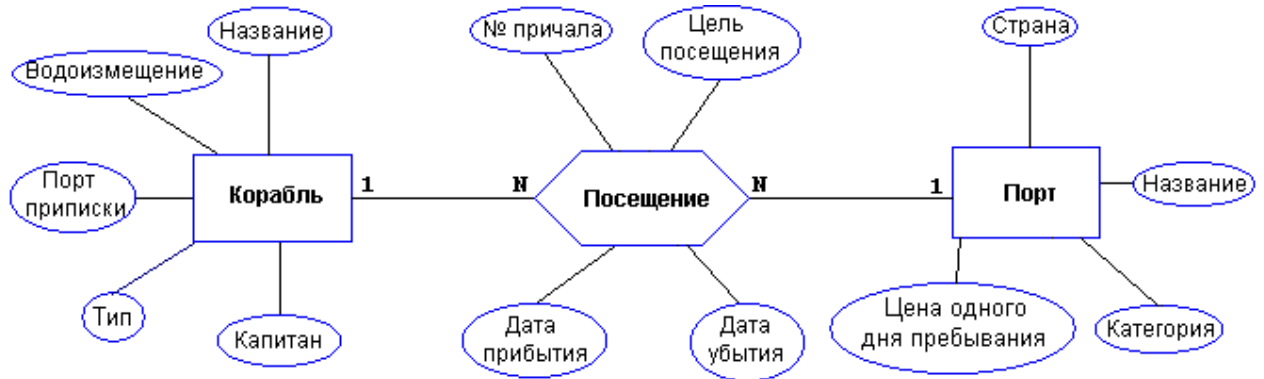
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №20

Предметная область для практических заданий №2.\* №3.\*: **Судоходство**

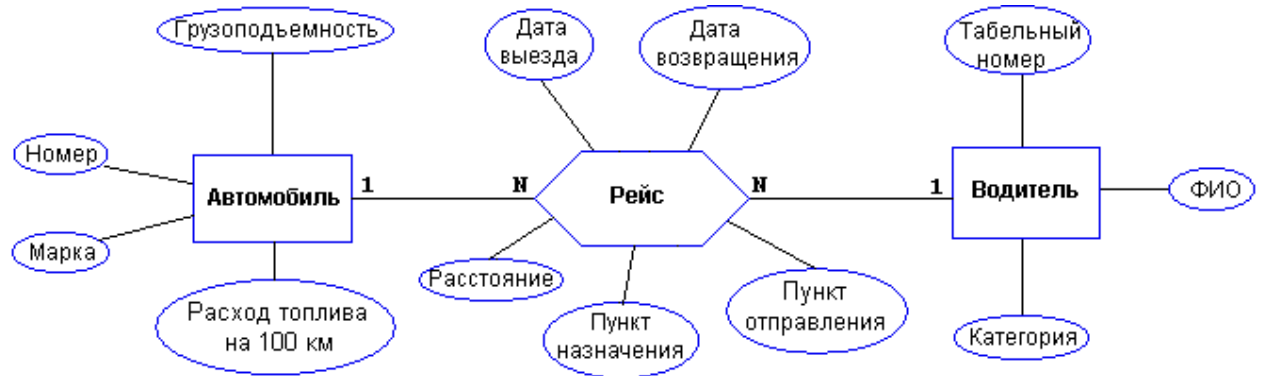
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №21

Предметная область для практических заданий №2.\* №3.\*: **Грузоперевозки**

Пример схемы для задания №2.1, от которого можно отталкиваться:

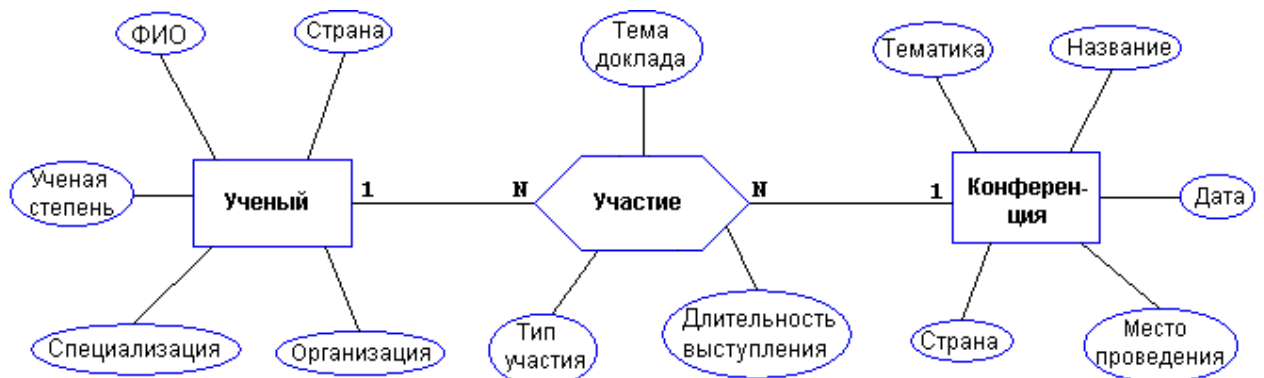


## Вариант №22

Предметная область для практических заданий №2.\* №3.\*:

**Научные конференции**

Пример схемы для задания №2.1, от которого можно отталкиваться:

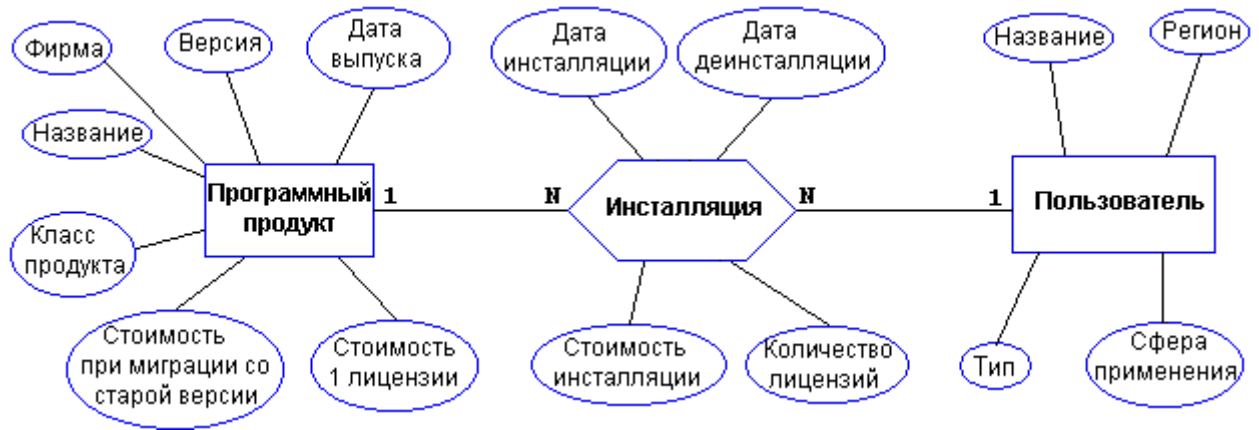


## Вариант №23

Предметная область для практических заданий №2.\* №3.\*:

### Программные продукты

Пример схемы для задания №2.1, от которого можно отталкиваться:



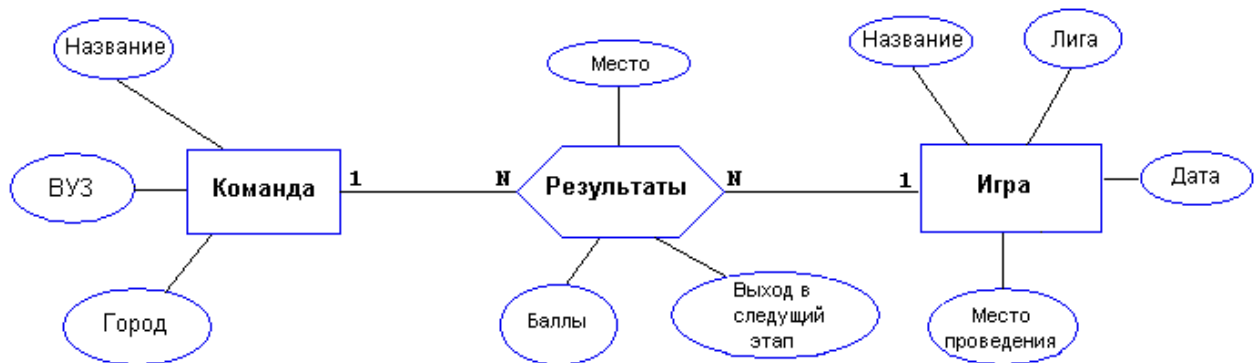
Класс: ОС, сервер приложений, СУБД, Web-сервер, система программирования и т.д.

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

## Вариант №24

Предметная область для практических заданий №2.\* №3.\*: КВН

Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №25

Предметная область для практических заданий №2.\* №3.\*: **Добыча ресурсов**

Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №26

Предметная область для практических заданий №2.\* №3.\*: **Театр**

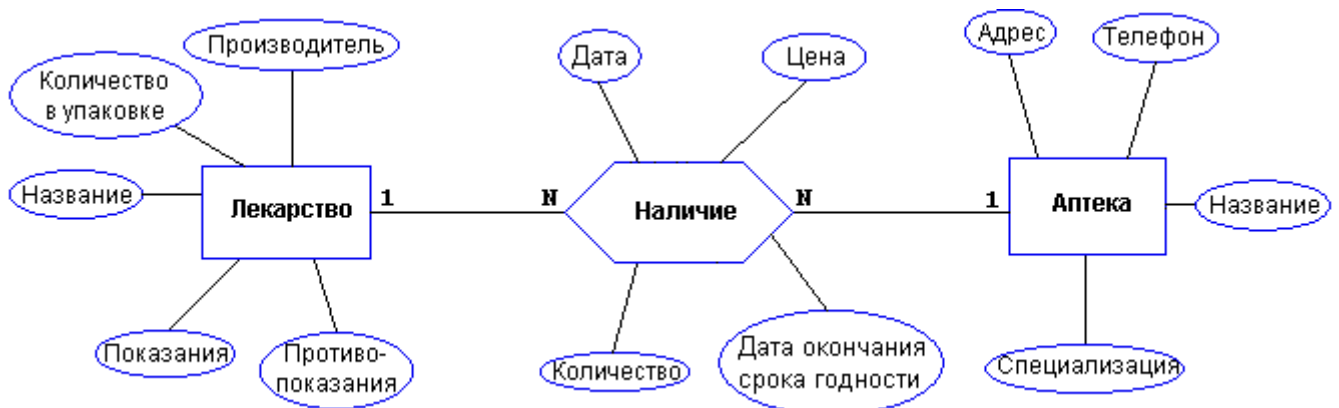
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №27

Предметная область для практических заданий №2.\* №3.\*: **Справочная аптек**

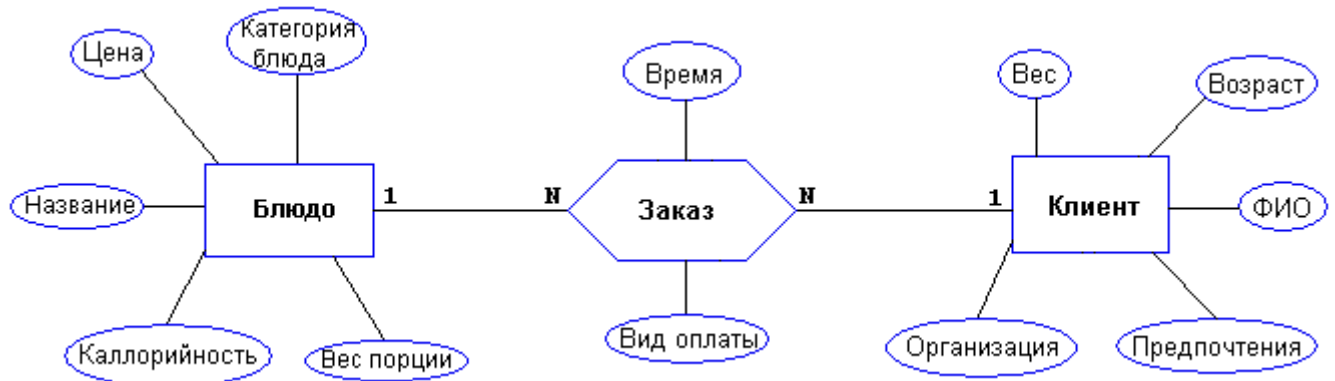
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №28

Предметная область для практических заданий №2.\* №3.\*: **Столовая**

Пример схемы для задания №2.1, от которого можно отталкиваться:

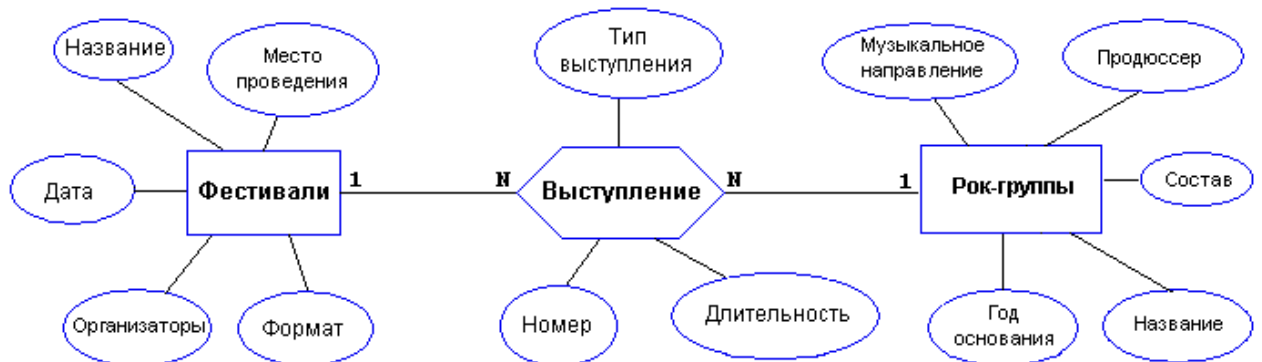


Категория блюда: первое, второе, гарнир, десерт и т.д.

## Вариант №29

Предметная область для практических заданий №2.\* №3.\*: **Рок-группы**

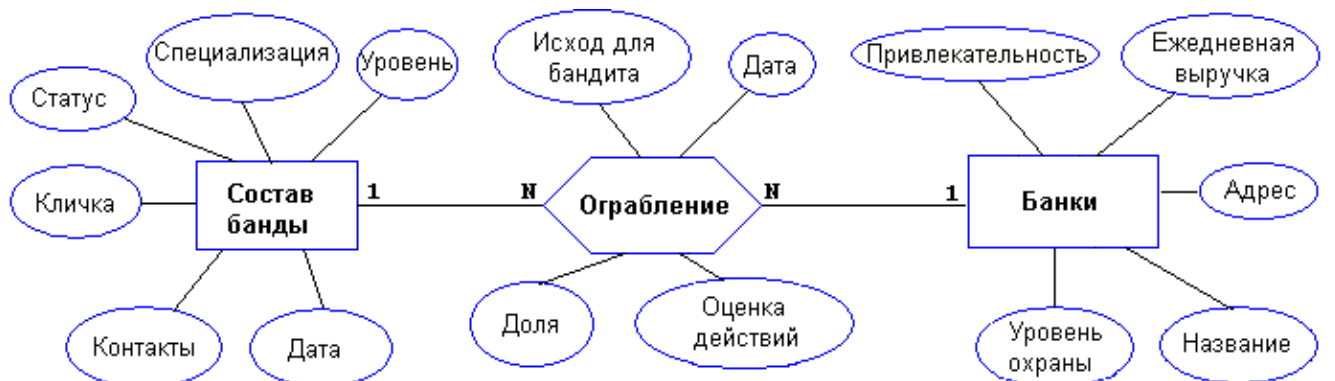
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №30

Предметная область для практических заданий №2.\* №3.\*: **ОПГ**

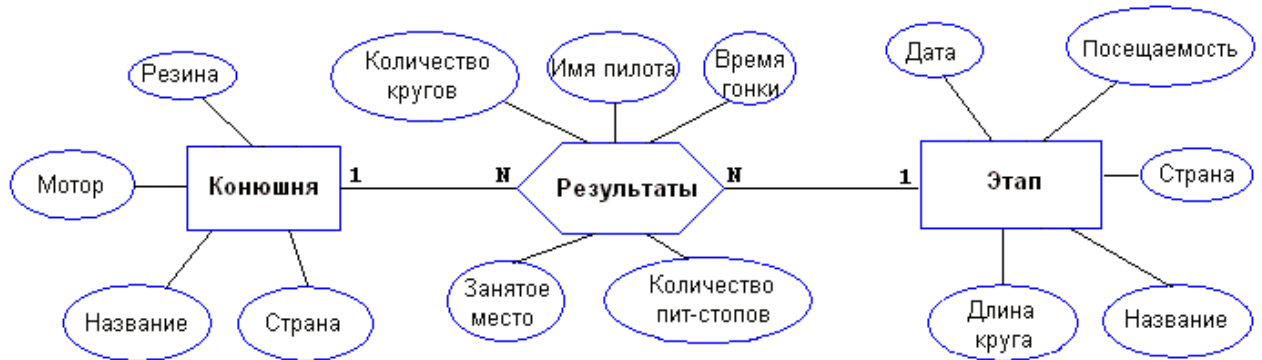
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №31

Предметная область для практических заданий №2.\* №3.\*: **Формула 1**

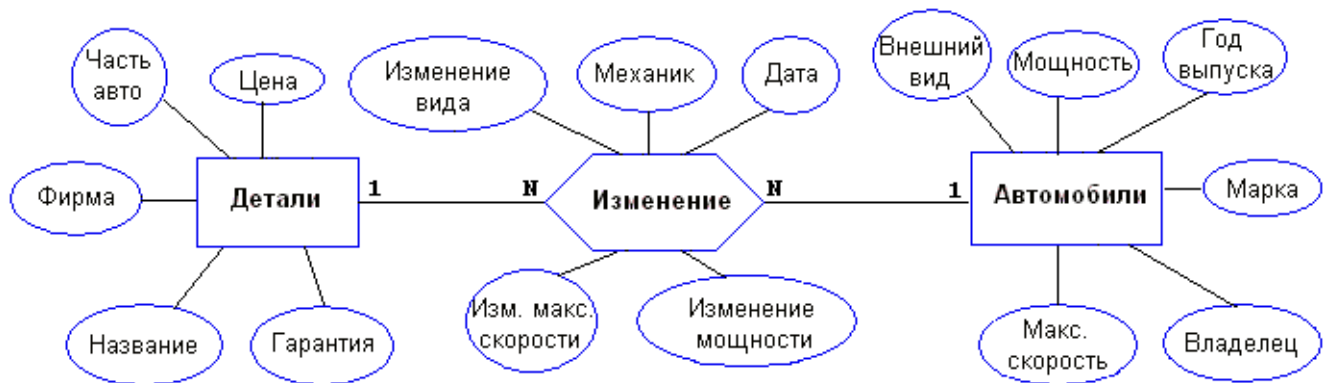
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №32

Предметная область для практических заданий №2.\* №3.\*: **Тюнинг**

Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №33

Предметная область для практических заданий №2.\* №3.\*:

**Тележурналистика**

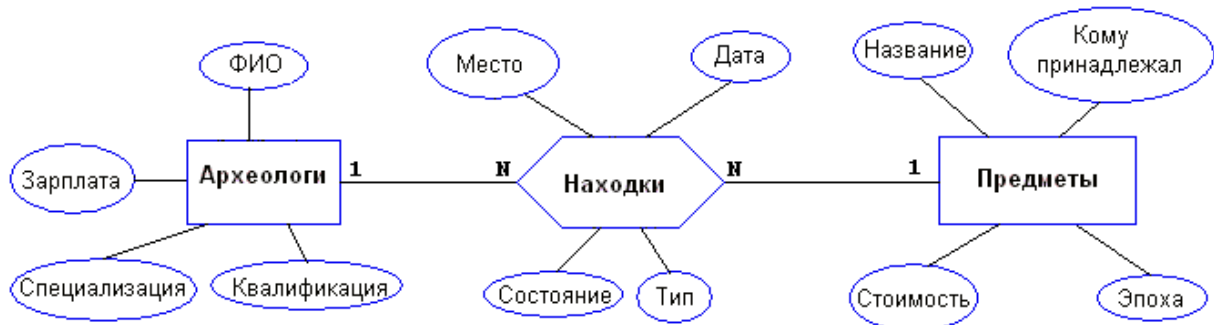
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №34

Предметная область для практических заданий №2.\* №3.\*: **Археология**

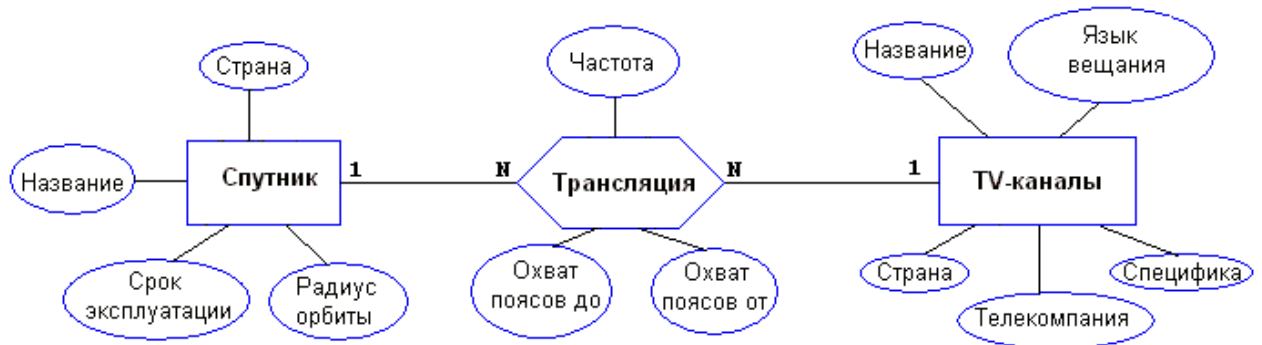
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №35

Предметная область для практических заданий №2.\* №3.\*: **Телевещание**

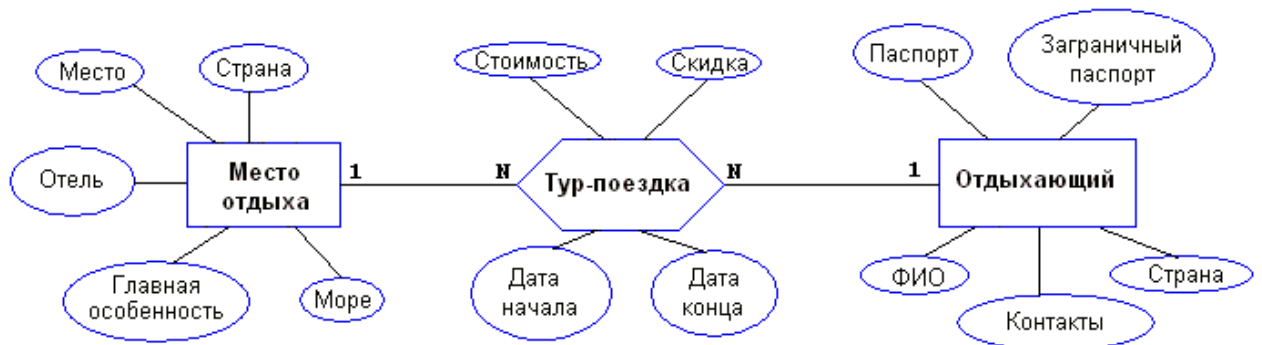
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №36

Предметная область для практических заданий №2.\* №3.\*: **Тур-фирма**

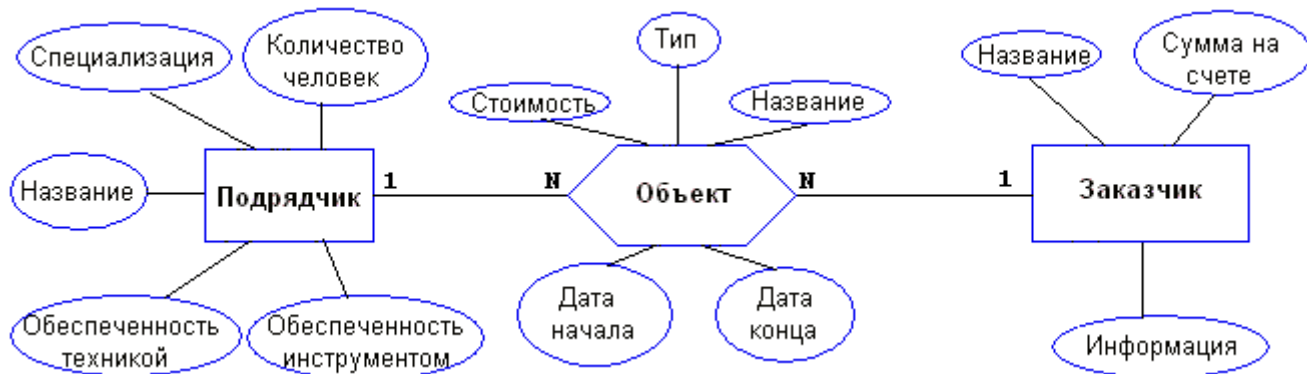
Пример схемы для задания №2.1, от которого можно отталкиваться:



## Вариант №37

Предметная область для практических заданий №2.\* №3.\*: **Строительство**

Пример схемы для задания №2.1, от которого можно отталкиваться:

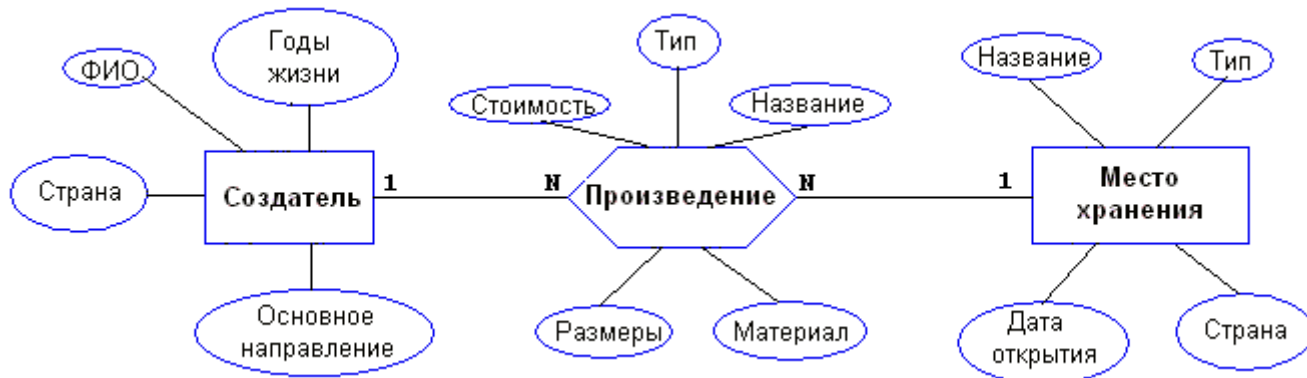


Тип объекта: промышленный, частный, специальный, хозяйственный и т.д

## Вариант №38

Предметная область для практических заданий №2.\* №3.\*: **Искусство**

Пример схемы для задания №2.1, от которого можно отталкиваться:



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.