Санкт-Петербургское государственное бюджетное
профессиональное образовательное учреждение
«Академия управления городской средой, градостроительства и печати»

УТВЕРЖДАЮ
Заместитель директора
по учебно-производственной работе
_____________ О.В. Фомичева
«26» ____12____ 2023 г.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ
по выполнению практических работ
по учебной дисциплине
ОП.05 ОСНОВЫ ПРОЕКТИРОВАНИЯ БАЗ ДАННЫХ

для специальности

09.02.06 Сетевое и системное администрирование

Санкт-Петербург
2023 г.

Методические рекомендации рассмотрены на заседании методического совета СПб ГБПОУ «АУГСГиП»

Протокол № 2 от «29» 11 2023 г.

Методические рекомендации одобрены на заседании цикловой комиссии информационных технологий

Протокол № 4 от «21» 11 2023 г.

Председатель цикловой комиссии: Караченцева М.С. ______________

Разработчики: преподаватели СПб ГБПОУ «АУГСГиП»

С О Д Е Р Ж А Н И Е

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА 4
1. Перечень практических работ по дисциплине «Основы проектирования баз данных» 5
2. Описание порядка выполнения практических работ 7
2.1. Практическая работа № 1 «Построение моделей данных» 7
2.2 Практическая работа № 2 «Построение реляционной модели данных. Определение ключей и связей между объектами» 7
2.3 Практическая работа № 3 «Выполнение операций реляционной алгебры» 9
2.4 Практическая работа № 4 «Проектирование БД. Анализ предметной области. Построение инфологической модели.» 11
2.5. Практическая работа № 5 «Приведение таблицы к нормальной форме. ER-диаграмма» 11
2.6. Практическая работа № 6 «Установка системы управления базами данных PostgreSQL» 11
2.7 Практическая работа № 7 «Работа с программой psql — интерактивным терминалом PostgreSQL» 13
2.8 Практическая работа № 8 «Основные операции с таблицами: добавление строк, упорядочивание по атрибутам. Группировка данных» 14
2.9 Практическая работа № 9 «Добавление ограничений» 23
2.10 Практическая работа № 10 «Модификация таблиц для нормализации отношений» 24
2.11 Практическая работа № 11 «Создание представлений» 27
2.12 Практическая работа № 12 «Работа с командой SELECT» 29
2.13. Практическая работа № 13 «Создание запросов на минимальные и максимальные значения» 30
2.14. Практическая работа № 14 «Создание подзапросов. Выборка данных» 34
2.15 Практическая работа № 15 «Работа с функцией unnest» 38
2.16 Практическая работа № 16 «Создание запросов с общим табличным выражением» 39
2.17 Практическая работа № 17 «Создание простых индексов» 41
2.18 Практическая работа № 18 «Создание индексов на основе выражений» 42
2.19 Практическая работа № 19 «Использование транзакций» 43
2.20 Практическая работа № 20 «Работа с командой EXPLAIN» 45
2.21 Практическая работа № 21 «Изменение схемы данных для повышения производительности базы данных» 49
2.22 Практическая работа № 22 «Модификация запросов для повышения производительности базы данных» 50
2.23. Практическая работа № 23 «Использование изоляций» 52
2.24. Практическая работа № 24 «Использование блокировок — встроенных механизмов защиты информации» 56

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Методические рекомендации по выполнению практических работ предназначены для организации работы на практических занятиях по учебной дисциплине «Основы проектирования баз данных», которая является важной составной частью в системе подготовки специалистов среднего профессионального образования по специальности 09.02.06 Сетевое и системное администрирование.

Практические занятия являются неотъемлемым этапом изучения учебной дисциплины и проводятся с целью:

- формирования практических умений в соответствии с требованиями к уровню подготовки обучающихся, установленными рабочей программой учебной дисциплины;
- обобщения, систематизации, углубления, закрепления полученных теоретических знаний;
- готовности использовать теоретические знания на практике.

Практические занятия способствуют формированию в дальнейшем при изучении профессиональных модулей, следующих общих и профессиональных компетенций:

ОК 01. Выбирать способы решения задач профессиональной деятельности применительно к различным контекстам.

ОК 02. Использовать современные средства поиска, анализа и интерпретации информации и информационные технологии для выполнения задач профессиональной деятельности.

ОК 03. Планировать и реализовывать собственное профессиональное и личностное развитие, предпринимательскую деятельность в профессиональной сфере, использовать знания по правовой и финансовой грамотности в различных жизненных ситуациях.

ОК 04. Эффективно взаимодействовать и работать в коллективе и команде.

ОК 05. Осуществлять устную и письменную коммуникацию на государственном языке Российской Федерации с учетом особенностей социального и культурного контекста.

ОК 09. Пользоваться профессиональной документацией на государственном и иностранном языках.

ПК 1.1. Документировать состояния инфокоммуникационных систем и их составляющих в процессе наладки и эксплуатации.

ПК 1.2. Поддерживать работоспособность аппаратно-программных средств устройств инфокоммуникационных систем.

В методических рекомендациях предлагаются к выполнению практические работы, предусмотренные учебной рабочей программой дисциплины «Основы проектирования баз данных».

При разработке содержания практических работ учитывался уровень сложности освоения студентами соответствующей темы, общих и профессиональных компетенций, на формирование которых направлена дисциплина.

Выполнение практических работ позволяет освоить комплекс работ по проектированию баз данных, организации SQL запросов, работе с СУБД, защите информации в базах данных.

Методические рекомендации имеют практическую направленность и значимость. Формируемые в процессе практических занятий умения могут быть использованы студентами в будущей профессиональной деятельности.

Оценки за выполнение практических работ выставляются по пятибалльной системе. Оценки за практические работы являются обязательными текущими оценками по учебной дисциплине и выставляются в журнале теоретического обучения.

1. Перечень практических работ по дисциплине «Основы проектирования баз данных»

№ раздела, темы	Освоение умений в процессе занятия	Тема практического занятия	Кол-во часов
Тема 1	проектировать реляционную базу данных	Практическая работа № 1 Построение моделей данных	2
Тема 2	проектировать реляционную базу данных	Практическая работа № 2 Построение реляционной модели данных. Определение ключей и связей между объектами	2
		Практическая работа № 3 Выполнение операций реляционной алгебры.	2
Тема 3	проектировать реляционную базу данных	Практическая работа № 4 Проектирование БД. Анализ предметной области. Построение инфологической модели.	2
		Практическая работа № 5 Приведение таблицы к нормальной форме. ER-диаграмма	2
Тема 4	работать с системами управления базами данных	Практическая работа№ 6 Установка системы управления базами данных PostgreSQL	2
		Практическая работа№ 7 Работа с программой psql — интерактивным терминалом PostgreSQL. Развёртывание учебной базы данных	2
Тема 5	проектировать реляционную базу данных использовать язык запросов для программного извлечения сведений из баз данных	Практическая работа№ 8 Основные операции с таблицами: добавление строк, упорядочивание по атрибутам. Группировка данных	2
		Практическая работа№ 9 Добавление ограничений.	2
		Практическая работа№ 10 Модификация таблиц для нормализации отношений.	2
		Практическая работа№ 11 Создание представлений.	2
		Практическая работа№ 12 Работа с командой SELECT.	2
		Практическая работа№ 13 Создание запросов на минимальные и максимальные значения.	2
		Практическая работа№ 14 Создание подзапросов. Выборка данных	2
		Практическая работа№ 15 Работа с функцией unnest.	2
		Практическая работа№ 16 Создание запросов с общим табличным выражением.	2

№ раздела, темы	Освоение умений в процессе занятия	Тема практического занятия	Кол-во часов
		Практическая работа№ 17 Создание простых индексов.	2
		Практическая работа№ 18 Создание индексов на основе выражений.	2
		Практическая работа№ 19 Использование транзакций.	2
		Практическая работа№ 20 Работа с командой EXPLAIN.	2
		Практическая работа№ 21 Изменение схемы данных для повышения производительности базы данных.	2
		Практическая работа№ 22 Модификация запросов для повышения производительности базы данных	2
Тема 6	использовать встроенные механизмы защиты информации в системах управления базами данных	Практическая работа№ 23 Использование изоляций	2
		Практическая работа№ 24 Использование блокировок — встроенных механизмов защиты информации	2

2. Описание порядка выполнения практических работ

2.1. Практическая работа № 1
«Построение моделей данных»

Задание: построить сетевую, иерархическую и реляционную модели для объекта по своему варианту.

Варианты заданий:

Вариант 1	Поликлиника	Вариант 9	Поставщик систем безопасности (видеонаблюдение)	Вариант 17	Дорожно-строительная компания
Вариант 2	Библиотека	Вариант 10	Провайдер	Вариант 18	Ферма
Вариант 3	Городская телефонная сеть (АТС, абоненты, оплата, переговоры).	Вариант 11	Компания, занимающаяся разработкой программного обеспечения	Вариант 19	УВД
Вариант 4	Авиарейсы (самолеты, пилоты, рейсы, пассажиры).	Вариант 12	НИИ	Вариант 20	Детский сад
Вариант 5	Предприятие торговли (отделы, товары, продавцы,...)	Вариант 13	Франчайзинговая компания	Вариант 21	Рекламное агентство
Вариант 6	Университет	Вариант 14	Банк	Вариант 22	Тракторный завод
Вариант 7	Строительная компания	Вариант 15	Страховая компания	Вариант 23	Казино
Вариант 8	Интернет-магазин компьютерной техники	Вариант 16	Компания-разработчик DLP-систем	Вариант 24	Гостиница

2.2 Практическая работа № 2
«Построение реляционной модели данных. Определение ключей и связей между объектами»

Задание:

1. Откройте программу Access и создайте новую базу данных по названию банка из вашего варианта. Настройте сохранение базы данных в Вашу папку.
2. Требуется создать 1 таблицу под названием Клиенты банка.
3. В таблице должны быть следующие поля: Код клиента, Фамилия, Имя, Отчество, пол, дата рождения, адрес, номер телефона, паспортные данные. Подобрать правильный тип данных для каждого поля. Пол клиента должен выбираться из фиксированного набора значений (мужской, женский).
4. Заполните таблицу для 5 клиентов вашего банка.
5. Для таблицы измените цвет фона, цвет текста, размер текста.

Варианты банков:

1 вариант	Сбербанк
2 вариант	ВТБ
3 вариант	Банк Санкт-Петербург
4 вариант	Балтийский банк
5 вариант	Альфа-банк
6 вариант	Внешпромбанк
7 вариант	Газпромбанк
8 вариант	Банк Открытие

6. Создать вторую таблицу к вашей базе данных под названием Кредиты банка.
7. В таблице должны быть следующие поля: Код кредита, Название кредита Процентная ставка по кредиту, Срок кредита, Условия кредита. Подобрать правильный тип данных для каждого поля. Для Срока кредита сделать фиксированный набор значений.
8. Заполните таблицу для 5 кредитов вашего банка.
9. Для таблицы измените цвет фона, цвет текста, размер текста.
10. Создать третью таблицу к вашей базе данных под названием Вклады банка.
11. В таблице должны быть следующие поля: Код вклада, Название вклада Процентная ставка по вкладу, Срок вклада, Условия вклада. Подобрать правильный тип данных для каждого поля. Для Срока вклада сделать фиксированный набор значений.
12. Заполните таблицу для 5 вкладов по вашему банку.
13. Для таблицы измените цвет фона, цвет текста, размер текста.
14. Откройте таблицу Клиенты банка. Добавьте поля Вклад банка и Кредит банка, Сумма. Создайте с помощью мастера подстановок раскрывающиеся списки из таблиц Вклады банка и Кредиты банка.
15. Заполните три новых поля данными в таблице Клиенты банка.
16. В каждой таблице создайте ключевые поля (тип данных счётчик).
17. Создайте схему данных (отношения один ко многим).

2.3 Практическая работа № 3
«Выполнение операций реляционной алгебры»

Задание:

Даны следующие таблицы из реляционной базы данных

ПРЕДПРИЯТИЕ

Пред#	Название	Рейтинг	Город
180	Электроника	230	Воронеж
230	Гормолзавод	300	Москва
150	Сельмаш	140	Воронеж
190	Хлебозавод	300	Курск
270	Рудгормаш	240	Москва

где

Пред# – номер предприятия, номер общий по некоторым группам городов;
Название – название предприятия;
Рейтинг – рейтинг предприятия по некоторым показателям;

ПРОДУКЦИЯ

Прод#	Наименование	Количество	Город
10	Магнитофоны	12000	Воронеж
20	Кровати	15000	Москва
30	Тракторы	20000	Воронеж
40	Кухни	30000	Орел
50	Продукты	10000	Воронеж

где

Прод# – номер продукции;
Наименование – наименование продукции;
Количество – количество продукции, выпускаемой в год в данном городе;

ЛИЧНОСТЬ

Лич#	Фамилия	Город	День_рожд	Пред#
55	Иванов	Воронеж	15.03.02	180
10	Петров	Москва	17.02.95	230
100	Сидоров	Воронеж	03.12.93	150
190	Иванов	Курск	18.04.91	190

где

Лич# – номер личности;
Пред# – номер предприятия, где работает данная личность.

ПРЕД_ПРОД

Пред#	Прод#	Год	Выработка
150	30	2000	150
180	10	2000	100
190	50	2001	50
230	50	2001	120
270	20	2002	50

где
Пред# – номер предприятия;
Прод# – номер продукции;
Год – год выпуска продукции;
Выработка (тыс.руб) – количество продукции данного предприятия.
Составить выражения реляционной алгебры по следующим пунктам и показать результирующую таблицу:

1) Получить названия предприятий, производящих продукцию с номером 30.
2) Получить имена предприятий, производящих продукцию всех сортов.
3) Получить номера предприятий, производящих по крайней мере ту продукцию, которую выпускает предприятие с номером 190.
4) Получить пары предприятий, находящихся в одном городе.
5) Получить имена предприятий, не производящих продукцию с номером 50.
6) Выбрать информацию обо всех предприятиях.
7) Выбрать название продукции, у которой количество потребления в городе находится в диапазоне от 12000 до 15000.
8) Выбрать название предприятий, у которых выработка продукции в 2001 г. на единицу работающего составила 100 тыс. руб.
9) Выбрать название предприятий, которые производят заданную продукцию.
10) Выбрать все пары названий городов, для которых предприятие из первого города, а интересующая продукция во втором городе.
11) Выбрать фамилии людей, которые работают на предприятии с заданным названием (?).
12) Выбрать фамилии людей, у которых город проживания совпадает с городом нахождения предприятия.
13) Определить фамилии людей, начинающиеся на «И», работающих на предприятии с заданным названием предприятия.
14) Определить название предприятий, производящих продукцию с заданным номером (?) в заданном году (?).
15) Определить название продукции с заданным номером, имеющей выработку на единицу работающего > 100 тыс. руб.
16) Определить номера предприятий из Воронежа с рейтингом выше 20.
17) Определить номера предприятий, имеющих в списке работающих по крайней мере одного «Иванов».
18) Найти имена предприятий, производящих по крайней мере одну продукцию с номером 50.
19) Определить номера продукции, производящих по крайней мере все виды продукции, производимые предприятием с номером 270.
20) Получить номера продукции, которая имеет количество более 15000, либо производится предприятием с номером 270, либо то и другое.

2.4 Практическая работа № 4
«Проектирование БД. Анализ предметной области. Построение инфологической модели.»

Задание:

1. Спроектировать базу данных для предметной области по вашему варианту:

✓ проанализировать предметную область, то есть определить количество необходимых таблиц (не меньше пяти);

✓ определить необходимые атрибуты для отношений;

✓ определить связи отношений (1 N);

✓ отношения должны быть не меньше, чем во второй нормальной форме.

2. Построить инфологическую модель вашей базы данных.

Вариант выбрать тот же, что был в Практической работе № 1.

2.5. Практическая работа № 5
«Приведение таблицы к нормальной форме. ER-диаграмма»

Задание:

1. Построить ER-диаграмму по вашему варианту.
2. Создать базу данных по вашему варианту в MS Access.
3. Создать необходимые связи.
4. Добавить 4-ую таблицу к вашей базе данных.
5. Привести все таблицы к 3-ей нормальной форме.
6. Создать запросы по заданию к вашему варианту.

2.6. Практическая работа № 6
«Установка системы управления базами данных PostgreSQL»

Задание:

Установка на Windows

1. С официального сайта скачать PostgreSQL версии 11.5 для Windows 64x и установить на вашу виртуальную машину с Windows 10. (при необходимости можно использовать пояснения по ссылке: https://o7planning.org/ru/10713/installing-and-configuring-postgresql-database)

2. Запомнить/записать пароль и порт, по которому работает PostgreSQL.

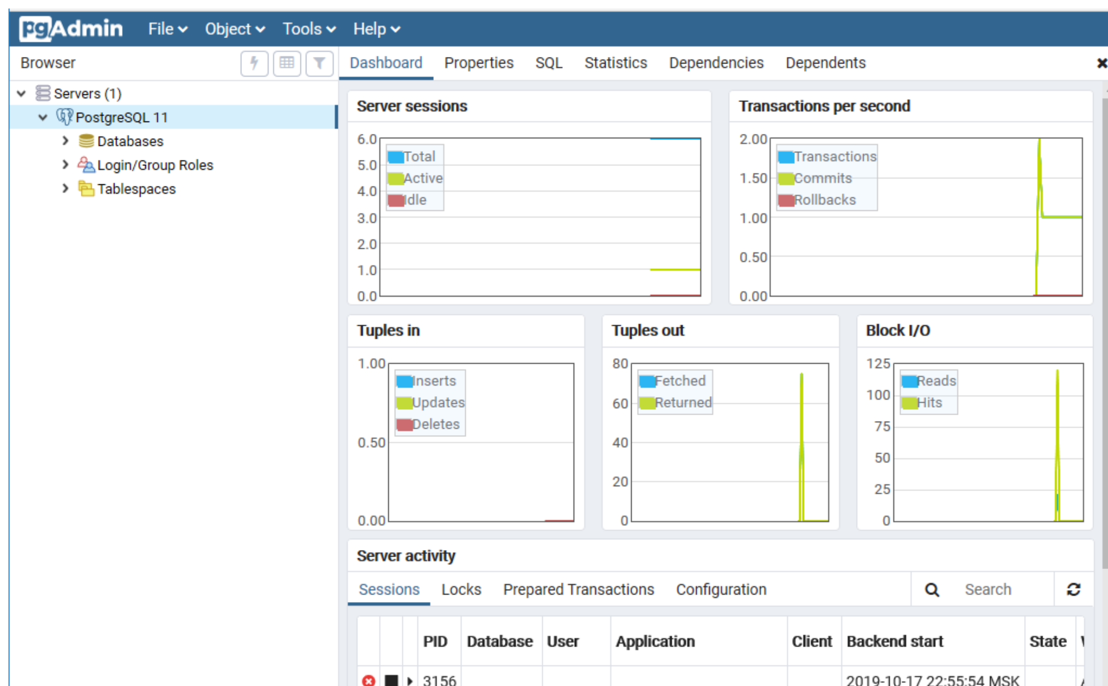
3. За время установки и по завершении установки для отчёта сделайте скриншот.

4. После завершения установки перезагрузите вашу виртуальную машину.

5. По ссылке https://edu.postgrespro.ru/demo_small. zip скачайте тестовую базу данных для развёртывания.

6. Запустите из папки с PostgreSQL pgAdmin4. Убедитесь, что он запускается и вы в него заходите. Сделайте для отчёта скриншот.

При правильной работы вы должны увидеть:



Установка на Linux

7. Import the repository key from https://www.postgresql.org/media/keys/ACCC4CF8.asc:

```
sudo apt-get install curl ca-certificates gnupg
curl https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add -
```

8. Create /etc/apt/sources.list.d/pgdg.list. The distributions are called codename-pgdg. In the example, replace buster with the actual distribution you are using:

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

9. (You may determine the codename of your distribution by running lsb_release -c). For a shorthand version of the above, presuming you are using a supported release:

```
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt/ $(lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list'
```

10. Finally, update the package lists, and start installing packages:

```
sudo apt-get update
sudo apt-get install postgresql-11 pgadmin4
```

11. Alternately, this shell script will automate the repository setup. The script is included in the postgresql-common package in Debian and Ubuntu, so you can also run it straight from there:

```
sudo apt install postgresql-common
sudo sh /usr/share/postgresql-common/pgdg/apt.postgresql.org.sh
```

12. Сброс пароля:

sudo -u postgres psql postgres

\password postgres

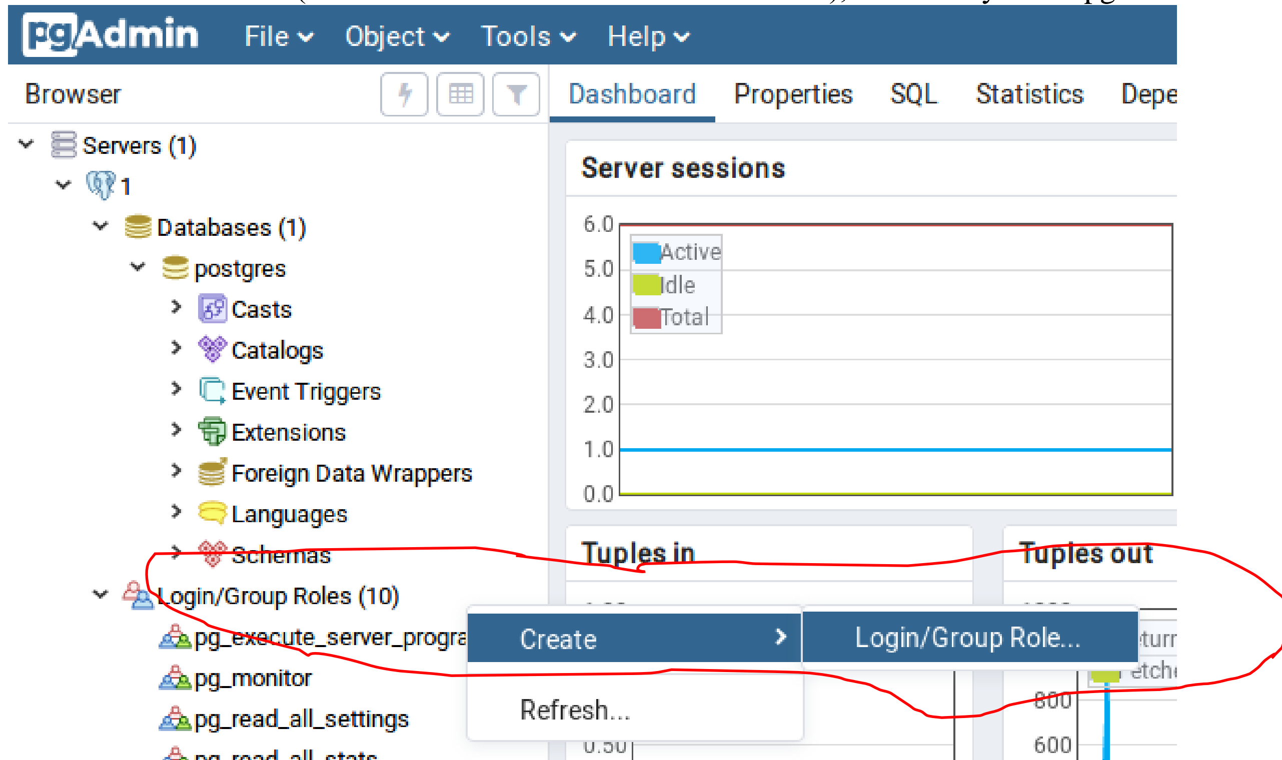
Enter new password:

13. Запустите из папки с PostgreSQL pgAdmin4 или терминала и убедитесь, что он запускается и вы в него заходите. Сделайте для отчёта скриншот. При правильной работе вы должны увидеть такое же окно, как при установке в Windows.

2.7 Практическая работа № 7
«Работа с программой psql — интерактивным терминалом PostgreSQL»

Задание:
Работаем в Linux

1. В терминале запустить psql.
Если выдаёт замечание, что роли не существует, это значит, что интерактивный терминал psql запущен, но нужно создать роль.
Либо createuser role (вместо role имя вашего пользователя), либо запустить pgadmin4 →



→ создать пользователя с привилегиями суперпользователя (можно установить на всех пунктах «да»).

Скриншот с окном привилегий и роли в отчёт.

2. Открыть новый терминал.
Набрать в терминале psql. (Комментарий на отсутствие роли не должен больше появляться)
С помощью createuser --help выяснить, а затем создать пользователя root с правами подключения к серверу по умолчанию и пользователя с именем test без подключения к серверу.
Скриншот с командами в отчёт.
Убедиться, что в pgadmin (refresh — обновление)появились созданные вами пользователи и просмотреть в свойствах их привилегии.

3. С помощью команды dropuser удалить пользователя test. Информацию как воспользоваться данной командой узнать с помощью dropuser --help или с помощью документации PostgreSQL.
Скриншот с командой в терминале и информацией из pgAdmin об удалении test в отчёт.

4. С помощью команды created –O user test создать новую базу данных, где user ваш пользователь с правами суперпользователя, test — название создаваемой базы данных, параметр –O указывает пользователя в качестве владельца создаваемой базы.
Создайте ещё одну базу данных под именем test1.
Скриншот с командой и скриншот с информацией о появлении бд test и test1 в pgAdmin в отчёт.

5. С помощью команды dropdb удалить базу данных test1. Информацию как воспользоваться данной командой узнать с помощью dropdb --help или с помощью документации PostgreSQL.
Скриншот с командой в терминале и информацией из pgAdmin об удалении test1 в отчёт.

6. С помощью команды psql –l (эл) выведите на экран список баз данных. Обратите внимание какие базы данных принадлежат каким пользователям.
Скриншот в отчёт со списком баз данных.
С помощью ctrl+Z можно выйти из режима просмотра.

7. С помощью команды pg_dumpall выполнить бэкап всех баз данных.
В отчёт подробно (как вы поняли текст бэкапа) расписать, что сделала данная команда.

8. С помощью команды pip install pgcli установить утилиту с авто-дополнениями и подсветкой синтаксиса. В процессе установки попросят установить pip — устанавливаем.
С помощью команды pgcli –U user –W test запускаем данную утилиту. User — ваш пользователь с правами суперпользователя. В процессе могут попросить ещё раз установку — соглашаемся.
Войдя в утилиту и увидите test: — программа ждёт ваших команд. Попробуйте создать таблицу в любым именем (create table …). Как данная утилита подсказывает ошибки? — ваш ответ и скриншот с вашими вариантами в отчёт.
С помощью команды exit выйти из данной утилиты.

2.8 Практическая работа № 8
«Основные операции с таблицами: добавление строк, упорядочивание по атрибутам. Группировка данных»

Задание 1:
Работаем в Linux

1. Запустить терминал. НЕ ЗАПУСКАТЬ pgadmin4.

2. С помощью команды wget (команда скачивает из интернета файлы) и ссылки https://edu.postgrespro.ru/demo_small.zip скачать учебную базу данных «Авиаперевозки». Куда выполняется скачивание? — ответить в отчёте.

3. С помощью команды unzip demo_small.zip распаковать учебную базу. Куда распаковался архив? — ответить в отчёт.

4. С помощью команды sudo su postgres перейдите на использование пользователя postgres, так как вся работа с PostgreSQL осуществляется под пользователем postgres.

5. С помощью команды psql -f demo_small.sql -U postgres создать базу данных в вашем кластере PostgreSQL. (должна появится информация: postgres=$)
С помощью файла «Шпаргалка по основным командам…» объяснить в отчёте команду и её параметры. (объяснить что значит параметр –f, параметр -U)

6. С помощью команды psql -d demo -U postgres подключиться к вашей базе данных. Сделать скриншот с командами. (должна появится информация: demo=$)
С помощью файла «Шпаргалка по основным командам…» объяснить в отчёте команду и её параметры.
Если вы всё делаете правильно, но остаётесь в режиме postgres=$, то наберите команду \connect demo ((должна появится информация: demo=$).
7. Запустить pgadmin4. Убедиться, что там появилась бд demo, убедиться, что появились таблицы в ней из demo_small.sql. Сделать скриншот в отчёт.

Теория:
Для создания таблиц в языке SQL служит команда CREATE TABLE. Ее полный синтаксис представлен в документации на PostgreSQL, а упрощенный синтаксис таков:

CREATE TABLE "имя_таблицы"

(имя_поля тип_данных [ограничения_целостности],

имя_поля тип_данных [ограничения_целостности],

имя_поля тип_данных [ограничения_целостности],

[ограничение_целостности],
[первичный_ключ],
[внешний_ключ]);

В квадратных скобках показаны необязательные элементы команды. После команды нужно поставить символ«;».

Описание атрибута	Имя атрибута	Тип данных	Тип PostgreSQL	Ограничения
Код самолета, IATA	aircraft_code	Символьный	char(3)	NOT NULL
Модель самолета	model	Символьный	text	NOT NULL
Максимальная дальность полета, км	range	Числовой	integer	NOT NULL range > 0

8. Команда для создания первой таблицы «Самолеты» такова:
CREATE TABLE aircrafts (aircraft_code char(3) NOT NULL, model text NOT NULL, range integer NOT NULL, CHECK (range > 0), PRIMARY KEY (aircraft_code));
9. Для просмотра используется команда \d aircrafts. Скриншот в отчет. (Shift+ZZ для выхода из режима просмотра)
10. Запустите pgadmin4, найдите эту таблицу там, сделайте скриншот.
Теория:
Теперь можно приступить к вводу данных в таблицу «Самолеты». Для выполнения этой операции служит команда INSERT. Ее упрощенный формат таков:
INSERT INTO имя_таблицы [(имя_атрибута, имя_атрибута, ...)] VALUES (значение_атрибута, значение_атрибута, ...);
11. Ввести одну строку с помощью команды
INSERT INTO aircrafts (aircraft_code, model, range) VALUES ('SU9', 'Sukhoi SuperJet-100', 3000);
12. Для просмотра используется выборка SELECT:
SELECT * FROM aircrafts;
Скриншот в отчёт.
13. Теперь добавить 8 строк:
INSERT INTO aircrafts (aircraft_code, model, range) VALUES ('773', 'Boeing 777-300', 11100), ('763', 'Boeing 767-300', 7900), ('733', 'Boeing 737-300', 4200), ('320', 'Airbus A320-200', 5700), ('321', 'Airbus A321-200', 5600), ('319', 'Airbus A319-100', 6700), ('CN1', 'Cessna 208 Caravan', 1200), ('CR2', 'Bombardier CRJ-200', 2700);
14. Просмотреть и сделать скриншот в отчёт.
15. С помощью команды exit выйти из базы данных demo.
16. С помощью su войти под вашим пользователем.
17. С помощью shutdown now выключить Ubuntu.

Задание 2:

1. Запустить терминал.
2. Перейти на пользователя postgres (команда прописана в предыдущей практической работе);
3. Запустить psql (команда прописана в предыдущей практической работе);
4. Подключиться к базе данных demo (команда прописана в предыдущей практической работе);
5. Вывести на экран содержимое таблицы Самолёты, чтобы убедиться что всё правильно работает.
Сделать скриншот всех команд выше.

6. Для упорядочивания строк по значению атрибута model и порядка расположения столбцов воспользуйтесь командой:
SELECT model, aircraft_code, range

FROM aircrafts

ORDER BY model;
В отчёт скриншот и ответ на вопрос: Что изменилось?

7. Выбираемые строки из таблицы можно изменить и ограничить выбор с помощью WHERE в команде SELECT:
SELECT model, aircraft_code, range
FROM aircrafts
WHERE range >= 4000 AND range <= 6000;
Данная команда выбирает модели самолётов, у которых максимальная дальность полёта находится в пределах от 4 до 6 тысяч км включительно.
8. Напишите команду для отображения на экране моделей самолёта с максимальной дальностью полёта больше 1000, но меньше 3000 км.
9. Напишите команду для отображения на экране только самолёта Boeing 737-300 (так как это текст, то используются **одинарные** кавычки).
В отчёт скриншот с п.7, п.8, п.9.
10. С помощью команды
UPDATE aircrafts SET range = 3500
WHERE aircraft_code = 'SU9';
обновите данные для самолёта Sukhoi SuperJet, теперь его дальность полёта в таблице изменилась с 3000 км до 3500 км.
11. Выведите информацию на экран только об изменении этой строки в таблице.
В отчёт скриншот с п.10 и п.11
12. С помощью команды DELETE можно удалять строки из таблицы либо все, либо по условию: DELETE FROM имя_таблицы WHERE условие;
Удалите строку с кодом CN1 с помощью команды:
DELETE FROM aircrafts WHERE aircraft_code = 'CN1';
При успешном удалении строки СУБД выводит информацию DELETE 1.
На экран выведите информацию со всеми оставшимися строками.
13. С помощью команды удалите информацию о самолетах с дальностью полета более 10 000 км, а также с дальностью полета менее 3000км:
DELETE FROM aircrafts WHERE range > 10000 OR range < 3000;
На экран выведите информацию со всеми оставшимися строками.
14. С помощью команды DELETE FROM aircrafts; удалите все строки в таблице.
На экран выведите информацию со строками таблицы.
В отчёт скриншот с п.12, п.13, п.14.
15. С помощью клавиши «стрелка вверх» на клавиатуре найдите две команды с командой INSERT для добавления восьми строк и для добавления одной строки в таблицу Самолёты. Выполните эти команды.
Выведите информацию о строках таблицы на экран (должно быть снова 9 строк.)
В отчёт скриншот с командами п.15.
16. Наберите команду /s — просмотр всех команд, которые вы набирали. (Shift+ZZ для корректного выхода из режима просмотра).
17. Для создания второй таблицы используйте следующую команду:
CREATE TABLE seats
(aircraft_code char(3) NOT NULL,
seat_no varchar(4) NOT NULL,
fare_conditions varchar(10) NOT NULL,
CHECK (fare_conditions IN ('Economy', 'Comfort', 'Business')),
PRIMARY KEY (aircraft_code, seat_no),

FOREIGN KEY (aircraft_code)
REFERENCES aircrafts (aircraft_code)
ON DELETE CASCADE);
Скриншот с п.17 в отчёт.

18. Открыть учебное пособие на стр.31-33 и к команде в п.17 дать подробное описание к каждой строчке.

19. С помощью команды \d seats просмотреть данные таблицы. (Shift+ZZ для корректного выхода из режима просмотра).

20. С помощью команды \d просмотрите список ваших таблиц.

21. Наберите следующую команду по добавлению данных в таблицу «Места»:
INSERT INTO seats VALUES ('123', '1A', 'Business');
СУБД вам выдаст ошибку. С помощью учебного пособия на стр.34 в отчёт написать о причине ошибки.

22. С помощью команды:
INSERT INTO seats VALUES
('SU9', '1A', 'Business'),
('SU9', '1B', 'Business'),
('SU9', '10A', 'Economy'),
('SU9', '10B', 'Economy'),
('SU9', '10F', 'Economy'),
('SU9', '20F', 'Economy');
внесите записи в таблицу «Места».
Выведите на экран содержимое таблицы Места.

23. Для всех моделей самолётов внести те же места, что для SU9. Для этого необходимо для каждого самолёта повторить команду INSERT INTO seats VALUES, изменив ‘SU9’ на номер другого самолёта. В итоге должно получится 54 строки.

24. На экран вывести все данные по местам самолётов. В отчёт скриншот.

Задание 3:

1. Запустить терминал, перейти на пользователя postgres, подключиться к demo.

2. Проверить наличие таблиц и их данных (вывести на экран данные одной таблицы и данные второй таблицы).

3. Для подсчёта строк используют функцию count. Предположим, что необходимо подсчитать количество мест в определённом самолёте, для этого нужно использовать следующую команду:
SELECT count(*) FROM seats WHERE aircraft_code = 'SU9';

4. Для подсчёта мест во всех самолётах сразу используют следующую команду:
SELECT aircraft_code, count(*) FROM seats
GROUP BY aircraft_code;
где происходит операция группировки данных по атрибуту «Код самолёта» из строк таблицы «Места». Group by группирует по атрибуту.
В отчёт скриншот с командами выше.

5. Для сортировки данной выборки по числу мест в самолётах используют команду:
SELECT aircraft_code, count(*) FROM seats
GROUP BY aircraft_code
ORDER BY count;

6. Чтобы была наглядна представлена сортировка по числу мест в самолётах, добавьте всем самолётам дополнительные места.

7. Повторить команду с пункта 5 и сделать скриншот результата.

8. Чтобы выполнить группировку по двум атрибутам нужно применить следующую команду:
SELECT aircraft_code, fare_conditions, count(*) FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
В отчёт вставить скриншот с результатом и ответом на вопрос: какие два атрибута были использованы для группировки?

9. Попробуйте ввести в таблицу aircrafts строку с таким значением атрибута «Код самолета»(aircraft_code),которое вы уже вводили:
INSERT INTO aircrafts VALUES ('SU9', 'Sukhoi SuperJet-100', 3000);
Ответьте в отчёте: что у вас получилось?

10. Самостоятельно напишите команду для выборки всех строк из таблицы aircrafts, чтобы строки были упорядочены по убыванию (для этого нужно после имени атрибута в предложении ORDER BY добавить ключевое слово DESC) значения атрибута «Максимальная дальность полета, км» (range).
Скриншот с командой и результатом в отчёт.

11. Команда UPDATE позволяет в процессе обновления выполнять арифметические действия над значениями, находящимися в строках таблицы. Представим себе, что двигатели самолета Sukhoi SuperJet стали в два раза экономичнее, вследствие чего дальность полета этого лайнера возросла ровно в два раза. Команда UPDATE позволяет увеличить значение атрибута range в строке, хранящей информацию об этом самолете, даже не выполняя предварительно выборку с целью выяснения текущего значения этого атрибута. При присваивании нового значения атрибуту range можно справа от знака «=» написать не только числовую константу, но и целое выражение. В нашем случае оно будет простым: range = range * 2. Самостоятельно напишите команду UPDATE полностью, при этом не забудьте, что увеличить дальность полета нужно только у одной модели —Sukhoi SuperJet-100, поэтому необходимо использовать условие WHERE. Затем с помощью команды SELECT проверьте полученный результат.
Скриншот с командой и результатом в отчёт.

12. Если в предложении WHERE команды DELETE вы укажете логически и синтаксически корректное условие, но строк, удовлетворяющих этому условия, в таблице не окажется, то в ответ СУБД выведет сообщение
DELETE 0
Такая ситуация не является ошибкой или сбоем в работе СУБД. Например, если после удаления какой-то строки вы повторно попытаетесь удалить ее же, то получите именно такое сообщение. Самостоятельно смоделируйте описанную ситуацию, подобрав условие, которому гарантированно не соответствует ни одна строка в таблице «Самолеты» (aircrafts).
Самостоятельно смоделируйте описанную ситуацию, подобрав условие, которому гарантированно не соответствует ни одна строка в таблице «Места»
Скриншот с командами и результатом в отчёт.

13. Выведите на экран данные по **всем атрибутам** из таблицы «Места».
Скриншот в отчёт.

14. Выведите на экран данные по **всем атрибутам** из таблицы «Самолёты».
Скриншот в отчёт.

15. Выйдите из demo (/q), смените пользователя(su user) на вашего пользователя, выключите виртуальную машину (shutdown now).

Задание 4:

1. Предположим, что нам необходимо сформировать и сохранить в базе данных в удобной форме графики работы пилотов авиакомпании, т. е. номера дней недели, когда они совершают полеты. Создадим таблицу, в которой эти графики будут храниться в виде единых списков, т. е. в виде одномерных массивов.

```
CREATE TABLE pilots
( pilot_name text,
schedule integer[]);
```

2. Далее добавим в таблицу четыре строки. Массив в команде вставки представлен в виде строкового литерала с указанием типа данных и квадратных скобок, означающих массив. Обратите внимание, что все массивы имеют различное число элементов.

```
INSERT INTO pilots
VALUES ( 'Ivan', '{1, 3, 5, 6, 7 }'::integer[] ),
( 'Petr', '{1, 2, 5, 7 }'::integer[] ),
( 'Pavel', '{2, 5}'::integer[] ),
( 'Boris', '{3, 5, 6 }'::integer[] );
```

Для просмотра команда: SELECT * FROM pilots;

3. Предположим, что руководство компании решило, что каждый пилот должен летать 4 раза в неделю. Значит, нам придется обновить значения в таблице. Пилоту по имени Boris добавим один день с помощью операции конкатенации:

```
UPDATE pilots
SET schedule = schedule || 7
WHERE pilot_name = 'Boris';
```

Просмотрите, что получилось.
Скриншот вставьте в отчёт.

4. Пилоту по имени Pavel добавим один день в конец списка (массива) с помощью функции array_append:

```
UPDATE pilots
SET schedule = array_append( schedule, 6 )
WHERE pilot_name = 'Pavel';
```

Просмотрите, что получилось.

5. Пилоту по имени Павел добавим один день в начало списка с помощью функции array_prepend (обратите внимание, что параметры функции поменялись местами):

```
UPDATE pilots
SET schedule = array_prepend( 1, schedule )
WHERE pilot_name = 'Pavel';
```

Просмотрите, что получилось.
Скриншот вставьте в отчёт.

6. У пилота по имени Ivan имеется лишний день в графике. С помощью функции array_remove удалим из графика пятницу (второй параметр функции указывает значение элемента массива, а не индекс):

```
UPDATE pilots
SET schedule = array_remove( schedule, 5 )
WHERE pilot_name = 'Ivan';
```

Просмотрите, что получилось.

7. У пилота по имени Petr изменим дни полетов, не изменяя их общего количества. Воспользуемся индексами для работы на уровне отдельных элементов массива. По умолчанию нумерация индексов начинается с единицы, а не с нуля. При необходимости ее можно изменить. К элементам одного и того же массива можно обращаться в предложении SET по отдельности, как будто это разные столбцы.

UPDATE pilots
SET schedule[1] = 2, schedule[2] = 3
WHERE pilot_name = 'Petr';
Просмотрите, что получилось.

8. Можно тоже самое сделать, используя срез (slice) массива:
UPDATE pilots
SET schedule[1:2] = ARRAY[2, 3]
WHERE pilot_name = 'Petr';
Просмотрите, что получилось.
Скриншот вставьте в отчёт.
В вышеприведенной команде запись 1:2 означает индексы первого и последнего элементов диапазона массива. Нотация с использованием ключевого слова ARRAY — это альтернативный способ создания массива (он соответствует стандарту SQL). Таким образом, присваивание новых значений производится сразу целому диапазону элементов массива.

9. Теперь продемонстрируем основные операции, которые можно применять к массивам, выполняя выборки из таблиц.
Получим список пилотов, которые летают каждую среду:
SELECT * FROM pilots
WHERE array_position(schedule, 3) IS NOT NULL;
Функция array_position возвращает индекс первого вхождения элемента с указанным значением в массив. Если же такого элемента нет, она возвратит NULL.

10. Выберем пилотов, летающих по понедельникам и воскресеньям:
SELECT *FROM pilots
WHERE schedule @> '{ 1, 7 }'::integer[];
Оператор @> означает проверку того факта, что в левом массиве содержатся все элементы правого массива. Конечно, при этом в левом массиве могут находиться и другие элементы, что мы и видим в графике этого пилота.
Скриншот вставьте в отчёт.

11. Далее выясним, кто летает по вторникам и/или по пятницам? Для получения ответа воспользуемся оператором &&, который проверяет наличие общих элементов у массивов, т. е. пересекаются ли их множества значений. В нашем примере число общих элементов, если они есть, может быть равно одному или двум.
SELECT *FROM pilots WHERE schedule && ARRAY[2, 5];

12. Теперь выясним кто не летает ни во вторник, ни в пятницу? Для получения ответа добавим в предыдущую SQL-команду отрицание NOT:
SELECT *FROM pilots
WHERE NOT (schedule && ARRAY[2, 5]);

13. Другой вариант представления с помощью функции unnest:
SELECT unnest(schedule) AS days_of_week FROM pilots
WHERE pilot_name = 'Ivan';
Скриншот вставьте в отчёт.

Тип данных JSON
Предположим, что руководство авиакомпании поддерживает стремление пилотов улучшать свое здоровье, повышать уровень культуры и расширять кругозор. Поэтому разработчики базы данных авиакомпании получили задание создать специальную таблицу, в которую будут заноситься сведения о тех видах спорта, которыми занимается пилот, будет отмечаться наличие у него домашней библиотеки, а также фиксироваться количество стран, которые он посетил в ходе туристических поездок.

14. Для этого создадим следующую таблицу со следующими данными:
```
CREATE TABLE pilot_hobbies
( pilot_name text,
hobbies jsonb);

INSERT INTO pilot_hobbies
VALUES ( 'Ivan',
'{ "sports": ["футбол", "плавание" ],
"home_lib": true, "trips": 3
}'::jsonb ),
( 'Petr','{ "sports": ["теннис", "плавание" ],
"home_lib": true, "trips": 2
}'::jsonb ),
( 'Pavel','{ "sports": ["плавание" ],
"home_lib": false, "trips": 4
}'::jsonb ),
( 'Boris',
'{ "sports": ["футбол", "плавание", "теннис" ],
"home_lib":true, "trips": 0
}'::jsonb );
```

Для просмотра данных таблицы: SELECT * FROM pilot_hobbies;
Вставьте скриншот в отчёт.
При выводе строк из таблицы порядок ключей в JSON-объектах не был сохранен.

15. Предположим, что нужно сформировать футбольную сборную команду нашей авиакомпании для участия в турнире.
Мы можем выбрать всех футболистов таким способом:
```
SELECT * FROM pilot_hobbies
WHERE hobbies @> '{ "sports": [ "футбол" ] }'::jsonb;
```
Или так:
```
SELECT pilot_name, hobbies->'sports' AS sports FROM pilot_hobbies
WHERE hobbies->'sports' @> '[ "футбол" ]'::jsonb;
```
В этом решении мы выводим только информацию о спортивных предпочтениях пилотов. Внимательно посмотрите, как используются одинарные и двойные кавычки. Операция «–>» служит для обращения к конкретному ключу JSON-объекта. При создании столбца с типом данных json или jsonb не требуется задавать структуру объектов, т. е. конкретные имена ключей. Поэтому в принципе возможна ситуация, когда в разных строках в JSON-объектах будут использоваться различные наборы ключей.

16. Ключа «sport» в наших объектах нет. Что покажет вызов функции count?
```
SELECT count( * ) FROM pilot_hobbies
WHERE hobbies ? 'sport';
```
Что у вас отобразилось на экране? Вставьте скриншот в отчёт.

17. Предположим, что пилот по имени Boris решил посвятить себя только хоккею. Тогда в базе данных мы выполним такую операцию:
```
UPDATE pilot_hobbies
SET hobbies = hobbies || '{ "sports": [ "хоккей" ] }'
WHERE pilot_name = 'Boris';
```
Проверим, что получилось:
```
SELECT pilot_name, hobbies
FROM pilot_hobbies
WHERE pilot_name = 'Boris';
```

18. Если впоследствии Boris захочет возобновить занятия футболом, то с помощью функции jsonb_set можно будет обновить сведения о нем в таблице:

```
UPDATE pilot_hobbies
SET hobbies = jsonb_set( hobbies, '{ sports, 1 }', '"футбол"' )
WHERE pilot_name = 'Boris';
```

Второй параметр функции указывает путь в пределах JSON-объекта, куда нужно добавить новое значение. В данном случае этот путь состоит из имени ключа (sports) и номера добавляемого элемента в массиве видов спорта (номер 1). Нумерация элементов начинается с нуля. Третий параметр имеет тип jsonb, поэтому его литерал заключается в одинарные кавычки, а само добавляемое значение берется в двойные кавычки. В результате получается — '"футбол"'.

Проверим успешность выполнения этой операции:

```
SELECT pilot_name, hobbies FROM pilot_hobbies
WHERE pilot_name = 'Boris';
```

Вставьте скриншот в отчёт.

19. Предположим, что пилоты авиакомпании имеют возможность высказывать свои пожелания насчет конкретных блюд, из которых должен состоять их обед во время полета. Для учета пожеланий пилотов необходимо создать таблицу pilots1:

```
CREATE TABLE pilots1
( pilot_name text,
schedule integer[],
meal text[]);
```

Добавим строки в таблицу:

```
INSERT INTO pilots1
VALUES ( 'Ivan', '{ 1, 3, 5, 6, 7 }'::integer[],
'{ "сосиска", "макароны", "кофе" }'::text[] ),
( 'Petr', '{ 1, 2, 5, 7 }'::integer [],
'{ "котлета", "каша", "кофе" }'::text[] ),
( 'Pavel', '{ 2, 5 }'::integer[],
'{ "сосиска", "каша", "кофе" }'::text[] ),
( 'Boris', '{ 3, 5, 6 }'::integer[],
'{ "котлета", "каша", "чай" }'::text[] );
```

Обратите внимание, что каждое из текстовых значений, включаемых в литерал массива, заключается в двойные кавычки, а в качестве типа данных указывается text[].

Вот что получилось:

```
SELECT * FROM pilots1;
```

Давайте получим список пилотов, предпочитающих на обед сосиски:

```
SELECT * FROM pilots1 WHERE meal[ 1 ] = 'сосиска';
```

Вставьте скриншот в отчёт.

Предположим, что руководство авиакомпании решило, что пища пилотов должна быть разнообразной. Оно позволило им выбрать свой рацион на каждый из четырех дней недели, в которые пилоты совершают полеты. Для нас это решение руководства выливается в необходимость модифицировать таблицу, а именно: столбец meal теперь будет содержать двумерные массивы. Определение этого столбца станет таким: meal text[][]

Задание.

Создайте новую версию таблицы и соответственно измените команду INSERT, чтобы в ней содержались литералы двумерных массивов. Они будут выглядеть примерно так:

```
'{ {"сосиска","макароны", "кофе" },
{"котлета","каша", "кофе" },
{"сосиска","каша", "кофе" },
{"котлета","каша", "чай" } }'::text[][]
```

Сделайте ряд выборок и обновлений строк в этой таблице. Для обращения к элементам двухмерного массива нужно использовать два индекса. Не забывайте, что по умолчанию номера индексов начинаются с единицы.

20. Для обновления скалярных значений, например, по ключу trips, можно сделать так:

```
UPDATE pilot_hobbies
SET hobbies = jsonb_set( hobbies, '{ trips }', '10' )
WHERE pilot_name = 'Pavel';
```

Второй параметр функции — это путь в пределах JSON-объекта. Он теперь представляет собой лишь имя ключа. Однако его необходимо заключить в фигурные скобки. Третий параметр — это новое значение. Хотя оно числовое, но все равно требуется записать его в одинарных кавычках.

```
SELECT pilot_name, hobbies->'trips' AS trips FROM pilot_hobbies;
```

Задание.

Самостоятельно выполните изменение значения по ключу home_lib в одной из строк таблицы.

2.9 Практическая работа № 9 «Добавление ограничений»

Задание:

1. Создайте базу данных edu: **createdb -U postgres edu**

2. Подключитесь к ней: **psql -d edu -U postgres**

3. Создайте две таблицы с ограничениями:

```
CREATE TABLE students
( record_book numeric( 5 ) NOT NULL, name text NOT NULL, doc_ser numeric( 4 ), doc_num numeric( 6 ), PRIMARY KEY ( record_book )
);
CREATE TABLE progress
( record_book numeric( 5 ) NOT NULL, subject text NOT NULL,
acad_year text NOT NULL,
term numeric( 1 ) NOT NULL CHECK ( term = 1 OR term = 2 ),
mark numeric( 1 ) NOT NULL CHECK ( mark >= 3 AND mark <= 5 ) DEFAULT 5,
FOREIGN KEY ( record_book )
REFERENCES students ( record_book )
ON DELETE CASCADE
ON UPDATE CASCADE );
```

4. Внесите в каждую таблицу по три строки данных.

5. В отчёте объясните каждую строку кода и ответьте на вопрос: какие ограничения используются в данной базе данных при её создании?

6. На экран терминала вывести все данные из таблицы Студенты.

7. На экран терминала вывести все данные из таблицы Успеваемость. В отчёт вставить скриншот с данными из двух таблиц.

2.10 Практическая работа № 10
«Модификация таблиц для нормализации отношений»

Задание:

1. 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 сформируем такую команду для добавления столбца:

ALTER TABLE airports
ADD COLUMN speed integer NOT NULL CHECK(speed >= 300);
Убедитесь, что появляется ошибка, указанная в лекционной части.
В отчёт вставьте скриншот.

2. Выполните модификацию другим:
ALTER TABLE aircrafts ADD COLUMN speed integer;
UPDATE aircrafts SET speed = 807 WHERE aircraft_code = '733';
UPDATE aircrafts SET speed = 851 WHERE aircraft_code = '763';
UPDATE aircrafts SET speed = 905 WHERE aircraft_code = '773';
UPDATE aircrafts SET speed = 840 WHERE aircraft_code IN ('319', '320', '321');
UPDATE aircrafts SET speed = 786 WHERE aircraft_code = 'CR2';
UPDATE aircrafts SET speed = 341 WHERE aircraft_code = 'CN1';
UPDATE aircrafts SET speed = 830 WHERE aircraft_code = 'SU9';

Далее выведем полученный результат с помощью команды:
SELECT * FROM aircrafts;
Внесём ограничения:
ALTER TABLE aircrafts ALTER COLUMN speed SET NOT NULL;
ALTER TABLE aircrafts ADD CHECK(speed >= 300);
Проверьте, как изменилось определение таблицы, с помощью команды
\d aircrafts
В отчёт скриншот с результатами.

3. Удаляем ограничение:
ALTER TABLE aircrafts ALTER COLUMN speed DROP NOT NULL;
ALTER TABLE aircrafts DROP CONSTRAINT aircrafts_speed_check;

4. Удаляем новый столбец:
ALTER TABLE aircrafts DROP COLUMN speed;
В отчёт скриншот с результатами.

5. С помощью следующей команды создать ещё одну таблицу в нашей учебной базе данных:
CREATE TABLE airports
(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 аэро-
порта
PRIMARY KEY (airport_code)
);

Изменим тип данных для атрибутов «Координаты аэропорта: долгота» (longitude) и «Координаты аэропорта: широта» (latitude) с float (double precision) на numeric(5, 2). Сделать это можно с помощью одной команды, поскольку команда ALTER TABLE поддерживает и выполнение более одного действия за один раз.
Сначала посмотрим, с какой точностью выводятся значения этих атрибутов до изменения типа данных, затем изменим тип данных для двух столбцов, опять выведем содержимое таблицы на экран и убедимся, что значения были округлены в соответствии с правилами округления.

```
SELECT * FROM airports;
ALTER TABLE airports
ALTER COLUMN longitude SET DATA TYPE numeric( 5,2 ),
ALTER COLUMN latitude SET DATA TYPE numeric( 5,2 );
SELECT * FROM airports;
```

6. Предположим, что по результатам опытной эксплуатации базы данных «Авиаперевозки» мы пришли к выводу о том, что необходимо создать таблицу, содержащую коды и наименования классов обслуживания. Назовем ее «Классы обслуживания» (fare_conditions). В ее состав включим два столбца: «Код класса обслуживания» и «Наименование класса обслуживания». Имена столбцам присвоим с учетом принципов формирования имен аналогичных столбцов в других таблицах, например, в таблице «Аэропорты» (airports).

```
CREATE TABLE fare_conditions
( fare_conditions_code integer, fare_conditions_name varchar( 10 ) NOT NULL,
PRIMARY KEY ( fare_conditions_code ));
```

Добавим в новую таблицу необходимые данные:

```
INSERT INTO fare_conditions
VALUES ( 1, 'Economy' ),
( 2, 'Business' ),
( 3, 'Comfort' );
```

В отчёт скриншот с результатами.

7. Поскольку мы ввели в обращение числовые коды для классов обслуживания, то необходимо модифицировать определение таблицы «Места» (seats), а именно: тип данных столбца «Класс обслуживания» (fare_conditions) изменить с varchar(10) на integer. Для реализации такой задачи служит фраза USING команды ALTER TABLE. Однако такой вариант команды не сработает:

```
ALTER TABLE seats
ALTER COLUMN fare_conditions SET DATA TYPE integer
USING ( CASE WHEN fare_conditions = 'Economy' THEN 1
WHEN fare_conditions = 'Business' THEN 2 ELSE 3
END );
```

Для замены исходных значений на новые мы используем конструкцию CASE WHEN ... THEN ... ELSE ... END.
Выполнить операцию не удастся, СУБД выдаст сообщение об ошибке: ОШИБКА: ограничение-проверку "seats_fare_conditions_check" нарушает… некоторая строка
В определении таблицы есть ограничение CHECK, которое требует, чтобы значение столбца fare_conditions выбиралось из списка: «Economy», «Comfort», «Business». При замене символьных значений на числовые это ограничение будет заведомо нарушаться.

Следовательно, необходимо в команду ALTER TABLE добавить операцию удаления этого ограничения.
Пробуем новый вариант команды:

```
ALTER TABLE seats
DROP CONSTRAINT seats_fare_conditions_check,
ALTER COLUMN fare_conditions SET DATA TYPE integer
USING ( CASE WHEN fare_conditions = 'Economy' THEN 1
WHEN fare_conditions = 'Business' THEN 2
ELSE 3 END
);
```

8. Проверим результат работы с помощью команды SELECT * FROM seats;

```
aircraft_code | seat_no | fare_conditions
---------------+---------+-----------------
319 | 2A | 2
319 | 2C | 2
319 | 2D | 2
```

В отчёт скриншот с результатами.

9. Далее необходимо связать таблицы «Места» (seats) и «Классы обслуживания» (fare_conditions) по внешнему ключу:

```
ALTER TABLE seats
ADD FOREIGN KEY ( fare_conditions )
REFERENCES fare_conditions ( fare_conditions_code );
```

Посмотрев описание таблицы «Места» (seats), увидим, что внешний ключ успешно создан.

```
\d seats
"seats_fare_conditions_fkey" FOREIGN KEY (fare_conditions)
REFERENCES fare_conditions(fare_conditions_code)
```

10. Атрибуты внешнего ключа не обязательно должны ссылаться только на одноименные атрибуты ссылочной таблицы. Сейчас мы на практике успешно проверили это утверждение. Однако для удобства сопровождения базы данных имеет смысл переименовать столбец fare_conditions в таблице «Места» (seats), т. е. дать ему имя fare_conditions_code, поскольку в этой таблице хранится именно код класса обслуживания. Давайте так и поступим:

```
ALTER TABLE seats
RENAME COLUMN fare_conditions TO fare_conditions_code;
```

Если теперь посмотреть описание таблицы, то можно увидеть, что имя атрибута, являющегося внешним ключом, изменилось, а вот имя ограничения seats_fare_conditions_fkey осталось неизменным, хотя оно и было первоначально сформировано самой СУБД. Это шаблонное имя ограничения составляется из имени таблицы и имени первого (и единственного в данном случае) атрибута внешнего ключа.

```
"seats_fare_conditions_fkey" FOREIGN KEY (fare_conditions_code)
,→ REFERENCES fare_conditions(fare_conditions_code)
```

В отчёт скриншот с результатами.

11. Давайте переименуем это ограничение, чтобы поддержать соблюдение правила именования ограничений:

```
ALTER TABLE seats
RENAME CONSTRAINT seats_fare_conditions_fkey TO seats_fare_conditions_code_fkey;
```

Проверим, что получилось:

```
\d seats
```

12. Вернемся к таблице «Классы обслуживания» (fare_conditions). Мы предусмотрели в ней первичный ключ, но ведь значения атрибута «Наименование класса обслуживания»

(fare_conditions_name) также должны быть уникальными, дублирование значений не допускается. Давайте добавим ограничение уникальности по этому столбцу:

```
ALTER TABLE fare_conditions ADD UNIQUE ( fare_conditions_name );
\d fare_conditions
```

В отчёт скриншот с результатами.

2.11 Практическая работа № 11 «Создание представлений»

Задание:

1. Создадим простое представление. В одной из работ подсчитывали количество мест в салонах для всех моделей самолетов с учетом класса обслуживания (бизнес-класс и экономический класс). На его основе 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;
```

Скриншот вставьте в отчёт.

2. Далее создадим изменённое представление:

```
CREATE OR REPLACE VIEW seats_by_fare_cond AS
SELECT a.model, s.aircraft_code, s.fare_conditions, count( * ) AS num_seats
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
```

Однако СУБД выдаст сообщение об ошибке:

Удалить представление с помощью команды:

```
DROP VIEW seats_by_fare_cond;
```

Попробовать создать иначе:

```
CREATE OR REPLACE VIEW seats_by_fare_cond AS
SELECT a.model, s.aircraft_code, s.fare_conditions, count( * ) AS
num_seats
FROM seats
GROUP BY aircraft_code, fare_conditions
ORDER BY aircraft_code, fare_conditions;
```

Если снова будет ошибка, то удалить **DROP VIEW seats_by_fare_cond;**

Создать:

```
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;
```

Скриншот вставьте в отчёт.

3. Для создания следующих представлений понадобится новая таблица Рейсы.

Команда для создания новой таблицы Рейсы:

```
CREATE TABLE flights
( flight_id serial NOT NULL,
flight_no char( 6 ) NOT NULL,
scheduled_departure timestamptz NOT NULL,
scheduled_arrival timestamptz NOT NULL,
departure_airport char( 3 ) NOT NULL,
arrival_airport char( 3 ) NOT NULL,
status varchar( 20 ) NOT NULL,
aircraft_code char( 3 ) NOT NULL,
actual_departure timestamptz,
actual_arrival timestamptz,
CHECK ( scheduled_arrival > scheduled_departure ),
CHECK ( actual_arrival IS NULL OR ( actual_departure IS NOT NULL AND actual_arrival IS NOT NULL AND actual_arrival > actual_departure )),
CHECK ( status IN ( 'On Time', 'Delayed', 'Departed', 'Arrived', 'Scheduled', 'Cancelled' )),
PRIMARY KEY ( flight_id ),
UNIQUE ( flight_no, scheduled_departure ),
FOREIGN KEY ( aircraft_code )
REFERENCES aircrafts ( aircraft_code ),
FOREIGN KEY ( arrival_airport )
REFERENCES airports ( airport_code ),
FOREIGN KEY ( departure_airport )
REFERENCES airports ( airport_code ));
```

Описание столбцов:

Описание атрибута	**Имя атрибута**	**Тип PostgreSQL**
Идентификатор рейса	flight_id	integer
Номер рейса	flight_no	char(6)
Время вылета по расписанию	scheduled_departure	timestamptz
Время вылета по расписанию, местное время в пункте отправления	scheduled_departure_local	timestamp
Время прилета по расписанию	scheduled_arrival	timestamptz
Время прилета по расписанию, местное время в пункте прибытия	scheduled_arrival_local	timestamp
Планируемая продолжительность полета	scheduled_duration	interval
Код аэропорта отправления	departure_airport	char(3)
Название аэропорта отправления	departure_airport_name	text
Город отправления	departure_city	text
Код аэропорта прибытия	arrival_airport	char(3)
Название аэропорта прибы-	arrival_airport_name	text

тия		
Город прибытия	arrival_city	text
Статус рейса	Status	varchar(20)
Код самолета, IATA	aircraft_code	char(3)
Фактическое время вылета	actual_departure	timestamptz
Фактическое время вылета, местное время в пункте отправления	actual_departure_local	timestamp
Фактическое время прилета	actual_arrival	timestamptz
Фактическое время прилета, местное время в пункте прибытия	actual_arrival_local	timestamp
Фактическая продолжительность полета	actual_duration	interval

Далее вывести информацию о созданной таблице на экран.
В отчёт скриншот.
В отчёте написать комментарии (объяснения) к каждой строке создания таблицы. За дополнительной информацией по типам данным можно обратиться к документации Postgres в Интернете.

4. Модифицировать созданную таблицу так, чтобы добавить следующие столбцы:
— название аэропорта отправления;
— город отправления;
— название аэропорта прибытия;
— город прибытия.
В отчёт скриншот.

5. Создать выборку, выводящую на экран номер рейса, название аэропорта отправления и название аэропорта прибытия. Упорядочить выборку по аэропорту отправления.

6. Создать представление 1flight на основе выборки в предыдущем пункте.

7. Создать выборку, выводящую на экран город отправления, город прибытия, фактическое время вылета, фактическое время прилёта.

8. Создать представление time на основе выборки в предыдущем пункте.
В отчёт скриншот.

9. Добавить в таблицу flights шесть строк.

10. Вывести на экран содержимое таблицы.
В отчёт скриншот.

11. Вывести на экран представление 1flight

12. Вывести на экран представление time.
В отчёт скриншот.

2.12 Практическая работа № 12
«Работа с командой SELECT»

Задание:

1. Добавьте в таблицу Аэропорты 5 строк. (аэропорты следующих городов: Санкт-Петербург, Москва, Екатеринбург, Сыктывкар, Адлер)
2. Выполните выборку всех строк таблиц Самолёты и таблицы Аэропорты.
В отчёт скриншот всех команд.

3. Выполните выборку из таблицы Самолёты всех самолётов компании Airbus с помощью команды:
SELECT * FROM aircrafts WHERE model LIKE 'Airbus%';

4. Выполните выборку из таблицы Самолёты всех самолётов модели Boening.

5. Выполните выборку из таблицы Самолёты всех самолётов с range больше 6000.
В отчёт скриншот всех команд.

6. Выполните выборку из таблицы Аэропорты всех аэропортов без Москвы и Санкт-Петербурга.

7. Выполните выборку из таблицы Аэропорты всех аэропортов с названием в 5 символов.
В отчёт скриншот.

8. Выполните выборку из таблицы Аэропорты городов по известным первым буквам Р и А.

9. Выполните выборку из таблицы Аэропорты всех аэропортов по известному последнему символу о.
В отчёт скриншот всех команд.

10. Выполните выборку из таблицы Самолёты всех самолётов с дальностью полёта в диапазоне от 6000 до 12000 км.

11. При выборке данные можно создавать вычисляемые столбцы. С помощь следующей команды переведите км в мили:
SELECT model, range, range / 1.609 AS miles FROM aircrafts;

12. С помощью команды **SELECT model, range, round(range / 1.609, 2) AS miles FROM aircrafts;** округлите до двух десятичных знаков.

13. Переделайте команды из предыдущего пункта так, чтобы мили округлялись до целых.
В отчёт скриншот всех команд.

14. С помощью следующей команды выполните упорядочивание в таблице Самолёты по убыванию дальности полёта: **SELECT * FROM aircrafts ORDER BY range DESC;**

15. Сделайте выборку из таблицы Аэропорты только по Часовому поясу.

16. Чтобы вывести информацию о Часовых поясах без повторяющихся значений воспользуйтесь следующей командой: **SELECT DISTINCT timezone FROM airports ORDER BY 1;**
В отчёт скриншот всех команд.

17. С помощью команды exit выйти из demo

18. С помощью su «вашего пользователя» выйти из postgres-пользователя и с помощью shutdown now выключить Ubuntu.

2.13. Практическая работа № 13 «Создание запросов на минимальные и максимальные значения»

Задание:

1. Подключиться к базе данных demo.
В таблице «Самолеты» (aircrafts) есть столбец «Максимальная дальность полета» (range). Мы можем дополнить вывод данных из этой таблицы столбцом «Класс самолета», имея в виду принадлежность каждого самолета к классу дальнемагистральных, среднемаги-

стральных или ближнемагистральных судов.
Команда для данных действий:

```
SELECT model, range,
CASE WHEN range < 2000 THEN 'Ближнемагистральный'
WHEN range < 5000 THEN 'Среднемагистральный'
ELSE 'Дальнемагистральный'
END AS type
FROM aircrafts
ORDER BY model;
```

В отчёт скриншот с результатом и с текстом пояснения конструкции команды.

2. В тех случаях, когда информации, содержащейся в одной таблице, недостаточно для получения требуемого результата, используют соединение (join) таблиц. С помощью команды:

```
SELECT a.aircraft_code, a.model, s.seat_no, s.fare_conditions
FROM seats AS s
JOIN aircrafts AS a
ON s.aircraft_code = a.aircraft_code
WHERE a.model ~ '^Cessna' ORDER BY s.seat_no;
```

объединим атрибуты двух таблиц Самолёты и Места. Обратите внимание на имена атрибутов — в выборке использованы псевдонимы атрибутов ,а объявлены они с помощью ключевого слова AS.
В отчёте написать комментарии к каждой строчке команды.

3. Напишите такие же запросы по местам для следующих самолётов:
Airbus A320-200
Bombardier CRJ-200
В отчёт скриншот с результатом.

4. В соединении одна и та же таблица может участвовать дважды, т. е. формировать соединение таблицы с самой собой. В качестве примера рассмотрим запрос для создания представления «Рейсы» (flights_v):

```
CREATE OR REPLACE VIEW flights_v AS
SELECT f.flight_id, f.flight_no,
f.scheduled_departure,
timezone( dep.timezone, f.scheduled_departure )
AS scheduled_departure_local,
f.scheduled_arrival,
timezone( arr.timezone, f.scheduled_arrival )
AS scheduled_arrival_local,
f.scheduled_arrival - f.scheduled_departure
AS scheduled_duration,
f.departure_airport, dep.airport_name AS departure_airport_name, dep.city AS departure_city, f.arrival_airport, arr.airport_name AS arrival_airport_name, arr.city AS arrival_city, f.status,
f.aircraft_code,
f.actual_departure,
timezone( dep.timezone, f.actual_departure )
AS actual_departure_local,
f.actual_arrival,
timezone( arr.timezone, f.actual_arrival )
AS actual_arrival_local,
f.actual_arrival - f.actual_departure AS actual_duration
FROM flights f, airports dep, airports arr
WHERE f.departure_airport = dep.airport_code AND f.arrival_airport = arr.airport_code;
```

В этом представлении используется не только таблица «Рейсы» (flights), но также и таблица «Аэропорты» (airports).
Выполнить данную команду
Вывести на экран содержимое данного представления.
В отчёт скриншот и пояснения по результату (что выводит данное представление?)

5. С помощью следующей команды выполняется запрос с подсчётом строк в соединённых двух одинаковых таблиц:
SELECT count(*)
FROM airports a1, airports a2
WHERE a1.city <> a2.city;
СУБД соединяет каждую строку первой таблицы с каждой строкой второй таблицы, т. е. формирует декартово произведение таблиц — все попарные комбинации строк из двух таблиц. Затем СУБД отбрасывает те комбинированные строки, которые не удовлетворяют условию, приведенному в предложении WHERE. В нашем примере условие как раз и отражает требование о том, что рейсов из одного города в тот же самый город быть не должно.
Выполните команду.

6. Во втором варианте запроса мы используем соединение таблиц на основе неравенства значений атрибутов. Тем самым мы перенесли условие отбора результирующих строк из предложения WHERE в предложение FROM.
SELECT count(*)
FROM airports a1
JOIN airports a2 ON a1.city <> a2.city;
Выполните команду.

7. Третий вариант предусматривает явное использование декартова произведения таблиц. Для этого служит предложение CROSS JOIN. Лишние строки, как и в первом варианте, отсеиваем с помощью предложения WHERE:
SELECT count(*)
FROM airports a1 CROSS JOIN airports a2 WHERE a1.city <> a2.city;
Выполните команду и вставьте скриншот с результатами.

8. Создайте такие же три запроса с таблицей Рейсы.
Скриншот с результатами в отчёт.
9. Создайте дополнительную таблицу Бронирования:
CREATE TABLE bookings
(book_ref char(6) NOT NULL, -- номер бронирования
book_date timestamptz NOT NULL, -- дата бронирования
total_amount numeric(10, 2) NOT NULL, -- полная стоимость бронирования (возможные значения в 10 цифр с плавающей точкой и двумя цифрами после запятой)
PRIMARY KEY (book_ref));-- первичный ключ – номер бронирования
И таблицу Билеты:
CREATE TABLE tickets
(ticket_no char(13) NOT NULL, -- номер билета
book_ref char(6) NOT NULL, -- номер бронирования (должен соответствовать данным из таблицы Бронирования)
passenger_id varchar(20) NOT NULL, -- идентификатор пассажира
passenger_name text NOT NULL, -- имя пассажира
contact_data jsonb, -- контактные данные пассажира
PRIMARY KEY (ticket_no), -- первичный ключ – номер билета
FOREIGN KEY (book_ref) -- внешний ключ номер бронирования
REFERENCES bookings (book_ref)); -- ссылочный ключ на таблицу Бронирования по номеру бронирования.
И таблицу Перелёты:
CREATE TABLE ticket_flights

(ticket_no char(13) NOT NULL, -- Номер билета
flight_id integer NOT NULL, -- Идентификатор рейса
fare_conditions varchar(10) NOT NULL, -- Класс обслуживания
amount numeric(10, 2) NOT NULL, -- Стоимость перелета
CHECK (amount >= 0), -- ограничение -- стоимость перелёта больше или равна 0
CHECK (fare_conditions IN ('Economy', 'Comfort', 'Business')), -- ограничение – класс обслуживания экономный, комфорт, бизнес
PRIMARY KEY (ticket_no, flight_id), -- первичные ключи номер билета и номер рейса
FOREIGN KEY (flight_id) -- внешний ключ номер рейса
REFERENCES flights (flight_id), -- ссылочный ключ на таблицу Рейсы по номеру рейса
FOREIGN KEY (ticket_no) -- внешний ключ номер билета
REFERENCES tickets (ticket_no) -- ссылочный ключ на таблицу Билеты по номеру билета
);

И таблицу Посадочные талоны:

CREATE TABLE boarding_passes
(ticket_no char(13) NOT NULL, -- Номер билета
flight_id integer NOT NULL, -- Идентификатор рейса
boarding_no integer NOT NULL, -- Номер посадочного талона
seat_no varchar(4) NOT NULL, -- Номер места
PRIMARY KEY (ticket_no, flight_id),
UNIQUE (flight_id, boarding_no),
UNIQUE (flight_id, seat_no),
FOREIGN KEY (ticket_no, flight_id)
REFERENCES ticket_flights (ticket_no, flight_id)
);

10. Вывести на экран по очереди таблицы. В отчёт скриншот.

11. Добавить по три строки в каждую таблицу (в таблицу бронирования добавить строку с суммой в 1 204 500 рублей).

12. При выполнении выборок зачастую выполняются многотабличные запросы, включающие три таблицы и более. В качестве примера рассмотрим такую задачу: определить число пассажиров, не пришедших на регистрацию билетов и, следовательно, не вылетевших в пункт назначения. Будем учитывать только рейсы, у которых фактическое время вылета не пустое, т. е. рейсы, имеющие статус «Departed» или «Arrived».

SELECT count(*)
FROM (ticket_flights t
JOIN flights f ON t.flight_id = f.flight_id)
LEFT OUTER JOIN boarding_passes b
ON t.ticket_no = b.ticket_no AND t.flight_id = b.flight_id
WHERE f.actual_departure IS NOT NULL AND b.flight_id IS NULL;

В отчёт скриншот с результатом выборки и ответ на следующий вопрос: какие таблицы были использованы в этом запросе?

13. Для выработки финансовой стратегии нашей авиакомпании требуется следующая информация: распределение количества бронирований по диапазонам сумм с шагом в сто тысяч рублей. Максимальная сумма в одном бронировании составляет 1 204 500 рублей. Учтем это при формировании диапазонов стоимостей. Виртуальной таблице, создаваемой с помощью ключевого слова VALUES, присваивают имя с помощью ключевого слова AS. После имени в круглых скобках приводится список имен столбцов этой таблицы.

SELECT r.min_sum, r.max_sum, count(b.*)
FROM bookings b
RIGHT OUTER JOIN
(VALUES (0, 100000), (100000, 200000),

```
( 200000, 300000 ),                    ( 300000, 400000 ),
( 400000, 500000 ),                    ( 500000, 600000 ),
( 600000, 700000 ),                    ( 700000, 800000 ),
( 800000, 900000 ),                    ( 900000, 1000000 ),
( 1000000, 1100000 ), ( 1100000, 1200000 ),
( 1200000, 1300000 )
) AS r ( min_sum, max_sum )
ON b.total_amount >= r.min_sum AND b.total_amount < r.max_sum
GROUP BY r.min_sum, r.max_sum
ORDER BY r.min_sum;
```

В этом запросе использовали внешнее соединение. Сделано это для того, чтобы в случаях, когда в каком-то диапазоне не окажется ни одного бронирования, результирующая строка выборки все же была бы сформирована. А правое соединение было выбрано только потому, что в качестве первой, базовой, таблицы мы выбрали таблицу «Бронирования» (bookings), но именно в ней может не оказаться ни одной строки для соединения с какой-либо строкой виртуальной таблицы. А все строки виртуальной таблицы, стоящей справа от предложения RIGHT OUTER JOIN, должны быть обязательно представлены в выборке: это позволит сразу увидеть «пустые» диапазоны, если они будут.

В этом запросе можно использовать и левое внешнее соединение, если поменять таблицы местами.

В отчёт скриншот с результатом запроса и комментарии по каждой строке запроса.

2.14. Практическая работа № 14 «Создание подзапросов. Выборка данных»

Задание 1:

1. Подключиться к базе данных demo.
Выведите информацию о всех ваших таблицах, представлениях и их полях с помощью команды:

SELECT table_name, column_name
FROM information_schema.columns
WHERE table_schema='public';

Затем добавьте к этой команде сортировку по имени таблицы и снова выведите на экран.
В отчёт скриншот с результатом.

2. Нахождение среднего, максимального и минимального значения с помощью агрегатных функций:
Для расчета среднего значения по столбцу используется функция avg (от слова average).
SELECT avg(total_amount) FROM bookings;
Для получения максимального значения по столбцу используется функция max.
SELECT max(total_amount) FROM bookings;
Для получения минимального значения по столбцу используется функция min.
SELECT min(total_amount) FROM bookings;
Все команды выполнить.
К любой из иных ваших таблиц применить данные функции.
Скриншот с результатами в отчёт.

3. Кроме обычных агрегатных функций существуют и так называемые оконные функции (window functions). Эти функции предоставляют возможность производить вычисления на множестве строк, логически связанных с текущей строкой, т. е. имеющих то или иное отношение к ней.
При работе с оконными функциями используются концепции раздела (partition) и оконного кадра (window frame). Предположим, что нужно вывести Количество проданных билетов в виде накопленного показателя, суммирование должно производиться в

пределах каждого календарного месяца.

```
SELECT b.book_ref,
b.book_date,
extract( 'month' from b.book_date ) AS month,
extract( 'day' from b.book_date ) AS day,
count( * ) OVER (
PARTITION BY date_trunc( 'month', b.book_date )
ORDER BY b.book_date
) AS count
FROM ticket_flights tf
JOIN tickets t ON tf.ticket_no = t.ticket_no
JOIN bookings b ON t.book_ref = b.book_ref
ORDER BY b.book_date;
```

Пояснение по date_trunc:

date_trunc(text, interval)	interval	Отсекает компоненты даты до заданной точности; см. также Подраздел 9.9.2	date_trunc('hour', interval '2 days 3 hours 40 minutes')	2 days 03:00:00

Выполнить данный запрос.
Прокомментировать построчно в отчёте выделенный цветом фрагмент запроса.

4. С помощью оконной функции rank можно проранжировать аэропорты в пределах каждого часового пояса на основе их географической широты. Причем будем присваивать более высокий ранг тому аэропорту, который находится севернее.

```
SELECT airport_name, city, round( latitude::numeric, 2 ) AS ltd, timezone,
rank() OVER (
PARTITION BY timezone
ORDER BY latitude DESC)
FROM airports
WHERE timezone IN ( ‘+3’ ) ORDER BY timezone, rank;
```

Выполнить запрос, просмотреть его результат.
Придумать похожий запрос на ранжирования с помощью функции rank по любой другой вашей таблице.
В отчёт скриншот с результатами.

5. Создадим подзапрос по следующему заданию:
Предположим, что сотрудникам аналитического отдела потребовалось провести статистическое исследование финансовых результатов работы авиакомпании. В качестве первого шага они решили подсчитать количество операций бронирования, в которых общая сумма превышает среднюю величину по всей выборке.

```
SELECT count( * ) FROM bookings
WHERE total_amount > ( SELECT avg( total_amount ) FROM bookings );
```

Выполнить запрос с подзапросом.
Построчно прокомментировать в отчёте запрос.
Далее выполнить запрос с подзапросом на нахождение самого западного и самого восточного аэропорта:

```
SELECT airport_name, city, longitude
FROM airports
WHERE longitude IN (
( SELECT max( longitude ) FROM airports ),
( SELECT min( longitude ) FROM airports ))
ORDER BY longitude;
```

Придумать такой же запрос на нахождение крайних значений по таблице Самолёты.
В отчёт скриншот с результатами.

6. Создадим ещё один запрос с подзапросом по следующему заданию:
Предположим, что для выработки ценовой политики авиакомпании необходимо знать, как

распределяются места разных классов в самолетах всех типов.

```
SELECT a.model,
( SELECT count( * )
FROM seats s
WHERE s.aircraft_code = a.aircraft_code
AND s.fare_conditions = 'Business'
) AS business,
( SELECT count( * )
FROM seats s
WHERE s.aircraft_code = a.aircraft_code
AND s.fare_conditions = 'Comfort'
) AS comfort,
( SELECT count( * )
FROM seats s
WHERE s.aircraft_code = a.aircraft_code
AND s.fare_conditions = 'Economy'
) AS economy
FROM aircrafts a
ORDER BY 1;
```

В результате данного запроса у вас на экране отобразится ошибка, связанная с неизвестным значением Business.
Ваша задача состоит в том, чтобы разобраться, исправить некорректность в запросе (ввести те данные, которые соответствуют данным вашей таблицы).
В результате правильного запроса вы на экране должны увидеть следующий результат:

```
model         | business | comfort | economy
--------------------+----------+---------+---------
 Airbus A319-100    |       4 |       2 |       0
 Airbus A320-200    |       4 |       3 |       0
 Airbus A321-200    |       5 |       3 |       0
 Boeing 737-300     |       4 |       2 |       0
 Boeing 767-300     |       4 |       2 |       0
 Boeing 777-300     |       4 |       2 |       0
 Bombardier CRJ-200 |       4 |       3 |       0
 Cessna 208 Caravan |       4 |       5 |       0
 Sukhoi SuperJet-100 |       4 |       3 |       0
```

В отчёт скриншот с вашей командой и результатом.

7. Выполнить следующие запросы:

```
SELECT count( * ) FROM tickets;
SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% %';
SELECT count( * ) FROM tickets WHERE passenger_name LIKE '% % %';
```

Сравните полученные результаты. Прокомментируйте различия в отчёте по этим запросам.

Задание 2:

Подключиться к базе данных demo.

1. В предыдущей практической работе был следующий запрос:

```
SELECT a.model,
( SELECT count( * )
FROM seats s
WHERE s.aircraft_code = a.aircraft_code
```

```sql
AND s.fare_conditions = '1') AS business,
( SELECT count( * )
FROM seats s
WHERE s.aircraft_code = a.aircraft_code
AND s.fare_conditions = '2') AS comfort,
( SELECT count( * )
FROM seats s WHERE s.aircraft_code = a.aircraft_code
AND s.fare_conditions = '3') AS economy
FROM aircrafts a
ORDER BY 1;
```

В результате его выполнения вы получили данные по местам разных классов. Добавьте данные в нужные таблицы так, чтобы не было нулевых значений в целом классе и выполните этот запрос снова. Вставьте скриншот в отчёт.

2. Составить запрос с подзапросом, выводящим на экран количество забронированных билетов с суммой бронирования больше минимальной суммы. Вставьте скриншот в отчёт с командой и её результатом

3. Выполните следующий запрос с подзапросами:

```sql
SELECT flight_no, departure_city, arrival_city
FROM flights
WHERE departure_city IN (
SELECT city
FROM airports
WHERE timezone = '+3')
AND arrival_city IN (
SELECT city
FROM airports
WHERE timezone = '+3' );
```

В отчёте поясните что показывает данный запрос, выполните его с другими часовыми поясами, а также для вывода на экран замените атрибуты на русские слова (

```
Номер_рейса | Город_отправления | Город_прибытия
------------+-------------------+----------------
```

).

В отчёт вставить скриншот с результатами.

4. В таблице Аэропорты названия городов замените на русские слова.
Следующий запрос показывает в какие города нет рейсов из Москвы:

```sql
SELECT DISTINCT a.city FROM airports a
WHERE NOT EXISTS (
SELECT * FROM routes r
WHERE r.departure_city = 'Москва'
AND r.arrival_city = a.city)
AND a.city <> 'Москва'
ORDER BY city;
```

Данный запрос ссылается на материализованное представление routes. В нашей базе данных его нет, поэтому замените его на таблицу Рейсы и выполните. Добавьте данные в таблицы таким образом, чтобы рейсы из Москвы, из Сыктывкара, из Екатеринбурга были (можно по одной строчке из каждого города). Выполните данный запрос снова, а также с двумя другими городами. В отчёт вставить скриншот с результатами.

5. Выполнить следующий запрос:

```sql
SELECT s2.model, string_agg( s2.fare_conditions || ' ('||s2.num || ')', ', ')
```

FROM (SELECT a.model, s.fare_conditions, count(*) AS num
FROM aircrafts a
JOIN seats s ON a.aircraft_code = s.aircraft_code
GROUP BY 1, 2
ORDER BY 1, 2) AS s2
GROUP BY s2.model
ORDER BY s2.model;
В отчёт написать, что это запрос делает. Поменять название столбца string_agg. В отчёт вставить скриншот с результатами.

6. Выполните следующий запрос:
SELECT aa.city, aa.airport_code, aa.airport_name
FROM (
SELECT city, count(*)
FROM airports
GROUP BY city
HAVING count(*) > 1) AS a
JOIN airports AS aa ON a.city = aa.city
ORDER BY aa.city, aa.airport_name;
Данный запрос показывает города, в которых более одного аэропорта. Если у вас в ответе 0 строк, то добавьте в таблицу Аэропорты какой-нибудь дополнительный аэропорт к городу.

7. Составьте выборку рейсов, где используются самолеты компании Boeing? В выборке вместо кода модели должно выводиться ее наименование, например, вместо кода 733 должно быть Boeing 737-300.
В отчёт вставить скриншот с результатами.

2.15 Практическая работа № 15 «Работа с функцией unnest»

Задание:

1. Подключиться к базе данных demo. Создать таблицу Гражданство с атрибутами pas_id и Гражданство. С помощью функции unnest добавьте трём пассажирам по их pas_id гражданство RUS, четверым — BY.
В отчёт вставьте скриншот с командами и результатом.

2. Подключитесь к базе данных edu. Создать таблицу Группы с атрибутами Фамилия и номер_группы. С помощью функции unnest добавьте пяти студентам одну группу и трём другую группу.
В отчёт вставьте скриншот с командами и результатом.

3. Самые крупные самолеты в нашей авиакомпании — это Boeing 777-300. Выяснить, между какими парами городов они летают, поможет запрос:
SELECT DISTINCT departure_city, arrival_city
FROM flights f
JOIN aircrafts a ON f.aircraft_code = a.aircraft_code
WHERE a.model = 'Boeing 777-300' ORDER BY 1;
Если в вашем запросе нет данных, то добавьте в таблицу flights необходимые данные. Составьте подобный запрос с двумя другими самолётами, добавив номер рейса в результирующую выборку.

Создайте представление по данному запросу.
Скриншот с командами и результатом в отчёт.

4. Составьте запрос с максимальными и минимальными ценами билетов на все направления.
Скриншот с командой и результатом в отчёт.

5. Составить запрос со списком пассажиров с любого одного рейса с их местами.
Скриншот с командой и результатом в отчёт.

2.16 Практическая работа № 16
«Создание запросов с общим табличным выражением»

Задание:

1. Подключиться к базе данных demo.
Выполните следующий запрос с использованием общего табличного выражения (Common Table Expression — CTE):

```
WITH ts AS
( SELECT f.flight_id, f.flight_no,
f.scheduled_departure_local,
f.departure_city,
f.arrival_city,
f.aircraft_code, count( tf.ticket_no ) AS fact_passengers,
( SELECT count( s.seat_no )
FROM seats s
WHERE s.aircraft_code = f.aircraft_code
) AS total_seats
FROM flights_v f
JOIN ticket_flights tf ON f.flight_id = tf.flight_id
WHERE f.status = 'Arrived'
GROUP BY 1, 2, 3, 4, 5, 6
)
SELECT ts.flight_id, ts.flight_no,
ts.scheduled_departure_local,
ts.departure_city, ts.arrival_city,
a.model, ts.fact_passengers, ts.total_seats,
round( ts.fact_passengers::numeric / ts.total_seats::numeric, 2 ) AS fraction
FROM ts
JOIN aircrafts AS a ON ts.aircraft_code = a.aircraft_code
ORDER BY ts.scheduled_departure_local;
```

В отчёт вставить скриншот с результатом выполнения табличного выражения.
В отчёте прокомментировать построчно данное табличное выражение

2. Выполните запрос, формирующий диапазоны сумм бронирований с помощью рекурсивного общего табличного выражения:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
( VALUES ( 0, 100000 )
UNION ALL
SELECT min_sum + 100000, max_sum + 100000
FROM ranges
WHERE max_sum <
( SELECT max( total_amount ) FROM bookings ) )
SELECT * FROM ranges;
```

В отчёт вставить скриншот с результатом выполнения табличного выражения. В отчёте прокомментировать построчно данное табличное выражение

3. Выполните общее табличное выражение с выборкой из таблицы bookings:

```
WITH RECURSIVE ranges ( min_sum, max_sum ) AS
( VALUES( 0, 100000 )
UNION ALL
SELECT min_sum + 100000, max_sum + 100000
FROM ranges
WHERE max_sum <
( SELECT max( total_amount ) FROM bookings ) )
SELECT r.min_sum, r.max_sum, count( b.* )
FROM bookings b
RIGHT OUTER JOIN ranges r
ON b.total_amount >= r.min_sum
AND b.total_amount < r.max_sum
GROUP BY r.min_sum, r.max_sum ORDER BY r.min_sum;
```

В отчёт вставить скриншот с результатом выполнения табличного выражения. В отчёте прокомментировать построчно данное табличное выражение

4. Создать по следующему коду материализованное представление Маршруты (разобраться с ошибкой: столбец "duration" имеет псевдотип record):

```
CREATE MATERIALIZED VIEW routes AS
WITH f3 AS
( SELECT f2.flight_no, f2.departure_airport, f2.arrival_airport, f2.aircraft_code,
f2.duration,
array_agg( f2.days_of_week ) AS days_of_week
FROM ( SELECT f1.flight_no, f1.departure_airport, f1.arrival_airport,
f1.aircraft_code, f1.duration, f1.days_of_week
FROM ( SELECT flights.flight_no, flights.departure_airport, flights.arrival_airport,
flights.aircraft_code,
( flights.scheduled_arrival, flights.scheduled_departure
) AS duration,
( to_char( flights.scheduled_departure,
'ID'::text ))::integer AS days_of_week
FROM flights
) f1
GROUP BY f1.flight_no, f1.departure_airport, f1.arrival_airport, f1.aircraft_code,
f1.duration, f1.days_of_week
ORDER BY f1.flight_no, f1.departure_airport, f1.arrival_airport, f1.aircraft_code,
f1.duration, f1.days_of_week
) f2
GROUP BY f2.flight_no, f2.departure_airport, f2.arrival_airport, f2.aircraft_code,
f2.duration )
SELECT f3.flight_no, f3.departure_airport,
dep.airport_name AS departure_airport_name, dep.city AS departure_city,
f3.arrival_airport,
arr.airport_name AS arrival_airport_name, arr.city AS arrival_city, f3.aircraft_code,
f3.duration, f3.days_of_week
FROM f3, airports dep, airports arr
WHERE f3.departure_airport = dep.airport_code
AND f3.arrival_airport = arr.airport_code;
```

2.17 Практическая работа № 17
«Создание простых индексов»

Задание:

Подключиться к базе данных demo.

1. В качестве примера создадим индекс для таблицы «Аэропорты» (airports) по столбцу airport_name:

CREATE INDEX ON airports (airport_name);

Посмотрим описание нового индекса:

\d airports

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

\timing on

Теперь psql будет сообщать время, затраченное на выполнение всех команд.

3. Для практической проверки влияния индекса на скорость выполнения выборок сначала выполним следующий запрос:

SELECT count(*) FROM tickets WHERE passenger_name = 'IVAN IVANOV';

Имя пассажира использовать из вашей таблицы.

В отчёт скриншот с результатом.

4. 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);

Посмотрим описание нового индекса:

\d tickets

5. Теперь выполним ту же выборку из таблицы tickets:

SELECT count(*) FROM tickets WHERE passenger_name = 'IVAN IVANOV';

В отчёт скриншот с результатом и ответить на вопрос: как изменилось время при наличии индекса?

6. Просмотрите список всех индексов в текущей базе данных с помощью команды \di.

В отчёте ответьте на вопрос: сколько индексов существует в вашей базе данных?

7. Давайте удалим созданный нами индекс для таблицы tickets:

DROP INDEX passenger_name;

8. Создайте индекс по нескольким столбцам:

CREATE INDEX tickets_book_ref_test_key ON tickets (book_ref);

9. Выполните запрос, в котором используется предложение LIMIT:

SELECT * FROM tickets ORDER BY book_ref LIMIT 5;

В отчёт вставьте скриншот с результатами.

10. Удалите этот индекс и повторите запрос.

В отчёте ответьте на вопрос: как изменилось время?

11. Перейдите к базе данных edu.
    - Включите секундомер.
    - Придумайте и выполните любой запрос к таблице.
    - Затем создайте к этой таблице 1 индекс по одному столбцу.
    - Выполните этот запрос ещё раз.
    - В отчёт поместите скриншот и комментарий по времени выполнения запросов.

— Придумайте и выполните запрос к другой таблице.
— Создайте индекс по двум столбцам.
— Повторите выполнение запроса.
— В отчёт поместите скриншот и комментарий по времени выполнения запросов.

2.18 Практическая работа № 18 «Создание индексов на основе выражений»

Задание:

Подключиться к базе данных demo.
Включите секундомер.
1. Индексы могут использоваться для обеспечения уникальности значений атрибутов в строках таблицы.
Создадим уникальный индекс по столбцу model для таблицы «Самолеты» (aircrafts):

```
CREATE UNIQUE INDEX aircrafts_unique_model_key
ON aircrafts ( model );
```

В этом случае мы уже не сможем ввести в таблицу aircrafts строки, имеющие одинаковые наименования моделей самолетов.
2. Удалите данные индекс.
3. В команде создания индекса можно использовать не только имена столбцов, но также функции и скалярные выражения, построенные на основе столбцов таблицы. Например, если мы захотим запретить значения столбца model в таблице aircrafts, отличающиеся только регистром символов, то создадим такой индекс:

```
CREATE UNIQUE INDEX aircrafts_unique_model_key
ON aircrafts ( lower( model ) );
```

Встроенная функция lower преобразует все символы в нижний регистр. Индекс строится уже на основе преобразованных значений, поэтому при поиске строки в таблице искомое значение сначала переводится в нижний регистр, а затем осуществляется поиск в индексе.
4. Попробуйте теперь добавить строку:

```
INSERT INTO aircrafts VALUES ( '123', 'Cessna 208 CARAVAN', 1300);
```

Должна появится ошибка.
В отчёт скриншот вставьте.
5. Создание частичных индексов. Сначала выполните запрос:

```
SELECT * FROM bookings
WHERE total_amount > 1000000 ORDER BY
book_date DESC;
```

В отчёт вставьте скриншот с запросом, его временем выполнения и ответом на вопрос: что данный запрос показывает?
Создадим индекс:

```
CREATE INDEX bookings_book_date_part_key
ON bookings ( book_date )
WHERE total_amount > 1000000;
```

6. Выполните снова запрос.
В отчёт скриншот с запросом, его временем и ответом на вопросы:
Как изменилось время?
Чем отличается данный частичный индекс от простых индексов?
7. Выполните следующие запросы:

```
SELECT * FROM bookings WHERE total_amount > 1100000 ...
SELECT * FROM bookings WHERE total_amount > 900000 ...
```

В отчёт скриншот.
В отчёте ответьте на вопрос:
В каком запросе использовался индекс, а в каком нет?

2.19 Практическая работа № 19 «Использование транзакций»

Задание:

Подключиться к базе данных demo.
Включите секундомер на одном терминале.
1. Для проведения экспериментов воспользуемся таблицей «Самолеты» (aircrafts).
CREATE TABLE aircrafts_tmp AS SELECT * FROM aircrafts;
Для организации выполнения параллельных транзакций с использованием утилиты psql будем запускать ее на двух терминалах.
Для изучения уровня изоляции READ UNCOMMITTED проделаем следующие эксперименты.
2. На первом терминале выполним следующие команды:
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SHOW transaction_isolation;
UPDATE aircrafts_tmp
SET range = range + 100
WHERE aircraft_code = 'SU9';
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
3. Откройте второй терминал и также войдите в вашу базу данных.
Начнем транзакцию на втором терминале (все, что происходит на втором терминале, показано на сером фоне):

BEGIN;
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT * FROM aircrafts_tmp 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.
4. Не будем фиксировать произведенное изменение в базе данных, а воспользуемся командой ROLLBACK для отмены транзакции, т. е. для ее отката. На первом терминале:

ROLLBACK;
На втором терминале сделаем так же:
ROLLBACK;
5. Теперь обратимся к уровню изоляции READ COMMITTED. Именно этот уровень установлен в PostgreSQL по умолчанию.
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.
На первом терминале выполним следующие команды:
BEGIN ISOLATION LEVEL READ COMMITTED;
SHOW transaction_isolation;
UPDATE aircrafts_tmp
SET range = range + 100

WHERE aircraft_code = 'SU9';
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
Мы видим, 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.
6. На втором терминале так и сделаем. Во второй транзакции попытаемся обновить эту же строку таблицы airctafts_tmp, но для того, чтобы впоследствии разобраться, какое из изменений прошло успешно и было зафиксировано, добавим к значению атрибута range не 100, а 200.

BEGIN;
UPDATE aircrafts_tmp
SET range = range + 200
WHERE aircraft_code = 'SU9';
И вот мы видим, 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.
7. Завершим первую транзакцию с фиксацией изменений:

COMMIT;
Перейдя на второй терминал, мы увидим, что команда UPDATE завершилась.
Теперь на втором терминале, не завершая транзакцию, посмотрим, что стало с нашей строкой в таблице aircrafts_tmp:
SELECT * FROM aircrafts_tmp WHERE aircraft_code = 'SU9';
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.
8. Завершим транзакцию на втором терминале, но при этом вместо команды COMMIT воспользуемся эквивалентной командой END, которая является расширением PostgreSQL:

END;
9. Для иллюстрации эффекта неповторяющегося чтения данных проведем совсем простой эксперимент также на двух терминалах.
На первом терминале:

BEGIN;
SELECT * FROM aircrafts_tmp;
На втором терминале:
BEGIN;
DELETE FROM aircrafts_tmp WHERE model ~ '^Boe';
SELECT * FROM aircrafts_tmp;
Сразу завершим вторую транзакцию:
END;
Повторим выборку в первой транзакции:
SELECT * FROM aircrafts_tmp;

Видим, что теперь получен другой результат, т. к. вторая транзакция завершилась в момент времени между двумя запросами. Таким образом, налицо эффект неповторяющегося чтения данных, который является допустимым на уровне изоляции READ COMMITTED.
Завершим и первую транзакцию:
END;
COMMIT;

2.20 Практическая работа № 20 «Работа с командой EXPLAIN»

Задание:

Подключиться к базе данных demo.
Прежде чем приступить к непосредственному выполнению каждого запроса, PostgreSQL формирует план его выполнения. Чтобы достичь хорошей производительности, этот план должен учитывать свойства данных. Планированием занимается специальная подсистема — планировщик (planner).
1. Просмотреть план выполнения запроса можно с помощью команды EXPLAIN.
EXPLAIN SELECT * FROM aircrafts;

Поскольку в этом запросе нет предложения WHERE, он должен просмотреть все строки таблицы, поэтому планировщик выбирает последовательный просмотр (sequential scan). В скобках приведены важные параметры плана.
Первое число означает оценку ресурсов, требуемых для того, чтобы приступить к выводу данных. В нашем примере эта оценка равна нулю, поскольку никакие дополнительные операции с выбранными строками не предполагаются, и PostgreSQL может сразу же выводить прочитанные строки.
Второе число — это оценка общей стоимости выполнения запроса. Формируя эту оценку, планировщик исходит из предположения, что данный узел плана запроса выполняется до конца, т. е. извлекаются все имеющиеся строки таблицы.
Далее в выводе идет общее число строк, которые должны быть извлечены (возвращены) на данном узле плана, также при условии выполнения этого узла до полного завершения.
Последним параметром узла плана идет оценка среднего размера строк, которые выводятся на данном узле плана запроса.
2. В том случае, когда нас не интересуют численные оценки, можно воспользоваться параметром COSTS OFF:
EXPLAIN (COSTS OFF) SELECT * FROM aircrafts;
3. Сформируем запрос с предложением WHERE:
EXPLAIN SELECT * FROM aircrafts WHERE model ~ 'Air';

В данном случае планировщик неточно оценил число выбираемых строк — фактически их будет три.
Обратите внимание, что по своей форме вывод команды EXPLAIN также является выборкой, поэтому в конце выборки, как обычно, выводится информация о числе строк в ней, т. е. в дереве плана.
4. Теперь усложним запрос, добавив в него сортировку данных:
EXPLAIN SELECT * FROM aircrafts ORDER BY aircraft_code;

Дополнительный узел обозначен на плане символами «−>».
5. Обратимся к таблице «Бронирования» (bookings) для иллюстрации сканирования по индексу.
EXPLAIN SELECT * FROM bookings ORDER BY book_ref;

6. Если к сортировке добавить еще и условие отбора строк, то это отразится в дополнительной строке верхнего (и единственного) узла плана.
EXPLAIN
SELECT * FROM bookings

WHERE book_ref > '0000FF' AND book_ref < '000FFF'
ORDER BY book_ref;

Обратите внимание, что поскольку столбец, по которому производится отбор строк, является индексируемым, то их отбор реализуется не через Filter, а через Index Cond.

7. Теперь проиллюстрируем метод сканирования на основе битовой карты на примере таблицы «Места» (seats).
EXPLAIN SELECT * FROM seats WHERE aircraft_code = 'SU9';

В этом плане в нижнем узле строится битовая карта, а в верхнем узле с помощью этой карты сканируются страницы таблицы seats. Здесь также для отбора строк в соответствии с предложением WHERE используется индекс — Index Cond. Обратите внимание, что значение параметра width при создании битовой карты равно нулю, поскольку сами строки на этом этапе еще не выбираются.
8. Если нам будет нужно выбрать только номера бронирований в каком-то диапазоне, то обращения к таблице не потребуется: достаточно сканирования исключительно по индексу.
EXPLAIN
SELECT book_ref
FROM bookings
WHERE book_ref < '000FFF'
ORDER BY book_ref;
9. Посмотрим, как отражаются в планах выполнения запросов различные агрегатные функции. Начнем с простого подсчета строк.
EXPLAIN
SELECT count(*)
FROM seats
WHERE aircraft_code = 'SU9';

В верхнем узле плана выполняется агрегирование — Aggregate. А в нижних узлах подготавливаются строки с помощью сканирования на основе формирования битовой карты.
10. А в этом примере агрегирование связано уже с вычислениями на основе значений конкретного столбца, а не просто с подсчетом строк.
EXPLAIN SELECT avg(total_amount) FROM bookings;

В отчёте ответить на вопрос: чем отличается этот план от предыдущего?

11. Теперь обратимся к методам, которые используются для формирования соединений наборов строк. Начнем с метода вложенного цикла (nested loop). Для получения списка мест в салонах самолетов Airbus с указанием класса обслуживания сформируем запрос, в котором соединяются две таблицы: «Места» (seats) и «Самолеты» (aircrafts).
EXPLAIN
SELECT a.aircraft_code, a.model,
s.seat_no, s.fare_conditions
FROM seats s
JOIN aircrafts a ON s.aircraft_code = a.aircraft_code
WHERE a.model ~ '^Air'
ORDER BY s.seat_no;

Результат в базе данных преподавателя:

```
                               QUERY PLAN
---------------------------------------------------------------------------
-
 Sort  (cost=3.07..3.09 rows=7 width=55)
   Sort Key: s.seat_no
   ->  Hash Join  (cost=1.12..2.97 rows=7 width=55)
         Hash Cond: (s.aircraft_code = a.aircraft_code)
         ->  Seq Scan on seats s  (cost=0.00..1.62 rows=62 width=11)
         ->  Hash  (cost=1.11..1.11 rows=1 width=48)
               ->  Seq Scan on aircrafts a  (cost=0.00..1.11 rows=1 width=48)
                     Filter: (model ~ '^Air'::text)
(8 строк)
```

Результат в учебном пособии:

```
     Sort Key: s.seat_no
     -> Nested Loop (cost=5.43..17.90 rows=149 width=59)
           -> Seq Scan on aircrafts a (cost=0.00..1.11 rows=1 width=48) Filter: (model ~ '^Air'::text)
                    -> Bitmap Heap Scan on seats s (cost=5.43..15.29 rows=149
  width=15)
                Recheck Cond: (aircraft_code = a.aircraft_code) -> Bitmap Index Scan on
                seats_pkey (cost=0.00..5.39
  rows=149 width=0)
                      Index Cond: (aircraft_code = a.aircraft_code)
(9 строк)
```

В отчёт поместите ваш результат планирования. Сравните все три варианта Ответьте на вопрос: как вы думаете почему планы отличаются?

12. Следующий метод соединения наборов строк — соединение хешированием (hash join). Получим список маршрутов с указанием модели самолета, выполняющего рейсы по этим маршрутам. Воспользуемся таблицами «Маршруты» (routes) и «Самолеты» (aircrafts).

EXPLAIN
SELECT r.flight_no, r.departure_airport_name,
r.arrival_airport_name, a.model
FROM routes r
JOIN aircrafts a ON r.aircraft_code = a.aircraft_code
ORDER BY flight_no;

На самом внутреннем уровне плана последовательно сканируется (Seq Scan) таблица aircrafts, и формируется хеш-таблица, ключами которой являются значения атрибута aircraft_code, т. к. именно по нему выполняется соединение таблиц. Затем последовательно сканируется (Seq Scan) таблица routes, и для каждой ее строки выполняется поиск значения атрибута aircraft_code среди ключей хеш-таблицы: Hash Cond: (r.aircraft_code = a.aircraft_code). Если такой поиск успешен, значит, формируется комбинированная результирующая строка выборки. На верхнем уровне плана сформированные строки сортируются. Обратите внимание, что хеш-таблица создана на основе той таблицы, число строк в которой меньше, т. е. aircrafts. Таким образом, поиск в ней будет выполняться быстрее, чем если бы хеш-таблица была создана на основе таблицы routes.

13. Последний из методов соединения наборов строк — соединение слиянием (merge join). Для иллюстрации воспользуемся простым запросом, построенным на основе таблиц «Билеты» (tickets) и «Перелеты» (ticket_flights). Он выбирает для каждого билета все перелеты, включенные в него. Конечно, это очень упрощенный запрос, в реальной ситуации он не представлял бы большой практической пользы, но в целях упрощения плана и повышения наглядности, воспользуемся им.

EXPLAIN
SELECT t.ticket_no, t.passenger_name, tf.flight_id, tf.amount
FROM tickets t

JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
ORDER BY t.ticket_no;
Результат в Учебном пособии:

```
                    QUERY PLAN
---------------------------------------------------------------
 Merge Join (cost=1.51..98276.90 rows=1045726 width=40)
    Merge Cond: (t.ticket_no = tf.ticket_no)
    -> Index Scan using tickets_pkey on tickets t (cost=0.42..17230.42
 rows=366733 width=30)
    -> Index Scan using ticket_flights_pkey on ticket_flights tf
 (cost=0.42..67058.74 rows=1045726 width=24)
(4 строки)
```

Поместите в отчёт ваш результат.
Ответьте на следующие вопросы:
— происходит ли у вас сканирование по индексам как в Учебном пособии?
— какова оценка стоимости выполнения всех операций (cost в merge join) в вашем планировании? Если она отличается от значения в Учебном пособии, то почему она может отличаться?
Управление планировщиком
Для управления планировщиком предусмотрен целый ряд параметров. Их можно изменить на время текущего сеанса работы с помощью команды SET. Конечно, изменять параметры в производственной базе данных следует только в том случае, когда вы обоснованно считаете, что планировщик ошибается. Однако для того чтобы научиться видеть ошибки планировщика, нужен большой опыт. Поэтому следует рассматривать приведенные далее команды управления планировщиком лишь с позиции изучения потенциальных возможностей управления им, а не как рекомендацию к бездумному изменению этих параметров в реальной работе.
14. Например, чтобы запретить планировщику использовать метод соединения на основе хеширования, нужно сделать так:
SET enable_hashjoin = off;

15. Чтобы запретить планировщику использовать метод соединения слиянием, нужно сделать так:
SET enable_mergejoin = off;

16. А для того чтобы запретить планировщику использовать соединение методом вложенного цикла, нужно сделать так:
SET enable_nestloop = off;
По умолчанию все эти параметры имеют значение «on» (включено).

17. Теперь повторим предыдущий запрос:
EXPLAIN
SELECT t.ticket_no, t.passenger_name, tf.flight_id, tf.amount
FROM tickets t
JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
ORDER BY t.ticket_no;

В отчёт скриншот и ответ на вопрос:
Что изменилось в планировании при включённых запретах планировщику?
18. В команде EXPLAIN можно указать опцию ANALYZE, что позволит выполнить запрос и вывести на экран фактические затраты времени на выполнение запроса и число фактически выбранных строк. При этом, хотя запрос и выполняется, сами результирующие строки не выводятся.
Сначала разрешим планировщику использовать метод соединения слиянием:
SET enable_mergejoin = on;

19. Повторим предыдущий запрос с опцией ANALYZE.
EXPLAIN ANALYZE
SELECT t.ticket_no, t.passenger_name, tf.flight_id, tf.amount
FROM tickets t
JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
ORDER BY t.ticket_no;

В отчёт скриншот и ответ на вопрос:
Что изменилось в планировании? Какие данные добавились?
20. Если модифицировать запрос, добавив предложение WHERE, то точного совпадения оценки числа выбираемых строк и фактического их числа уже не будет.

EXPLAIN ANALYZE
SELECT t.ticket_no, t.passenger_name, tf.flight_id, tf.amount
FROM tickets t
JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
WHERE amount > 50000
ORDER BY t.ticket_no;

В отчёт скриншот и ответ на вопрос:
Что изменилось в планировании?
21. Обратимся еще раз к запросу, который мы уже рассматривали выше, и выполним его с опцией ANALYZE. В плане этого запроса нас будет интересовать фактический параметр loops.
EXPLAIN ANALYZE
SELECT a.aircraft_code, a.model,
s.seat_no, s.fare_conditions
FROM seats s
JOIN aircrafts a ON s.aircraft_code = a.aircraft_code
WHERE a.model ~ '^Air'
ORDER BY s.seat_no;

В отчёт скриншот и пояснения к планированию.

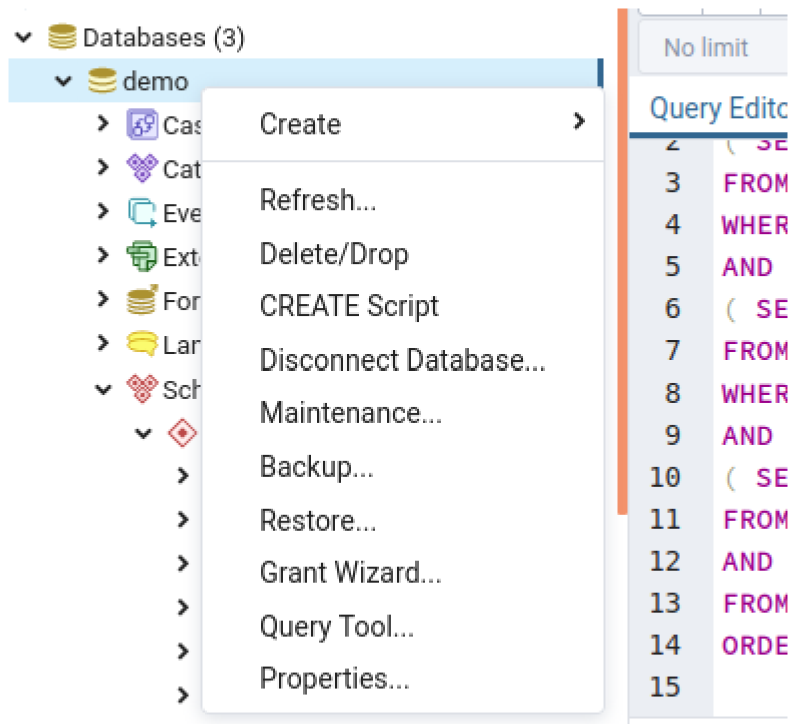
22. До сих пор мы рассматривали только выборки, т. е. такие запросы, которые не изменяют хранимых данных. Однако, кроме выборок, есть такие операции, как вставка, обновление и удаление строк. Нужно помнить, что хотя результаты выборки и не выводятся, тем не менее, она фактически все равно выполняется. Поэтому если требуется исследовать план выполнения запроса, модифицирующего данные, то для того, чтобы изменения на самом деле произведены не были, нужно воспользоваться транзакцией с откатом изменений.
BEGIN;
EXPLAIN ANALYZE
UPDATE aircrafts
SET range = range + 100
WHERE model ~ '^Air';
ROLLBACK;

2.21 Практическая работа № 21
«Изменение схемы данных для повышения производительности базы данных»

Задание:

1. Запустить pgadmin4.

2. Открыть базу данных demo.



3. В контекстном меню выбрать Query Tool. В открывшееся справа окно вставить любой запрос (например, из работ 18, 19). Далее выбрать кнопку ⏵. Посмотрите как выглядит результат выполнения в графическом режиме.

4. Выполните так по три запроса к двум базам данных (demo и edu). Для каждого запроса включайте Explain (иконка – палец вверх) Результаты выполнения вставьте в отчёт.

5. В базе данных edu создайте кроме схемы данных по умолчанию public схему my.

6. Добавьте все таблицы из public в my. Не забывайте Обновлять для того, чтобы увидеть результат.

7. Поменяйте названия любых трёх атрибутов.

2.22 Практическая работа № 22 «Модификация запросов для повышения производительности базы данных»

Задание:

Подключиться к базе данных demo.
Перед выполнением упражнений нужно восстановить измененные значения параметров:
SET enable_hashjoin = on;
SET enable_nestloop = on;
Необходимым условием для того, чтобы планировщик выбрал правильный план, является наличие актуальной статистики. Если вы предполагаете, что планировщик опирается на неактуальную статистику, можно ее принудительно обновить с помощью команды ANALYZE.
1. Обновить статистику для таблицы aircrafts можно, выполнив команду **ANALYZE aircrafts;**

2. В качестве примера ситуации, в которой оптимизация запроса представляется обоснованной, рассмотрим следующую задачу. Предположим, что необходимо определить степень загруженности кассиров нашей авиакомпании в сентябре 2016 г. Для этого, в частности, требуется выявить распределение числа операций бронирования по числу билетов, оформленных в рамках этих операций. Другими словами, это означает, что нужно подсчи-

тать число операций бронирования, в которых был оформлен только один билет, число операций, в которых было оформлено два билета и т. д.

Эту задачу можно переформулировать так: для каждой строки, отобранной из таблицы «Бронирования» (bookings), нужно подсчитать соответствующие строки в таблице «Билеты» (tickets). Речь идет о строках, в которых значение поля book_ref такое же, что и в текущей строке таблицы bookings. Буквальное следование такой формулировке задачи приводит к получению запроса с коррелированным подзапросом в предложении SELECT. Но это еще не окончательное решение. Теперь нужно сгруппировать полученный набор строк по значениям числа оформленных билетов. Получаем такой запрос:

```
EXPLAIN
SELECT num_tickets, count( * ) AS num_bookings
FROM (
SELECT b.book_ref,
( SELECT count( * ) FROM tickets t WHERE t.book_ref = b.book_ref
)
FROM bookings b
WHERE date_trunc( 'mon', book_date ) = '2016-09-01'
) AS count_tickets( book_ref, num_tickets )
GROUP by num_tickets
ORDER BY num_tickets DESC;
```

В этом плане получены очень большие оценки общей стоимости выполнения запроса. Универсальной зависимости между оценкой стоимости и реальным временем выполнения запроса не существует. Не всегда можно даже приблизительно предположить, в какие затраты времени выльется та или иная оценка стоимости. Планировщик предполагает, что из таблицы tickets в подзапросе будет извлекаться всего по две строки, и эту операцию нужно будет проделать n раз: столько строк предположительно будет выбрано из таблицы bookings. Как видно из плана, для просмотра строк в таблице tickets используется ее последовательное сканирование. В результате оценка стоимости этого узла плана получается высокой.

Что можно сделать для ускорения выполнения запроса?

3. Давайте создадим индекс для таблицы tickets по столбцу book_ref, по которому происходит поиск в ней.

```
CREATE INDEX tickets_book_ref_key ON tickets ( book_ref );
```

4. Повторим запрос, добавив параметр ANALYZE в команду EXPLAIN.

В отчёт скриншот с планом и пояснением: помог индекс для повышения производительности?

5. Кроме создания индекса есть и другой способ: замена коррелированного подзапроса соединением таблиц:

```
EXPLAIN ANALYZE
SELECT num_tickets, count( * ) AS num_bookings
FROM (
SELECT b.book_ref, count( * )
FROM bookings b, tickets t
WHERE date_trunc( 'mon', b.book_date ) = '2016-09-01' AND t.book_ref = b.book_ref
GROUP BY b.book_ref
) AS count_tickets( book_ref, num_tickets )
GROUP by num_tickets
ORDER BY num_tickets DESC;
```

В отчёт скриншот и пояснения по результату планирования (помогла ли модификация запроса для повышения производительности).

Самостоятельно выполните команду EXPLAIN для запроса, содержащего общее табличное выражение (CTE). Посмотрите, на каком уровне находится узел плана, отвечающий за это выражение, как он оформляется. Учтите, что общие табличные выражения всегда ма-

териализуются, т. е. вычисляются однократно и результат их вычисления сохраняется в памяти, а затем все последующие обращения в рамках запроса направляются уже к этому материализованному результату.

6. Замена коррелированного подзапроса соединением таблиц является одним из способов повышения производительности.
Предположим, что мы задались вопросом: сколько маршрутов обслуживают самолеты каждого типа? При этом нужно учитывать, что может иметь место такая ситуация, когда самолеты какого-либо типа не обслуживают ни одного маршрута. Поэтому необходимо использовать не только представление «Маршруты» (routes), но и таблицу «Самолеты» (aircrafts).
Это первый вариант запроса, в нем используется коррелированный подзапрос.

```
EXPLAIN ANALYZE
SELECT a.aircraft_code AS a_code,
a.model,
( SELECT count( r.aircraft_code ) FROM routes r WHERE r.aircraft_code =
a.aircraft_code
) AS num_routes
FROM aircrafts a
GROUP BY 1, 2
ORDER BY 3 DESC;
```

А в этом варианте коррелированный подзапрос раскрыт и заменен внешним соединением. Причина использования внешнего соединения в том, что может найтись модель самолета, не обслуживающая ни одного маршрута, и если не использовать внешнее соединение, она вообще не попадет в результирующую выборку.

```
EXPLAIN ANALYZE
SELECT a.aircraft_code AS a_code,
a.model,
count( r.aircraft_code ) AS num_routes
FROM aircrafts a
LEFT OUTER JOIN routes r
ON r.aircraft_code = a.aircraft_code
GROUP BY 1, 2
ORDER BY 3 DESC;
```

Исследуйте планы выполнения обоих запросов. Попытайтесь найти объяснение различиям в эффективности их выполнения. Чтобы получить усредненную картину, выполните каждый запрос несколько раз. — результаты исследования, пояснения ваши вставить в отчёт.

2.23. Практическая работа № 23 «Использование изоляций»

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.
10. На первом терминале:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Сначала посмотрим содержимое таблицы:

```
SELECT * FROM aircrafts_tmp;
```

Обратите внимание, что после уже проведенных экспериментов в таблице осталось меньше строк, чем было вначале.
На втором терминале проведем ряд изменений:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Добавим одну строку:

```
INSERT INTO aircrafts_tmp VALUES ( 'IL9', 'Ilyushin IL96', 9800 );
```

А одну строку обновим:
UPDATE aircrafts_tmp SET range = range + 100
WHERE aircraft_code = '320';
END;
Переходим на первый терминал.
SELECT * FROM aircrafts_tmp;
На первом терминале ничего не изменилось: фантомные строки не видны, и также не видны изменения в уже существующих строках. Это объясняется тем, что снимок данных выполняется на момент начала выполнения первого запроса транзакции.
Завершим первую транзакцию тоже:
END;
А теперь посмотрим, что изменилось в таблице:
SELECT * FROM aircrafts_tmp;
Как видим, одна строка добавлена, а значение атрибута range у самолета Airbus A320200 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.
11. Теперь покажем ошибки сериализации.
Начнем транзакцию на первом терминале:
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE aircrafts_tmp
SET range = range + 100
WHERE aircraft_code = '320';
На втором терминале попытаемся обновить ту же строку:
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
UPDATE aircrafts_tmp
SET range = range + 200
WHERE aircraft_code = '320';
Команда UPDATE на втором терминале ожидает завершения первой транзакции. Перейдя на первый терминал, завершим первую транзакцию:

END;
Перейдя на второй терминал, увидим сообщение об ошибке:
ОШИБКА: не удалось сериализовать доступ из-за параллельного изменения
Поскольку обновление, произведенное в первой транзакции, не было зафиксировано на момент начала выполнения первого (и, в данном частном случае, единственного) запроса во второй транзакции, то возникает эта ошибка. Это объясняется вот чем. При выполнении обновления строки команда UPDATE во второй транзакции видит, что строка уже изменена. На уровне изоляции REPEATABLE READ снимок данных создается на момент начала выполнения первого запроса транзакции и в течение транзакции уже не меняется, т. е. новая версия строки не считывается, как это делалось на уровне READ COMMITTED. Но если выполнить обновление во второй транзакции без повторного считывания строки из таблицы, тогда будет иметь место потерянное обновление, что недопустимо. В результате генерируется ошибка, и вторая транзакция откатывается. Мы вводим команду END на втором терминале, но PostgreSQL выполняет не фиксацию (COMMIT), а откат:
END;
Если выполним запрос, то увидим, что было проведено только изменение в первой транзакции:

SELECT * FROM aircrafts_tmp WHERE aircraft_code = '320';
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, кото-

рая была прервана системой из-за обнаружения зависимостей чтения/записи между транзакциями

12. Для проведения эксперимента создадим специальную таблицу, в которой будет всего два столбца: один — числовой, а второй — текстовый. Назовем эту таблицу — modes.

CREATE TABLE modes (num integer, mode text);

Добавим в таблицу две строки.

INSERT INTO modes VALUES (1, 'LOW'), (2, 'HIGH');

Итак, содержимое таблицы имеет вид:

SELECT * FROM modes;

На первом терминале начнем транзакцию и обновим одну строку из тех двух строк, которые были показаны в предыдущем запросе.

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

В команде обновления строки будем использовать предложение RETURNING. Поскольку значение поля num не изменяется, то будет видно, какая строка была обновлена. Это особенно пригодится во второй транзакции.

UPDATE modes
SET mode = 'HIGH'
WHERE mode = 'LOW' RETURNING *;

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

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE modes
SET mode = 'LOW'
WHERE mode = 'HIGH' RETURNING *;

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.

Посмотрим, что получилось в первой транзакции:

SELECT * FROM modes;

А во второй транзакции:

SELECT * FROM modes;

Заканчиваем эксперимент. Сначала завершим транзакцию на первом терминале:

COMMIT;

А потом на втором терминале:

COMMIT;

ОШИБКА: не удалось сериализовать доступ из-за зависимостей чтения/записи между транзакциями

ПОДРОБНОСТИ: Reason code: Canceled on identification as a pivot, during

SELECT * FROM modes;

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:

Но этот результат 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.

14. Теперь обе транзакции можно выполнять на одном терминале. Первый вариант их упорядочения такой:

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE modes
SET mode = 'HIGH'
WHERE mode = 'LOW' RETURNING *;
END;
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE modes
SET mode = 'LOW'
WHERE mode = 'HIGH' RETURNING *;
END;
SELECT * FROM modes;

15. Во втором варианте упорядочения поменяем транзакции местами. Конечно, предварительно нужно привести таблицу в исходное состояние.

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE modes
SET mode = 'LOW'
WHERE mode = 'HIGH' RETURNING *;
END;
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;
UPDATE modes
SET mode = 'HIGH'
WHERE mode = 'LOW' RETURNING *;
END;
SELECT * FROM modes;

Теперь результат отличается от того, который был получен при реализации первого варианта упорядочения транзакций.

Изменение порядка выполнения транзакций приводит к разным результатам. Однако если бы при параллельном выполнении транзакций была зафиксирована и вторая из них, то полученный результат не соответствовал бы ниодному из продемонстрированных возможных результатов последовательного выполнения транзакций. Таким образом, выполнить сериализацию этих транзакций невозможно. Обратите внимание, что вторая команда UPDATE в обоих случаях обновляет не одну строку, а две.

16. Продемонстрируем использование транзакций на примере базы данных «Авиаперевозки». Для этого создадим новое бронирование и оформим два билета с двумя перелетами в каждом. Выберем в качестве уровня изоляции READ COMMITTED.

BEGIN;

Сначала добавим запись в таблицу «Бронирования», причем, значение поля total_amount назначим равным 0. После завершения ввода строк в таблицу «Перелеты» мы обновим это значение: оно станет равным сумме стоимостей всех забронированных перелетов. В качестве даты бронирования возьмем дату, которая была принята в качестве текущей в базе данных. Эту дату выдает функция now(), созданная в схеме bookings.

INSERT INTO bookings (book_ref, book_date, total_amount)
VALUES ('ABC123', bookings.now(), 0);

17. Оформим два билета на двух разных пассажиров.

INSERT INTO tickets
(ticket_no, book_ref, passenger_id, passenger_name)
VALUES ('9991234567890', 'ABC123', '1234 123456','IVAN PETROV');
INSERT INTO tickets

(ticket_no, book_ref, passenger_id, passenger_name)
VALUES ('9991234567891', 'ABC123', '4321 654321','PETR IVANOV');

18. Отправьте обоих пассажиров по маршруту Москва — Сыктывкар и обратно. Для этого вам нужно две следующие команды изменить под ваши данные

```
INSERT INTO ticket_flights
( ticket_no, flight_id, fare_conditions, amount )
VALUES ( '9991234567890', 5572, 'Business', 12500 ),
( '9991234567890', 13881, 'Economy', 8500 );
INSERT INTO ticket_flights
( ticket_no, flight_id, fare_conditions, amount )
VALUES ( '9991234567891', 5572, 'Business', 12500 ),
( '9991234567891', 13881, 'Economy', 8500 );
```

19. Подсчитаем общую стоимость забронированных билетов и запишем ее в строку таблицы «Бронирования»:

```
UPDATE bookings
SET total_amount = (
SELECT sum( amount )
FROM ticket_flights
WHERE ticket_no IN (
SELECT ticket_no
FROM tickets
WHERE book_ref = 'ABC123'))
WHERE book_ref = 'ABC123';
```

Проверим, что получилось.

```
SELECT * FROM bookings WHERE book_ref = 'ABC123';
```

2.24. Практическая работа № 24
«Использование блокировок — встроенных механизмов защиты информации»

Задание:

Подключиться к базе данных demo.

Включите секундомер на одном терминале.

1. На первом терминале организуйте транзакцию с уровнем изоляции READ COMMITTED
и выполните следующую команду:

```
SELECT * FROM aircrafts_tmp WHERE model ~ '^Air' FOR UPDATE;
```

2. На втором терминале организуйте аналогичную транзакцию и выполните точно такую же команду. Вы увидите, что ее выполнение будет приостановлено.

```
SELECT * FROM aircrafts_tmp WHERE model ~ '^Air' FOR UPDATE;
```

3. На первом терминале обновите одну строку, а затем завершите транзакцию:

```
UPDATE aircrafts_tmp
SET range = 5800
WHERE aircraft_code = '320';
```

Перейдя на второй терминал, вы увидите, что там была, наконец, выполнена выборка, которая показала уже измененные данные.

4. Завершите и вторую транзакцию.

5. Аналогичным образом можно организовать блокировки на уровне таблиц. Также на первом терминале организуйте транзакцию с уровнем изоляции READ COMMITTED и выполните команду блокировки всей таблицы в самом строгом режиме, в котором другим транзакциям доступ к этой таблице запрещен полностью:

```
LOCK TABLE aircrafts_tmp IN ACCESS EXCLUSIVE MODE;
```

6. На втором терминале выполните совершенно «безобидную» команду:

```
SELECT * FROM aircrafts_tmp WHERE model ~ '^Air';
```

Вы увидите, что выполнение команды SELECT на втором терминале будет задержано.
7. Прервите транзакцию на первом терминале командой ROLLBACK. Вы увидите, что на втором терминале команда будет успешно выполнена.
8. С помощью Документации по PostgreSQL (раздел 13.3 «Явные блокировки») найдите и вставьте в отчёт информацию о режимах блокировок на уровне таблицы.
9. С помощью Документации по PostgreSQL выясните на каких ещё уровнях бывают блокировки. Перечислите их в отчёте.
10. Самостоятельно ознакомьтесь с предложением FOR SHARE команды SELECT и выполните необходимые эксперименты. Используйте документацию: раздел 13.3.2 «Блокировки на уровне строк» и описание команды SELECT.