**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

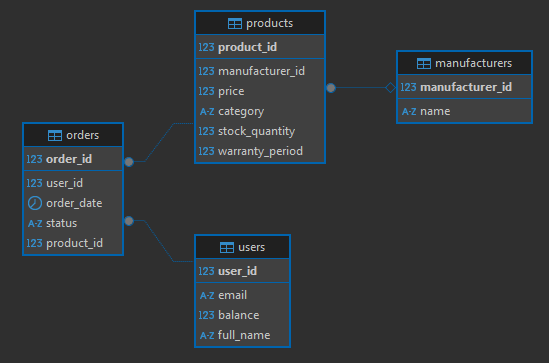
Оценка:

Дата: 06.05.2025

Москва, 2025

# Ход лабораторной работы

Схема базы данных, которая используется во всех пунктах лабораторной:



**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)*

**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.

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

**Практике**

Транзакции – технология, которая обеспечивает атомарность действий. То есть защищает БД от случаев, в которых часть запросов выполнилась, а часть – нет. Транзакция откатывает все изменения в БД если один из запросов к БД прошел неудачно.   
Приведем пример одной из транзакций, реализованных в лабораторной работе.

func PlaceOrder(tx \*sql.Tx, userID, productID, qty int) error {

    // 1. Получаем цену и остаток

    var price float64

    var stock int

    err := tx.QueryRow(

        `SELECT price, stock\_quantity

           FROM products

          WHERE product\_id = $1

            FOR UPDATE`, productID,

    ).Scan(&price, &stock)

    if err != nil {

        return err

    }

    if stock < qty {

        return fmt.Errorf("not enough stock")

    }

    // 2. Проверяем баланс

    var balance float64

    err = tx.QueryRow(

        `SELECT balance

           FROM users

          WHERE user\_id = $1

            FOR UPDATE`, userID,

    ).Scan(&balance)

    if err != nil {

        return err

    }

    total := price \* float64(qty)

    if balance < total {

        return fmt.Errorf("not enough funds")

    }

    // 3. Списываем у пользователя

    if \_, err = tx.Exec(

        `UPDATE users

            SET balance = balance - $1

          WHERE user\_id = $2`, total, userID,

    ); err != nil {

        return err

    }

    // 4. Урезаем склад

    if \_, err = tx.Exec(

        `UPDATE products

            SET stock\_quantity = stock\_quantity - $1

          WHERE product\_id = $2`, qty, productID,

    ); err != nil {

        return err

    }

    // 5. Вставляем заказ

    \_, err = tx.Exec(

        `INSERT INTO orders(user\_id, product\_id, order\_date, status)

             VALUES($1, $2, NOW(), 'purchase')`,

        userID, productID,

    )

    return err

}

func RestockProduct(tx \*sql.Tx, productID int, qty int) error {

    \_, err := tx.Exec(`

        UPDATE products

        SET stock\_quantity = stock\_quantity + $1

        WHERE product\_id = $2

    `, qty, productID)

    return err

}

func (s \*Store) PerformTransaction(isolationLevel string) error {

    // Начинаем транзакцию

    tx, err := s.DB.Begin()

    if err != nil {

        return err

    }

    // Устанавливаем уровень изоляции внутри транзакции

    \_, err = tx.Exec(fmt.Sprintf("SET TRANSACTION ISOLATION LEVEL %s", isolationLevel))

    if err != nil {

        tx.Rollback()

        return err

    }

    // Пример 1: Покупка товара (PlaceOrder)

    err = PlaceOrder(tx, 530496790 /\*user\*/, 5000088 /\*product\*/, 2 /\*qty\*/)

    if err != nil {

        tx.Rollback()

        return err

    }

    // Пример 2: Пополнение товара (RestockProduct)

    err = RestockProduct(tx, 5000088 /\*product\*/, 10 /\*добавить\*/)

    if err != nil {

        tx.Rollback()

        return err

    }

    // Подтверждение транзакции

    if err := tx.Commit(); err != nil {

        return err

    }

    // Выводим результат: баланс пользователя и остаток товара

    var balance float64

    err = s.DB.QueryRow("SELECT balance FROM users WHERE user\_id = $1", 530496790).Scan(&balance)

    if err != nil {

        return err

    }

    var quantity int

    err = s.DB.QueryRow("SELECT stock\_quantity FROM products WHERE product\_id = $1", 5000088).Scan(&quantity)

    if err != nil {

        return err

    }

    fmt.Printf("User 530496790 balance: %.2f\n", balance)

    fmt.Printf("Product 5000088 quantity: %d\n", quantity)

    return nil

}

В данной транзакции есть несколько условий проверяющих успешность выполнения запросов. Если хотя бы часть транзакции не пройдет успешно, то произойдет rollback и действия отменятся, а если все пройдет успешно то в конце функции все изменения закоммитятся в БД.

**Виды изоляции**

1. Read uncommitted – чтение не подтвержденных данных, самый низкий уровень, позволяет читать данные через БД, которые другая транзакция еще не закоммитила.
2. Read commited – чтение подтвержденных данных, то есть читать можно только те данные, которые были подтверждены другой транзакцией.
3. Repeatable read – запрещает “грязное чтение” и “non-repeatable read”, когда транзакция читает дважды одни и те же данные и между этими чтениями другая транзакция успевает их изменить. Все строки которые читаются в транзакции останутся неизменными до конца транзакции.
4. Serializable – самый жесткий уровень безопасности. Dirty reads, non-repeatable reads, phantom reads невозможны. Каждая транзакция работает со снимком данных на момент ее начала, postgresql следит за возможными конфликтами чтения и записи. Так же если обнаружится что параллельное выполнение транзакций может нарушить логику, psql откатит одну из транзакций с ошибкой.

**Аномалии**

Аномалии – явления, которые могут привести к получению неверных данных вследствие работы с неправильными уровнями изоляции. К примеру – чтение незакоммиченных данных одной транзакцией может привести к ошибкам, если другая транзакция изменит данные перед коммитом.

Пример фантомного чтения:  
// DemoPhantomRead показывает фантомные строки: COUNT(\*) меняется внутри одной транзакции.

func (s \*Store) DemoPhantomRead() {

    fmt.Println(">>> Phantom Read (READ COMMITTED) <<<")

    var wg sync.WaitGroup

    wg.Add(2)

    // Tx1

    go func() {

        defer wg.Done()

        tx1, err := s.DB.BeginTx(context.Background(), &sql.TxOptions{Isolation: sql.LevelReadCommitted})

        if err != nil {

            log.Println("Tx1 begin:", err)

            return

        }

        defer tx1.Rollback()

        var cnt1, cnt2 int

        if err := tx1.QueryRow("SELECT COUNT(\*) FROM orders WHERE product\_id = $1", 1003461).Scan(&cnt1); err != nil {

            log.Println("Tx1 first count:", err)

            return

        }

        fmt.Printf("Tx1 first count = %d\n", cnt1)

        time.Sleep(500 \* time.Millisecond) // ждём, пока Tx2 вставит новую строку

        if err := tx1.QueryRow("SELECT COUNT(\*) FROM orders WHERE product\_id = $1", 1003461).Scan(&cnt2); err != nil {

            log.Println("Tx1 second count:", err)

            return

        }

        fmt.Printf("Tx1 second count = %d\n", cnt2)

        tx1.Commit()

    }()

    // Tx2

    go func() {

        defer wg.Done()

        time.Sleep(100 \* time.Millisecond) // даём Tx1 выполнить первый COUNT

        tx2, err := s.DB.BeginTx(context.Background(), &sql.TxOptions{Isolation: sql.LevelReadCommitted})

        if err != nil {

            log.Println("Tx2 begin:", err)

            return

        }

        defer tx2.Rollback()

        // здесь ставим свой user\_id

        if \_, err := tx2.Exec(

            `INSERT INTO orders(user\_id, product\_id, order\_date, status)

             VALUES($1, $2, NOW(), 'purchase')`,

            530496790, // ваш user\_id

            1003461,   // product\_id

        ); err != nil {

            log.Println("Tx2 insert:", err)

            return

        }

        tx2.Commit()

        fmt.Println("Tx2 committed new order")

    }()

    wg.Wait()

}

**№3**

В рамках лабораторной работы были установлены и задействованы следующие расширения:

* **pg\_bigm** — используется для быстрого полнотекстового поиска с поддержкой биграмм.
* **pg\_trgm** — основан на триграммах, подходит для поиска по схожим строкам (например, для поиска с ошибками или по неполному вводу).
* **pgcrypto** — используется для шифрования/хеширования данных, таких как личная информация о пользователях, пароли и другая ценная информация.

**pg\_bigm и pg\_trgm**

Рассмотрим пример кода с поиском с расширениями

func (s \*Store) FuzzySearchProducts(

    ctx context.Context,

    term string,

    method string,

    thresh float64,

    limit int,

) ([]Product, error) {

    sqlQuery := fmt.Sprintf(

        `SELECT product\_id, category, price FROM products WHERE similarity(category, $1) > $2 ORDER BY similarity(category, $1) DESC LIMIT $3`,

    )

    rows, err := s.DB.QueryContext(ctx, sqlQuery, term, thresh, limit)

    if err != nil {

        return nil, err

    }

    defer rows.Close()

    var res []Product

    for rows.Next() {

        var p Product

        if err := rows.Scan(&p.ID, &p.Category, &p.Price); err != nil {

            return nil, err

        }

        res = append(res, p)

    }

    return res, nil

}

Оба расширения — pg\_bigm и pg\_trgm — предназначены для реализации полнотекстового поиска в PostgreSQL, однако работают они с использованием разных принципов и используются в разных сценариях.

Расширение **pg\_bigm** реализует поиск на основе биграмм — то есть, сочетаний двух символов, которые извлекаются из текста. Основная цель pg\_bigm — ускорить поиск по подстрокам с использованием стандартного оператора LIKE, особенно в случае выражений вида LIKE '%слово%', которые по умолчанию не используют индекс и работают медленно. Установив индекс на основе pg\_bigm, можно значительно ускорить такие запросы.

pg\_bigm показывает наилучшую производительность при работе с длинными текстами и точными совпадениями. Также оно особенно эффективно для языков, не использующих пробелы между словами (например, китайского или японского).

Однако у pg\_bigm есть и недостатки: оно не поддерживает поиск по "похожести" или с допущением опечаток. Кроме того, индексы, создаваемые этим расширением, могут занимать значительный объём памяти.

Расширение **pg\_trgm**, в отличие от pg\_bigm, использует триграммы — трёхсимвольные последовательности, полученные из текста. Оно особенно полезно в ситуациях, когда нужно реализовать поиск по части слова, автодополнение, либо fuzzy-поиск — то есть поиск с допущением ошибок или неточного ввода.

pg\_trgm поддерживает не только ускорение LIKE-запросов, но и специальные операторы, такие как % (поиск по похожести), <-> (расстояние между строками), а также функцию similarity(), которая позволяет измерять степень сходства между строками. Это делает расширение особенно полезным в пользовательских интерфейсах, например, при реализации поиска по имени, фамилии, или другим полям, которые часто вводятся с ошибками.

Из недостатков стоит отметить, что pg\_trgm может работать медленнее, чем pg\_bigm, на очень больших объемах текста. Также, как и в случае с pg\_bigm, триграммные индексы могут занимать довольно много места в базе данных.

**Pgcrypto**

Расширение pgcrypto использовано для симметричного шифрования и дешифрования данных с помощью функций pgp\_sym\_encrypt и pgp\_sym\_decrypt. Основная задача — защита пользовательских email-адресов при сохранении в базе данных, что особенно актуально при работе с чувствительными данными.

Метод CreateUserEncrypted сохраняет нового пользователя в таблицу users, при этом email шифруется функцией pgp\_sym\_encrypt(email, key). Ключ шифрования передаётся как параметр и должен быть известен при расшифровке.

Метод GetUserDecrypted извлекает зашифрованный email из таблицы users и расшифровывает его через pgp\_sym\_decrypt(email\_enc, key)::text. Это позволяет работать с зашифрованными данными, не храня их в открытом виде.

Метод EncryptRandomUsers берёт случайные записи из таблицы users, шифрует email-адреса и сохраняет результат в отдельную таблицу encrypted\_users. Это позволяет демонстрировать работу шифрования на больших объемах данных.

func (s \*Store) CreateUserEncrypted(

    ctx context.Context,

    userID int,

    email, name string,

    balance float64,

    key string,

) error {

    \_, err := s.DB.ExecContext(ctx, `

        INSERT INTO users(user\_id, email\_enc, balance, name)

        VALUES ($1, pgp\_sym\_encrypt($2, $5), $3, $4)

    `, userID, email, balance, name, key)

    return err

}

func (s \*Store) GetUserDecrypted(

    ctx context.Context,

    userID int,

    key string,

) (email, name string, balance float64, err error) {

    row := s.DB.QueryRowContext(ctx, `

        SELECT pgp\_sym\_decrypt(email\_enc, $2)::text AS email, name, balance

        FROM users

        WHERE user\_id = $1

    `, userID, key)

    err = row.Scan(&email, &name, &balance)

    return

}

// EncryptRandomUsers берёт count случайных записей из users и сохраняет их зашифрованные email в encrypted\_users

func (s \*Store) EncryptRandomUsers(ctx context.Context, key string, total int) error {

    const batchSize = 10\_000

    start := time.Now()

    log.Println("Создание таблицы encrypted\_users (если не существует)...")

    \_, err := s.DB.ExecContext(ctx, `

        CREATE TABLE IF NOT EXISTS encrypted\_users (

            user\_id INT PRIMARY KEY,

            email\_enc BYTEA

        )

    `)

    if err != nil {

        return fmt.Errorf("create table: %w", err)

    }

    log.Printf("Начинаем шифрование %d пользователей батчами по %d...\n", total, batchSize)

    for offset := 0; offset < total; offset += batchSize {

        select {

        case <-ctx.Done():

            return fmt.Errorf("context canceled: %w", ctx.Err())

        default:

        }

        log.Printf("Шифруем пользователей: %d – %d...", offset+1, offset+batchSize)

        \_, err := s.DB.ExecContext(ctx, `

            INSERT INTO encrypted\_users (user\_id, email\_enc)

            SELECT user\_id, pgp\_sym\_encrypt(email, $1)

            FROM users

            ORDER BY random()

            LIMIT $2

            ON CONFLICT (user\_id) DO NOTHING

        `, key, batchSize)

        if err != nil {

            return fmt.Errorf("batch offset %d failed: %w", offset, err)

        }

    }

    log.Printf("✅ Шифрование завершено за %s\n", time.Since(start))

    return nil

}

// GetDecryptedEmailByID возвращает расшифрованный email по user\_id

func (s \*Store) GetDecryptedEmailByID(

    userID int,

    key string,

) (string, error) {

    var email string

    err := s.DB.QueryRow(`

        SELECT pgp\_sym\_decrypt(email\_enc, $2)::text

        FROM encrypted\_users

        WHERE user\_id = $1

    `, userID, key).Scan(&email)

    if err != nil {

        return "", fmt.Errorf("decrypt email for user %d: %w", userID, err)

    }

    return email, nil

}

Расширение pgcrypto является мощным встроенным инструментом для реализации защиты данных в PostgreSQL. Оно позволяет надёжно шифровать, расшифровывать и хешировать данные, полностью контролируя процесс на уровне базы данных. Несмотря на некоторые ограничения, связанные с производительностью и необходимостью управления ключами, pgcrypto остаётся одним из самых удобных и надёжных способов защиты информации в системах на базе PostgreSQL.