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.

Лапшин А.В.

Фомин М.М.

Кудрявцев А.П.

Москва, 2023

# 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 понятием нормализации в БД.
* Изучить типы связей между сущностями и таблицами БД.
* Ознакомится с операторами создания БД.
* Изучение типов данных.
* Научится добавлять записи в таблицы.
* Научиться удалять и изменять записи в таблице.
* Ознакомиться с механизмами контроля согласованности БД, транзакциями и триггерами.

# 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 рис. 1.

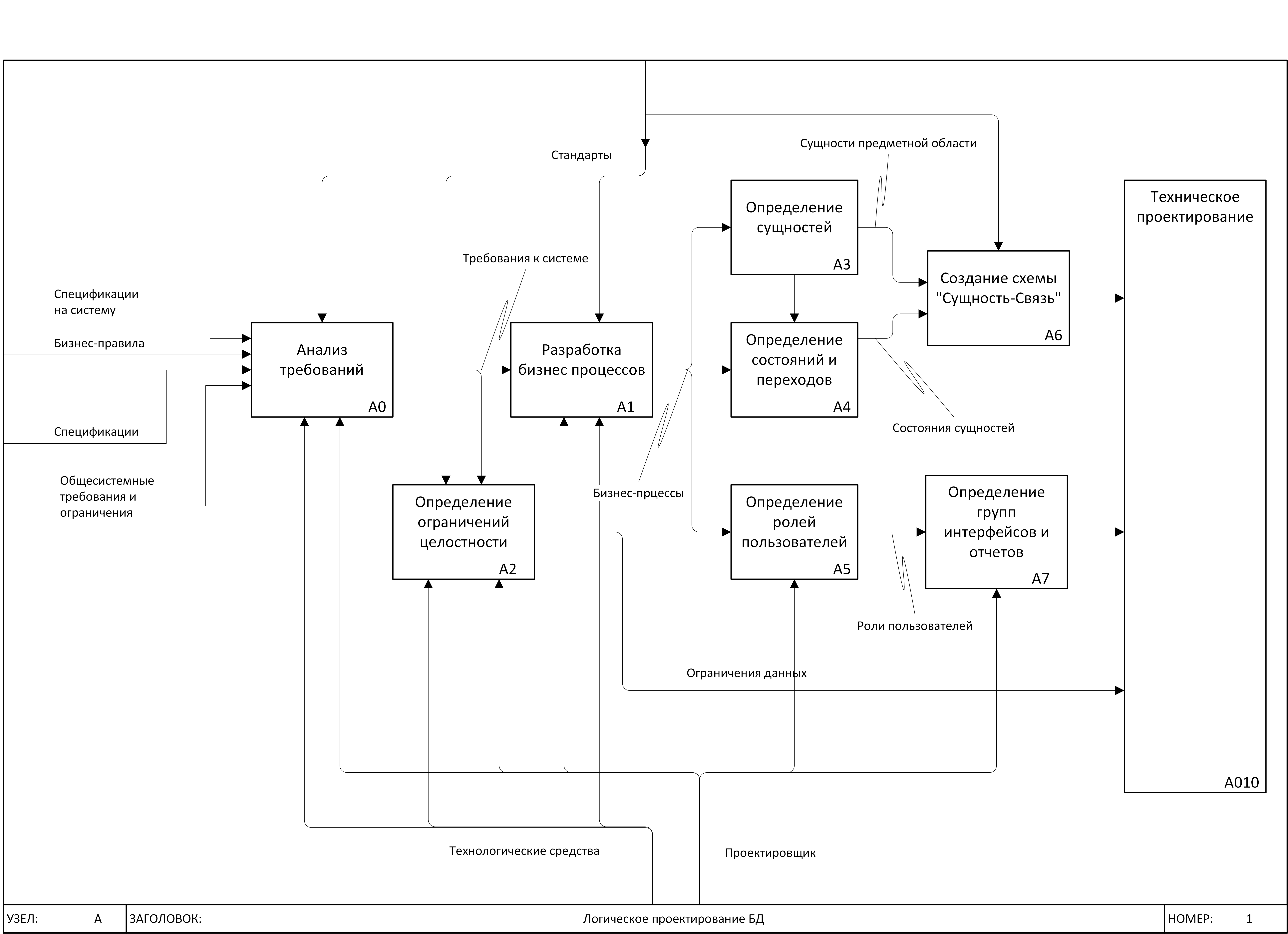


Рисунок 1. Бизнес-процесс логического проектирования БД

### 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. Пример на рис. 2.

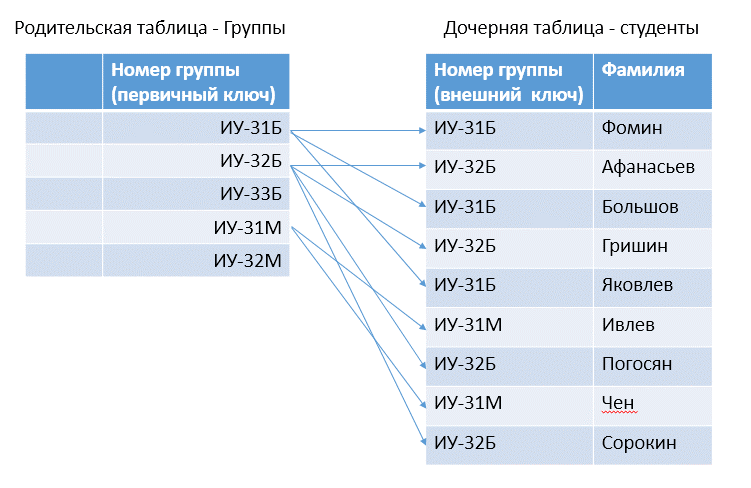


Рисунок 2. Связь между таблицами

В таблицах на рис. 2. 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» (НГ) можно по значениям внешнего ключа равным НГ выбрать всех студентов этой группы и наоборот, по значению внешнего ключа (НГ конкретного студента) можно найти нужный экземпляр сущности «Группа».

Так работают связи типа «один ко многим» для таблиц (сущностей) в реляционной БД.

Но ведь номер группы меняется каждый семестр и, следовательно, связывать таблицы по этому полю нельзя, иначе связи в вашей БД будут становиться некорректными каждый семестр. Чтобы такого не случилось, первичными ключами служат только суррогатные ключи. В настоящее время при проектировании БД первичным ключом сущности всегда служит специальный атрибут, не несущей никакой информации об экземпляре сущности, а только идентифицирующий ее - такой ключ называется **суррогатным**, а его значение генерируется СУБД при добавлении экземпляра сущности (записи в таблицу). При использовании суррогатных ключей связи между таблицами получатся как на рис. 3.

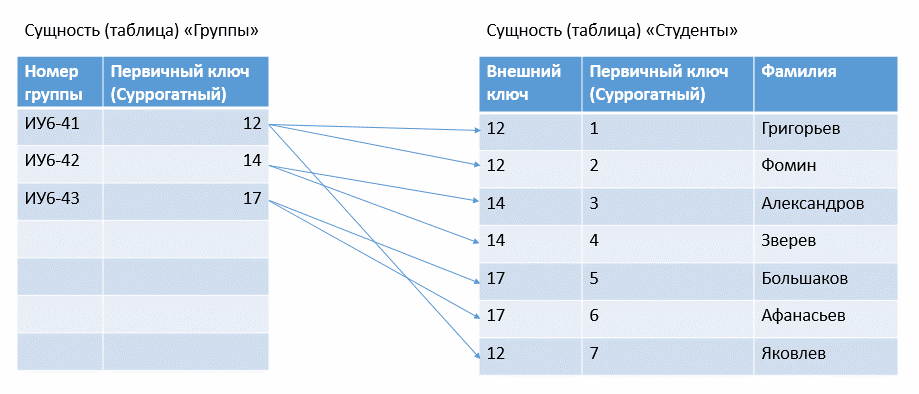


Рисунок 3. Связь между таблицами при помощи суррогатного ключа

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 базы данных «Авиаперелеты»

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

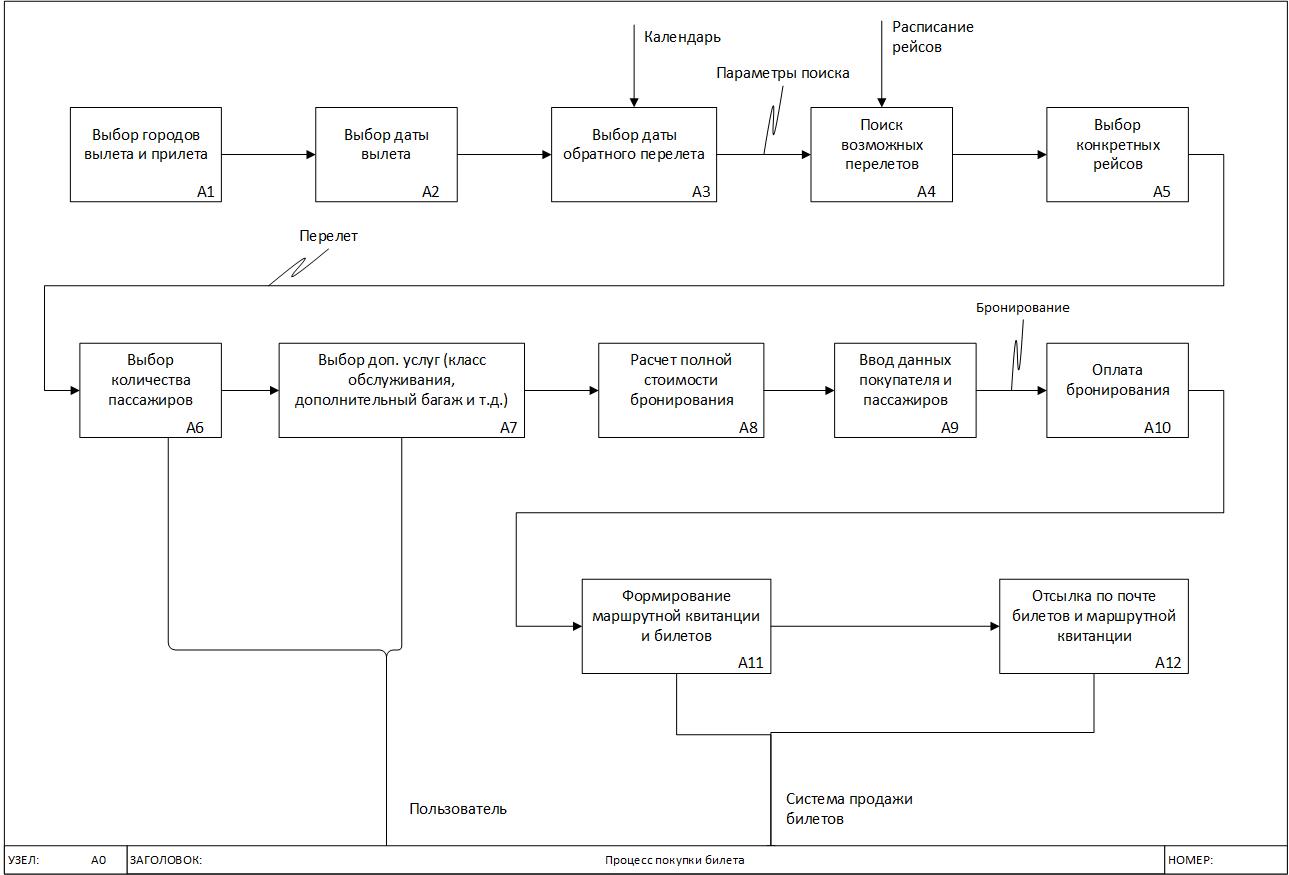


Рисунок 4. Бизнес-процесс покупки билета на сайте авиакомпании

Исходя из бизнес-процесса, показанного на рис. 4 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 упрощенную схему данных. Где связи обозначены в виде стрелок, которые означают виды как на рисунке 5. Сама схема данных приведена на рис 6.



Рисунок 5. Обозначение связей

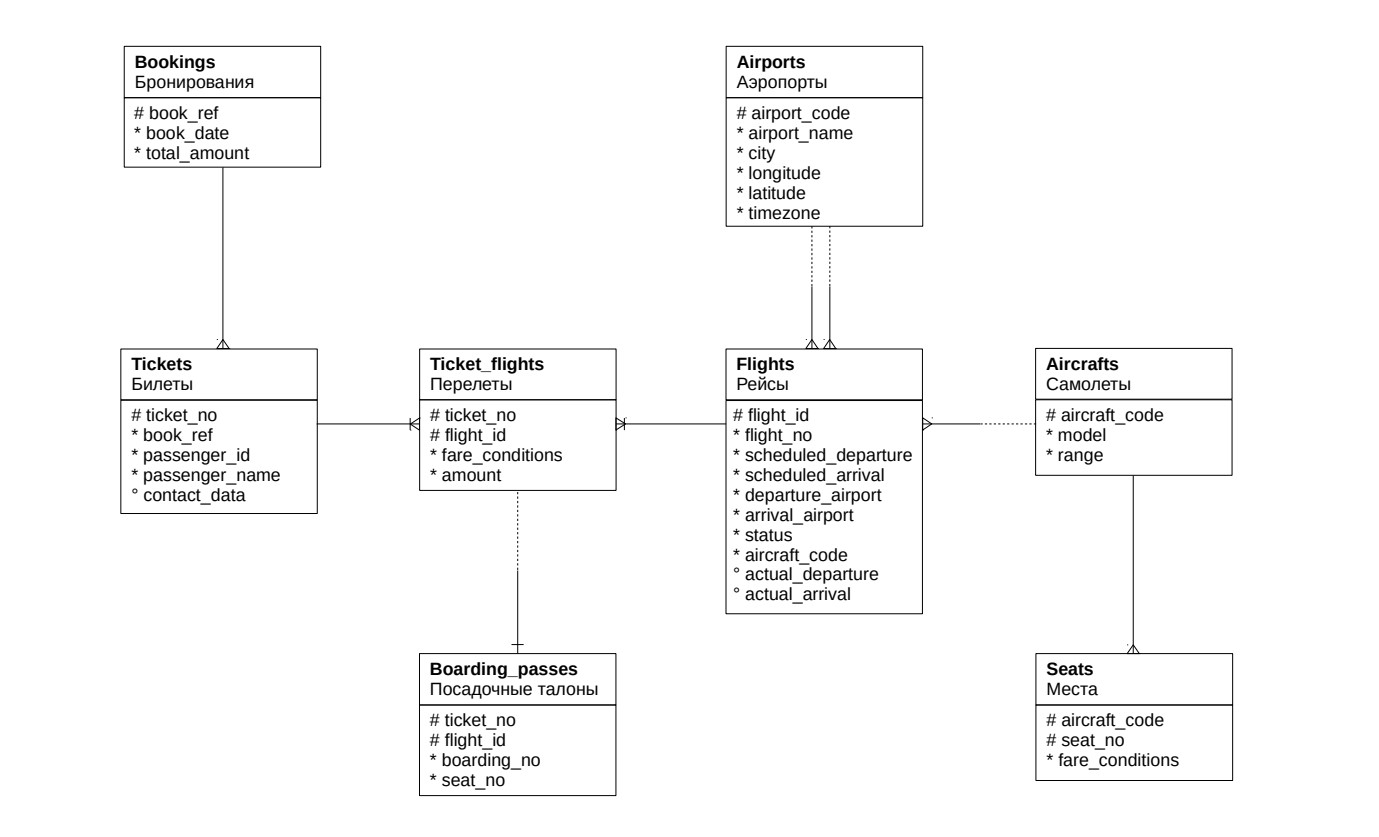


Рисунок 6. Схема данных

## 1.3. Практическое задание

Второе практическое задание связано с проектированием схемы базы данных для работы приложения (WEB/Mobile/Desktop). 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.**

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

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

## 1.3.1 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.

## 1.3.2. Темы для самостоятельной проработки

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

## 1.3.3. Вопросы для самостоятельной проработки

* **Понятие первичного и вторичного ключей.**
* **Чем отличается первичный ключ от индекса?**
* **Что такое кластерный ключ?**
* **Какие типы связей существуют?**

# 2. Создание и заполнения таблиц.

## 2.1. Теоретическая часть и практические примеры

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

### 2.1.1. Оператор 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 полей.
* Типы данных полей.
* Атрибуты и ограничения.

Реляционные базы данных хранят данные в таблицах, и каждая таблица содержит набор полей. У поля есть название и тип данных, так же можно указать атрибуты и ограничения, такие параметры как NOT NULL (то есть обязательное поля 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:

* Начинается с буквы;
* Иметь длину 1–30 символов ;
* Содержит только A–Z, a–z, 0–9, \_, $ и #;
* Не дублирует имя другого объекта, принадлежащего тому же самому пользователю.

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 ));

**Пояснения к написанию запроса:**

* Предложение CREATE TABLE указывает имя создаваемой таблицы и описывает ее поля.
* В скобках для каждого поля указывается два обязательных параметра: название и тип данных. Далее следуют ограничения, накладываемые на поле: в нашем случае, это ограничение, связанное с обязательным заполнением (NOT NULL) т.е. все поля таблицы аэропортов должны быть заполнены.
* После тогда, когда наполнили таблицу будущими полями последним значением указываем первичный ключ с помощью ключевого слова PRIMARY KEY и указанием в скобках поля для назначение его ключом.
* Двойное тире «--» означает, что далее в этой строке содержится комментарий (то есть часть кода, которая не будет прочитана транслятором).

Как это выглядит в pgAdmin можно посмотреть рисунке 7.

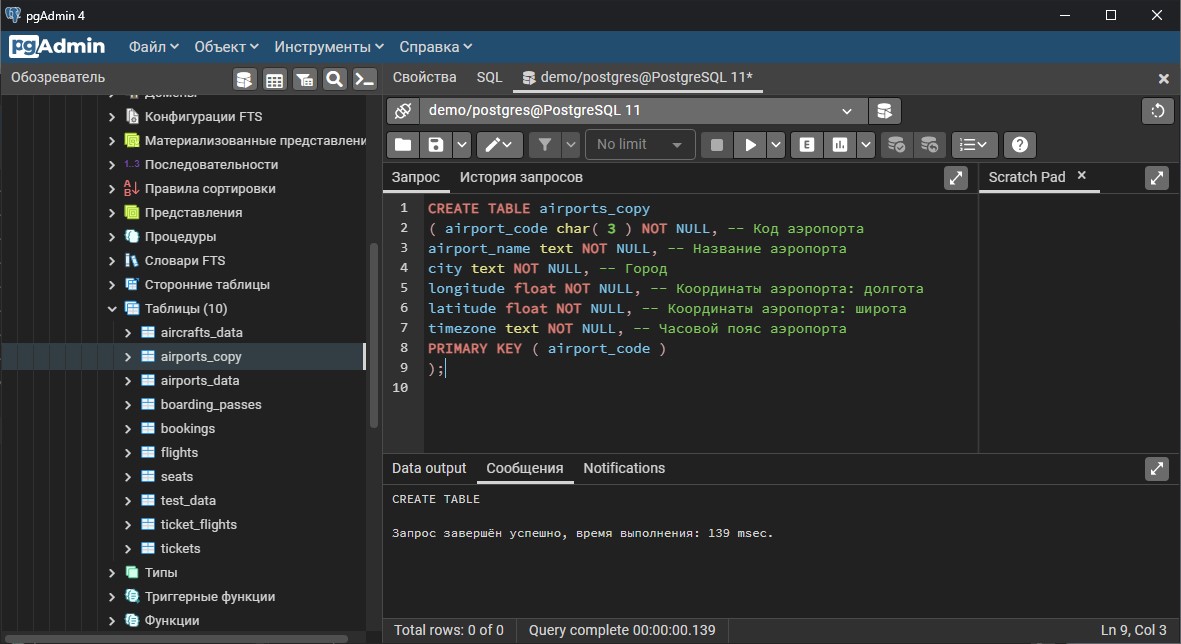


Рисунок. 7. Запрос создания таблицы

### 2.1.2. Оператор ALTER TABLE

Для изменения структуры таблицы используется оператор ALTER TABLE. Оператор ALTER TABLE также используется для переименования таблиц.

Предположим, что нам понадобилось добавить в базу данных сведения о крейсерской скорости полета всех моделей самолетов, которые эксплуатируются в нашей авиакомпании. Для этого необходимо добавить поле в таблицу «Самолеты» (aircrafts). Дадим ему имя speed (наверное, можно предложить и более длинное имя — cruise\_speed). Наложим ограничение на минимальное значение крейсерской скорости, выраженное в километрах в час: CHECK(speed >= 300). В результате в каждой записи должно появиться поле speed. Это поле не должно быть пустым, то есть надо добавить ограничение NOT NULL, правда тогда СУБД выдаст ошибку:

**«ОШИБКА: столбец "speed" содержит значения NULL»**.

Такая ошибка возникает из-за того, что добавленное поле не удовлетворяет поставленному ограничению. Для преодоления этой ситуации надо указать необходимость заполнения поля по умолчанию, как в запросе.

ALTER TABLE airports\_data

ADD COLUMN speed integer CHECK(speed >= 300) NOT NULL DEFAULT 500;

**Пояснения к написанию запроса:**

* Предложение ALTER TABLE указывает название таблицы, которая изменяется.
* Далее следуют операции с таблицей (в данном случае добавляется поле) ADD COLUMN с описанием поля, которое нужно добавить. Синтаксис описания как при создании таблицы (название, тип данных, ограничения).

Как выглядит запрос в среде pgAdmin можно посмотреть рисунке 8.

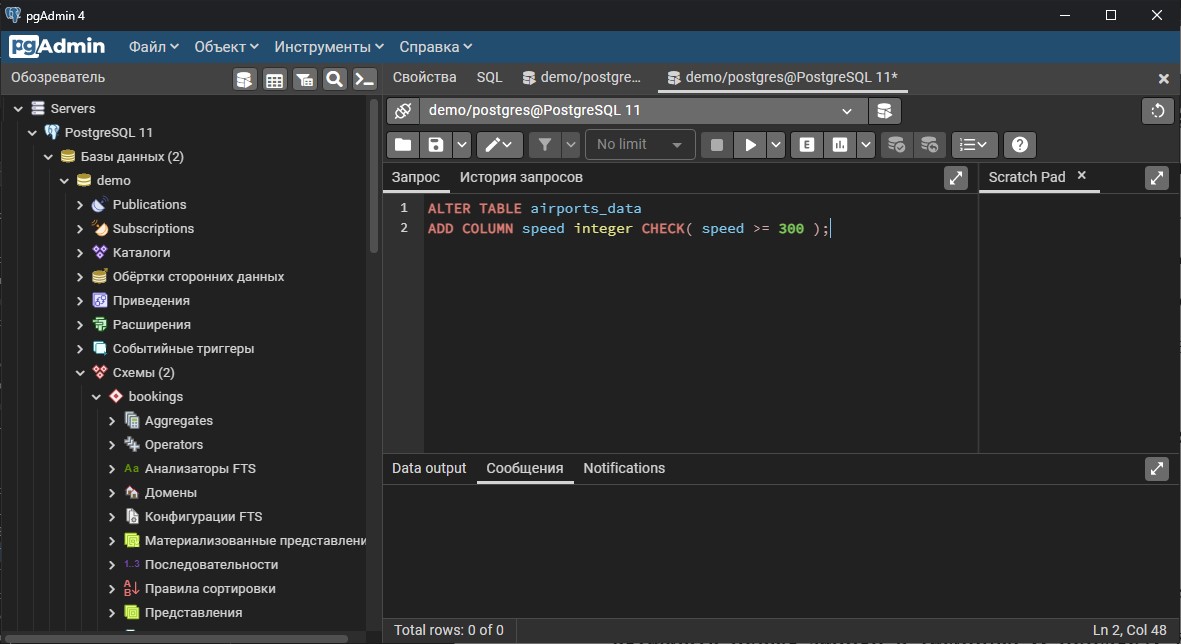


Рисунок 8. Изменение таблицы

### 2.1.3. ОГРАНИЧЕНИЯ

Типы данных сами по себе ограничивают множество данных, которые можно сохранить в таблице. Однако для многих приложений такие ограничения слишком грубые. Например, поле, содержащий цену продукта, должен принимать только положительные значения. Возможно, вы также захотите ограничить данные поля по отношению к другим полям или записям. 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 полей и таблиц. Ограничения дают вам возможность управлять данными в таблицах так, как вы захотите. Попытка сохранить в поле значение, нарушающее ограничения, приведет к ошибке.

### 2.1.4. Ограничения-проверки (CHECK)

Ограничение-проверка состоит из ключевого слова CHECK, за которым идёт условие в скобках, которое позволяет проверить, удовлетворяют ли данные определенным требованиям. Это условие обычно включает поле, для которого задаётся ограничение, иначе оно не имеет смысла.

### 2.1.5. Ограничения NOT NULL

Ограничение NOT NULL указывает, что поле не может быть пустым (содержать NULL), оно обязательно должно быть заполнено.

### 2.1.6. Ограничения уникальности (UNIQUE)

Ограничения уникальности гарантируют, что данные в определённом поле уникальны среди всех записей таблицы.

### 2.1.7. Ограничение первичного ключа (PRIMARY KEY)

Ограничение первичного ключа означает, что поле является уникальным идентификатором записей таблицы.

### 2.1.8. Ограничение внешнего ключа (REFERENCES)

Ограничение внешнего ключа указывает, что значения этого поля соответствуют значениям первичного ключа некоторой таблицы.

### 2.1.9. 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 утилиты psql, где «boarding\_passes» имя таблицы рисунок 9:

\d boarding\_passes

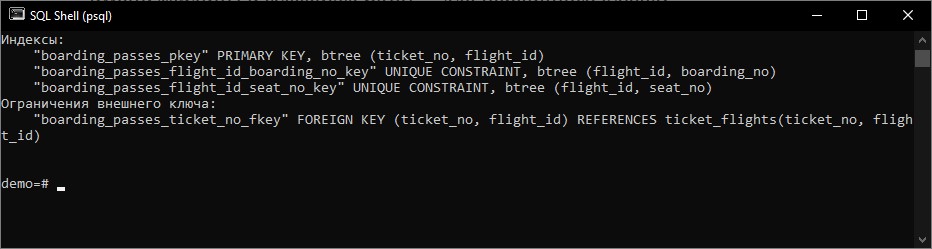


Рисунок 9 – Индексы

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.

В описании также присутствует список столбцов, по которым создан индекс, и тип индекса — в данном случае это btree, т. е. B-дерево. PostgreSQL может создавать индексы различных типов, но по умолчанию используется так называемое B-дерево. Такой индекс подходит для большинства типовых задач. В этой главе мы будем рассматривать только индексы на основе Bдерева. 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 airports\_data ( airport\_name );

Посмотрим описание нового индекса, рисунок 10:

\d airports

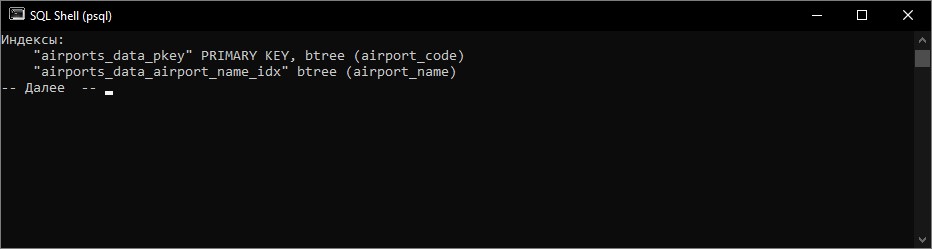


Рисунок 10 – Новый индекс

**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:

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

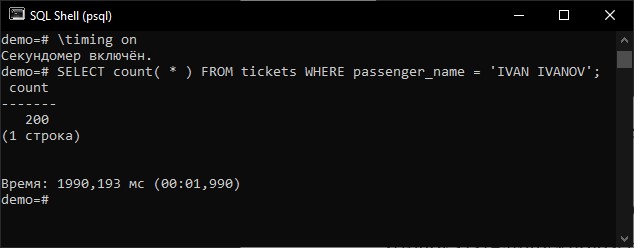


Рисунок 11 – Время без индекса

Показатели времени, полученные на вашем компьютере, конечно, будут отличаться от значений, приведенных в книге, 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, рисунок 12.

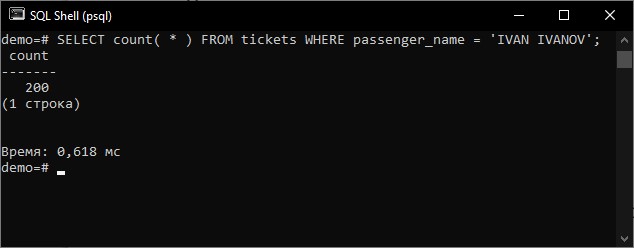


Рисунок 12 – Время с индексом

**Для удаления индекса** используется команда:

DROP INDEX 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 Oracle 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.

## 2.2. 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 полей).** Необходимо определить первичные и внешние ключи, а также ограничения полей (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. 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.

Необходимо подготовить два запроса:

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

Для каждого из этих запросов необходимо провести следующие шаги:

* Получить план выполнения запроса без использования индексов.
* Получить статистику (IO и Time) выполнения запроса без использования индексов.
* Создать нужные индексы, позволяющие ускорить запрос.
* Получить статистику выполнения запроса с использованием индексов и сравнить с первоначальной статистикой.

## 2.3. Темы для самостоятельной проработки

* Операторы DDL - 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

• Полное описание синтаксиса встретившихся команд https://postgrespro.ru/docs/postgrespro/11/sql-commands

* EXPLAIN

https://postgrespro.ru/docs/postgrespro/11/using-explain https://postgrespro.ru/docs/postgrespro/11/planner-stats https://postgrespro.ru/docs/postgrespro/11/explicit-joins

* Индексы

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

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

## 2.3. 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?
* Что такое IDENTITY?
* Рассказать о значениях по умолчанию и неопределенных значениях.
* Как можно хранить даты и время?
* Рассказать о числовых типах данных.
* В каких случаях имеет смысл создавать индексы? Какие колонки следует включать в индекс и почему?
* Рассказать о проблеме фрагментации индексов. Как бороться с фрагментацией?
* Имеет ли значение порядок указания колонок при создании индекса?

# 3. Операторы манипулирования данными (DML)

## 3.1. Теоретическая часть и практические примеры

### 3.1.1. Оператор INSERT

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.

|  |
| --- |
| INSERT INTO aircrafts\_data |
| VALUES  ('152', '{ "en": "Ti2-153N", "ru": "Ту-154М" }', 6601),  ('153', '{ "en": "Ti2-153N", "ru": "Ту-154М" }', 6601); |

**Пояснения к написанию запроса:**

* 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 выглядит запрос в среде pgAdmin можно посмотреть рисунке 13.

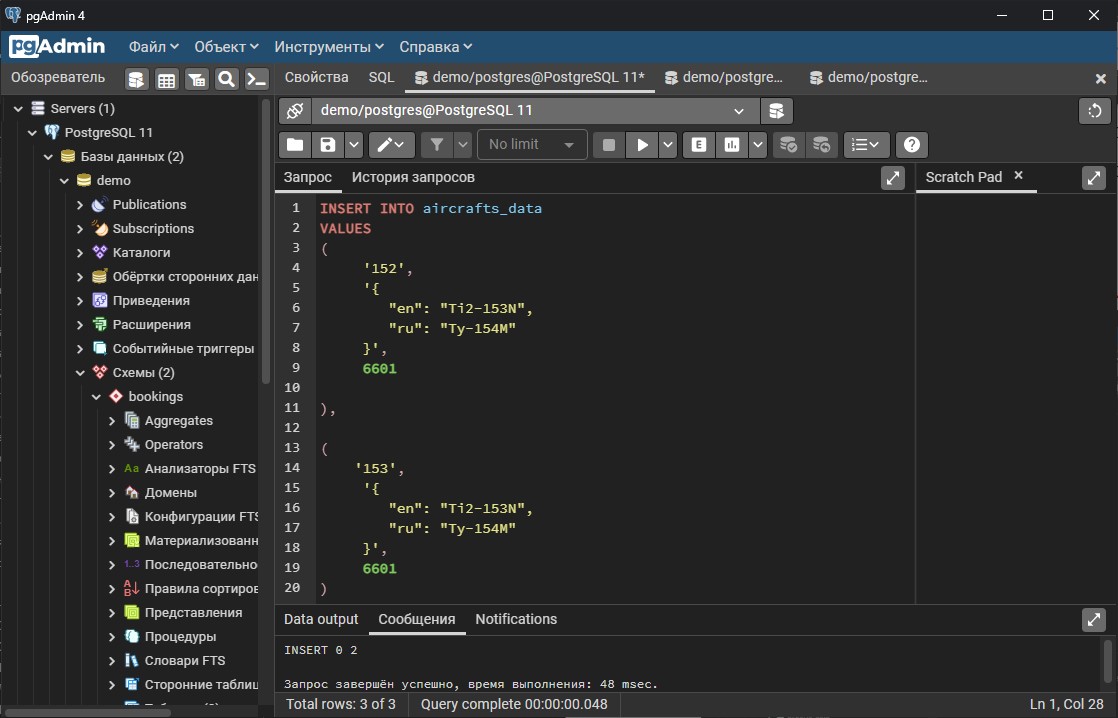


Рис. 13. Оператор INSERT

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

### 3.1.2. Оператор DELETE

Оператор DELETE предназначен для удаления записей из таблицы.

**Запрос 4:**

DELETE

FROM aircrafts\_data

WHERE aircraft\_code = ‘153’;

**Пояснения к написанию запроса:**

* Ключевое слово DELETE указывает на выполняемое действие (удаление);
* Предложение FROM указывает из какой таблицы будут удаляться записи (таблица aircraft\_code);
* Предложение WHERE необходимо для того, чтобы удалить конкретные записи из таблицы (самолет с номером = 153), при отсутствии предложения WHERE удалятся все записи в таблице.

Как выглядит работа запроса DELETE в среде pgAdmin можно посмотреть рисунке 14.

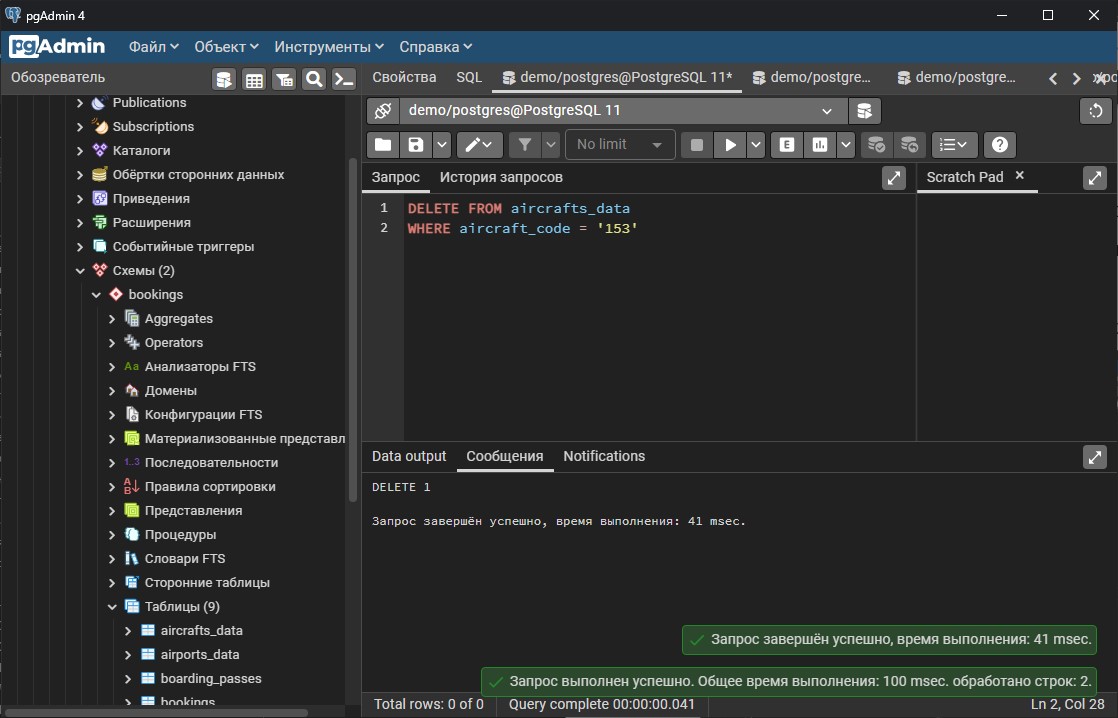


Рис. 14. Оператор DELETE

### 3.1.2. Оператор 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».

UPDATE aircrafts\_data

SET "range" = 6600, model = '{ "en": "Tu-154M", "ru": "Ту-154М" }'

WHERE aircraft\_code = '152';

**Пояснения к написанию запроса:**

* Ключевое слово UPDATE указывает имя таблицы, в которой будут изменены записи
* Предложение SET определяет поля таблицы и их новое значение
* Предложение WHERE определяет изменяемые записи ( в данном случае изменяются значения у записей с кодом самолета «152»)

Как выглядит запрос в среде pgAdmin можно посмотреть рисунке 15.

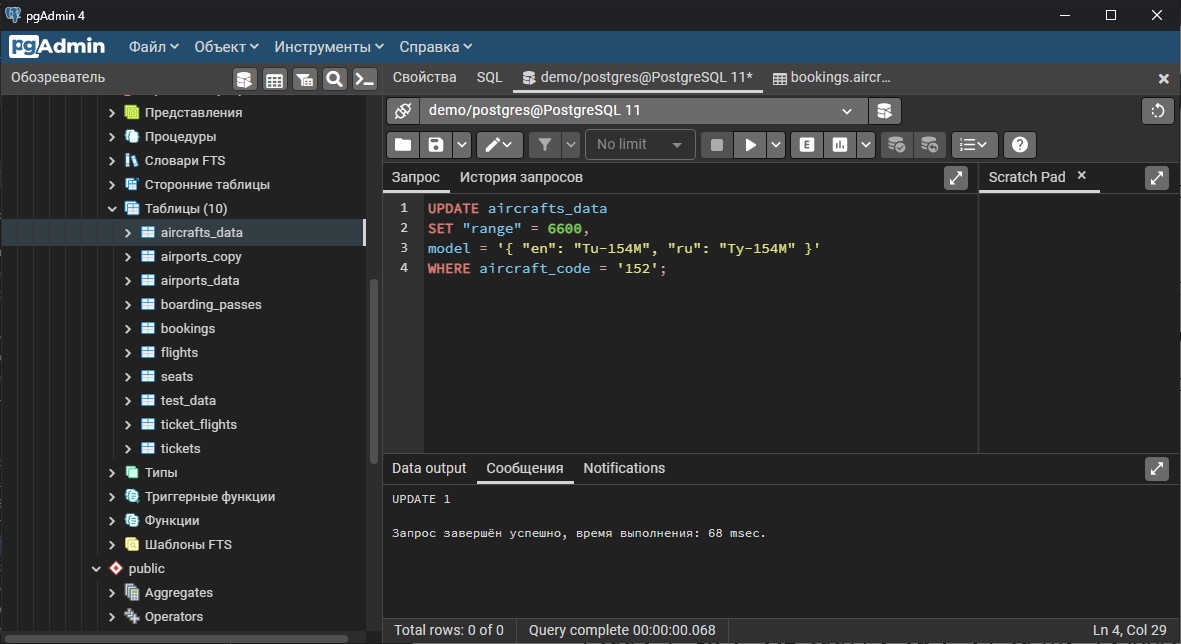


Рис. 15. Оператор UPDATE

## 3.2. 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.

## 3.3. 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

* Оператор INSERT.

https://postgrespro.ru/docs/postgrespro/11/dml-insert https://postgrespro.ru/docs/postgrespro/11/dml-returning

* Декларативные ограничения целостности.

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

• Полное описание синтаксиса встретившихся команд https://postgrespro.ru/docs/postgrespro/11/sql-commands

## 3.3. 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.
* Каким образом можно вставить несколько строк с помощью одного оператора INSERT?

# 4. Представления

## 4.1. Теоретическая часть

При работе с базами данных зачастую приходится многократно выполнять одни и те же запросы, которые могут быть весьма сложными и требовать обращения к нескольким таблицам. Чтобы избежать необходимости многократного формирования таких запросов, можно использовать так называемые представления. Если речь идет о выборке данных, то представления практически неотличимы от таблиц с точки зрения обращения к ним в командах SELECT.

Упрощенный синтаксис команды CREATE VIEW, предназначенной для создания представлений, таков:

CREATE VIEW имя-представления [(имя-столбца [, ...])] AS запрос;

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. Практическое задание

* Составить SQL-скрипты для создания нескольких представлений, которые позволяли бы упростить манипуляции с данными или позволяли бы ограничить доступ к данным, предоставляя только необходимую информацию.
* Продемонстрировать изменение и вставку данных через представления.
* Продемонстрировать невозможность изменения данных через представление.
* Продемонстрировать полезность материализованного представления.

## 4.3. Темы для самостоятельной проработки

* Представления.

https://postgrespro.ru/docs/postgrespro/11/sql-createview https://postgrespro.ru/docs/postgrespro/11/rules-views https://postgrespro.ru/docs/postgrespro/11/rules-materializedviews

* Полное описание синтаксиса встретившихся команд. https://postgrespro.ru/docs/postgrespro/11/sql-commands

## 4.4. Вопросы для самостоятельного изучения

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

# 5. Транзакции

## 5.1. Теоретическая часть и практические примеры

### 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 = 'Alice');

-- 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.

Таблица 1. Уровни изоляции и аномалии в соответствии со стандартом SQL2003

|  |  |  |  |
| --- | --- | --- | --- |
| **Уровень изоляции** | **Грязное чтение** | **Неповторяющееся чтение** | **Фантомные записи** |
| *READ UNCOMMITED* | Возможно | Возможно | Возможно |
| *READ COMMITED* | Невозможно | Возможно | Возможно |
| *REPEATABLE READ* | Невозможно | Невозможно | Возможно |
| *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 как на рисунке 16.

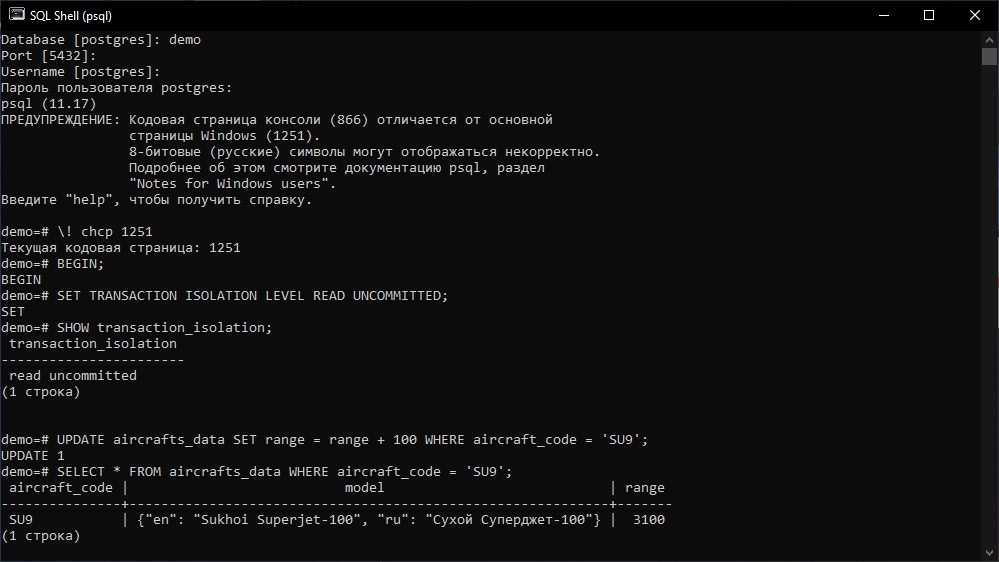


Рис. 16. Транзакция №1 (Терминал №1)

BEGIN;

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

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 17.

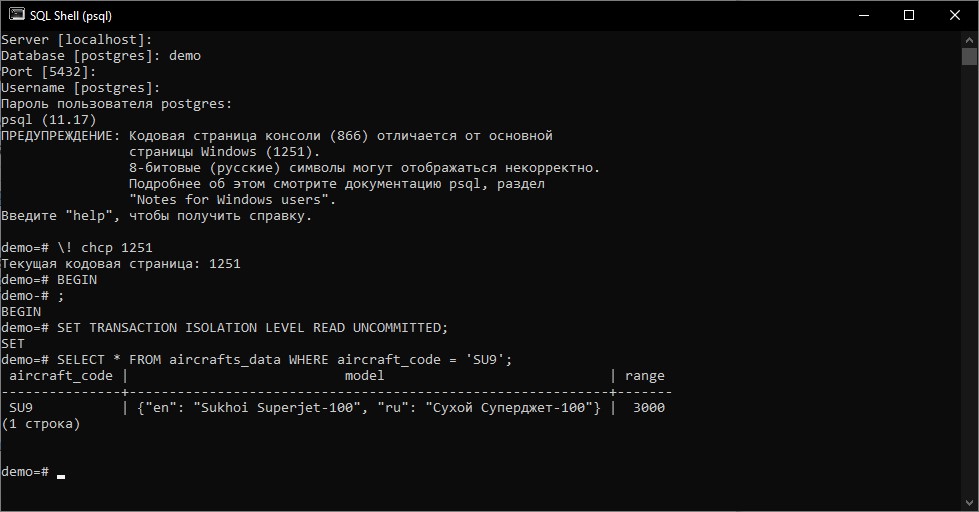


Рисунок 17 – Транзакция №1 (Терминал №2)

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 airctafts\_data, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута range не 100, а 200.

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 обоих консолях, для этого пишем команду «ROLLBACK;».

**READ COMMITTED.** Не допускается чтение «грязных» (незафикси-рованных) 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.

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 18.

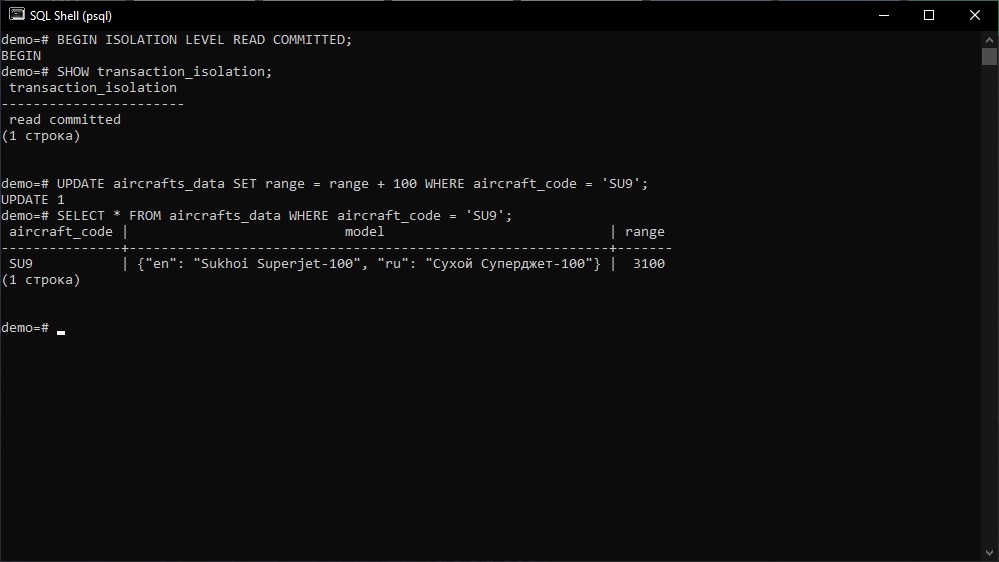


Рис. 18. Транзакция №2 (Терминал №1)

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 19. Во второй транзакции попытаемся обновить эту же строку таблицы airctafts\_data, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута range не 100, а 200.

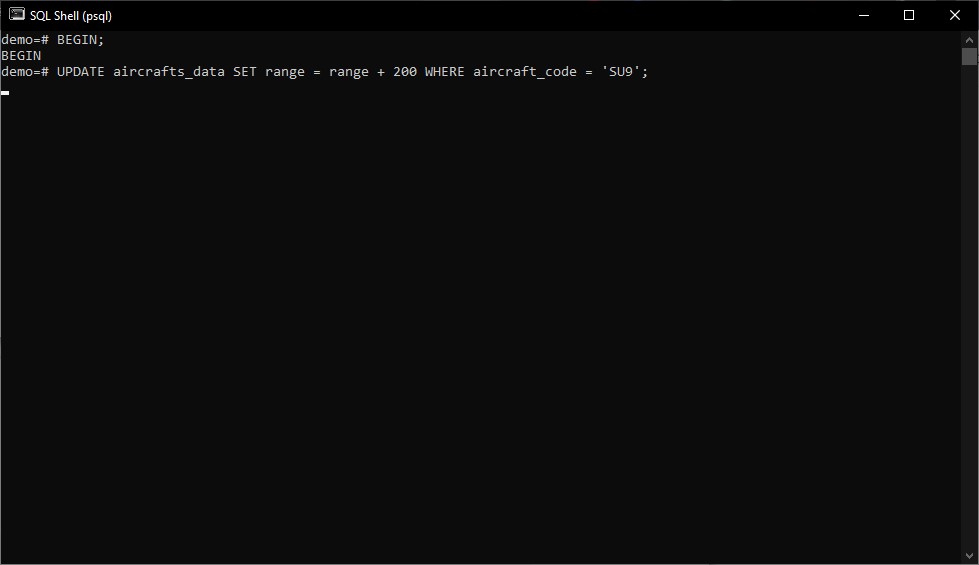


Рис. 19. Транзакция №2 (Терминал №2)

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\_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 второю консоль уже видим завершенную команду UPDATE, рисунок 20.

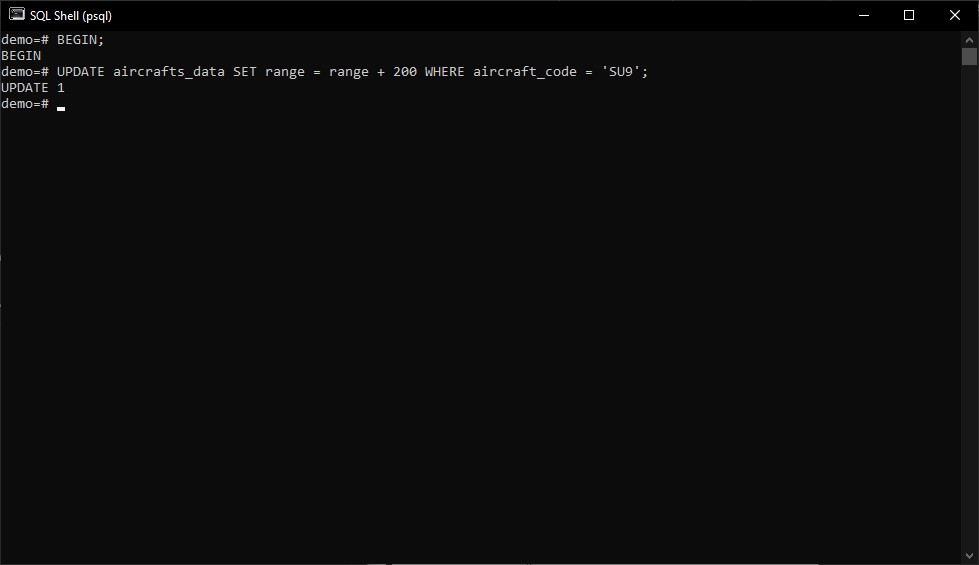


Рисунок 20. UPDATE (Терминал №2)

Завершим транзакцию во первой консоли с помощью оператора COMMIT.

Теперь на втором терминале, не завершая транзакцию, посмотрим, что стало с нашей строкой в таблице aircrafts\_data, рисунок 21.



Рисунок 21. 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 22.

BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

SELECT \* FROM aircrafts\_data;

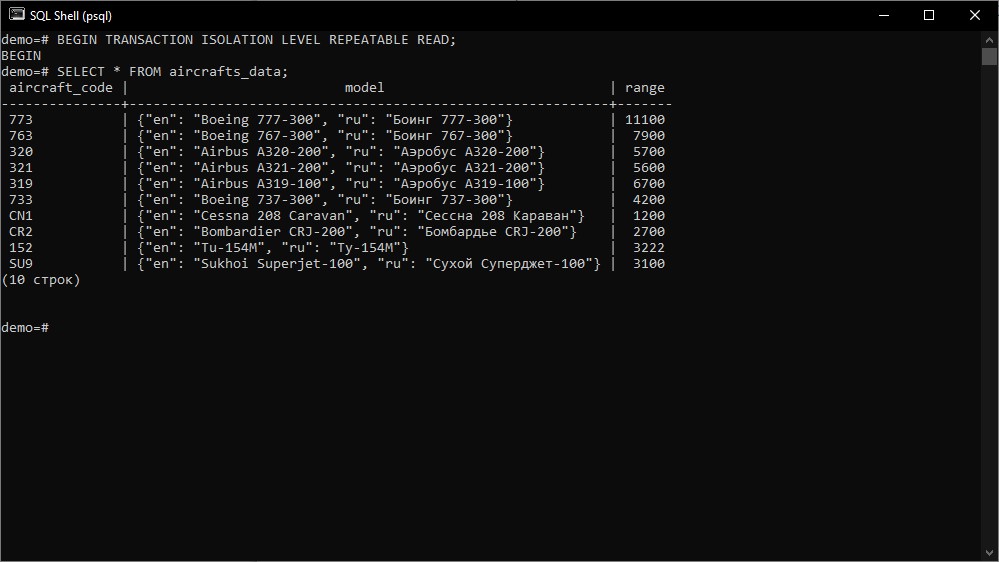


Рис. 22. Транзакция №3 (Терминал №1)

На втором терминале проведем ряд изменений, рисунок 23.

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;

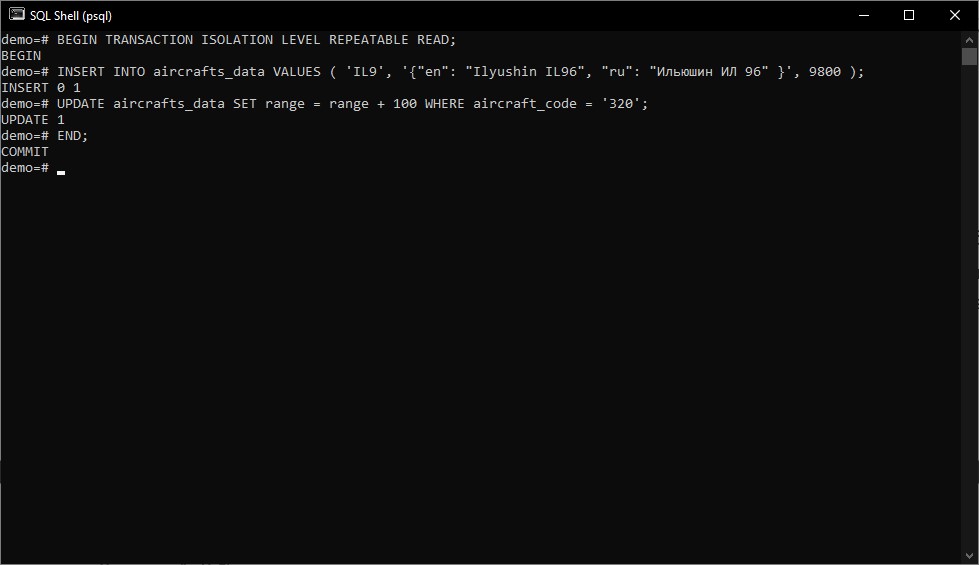


Рисунок 23. Транзакция №3 (Терминал №2)

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 24.

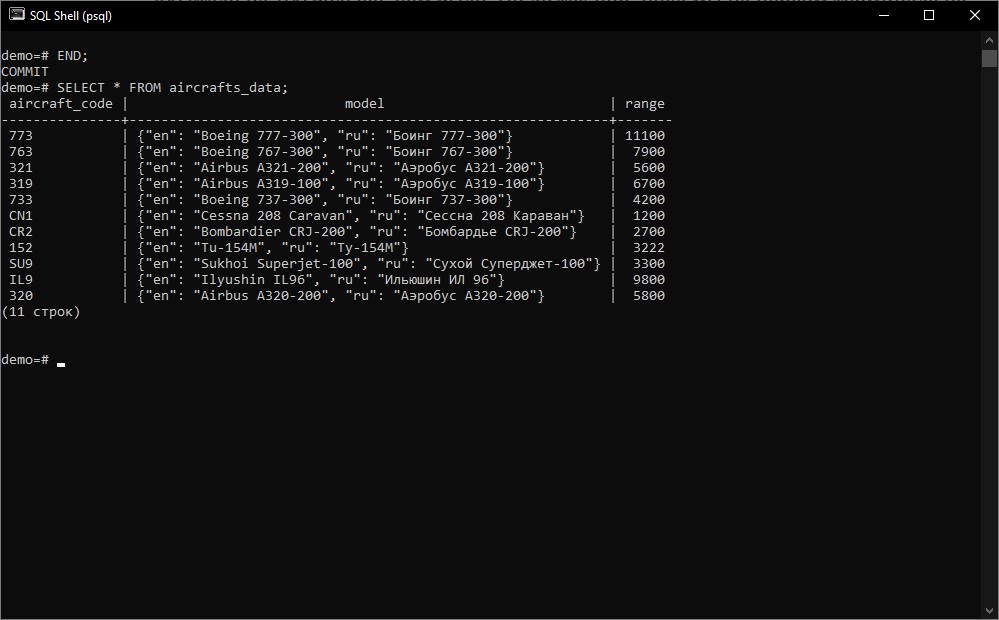


Рисунок 24. Транзакция №3 (Терминал №1)

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 поля: один — числовой, а второй — текстовый. Назовем эту таблицу — modes, рисунок 25. На первом терминале начнем транзакцию и обновим одну строку из тех двух строк, которые были показаны в предыдущем запросе.

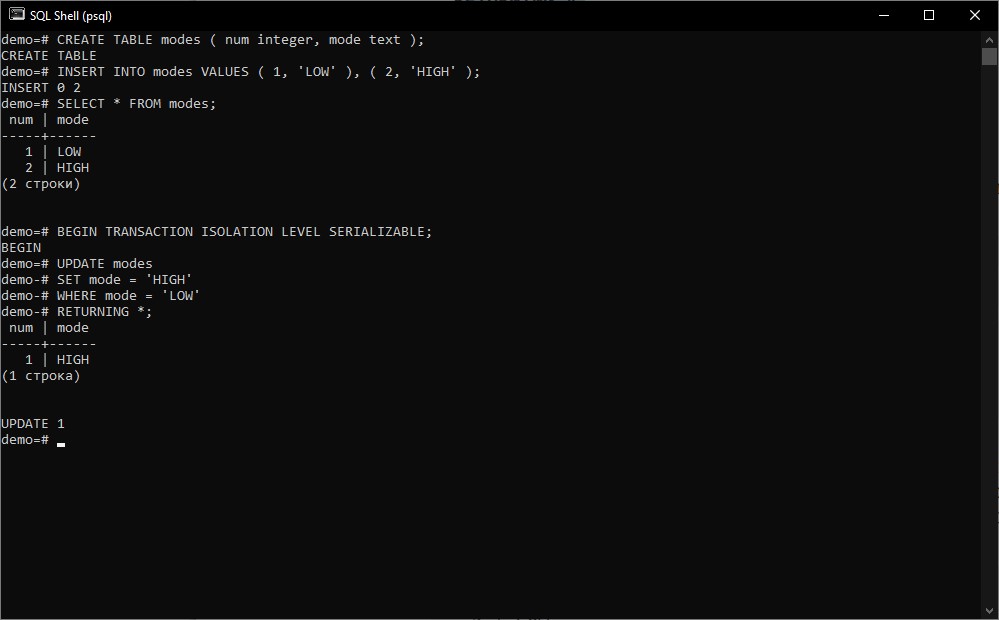


Рисунок 25. Транзакция №4 (Терминал №1)

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 26.

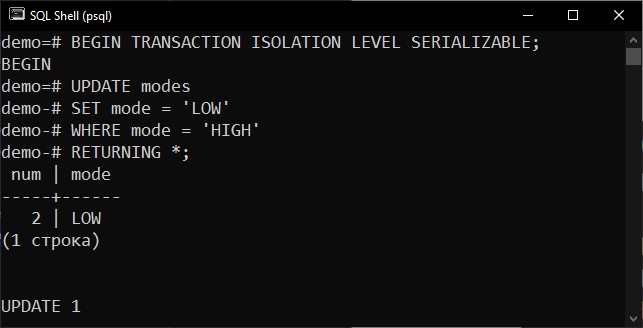


Рисунок 26. Транзакция №4 (Терминал №2)

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.

**Использование различных уровней изоляции транзакций**

В большинстве приложений уровень изолированности редко меняется и используется значение по умолчанию (например, в MySQL это **repeatable read**, в PostgreSQL — **read committed**).

Но периодически возникают, задачи, в которых поиск лучшего баланса между высокой согласованностью данных или скоростью выполнения транзакций может помочь решить некоторую прикладную задачу.

Выбирая уровень изолированности транзакций, мы пытаемся прийти к консенсусу в выборе между высокой согласованностью данных и скоростью выполнения этих самых транзакций.

## 5.5. Практическое задание

Практическое задание посвящено контролю целостности данных, который производится с помощью механизма транзакций. Транзакции позволяют рассматривать группу операций как единое целое, либо отрабатывают все операции, либо ни одной. Это позволяет избегать несогласованности данных. 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.

## 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

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 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?

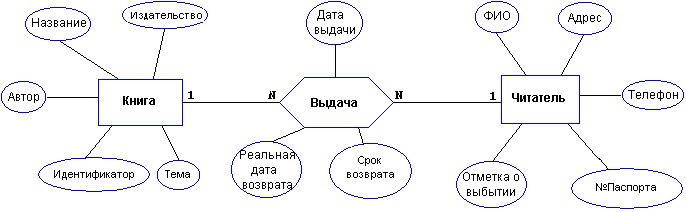
# Список использованных источников

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.
4. Стоунз Р., Мэтью Н. PostgreSQL. Основы. – Пер. с англ. – СПб: Символ\_Плюс, 2002. – 640 с., ил. ISBN 5\_93286\_043\_X
5. Болье, А. Изучаем SQL. Генерация, выборка и обработка данных, 3-е изд./ пер. с англ. И.В. Красикова. — Киев.: “Диалектика”,2021. — 402 с.: ил. ISBN 978-1-492-05761-1

# Приложение. Схемы предметных областей

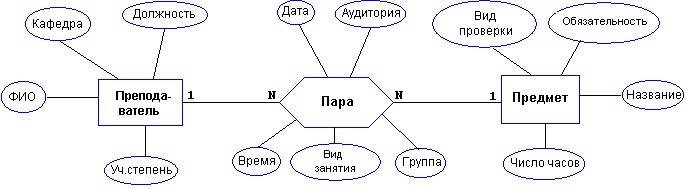
# Вариант №1

Предметная область для практических заданий: **Библиотека**



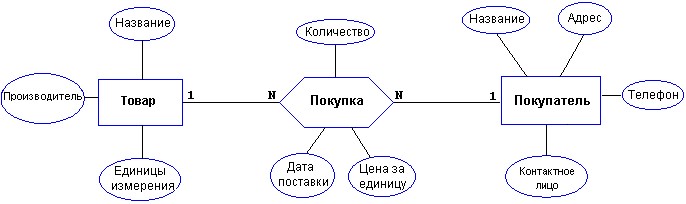
# Вариант №2

Предметная область для практических заданий: **Университет**



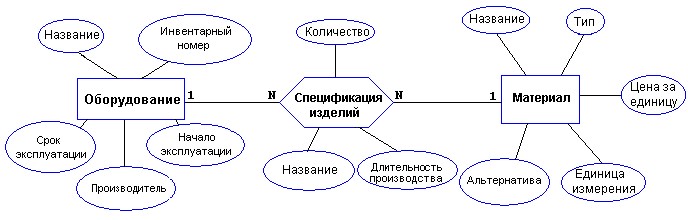
# Вариант №3

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



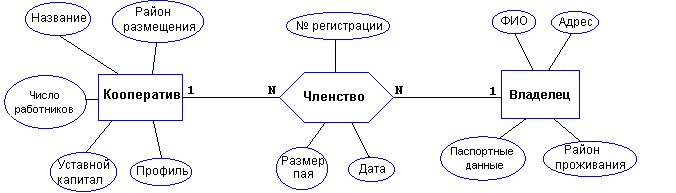
# Вариант №4

Предметная область для практических заданий: **Производство**



# Вариант №5

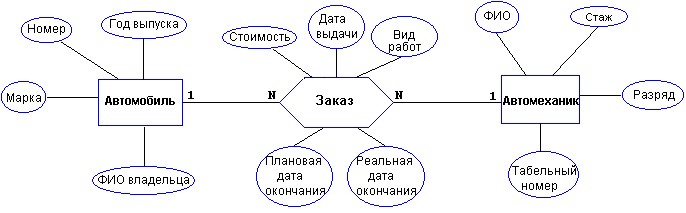
Предметная область для практических заданий: **Кооперативы**



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

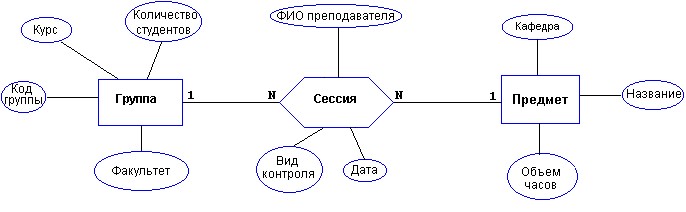
# Вариант №6

Предметная область для практических заданий: **Автомастерская**



# Вариант №7

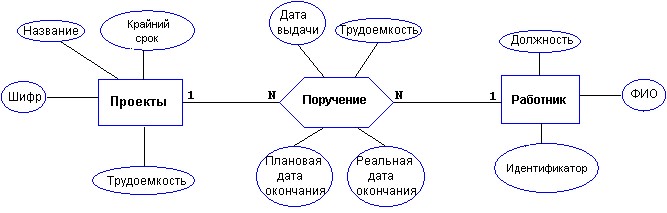
Предметная область для практических заданий:Сессия



Вид контроля - зачет, экзамен.

# Вариант №8

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

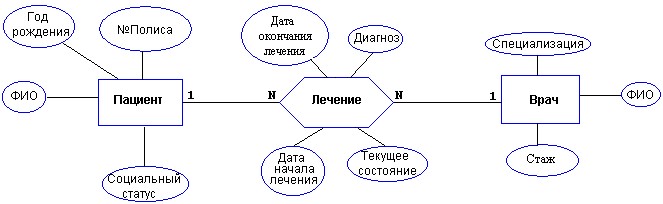


Категория дисциплины - гуманитарная, математическая, компьютерная и т.д.

Вид контроля - зачет, экзамен.

# Вариант №9

Предметная область для практических заданий: **Поликлиника**

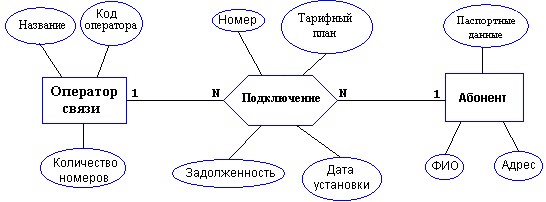


Текущее состояние - средней тяжести, тяжелое, направлен в стационар, умер. Социальный статус пациента - учащийся, работающий, врем. неработающий, инвалид, пенсионер

Специализация врача - терапевт, хирург и т.п.

# Вариант №10

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



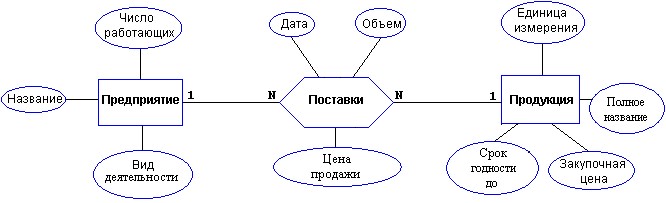
# Вариант № 11

Предметная область для практических заданий: **Спорт**



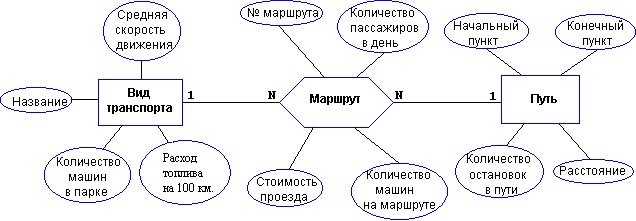
# Вариант №12

Предметная область для практических заданий: **Поставки**



# Вариант №13

Предметная область для практических заданий: **Транспорт**



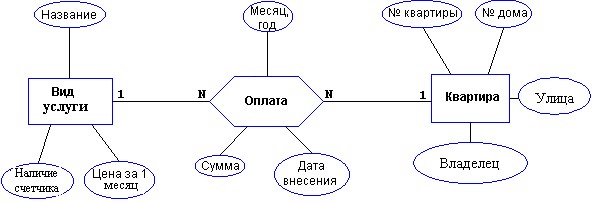
# Вариант № 14

Предметная область для практических заданий: **География**



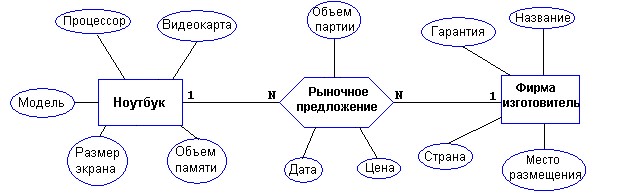
# Вариант №15

Предметная область для практических заданий: **Домоуправление**



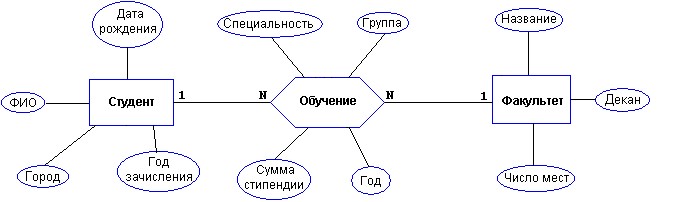
# Вариант №16

Предметная область для практических заданий: **Ноутбуки**



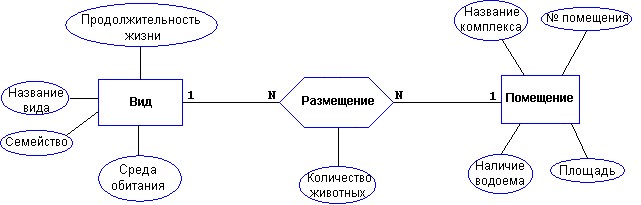
# Вариант №17

Предметная область для практических заданий: **Деканат**



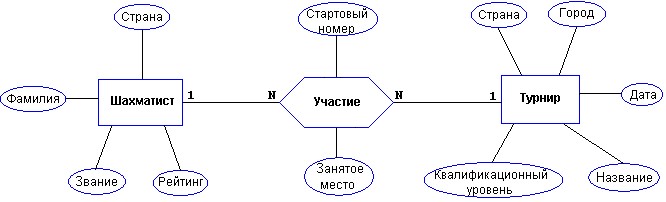
# Вариант №18

Предметная область для практических заданий: **Зоопарк**



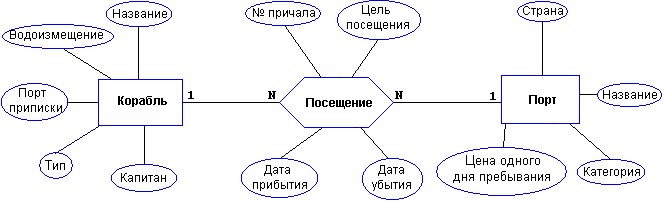
# Вариант №19

Предметная область для практических заданий: **Шахматы**



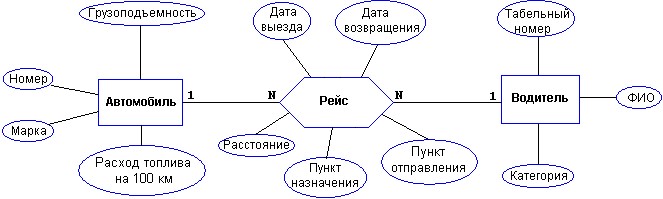
# Вариант №20

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



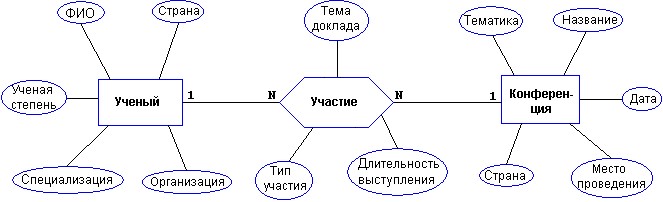
# Вариант №21

Предметная область для практических заданий: **Грузоперевозки**



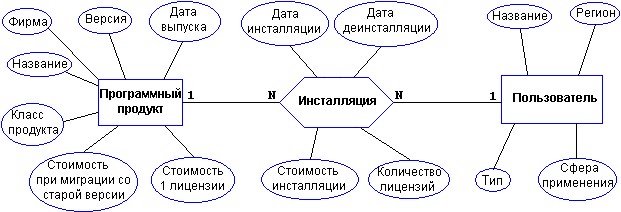
# Вариант №22

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



# Вариант №23

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

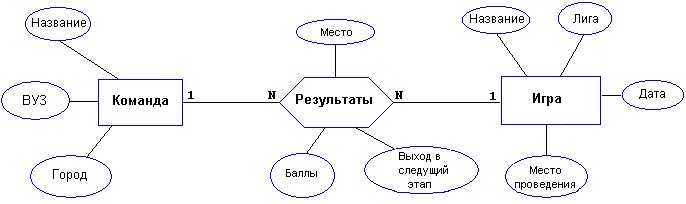


Класс: ОС, сервер приложений, СУБД, Web-сервер, система программирования и т.д.

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

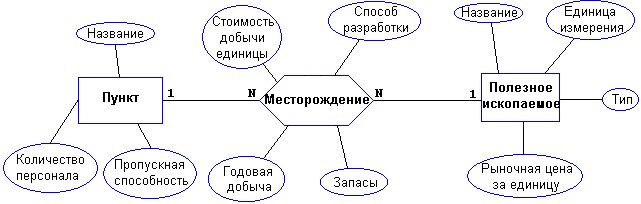
# Вариант №24

Предметная область для практических заданий: **КВН**



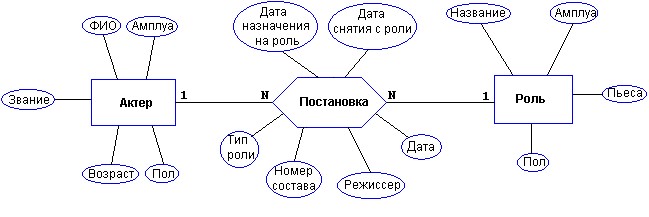
# Вариант №25

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



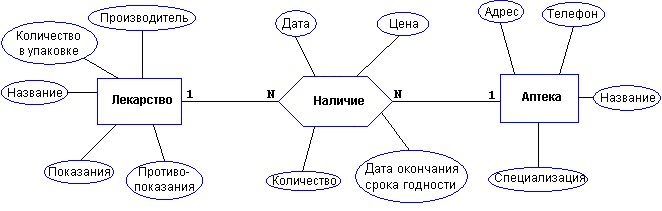
# Вариант №26

Предметная область для практических заданий: **Театр**



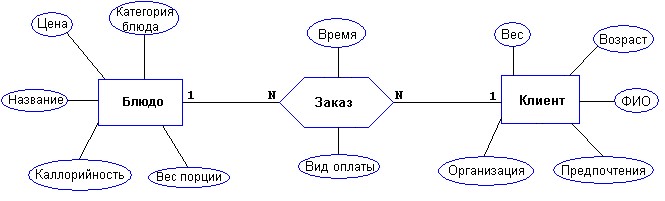
# Вариант №27

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



# Вариант №28

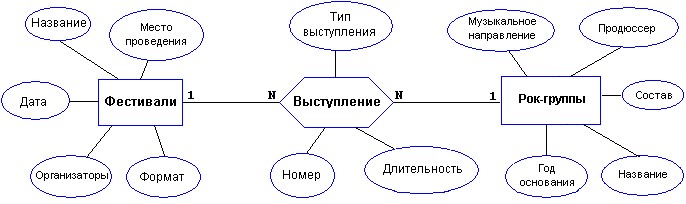
Предметная область для практических заданий: **Столовая**



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

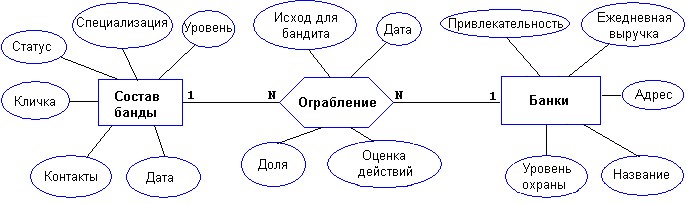
# Вариант №29

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



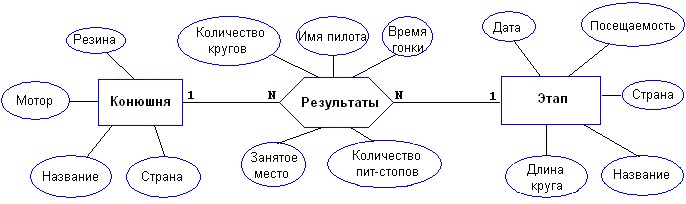
# Вариант №30

Предметная область для практических заданий: **ОПГ**



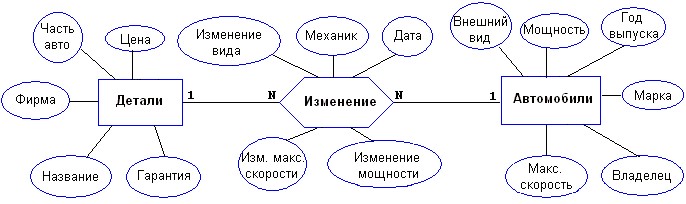
# Вариант №31

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



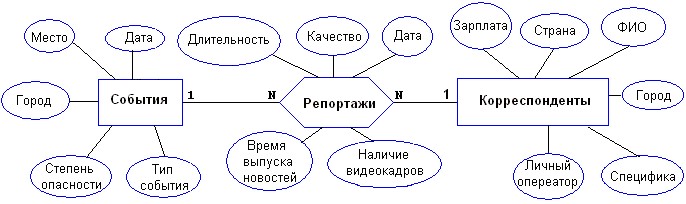
# Вариант №32

Предметная область для практических заданий: **Тюнинг**



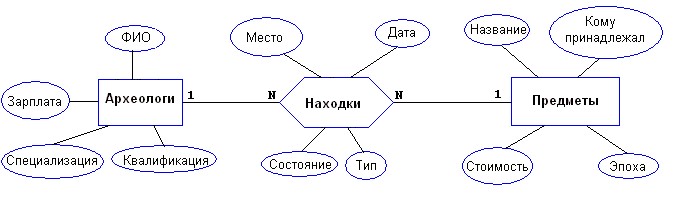
# Вариант №33

Предметная область для практических заданий: **Тележурналистика**



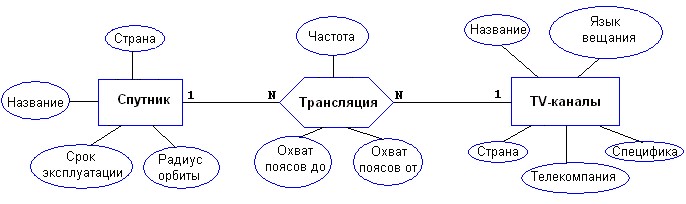
# Вариант №34

Предметная область для практических заданий: **Археология**



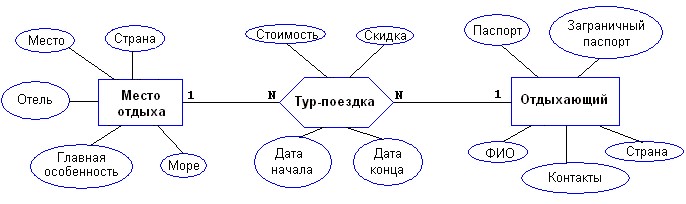
# Вариант №35

Предметная область для практических заданий: **Телевещание**



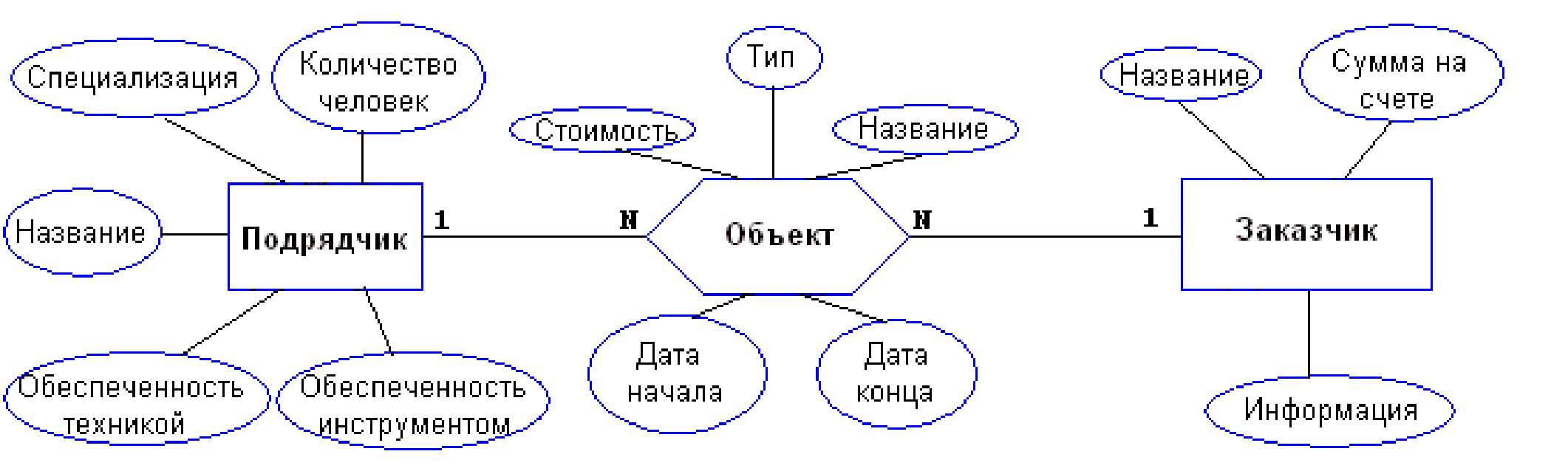
# Вариант №36

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.