**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский Авиационный Институт»**

**(Национальный Исследовательский Университет)**

**Институт: №8 «Информационные технологии   
и прикладная математика»   
Кафедра: 806 «Вычислительная математика   
и программирование»**

Лабораторная работа № 1   
по курсу «Базы данных»

Группа: М8О-307Б-22

Студент: Е. С. Кострюков

Преподаватель: А. В. Малахов

Оценка:

Дата: 06.05.2025

Москва, 2025

# Ход лабораторной работы

Схема базы данных, которая используется во всех пунктах лабораторной

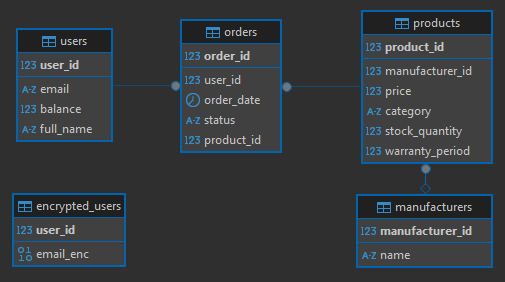


Таблица encrypted\_users была создана в последнем пункте лабораторной работы для демонстрации шифрования пользователей с помощью расширения pgcrypto.

**1.1: Типы индексов и их использование на практике**

B-TREE

Для ускорения выполнения запросов по числовым значениям, в частности по цене товаров, был создан B-Tree индекс. Он основан на структуре B-деревьев и применяется для диапазонных запросов и точного поиска по значениям.

Выполнено сравнение одного и того же SQL-запроса с отключёнными и включёнными индексами.

Запрос:

*SELECT COUNT(\*) FROM products WHERE price BETWEEN 250 AND 500*

Результаты:

*=== Benchmark: без индексов ===*

*Price Range: 0.1678s (rows=26455)*

*=== Benchmark: с индексами ===*

*Price Range: 0.0820s (rows=26455)*

Как видно, индекс позволил сократить время выполнения запроса почти в 2 раза. Это особенно важно при больших объёмах данных, где полный перебор строк занимает значительно больше ресурсов.

**BRIN**

BRIN (Block Range Index) особенно эффективен при работе с большими таблицами, где значения поля (например, даты) отсортированы или хотя бы частично упорядочены. Этот индекс позволяет быстро определить, в каких блоках страницы данных могут находиться строки с нужными значениями.

Запрос:

*SELECT COUNT(\*) FROM orders WHERE order\_date BETWEEN '2019-11-05' AND '2019-11-06'*

Результаты:

*=== Benchmark: без индексов ===*

*Date Range: 1.2000s (rows=41950)*

*=== Benchmark: с индексами ===*

*Date Range: 0.0620s (rows=41950)*

Использование BRIN позволило значительно ускорить выполнение запроса, снизив время выполнения примерно в 19 раз. При этом таблица содержала десятки тысяч записей и не была строго отсортирована по дате, что подчёркивает эффективность индекса.

**GIN**

GIN (Generalized Inverted Index) используется для полнотекстового поиска по текстовым полям, например по *full\_name* в таблице *users*. Индексы этого типа позволяют быстро находить строки, содержащие нужные слова, путём построения обратного индекса.

В данном случае были проведены два разных запроса:

* Без использования индекса (векторизация текста выполнялась после ограничения выборки через LIMIT);
* С использованием индекса (условие @@ применялось до LIMIT, позволяя PostgreSQL задействовать GIN).

Запрос с индексами:

*SELECT COUNT(\*) FROM (*

*SELECT \* FROM users*

*WHERE full\_name IS NOT NULL*

*AND to\_tsvector('english', full\_name) @@ plainto\_tsquery('english', 'Anonymous')*

*LIMIT 250000*

*) AS limited;*

Запрос без индексов:

*SELECT COUNT(\*) FROM (*

*SELECT full\_name FROM users*

*WHERE full\_name IS NOT NULL*

*AND length(full\_name) < 1000*

*LIMIT 250000*

*) AS limited*

*WHERE to\_tsvector('english', full\_name) @@ plainto\_tsquery('english', 'Anonymous');*

Результаты:

*=== Benchmark: без индексов ===*

*FullName GIN Search: 3.5963s (rows=250000)*

*=== Benchmark: с индексами ===*

*FullName GIN Search: 2.4132s (rows=250000)*

Хотя в обоих случаях применялись LIMIT, правильное использование индекса (с фильтрацией до ограничения выборки) дало заметный прирост производительности. Без индексов выполнение запроса было 1.5 раза медленнее и обрабатывало больше строк.

**Выводы**

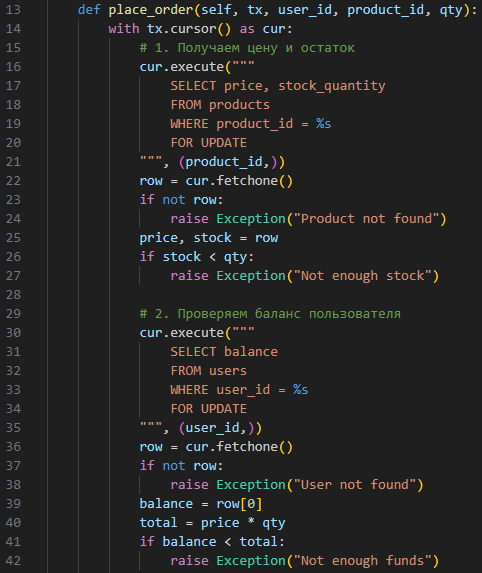
Проведённые тесты подтверждают, что применение индексов существенно снижает время выполнения запросов. Особенно это заметно на больших объёмах данных. При этом важно правильно формулировать запросы: структура SQL-выражения может повлиять на выбор PostgreSQL использовать индекс или нет. Создание самих индексов может занять некоторое время, но результат в производительности того стоит.

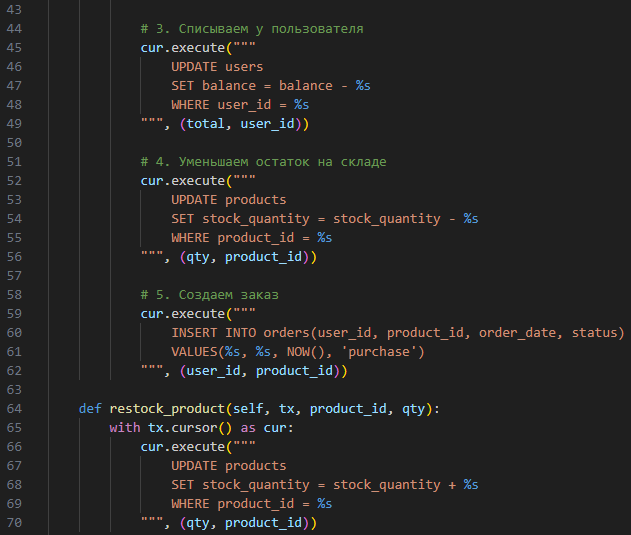
**1.2: Транзакции в PostgreSQL: виды и использование на**

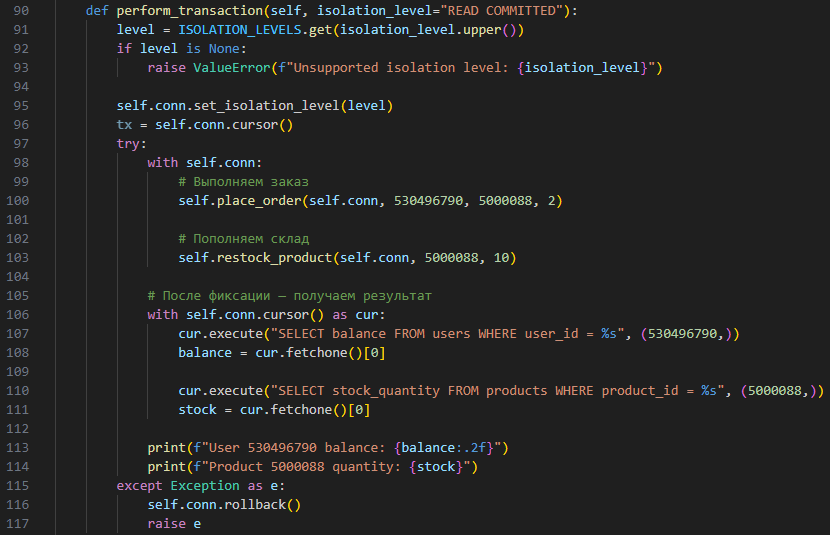
**Практике**

Транзакции — это механизм, обеспечивающий целостность и согласованность данных при выполнении нескольких операций в базе данных. Они позволяют гарантировать, что все действия внутри транзакции будут выполнены полностью, либо не будут применены вовсе. Если в процессе выполнения одной из операций возникнет ошибка, вся цепочка изменений будет отменена — это называется откатом (rollback). Таким образом, БД защищается от частично выполненных изменений, которые могли бы привести к неконсистентному состоянию данных.

В лабораторной работе реализован пример такой транзакции (ниже приведён скриншот из кода), где последовательно выполняется несколько SQL-запросов. Каждый шаг сопровождается проверкой на успешность выполнения. В случае сбоя на любом из этапов, происходит автоматический откат. Если же всё завершается успешно, изменения фиксируются в базе с помощью команды commit.



****

****

**Уровни изоляции транзакций**

1. **Read Uncommitted**

Самый низкий уровень изоляции. Транзакция может читать данные, которые ещё не были зафиксированы другими транзакциями (так называемые «грязные» данные). Такой подход потенциально небезопасен и практически не используется в PostgreSQL (его поведение аналогично Read Committed).

1. **Read Committed**

Транзакция может читать только те данные, которые уже были зафиксированы на момент начала каждого отдельного запроса. Это предотвращает чтение «грязных» данных, но допускает ситуацию, когда при повторном чтении данных они уже могут измениться.

1. **Repeatable Read**

Повышенный уровень изоляции. Все данные, считанные в рамках одной транзакции, сохраняются в неизменном виде до её завершения. Это исключает как грязное чтение, так и "неповторяемое чтение" — ситуацию, когда одни и те же строки возвращают разные результаты в рамках одной транзакции.

1. **Serializable**

Наиболее строгий уровень. Все транзакции выполняются так, будто они идут строго последовательно, даже если они исполняются параллельно. Это исключает все возможные аномалии: грязное чтение, неповторяемое чтение и фантомное чтение. В PostgreSQL этот уровень реализован через контроль версий (MVCC) и механизм блокировок. При обнаружении конфликта PostgreSQL может принудительно откатить одну из транзакций.

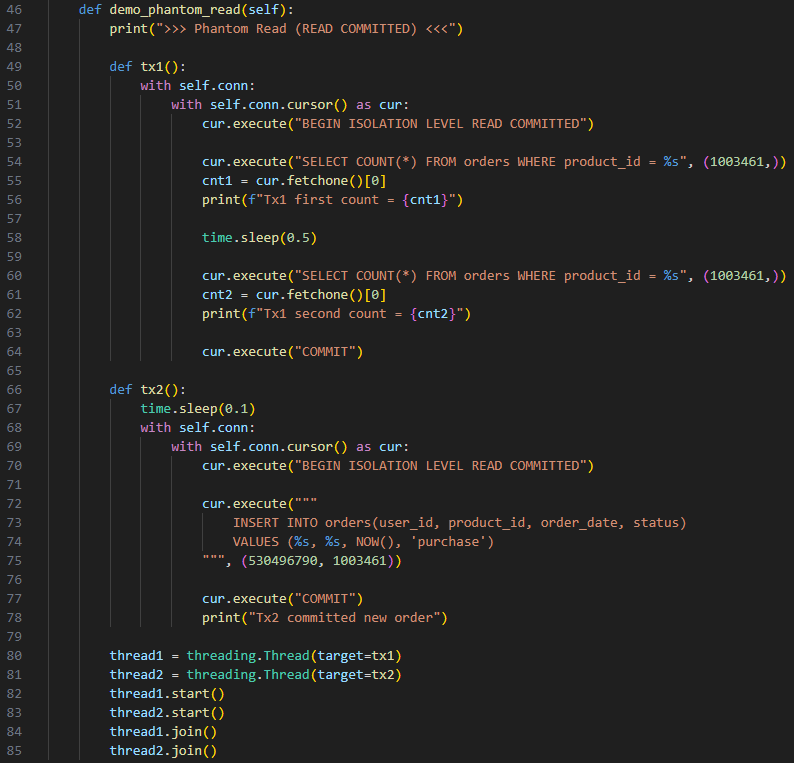
**Аномалии параллельного выполнения**

При недостаточной изоляции транзакций могут возникать **аномалии** — непредвиденные или некорректные результаты работы запросов. Некоторые из наиболее распространённых:

* **Dirty read (грязное чтение)** — одна транзакция читает данные, которые были изменены, но ещё не зафиксированы другой транзакцией. Если та будет отменена, прочитанные данные окажутся недействительными.
* **Non-repeatable read (неповторяемое чтение)** — при повторном чтении одних и тех же данных транзакция получает разные результаты, потому что другая транзакция успела их изменить.
* **Phantom read (фантомное чтение)** — при повторном выполнении запроса с одинаковым условием появляются новые строки, добавленные другой транзакцией.

Такие эффекты можно предотвратить только с помощью правильного выбора уровня изоляции. PostgreSQL по умолчанию использует уровень Read Committed, который обеспечивает базовую защиту, но не исключает всех возможных аномалий.

Пример фантомного чтения:



**1.3: Использование расширений PostgreSQL для**

**полнотекстового поиска и криптографических операций**

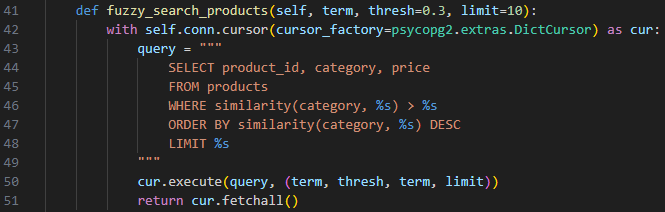
В рамках последнего этапа лабораторной работы были установлены и применены следующие расширения PostgreSQL:

* **pg\_bigm** — обеспечивает быстрый полнотекстовый поиск, используя биграммы (двухсимвольные комбинации).
* **pg\_trgm** — реализует поиск на основе триграмм, что делает его особенно полезным для "неточного" поиска, поиска с ошибками и автодополнения.
* **pgcrypto** — предоставляет функции для шифрования, дешифрования и хеширования данных прямо на уровне базы данных.

**pg\_bigm и pg\_trgm**

Оба расширения направлены на оптимизацию текстового поиска, но делают это разными способами.

Пример кода поиска с расширениями:



**pg\_bigm** опирается на разбиение текста на биграммы — пары символов. Это расширение особенно эффективно при использовании SQL-запросов с шаблонами типа LIKE '%подстрока%', которые в стандартной конфигурации не используют индексы и, следовательно, работают медленно. Благодаря индексам на основе pg\_bigm, такие запросы выполняются значительно быстрее.

Наибольшую выгоду от pg\_bigm получают сценарии, где требуется точный поиск по подстроке в длинных текстах. Особенно хорошо оно себя показывает при работе с языками, в которых слова не разделяются пробелами, например, китайским или японским.

Основные недостатки — отсутствие поддержки поиска по "похожести" (fuzzy search) и высокий объём создаваемых индексов, что может повлиять на размер базы данных.

В отличие от него, **pg\_trgm** использует триграммы — трехсимвольные фрагменты строк. Это расширение идеально подходит для реализации поиска по частичному совпадению, автодополнению, а также для поиска с допущением опечаток.

Оно поддерживает не только LIKE, но и специализированные операторы (%, <->) и функцию similarity(), которая позволяет определить степень сходства между строками. Это делает pg\_trgm незаменимым для интерфейсов с пользовательским вводом (например, поиск по имени или почте с ошибкой в написании).

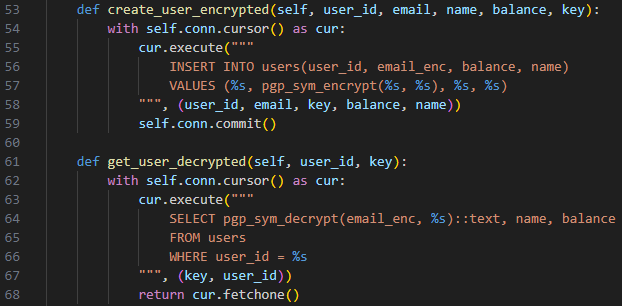
Из минусов — несколько меньшая производительность на больших объемах текста по сравнению с pg\_bigm, а также увеличение размера базы данных за счёт триграммных индексов.

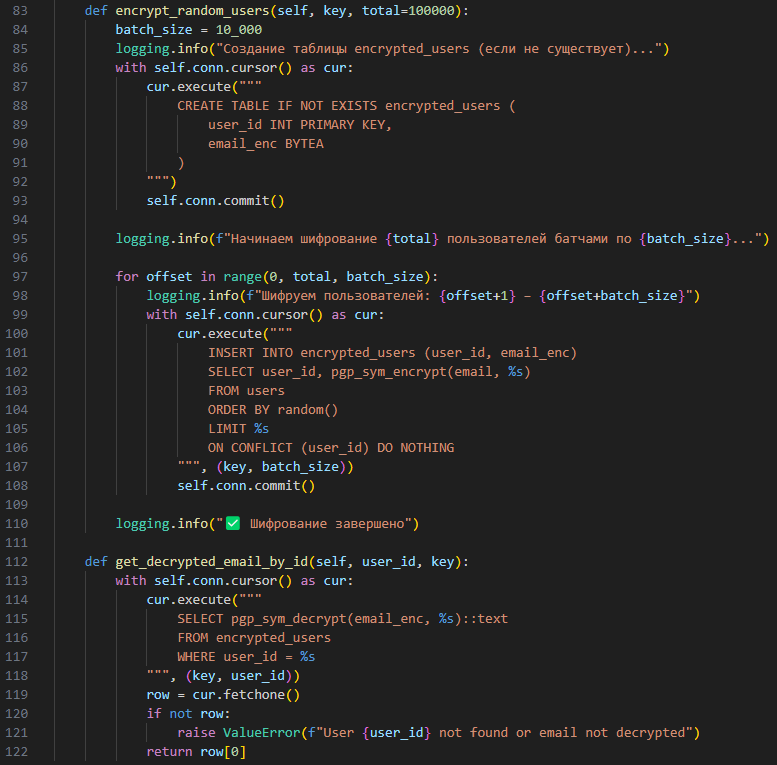
**Pgcrypto**

Для защиты конфиденциальной информации, в частности, email-адресов пользователей, использовалось расширение pgcrypto, которое реализует встроенные криптографические функции.

Основные методы:

* **CreateUserEncrypted** — при создании пользователя email шифруется функцией pgp\_sym\_encrypt(email, key) и сохраняется в таблице. Таким образом, адрес не хранится в открытом виде.
* **GetUserDecrypted** — позволяет извлечь зашифрованный email и расшифровать его при помощи pgp\_sym\_decrypt(email\_enc, key)::text, если известен ключ.
* **EncryptRandomUsers** — выборка случайных пользователей из основной таблицы, шифрование их адресов и сохранение результата в отдельной таблице encrypted\_users. Это используется для демонстрации массового шифрования.





Расширение pgcrypto является удобным средством реализации шифрования на уровне базы данных и обеспечивает базовую защиту персональных данных. При этом разработчику необходимо самому управлять безопасным хранением ключей, а также учитывать возможное влияние шифрования на производительность.