Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

ОТЧЕТ

по лабораторной работе № 3

на тему

ОСВОЕНИЕ ПРИКЛАДНОГО ИНТЕРФЕЙСА СУБД BERKELEYDB. РАЗРАБОТКА КОНВЕРТОРА БАЗЫ ДАННЫХ POSTGRESQL В НАБОР БАЗ ДАННЫХ BERKELEYDB. АДАПТАЦИЯ СПЕЦИФИКАЦИЙ ПРИЛОЖЕНИЯ

ВАРИАНТ №11 (ШКОЛА)

Студент: А.Н. Климович

Преподаватель: Ю.Ю. Желтко

МИНСК 2024

**СОДЕРЖАНИЕ**

[1 ЦЕЛЬ РАБОТЫ 4](#_Toc182190059)

[2 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ 5](#_Toc182190060)

[2.1 Задание 5](#_Toc182190061)

[2.1.1 Извлечение схемы и данных из PostgreSQL 5](#_Toc182190062)

[2.1.2 Конвертация данных в Berkeley DB 5](#_Toc182190063)

[2.1.3 Запись данных в Berkeley DB 5](#_Toc182190064)

[2.1.4 Адаптация спецификаций приложения 5](#_Toc182190065)

[2.2 Требования к реализации 5](#_Toc182190066)

[2.2.1 Подключение к PostgreSQL 5](#_Toc182190067)

[2.2.2 Работа с Berkeley DB 5](#_Toc182190068)

[2.2.3 Конвертация данных 5](#_Toc182190069)

[3 ОПИСАНИЕ СТРУКТУРЫ ХРАНЕНИЯ ДАННЫХ В BERKELEY  DB 6](#_Toc182190070)

[4 ПРИМЕР SQL-ЗАПРОСОВ ДЛЯ ИЗВЛЕЧЕНИЯ СХЕМЫ И  ДАННЫХ ИЗ POSTGRESQL 7](#_Toc182190071)

[4.1 Пример 1 7](#_Toc182190072)

[4.2 Пример 2 7](#_Toc182190073)

[4.3 Пример 3 7](#_Toc182190074)

[4.4 Пример 4 7](#_Toc182190075)

[5 КОД КОНВЕРТЕРА ДАННЫХ ИЗ POSTGRESQL В BERKELEY  DB, ВКЛЮЧАЯ СЕРИАЛИЗАЦИЮ И ДЕСЕРИАЛИЗАЦИЮ 8](#_Toc182190076)

[5.1 Описание функционала 8](#_Toc182190077)

[5.2 Используемые библиотеки 8](#_Toc182190078)

[5.2.1 psycopg2 8](#_Toc182190079)

[5.2.2 bsddb3 8](#_Toc182190080)

[5.2.3 pickle 8](#_Toc182190081)

[5.2.4 os 8](#_Toc182190082)

[5.3 Описание ключевых методов 9](#_Toc182190083)

[5.3.1 \_\_init\_\_ 9](#_Toc182190084)

[5.3.2 fetch\_tables\_info 9](#_Toc182190085)

[5.3.3 fetch\_data 9](#_Toc182190086)

[5.3.4 serialize\_data 9](#_Toc182190087)

[5.3.5 generate\_combined\_key 9](#_Toc182190088)

[5.3.6 create\_berkeley\_db 9](#_Toc182190089)

[5.3.7 migrate\_data 9](#_Toc182190090)

[5.3.8 \_\_del\_\_ 9](#_Toc182190091)

[5.4 Код конвертера 10](#_Toc182190092)

[5.5 Пример использования 12](#_Toc182190093)

[5.6 Тестирование 13](#_Toc182190094)

[5.6.1 Описание тестов 13](#_Toc182190095)

[5.6.2 Код для тестов 14](#_Toc182190096)

[5.6.3 Результаты тестирования 17](#_Toc182190097)

[6 ПРИМЕРЫ ОПЕРАЦИЙ ВСТАВКИ, ЧТЕНИЯ, ОБНОВЛЕНИЯ И  УДАЛЕНИЯ ДАННЫХ (CRUD) В BERKELEY DB 18](#_Toc182190098)

[6.1 Развертывание Berkeley DB 18](#_Toc182190099)

[6.2 Реализация CRUD для Berkeley DB 19](#_Toc182190100)

[6.3 Код класса BerkeleyDBManager 20](#_Toc182190101)

[6.4 Пример использования 22](#_Toc182190102)

[6.5 Тестирование 24](#_Toc182190103)

[6.5.1 Описание тестов 24](#_Toc182190104)

[6.5.2 Код для тестов 24](#_Toc182190105)

[6.5.3 Результаты тестирования 28](#_Toc182190106)

[7 ОПИСАНИЕ ИЗМЕНЕНИЙ В СПЕЦИФИКАЦИЯХ  ПРИЛОЖЕНИЯ, АДАПТИРОВАННЫХ ДЛЯ РАБОТЫ С  BERKELEY DB 29](#_Toc182190107)

[8 ПРЕИМУЩЕСТВА И НЕДОСТАТКИ POSTGRESQL И  BERKELEY DB 30](#_Toc182190108)

[8.1 Преимущества PostgreSQL 30](#_Toc182190109)

[8.2 Недостатки PostgreSQL 30](#_Toc182190110)

[8.3 Преимущества Berkeley DB 30](#_Toc182190111)

[8.4 Недостатки Berkeley DB 31](#_Toc182190112)

[8.5 Вывод 31](#_Toc182190113)

[ЗАКЛЮЧЕНИЕ 32](#_Toc182190114)

# **1 ЦЕЛЬ РАБОТЫ**

1. Научиться преобразовывать реляционные базы данных (PostgreSQL) в формат ключ-значение (Berkeley DB).

2. Освоить процесс сериализации и десериализации данных для хранения в нереляционной базе данных.

3. Выполнить адаптацию существующих спецификаций приложения для работы с Berkeley DB.

# **2 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ**

## **2.1 Задание**

### **2.1.1 Извлечение схемы и данных из PostgreSQL**

Используя подключение к PostgreSQL, извлеките структуру (схему) и данные из таблиц базы данных PostgreSQL. Для этого используйте SQL-запросы и выполните сериализацию данных.

### **2.1.2 Конвертация данных в Berkeley DB**

Для каждой таблицы в PostgreSQL создайте соответствующую базу данных Berkeley DB.

Используйте первичные ключи таблиц PostgreSQL в качестве ключей для Berkeley DB, а значения столбцов – в виде сериализованных структур.

### **2.1.3 Запись данных в Berkeley DB**

Реализуйте запись данных в формате ключ-значение в Berkeley DB. Для этого используйте соответствующие функции библиотеки Berkeley DB.

### **2.1.4 Адаптация спецификаций приложения**

Проанализируйте спецификации приложения, которые ранее работали с PostgreSQL, и адаптируйте их для работы с Berkeley DB.

Убедитесь, что операции вставки, обновления, удаления и поиска данных выполняются корректно с использованием нового формата хранения (ключ-значение).

## **2.2 Требования к реализации**

### **2.2.1 Подключение к PostgreSQL**

Написать скрипт, который подключается к базе данных PostgreSQL и извлекает данные из выбранных таблиц.

### **2.2.2 Работа с Berkeley DB**

Для каждой таблицы PostgreSQL создается отдельная база данных Berkeley DB.

Данные в Berkeley DB должны быть организованы в формате ключ-значение. Ключи должны соответствовать первичным ключам PostgreSQL.

### **2.2.3 Конвертация данных**

Данные из PostgreSQL должны быть сериализованы (например, в формате JSON) перед записью в Berkeley DB. При чтении данных необходимо выполнять десериализацию.

# **3 ОПИСАНИЕ СТРУКТУРЫ ХРАНЕНИЯ ДАННЫХ В BERKELEY  DB**

Данные PostgreSQL сохраняются в Berkeley DB по следующей схеме:

– ключом является строковое представление id любой записи в таблицах PostgreSQL;

– значением является сериализованные данные из PostgreSQL (JSON формат).

В таблице 3.1 представлено описание структуры хранения данных в Berkeley DB модели «Школа».

Таблица 3.1 – Структура хранения данных модели «Школа»

|  |  |  |
| --- | --- | --- |
| Имя таблицы | Ключ | Значение |
| class | id | id, class\_name, class\_teacher\_id |
| class\_teacher | {class\_id}\_{teacher\_id} | class\_id, teacher\_id |
| student | id | id, first\_name, last\_name, gender\_type, email, class\_id |
| student\_subject | {student\_id}\_{subject\_id} | student\_id, subject\_id |
| subject | id | id, subject\_name |
| teacher | id | id, first\_name, last\_name, age, phone\_no, gender\_type, subject\_id |
| Примечание – Значения в фигурных скобках, разделенные символом «\_» обозначает конкатенацию двух значений через символ «\_» | | |

В таблице 3.2 представлен пример хранения данных в Berkeley DB.

Таблица 3.2 – Пример хранения данных в Berkeley DB

|  |  |  |
| --- | --- | --- |
| Имя таблицы | Ключ | Значение (JSON) |
| class | 1 | {id: 1, class\_name: “1A”, class\_teacher\_id: 1} |
| class\_teacher | 1\_2 | {class\_id: 1, teacher\_id: 2} |
| student | 1 | {id: 1, first\_name: “alex”, last\_name: “black”, gender\_type: “male”, email: “alex\_b@mail.ru”, class\_id: 1} |
| student\_subject | 1\_2 | {student\_id: 1, subject\_id: 2} |
| subject | 1 | {id: 1, subject\_name: “English”} |

# **4 ПРИМЕР SQL-ЗАПРОСОВ ДЛЯ ИЗВЛЕЧЕНИЯ СХЕМЫ И  ДАННЫХ ИЗ POSTGRESQL**

## **4.1 Пример 1**

Далее приведен SQL-запрос для извлечения имен всех таблиц из PostgreSQL:

SELECT table\_name

FROM information\_schema.tables

WHERE table\_schema = 'public';

## **4.2 Пример 2**

Далее приведен SQL-запрос для получения столбцов таблицы studentиз PostgreSQL:

SELECT column\_name

FROM information\_schema.columns

WHERE table\_name = 'student';

## **4.3 Пример 3**

Далее приведен SQL-запрос для определения первичного ключа таблицы student:

SELECT a.column\_name

FROM information\_schema.table\_constraints t

JOIN information\_schema.key\_column\_usage a

ON a.constraint\_name = t.constraint\_name

WHERE t.constraint\_type = 'PRIMARY KEY'

AND t.table\_name = 'student';

## **4.4 Пример 4**

Далее приведен SQL-запрос для получения данных из таблицы class по столбцам id, class\_name, class\_teacher\_id:

SELECT id, class\_name, teacher\_id FROM class;

# **5 КОД КОНВЕРТЕРА ДАННЫХ ИЗ POSTGRESQL В BERKELEY  DB, ВКЛЮЧАЯ СЕРИАЛИЗАЦИЮ И ДЕСЕРИАЛИЗАЦИЮ**

## **5.1 Описание функционала**

Данный скрипт реализует конвертер данных, который переводит данные из базы данных PostgreSQL в Berkeley DB.

Класс PostgreSQLToBerkeleyDB выполняет следующие основные задачи:

1. Подключение к базе данных PostgreSQL.

2. Извлечение информации о таблицах и их схемах из PostgreSQL.

3. Получение данных из PostgreSQL и их сериализация.

3. Создание баз данных Berkeley DB и запись в них данных в виде ключ-значение.

4. Закрытие соединений с базами данных PostgreSQL и Berkeley DB.

## **5.2 Используемые библиотеки**

### **5.2.1 psycopg2**

Библиотека для работы с PostgreSQL через Python. Она позволяет подключаться к базе данных, выполнять SQL-запросы и извлекать данные.

Используется для подключения к PostgreSQL и выполнения запросов для получения информации о таблицах, их схемах и данных.

### **5.2.2 bsddb3**

Библиотека для работы с Berkeley DB в Python. Berkeley DB – это библиотека для управления ключ-значение, которая позволяет сохранять данные в формате хэш-таблиц.

Используется для создания и работы с файлами баз данных формата Berkeley DB.

### **5.2.3 pickle**

Стандартная библиотека Python для сериализации и десериализации объектов.

Используется для преобразования данных строк таблиц PostgreSQL в сериализованный формат перед записью в Berkeley DB.

### **5.2.4 os**

Стандартная библиотека для работы с файловой системой.

Используется для создания директории, если она не существует, перед сохранением баз данных Berkeley DB.

## **5.3 Описание ключевых методов**

### **5.3.1 \_\_init\_\_**

Подключается к базе данных PostgreSQL через psycopg2.connect.

Проверяет наличие директории для хранения баз данных Berkeley DB и создает ее при необходимости.

### **5.3.2 fetch\_tables\_info**

Извлекает список всех таблиц в базе данных PostgreSQL.

Далее получает список столбцов для каждой таблицы и определяет первичные ключи.

Возвращает структуру данных с информацией о каждой таблице.

### **5.3.3 fetch\_data**

Получает все строки данных из указанной таблицы PostgreSQL с учетом порядка столбцов.

Для извлечения данных использует SQL-запрос.

### **5.3.4 serialize\_data**

Преобразует данные строки в словарь, где ключами являются имена столбцов, а значениями — соответствующие данные.

### **5.3.5 generate\_combined\_key**

Генерирует комбинированный ключ для таблиц, где первичный ключ состоит из нескольких столбцов.

Объединяет значения всех столбцов первичного ключа в одну строку.

### **5.3.6 create\_berkeley\_db**

Создает базу данных Berkeley DB для указанной таблицы и записывает в нее данные в формате ключ-значение.

Ключом используется либо значение первичного ключа таблицы, либо сгенерированный комбинированный ключ (если ключ состоит из нескольких столбцов).

Значения записываются в сериализованном формате с помощью pickle.

### **5.3.7 migrate\_data**

Основной процесс миграции данных. Получает информацию о таблицах, извлекает данные из PostgreSQL и записывает их в соответствующие файлы баз данных Berkeley DB.

### **5.3.8 \_\_del\_\_**

Закрывает соединение с базой данных PostgreSQL.

## **5.4 Код конвертера**

import psycopg2

import bsddb3

import pickle

import os

class PostgreSQLToBerkeleyDB:

PATH\_TO\_SAVE = "app/database/berkeley/data"

def \_\_init\_\_(self, postgres\_url: str):

# Подключение к PostgreSQL

self.conn = psycopg2.connect(postgres\_url)

self.cursor = self.conn.cursor()

# Создание директории, если ее нет

if not

os.path.exists(PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE):

os.makedirs(PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE)

print(f"Директория '{PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE}' была создана.")

else:

print(f"Директория '{PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE}' уже существует.")

def fetch\_tables\_info(self):

"""Получить список таблиц и их схемы в PostgreSQL."""

# Извлекаем имена всех таблиц в базе данных

self.cursor.execute("""

SELECT table\_name

FROM information\_schema.tables

WHERE table\_schema = 'public';

""")

tables = self.cursor.fetchall()

tables\_info = {}

for table in tables:

table\_name = table[0]

# Получаем столбцы таблицы

self.cursor.execute(

f"SELECT column\_name FROM information\_schema.columns WHERE table\_name = '{table\_name}';")

columns = self.cursor.fetchall()

# Определим первичный ключ таблицы

self.cursor.execute(f"""

SELECT a.column\_name

FROM information\_schema.table\_constraints t

JOIN information\_schema.key\_column\_usage a

ON a.constraint\_name = t.constraint\_name

WHERE t.constraint\_type = 'PRIMARY KEY'

AND t.table\_name = '{table\_name}';

""")

primary\_keys = self.cursor.fetchall() # Могут быть несколько ключей

primary\_keys = [pk[0] for pk in primary\_keys] if primary\_keys else None

tables\_info[table\_name] = {

"columns": [col[0] for col in columns],

"primary\_keys": primary\_keys

}

return tables\_info

def fetch\_data(self, table\_name: str, columns):

columns\_str = ", ".join(columns)

self.cursor.execute(

f"SELECT {columns\_str} FROM {table\_name};"

)

rows = self.cursor.fetchall()

return rows

def serialize\_data(self, data, columns):

serialized\_data = {}

for i in range(len(columns)):

column\_name = columns[i]

column\_value = data[i]

serialized\_data[column\_name] = column\_value

return serialized\_data

def generate\_combined\_key(self, row, columns, primary\_keys):

key\_values = [

str(row[columns.index(col)]) for col in primary\_keys

]

combined\_key = "\_".join(key\_values)

return combined\_key

def create\_berkeley\_db(self, table\_name: str, data, columns, primary\_keys):

db\_name = f"{PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE}/{table\_name}.db"

db = bsddb3.hashopen(db\_name, 'c')

for row in data:

# Если у таблицы есть первичный ключ, используем его

if primary\_keys:

key = self.generate\_combined\_key(

row, columns, primary\_keys

).encode()

else:

key = str(row[columns.index(

primary\_keys[0]

)]).encode() # для остальных случаев

value = pickle.dumps(

self.serialize\_data(row, columns)

) # Сериализуем данные

db[key] = value

# Закрыть базу данных Berkeley DB

db.close()

def migrate\_data(self):

# Получаем информацию о таблицах

tables\_info = self.fetch\_tables\_info()

# Для каждой таблицы:

for table\_name, table\_info in tables\_info.items():

columns = table\_info["columns"]

primary\_keys = table\_info["primary\_keys"]

rows = self.fetch\_data(table\_name, columns)

# Создаем Berkeley DB и записываем данные

self.create\_berkeley\_db(

table\_name, rows, columns, primary\_keys

)

print(f"Данные таблицы '{table\_name}' успешно мигрированы в Berkeley DB.")

def \_\_del\_\_(self):

"""Закрыть подключение к PostgreSQL."""

self.cursor.close()

self.conn.close()

## **5.5 Пример использования**

Для использования конвертера достаточно создать экземпляр класса, указав имя таблицы, а затем использовать методы для выполнения конвертации данных. Например:

from app.database.berkeley import PostgreSQLToBerkeleyDB

if \_\_name\_\_ == '\_\_main\_\_':

# URL подключения к PostgreSQL

postgres\_url = 'postgresql://postgres:1234@localhost:5000/school'

# Создаем объект для миграции данных

migration = PostgreSQLToBerkeleyDB(postgres\_url)

# Мигрируем данные из PostgreSQL в Berkeley DB

migration.migrate\_data()

print("Миграция данных завершена.")

На рисунке 5.1 приведен результат выполнения данного кода.

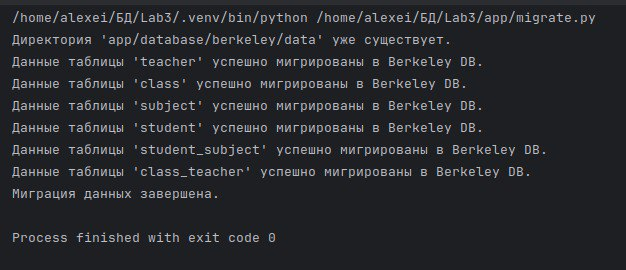


Рисунок 5.1 – Результат использования конвертера

## **5.6 Тестирование**

Для тестирования конвертера данных из PostgreSQL в Berkeley DB использовался фреймворк pytest. Были разработаны и успешно пройдены различные юнит-тесты, которые проверяют ключевые аспекты работы конвертера, такие как инициализация, извлечение данных, сериализация, а также создание и миграция данных в Berkeley DB.

### **5.6.1 Описание тестов**

Для полного тестирования было написано 7 тестов:

1. Тест инициализации и создания директории – проверяет, что при инициализации объекта миграции происходит создание необходимой директории для сохранения базы данных Berkeley, если она отсутствует.

2. Тест метода fetch\_tables\_info – этот тест проверяет правильность получения списка таблиц, их столбцов и первичных ключей из PostgreSQL. Для этого были замокированы результаты запросов, и тест проверяет соответствие ожидаемых и фактических данных.

3. Тест метода fetch\_data – проверяет, что метод fetch\_data корректно извлекает строки из указанной таблицы, используя соответствующие столбцы.

4. Тест сериализации данных – для подтверждения, что метод serialize\_data правильно преобразует строку данных в сериализованную форму, где значения столбцов ассоциируются с их именами.

5. Тест генерации комбинированного ключа – проверяет, что метод generate\_combined\_key корректно генерирует комбинированный ключ на основе данных строки и первичных ключей таблицы.

6. Тест создания Berkeley DB – тестируется процесс создания базы данных в Berkeley DB. Мокается процесс открытия базы данных, записи данных и закрытия базы.

7. Тест миграции данных – этот тест проверяет весь процесс миграции данных из PostgreSQL в Berkeley DB, включая получение данных из таблиц, сериализацию и создание базы данных.

### **5.6.2 Код для тестов**

import pytest

from unittest import mock

import pickle

from app.database.berkeley import PostgreSQLToBerkeleyDB

@pytest.fixture

def pg\_to\_berkeley():

with mock.patch('psycopg2.connect') as mock\_connect:

mock\_conn = mock.MagicMock()

mock\_cursor = mock.MagicMock()

mock\_connect.return\_value = mock\_conn

mock\_conn.cursor.return\_value = mock\_cursor

# Инициализация объекта

postgres\_url = "postgresql://user:password@localhost:5432/mydb"

migrator = PostgreSQLToBerkeleyDB(postgres\_url)

yield migrator, mock\_conn, mock\_cursor

# Тест инициализации и создания директории

def test\_init(pg\_to\_berkeley):

migrator, mock\_conn, mock\_cursor = pg\_to\_berkeley

# Проверяем создание директории, если ее нет

with mock.patch('os.makedirs') as mock\_makedirs, \

mock.patch('os.path.exists', return\_value=False):

migrator.\_\_init\_\_("postgresql://user:password@localhost:5432/mydb")

mock\_makedirs.assert\_called\_once\_with(PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE)

# Тест метода fetch\_tables\_info

def test\_fetch\_tables\_info(pg\_to\_berkeley):

migrator, mock\_conn, mock\_cursor = pg\_to\_berkeley

# Мокаем возвращаемые значения для таблиц и столбцов

mock\_cursor.fetchall.side\_effect = [

[('table1',), ('table2',)], # Таблицы

[('col1',), ('col2',)], # Столбцы для table1

[('pk\_col1',)], # Первичный ключ для table1

[('colA',), ('colB',)], # Столбцы для table2

[] # Нет первичного ключа для table2

]

tables\_info = migrator.fetch\_tables\_info()

expected = {

'table1': {

'columns': ['col1', 'col2'],

'primary\_keys': ['pk\_col1']

},

'table2': {

'columns': ['colA', 'colB'],

'primary\_keys': None

}

}

assert tables\_info == expected

assert mock\_cursor.execute.call\_count == 5

# Тест метода fetch\_data

def test\_fetch\_data(pg\_to\_berkeley):

migrator, mock\_conn, mock\_cursor = pg\_to\_berkeley

mock\_cursor.fetchall.return\_value = [('row1\_col1', 'row1\_col2'), ('row2\_col1', 'row2\_col2')]

columns = ['col1', 'col2']

data = migrator.fetch\_data('table1', columns)

mock\_cursor.execute.assert\_called\_with("SELECT col1, col2 FROM table1;")

assert data == [('row1\_col1', 'row1\_col2'), ('row2\_col1', 'row2\_col2')]

# Тест сериализации данных

def test\_serialize\_data(pg\_to\_berkeley):

migrator, \_, \_ = pg\_to\_berkeley

row = ['value1', 'value2']

columns = ['col1', 'col2']

serialized = migrator.serialize\_data(row, columns)

expected = {'col1': 'value1', 'col2': 'value2'}

assert serialized == expected

# Тест генерации комбинированного ключа

def test\_generate\_combined\_key(pg\_to\_berkeley):

migrator, \_, \_ = pg\_to\_berkeley

row = ['value1', 'value2']

columns = ['col1', 'col2']

primary\_keys = ['col1']

key = migrator.generate\_combined\_key(row, columns, primary\_keys)

assert key == 'value1'

# Тест создания Berkeley DB

def test\_create\_berkeley\_db(pg\_to\_berkeley):

migrator, \_, \_ = pg\_to\_berkeley

# Мокаем Berkeley DB

with mock.patch('bsddb3.hashopen') as mock\_db\_open, \

mock.patch('app.database.berkeley.PostgreSQLToBerkeleyDB.generate\_combined\_key',

return\_value='key1'):

mock\_db = mock.MagicMock()

mock\_db\_open.return\_value = mock\_db

table\_name = 'table1'

columns = ['col1', 'col2']

primary\_keys = ['col1']

data = [('value1', 'value2')]

migrator.create\_berkeley\_db(table\_name, data, columns, primary\_keys)

# Проверяем, что база данных открыта и запись добавлена

mock\_db\_open.assert\_called\_with(f"{PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE}/{table\_name}.db", 'c')

mock\_db.\_\_setitem\_\_.assert\_called\_once\_with(b'key1', pickle.dumps({'col1': 'value1', 'col2': 'value2'}))

mock\_db.close.assert\_called\_once()

# Тест миграции данных

def test\_migrate\_data(pg\_to\_berkeley):

migrator, \_, mock\_cursor = pg\_to\_berkeley

# Мокаем таблицы и данные

mock\_cursor.fetchall.side\_effect = [

[('table1',)], # Таблицы

[('col1',), ('col2',)], # Столбцы для table1 (исправлено: два столбца)

[('col1',)], # Первичный ключ для table1

[('value1', 'value2')] # Данные таблицы (два столбца)

]

with mock.patch('app.database.berkeley.PostgreSQLToBerkeleyDB.create\_berkeley\_db') as mock\_create\_db:

migrator.migrate\_data()

# Проверяем, что создание Berkeley DB было вызвано с нужными аргументами

mock\_create\_db.assert\_called\_once\_with(

'table1', [('value1', 'value2')], ['col1', 'col2'], ['col1']

)

### **5.6.3 Результаты тестирования**

Все тесты были успешно пройдены (рисунок 5.2), что подтверждает корректность работы конвертера. Это обеспечивает высокую уверенность в том, что система может стабильно и правильно выполнять миграцию данных.

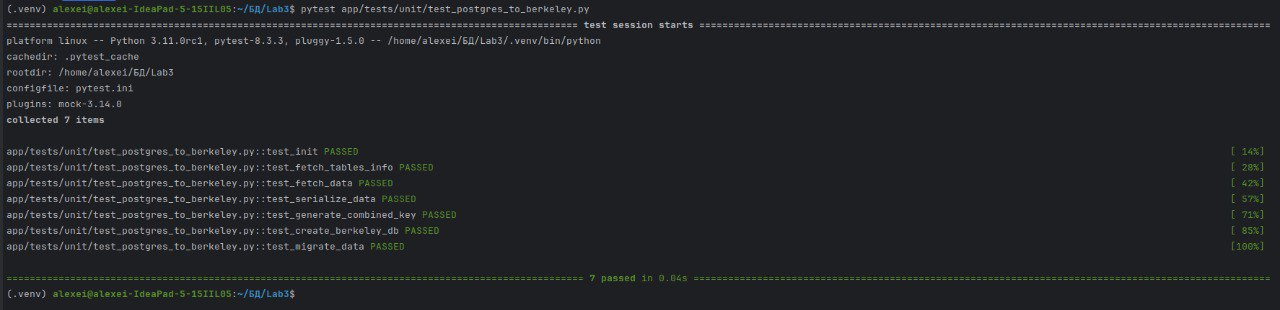


Рисунок 5.2 – Результаты тестирования конвертера

# **6 ПРИМЕРЫ ОПЕРАЦИЙ ВСТАВКИ, ЧТЕНИЯ, ОБНОВЛЕНИЯ И  УДАЛЕНИЯ ДАННЫХ (CRUD) В BERKELEY DB**

## **6.1 Развертывание Berkeley DB**

Для развертывания базы данных Berkeley DB и PostgreSQL использовался Docker, что значительно упрощает настройку и управление окружением. Docker предоставляет возможность контейнеризации, что позволяет разворачивать независимые и изолированные экземпляры приложений и баз данных на одном сервере.

Само приложение работает на Ubuntu, а для управления базами данных использовался Docker Compose, который позволяет легко развернуть несколько сервисов с помощью одного файла конфигурации.

Далее приведен текст файла docker-compose.yml:

version: '3.9'

services:

postgres:

container\_name: postgres\_db

image: postgres:16

env\_file:

- .env

environment:

POSTGRES\_DB: ${DB\_NAME}

POSTGRES\_USER: ${DB\_USER}

POSTGRES\_PASSWORD: ${DB\_PASS}

ports:

- "5000:5432"

volumes:

- postgres\_data:/var/lib/postgresql/data

- ./postgres/scripts:/docker-entrypoint-initdb.d

berkeleydb:

image: lncm/berkeleydb:db-4.8.30.NC

container\_name: berkeley\_db

volumes:

- berkeleydb\_data:/data

ports:

- "6000:6000"

tty: true

volumes:

postgres\_data:

berkeleydb\_data:

## **6.2 Реализация CRUD для Berkeley DB**

Для выполнения основных операций CRUD (Create, Read, Update, Delete) была реализована функциональность с использованием базы данных Berkeley DB. Для этого был создан класс BerkeleyDBManager, который инкапсулирует все необходимые операции для работы с этой базой данных, используя модуль bsddb3 для взаимодействия с Berkeley DB и модуль pickle для сериализации и десериализации данных.

Класс BerkeleyDBManager предоставляет методы для создания, чтения, обновления, удаления и извлечения всех записей из базы данных Berkeley DB. Каждый из этих методов работает с ключами и значениями, где ключи преобразуются в строковый формат, а значения сериализуются с помощью модуля pickle.

При инициализации класса создается имя файла базы данных на основе имени таблицы и путь к директории для хранения данных. Открывается база данных Berkeley DB с использованием метода hashopen, который создает или открывает существующую базу данных для чтения и записи.

Метод create(key, value) используется для добавления новой записи в базу данных. Он преобразует ключ в строковый формат и сериализует значение с помощью pickle. Далее сохраняет запись в базе данных с использованием ключа и сериализованного значения. В случае успеха выводится информационное сообщение, а в случае ошибки – сообщение об ошибке.

Для чтения записи по заданному ключу используется метод read(key). Он преобразует ключ в строковый формат и проверяет его наличие в базе данных. Если запись найдена, десериализует значение и возвращает его. Если же запись не найдена, выводится предупреждающее сообщение. Для чтения всех записей из базы данных используется read\_all().

Метод update(key, value) обновляет существующую запись в базе данных. Сначала он проверяет наличие записи с заданным ключом. Далее сериализует новое значение и обновляет запись в базе данных. В случае отсутствия записи выводится предупреждающее сообщение.

Для удаления записи по ключу используется метод delete(key). Он проверяет наличие записи с заданным ключом и удаляет ее из базы данных. Если запись не найдена, выводится предупреждающее сообщение.

В методе \_\_del\_\_() выполняется закрытие базы данных для освобождения ресурсов. При успешном закрытии базы данных выводится информационное сообщение.

Кроме вышеописанного функционала применяется логирование операций, обеспечивающее удобный механизм для отслеживания успешных и неудачных операций в базе данных. Логирование помогает быстро идентифицировать ошибки и отслеживать действия приложения.

## **6.3 Код класса BerkeleyDBManager**

import bsddb3

import pickle

from app.logger import logger

from .postgres\_to\_berkeley import PostgreSQLToBerkeleyDB

class BerkeleyDBManager:

PATH\_TO\_SAVE = PostgreSQLToBerkeleyDB.PATH\_TO\_SAVE

def \_\_init\_\_(self, table\_name: str):

"""Инициализация класса для работы с Berkeley DB."""

self.db\_name = f"{BerkeleyDBManager.PATH\_TO\_SAVE}/{table\_name}.db"

self.db = bsddb3.hashopen(self.db\_name, 'c')

def create(self, key, value):

"""Добавление записи в BerkeleyDB."""

try:

key\_encoded = str(key).encode()

value\_serialized = pickle.dumps(value)

self.db[key\_encoded] = value\_serialized

logger.info(

f"Запись с ключом {key} успешно добавлена."

)

except Exception as e:

logger.error(f"Ошибка при добавлении записи: {e}")

def read(self, key):

"""Чтение записи из BerkeleyDB по ключу."""

try:

key\_encoded = str(key).encode()

if key\_encoded in self.db:

value = pickle.loads(self.db[key\_encoded

return value

else:

logger.warning(

f"Запись с ключом {key} не найдена."

)

return None

except Exception as e:

logger.error(f"Ошибка при чтении записи: {e}")

return None

def update(self, key, value):

"""Обновление записи в BerkeleyDB по ключу."""

try:

key\_encoded = str(key).encode()

if key\_encoded in self.db:

value\_serialized = pickle.dumps(value)

self.db[key\_encoded] = value\_serialized

logger.info(

f"Запись с ключом {key} успешно обновлена."

)

else:

logger.warning(

f"Запись с ключом {key} не найдена."

)

except Exception as e:

logger.error(f"Ошибка при обновлении записи: {e}")

def delete(self, key):

"""Удаление записи из BerkeleyDB по ключу."""

try:

key\_encoded = str(key).encode()

if key\_encoded in self.db:

del self.db[key\_encoded]

logger.info(

f"Запись с ключом {key} успешно удалена."

)

else:

logger.warning(

f"Запись с ключом {key} не найдена."

)

except Exception as e:

logger.error(f"Ошибка при удалении записи: {e}")

def read\_all(self):

"""Чтение всех записей из BerkeleyDB."""

try:

all\_data = []

for key in self.db.keys():

value = pickle.loads(self.db[key])

all\_data.append((key.decode(), value))

return all\_data

except Exception as e:

logger.error(f"Ошибка при чтении всех записей: {e}")

return []

def \_\_del\_\_(self):

"""Закрытие соединения с базой данных BerkeleyDB."""

try:

self.db.close()

logger.info(

f"База данных {self.db\_name} успешно закрыта."

)

except Exception as e:

logger.error(f"Ошибка при закрытии базы данных: {e}")

## **6.4 Пример использования**

Для использования BerkeleyDBManager достаточно создать экземпляр класса, указав имя таблицы, а затем использовать методы для выполнения CRUD операций. Например:

from app.database.berkeley import BerkeleyDBManager

from app.schemas import StudentBase, GenderType

if \_\_name\_\_ == "\_\_main\_\_":

db\_manager = BerkeleyDBManager("student")

student = StudentBase(

id=66,

first\_name="Vasya",

last\_name="Pupkin",

email="<EMAIL>",

gender=GenderType,

class\_id=1

).model\_dump()

db\_manager.create(66, student)

# Пример чтения всех записей

all\_students = db\_manager.read\_all()

print("Все записи:")

for key, student in all\_students:

print(f"Ключ: {key}, Данные: {student}")

# Пример обновления данных

student['email'] = "vasya\_pupkin@mail.ru"

db\_manager.update(66, student)

# Чтение одной записи

student = db\_manager.read(66)

print(student)

# Пример удаления записи

db\_manager.delete(66)

# Пример чтения всех записей

all\_students = db\_manager.read\_all()

print("Все записи:")

for key, student in all\_students:

print(f"Ключ: {key}, Данные: {student}")

На рисунках 6.1 – 6.3 приведены результаты использования данного кода.

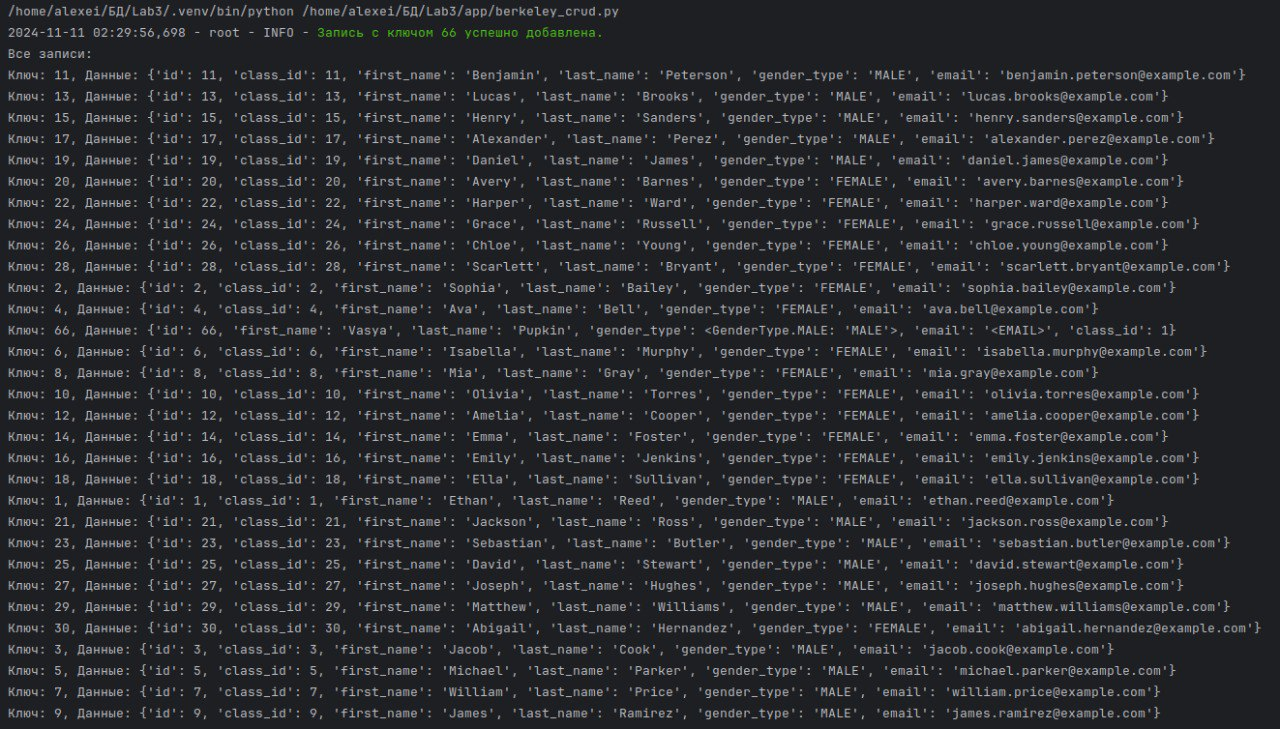


Рисунок 6.1 – Пример вставки данных в Berkeley DB



Рисунок 6.2 – Пример обновления данных в Berkeley DB

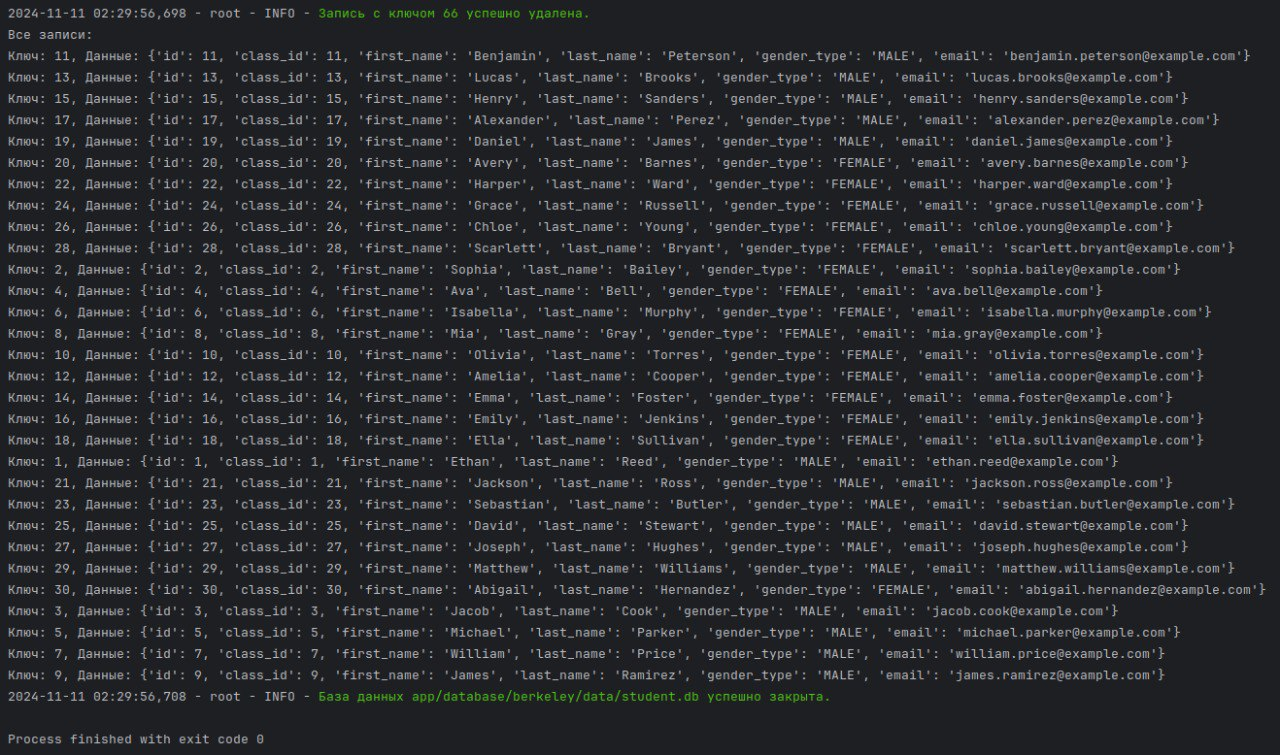


Рисунок 6.3 – Пример удаления данных из Berkeley DB

## **6.5 Тестирование**

Для тестирования класса BerkeleyDBManager использовался фреймворк pytest. Были написаны юнит-тесты для проверки основных операций с базой данных Berkeley DB, таких как создание, чтение, обновление, удаление записей и получение всех данных. В тестах применялись моки для взаимодействия с базой данных и логированием. Все тесты успешно пройдены, что подтверждает стабильную работу менеджера базы данных.

### **6.5.1 Описание тестов**

Для полного тестирования BerkeleyDBManager было написано 9 тестов:

1. Тест метода create – проверяет, что метод корректно создает запись в базе данных с сериализацией данных с использованием pickle. Тест также удостоверяется, что информация о созданной записи логируется.

2. Тест метода read, когда запись существует – мокируется существование записи, и проверяется, что метод успешно возвращает десериализованное значение.

3. Тест метода read, когда записи нет – тестирует поведение при попытке чтения несуществующей записи. Ожидается, что вернется None, а в лог будет записано предупреждение.

4. Тест метода update, когда запись существует – проверяет корректность обновления записи в базе данных и логирование этого действия.

5. Тест метода update, когда записи нет – удостоверяется, что при отсутствии записи в базе данных обновление не выполняется, а в лог добавляется предупреждение.

6. Тест метода delete, когда запись существует – проверяет успешное удаление записи из базы данных и логирование действия.

7. Тест метода delete, когда записи нет – удостоверяется, что при отсутствии записи удаление не выполняется, а в лог записывается предупреждение.

8. Тест метода read\_all – проверяет корректность чтения всех записей из базы данных, их десериализацию и возврат в виде списка.

9. Тест метода \_\_del\_\_, проверка закрытия базы – тестирует правильность закрытия базы данных при уничтожении экземпляра BerkeleyDBManager и логирование этого процесса.

### **6.5.2 Код для тестов**

import pytest

from unittest import mock

from app.database.berkeley.database import BerkeleyDBManager

import pickle

@pytest.fixture

def db\_manager():

# Создаем mock для базы данных

mock\_db = mock.MagicMock()

# Мокаем `bsddb3.hashopen`, чтобы возвращался наш mock-объект базы данных

with mock.patch('bsddb3.hashopen', return\_value=mock\_db):

manager = BerkeleyDBManager('test\_table')

yield manager

# Тест метода create

def test\_create(db\_manager):

with mock.patch('app.logger.logger.info') as mock\_logger\_info:

key, value = 'test\_key', {'field': 'test\_value'}

db\_manager.create(key, value)

db\_manager.db.\_\_setitem\_\_.assert\_called\_with(

b'test\_key', pickle.dumps(value)

)

mock\_logger\_info.assert\_called\_with(

f"Запись с ключом {key} успешно добавлена."

)

# Тест метода read, когда запись существует

def test\_read\_existing(db\_manager):

key, value = 'test\_key', {'field': 'test\_value'}

encoded\_key = key.encode()

# Мокаем возвращаемое значение и проверку наличия ключа

db\_manager.db.\_\_contains\_\_.return\_value = True

db\_manager.db.\_\_getitem\_\_.return\_value = pickle.dumps(value)

with mock.patch('app.logger.logger.warning') as mock\_logger\_warning:

result = db\_manager.read(key)

# Проверяем, что значение успешно десериализовано

assert result == value

mock\_logger\_warning.assert\_not\_called()

# Тест метода read, когда записи нет

def test\_read\_non\_existing(db\_manager):

db\_manager.db.\_\_contains\_\_.return\_value = False # Мокаем, что ключа нет

with mock.patch('app.logger.logger.warning') as mock\_logger\_warning:

result = db\_manager.read('non\_existing\_key')

# Проверяем, что вернется None, если ключ не найден

assert result is None

mock\_logger\_warning.assert\_called\_with(

"Запись с ключом non\_existing\_key не найдена."

)

# Тест метода update, когда запись существует

def test\_update\_existing(db\_manager):

key, value = 'test\_key', {'field': 'new\_value'}

db\_manager.db.\_\_contains\_\_.return\_value = True

with mock.patch('app.logger.logger.info') as mock\_logger\_info:

db\_manager.update(key, value)

# Проверяем, что значение обновлено

db\_manager.db.\_\_setitem\_\_.assert\_called\_with(b'test\_key', pickle.dumps(value))

mock\_logger\_info.assert\_called\_with(

f"Запись с ключом {key} успешно обновлена."

)

# Тест метода update, когда записи нет

def test\_update\_non\_existing(db\_manager):

key, value = 'test\_key', {'field': 'new\_value'}

db\_manager.db.\_\_contains\_\_.return\_value = False

with mock.patch('app.logger.logger.warning') as mock\_logger\_warning:

db\_manager.update(key, value)

# Проверяем, что запись не обновлена, если ключа нет

db\_manager.db.\_\_setitem\_\_.assert\_not\_called()

mock\_logger\_warning.assert\_called\_with(

f"Запись с ключом {key} не найдена."

)

# Тест метода delete, когда запись существует

def test\_delete\_existing(db\_manager):

key = 'test\_key'

db\_manager.db.\_\_contains\_\_.return\_value = True

with mock.patch('app.logger.logger.info') as mock\_logger\_info:

db\_manager.delete(key)

# Проверяем, что запись удалена

db\_manager.db.\_\_delitem\_\_.assert\_called\_with(

b'test\_key'

)

mock\_logger\_info.assert\_called\_with(

f"Запись с ключом {key} успешно удалена."

)

# Тест метода delete, когда записи нет

def test\_delete\_non\_existing(db\_manager):

key = 'non\_existing\_key'

db\_manager.db.\_\_contains\_\_.return\_value = False

with mock.patch('app.logger.logger.warning') as mock\_logger\_warning:

db\_manager.delete(key)

# Проверяем, что ничего не удалено, если ключа нет

db\_manager.db.\_\_delitem\_\_.assert\_not\_called()

mock\_logger\_warning.assert\_called\_with(

f"Запись с ключом {key} не найдена."

)

# Тест метода read\_all

def test\_read\_all(db\_manager):

# Мокаем ключи и значения

db\_manager.db.keys.return\_value = [b'test\_key1', b'test\_key2']

db\_manager.db.\_\_getitem\_\_.side\_effect = [pickle.dumps({'field': 'value1'}), pickle.dumps({'field': 'value2'})]

with mock.patch('app.logger.logger.error') as mock\_logger\_error:

result = db\_manager.read\_all()

# Проверяем, что все записи прочитаны и десериализованы

expected = [('test\_key1', {'field': 'value1'}), ('test\_key2', {'field': 'value2'})]

assert result == expected

mock\_logger\_error.assert\_not\_called()

# Тест метода \_\_del\_\_, проверка закрытия базы

def test\_del(db\_manager):

# Мокаем метод close для базы данных

with mock.patch.object(db\_manager.db, 'close', return\_value=None) as mock\_close, \

mock.patch('app.logger.logger.info') as mock\_logger\_info:

db\_manager.\_\_del\_\_()

# Проверяем, что метод close был вызван

mock\_close.assert\_called\_once()

# Проверяем, что запись в логгере выполнена

mock\_logger\_info.assert\_called\_with(f"База данных {db\_manager.db\_name} успешно закрыта.")

### **6.5.3 Результаты тестирования**

Тестирование класса BerkeleyDBManager показало, что все основные операции с базой данных работают корректно, а процессы создания, чтения, обновления и удаления данных логируются. Тесты успешно прошли (рисунок 6.4), что подтверждает надёжность класса для работы с базой данных Berkeley DB.

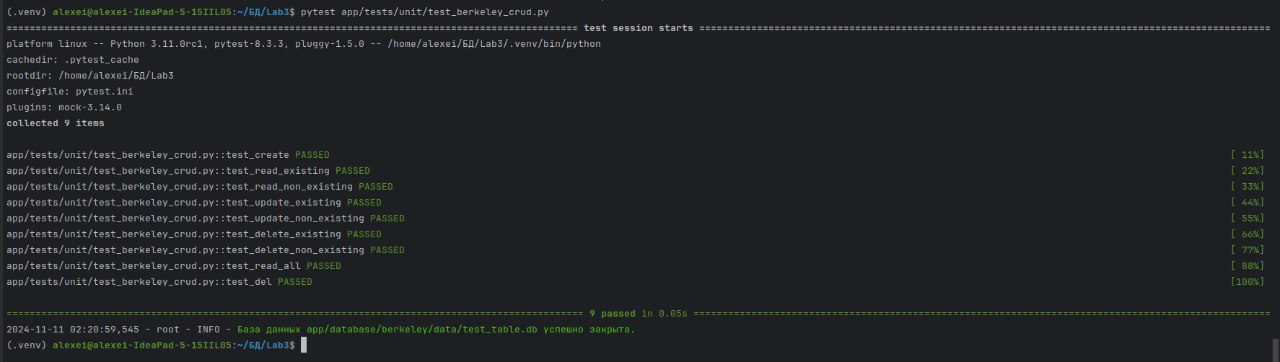


Рисунок 6.4 – Результаты тестирования класса BerkeleyDBManager

# **7 ОПИСАНИЕ ИЗМЕНЕНИЙ В СПЕЦИФИКАЦИЯХ  ПРИЛОЖЕНИЯ, АДАПТИРОВАННЫХ ДЛЯ РАБОТЫ С  BERKELEY DB**

В отличие от реляционных баз данных, таких как PostgreSQL, Berkeley DB не поддерживает внешние ключи и реляционные связи. В результате все связи между таблицами (например, «один ко многим» или «многие ко многим») теперь должны обрабатываться на уровне приложения. Это потребовало внесения изменений в бизнес-логику для управления целостностью данных и их согласованностью.

Berkeley DB не имеет встроенного языка запросов (SQL), что потребовало адаптации обработки запросов. Теперь для выборки данных с определёнными условиями необходимо реализовать логику загрузки и фильтрации на стороне приложения. Это привело к усложнению операций, таких как выборка данных из нескольких таблиц, поскольку все подобные действия выполняются вручную через код.

Berkeley DB не предоставляет встроенных индексов для ускорения поиска данных, как это делает PostgreSQL. Для эффективного поиска часто запрашиваемых данных приложение должно самостоятельно управлять индексированием. Это может включать создание вспомогательных структур данных, таких как словари или хэш-таблицы, для имитации индексной функциональности и повышения производительности.

Хотя Berkeley DB поддерживает транзакции и обеспечивает свойства ACID, управление транзакциями более ограничено по сравнению с реляционными базами данных. Настройка транзакционной обработки и механизмов отката операций (rollback) требует дополнительных усилий со стороны разработчиков. В спецификациях пришлось предусмотреть ручное управление транзакциями и настройку блокировок, что обеспечило сохранность данных в многопользовательской среде.

В Berkeley DB отсутствуют стандартные инструменты для администрирования, такие как автоматизированное резервное копирование и восстановление данных. Поэтому спецификации были дополнены процедурами для ручного резервного копирования и восстановления. Это включает копирование файлов базы данных и обеспечение целостности данных с учётом используемой сериализации. В случае восстановления данных, необходимо учитывать особенности формата хранения и обеспечивать корректную загрузку всех данных.

Эти дополнительные моменты подчеркивают не только необходимость адаптации к особенностям Berkeley DB, но и акцентируют внимание на важных аспектах производительности, сохранности данных и удобстве администрирования, которые отличаются от привычных реляционных баз данных.

# **8 ПРЕИМУЩЕСТВА И НЕДОСТАТКИ POSTGRESQL И  BERKELEY DB**

## **8.1 Преимущества PostgreSQL**

PostgreSQL поддерживает реляционные связи, внешние ключи, сложные запросы (JOIN), индексы, триггеры и представления, что делает её мощным инструментом для работы с структурированными данными.

Данная СУБД обеспечивает надёжную поддержку транзакций с соблюдением ACID-свойств (атомарность, согласованность, изоляция, долговечность), что критично для приложений, требующих высокую степень надёжности и защиты данных.

PostgreSQL поддерживает пользовательские типы данных, функции, языки программирования для процедур, что позволяет адаптировать её под специфические нужды проекта.

Также PostgreSQL имеет встроенную поддержку масштабирования, партиционирования таблиц, оптимизации запросов и индексации, что важно для работы с большими объёмами данных.

Кроме того, PostgreSQL имеет большое сообщество разработчиков, обширную документацию и множество сторонних инструментов для резервного копирования, мониторинга, администрирования и миграции.

## **8.2 Недостатки PostgreSQL**

Несмотря на мощные возможности, администрирование и настройка PostgreSQL могут быть сложными, особенно при необходимости работы с большими объёмами данных или оптимизации производительности.

PostgreSQL требует больше ресурсов (ОЗУ, процессор) по сравнению с простыми базами данных, такими как Berkeley DB, что может быть минусом для малоресурсных систем.

Некоторые операции могут быть медленными или неэффективными в SQL (например, массовая обработка данных или сложные аналитические запросы), что иногда требует дополнительной оптимизации.

## **8.3 Преимущества Berkeley DB**

Berkeley DB специализирована для быстрого выполнения операций с ключами и значениями, что делает её эффективной для приложений, где нужно обрабатывать большое количество простых операций чтения и записи.

В отличие от PostgreSQL, Berkeley DB может работать на устройствах с ограниченными вычислительными мощностями, что делает её идеальной для встраиваемых систем и IoT-устройств.

Berkeley DB работает в виде встроенной библиотеки, не требуя отдельного сервера для хранения данных, что упрощает установку и эксплуатацию.

Данная СУБД поддерживает транзакции и обеспечивает сохранность данных с использованием журнала транзакций, что критично для надёжности.

Благодаря модели ключ-значение, Berkeley DB предоставляет гибкость в выборе форматов данных, включая бинарные объекты, сериализованные структуры и другие нетипичные для реляционных СУБД форматы.

## **8.4 Недостатки Berkeley DB**

Berkeley DB не поддерживает реляционные связи, внешние ключи и сложные запросы. Все отношения между данными должны обрабатываться на уровне приложения, что усложняет работу с взаимосвязанными данными.

Также Berkeley DB не предоставляет язык SQL для работы с данными. Это означает, что выборки и фильтрация данных должны быть реализованы вручную в коде, что усложняет разработку при работе с большими и сложными структурами данных.

В Berkeley DB индексы не поддерживаются нативно. Для ускорения поиска разработчикам приходится реализовывать собственные механизмы индексации данных.

Berkeley DB предоставляет меньше инструментов для мониторинга состояния базы данных и администрирования по сравнению с PostgreSQL, что требует дополнительных усилий по созданию собственных решений для управления и наблюдения за базой.

В отличие от PostgreSQL, которая может работать в распределённых и кластерных системах, Berkeley DB больше ориентирована на использование в одиночных системах или встраиваемых устройствах, что ограничивает её использование в крупномасштабных распределённых системах.

## **8.5 Вывод**

PostgreSQL – мощная, полноценная реляционная база данных с обширными возможностями для работы с большими объёмами данных, сложными связями и запросами. Она идеально подходит для крупных проектов с высокими требованиями к целостности данных и поддержке сложных аналитических операций.

Berkeley DB, в свою очередь, ориентирована на простоту и производительность в сценариях, где не требуется сложная реляционная логика и высокие ресурсы. Её основное преимущество — быстрая обработка данных и низкие требования к ресурсам, что делает её подходящей для встраиваемых систем и приложений с простыми операциями. Однако отсутствие реляционной модели и встроенной поддержки SQL может стать серьёзным ограничением в сложных проектах.

# **ЗАКЛЮЧЕНИЕ**

В ходе лабораторной работы были выполнены ключевые шаги по преобразованию данных из реляционной модели PostgreSQL в формат ключ-значение, используемый в Berkeley DB. Этот процесс включал сериализацию данных, их запись в нереляционную базу данных и адаптацию бизнес-логики приложения для корректной работы с новым форматом хранения.

Анализ различных подходов к хранению данных показал, что Berkeley DB демонстрирует высокую эффективность при выполнении операций, связанных с быстрым доступом по ключам. Однако она ограничена в функциональности по сравнению с PostgreSQL, особенно при выполнении сложных запросов и работе с взаимосвязанными данными. Процесс адаптации продемонстрировал важность тщательного анализа и доработки логики приложения при переходе от реляционной к нереляционной базе данных для обеспечения производительности и целостности информации.

Выполненная работа позволила глубже понять различия между реляционными и нереляционными системами хранения данных, а также их применение в различных типах приложений. Berkeley DB подходит для сценариев, где важна простота доступа к данным и минимальные требования к ресурсам, в то время как PostgreSQL остаётся предпочтительным выбором для приложений с более сложными требованиями к связям между данными и выполнению запросов.