МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Методические указания по дисциплине

ТЕХНОЛОГИИ ХРАНЕНИЯ И ОБРАБОТКИ БОЛЬШИХ ДАННЫХ

Направление подготовки:

09.04.02 «Информационные системы и технологии»

Квалификация выпускника: магистр

Ставрополь, 2026

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

1. Цели и задачи освоения дисциплины. Методические указания по дисциплине «Технологии хранения и обработки больших данных» для студентов направления 09.04.02 «Информационные системы и технологии» охватывают теоретические аспекты проектирования и разработки приложений для высокопроизводительных систем, строящихся на основе систем управления базами данных. Внимание уделяется доступной технологии разработки приложений – стеку Python. В качестве СУБД в пособии предлагается использование PostgreSQL. Пособие предназначено для студентов, обладающих теоретическими знаниями в области проектирования приложений и практическими навыками программирования (предпочтительно языки Python, C#, HTML, JavaScript, Java, основы Linux). Цель учебного пособия: сформировать у студентов целостный взгляд на современные тенденции в области построения масштабируемых ИС; обеспечить студентов обширным теоретическим материалом, достаточным для освоения методик разработки приложений на основе СУБД.

2. Перечень планируемых результатов обучения по дисциплине (модулю), соотнесённых с планируемыми результатами освоения образовательной программы.

Код, формулировка компетенции	Код, формулировка индикатора	Планируемые результаты обучения по дисциплине (модулю), характеризующие этапы формирования компетенций, индикаторов
ОПК-5: Способен модернизировать и разрабатывать программное, техническое, организационное обеспечение информационных систем	ИД-1$_{ОПК-5}$ Анализирует программное, техническое, организационное обеспечение информационных систем	Анализирует программное, техническое, организационное обеспечение информационных систем
	ИД-2$_{ОПК-5}$ Модернизирует программное, техническое, организационное обеспечение информационных систем	Модернизирует программное, техническое, организационное обеспечение информационных систем
	ИД-3$_{ОПК-5}$ Разрабатывает программное, техническое, организационное обеспечение информационных систем	Разрабатывает программное, техническое, организационное обеспечение информационных систем

ЛАБОРАТОРНАЯ РАБОТА 1. УСТАНОВКА И НАСТРОЙКА СЕРВЕРА РЕЛЯЦИОННОЙ СУБД

1.1. Цель и содержание

Цель лабораторной работы: изучение принципов конфигурирования сервера СУБД и получение навыков настройки сервера.

В рамках лабораторной работы необходимо решить следующие задачи:

– установить сервер РСУБД;

– произвести первичную настройку сервера РСУБД;

– выполнить проверку работоспособности сервера;

– выполнить основные операции администрирования.

1.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

1.3. Теоретическая часть

PostgreSQL – свободная современная СУБД с широкими возможностями. Её используют такие компании, как Alibaba, Instagram, Skype, Yahoo и многие другие. Это говорит о надёжности системы, и при этом она является простой в установке, использовании и обслуживании. PostgreSQL является кроссплатформенной СУБД с открытым исходным кодом, поэтому её можно установить практически на любой сервер.Рассмотрим установку и настройку на примере Ubuntu:

1. Определение источника установки. PostgreSQL является очень популярным сервером баз данных, поэтому присутствует в официальных

репозиториях Ubuntu. Однако в PPA разработчиков PostgreSQL можно найти самую свежую версию. Если у вас нет потребности в самых последних возможностях данной СУБД, то текущий шаг можно пропустить. Иначе добавьте репозиторий PostgreSQL в системный список источников (рекомендуется реализовать данный шаг в рамках данного лабораторного практикума):

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

```
wget -q https://www.postgresql.org/media/keys/ACCC4CF8.asc -O - | sudo apt-key add -
```

2. Установка PostgreSQL. Необходимо выполнить команды (эта команда произведёт обновление индекса, что позволит устанавливать свежие и актуальные пакеты):

```
sudo apt-get update
```

Установка PostgreSQL из официальных репозиториев и из PPA производится одинаково. Загрузим и установим пакеты PostgreSQL и contrib (contrib предоставляет некоторый дополнительный функционал и утилиты):

```
sudo apt-get install postgresql postgresql-contrib
```

3. Подключение к серверу баз данных. Во время установки программы в системе автоматически была создана учётная запись администратора баз данных – postgres. На данном этапе доступ к системе баз данных можно получить только через неё. Вы можете либо переключиться в сессию учётной записи postgres и запустить там оболочку программы:

```
sudo su - postgres
psql
```

Можно также запустить оболочку от имени postgres без переключения сессии:

```
sudo -u postgres psql
```

Попав тем или иным способом в командную строку psql, вам необходимо знать, как из неё выйти. Это можно сделать с помощью ввода команды выхода:

```
\q
```

4. Создание новой роли. Если вы производили установку по инструкции, то к этому моменту в вашей СУБД есть только одна роль – postgres. Рекомендуется не использовать данную роль для работы со своими базами данных, а создавать для каждой базы новую роль (или несколько при необходимости). Для создания новой роли предусмотрены два стандартных способа:

– интерактивный режим, в котором достаточно ответить на несколько простых вопросов;

– команда для создания роли через командную строку СУБД.

Воспользуйтесь командой (не забудьте заменить username на желаемое имя пользователя, а password – на пароль для этого пользователя; в данной работе используются значения puser: puserxxx):

```
create user username with password 'password';
```

Имя указывается без кавычек, а пароль – в одинарных кавычках.

5. Создание базы данных. Находясь в режиме командной строки psql, создать базу данных можно командой create database и указав название базы данных. Например, чтобы воздать БД с именем vscale_db, выполните команду:

```
create database testing_db;
```

6. Назначение прав. Созданной ранее роли нужно назначить права на базу данных. В большинстве проектов, где у будет использоваться всего один пользователь базы данных, ему будут требоваться полные права. Выдать их можно следующим образом:

```
grant all privileges on database testing_db to username;
```

где testing_db – название базы данных, выбранное на шаге 5, а username – имя пользователя, заданное на шаге 4.

Вся минимально требующаяся предварительная настройка завершена.

7. Для подключения к БД необходимо воспользоваться командой:

```
psql -h localhost vscale_db username
```

1.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

1.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

1.6. Методика и порядок выполнения работы

1.6.1. Решение учебной задачи

Постановка задачи: Создать базу данных «cloud», содержащую одну таблицу «person». Выполнить следующие действия:

1. Добавить в таблицу «users» 1000 записей.

2. Продемонстрировать работу SQL-операторов выборки с условием.

3. Продемострировать работу SQL-оператора удаления записей.

Решение. Для реализации поставленной задачи необходимо выполнить следующие действия:

1. Создание базы данных cloud коммандой:

```
create database cloud
```

2. Назначим требуемые права:

```
grant all privileges on database cloud to puser;
```

3. Выйдем из консоли psql и выполнив команду подключения к серверу под полдьзователем puser, перейдем к выполнению задания (рисунок 1.1):

```
psql –h localhost cloud puser
```

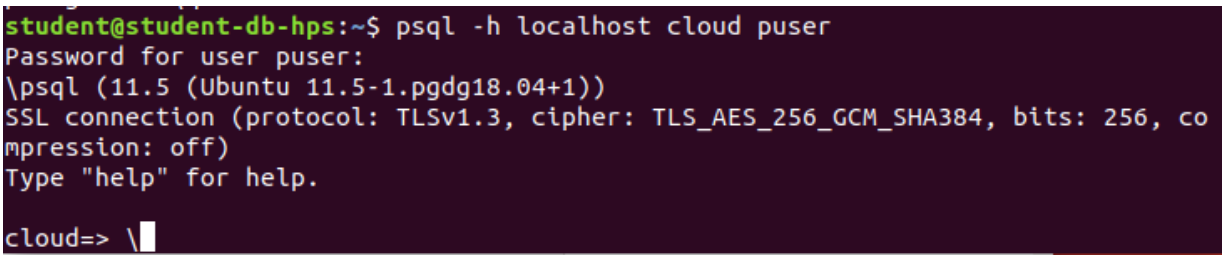

Рисунок 1.1 – Подключение к БД под учетными данными пользователя

4. Выведем таблицу базы данных cloud с использованием команды (на данном этапе работы с СУБД таблиц не создано):

```
\d
```

5. Необходимо выполнить команду для создания новой таблицы:

```
CREATE TABLE person (
id              BIGSERIAL NOT NULL PRIMARY KEY,
        first_name      VARCHAR(50) NOT NULL,
        last_nmae       VARCHAR(50) NOT NULL,
        gender          VARCHAR(8) NOT NULL,
        bdate           DATE,
        email           VARCHAR(50) );
```

После создания таблицы необходимо проанализировать вывод команд \d, \dt, \ds и ‘\d person’.

6. Произведем добавление записи в таблицу person. Для этого необходимо выполнить команду:

```
INSERT INTO person (
first_name, last_nmae, gender, bdate)
VALUES (
    ‘Evgeny’, ‘Nikolaev’, ‘male’, ‘1981-02-23’);
```

Выполним еще одну команду для добавления еще одной записи:

```
INSERT INTO person (
first_name, last_nmae, gender, bdate, email)
VALUES (
    ‘Mary, ‘Petrova’, ‘female’, ‘1989-03-11', 'mary@company.com');
```

Для выполнения задания необходимо гораздо больше записей, что приведет к определенное проблеме, так как выполнение отдельных команд вставки единственной записи (и так 1000 раз) не является рациональным решением.

7. Воспользуемся открытым ресурсом [1] для генерации 1000 записей, подходящих под структуру созданной таблицы (рисунок 1.2).

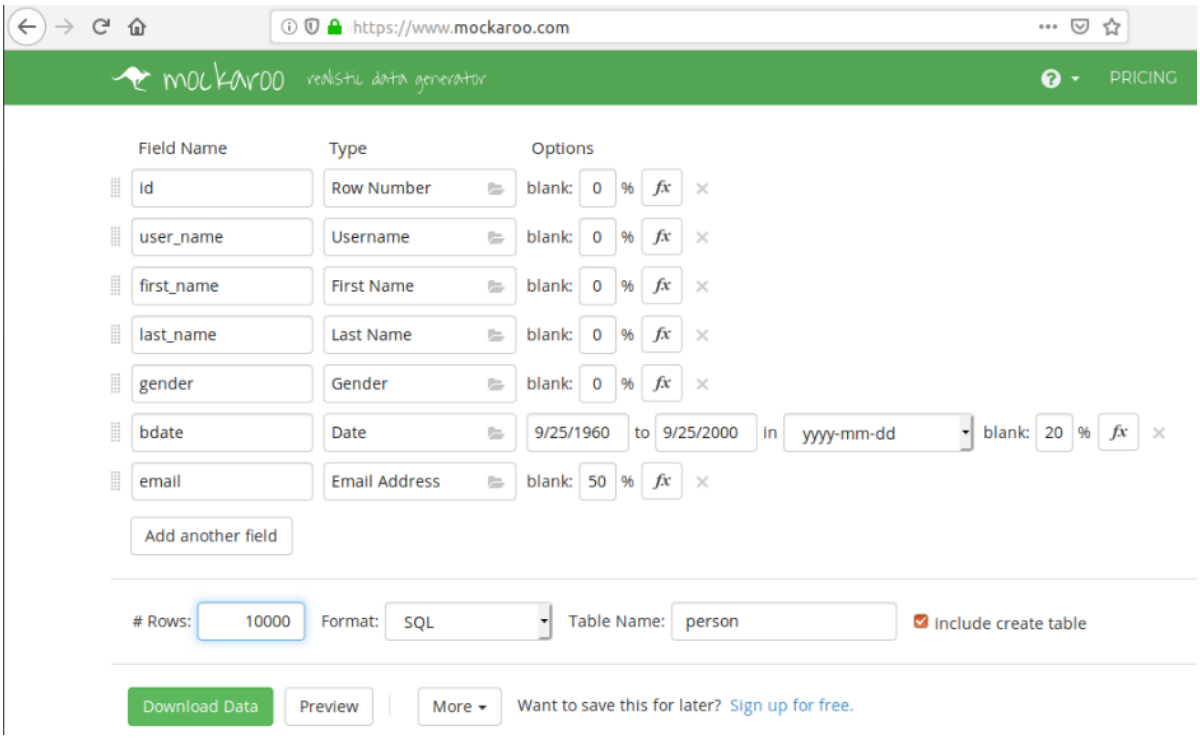

Рисунок 1.2 – Настройка генератора фиктивных данных

В качестве нового поля добавлено поле user_name. Установлен также флажок «Include create table». С использованием кнопки «Preview» можно просмотреть генерируемые данные и команды SQL. Необходимо загрузить файл *.sql для дальнейшего использования. Сгенерированный файл person.sql необходимо получить у преподавателя.

8. Откроем person.sql с использованием vscode (можно использовать любой другой редактор) и выполним редактирование команды SQL (рисунок 1.3).

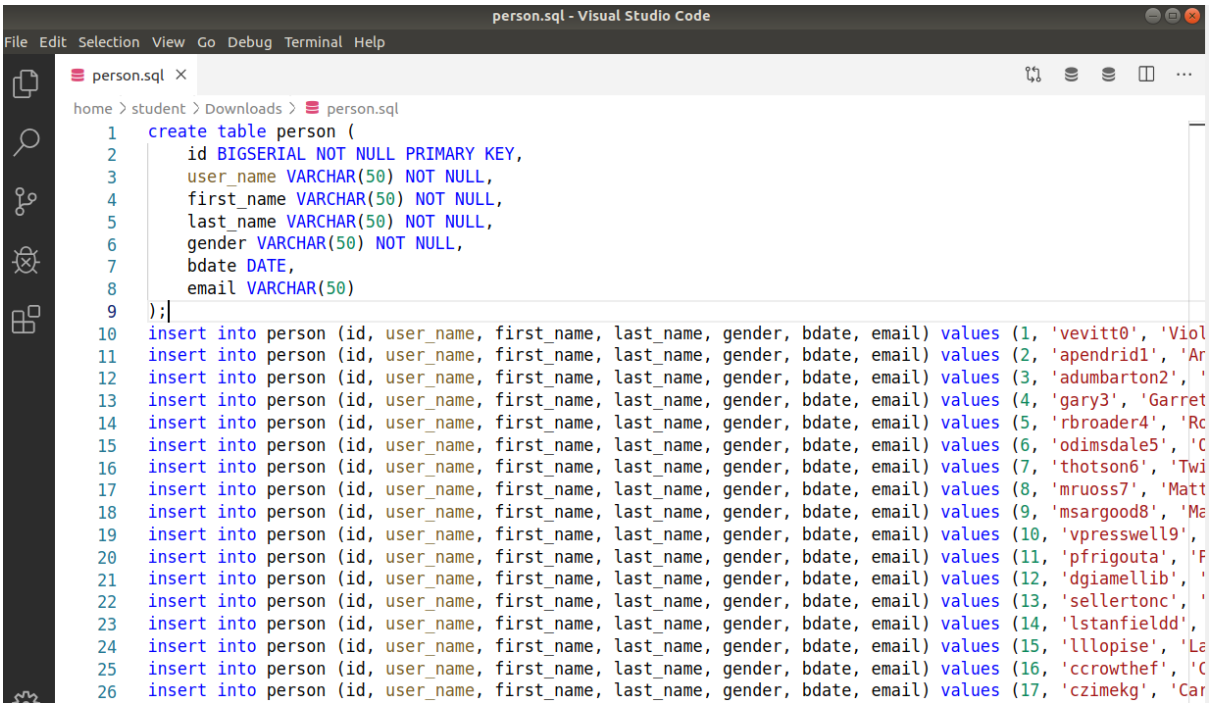

Рисунок 1.3 – Команды SQL для создания таблицы и вставки 1000 записей

После соответствующей правки файла person.sql можно копировать весь код и вставить его в консоль, но данный метод не является оптимальным. Во-первых перед вставкой записей необходимо удалить имеющуюся таблицу person командой:

drop table person;

После этого в консоли psql необходимо вызвать команду \i FILE. Команда выполняет инструкции SQL, содержащиеся в файле FILE. В данном случае необходимо выполнить следующую команду:

\i /home/student/Downloads/person.sql

9. Осуществим выборку первых десяти записей из таблицы person, предварительно упорядочив их по полю user_name:

select * from person order by user_name limit 10;

Результат выполнения команды показан на рисунке 1.4.

```
cloud=> select * from person order by user_name limit 10;
 id  |  user_name   | first_name | last_name | gender |   bdate    |           email
-----+--------------+------------+-----------+--------+------------+----------------------
 304 | aalasdair8f  | Archie     | Alasdair  | Male   | 1989-08-02 |
 422 | aallenbp     | Avictor    | Allen     | Male   |            |
 861 | aandressnw   | Anatollo   | Andress   | Male   |            |
 644 | abalmforthhv | Amil       | Balmforth | Female |            |
 612 | abarltropgz  | Alon       | Barltrop  | Male   | 1988-05-07 |
 632 | abatchanhj   | Ameline    | Batchan   | Female | 1972-04-07 | abatchanhj@blog.com
 263 | abeckworth7a | Abagael    | Beckworth | Female | 1979-03-15 | abeckworth7a@w3.org
 367 | abeltona6    | Ag         | Belton    | Female | 1984-04-02 |
 521 | abenmoreeg   | Aurilia    | Benmore   | Female | 1963-12-05 | abenmoreeg@telegraph.
 324 | abes8z       | Alice      | Bes       | Female | 1999-07-28 | abes8z@exblog.jp
(10 rows)

cloud=>
```

Рисунок 1.4 – Выборка записей

10. Выполните задание в соответствии с индивидуальным вариантом.

1.6.2. Выполнение индивидуального задания

Для выполнения индивидуального задания необходимо создать базу данных, пользователя с правами владельца; создать таблицу в базе данных и добавить в таблицу 5000 записей. Базы данных создаются в соответствии с вариантами, представленными в таблице 1.1. Таблица базы данных создается студентом по своему усмотрению на основании предложенной предметной области. Студент по согласованию с преподавателем может предложить свою тематику для создания базы данных.

Таблица 1.1 – Варианты индивидуальных заданий

№ варианта	Постановка задачи
1	Каталог потребительских товаров
2	Данные о пользователях
3	Финансовые счета клиентов
4	Коммерческие организации
5	База сайтов
6	База программного обеспечения
7	Каталог производственного оборудования
8	Автомобильная база
9	История пользовательской активности в сети
10	Каталог объектов недвижимости
11	База заемщиков

12	Органайзер
13	Учет отработанного времени
14	Планирование мероприятий
15	Каталог курсов
16	Биржевые торги
17	База абонентов
18	Транспортная система
19	Бизнес-партнеры
20	База MNIST
21	База «Ирисы Фишера»
22	Набор данных «Титаник»
23	Набор данных «Стартапы»

1.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

1.8. Контрольные вопросы

1. Перечислите последовательность действий по установке СУБД PostgreSQL в ОС Linux. Что такое psql?

2. Опишите процесс создания базы данных, включая создание специализированного пользователя-владельца добавляемой базы данных. Какой пользователь используется для подключения к СУБД по-умолчанию?

3. Опишите основные команды для создания и удаления базы данных. Назовите команду для просмотра баз данных, доступных на сервере. Что значит роль «владелец базы данных»? Какой командой необходимо делегировать права владельца конкретному пользователю?

4. Как создать таблицу в БД с использованием консоли psql? Какая команда удаляет таблицу?

5. Для чего предназначены команды \d, \dt, \d TABLE?

6. Приведите пример и поясните синтаксис команды вставки записи в таблицу.

7. Поясните механизм генерации фейковых данных для проекта на основе использования баз данных. С помощью каких сервисов можно реализовать получение фиктивных данных?

8. Поясните синтаксис и назначение команды \i FILE. Какую информацию должен содержать файл, задаваемый параметром FILE?

9. Приведите пример запроса для выборки всех записей из таблицы реляционной базы данных. Приведите пример выборки первых 5 записей при работе с таблицей PostgreSQL.

10. Используя базу данных cloud и данные таблицы person необходимо ответить на следующие вопросы:

10.1. Количество людей в таблице person для которых отсутствует информация о e-mail.

10.2. Количество людей для которых отсутствуют сведения о дате рождения.

10.3. Если упорядочить все записи в алфавитном порядке по полю user_name, то какой пользователь будет находиться на третьемместе с конца?

10.4. Сколько человек мужского поля в таблице person?

10.5. Если упорядочить все записи по алфавиту по полю last_name, то какая женщина будет находиться на первом месте?

1.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 2. ОСНОВНЫЕ КОМАНДЫ МАНИПУЛИРОВАНИЯ ДАННЫМИ

2.1. Цель и содержание

Цель лабораторной работы: Научиться манипулировать данными, содержащимися в таблицах реляционных СУБД (реализация основных CRUD-операций).

В рамках лабораторной работы необходимо решить следующие задачи:

– получение навыков выборки данных;

– получение навыков вставки записей в таблицу;

– получение навыков обновления данных;

– получение навыков удаления данных.

2.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

2.3. Теоретическая часть

Пусть в базе имеется таблица car, содержащая сведения о зарегистрированных автомобилях. Пусть требуется получить из базы информацию об автомобилях с тремя дверями, отсортировав ее по идентификатору автомобиля. Результат должен содержать не более 5 строк, поскольку пользовательский ин терфейс выводит данные страницами по пять записей. Запрос имеет вид:

```
SELECT car_id, registration_number, manufacture_year
FROM car_portal_app.car
WHERE number_of_doors=3
```

```
ORDER BY car_id
LIMIT 5;
```

Результат будет следующим:

```
car_id | registration_number | manufacture_year
--------+---------------------+-----------------
2 | VSVW4565 | 2014
5 | BXGK6290 | 2009
6 | ORIU9886 | 2007
7 | TGVF4726 | 2009
8 | JISW6779 | 2013
(5 rows)
```

2.3.1. Выборка данных

Упрощенная синтаксическая диаграмма команды SELECT выглядит следующим образом:

```
SELECT [DISTINCT | ALL] <expression>[[AS] <output_name>][, …]
[FROM <table>[, <table>… | <JOIN clause>…] [WHERE <condition>]
[GROUP BY <expression>|<output_name>|<output_number> [,…]]
[HAVING <condition>]
[ORDER BY <expression>|<output_name>|<output_number> [ASC | DESC] [NULLS
FIRST | LAST] [,…]]
[OFFSET <expression>] [LIMIT <expression>];
```

Опущены некоторые элементы, например фразы WINDOW , WITH и FOR UPDATE. Полную синтаксическую диаграмму см. на странице http://www.postgresql.org/docs/current/static/sql-select.html.

Логическая последовательность операций, выполняемых командой SELECT, такова:

1) выбрать все записи из всех исходных таблиц. Если во фразе FROM есть подзапросы, то они вычисляются первыми;

2) образовать все возможные комбинации этих записей и отбросить те из них, для которых не удовлетворяются условия JOIN , а в случае внешних соединений установить некоторые поля в таких комбинациях в NULL;

3) отфильтровать комбинации, не удовлетворяющие предикату во фразе WHERE;

4) построить группы в соответствии со значениями выражений в списке GROUP BY;

5) оставить только группы, удовлетворяющие условиям HAVING;

6) вычислить выражения в списке выборки;

7) исключить строки-дубликаты, если присутствует ключевое слово DISTINCT;

8) применить теоретико-множественные операции UNION, EXCEPT и INTERSECT;

9) отсортировать строки в соответствии с фразой ORDER BY;

10) оставить только записи, отвечающие условиям во фразах OFFSET и LIMIT.

На самом деле PostgreSQL оптимизирует этот алгоритм, выполняя шаги в другом порядке или даже одновременно. Например, если задана фраза LIMIT 1, то не имеет смысла выбирать все строки из исходных таблиц, достаточно только одной, удовлетворяющей условию WHERE.

Список выборки. После ключевого слова SELECT задается список столбцов или выражений, выбираемых из базы данных. Он называется списком выборки и определяет структуру результата: количества, имена и типы выбранных значений.

Каждому выражению в списке выборки сопоставляется имя. Если имя не задано пользователем, то сервер выбирает его автоматически, и в большинстве случаев оно отражает источник данных: имя столбца таблицы или имя функции. В остальных случаях имя имеет вид ?column? . Допустимо, а иногда даже желательно назвать выбранное выражение по-другому. Для этого предназначено ключевое слово AS , например:

SELECT name AS fio ...

Выражения в списке выборки называются однозначными, или скалярными выражениями, поскольку каждому выражению соответствует только одно значение (хотя оно может быть и массивом).

Иногда скалярные выражения называют SQL-выражениями или просто выражениями. У любого SQL-выражения есть тип данных, определяемый типом (или типами) входных данных. Во многих случаях тип данных можно явно изменить. Каждый элемент списка выборки становится столбцом выходного набора данных, имеющим тип соответствующего выражения.

SQL-выражения могут содержать:

– имена столбцов (в большинстве случаев);

– константы;

– операторы;

– скобки, задающие порядок вычислений;

– вызовы функций;

– агрегатные выражения;

– скалярные подзапросы;

– приведения типов;

– условные выражения.

2.3.2. Обновление данных

Команда UPDATE применяется для изменения данных в строках таблицы без изменения их количества. Вот ее синтаксис:

```
UPDATE <table_name>
SET <field_name> = <expression>[, ...]
[FROM <table_name> [JOIN clause]]
[WHERE <condition>];
```

Использовать команду UPDATE можно двумя способами. Первый похож на простую команду SELECT и называется подвыборкой. Второй служит для обновления данных в одной таблице, исходя из данных в других таблицах, и похож на команду SELECT из нескольких таблиц. В большинстве случаев нужного результата можно добиться любым из этих способов.

В PostgreSQL за одну операцию можно обновить только одну таблицу. В других базах данных при некоторых условиях допускается одновременное обновление нескольких таблиц.

UPDATE с подвыборкой. Выражение для задания нового значения является обычным SQL-выражением. В этом выражении допускается ссылка на обновляемое поле, и вместо него подставляется старое значение:

```
UPDATE t SET f = f+1 WHERE a = 5;
```

Часто в командах UPDATE употребляются подзапросы. Чтобы сослаться из подзапроса на обновляемую таблицу, ей необходимо сопоставить псевдоним:

```
car_portal=> UPDATE car_portal_app.a updated SET a_text =
(SELECT b_text FROM car_portal_app.b WHERE b_int = updated.a_int);
UPDATE 7
```

Если подзапрос не возвращает результата, то в поле будет записано NULL .Обратите внимание, что результатом команды UPDATE является слово UPDATE, за которым следует количество обновленных записей. Фраза WHERE аналогична используемой в команде SELECT . Если она не задана, то обновляются все записи.

UPDATE с дополнительными таблицами. Другой способ обновить строки таблицы – воспользоваться фразой FROM так же, как это делается в команде SELECT:

```
UPDATE car_portal_app.a SET a_int = b_int FROM car_portal_app.b
WHERE a.a_text=b.b_text;
```

Обновляются все строки a , для которых в b существуют строки с таким же значением в текстовом поле. Новое значение числового поля берется из таблицы b. Технически это не что иное, как внутреннее соединение двух таблиц. Однако синтаксис отличается. Поскольку таблица a не является частью фразы FROM, то обычный синтаксис соединения здесь неприменим, и таблицы соединяются по условию во фразе WHERE . Если бы использовалась

еще одна таблица, то ее можно было бы соединить с b , применив обычный синтаксис соединения, внутреннего или внешнего.

Во многих случаях синтаксис команды UPDATE с FROM выглядит более понятно. Например, следующая команда вносит в таблицу те же самые изменения, что предыдущая, но не так очевидна:

```
UPDATE car_portal_app.a
SET a_int = (SELECT b_int FROM car_portal_app.b WHERE a.a_text=b.b_text) WHERE
a_text IN (SELECT b_text FROM car_portal_app.b);
```

Еще одно преимущество формы с FROM состоит в том, что зачастую она работает гораздо быстрее. С другой стороны, можно получить непредсказуемые результаты, если одной записи обновляемой таблицы соответствует несколько записей из таблиц, указанных во фразе FROM:

```
UPDATE car_portal_app.a SET a_int = b_int FROM car_portal_app.b;
```

Этот запрос синтаксически корректен. Однако известно, что в таблице b несколько записей. Какая из них будет выбрана для обновления каждой строки, не определено, потому что условие WHERE отсутствует. Та же проблема имеет место, если фраза WHERE не определяет взаимно-однозначное соответствие.

```
car_portal=> UPDATE car_portal_app.a SET a_int = b_int FROM
car_portal_app.b WHERE b_int>=a_int;
UPDATE 6
```

Для каждой записи таблицы a существует более одной записи в таблице b, для которой b_int больше или равно a_int. Поэтому результат такого обновления не определен. Однако же PostgreSQL не запрещает такие команды, так что следует проявлять осторожность.

Команда обновления может вернуть обновленные записи, если присутствует фраза RETURNING, как в INSERT:

```
car_portal=> UPDATE car_portal_app.a SET a_int = 0 RETURNING *;
```

2.3.3. Добавление данных

Команда INSERT вставляет новые данные в таблицу. Строки всегда вставляются только в одну таблицу. Команда имеет следующий синтаксис:

INSERT INTO <table_name> [(<field_list>)]
{VALUES (<expression_list>)[,...]}|{DEFAULT VALUES}|<SELECT query>;

Имя таблицы, в которую вставляются записи, указывается после ключевых слов INSERT INTO . Существует два синтаксически различных варианта команды INSERT : для вставки одной или нескольких отдельных записей и для вставки целого набора записей.

Для вставки одной или нескольких записей используется ключевое слово VALUES , за которым следует список значений. Элементы списка соответствуют полям таблицы в порядке их следования. Если заполняются не все поля, то имена заполняемых полей должны быть указаны в скобках после имени таблицы. Те поля, что не указаны, получат значения по умолчанию, если таковые определены, или будут установлены в NULL.

Результатом успешно выполненной команды INSERT является слово INSERT , за которым следует OID вставленной строки (если вставлена только одна строка и для таблицы включены OID, в противном случае 0) и количество вставленных строк.

2.3.4. Удаление данных

Команда DELETE служит для удаления записей из базы. Как и в случае UPDATE, есть два способа удаления: с помощью подвыборки и с использованием других таблиц, одной или нескольких. Форма с подвыборкой имеет такой синтаксис:

DELETE FROM <table_name> [WHERE <condition>];

Записи, удовлетворяющие условию condition, будут удалены из таблицы. Если фраза WHERE опущена, то удаляются все записи. Форма DELETE с использованием другой таблицы аналогична форме UPDATE с фразой FROM . Только вместо FROM следует использовать ключевое слово USING, потому что FROM уже занято в другой части команды DELETE:

```
car_portal=> DELETE FROM car_portal_app.a USING car_portal_app.b
WHERE a.a_int=b.b_int;
DELETE 6
```

Эта команда удаляет из таблицы a все записи, для которых в таблице b существует запись с таким же значением числового поля. В результате выполнения команды сообщается, сколько записей было удалено.

Показанная выше команда эквивалентна такой:

```
DELETE FROM car_portal_app.a WHERE a_int IN
(SELECT b_int FROM car_portal_app.b);
```

Как и команды UPDATE и INSERT, команда DELETE возвращает удаленные строки, если присутствует ключевое слово RETURNING:

```
car_portal=> DELETE FROM car_portal_app.a RETURNING *;
```

Существует еще одна команда, которая изменяет данные, оставляя неизменной структуру таблицы, – TRUNCATE . Она очищает таблицу целиком и почти мгновенно. Эффект точно такой же, как при использовании команды DELETE без фразы WHERE. Особенно полезна она для больших таблиц.

```
car_portal=> TRUNCATE TABLE car_portal_app.a;
TRUNCATE TABLE
```

2.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

2.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно

производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

2.6. Методика и порядок выполнения работы

2.6.1. Постановка задачи

1. В соответствии с информацией, представленной в теоретической части лабораторной работы, реализуйте все команды CRUD.

2. Выполните индивидуальное задание.

2.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, создайте sql-скрипты для реализации следующих операций манипулирования данными:

1. Добавление данных (множества строк) в таблицу из внешнего файла.

2. Вставка одной записи в таблицу.

3. Удаление нескольких записей по критерию, задаваемому пользователем.

4. Удаление всей таблицы с записями.

5. Обновление записей с использованием вычислимых значений.

2.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

2.8. Контрольные вопросы

1. Какова логическая последовательность обработки запроса SELECT?

2. Опишите элементы, которые могут содержаться в SQL-выражениях.

3. Поясните принцип работы оператора UPDATE с подвыборкой.

4. Поясните принцип работы оператора UPDATE с дополнительной таблицей.

5. Опишите синтаксис оператора DELETE.

6. Опишите принцип работы и иснтаксис оператора TRUNCATE. ем данный оператор отличается от DELETE?

7. Опишите синтаксис оператора INSERT.

8. Опишите стратегию вставки нескольких записей в таблицу с использованием одного запроса INSERT.

2.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 3. ПРЕДСТАВЛЕНИЯ

3.1. Цель и содержание

Цель лабораторной работы: научиться использовать различные виды представлений в приложениях баз данных.

В рамках лабораторной работы необходимо решить следующие задачи:

– научиться использовать обычные представления;

– научиться использовать обновляемые представления;

– научиться использовать материализованные представления.

3.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

3.3. Теоретическая часть

3.3.1. Основы работы с представлениями

Представление можно считать именованным запросом или оберткой вокруг команды SELECT . Представления – существенный строительный блок реляционных баз данных с точки зрения UML-моделирования; его можно интерпретировать как метод UML-класса. Представления используются для следующих целей:

– чтобы упростить сложные запросы и повысить степень модульности кода;

– для повышения производительности посредством кеширования результатов и использования их в будущем;

– чтобы уменьшить объем SQL-кода;

– чтобы перебросить мост между реляционными базами данных и объектно-ориентированными языками (особенно в этом смысле полезны обновляемые представления);

– чтобы реализовать авторизацию на уровне строк – не давать доступа к строкам, не удовлетворяющим заданному условию;

– для реализации интерфейсов и уровня абстракции, расположенного между языками высокого уровня и реляционными базами;

– для реализации срочных изменений.

Представление должно отвечать текущим, а не предполагаемым будущим потребностям бизнеса. Его следует проектировать с учетом представления конкретной функциональности. Отметим, что чем больше в представлении атрибутов, тем больше усилий придется приложить для его рефакторинга. Кроме того, если представление агрегирует данные из многих таблиц и используется в качестве интерфейса, то возможно снижение производительности. Причин тому много, например неоптимальный план выполнения из-за устаревшей статистики некоторых таблиц и т. д.

Если сложная бизнес-логика реализуется в базе данных с помощью представлений и хранимых процедур, то рефакторинг базы, а в особенности базовых таблиц, может превратиться в кошмар. Чтобы избежать этого, подумайте о переносе бизнес-логики на уровень приложения.

У некоторых систем, например средств объектно-реляционного отображения, могут быть специальные требования, допустим наличие уникального ключа. Это ограничивает применимость в них представлений, однако в какой-то мере проблему можно сгладить, подменив первичные ключи оконными функциями, например row_number.

В PostgreSQL представление за кулисами моделируется как таблица с правилом _RETURN . То есть теоретически можно создать таблицу и преобразовать ее в представление, но делать так не рекомендуется. Дерево зависимостей представления строго контролируется, т. е. невозможно удалить

или структурно изменить представление, от которого зависят другие представления:

```
postgres=# CREATE VIEW test AS SELECT 1 as v;
CREATE VIEW
postgres=# CREATE VIEW test2 AS SELECT v FROM test;
CREATE VIEW
postgres=# CREATE OR REPLACE VIEW test AS SELECT 1 as val;
ERROR: cannot change name of view column "v" to "val"
```

Показанная ниже команда CREATE VIEW позволяет создать представление, а при наличии ключевого слова REPLACE заменить уже существующее представление. Имена атрибутов представления можно задать явно, в противном случае они наследуются от команды SELECT

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ (
column_name [, ...] ) ]
[ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
AS query
[ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

В примере ниже показано, как создать представление, которое выводит всю информацию о пользователе, кроме пароля. Это может быть полезно, чтобы ограничить доступ приложения к паролю. Отметим, что имена столбцов наследуются от столбцов в команде SELECT , как показывает метакоманда \d для объекта account_information

```
car_portal=> CREATE VIEW account_information AS SELECT account_id,
first_name, last_name, email FROM account;
CREATE VIEW
car_portal=> \d account_information
View "car_portal_app.account_information"
Column | Type | Collation | Nullable | Default
-----------+---------+-----------+----------+---------
account_id | integer | | |
first_name | text | | |
last_name | text | | |
email | text | | |
```

Имена столбцов представления можно задать и явно, как показано в примере ниже:

```
CREATE OR REPLACE VIEW account_information
(account_id,first_name,last_name,email) AS SELECT account_id, first_name, last_name,
email FROM account;
```

Если определение представления изменяется с помощью ключевого слова REPLACE , то списки столбцов в старом и новом представлениях должны быть одинаковы, включая имя, тип и порядок следования. В следующем примере показано, что произойдет при попытке изменить порядок столбцов:

```
car_portal=> CREATE OR REPLACE VIEW account_information AS SELECT account_id,
last_name, first_name, email FROM account;
ERROR: cannot change name of view column "first_name" to "last_name"
```

3.3.2. Категории представлений

В PostgreSQL имеется несколько категорий представлений.

1. Временные представления. Такое представление автоматически удаляется в конце сеанса. Если ключевое слово TEMPORARY или TEMP отсутствует, то жизненный цикл представления начинается в момент создания и заканчивается в момент явного удаления.

2. Рекурсивные представления. Рекурсивное представление напоминает рекурсивную функцию в языках программирования. Список столбцов в этом случае обязателен. Рекурсия, в частности рекурсивные представления и рекурсивные общие табличные выражения (CTE), позволяет строить очень сложные запросы, особенно для иерархически организованных данных.

3. Обновляемые представления. Обновляемые представления позволяют обращаться с представлениями, как с таблицами, т. е. выполнять команды INSERT, UPDATE и DELETE . Обновляемые представления в какой-то мере можно рассматривать как мост между реляционной и объектной моделями, они дают некий аналог полиморфизма.

4. Материализованные представления. Это, по сути дела, таблица, содержимое которой периодически обновляется заранее заданным запросом.

Материализованные представления повышают производительность запросов, которые выполняются долго, и часто применяются к статическим данным. Можно считать их вариантом кеширования. Рекурсия будет рассмотрена в последующих главах, а здесь мы сосредоточимся на обновляемых и материализованных представлениях.

3.3.3. Материализованные представления

Синтаксис определения материализованных и обычных представлений несколько различается. Материализованные представления – расширение PostgreSQL, но их поддерживают и другие СУБД, например Oracle. Как показано ниже, материализованное представление можно создать в определенном табличном пространстве, и для него можно задать параметром хранения storage_parameter , что вполне логично, т. к. материализованные представления – физические объекты:

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
[ (column_name [, ...] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ TABLESPACE tablespace_name ]
AS query
[ WITH [ NO ] DATA ]
```

В момент создания материализованного представления его можно заполнить или оставить пустым. Для заполнения пустого материализованного представления служит команда REFRESH MATERIALIZED VIEW с таким синтаксисом:

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name [ WITH [ NO ] DATA ]
```

Попытка выбрать данные из незаполненного представления заканчивается ошибкой, показанной в примере ниже:

```
car_portal=> CREATE MATERIALIZED VIEW test_mat AS SELECT 1 WITH NO DATA;
CREATE MATERIALIZED VIEW
car_portal=> TABLE test_mat;
ERROR: materialized view "test_mat" has not been populated
HINT: Use the REFRESH MATERIALIZED VIEW command.
```

Обновить представление, как предлагается в подсказке, можно следующим образом:

```
ar_portal=> REFRESH MATERIALIZED VIEW test_mat;
REFRESH MATERIALIZED VIEW
car_portal=> TABLE test_mat;
?column?
----------
1
(1 row)
```

Обновление материализованного представления – блокирующая команда, т. е. одновременно выполняемые команды SELECT будут приостановлены на время, пока производится обновление. Эту проблему можно решить с помощью параллельного обновления, но тогда над представлением должен быть построен уникальный индекс.

Материализованные представления часто используются совместно с хранилищами данных. В этом случае выполняется ряд запросов в интересах бизнес-аналитики и поддержки принятия решений. Данные в таких приложениях изменяются редко, но вычисления и агрегирование занимают много времени.

В общем случае материализованные представления применяются для следующих целей:

– формирование сводных отчетов;

– кеширование результатов повторяющихся запросов;

– повышение производительности благодаря однократной обработке данных.

Поскольку материализованные представления – это таблицы, их можно индексировать, что резко ускоряет работу с ними.

3.3.4. Обновляемые представления

По умолчанию простые представления в PostgreSQL являются автообновляемыми, т. е. к ним можно применять команды DELETE, INSERT и UPDATE , которые воздействуют на данные в базовой таблице. Если представление не является обновляемым (а значит, и простым) из-за нарушения одного из перечисленных ниже ограничений, то его все-таки можно сделать таковым с помощью триггеров и правил. Представление является автоматически обновляемым, если выполнены следующие условия:

– представление должно быть построено только над одной таблицей или одним обновляемым представлением;

– определение представления не содержит на верхнем уровне следующих фраз и теоретико-множественных операторов: DISTINCT, WITH, GROUP BY, OFFSET, HAVING, LIMIT, UNION, EXCEPT, INTERSECT;

– в списке select должны быть только сами столбцы базовой таблицы, использование функций и выражений не допускается. Кроме того, столбцы не должны повторяться;

– не должно быть установлено свойство security_barrier.

Для сайта торговли автомобилями можно определить обновляемое представление, показывающее только учетные записи, не принадлежащие продавцам:

```
CREATE VIEW user_account AS
SELECT account_id, first_name, last_name, email, password
FROM account WHERE account_id NOT IN (SELECT account_id FROM seller_account);
```

Для проверки попробуем вставить в него строку:

```
car_portal=> INSERT INTO user_account VALUES
(default,'first_name1','last_name1','test@email.com','password');
INSERT 0 1
```

Для автообновляемого представления нельзя изменить строку, которую представление не возвращает. Попробуем вставить учетную запись вместе с записью продавца, а затем удалить ее.

```
car_portal=> WITH account_info AS ( INSERT INTO user_account VALUES
(default,'first_name2','last_name2','test2@email.com','password') RETURNING
account_id)

INSERT INTO seller_account (account_id, street_name, street_number, zip_code, city)
SELECT account_id, 'street1', '555', '555', 'test_city' FROM account_info;

INSERT 0 1
```

Обратите внимание, что вставка в представление user_account прошла успешно. Но тем не менее удалить эту запись с помощью обновляемого представления не получится – помешает проверочное ограничение:

```
car_portal=> DELETE FROM user_account WHERE first_name = 'first_name2';
DELETE 0
car_portal=> SELECT * FROM account where first_name like 'first_name%';
account_id | first_name | last_name | email | password
-----------+------------+-----------+----------------+----------
482 | first_name1 | last_name1 | test@email.com | password
484 | first_name2 | last_name2 | test2@email.com | password
(2 rows)
```

Для управления поведением автоматически обновляемых представлений служит фраза WITH CHECK OPTION. Если она отсутствует, то команды UPDATE и INSERT успешно выполняются, даже если строка не видна в представлении, что рискованно с точки зрения безопасности.

3.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

3.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно

производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

3.6. Методика и порядок выполнения работы

3.6.1. Постановка задачи

1. В соответствии с информацией, представленной в теоретической части лабораторной работы, реализуйте несколько видов (не менее двух) представлений в базе данных.

2. Выполните индивидуальное задание.

3.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, создайте sql-скрипты для реализации следующих операций манипулирования данными:

1. Создайте материализованное представление (более двух).

2. Создайте обновляемое представление (более двух).

3. Продемонстрируйте использование представлений.

3.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

3.8. Контрольные вопросы

1. Напишите команду создания представления. Поясните основные параметры команды.

2. Что такое обновляемые представления? Чем они отличаются от обычных?

3. Опишите команду сохдания материализованнооого представления и ее параметры.

4. Чем материализованное представление отличается от обновляемого?

5. Перечислите категории представлений. Опишите каждую категорию.

3.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 4. ИНДЕКСЫ

4.1. Цель и содержание

Цель лабораторной работы: научиться применять индексы для повышения производительности баз данных.

В рамках лабораторной работы необходимо решить следующие задачи:

– научиться создавать различные категории индексов;

– получить навыки использования различных типов индексов;

– получить навыки по анализу планов выполнения запросов.

4.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

4.3. Теоретическая часть

Индекс – это физический объект базы данных, построенный над одним или несколькими столбцами таблицы. В PostgreSQL есть несколько типов индексов и, соответственно, способов их использования. Индексы применяются для решения следующих задач:

1. Оптимизация производительности – индекс позволяет эффективно выбирать из таблицы небольшое число строк. Что такое «небольшое» число, определяется общим количеством строк в таблице и параметрами плана выполнения;

2. Контроль ограничений – индекс позволяет проверять заданные для строк ограничения. Например, для проверки ограничения UNIQUE автоматически создается индекс по соответствующему столбцу.

В следующем примере показано, как использовать GIST-индекс для запрета перекрывающихся диапазонов дат.

```
CREATE TABLE no_date_overlap
( date_range daterange,
EXCLUDE USING GIST (date_range WITH &&)
);
```

Проверим, попытавшись создать перекрывающиеся диапазоны:

```
car_portal=# INSERT INTO no_date_overlap values('[2010-01-01, 2020-01-01)');
INSERT 0 1
car_portal=# INSERT INTO no_date_overlap values('[2010-01-01, 2017-01-01)');
ERROR: conflicting key value violates exclusion constraint
"no_date_overlap_date_range_excl"
DETAIL: Key (date_range)=([2010-01-01,2017-01-01)) conflicts with existing
key (date_range)=([2010-01-01,2020-01-01)).
```

4.3.1. Синтаксис создания индекса

Индексы создаются командой CREATE INDEX. Поскольку индекс – физический объект базы данных, мы можем указать табличное пространство и параметр хранения storage_parameter. Индекс можно строить по столбцам или по выражениям. Элементы индекса можно сортировать в порядке возрастания (ASC) или убывания (DESC). Кроме того, можно задать порядок сортировки для значений NULL – в начале или в конце индекса. Если индекс создается по текстовым полям, то можно также задать порядок сравнения (collation). Ниже приведен полный синтаксис команды.

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
table_name [ USING method ]
( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC
| DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
[ WITH ( storage_parameter = value [, ... ] ) ]
[ TABLESPACE tablespace_name ]
[ WHERE predicate ]
```

Для первичных и уникальных ключей индексы создаются автоматически.

4.3.2. Избирательность индекса

Рассмотрим таблицу account_history в базе данных для сайта торговли автомобилями. У ограничения уникальности UNIQUE (account_id, search_key, search_date) есть две цели. Первая – проверка того, что ни один ключ поиска не вставляется дважды для одной и той же даты, даже если пользователь искал по нему несколько раз. Вторая – быстрая выборка данных. Пусть требуется вывести последние 10 поисков данного пользователя. Запрос может выглядеть так:

```
SELECT search_key FROM account_history WHERE account_id = <account>
GROUP BY search_key ORDER BY max(search_date) limit 10;
```

Этот запрос возвращает 10 записей, содержащих ключи поиска search_key, упорядоченные по дате поиска search_date. Если таблица account_history содержит миллионы строк, то чтение всех данных займет очень много времени. Ноне в этом случае, поскольку наличие уникального индекса позволяет читать данные только для одного указанного пользователя.

Если таблица мала, то индексы над ней не используются. В этом случае планировщик PostgreSQL предпочитает просматривать таблицу целиком. Чтобы убедиться в этом, заполним таблицу account_history совсем небольшим набором данных:

```
WITH test_account AS (
INSERT INTO account VALUES (1000, 'test_first_name', 'test_last_name',
'test@email.com', 'password')
RETURNING account_id
),car AS ( SELECT i as car_model FROM (VALUES('brand=BMW'), ('brand=WV'))
AS foo(i)
),manufacturing_date AS ( SELECT 'year='|| i as date FROM generate_series
(2015, 2014, -1) as foo(i))
INSERT INTO account_history (account_id, search_key, search_date)
SELECT account_id, car.car_model||'&'||manufacturing_date.date, current_date
FROM test_account, car, manufacturing_date;
VACUUM ANALYZE;
```

Чтобы узнать, используется ли индекс, выполним такой запрос:

```
car_portal=> SELECT search_key FROM account_history WHERE account_id = 1000
GROUP BY search_key ORDER BY max(search_date) limit 10;
search_key
--------------------
brand=WV&year=2014
brand=BMW&year=2014
brand=WV&year=2015
brand=BMW&year=2015
(4 rows)

car_portal=> EXPLAIN SELECT search_key FROM account_history WHERE
account_id = 1000 GROUP BY search_key ORDER BY max(search_date) limit 10;
QUERY PLAN
------------------------------------------------------------------------------
Limit (cost=1.17..1.18 rows=3 width=23)
-> Sort (cost=1.17..1.18 rows=3 width=23)
Sort Key: (max(search_date))
-> HashAggregate (cost=1.12..1.15 rows=3 width=23)
Group Key: search_key
-> Seq Scan on account_history (cost=0.00..1.10 rows=4 width=23)
Filter: (account_id = 1000)
```

В этом примере индекс не используется. Планировщик решает, применять индекс или нет, сообразуясь со стоимостью плана выполнения. Для одного и того же запроса, но с разными параметрами планировщик может выбрать разные планы, ориентируясь на гистограмму распределения данных. Даже если набор данных велик, но предикат, заданный в условии, отфильтровывает мало данных, индекс использоваться не будет. Чтобы продемонстрировать такую ситуацию, добавим строки еще для одной учетной записи и выполним запрос снова:

```
WITH test_account AS (
INSERT INTO account VALUES (2000, 'test_first_name',
'test_last_name','test2@email.com', 'password') RETURNING account_id
),car AS ( SELECT i as car_model FROM (VALUES('brand=BMW'), ('brand=WV'),
```

```
('brand=Audi'), ('brand=MB')) AS foo(i)
),manufacturing_date AS ( SELECT 'year='|| i as date FROM generate_series
(2017, 1900, -1) as foo(i))
INSERT INTO account_history (account_id, search_key, search_date) SELECT
account_id, car.car_model||'&'||manufacturing_date.date, current_date
FROM test_account, car, manufacturing_date;
VACUUM ANALYZE;
```

Выполняем запрос для второй учетной записи:

```
car_portal=> EXPLAIN SELECT search_key FROM account_history WHERE
account_id = 2000 GROUP BY search_key ORDER BY max(search_date) limit 10;
QUERY PLAN
------------------------------------------------------------------------------
Limit (cost=27.10..27.13 rows=10 width=23)
-> Sort (cost=27.10..28.27 rows=468 width=23)
Sort Key: (max(search_date))
-> HashAggregate (cost=12.31..16.99 rows=468 width=23)
Group Key: search_key
-> Seq Scan on account_history (cost=0.00..9.95 rows=472 width=23)
Filter: (account_id = 2000)
(7 rows)
```

Индекс и на этот раз не используется, потому что выгоднее прочитать всю таблицу. Избирательность индекса очень низкая, т. к. для учетной записи 2000 имеется 427 строк, а для учетной записи 1000 – только четыре строки:

```
car_portal=> SELECT count(*), account_id FROM account_history group by account_id;
count | account_id
-------+------------
472 | 2000
4 | 1000
(2 rows)
```

Наконец, выполним тот же запрос для учетной записи 1000 . В этом случае избирательность высокая, поэтому индекс используется:

```
EXPLAIN SELECT search_key FROM account_history WHERE account_id = 1000
GROUP BY search_key ORDER BY max(search_date) limit 10;
```

```
QUERY PLAN

Limit (cost=8.71..8.72 rows=4 width=23)
-> Sort (cost=8.71..8.72 rows=4 width=23)
Sort Key: (max(search_date))
-> GroupAggregate (cost=8.60..8.67 rows=4 width=23)
Group Key: search_key
-> Sort (cost=8.60..8.61 rows=4 width=23)
Sort Key: search_key
-> Bitmap Heap Scan on account_history
(cost=4.30..8.56 rows=4 width=23)
Recheck Cond: (account_id = 1000)
-> Bitmap Index Scan on
account_history_account_id_search_key_search_date_key (cost=0.00..4.30
rows=4 width=0)
Index Cond: (account_id = 1000)
(11 rows)
```

4.3.3. Типы индексов

PostgreSQL поддерживает индексы разных типов, каждый из них применяется в определенных ситуациях.

1. B-дерево (B-tree). Это тип индекса по умолчанию, он выбирается, когда в команде CREATE INDEX тип не указан. Буква B означает balanced (сбалансированное), т. е. по разные стороны от каждого промежуточного узла дерева находится примерно одинаковое количество узлов. B-деревья эффективны для поиска по условию равенства, принадлежности диапазону и совпадения с null. Индекс типа B-tree можно строить для любых типов данных PostgreSQL.

2. Хеш-индекс. До версии PostgreSQL 10 поддержка хеш-индексов была неполной. Не гарантировалась транзакционная безопасность, и не поддерживалась потоковая репликация на ведомые узлы. В PostgreSQL 10 эти ограничения сняты. Хеш-индексы полезны для поиска по условию равенства.

3. Обобщенный обратный индекс (GIN). GIN-индекс полезен, когда несколько значений нужно отобразить на одну строку. Он используется со

сложными структурами данных, например массивами, и для полнотекстового поиска.

4. Обобщенное дерево поиска (GiST). GiST-индексы позволяют строить обобщенные сбалансированные древовидные структуры. Они полезны для индексации геометрических типов данных, а также для полнотекстового поиска.

5. GiST с двоичным разбиением пространства (SP-GiST). Аналогичны GIST-индексам и поддерживают деревья поиска по двоичному разбиению пространства. Вообще говоря, индексы типа GIST, GIN и SP-GIST предназначены для работы со сложными пользовательскими типами данных.

6. Блочно-диапазонный индекс (BRIN). Этот тип появился в версии PostgreSQL 9.5. BRIN-индекс полезен для очень больших таблиц, когда место на диске ограничено. Он медленнее B-деревьев, но занимает меньше места.

4.3.4. Категории индексов

Индексы можно классифицировать следующим образом:

1. Частичный индекс – индексируется только подмножество таблицы, удовлетворяющее заданному предикату; в определении индекса присутствует фраза WHERE . Идея в том, чтобы уменьшить размер индекса, сделав его более быстрым и удобным для обслуживания.

2. Уникальный индекс – гарантирует, что каждое значение встречается только один раз. В таблице account со столбцом email ассоциировано ограничение уникальности. Оно реализуется уникальным индексом, о чем свидетельствует метакоманда \d.

```
\d account
Table "car_portal_app.account"
Column | Type | Collation | Nullable | Default
------------+---------+-----------+----------+--------------------
----------------------
account_id | integer | | not null |
```

```
nextval('account_account_id_seq'::regclass)
first_name | text | | not null |
last_name | text | | not null |
email | text | | not null |
password | text | | not null |
Indexes:
"account_pkey" PRIMARY KEY, btree (account_id)
"account_email_key" UNIQUE CONSTRAINT, btree (email)
```

3. Индекс по нескольким столбцам применяется для поддержки запросов определенного вида. Рассмотрим запрос:

```
SELECT * FROM table WHERE column1 = constant1 and column2 = constant2 AND ...
columnn = constantn;
```

В этом случае можно создать индекс по столбцам column1 , column2, …, columnn, если n меньше или равно 32;

4. Индекс по выражению можно строить не только по нескольким столбцам, но и по выражениям, включающим вызовы функций.

Индекс может относиться сразу к нескольким категориям, например возможен уникальный частичный индекс. Это позволяет гибко приспосабливаться к бизнес-требованиям или добиваться максимального быстродействия. Так, в следующих главах мы воспользуемся индексом по результатам функции lower() или upper()

```
car_portal=> CREATE index on account(lower(first_name));
CREATE INDEX
```

Такой индекс позволяет искать учетную запись по имени владельца без учета регистра:

```
SELECT * FROM account WHERE lower(first_name) = lower('foo');
```

Индекс по выражению используется, только если выражение во фразе WHERE В ТОЧНОСТИ совпадает с выражением, по которому строился индекс.

Еще одно применение индекса по выражению – фильтрация строк после приведения к другому типу данных. Например, время вылета можно хранить в виде timestamp, но ищем мы часто по дате, а не по времени.

Как уже было сказано, индекс может быть одновременно уникальным и частичным. Предположим, что имеется таблица employee, в которой у каждого работника, за исключением главы компании, имеется начальник. Это можно смоделировать с помощью самоссылающейся таблицы:

```
REATE TABLE employee (employee_id INT PRIMARY KEY, supervisor_id INT);
ALTER TABLE employee ADD CONSTRAINT supervisor_id_fkey FOREIGN KEY
(supervisor_id) REFERENCES employee(employee_id);
```

Для гарантии того, что существует только одна строка без начальника, можно добавить такой уникальный индекс:

```
CREATE UNIQUE INDEX ON employee ((1)) WHERE supervisor_id IS NULL;
```

Уникальный индекс по константному выражению (1) допускает только одну строку со значением null. При вставке первой такой строки будет построен индекс с ключом 1. Вторая попытка вставить строку со значением null приведет к ошибке, потому что ключ 1 уже есть:

```
car_portal=> INSERT INTO employee VALUES (1, NULL);
INSERT 0 1
car_portal=> INSERT INTO employee VALUES (2, 1);
INSERT 0 1
car_portal=> INSERT INTO employee VALUES (3, NULL);
ERROR: duplicate key value violates unique constraint "employee_expr_idx"
DETAIL: Key ((1))=(1) already exists.
```

В настоящее время по нескольким столбцам можно строить только индексы типов B-tree, GIN , GIST и BRIN . При создании многостолбцового индекса порядок столбцов важен. Поскольку многостолбцовый индекс обычно большой, планировщик может предпочесть последовательный просмотр таблицы, а не поиск по индексу.

4.3.5. Рекомендации по работе с индексами

Часто бывает полезно индексировать столбцы, встречающиеся в предикатах и внешних ключах. Это дает PostgreSQL возможность не просматривать таблицу последовательно, а искать по индексу. Индексы дают выигрыш не только при выполнении SELECT, но также DELETE и UPDATE. Есть несколько случаев, когда индекс не используется; чаще всего это происходит, когда таблица мала.

Для больших таблиц надо тщательно планировать емкость системы хранения, потому что размеры индексов могут быть очень велики. Отметим также, что наличие индексов снижает скорость вставки, потому что требуется обновлять индексы, а на это тоже уходит время.

Есть несколько каталожных таблиц и функций, полезных для обслужива ния индексов, например таблица pg_stat_all_indexes, в которой хранится статистика использования индексов.

При создании индексов проверяйте, что индекса по тем же столбцам (или выражениям) еще не существует, иначе могут появиться дублирующие индексы. PostgreSQL в этом случае не выдает предупреждений:

```
car_portal=# CREATE index on car_portal_app.account(first_name);
CREATE INDEX
car_portal=# CREATE index on car_portal_app.account(first_name);
CREATE INDEX
```

Редко, но бывает, что индекс чрезмерно разрастается. Для перестраивания индекса PostgreSQL предоставляет команду REINDEX. Отметим, что REINDEX – блокирующая команда. Чтобы обойти эту проблему, можно параллельно (без блокировки) создать индекс, идентичный исходному, а затем удалить исходный.

Параллельное создание индекса – предпочтительное решение в активных системах, но оно требует больше ресурсов, чем обычное индексирование.

К тому же при параллельном индексировании можно столкнуться с подводными камнями; иногда создание индексов завершается неудачно, и тогда остается некорректный индекс. Его можно удалить или перестроить, соответственно, командами DROP и REINDEX. Ниже приведена команда (блокирующая) перестраивания индекса account_history_account_id_search_key_search_date_key:

```
car_portal=# REINDEX index
car_portal_app.account_history_account_id_search_key_search_date_key;
REINDEX
```

Другой, неблокирующий способ параллельного создания индекса выглядит так:

```
car_portal=# CREATE UNIQUE INDEX CONCURRENTLY ON
car_portal_app.account_history(account_id, search_key, search_date);
CREATE INDEX
```

Наконец, нужно удалить старый индекс. В данном случае мы не можем воспользоваться командой DROP INDEX, потому что индекс был создан неявно для поддержки ограничения уникальности. Чтобы избавиться от него, нужно удалить ограничение:

```
car_portal=# ALTER TABLE car_portal_app.account_history DROP CONSTRAINT
account_history_account_id_search_key_search_date_key;
ALTER TABLE
```

А затем снова добавить его:

```
car_portal=> ALTER TABLE account_history ADD CONSTRAINT
account_history_account_id_search_key_search_date_key UNIQUE USING INDEX
account_history_account_id_search_key_search_date_idx;
NOTICE: ALTER TABLE / ADD CONSTRAINT USING INDEX will rename index
"account_history_account_id_search_key_search_date_idx" to
"account_history_account_id_search_key_search_date_key"
ALTER TABLE
```

4.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС

Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

4.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

4.6. Методика и порядок выполнения работы

4.6.1. Постановка задачи

1. В соответствии с информацией, представленной в теоретической части лабораторной работы, реализуйте несколько видов (не менее двух) индексов в базе данных.

2. Выполните индивидуальное задание.

4.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, создайте sql-скрипты для добавления индексов в базу данных и демонстрации планов выполнения при использовании индексов:

1. Создайте индексы различных типов (более трех).

2. Создайте индексы различных категорий.

3. Продемонстрируйте, каким образом задействуются индексы при выполнении запросов. Продемонстрируйте планы выполнения запросов,

демонстрирующие повышение/ уменьшение скорости выполнения запроса при применении индексов.

4.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

4.8. Контрольные вопросы

1. Опишите назначение и особенности различных категорий индексов.
2. Опишите назначение и особенности различных типов индексов
3. Что понимается под избирательностью индексации?
4. Опишите основные рекомендации при работе с индексами.
5. Напишите запрос, демонстрирующий создание .
6. Опишите принципы функционирования B-tree-индексов.
7. Опишите принципы функционирования хеш-индексов.
8. Опишите принципы функционирования GIN-индексов.
9. Опишите принципы функционирования GiST-индексов.
10. Опишите принципы функционирования SP-GiST-индексов.

11. Опишите принципы функционирования и назначение BRIN-индексов.

4.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 5. КАРКАС ПРИЛОЖЕНИЯ С ИСПОЛЬЗОВАНИЕМ PYTHON

5.1. Цель и содержание

Цель лабораторной работы: научиться разрабатывать приложения баз данных с использованием языка Python.

В рамках лабораторной работы необходимо решить следующие задачи:

– использовать библиотеки Python для работы с СУБД;

– научиться выполнять простейшие запросы к БД;

– изучить принципы взаимодействия языков программирования и СУБД.

5.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

5.3. Теоретическая часть

В Python доступен сервисный модуль для подключения и работы с PostgreSQL, их несколько: Psycopg2, pg8000, py-postgresql, PyGreSQL, ocpgdb, bpgsql, SQLAlchemy. Для SQLAlchemy необходимо, чтобы все вышеперечисленное было установлено отдельно.

Прежде всего, интерфейсы или модули соответствуют спецификации API базы данных Python Database API v2.0 (PEP 249). Этот API был разработан для поддержания схожести между модулями Python, которые используются для доступа к базам данных. Другими словами, синтаксис, метод и способ доступа к базе данных одинаковы во всех модулях.

В рамках данного курса предлагается использование Psycopg2. Psycopg2 – самый популярный питонный драйвер для PostgreSQL. Он необходим для большинства программ Python и Postgres. Активно поддерживается и поддерживается основная версия питона, т.е. питон 3 и питон 2.

Psycopg2 является потоко-безопасным и разработан для многопоточных приложений с большой нагрузкой.

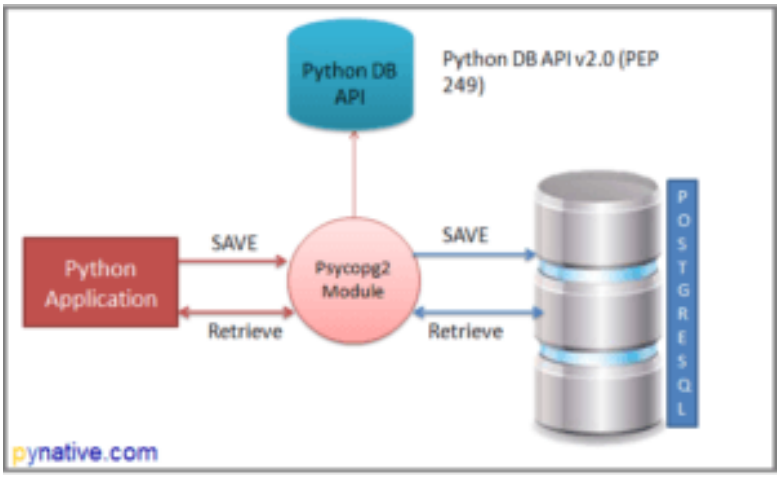

Рисунок 5.1 – Схема взаимодействия Python-приложения и СУБД через Psycopg2

5.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

5.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей

необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

5.6. Методика и порядок выполнения работы

5.6.1. Решение учебной задачи

Постановка задачи. Установите, настройте и протестируйте работоспособность Psycopg2. Создайте простейшее приложение Python для работы с СУБД.

Для решении задачи необходимо выполнить следующее:

1. Используя команду pip, вы можете установить Psycopg2 в любой операционной системе, включая Windows, MacO, Linux, Unix и Ubuntu. Используйте следующую команду pip для установки Psycopg2.

```
pip install psycopg2
```

Вы должны получить следующие сообщения после выполнения вышеуказанной команды.

```
Collecting psycopg2
Downloading psycopg2-2.7.5
Installing collected packages: psycopg2
Successfully installed psycopg2-2.7.5
```

Для подключения к СУБД разработчику необходимообладать следующей информацией:

1. Имя пользователя, которое вы используете для работы с PostgreSQL. Имя пользователя по умолчанию для базы данных PostgreSQL – postgres.

2. Пароль – пароль предоставляется пользователем во время установки PostgreSQL.

3. Имя хоста – это имя сервера или IP-адрес, на котором работает PostgreSQL. если вы работаете на localhost, то вы можете использовать localhost или его IP, т.е. 127.0.0.0.

4. Имя базы данных – имя базы данных, к которой вы хотите подключиться. Здесь мы используем базу данных с именем cloud.

Для подключения к СУБД с использованием приложения Python необходимо выполнить следующие щаги:

1. Используя метод connect() библиотеки psycopg2 с установленными параметрами установить соединение с PostgreSQL.

2. Для получения данных из БД необходимо создать курсор. Курсор создается с использованием ранее созданного соединения..

3. По завершении работы необходимо закрыть как курсор, таки объект соединения.

4. В процессе работы с СУБД средствами Ptrhon может возникнуть необходимость обработки исключений.

На рисунке 5.2 представлен образец кода, демонстрирующий все стадии:

```
import psycopg2

try:
   connection = psycopg2.connect(user = "puser",
                                 password = " puserxxx ",
                                 host = "127.0.0.1",
                                 port = "5432",
                                 database = "cloud")

   cursor = connection.cursor()
   print ( connection.get_dsn_parameters(),"\n")

   cursor.execute("SELECT version();")
   record = cursor.fetchone()
   print("You are connected to - ", record,"\n")

except (Exception, psycopg2.Error) as error :
   print ("Error while connecting to PostgreSQL", error)
finally:
```

```
#closing database connection.
    if(connection):
        cursor.close()
        connection.close()
        print("PostgreSQL connection is closed")
```

Рисунок 5.2 – Стадии взаимодействия Python-приложения и СУБД через Psycopg2

После выполнения кода будет выведена следующая информация:

```
{'user': 'postgres', 'dbname': 'pynative_DB', 'host': '127.0.0.1', 'port': '5432', 'tty': '',
'options': '', 'sslmode': 'prefer', 'sslcompression': '1', 'krbsrvname': 'postgres',
'target_session_attrs': 'any'}
You are connected to -  ('PostgreSQL 10.3')
PostgreSQL connection is closed
```

Рисунок 5.3 – Вывод Python-приложения

Рассмотрим схему работы с СУБД средствами Python более детально:

import psycopg2

Эта строка импортирует модуль psycopg2 в нашу программу. Используя классы и методы, определенные модулем psycopg2, мы можем взаимодействовать с PostgreSQL.

from psycopg2 import Error

Используя класс Error класса psycopg2, мы можем обрабатывать любые ошибки и исключения из базы данных, которые могут возникнуть при работе с PostgreSQL из Python. Используя этот подход, мы можем сделать наше приложение надежным. Этот модуль помогает описать ошибку более подробно. Возвращает сообщение об ошибке и код ошибки.

psycopg2.connect()

С помощью метода connect() мы можем создать соединение с экземпляром базы данных PostgreSQL. Это возвращает объект PostgreSQL Connection Object. Это соединение является потокобезопасным и может быть

разделено между многими потоками. Метод connect() принимает различные аргументы. В нашем примере для подключения PostgreSQL мы передали следующие аргументы подключения.

```
user = "puser", password = " puserxxx ", host = "127.0.0.1", port = "5432",
database = "cloud"
```

rcursor = connection.cursor()

Используя connection.cursor( ), мы можем создать объект курсора, который позволяет нам выполнять команду PostgreSQL через исходный код Python.

Мы можем создать столько курсоров, сколько захотим из одного объекта подключения. Курсоры, созданные из одного и того же соединения, не являются изолированными, то есть любые изменения, внесенные в базу данных курсором, сразу же видны другим курсорам.

Курсоры не являются потокобезопасными. После этого мы распечатали свойства соединения PostgreSQL, используя connection.get_dsn_parameters( ).

cursor.execute()

Используя метод execute курсора, мы можем выполнить операцию или запрос к базе данных. Метод execute принимает запрос SQL в качестве параметра. Мы можем получить результат запроса, используя методы курсора, такие как fetchone (), fetchmany (), fetcthall ().

В нашем примере мы выполняем SELECT version( ); запрос для получения версии PostgreSQL.

try-except-finally

Мы поместили весь наш код в блок try. Кроме того, чтобы перехватывать исключения базы данных и ошибки, которые могут возникнуть во время этого процесса.

cursor.close( ) и connection.close( )

Во избежание проблем с базой данных всегда рекомендуется закрывать курсор и объект соединения после завершения работы.

2. Рассмотрим процесс создания таблицы с использованием языка Python (рисунок 5.4)

```
import psycopg2
from psycopg2 import Error
try:
   connection = psycopg2.connect(user = "puser",
                                 password = "puserxxx",
                                 host = "127.0.0.1",
                                 port = "5432",
                                 database = "cloud")
   cursor = connection.cursor()
   create_table_query = '''CREATE TABLE mobile
        (ID INT PRIMARY KEY     NOT NULL,
        MODEL           TEXT    NOT NULL,
        PRICE        REAL); '''
   cursor.execute(create_table_query)
   connection.commit()
   print("Table created successfully in PostgreSQL ")
except (Exception, psycopg2.DatabaseError) as error :
   print ("Error while creating PostgreSQL table", error)
finally:
      if(connection):
         cursor.close()
         connection.close()
         print("PostgreSQL connection is closed")
```

Рисунок 5.4 – Создание таблицы на языке Python

В результате выполнения должно быть отображено сообщение:

Table created successfully in PostgreSQL

PostgreSQL connection is closed

5.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, создайте python-скрипты для подключения к базе данных, а также для выполнения следующих операций:

1. Создание таблицы.

2. Вывод информации о версии СУБД.

3. Необходимо реализовать и документировать все стадии, рассмотренные в методических указаниях.

5.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

5.8. Контрольные вопросы

1. Опишите основные стадии работы с СУБД на языке Python.

2. Опишите назначение и параметры метода psycopg2.connect( ).

3. Опишите назначение и параметры метода connection.cursor( ).

4. Опишите назначение и параметры метода cursor.execute( ).

5. Опишите назначение и параметры метода cursor.close( ).

6. Опишите модули, используемые для работы с СУБД на языке Python.

7. Какая команда используется для установки psycopg2?

8. Поясните механизм применения конструкции try-except-finally.

5.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 6. ОСНОВНЫЕ CRUD ОПЕРАЦИИ С ИСПОЛЬЗОВАНИЕМ PYTHON

6.1. Цель и содержание

Цель лабораторной работы: научиться реализовывать полный спектр CRUD-операций с использованием языка Python.

В рамках лабораторной работы необходимо решить следующие задачи:

– научиться использовать команды для выборки данных;

– научиться использовать команды для удаления данных;

– научиться использовать команды для обновления данных;

научиться использовать команды для добавления данных.

6.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

6.3. Теоретическая часть

Для работы с СУБД с использованием языка Python необходимо выполнить следующие щаги:

1. Используя метод connect() библиотеки psycopg2 с установленными параметрами установить соединение с PostgreSQL.

2. Для получения данных из БД необходимо создать курсор. Курсор создается с использованием ранее созданного соединения..

3. По завершении работы необходимо закрыть как курсор, таки объект соединения.

4. В процессе работы с СУБД средствами Ptrhon может возникнуть необходимость обработки исключений.

6.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

6.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

6.6. Методика и порядок выполнения работы

6.6.1. Решение учебной задачи

Постановка задачи. Разработайте приложение Python для реализации сервиса, реализующего полный спектр CRUD-операций.

1. Для реализации операции выборки данных необходимо использовать следующий код:

```
import psycopg2

try:
  connection = psycopg2.connect(user="puser",
                                password="puserxxx",
                                host="127.0.0.1",
```

```
                        port="5432",
                        database="cloud")
  cursor = connection.cursor()
  postgreSQL_select_Query = "select * from mobile"
  cursor.execute(postgreSQL_select_Query)
  mobile_records = cursor.fetchall()
  for row in mobile_records:
     print("Id = ", row[0], )
     print("Model = ", row[1])
     print("Price  = ", row[2], "\n")
except (Exception, psycopg2.Error) as error :
   print ("Error while fetching data from PostgreSQL", error)
finally:
   if(connection):
      cursor.close()
      connection.close()
```

Рисунок 6.1 – Получение данных

2. Для реализации операции удаления данных необходимо использовать следующий код:

```
sql_delete_query = """Delete from mobile where id = %s"""
      cursor.execute(sql_delete_query, (mobileId, ))
      connection.commit()
      count = cursor.rowcount
      print(count, "Record deleted successfully ")
```

Рисунок 6.2 – Удаление данных

3. Для реализации операции обновления данных необходимо использовать следующий код:

```
sql_update_query = """Update mobile set price = %s where id = %s"""
      cursor.execute(sql_update_query, (price, mobileId))
      connection.commit()
      count = cursor.rowcount
      print(count, "Record Updated successfully ")
```

Рисунок 6.3 – Обновление данных

4. Для реализации операции обновления данных необходимо использовать следующий код:

```
postgres_insert_query =
    """ INSERT INTO mobile (ID, MODEL, PRICE) VALUES (%s,%s,%s)"""
  record_to_insert = (5, 'One Plus 6', 950)
  cursor.execute(postgres_insert_query, record_to_insert)
  connection.commit()
  count = cursor.rowcount
  print (count, "Record inserted successfully into mobile table")
```

Рисунок 6.4 – Добавление данных

Данных операций достаточно, чтобы реализовать полноценный сервис, реализующий CRUD-операций.

6.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, создайте python-приложение для реализации CRUD-операций:

1. Добавление данных.
2. Удаление данных.
3. Обновление данных.
4. Выборка данных

6.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.
2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.
3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

6.8. Контрольные вопросы

1. Приведите пример исходного кода на языке Python для вставки записей в таблицу базы данных.

2. Приведите пример исходного кода на языке Python для обновления множества записей в таблице базы данных.

3. Приведите пример исходного кода на языке Python для удаления множества записей в таблице базы данных.

4. Приведите пример исходного кода на языке Python для выборки множества записей из таблицы базы данных.

5. Приведите пример исходного кода на языке Python для создания соединения с базой данных.

6. Опишите модули, используемые для работы с СУБД на языке Python.

6.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 7. ИСПОЛЬЗОВАНИЕ ХРАНИМЫХ ПРОЦЕДУР В PYTHON

7.1. Цель и содержание

Цель лабораторной работы: научиться реализовывать полный спектр CRUD-операций с использованием языка Python.

В рамках лабораторной работы необходимо решить следующие задачи:

– научиться использовать хранимые процедуры;

– научиться ркализовывать CRUD-операции.

7.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

7.3. Теоретическая часть

Для вызова хранимой процедуры PostgreSQL в программе Python используются следующие действия:

1. Создайте новое соединение с базой данных на сервере базы данных PostgreSQL, вызвав функцию connect( ) модуля psycopg.

```
conn = psycopg2.connect(dsn)
```

Метод connect( ) возвращает новый экземпляр класса соединения.

2. Создайте новый курсор, вызвав метод cursor( ) объекта соединения.

```
cur = conn.cursor()
```

3. Передайте имя хранимой процедуры и необязательные входные значения методу callproc( ) объекта курсора.

```
cur.callproc('stored_procedure_name', (value1,value2))
```

4. Метод callproc( ) переводит вызов хранимой процедуры и входные параметры в следующую команду:

```
SELECT * FROM stored_procedure_name(value1,value2);
```

Поэтому вы можете использовать метод execute () объекта курсора для вызова хранимой процедуры следующим образом:

```
cur.execute("SELECT * FROM stored_procedure_name( %s,%s); ",(value1,value2))
```

Оба выражения имеют одинаковый эффект.

5. После этого обработайте набор результатов, возвращенный хранимой процедурой, с помощью методов fetchone( ), fetchall( ) или fetchmany( ).

6. Вызовите метод close( ) для объектов курсора и соединения, чтобы закрыть связь с сервером базы данных PostgreSQL.

```
cur.close()
conn.close()
```

7.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

7.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей

необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

7.6. Методика и порядок выполнения работы

7.6.1. Решение учебной задачи

Постановка задачи: разработать метод Python для вызова хранимой процедуры БД.

Решение:

1. В базе данных необходимо опоределить хранимую процедуру:

```
CREATE OR REPLACE FUNCTION get_parts_by_vendor(id integer)
  RETURNS TABLE(part_id INTEGER, part_name VARCHAR) AS
$$
BEGIN
RETURN QUERY
SELECT parts.part_id, parts.part_name
FROM parts INNER JOIN vendor_parts on vendor_parts.part_id = parts.part_id WHERE vendor_id = id;
END; $$
LANGUAGE plpgsql;
```

Рисунок 7.1 – Определение хранимой процедуры

2. В среде разработки необходимо создать приложение:

```
import psycopg2
from config import config

def get_parts(vendor_id):
    conn = None
    try:
        params = config()
        conn = psycopg2.connect(**params)
        cur = conn.cursor()
        # cur.execute("SELECT * FROM get_parts_by_vendor( %s); ",(vendor_id,))
        cur.callproc('get_parts_by_vendor', (vendor_id,))
        row = cur.fetchone()
```

```
        while row is not None:
            print(row)
            row = cur.fetchone()
        cur.close()
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if conn is not None:
            conn.close()

if __name__ == '__main__':
    get_parts(1)
```

Рисунок 7.2 – Исходный код вызова процедуры на языке Python

7.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, доработайте python-приложение. Приложение должно содержать вызовы хранимых процедур (более трех).

7.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

7.8. Контрольные вопросы

1. Приведите пример исходного кода на языке Python для удаления множества записей в таблице базы данных.

2. Приведите пример исходного кода на языке Python для выборки множества записей из таблицы базы данных.

3. Приведите пример исходного кода на языке Python для создания соединения с базой данных.

4. Приведите пример исходного кода на языке Python для вызова хранимой процедура.

5. Поясните назначение методов fetchone( ), fetchall( ) или fetchmany( ), а также их параметры и возврат.

7.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 8. УПРАВЛЕНИЕ ТРЕНЗАКЦИЯМИ

8.1. Цель и содержание

Цель лабораторной работы: научиться тспользовать транзакции в приложениях Python.

В рамках лабораторной работы необходимо решить следующие задачи:

– научиться создавать транзакции на уровне СУБД;

– научиться управлять транзакциями на уровне клиентского приложения;

– изучить механизмы фиксации, отката транзакций, а также точки сохранения;

– освоить навыки работы с точками сохранения.

8.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

8.3. Теоретическая часть

8.3.1. Основы транзакций

В реляционной модели описана логическая единица обработки данных – транзакция. Можно сказать, что транзакция – это множество последовательно выполняемых операций. РСУБД предоставляет механизм блокировки, гарантирующий целостность транзакций.

Транзакция – это множество операций, в состав которого могут входить операции обновления, удаления, вставки и выборки данных. Часто эти операции погружаются в язык более высокого уровня или явно обертываются в блок транзакции, заключенный между командами BEGIN и END .

Транзакция считается успешно выполненной, если успешно выполнены все составляющие ее операции. Если какая-то операция транзакции завершается неудачно, то частично выполненные действия можно откатить.

Для явного управления транзакцией можно поставить команду BEGIN в ее начале и команду END или COMMIT в конце. В следующем примере показано, как выполнить команду SQL в транзакции:

```
BEGIN;
CREATE TABLE employee (id serial primary key, name text, salary numeric);
COMMIT;
```

Помимо обеспечения целостности данных транзакции позволяют легко отменить изменения, внесенные в базу в процессе разработки и отладки. В интерактивном режиме блок транзакции можно сочетать с командой SAVEPOINT, чтобы поставить отметку в точке сохранения состояния. Точка сохранения – это способ откатить не всю транзакцию целиком, а только ее часть. В примере ниже демонстрируется использование SAVEPOINT:

```
BEGIN;
UPDATE employee set salary = salary*1.1;
SAVEPOINT increase_salary;
UPDATE employee set salary = salary + 500 WHERE name ='john';
ROLLBACK to increase_salary;
COMMIT;
```

Здесь команда UPDATE employee SET salary = salary*1.1; зафиксирована в базе данных, а команда UPDATE employee SET salary = salary + 500 WHERE name = 'john'; отменена.

Все выполняемые PostgreSQL команды транзакционные, даже если блок транзакции не был открыт явно. Когда мы выполняем несколько команд, не обертывая их блоком транзакции, каждая команда выполняется в отдельной транзакции.

Поведением транзакций управляют драйверы базы данных и каркасы приложений типа Java EE или Spring. Например, в случае JDBC мы можем при желании задать режим автоматической фиксации.

8.3.2. Транзакции и свойства ACID

Гарантия атомарности, согласованности, изолированности и долговечности (ACID) операций – фундаментальное свойство реляционной базы данных.

Транзакция – это логическая единица выполнения, она неделима, или атомарна, т. е. либо выполнена целиком, либо не выполнена вовсе; и это утверждение справедливо вне зависимости от причины ошибки. Например, транзакция может завершиться неудачно из-за математической ошибки, неправильно написанного имени отношения и даже из-за аварии операционной системы.

После успешной фиксации транзакции все произведенные в ней изменения должны сохраняться даже после отказов оборудования; это свойство называется долговечностью. В многопользовательской среде пользователи могут одновременно выполнять несколько транзакций, каждая из которых содержит несколько операций. Любая транзакция должна выполняться, не мешая другим транзакциям, выполняющимся вместе с ней; это свойство называется изолированностью.

Наконец, согласованность – это не свойство транзакции как таковой, а желательное следствие изолированности и атомарности. Согласованность базы данных непосредственно связана с бизнес-требованиями, которые определяются с помощью правил, триггеров и ограничений. Если база данных находилась в согласованном состоянии до выполнения транзакции, то состояние должно быть согласовано и после ее завершения. Следить за согласованностью базы данных – задача разработчика, запрограммировавшего транзакцию.

Обеспечить свойства ACID нелегко. Например, долговечность – трудно исполнимое требование, потому что факторов, способных привести к потере данных, множество: крах операционной системы, прекращение электроснабжения, отказ жесткого диска и т. д. К тому же компьютерная

архитектура весьма сложна, в частности до появления на жестком диске данные проходят через несколько уровней системы хранения: оперативная память, буферы ввода-вывода, кеш диска.

8.3.3. Транзакции и конкурентность

Конкурентность можно определить как чередование действий во времени для создания иллюзии одновременного выполнения. При этом необходимо решить несколько проблем, в т. ч. как обеспечить безопасный доступ к глобальным ресурсам и как обрабатывать ошибки, когда имеется несколько контекстов выполнения.

Конкурентность встречается в компьютерах повсеместно, начиная с низкоуровневой аппаратной логики и кончая доступом к общим данным из любой точки земного шара. Примеры конкурентности можно найти в организации конвейера команд в микропроцессорах и в многопоточной архитектуре операционной системы. Даже в своей повседневной практике мы сталкиваемся с конкурентностью, работая, например, с системой Git. Взять типичную ситуацию – несколько разработчиков работают с одной и той же кодовой базой, при этом возникают конфликты, которые в конечном счете разрешаются в процедуре объединения.

Не следует путать конкурентность и параллелизм. Под параллелизмом понимается использование дополнительных вычислительных устройств, которые могут выполнить больше работы в единицу времени, тогда как конкурентность касается логического доступа к разделяемым ресурсам. Можно считать, что физический параллелизм – частный случай конкурентности. Другой ее формой является квантование процессорного времени, т. е. виртуальный параллелизм.

Во многих базах данных параллелизм является дополнением к конкурентности. В PostgreSQL параллелизм используется начиная с версии 9.6, чтобы увеличить скорость выполнения запросов. В СУБД Greenplum, ответвившейся от PostgreSQL, также применяется массово параллельная

обработка (massively parallel processing – MPP), чтобы повысить производительность и справиться с нагрузками, характерными для хранилищ данных.

8.3.4. Уровни изоляции транзакций

Разработчик может задать уровень изоляции транзакции, выполнив такую коман ду SQL:

```
SET TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
COMMITTED | READ UNCOMMITTED}
```

Команда SET TRANSACTION ISOLATION LEVEL должна вызываться внутри блока транзакции до начала запроса, иначе она не возымеет эффекта. Существует и другой способ:

```
BEGIN TRANSACTION ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |
READ
COMMITTED | READ UNCOMMITTED}
```

Наконец, можно изменить уровень изоляции для всей базы данных, например:

```
ALTER DATABASE <DATABASE NAME> SET DEFAULT_TRANSACTION_ISOLATION TO
SERIALIZABLE;
```

Как следует из этих примеров, существует четыре уровня изоляции:

1. SERIALIZABLE: обеспечивает самую строгую согласованность и освобождает разработчика от необходимости думать о конкурентности. Но за это приходится расплачиваться производительностью. В стандарте SQL уровень SERIALIZABLE подразумевается по умолчанию. В PostgreSQL по умолчанию подразумевается уровень READ COMMITTED.

2. REPEATABLE READ: следующий по строгости уровень изоляции. Он похож на READ COMMITTED тем, что допускает чтение только зафиксированных данных, но дополнительно гарантирует, что прочитанные данные не изменятся при повторном чтении.

3. READ COMMITTED: уровень, подразумеваемый по умолчанию в PostgreSQL. Разрешает транзакции читать только зафиксированные данные. При таком уровне предпочтение отдается производительности, а не точности.

4. READ UNCOMMITTED: самый нестрогий уровень изоляции. Допускает чтение незафиксированных данных.

Уровень READ UNCOMMITTED в PostgreSQL не поддерживается и трактуется так же, как READ COMMITTED . PostgreSQL поддерживает только три уровня изоляции.

8.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

8.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

8.6. Методика и порядок выполнения работы

8.6.1. Решение учебной задачи

Постановка задачи. Разработать модуль Python, демонстрирующий выпрлнение транзакций.

Для решения задачи необходимо реализовать следующий код:

```
import psycopg2
from psycopg2 import Error
try:
  connection = psycopg2.connect(user="puser",
                        password="puserxxx",
                        host="127.0.0.1",
                        port="5432",
                        database="cloud")
  connection.autocommit=False
  cursor = connection.cursor()
  amount = 2500
  query = """select balance from account where id = 624001562408"""
  cursor.execute(query)
  record = cursor.fetchone()
  balance_account_A  = int(record)
  balance_account_A -= amount
  sql_update_query = """Update account set balance = %s where id = 624001562408"""
  cursor.execute(sql_update_query,(balance_account_A,))

  query = """select balance from account where id = 2236781258763"""
  cursor.execute(query)
  record = cursor.fetchone()
  balance_account_B = int(record)
  balance_account_B += amount

  sql_update_query = """Update account set balance = %s where id =
2236781258763"""
  cursor.execute(sql_update_query, (balance_account_B,))

  connection.commit()
  print("Transaction completed successfully ")
```

```
except (Exception, psycopg2.DatabaseError) as error :
    print ("Error in transction Reverting all other operations of a transction ", error)
    connection.rollback()

finally:
    if(connection):
        cursor.close()
        connection.close()
        print("PostgreSQL connection is closed")
```

Рисунок 8.1 – Использование механизма транзакций в Python

На рисунке 8.2 представлен пример работы с транзакциями посредством оператора with:

```
with connection:
    with connection.cursor() as cursor:

    # Find item price
    query = """select price from itemstable where itemid = 876"""
    cursor.execute(query)
    record = cursor.fetchone()
    Itemprice = int(record)

    # find customer's ewallet balance
    query = """select balance from ewallet where userId = 23"""
    cursor.execute(query)
    record = cursor.fetchone()
    ewalletBalance  = int(record)
    new_EwalletBalance -= Itemprice

    # Withdraw from ewallet now
    sql_update_query = """Update ewallet set balance = %s where id = 23"""
    cursor.execute(sql_update_query,(new_EwalletBalance,))

    # add to company's account
    query = """select balance from account where accountId = 2236781258763"""
    cursor.execute(query)
    record = cursor.fetchone()
```

```
    accountBalance  = int(record)
    new_AccountBalance += Itemprice


    # Credit to  company account now
    sql_update_query = """Update account set balance = %s where id =
2236781258763"""
    cursor.execute(sql_update_query, (new_AccountBalance,))
    print("Transaction completed successfully ")
```

Рисунок 8.2 – Использование механизма транзакций в Python при использовании with (фрагмент кода)

В системах баз данных, используя уровни изоляции, мы можем определить, какой уровень целостности транзакции виден другим пользователям и системам.

Например, когда пользователь выполняет какое-либо действие или операцию, и операция еще не завершена, подробности этой операции доступны для других пользователей для выполнения некоторых одновременных действий. Например, если пользователь покупает какой-либо товар, данные этой операции передаются другим пользователям системы для подготовки счетов, получения товара и расчета скидки для ускорения процесса.

Если уровень изоляции низкий, многие пользователи могут получить доступ к одним и тем же данным одновременно. Но это также может привести ко многим проблемам параллелизма, таким как грязное чтение и потерянное обновление. Таким образом, вы должны использовать уровень изоляции, учитывая все эти моменты. Более высокий уровень изоляции может заблокировать другого пользователя или транзакцию, чтобы завершить себя в первую очередь.

В psycopg2.extensions определены следующие уровни изоляции:

psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT
psycopg2.extensions.ISOLATION_LEVEL_READ_UNCOMMITTED
psycopg2.extensions.ISOLATION_LEVEL_READ_COMMITTED
psycopg2.extensions.ISOLATION_LEVEL_REPEATABLE_READ
psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE

psycopg2.extensions.ISOLATION_LEVEL_DEFAULT

Для задания уровня изоляции используется метод:

conn.set_isolation_level(psycopg2.extensions.ISOLATION_LEVEL_AUTOCOMMIT)

8.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, создайте python-приложение для реализации механизма транзакций:

1. Раработайте не менее трех методов (функций) в своем приложении для работы с транзакциями.

2. Продемонстрируйте как фиксацию транзакции, так и ее откат.

3. Продемонстрируйте использование точек сохранения.

8.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

8.8. Контрольные вопросы

1. Опишите свойства ACID.

2. Какие из свойств ACID не характерны для NoSQL-подхода?

3. Опишите назначение транзакционного механизма.

4. В каких случаях может понадобиться откат транзакции?

5. Что такое точка сохранения? В каких случаях целесообразно использование данного механизма? Приведите примеры из практики.

6. Поясните механизм управления транзакциями в psycopg2.

7. Для чего используется оператор with при работе с транзакциями в Python?

8. Опишите смысл различных уровней изоляции транзакций.

8.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

ЛАБОРАТОРНАЯ РАБОТА 9. РАБОТА С BLOB В PYTHON

9.1. Цель и содержание

Цель лабораторной работы: научиться обрабатывать данные бинарного формата, хранящиеся в базе данных.

В рамках лабораторной работы необходимо решить следующие задачи:

– освоение принципов работы с BLOB-данными;

– научиться осуществлять выборку BLOB-данных.

9.2. Формируемые компетенции

Лабораторная работа направлена на формирование следующих компетенций: ОПК-5.

9.3. Теоретическая часть

Стандартный SQL определяет BLOB как большой двоичный объект для хранения двоичных данных в базе данных. С типом данных BLOB вы можете сохранить содержимое таблицы, документа и т. д. в таблицу. PostgreSQL не поддерживает BLOB, но вы можете использовать тип данных BYTEA для хранения двоичных данных.

9.4. Оборудование и материалы

Для выполнения лабораторной работы необходим персональный компьютер с установленной ОС Linux или виртуальная машина с гостевой ОС Linux. На компьютере должна быть обеспечена возможность установки и настройки СУБД PostgreSQL. В качестве редактора SQL-скриптов используется VS Code (возможно применение любого редактора).

9.5. Указания по технике безопасности

Студенты должны следовать общепринятой технике безопасности для пользователей персональных компьютеров. Не следует самостоятельно производить ремонт технических средств, установку и удаление программного обеспечения. В случае обнаружения неисправностей необходимо сообщить об этом администратору компьютерного класса (обслуживающему персоналу лаборатории).

9.6. Методика и порядок выполнения работы

9.6.1. Решение учебной задачи

Постановка задачи. Разработать модуль Python, реализующий добавление и чтение BLOB-данных из базы данных.

1. Создадим БД, а также мвязанные таблицы, представленные на рисунке 9.1.

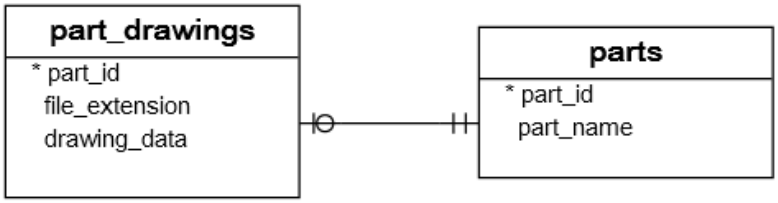

Рисунок 9.1 – Структура таблиц

2. Для решения задачи всьавки необходимо реализовать следующий код:

```
import psycopg2
from config import config

def write_blob(part_id, path_to_file, file_extension):
    conn = None
    try:
        drawing = open(path_to_file, 'rb').read()
        params = config()
```

```
        conn = psycopg2.connect(**params)
        cur = conn.cursor()
        cur.execute("INSERT INTO part_drawings(part_id,file_extension,drawing_data) " +
                    "VALUES(%s,%s,%s)",
                    (part_id, file_extension, psycopg2.Binary(drawing)))
        cur.close()
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if conn is not None:
            conn.close()
```

Рисунок 9.2 – Метод для вставки BLOB-данных в приложении Python

3. На рисунке 9.3 представлен пример извлечения BLOB-данных из таблицы БД:

```
def read_blob(part_id, path_to_dir):
    conn = None
    try:
        params = config()
        conn = psycopg2.connect(**params)
        cur = conn.cursor()
        cur.execute(""" SELECT part_name, file_extension, drawing_data
                        FROM part_drawings
                        INNER JOIN parts on parts.part_id = part_drawings.part_id
                        WHERE parts.part_id = %s """,
                    (part_id,))
        blob = cur.fetchone()
        open(path_to_dir + blob[0] + '.' + blob[1], 'wb').write(blob[2])
        cur.close()
    except (Exception, psycopg2.DatabaseError) as error:
        print(error)
    finally:
        if conn is not None:
            conn.close()
```

Рисунок 9.3 – Извлечение BLOB-данных в приложении Python

9.6.2. Выполнение индивидуального задания

В соответствии с вариантами индивидуальных заданий, выбранными в рамках лабораторной работы №1, создайте python-приложение для сохранения / чтения BLOB-данных:

1. Добавьте поле для хранения бинарных данных в базу данных (более двух).

2. Реализуйте метод чтения и восстановления документа из базы данных.

3. Реализуйте метод чтения и восстановления изображения из базы данных.

9.7. Содержание отчета и его форма

Отчет по лабораторной работе должен содержать:

1. Номер и название лабораторной работы; задачи лабораторной работы.

2. Реализация каждого пункта подраздела «Индивидуальное задание» с приведением исходного кода программы, диаграмм и графиков для визуализации данных.

3. Ответы на контрольные вопросы.

4. Экранные формы (консольный вывод) и листинг программного кода с комментариями, показывающие порядок выполнения лабораторной работы, и результаты, полученные в ходе её выполнения.

Отчет о выполнении лабораторной работы подписывается студентом и сдается преподавателю.

9.8. Контрольные вопросы

1. Что такое BLOB? Какой тип данных используется в PostgreSQL для хранения BLOB?

2. Для чего используется метод psycopg2.Binary( )?

3. Чем отличаются методы добавления бинарного содержимого различного типа (изображения, документы, аудио, …)?

4. Опишите назначение и параметры метода psycopg2.connect( ).

5. Опишите назначение и параметры метода connection.cursor( ).

6. Опишите назначение и параметры метода cursor.execute( ).

7. Опишите назначение и параметры метода cursor.close( ).

9.9. Список литературы

Для выполнения лабораторной работы необходимо использовать следующие информационные ресурсы и литературу: [1-5]

СПИСОК ЛИТЕРАТУРЫ

Основная литература

1. Медведкова И.Е. Базы данных [Электронный ресурс]: учебное пособие/ Медведкова И.Е., Бугаев Ю.В., Чикунов С.В.— Электрон. текстовые дан-ные.— Воронеж: Воронежский государственный университет инженерных технологий, 2014.— 104 с.— Режим доступа: http://www.iprbookshop.ru/47418.— ЭБС «IPRbooks».

2. Молдованова О.В. Информационные системы и базы данных [Электронный ресурс]: учебное пособие/ Молдованова О.В.— Электрон. тексто-вые данные.— Новосибирск: Сибирский государственный университет те-лекоммуникаций и информатики, 2014.— 178 с.— Режим доступа: http://www.iprbookshop.ru/45470.— ЭБС «IPRbooks».

Дополнительная литература

1. Баженова, И. Ю. Основы проектирования приложений баз данных / И. Ю. Баженова. – 2-е изд., испр. – М.: Национальный Открытый Университет «ИНТУИТ», 2016. – 238 с. [Электронный ресурс]. Режим доступа: http://biblioclub.ru/index.php?page=book&id=428933.

2. Павлова, Е.А. Технологии разработки современных информационных систем на платформе Microsoft.NET Электронный ресурс : учебное пособие / Е.А. Павлова. – Технологии разработки современных информационных систем на платформе Microsoft.NET,2022-12-01. – Москва : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016. - 128 с. – Книга находится в базовой версии ЭБС IPRbooks. – ISBN 978-5-9963-0003-7, экземпляров неограниченно

Электронные ресурсы

5. https://www.mockaroo.com – сервис для генерации данных.