

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
2 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ	5
2.1 Задание	5
2.1.1 Извлечение схемы и данных из PostgreSQL	5
2.1.2 Конвертация данных в Berkeley DB.....	5
2.1.3 Запись данных в Berkeley DB	5
2.1.4 Адаптация спецификаций приложения	5
2.2 Требования к реализации	5
2.2.1 Подключение к PostgreSQL	5
2.2.2 Работа с Berkeley DB	5
2.2.3 Конвертация данных.....	5
3 ОПИСАНИЕ СТРУКТУРЫ ХРАНЕНИЯ ДАННЫХ В BERKELEY DB ...	6
4 ПРИМЕР SQL-ЗАПРОСОВ ДЛЯ ИЗВЛЕЧЕНИЯ СХЕМЫ И ДАННЫХ ИЗ POSTGRES SQL.....	7
4.1 Пример 1	7
4.2 Пример 2	7
4.3 Пример 3	7
4.4 Пример 4	7
5 КОД КОНВЕРТЕРА ДАННЫХ ИЗ POSTGRES SQL В BERKELEY DB, ВКЛЮЧАЯ СЕРИАЛИЗАЦИЮ И ДЕСЕРИАЛИЗАЦИЮ	8
5.1 Описание функционала	8
5.2 Используемые библиотеки.....	8
5.2.1 psycopg2	8
5.2.2 bsddb3	8
5.2.3 pickle.....	8
5.2.4 os	8
5.3 Описание ключевых методов.....	9
5.3.1 __init__	9
5.3.2 fetch_tables_info.....	9
5.3.3 fetch_data.....	9
5.3.4 serialize_data.....	9
5.3.5 generate_combined_key	9
5.3.6 create_berkeley_db	9
5.3.7 migrate_data.....	9
5.3.8 __del__	9
5.4 Код конвертера	10
5.5 Пример использования	12
5.6 Тестирование	13
5.6.1 Описание тестов.....	13
5.6.2 Код для тестов	14
5.6.3 Результаты тестирования	17

6 ПРИМЕРЫ ОПЕРАЦИЙ ВСТАВКИ, ЧТЕНИЯ, ОБНОВЛЕНИЯ И УДАЛЕНИЯ ДАННЫХ (CRUD) В BERKELEY DB	18
6.1 Развертывание Berkeley DB	18
6.2 Реализация CRUD для Berkeley DB	19
6.3 Код класса BerkeleyDBManager.....	20
6.4 Пример использования	22
6.5 Тестирование	24
6.5.1 Описание тестов.....	24
6.5.2 Код для тестов	24
6.5.3 Результаты тестирования	28
7 ОПИСАНИЕ ИЗМЕНЕНИЙ В СПЕЦИФИКАЦИЯХ ПРИЛОЖЕНИЯ, АДАПТИРОВАННЫХ ДЛЯ РАБОТЫ С BERKELEY DB	29
8 ПРЕИМУЩЕСТВА И НЕДОСТАТКИ POSTGRESQL И BERKELEY DB	30
8.1 Преимущества PostgreSQL.....	30
8.2 Недостатки PostgreSQL	30
8.3 Преимущества Berkeley DB	30
8.4 Недостатки Berkeley DB.....	31
8.5 Вывод.....	31
ЗАКЛЮЧЕНИЕ	32

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 student из 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

# Закрывать 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):
    """Закрывать 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 =
    'postgres://postgres:1234@localhost:5000/school'

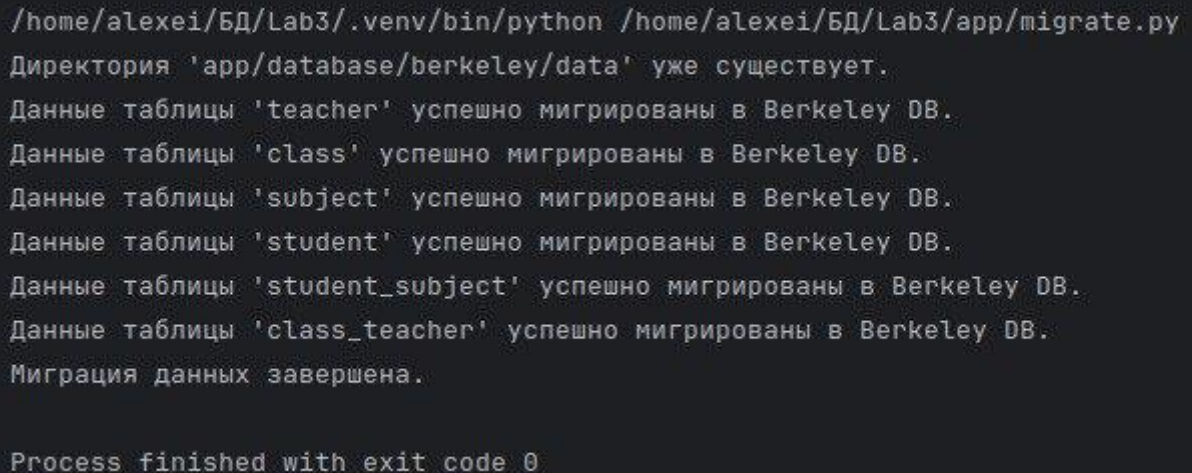
    # Создаем объект для миграции данных
    migration = PostgreSQLToBerkeleyDB(postgres_url)

```

```
# Мигрируем данные из PostgreSQL в Berkeley DB
migration.migrate_data()

print("Миграция данных завершена.")
```

На рисунке 5.1 приведен результат выполнения данного кода.



```
/home/alexei/БД/Lab3/.venv/bin/python /home/alexei/БД/Lab3/app/migrate.py
Директория 'app/database/berkeley/data' уже существует.
Данные таблицы 'teacher' успешно мигрированы в Berkeley DB.
Данные таблицы 'class' успешно мигрированы в Berkeley DB.
Данные таблицы 'subject' успешно мигрированы в Berkeley DB.
Данные таблицы 'student' успешно мигрированы в Berkeley DB.
Данные таблицы 'student_subject' успешно мигрированы в Berkeley DB.
Данные таблицы 'class_teacher' успешно мигрированы в Berkeley DB.
Миграция данных завершена.

Process finished with exit code 0
```

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.generat
e_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_T
O_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.

```

(.venv) alexei@alexei-IdeaPad-S-1511L05:~/6D/Lab3$ pytest app/tests/unit/test_postgres_to_berkeley.py
===== test session starts =====
platform linux -- Python 3.11.0rc1, pytest-8.3.3, pluggy-1.5.0 -- /home/alexei/6D/Lab3/.venv/bin/python
cachedir: .pytest_cache
rootdir: /home/alexei/6D/Lab3
configfile: pytest.ini
plugins: mock-3.14.0
collected 7 items

app/tests/unit/test_postgres_to_berkeley.py::test_init PASSED [ 14%]
app/tests/unit/test_postgres_to_berkeley.py::test_fetch_tables_info PASSED [ 28%]
app/tests/unit/test_postgres_to_berkeley.py::test_fetch_data PASSED [ 42%]
app/tests/unit/test_postgres_to_berkeley.py::test_serialize_data PASSED [ 57%]
app/tests/unit/test_postgres_to_berkeley.py::test_generate_combined_key PASSED [ 71%]
app/tests/unit/test_postgres_to_berkeley.py::test_create_berkeley_db PASSED [ 85%]
app/tests/unit/test_postgres_to_berkeley.py::test_migrate_data PASSED [100%]

===== 7 passed in 0.06s =====
(.venv) alexei@alexei-IdeaPad-S-1511L05:~/6D/Lab3$

```

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.

```

/home/alexei/БД/Lab3/.venv/bin/python /home/alexei/БД/Lab3/app/berkeley_crud.py
2024-11-11 02:29:56,698 - root - INFO - Запись с ключом 66 успешно добавлена.
Все записи:
Ключ: 11, Данные: {'id': 11, 'class_id': 11, 'first_name': 'Benjamin', 'last_name': 'Peterson', 'gender_type': 'MALE', 'email': 'benjamin.peterson@example.com'}
Ключ: 13, Данные: {'id': 13, 'class_id': 13, 'first_name': 'Lucas', 'last_name': 'Brooks', 'gender_type': 'MALE', 'email': 'lucas.brooks@example.com'}
Ключ: 15, Данные: {'id': 15, 'class_id': 15, 'first_name': 'Henry', 'last_name': 'Sanders', 'gender_type': 'MALE', 'email': 'henry.sanders@example.com'}
Ключ: 17, Данные: {'id': 17, 'class_id': 17, 'first_name': 'Alexander', 'last_name': 'Perez', 'gender_type': 'MALE', 'email': 'alexander.perez@example.com'}
Ключ: 19, Данные: {'id': 19, 'class_id': 19, 'first_name': 'Daniel', 'last_name': 'James', 'gender_type': 'MALE', 'email': 'daniel.james@example.com'}
Ключ: 20, Данные: {'id': 20, 'class_id': 20, 'first_name': 'Avery', 'last_name': 'Barnes', 'gender_type': 'FEMALE', 'email': 'avery.barnes@example.com'}
Ключ: 22, Данные: {'id': 22, 'class_id': 22, 'first_name': 'Harper', 'last_name': 'Ward', 'gender_type': 'FEMALE', 'email': 'harper.ward@example.com'}
Ключ: 24, Данные: {'id': 24, 'class_id': 24, 'first_name': 'Grace', 'last_name': 'Russell', 'gender_type': 'FEMALE', 'email': 'grace.russell@example.com'}
Ключ: 26, Данные: {'id': 26, 'class_id': 26, 'first_name': 'Chloe', 'last_name': 'Young', 'gender_type': 'FEMALE', 'email': 'chloe.young@example.com'}
Ключ: 28, Данные: {'id': 28, 'class_id': 28, 'first_name': 'Scarlett', 'last_name': 'Bryant', 'gender_type': 'FEMALE', 'email': 'scarlett.bryant@example.com'}
Ключ: 2, Данные: {'id': 2, 'class_id': 2, 'first_name': 'Sophia', 'last_name': 'Bailey', 'gender_type': 'FEMALE', 'email': 'sophia.bailey@example.com'}
Ключ: 4, Данные: {'id': 4, 'class_id': 4, 'first_name': 'Ava', 'last_name': 'Bell', 'gender_type': 'FEMALE', 'email': 'ava.bell@example.com'}
Ключ: 66, Данные: {'id': 66, 'first_name': 'Vasya', 'last_name': 'Pupkin', 'gender_type': <GenderType.MALE: 'MALE'>, 'email': '<EMAIL>', 'class_id': 1}
Ключ: 6, Данные: {'id': 6, 'class_id': 6, 'first_name': 'Isabella', 'last_name': 'Murphy', 'gender_type': 'FEMALE', 'email': 'isabella.murphy@example.com'}
Ключ: 8, Данные: {'id': 8, 'class_id': 8, 'first_name': 'Mia', 'last_name': 'Gray', 'gender_type': 'FEMALE', 'email': 'mia.gray@example.com'}
Ключ: 10, Данные: {'id': 10, 'class_id': 10, 'first_name': 'Olivia', 'last_name': 'Torres', 'gender_type': 'FEMALE', 'email': 'olivia.torres@example.com'}
Ключ: 12, Данные: {'id': 12, 'class_id': 12, 'first_name': 'Amelia', 'last_name': 'Cooper', 'gender_type': 'FEMALE', 'email': 'amelia.cooper@example.com'}
Ключ: 14, Данные: {'id': 14, 'class_id': 14, 'first_name': 'Emma', 'last_name': 'Foster', 'gender_type': 'FEMALE', 'email': 'emma.foster@example.com'}
Ключ: 16, Данные: {'id': 16, 'class_id': 16, 'first_name': 'Emily', 'last_name': 'Jenkins', 'gender_type': 'FEMALE', 'email': 'emily.jenkins@example.com'}
Ключ: 18, Данные: {'id': 18, 'class_id': 18, 'first_name': 'Ella', 'last_name': 'Sullivan', 'gender_type': 'FEMALE', 'email': 'ella.sullivan@example.com'}
Ключ: 1, Данные: {'id': 1, 'class_id': 1, 'first_name': 'Ethan', 'last_name': 'Reed', 'gender_type': 'MALE', 'email': 'ethan.reed@example.com'}
Ключ: 21, Данные: {'id': 21, 'class_id': 21, 'first_name': 'Jackson', 'last_name': 'Ross', 'gender_type': 'MALE', 'email': 'jackson.ross@example.com'}
Ключ: 23, Данные: {'id': 23, 'class_id': 23, 'first_name': 'Sebastian', 'last_name': 'Butler', 'gender_type': 'MALE', 'email': 'sebastian.butler@example.com'}
Ключ: 25, Данные: {'id': 25, 'class_id': 25, 'first_name': 'David', 'last_name': 'Stewart', 'gender_type': 'MALE', 'email': 'david.stewart@example.com'}
Ключ: 27, Данные: {'id': 27, 'class_id': 27, 'first_name': 'Joseph', 'last_name': 'Hughes', 'gender_type': 'MALE', 'email': 'joseph.hughes@example.com'}
Ключ: 29, Данные: {'id': 29, 'class_id': 29, 'first_name': 'Matthew', 'last_name': 'Williams', 'gender_type': 'MALE', 'email': 'matthew.williams@example.com'}
Ключ: 30, Данные: {'id': 30, 'class_id': 30, 'first_name': 'Abigail', 'last_name': 'Hernandez', 'gender_type': 'FEMALE', 'email': 'abigail.hernandez@example.com'}
Ключ: 3, Данные: {'id': 3, 'class_id': 3, 'first_name': 'Jacob', 'last_name': 'Cook', 'gender_type': 'MALE', 'email': 'jacob.cook@example.com'}
Ключ: 5, Данные: {'id': 5, 'class_id': 5, 'first_name': 'Michael', 'last_name': 'Parker', 'gender_type': 'MALE', 'email': 'michael.parker@example.com'}
Ключ: 7, Данные: {'id': 7, 'class_id': 7, 'first_name': 'William', 'last_name': 'Price', 'gender_type': 'MALE', 'email': 'william.price@example.com'}
Ключ: 9, Данные: {'id': 9, 'class_id': 9, 'first_name': 'James', 'last_name': 'Ramirez', 'gender_type': 'MALE', 'email': 'james.ramirez@example.com'}

```

Рисунок 6.1 – Пример вставки данных в Berkeley DB

```

2024-11-11 02:29:56,698 - root - INFO - Запись с ключом 66 успешно обновлена.
{'id': 9, 'class_id': 9, 'first_name': 'James', 'last_name': 'Ramirez', 'gender_type': 'MALE', 'email': 'vasya.pupkin@mail.ru'}

```

Рисунок 6.2 – Пример обновления данных в Berkeley DB

```

2024-11-11 02:29:56,698 - root - INFO - Запись с ключом 66 успешно удалена.
Все записи:
Ключ: 11, Данные: {'id': 11, 'class_id': 11, 'first_name': 'Benjamin', 'last_name': 'Peterson', 'gender_type': 'MALE', 'email': 'benjamin.peterson@example.com'}
Ключ: 13, Данные: {'id': 13, 'class_id': 13, 'first_name': 'Lucas', 'last_name': 'Brooks', 'gender_type': 'MALE', 'email': 'lucas.brooks@example.com'}
Ключ: 15, Данные: {'id': 15, 'class_id': 15, 'first_name': 'Henry', 'last_name': 'Sanders', 'gender_type': 'MALE', 'email': 'henry.sanders@example.com'}
Ключ: 17, Данные: {'id': 17, 'class_id': 17, 'first_name': 'Alexander', 'last_name': 'Perez', 'gender_type': 'MALE', 'email': 'alexander.perez@example.com'}
Ключ: 19, Данные: {'id': 19, 'class_id': 19, 'first_name': 'Daniel', 'last_name': 'James', 'gender_type': 'MALE', 'email': 'daniel.james@example.com'}
Ключ: 20, Данные: {'id': 20, 'class_id': 20, 'first_name': 'Avery', 'last_name': 'Barnes', 'gender_type': 'FEMALE', 'email': 'avery.barnes@example.com'}
Ключ: 22, Данные: {'id': 22, 'class_id': 22, 'first_name': 'Harper', 'last_name': 'Ward', 'gender_type': 'FEMALE', 'email': 'harper.ward@example.com'}
Ключ: 24, Данные: {'id': 24, 'class_id': 24, 'first_name': 'Grace', 'last_name': 'Russell', 'gender_type': 'FEMALE', 'email': 'grace.russell@example.com'}
Ключ: 26, Данные: {'id': 26, 'class_id': 26, 'first_name': 'Chloe', 'last_name': 'Young', 'gender_type': 'FEMALE', 'email': 'chloe.young@example.com'}
Ключ: 28, Данные: {'id': 28, 'class_id': 28, 'first_name': 'Scarlett', 'last_name': 'Bryant', 'gender_type': 'FEMALE', 'email': 'scarlett.bryant@example.com'}
Ключ: 2, Данные: {'id': 2, 'class_id': 2, 'first_name': 'Sophia', 'last_name': 'Bailey', 'gender_type': 'FEMALE', 'email': 'sophia.bailey@example.com'}
Ключ: 4, Данные: {'id': 4, 'class_id': 4, 'first_name': 'Ava', 'last_name': 'Bell', 'gender_type': 'FEMALE', 'email': 'ava.bell@example.com'}
Ключ: 6, Данные: {'id': 6, 'class_id': 6, 'first_name': 'Isabella', 'last_name': 'Murphy', 'gender_type': 'FEMALE', 'email': 'isabella.murphy@example.com'}
Ключ: 8, Данные: {'id': 8, 'class_id': 8, 'first_name': 'Mia', 'last_name': 'Gray', 'gender_type': 'FEMALE', 'email': 'mia.gray@example.com'}
Ключ: 10, Данные: {'id': 10, 'class_id': 10, 'first_name': 'Olivia', 'last_name': 'Torres', 'gender_type': 'FEMALE', 'email': 'olivia.torres@example.com'}
Ключ: 12, Данные: {'id': 12, 'class_id': 12, 'first_name': 'Amelia', 'last_name': 'Cooper', 'gender_type': 'FEMALE', 'email': 'amelia.cooper@example.com'}
Ключ: 14, Данные: {'id': 14, 'class_id': 14, 'first_name': 'Emma', 'last_name': 'Foster', 'gender_type': 'FEMALE', 'email': 'emma.foster@example.com'}
Ключ: 16, Данные: {'id': 16, 'class_id': 16, 'first_name': 'Emily', 'last_name': 'Jenkins', 'gender_type': 'FEMALE', 'email': 'emily.jenkins@example.com'}
Ключ: 18, Данные: {'id': 18, 'class_id': 18, 'first_name': 'Ella', 'last_name': 'Sullivan', 'gender_type': 'FEMALE', 'email': 'ella.sullivan@example.com'}
Ключ: 1, Данные: {'id': 1, 'class_id': 1, 'first_name': 'Ethan', 'last_name': 'Reed', 'gender_type': 'MALE', 'email': 'ethan.reed@example.com'}
Ключ: 21, Данные: {'id': 21, 'class_id': 21, 'first_name': 'Jackson', 'last_name': 'Ross', 'gender_type': 'MALE', 'email': 'jackson.ross@example.com'}
Ключ: 23, Данные: {'id': 23, 'class_id': 23, 'first_name': 'Sebastian', 'last_name': 'Butler', 'gender_type': 'MALE', 'email': 'sebastian.butler@example.com'}
Ключ: 25, Данные: {'id': 25, 'class_id': 25, 'first_name': 'David', 'last_name': 'Stewart', 'gender_type': 'MALE', 'email': 'david.stewart@example.com'}
Ключ: 27, Данные: {'id': 27, 'class_id': 27, 'first_name': 'Joseph', 'last_name': 'Hughes', 'gender_type': 'MALE', 'email': 'joseph.hughes@example.com'}
Ключ: 29, Данные: {'id': 29, 'class_id': 29, 'first_name': 'Matthew', 'last_name': 'Williams', 'gender_type': 'MALE', 'email': 'matthew.williams@example.com'}
Ключ: 30, Данные: {'id': 30, 'class_id': 30, 'first_name': 'Abigail', 'last_name': 'Hernandez', 'gender_type': 'FEMALE', 'email': 'abigail.hernandez@example.com'}
Ключ: 3, Данные: {'id': 3, 'class_id': 3, 'first_name': 'Jacob', 'last_name': 'Cook', 'gender_type': 'MALE', 'email': 'jacob.cook@example.com'}
Ключ: 5, Данные: {'id': 5, 'class_id': 5, 'first_name': 'Michael', 'last_name': 'Parker', 'gender_type': 'MALE', 'email': 'michael.parker@example.com'}
Ключ: 7, Данные: {'id': 7, 'class_id': 7, 'first_name': 'William', 'last_name': 'Price', 'gender_type': 'MALE', 'email': 'william.price@example.com'}
Ключ: 9, Данные: {'id': 9, 'class_id': 9, 'first_name': 'James', 'last_name': 'Ramirez', 'gender_type': 'MALE', 'email': 'james.ramirez@example.com'}
2024-11-11 02:29:56,708 - root - INFO - База данных app/database/berkeley/data/student.db успешно закрыта.
Process finished with exit code 0

```

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__()

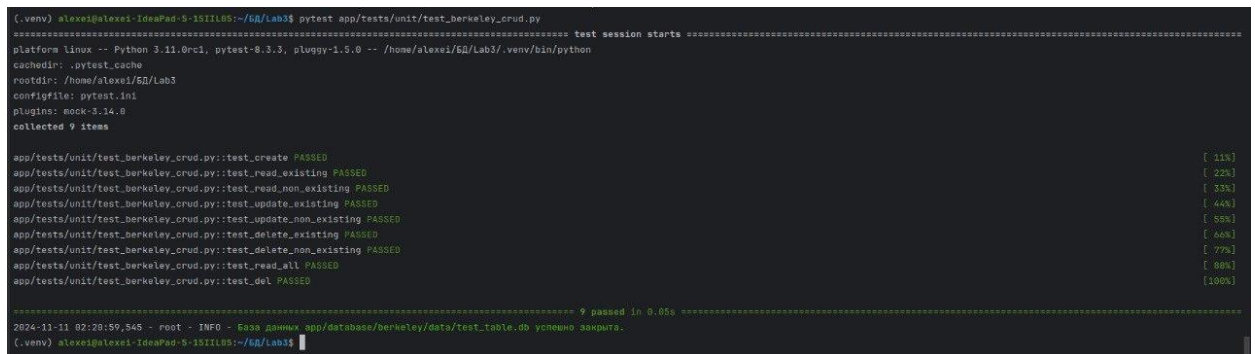
# Проверяем, что метод close был вызван
mock_close.assert_called_once()

# 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.



```

(.venv) alexei@alexei-IdeaPad-5-1511LBS:~/G/Lab3$ pytest app/tests/unit/test_berkeley_crud.py
===== test session starts =====
platform linux -- Python 3.11.0rc1, pytest-8.3.3, pluggy-1.5.0 -- /home/alexei/G/Lab3/.venv/bin/python
cachedir: .pytest.cache
rootdir: /home/alexei/G/Lab3
configfile: pytest.ini
plugins: mock-3.14.0
collected 9 items

app/tests/unit/test_berkeley_crud.py::test_create PASSED [ 11%]
app/tests/unit/test_berkeley_crud.py::test_read_existing PASSED [ 22%]
app/tests/unit/test_berkeley_crud.py::test_read_non_existing PASSED [ 33%]
app/tests/unit/test_berkeley_crud.py::test_update_existing PASSED [ 44%]
app/tests/unit/test_berkeley_crud.py::test_update_non_existing PASSED [ 55%]
app/tests/unit/test_berkeley_crud.py::test_delete_existing PASSED [ 66%]
app/tests/unit/test_berkeley_crud.py::test_delete_non_existing PASSED [ 77%]
app/tests/unit/test_berkeley_crud.py::test_read_all PASSED [ 88%]
app/tests/unit/test_berkeley_crud.py::test_del PASSED [100%]

===== 9 passed in 0.05s =====
2024-11-11 02:29:59,545 - root - INFO - База данных app/database/berkeley/data/test_table.db успешно закрыта.
(.venv) alexei@alexei-IdeaPad-5-1511LBS:~/G/Lab3$

```

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.