

Министерство образования и науки Российской Федерации  
Санкт-Петербургский Политехнический Университет Петра Великого

—  
Институт компьютерных наук и кибербезопасности

**ЛАБОРАТОРНАЯ РАБОТА № 4**

**«Физическая организация данных и индексация поиска»**

по дисциплине «Системы управления базами данных»

Выполнила

студентка гр. 5131001/10302

Куковьякина Д. А.

*<подпись>*

Преподаватель

А.

Зубков Е.

*<подпись>*

Санкт-Петербург

2025

## **Цель работы**

Получение навыков оценки влияния организации данных и метода доступа при поиске на производительность процессов выборки и модификации информации в базе данных.

## **Формулировка задания**

1. Определить размер файлов пользовательских и системных баз данных сервера СУБД.
2. Оценить рост хранилища данных на примере логгирования данных пользователя. Для проведения эксперимента:
  - a. Определить для пустой таблицы, сколько места она занимает:
    - i. Воспользоваться запросом к системным таблицам сервера или предустановленным процедурам.
    - ii. Оценить размер физического хранения файла на диске
  - b. Добавить в таблицу некоторое (от 10) количество записей.
  - c. Повторить оценку пространства.
    - i. Если оно не изменилось, удвоить количество данных, пока разница не станет заметна.
    - ii. Последовательными экспериментами определить, насколько возможно точно, размер одной записи. Для этого использовать экспериментальное определение размера экстента (блока записи) и число записей в нем. Учесть особенность организации данных в первом экстенте.
  - d. Повторить проверку с дальнейшим увеличением числа записей (с приростом, не менее пяти-шести раз).
3. Определить, какой объем понадобится для хранения N записей. Обосновать результат.
4. Проверить полученный результат, увеличив заполненное число записей в несколько раз (не менее пяти-шести проверок).
5. Провести секционирование таблицы и его оценку. Для этого:
  - a. Очистить таблицу.

b. Привести таблицу в не кластеризованный вид, удалить индексы (при наличии).

c. Автоматически заполнить таблицу значениями до размера в 1000000 записей. Данное количество может быть больше или меньше в зависимости от скорости оборудования. Если обработка записей происходит очень быстро (или очень медленно) их количество можно уменьшить (или, соответственно, увеличить) до получения таблицы с задержкой минимум в секундах, на операции простой выборки не кластеризованных данных.

d. Организовать секционирование созданной таблицы средствами СУБД (если они доступны) и/или вручную.

e. Организовать (при возможности) размещение секций на разных дисках.

6. Оценить скорость выполнения трех запросов к секционированным данным:

a. Для варианта размещения на одном диске (не менее 10 тестов)

b. Для варианта размещения на разных дисках (если он возможен, не менее 10 тестов)

Для теста составьте три запроса одной селективности, для выполнения двух из которых нужны данные только одной из двух секций (соответственно первой секции и второй секции для первого и второго запроса), а для третьего – обеих секций.

7. Провести эксперимент по определению производительности выборки с использованием индексов, согласно приведенному в теоретической части плану:

a. Создать тестовые отношения (без индекса и кластеризации, с индексом без кластеризации).

b. Разработать функцию по выборки одного случайного значения с замером времени операции.

c. Провести тесты, постепенно увеличивая объем данных в тестовых отношениях используя случайные данные с низкой селективностью.

8. Построить график зависимости времени выборки одной случайной записи от размера таблицы (то есть, общего числа записей в ней) для обоих случаев – без индекса и с индексом, и определить сложность операции выборки в каждом случае. Использовать один или два графика для представления обоих случаев, в зависимости от того, насколько они будут визуально показательны.

9. Аналогично провести эксперимент по определению производительности вставки с использованием индексов, согласно приведенному плану и построить графики.

d. Создать тестовые отношения:

i. Без индекса и кластеризации

ii. Без кластеризации с простым индексом

iii. Без кластеризации с уникальным индексом

iv. Без кластеризации с индексом по выражению

v. Без кластеризации с индексом по функции

e. Разработать функцию по вставки одного случайного значения с замером времени операции.

f. Провести тесты, постепенно увеличивая объем данных в тестовых отношениях используя случайные данные с низкой селективностью.

10. Построить график – зависимость времени занесения одной случайной записи от размера таблицы для каждого из пяти индексов.

11. Аналогично эксперименту с операцией вставки, на тех же типах тестовых отношений, провести эксперимент по определению производительности обновления данных и постройте графики.

## Ход работы

### Оценка размеров баз данных и таблиц

Первым делом необходимо определить размер пользовательских и системных баз данных. Для этого воспользуемся следующим запросом:

```
SELECT oid, datname, pg_size_pretty(pg_database_size(datname)) FROM pg_database
```

- `pg_database` — это системный каталог в PostgreSQL, который содержит информацию обо всех базах данных, существующих на сервере.
- `oid` (Object Identifier) — это столбец в `pg_database`, который содержит уникальный идентификатор объекта. Он понадобится в дальнейшем для нахождения папки базы данных.
- `datname` — это столбец в `pg_database`, который содержит имя базы данных.
- `pg_database_size(datname)` — это функция, которая возвращает размер базы данных (в байтах), имя которой указано в аргументе.
- `pg_size_pretty()` - это функция, которая преобразует размер в байтах в удобочитаемый формат (например, КБ, МБ, ГБ).

Результат представлен на рисунке 1.

	oid [PK] oid	datname name	pg_size_pretty text
1	5	postgres	7811 kB
2	16388	lab4	7891 kB
3	1	template1	7891 kB
4	4	template0	7657 kB

Рисунок 1 – Размеры баз данных

Системная база данных `template1` — это шаблон по умолчанию. При создании новой БД с помощью команды `CREATE DATABASE`, PostgreSQL копирует структуру и содержимое `template1` в новую базу данных. Он содержит стандартные объекты PostgreSQL, такие как системные таблицы, функции и типы данных. Видно, что у пользовательской БД `lab4` такой же размер.

Системная база данных template0 – это "чистый" шаблон базы данных. Он используется в тех случаях, когда необходимо создать новую базу данных с "чистого листа".

Следующим этапом является оценка роста хранилища. Для этого создадим таблицу для логгирования и узнаем ее размер запросом

```
SELECT oid, pg_relation_size('logs') FROM pg_class WHERE relname = 'logs';
```

- pg\_class — это системный каталог в PostgreSQL, который содержит метаданные о таблицах, представлениях, индексах и других "классах" базы данных.
- relname — это столбец в pg\_class, содержащий имя объекта (таблицы, представления и т. д.).
- pg\_relation\_size('logs') — это функция PostgreSQL, которая возвращает размер отношения в байтах.

Из рисунка 2 видно, что ее размер 0 байт. Также найдем файл, соответствующей таблице по OID. В его свойствах размер также 0 байт (рисунок 3).

	oid [PK] oid	pg_relation_size bigint
1	16424	0

Рисунок 2 – Размер таблицы


	16424
Тип файла:	Файл
Описание:	16424
Расположение:	C:\Program Files\PostgreSQL\17\data\base\16388
Размер:	0 байт
На диске:	0 байт

Рисунок 3 - Размер таблицы на диске

Добавим в таблицу 10 записей (рисунок 4) и увидим увеличение ее размера на 8 Кб (рисунок 5).

	username character varying (100)	action_type text	changed_table text	changes text	changed_on timestamp without time zone (6)
1	Ivan	Insert	Employee	Inserted new employee Masha	2021-08-25 12:30:00
2	Masha	Delete	Employee	Deleted employee Masha	2021-09-01 11:00:00
3	Petr	Delete	Product	Deleted product ID 123	2023-10-26 11:30:00
4	Elena	Insert	Customer	Inserted new customer John Doe	2023-10-26 12:45:00
5	Ivan	Update	Employee	Updated salary for Ivan	2023-10-26 14:00:00
6	Masha	Select	Orders	Viewed all orders	2023-10-26 15:15:00
7	Petr	Insert	Product	Inserted new product Laptop	2023-10-26 16:30:00
8	Elena	Delete	Customer	Deleted customer ID 456	2023-10-26 17:45:00
9	Ivan	Select	Employee	Viewed all employees	2023-10-26 15:15:00
10	Masha	Insert	Employee	Inserted new employee Olga	2023-10-27 09:15:00

Рисунок 4 – Новые записи

	oid [PK] oid	pg_relation_size bigint
1	16424	8192

Рисунок 5 – Размер при 10 записях

При добавлении еще 10 записей размер не меняется. Постепенно будем увеличивать число записей и при добавлении 91-й записи размер возрастает еще на 8 Кб (рисунок 6, 7).

	oid [PK] oid	pg_relation_size bigint	count bigint
1	16483	8192	90

Рисунок 6 – Размер при 90 записях

	oid [PK] oid	pg_relation_size bigint	count bigint
1	16483	16384	91

Рисунок 7 – Размер при 91 записи

Получается блок состоит из 90 записей и весит 8 Кб. Отсюда следует, что одна запись занимает примерно 91 байт.

Таким образом, для хранения 181 записи понадобится 24 кб. Результат проверки представлен на рисунках 8 и 9.

	oid [PK] oid	pg_relation_size bigint	count bigint
1	16483	16384	180

Рисунок 8 – Размер при 180 записях

	oid [PK] oid 🔒	pg_relation_size bigint 🔒	count bigint 🔒
1	16483	24576	181

Рисунок 9 – Размер при 181 записи

Для хранения 1000 записей понадобится 88 Кб (11 блоков) или 90 112 байт (рисунок 10).

	oid [PK] oid 🔒	pg_relation_size bigint 🔒	count bigint 🔒
1	16483	90112	1000

Рисунок 10 – Размер при 1000 записях

## Секционирование

Секционирование таблиц — это метод разделения большой таблицы на более мелкие части, называемые секциями. Каждая секция хранит подмножество данных исходной таблицы, что позволяет улучшить производительность.

Для каждой секции задаются границы, определяющие, какие данные попадают в эту секцию. Границы могут быть заданы в виде диапазонов значений (для секционирования по диапазонам) или списков значений (для секционирования по спискам). Каждая секция, по сути, является отдельной таблицей, но при этом все они логически объединены в одну секционированную таблицу.

При выполнении запроса к секционированной таблице PostgreSQL определяет, к каким секциям нужно обратиться для получения необходимых данных. Сканируются только те секции, которые содержат данные, соответствующие условиям запроса, что значительно ускоряет выполнение запроса.

Создадим секционированную таблицу. В качестве границ укажем дату: в одной секции будут храниться записи за 2023 год, во второй – за 2024 (рисунок 11).



```

DROP TABLE IF EXISTS logs_part CASCADE;
CREATE TABLE logs_part (
    username varchar(100) NOT NULL,
    action_type TEXT NOT NULL,
    changed_table TEXT NOT NULL,
    changes TEXT NOT NULL,
    changed_on TIMESTAMP(6) NOT NULL
) PARTITION BY RANGE (changed_on);

CREATE TABLE logs_2023 PARTITION OF logs_part
FOR VALUES FROM ('2023-01-01 00:00:00') TO ('2024-01-01 00:00:00');

CREATE TABLE logs_2024 PARTITION OF logs_part
FOR VALUES FROM ('2024-01-01 00:00:00') TO ('2025-01-01 00:00:00');

```

Рисунок 11 – Секционированная таблица

Реализуем функцию заполнения, в которой будет генерироваться только 2023 и 2024 год (рисунок 12).

```

CREATE OR REPLACE FUNCTION generate_logs(num_rows INT, target_table TEXT)
RETURNS VOID AS $$
DECLARE
    i INT := 1;
    username_prefix VARCHAR(50) := 'user_';
    action_types TEXT[] := ARRAY['Insert', 'Update', 'Delete', 'Select'];
    table_names TEXT[] := ARRAY['Employee', 'Department', 'Product', 'Customer', 'Orders'];
    random_year INT;
BEGIN
    WHILE i <= num_rows LOOP
        random_year := 2023 + (i % 2);

        EXECUTE FORMAT('INSERT INTO %I (username, action_type, changed_table, changes, changed_on) VALUES ($1, $2, $3, $4, $5)', target_table)
        USING
            username_prefix || i,
            action_types[1 + (i % array_length(action_types, 1))],
            table_names[1 + (i % array_length(table_names, 1))],
            'Generated log entry ' || i,
            make_date(random_year, (random()*11+1)::int, (random()*27+1)::int) +
            make_time((random()*22+1)::int, (random()*58+1)::int, (random()*58+1)::int);
        i := i + 1;
    END LOOP;
END;
$$ LANGUAGE plpgsql;

```

Рисунок 12 –Функция для заполнения таблиц

Каждую из таблиц заполним двумя миллионами строк и оценим скорость выполнения трех запросов. Для выполнения двух нужны данные только одной из двух секций (соответственно первой секции и второй секции для первого и второго запроса), а для третьего – обеих секций (таблица 1).

Таблица 1 – Оценка скорости запросов

SQL запрос	Время без секционирования	Среднее	Время с секционированием	Среднее
SELECT * FROM logs WHERE changed_on < '2023-12-31';	652	615	562	551
	637		552	
	574		584	
	606		532	
	604		524	
SELECT * FROM logs WHERE	566	602	465	511
	573		518	

changed_on > '2024-01-01';	603		515	
	635		505	
	631		552	
SELECT * FROM logs WHERE changed_on < '2023-12-31' OR changed_on > '2024-01-01';	979	1001	1026	1024
	1018		1085	
	1066		996	
	974		1014	
	972		999	

В случае запросов, которые обращаются только к одной секции, секционированная таблица демонстрирует выигрыш в производительности. Это связано с тем, что PostgreSQL может исключить ненужные секции из плана выполнения запроса и сканировать только ту секцию, которая содержит необходимые данные. В результате уменьшается объем данных, который нужно обработать, и сокращается время выполнения запроса.

В случае запроса, который обращается к обеим секциям, ситуация может измениться. На секционированной таблице PostgreSQL необходимо обратиться к обеим секциям и объединить результаты. Это может привести к дополнительным накладным расходам на управление секциями и объединение данных.

## Выборка

Индекс представляет собой структуру данных, которая содержит упорядоченные значения одного или нескольких столбцов таблицы и ссылки на соответствующие строки. Когда выполняется запрос к таблице с условием отбора по индексированному столбцу, SQL-сервер сначала обращается к индексу, чтобы найти нужные значения, а затем, используя ссылки, быстро находит соответствующие строки в таблице.

B-tree (Balanced tree) — это наиболее распространенный тип индекса, который используется по умолчанию в большинстве СУБД. Он представляет собой сбалансированное дерево, в котором листья содержат значения индексируемого столбца и указатели на соответствующие строки таблицы.

За счет ускорения доступа к данным индексы позволяют повысить эффективность операций выборки данных, однако при каждом изменении данных, связанных индексом, изменения также должны быть внесены и в сам индекс – что замедляет операции обновления, вставки и удаления данных (операции модификации).

В общем случае, выполняя запрос, оптимизатор СУБД может как воспользоваться индексом, так и проигнорировать его. Как правило, вторая ситуация встречается, когда в диапазон выборки попадает более половины всех кортежей и поиск их по индексным структурам теряет смысл.

Кластеризованный индекс определяет физический порядок данных в таблице. Можно создавать для таблицы лишь один кластеризованный индекс, т. к. строки таблицы нельзя упорядочить физически более чем одним способом.

Некластеризованный индекс — это отдельная структура, которая содержит копию индексируемого столбца (или нескольких столбцов) и указатели на соответствующие строки в таблице. Для каждого некластеризованного индекса СУБД создает дополнительную индексную структуру, которая сохраняется в индексных страницах:

- Если существует кластеризованный индекс, то закладка некластеризованного индекса показывает B+-дерево кластеризованного индекса таблицы.

- Если таблица не имеет кластеризованного индекса, закладка идентична идентификатору строки (RID — Row Identifier), состоящего из трех частей:

- а. адреса файла, в котором хранится таблица;
- б. адреса физического блока (страницы), в котором хранится строка;
- с. смещения строки в странице.

Создадим два отношения: одно без индексов, а второе с простым индексом по дате (рисунок 13).

```

DROP TABLE IF EXISTS logs_index CASCADE;
CREATE TABLE logs_index (
    username varchar(100) NOT NULL,
    action_type TEXT NOT NULL,
    changed_table TEXT NOT NULL,
    changes TEXT NOT NULL,
    changed_on TIMESTAMP(6) NOT NULL
);
CREATE INDEX idx_changed_on ON logs_index (changed_on);

```

Рисунок 13 – Отношение с индексом

Затем реализуем функцию (рисунок 14). Она будет заполнять таблицу 10, 100, 100 и т. д. записями, после чего 6 раз проводить SELECT случайной записи. Время каждой записи и текущее количество записей в таблице будут записываться в отдельную таблицу results.

```

CREATE OR REPLACE FUNCTION random_select(target_table TEXT)
RETURNS VOID AS $$
DECLARE
    record_counts int[] := ARRAY[10, 100, 1000, 10000, 100000, 1000000];
    records int;
    start_time timestamp;
    end_time interval;
    random_log TIMESTAMP(6);
BEGIN
    FOREACH records IN ARRAY record_counts LOOP
        PERFORM generate_logs(records, target_table);

        FOR i IN 1..6 LOOP
            EXECUTE FORMAT('SELECT changed_on FROM %I ORDER BY RANDOM() LIMIT 1', target_table) INTO random_log;

            start_time := clock_timestamp();
            EXECUTE FORMAT('SELECT * FROM %I WHERE changed_on = $1', target_table) USING random_log;
            end_time := clock_timestamp() - start_time;

            INSERT INTO results (execution_time, target_table, records)
            VALUES (extract(millseconds FROM end_time), target_table, records);
        END LOOP;

        EXECUTE FORMAT('TRUNCATE TABLE %I', target_table);
    END LOOP;
END;
$$ LANGUAGE plpgsql;

```

Рисунок 14 – Случайный select

Для выбора случайной записи используем запрос *SELECT changed\_on FROM %I ORDER BY RANDOM() LIMIT 1*. ORDER BY RANDOM() упорядочивает записи в случайном порядке, а LIMIT 1 ограничивает количество возвращаемых записей до одной. В результате запрос вернет только одно случайное значение из столбца changed\_on.

Для измерения времени используем функцию clock\_timestamp(). Она возвращает текущую дату и время с учетом часового пояса. Она отличается от других функций, таких как now() или current\_timestamp(), тем, что показывает время на момент выполнения оператора, а не на момент начала транзакции. Значение clock\_timestamp() может меняться несколько раз в

течение выполнения одного SQL-запроса, если этот запрос содержит несколько операторов или функций, требующих текущего времени.

Вызовем эту функцию для ранее созданных отношений после чего посмотрим результат (рисунок 15). Построим график зависимости (рисунок 16).

	records integer	target_table text	avg_time numeric
1	10	logs	0.038500000000000000
2	100	logs	0.020666666666666667
3	1000	logs	0.083333333333333333
4	10000	logs	0.521833333333333333
5	100000	logs	5.6561666666666667
6	1000000	logs	49.0413333333333333
7	10	logs_index	0.018500000000000000
8	100	logs_index	0.019166666666666667
9	1000	logs_index	0.020166666666666667
10	10000	logs_index	0.029833333333333333
11	100000	logs_index	0.064166666666666667
12	1000000	logs_index	0.118166666666666667

Рисунок 15 – Результаты измерения времени

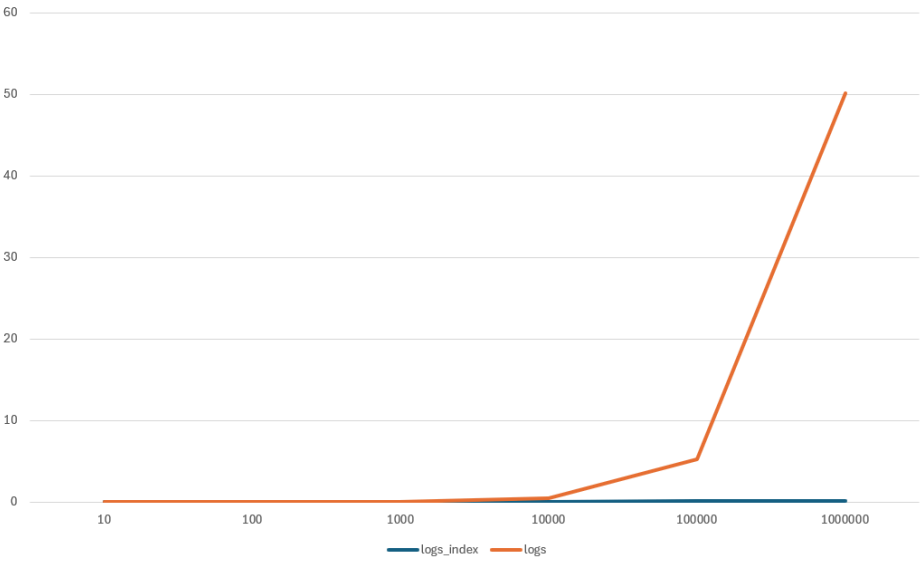


Рисунок 16 – Зависимость времени запроса от размера таблицы

**Вставка**

Простой индекс создается на одном столбце таблицы и содержит упорядоченный список значений этого столбца и указатели на

соответствующие строки в таблице. Он используется для ускорения поиска и извлечения данных по этому столбцу.

Уникальный индекс гарантирует, что все значения в индексируемом столбце (или комбинации столбцов) уникальны. При попытке добавить запись с неуникальным значением, база данных выдаст ошибку.

Индекс-выражение создается на основе выражения, включающего один или несколько столбцов таблицы, операторы и функции. Он используется для ускорения поиска по условиям, включающим это выражение.

Индекс-функция похож на индекс-выражение, но создается на основе функции, применяемой к одному или нескольким столбцам. Например, можно создать индекс на функции SUBSTRING (phone\_number, 1, 3), чтобы ускорить поиск по коду города в номере телефона.

Создадим новые отношения:

- Без индекса
- С простым индексом по столбцу даты
- С уникальным именем по имени пользователя (рисунок 17)
- С индексом-выражением, состоящим из имени пользователя и даты (рисунок 17)
- С индексом-функцией, извлекающей только саму дату без времени (рисунок 18)

```

DROP TABLE IF EXISTS logs_unique CASCADE;
CREATE TABLE logs_unique
(
    username varchar(100) NOT NULL UNIQUE,
    action_type TEXT NOT NULL,
    changed_table TEXT NOT NULL,
    changes TEXT NOT NULL,
    changed_on TIMESTAMP(6) NOT NULL
);
CREATE UNIQUE INDEX idx_unique ON logs_index (username);

DROP TABLE IF EXISTS logs_expr CASCADE;
CREATE TABLE logs_expr
(
    username varchar(100) NOT NULL,
    action_type TEXT NOT NULL,
    changed_table TEXT NOT NULL,
    changes TEXT NOT NULL,
    changed_on TIMESTAMP(6) NOT NULL
);
CREATE INDEX idx_expr ON logs_expr (username, changed_on);

```

## Рисунок 17 – Новые отношения

```

DROP TABLE IF EXISTS logs_func CASCADE;
CREATE TABLE logs_func
(
    username varchar(100) NOT NULL,
    action_type TEXT NOT NULL,
    changed_table TEXT NOT NULL,
    changes TEXT NOT NULL,
    changed_on TIMESTAMP(6) NOT NULL
);
CREATE INDEX idx_func ON logs_func (DATE(changed_on));

```

## Рисунок 18 – Новое отношение

Также изменим функцию для проведения оценки времени (рисунок 19).

```

end_time interval;
action_types TEXT[] := ARRAY['Insert', 'Update', 'Delete', 'Select'];
table_names TEXT[] := ARRAY['Employee', 'Department', 'Product', 'Customer', 'Orders'];
username_prefix VARCHAR(50) := 'insert_';
BEGIN
    FOREACH records IN ARRAY record_counts LOOP
        PERFORM generate_logs(records, target_table);

        FOR i IN 1..6 LOOP
            start_time := clock_timestamp();

            EXECUTE FORMAT('INSERT INTO %I (username, action_type, changed_table, changes, changed_on) VALUES ($1, $2, $3, $4, $5)', target_table)
            USING
                username_prefix || i,
                action_types[1 + (i % array_length(action_types, 1))],
                table_names[1 + (i % array_length(table_names, 1))],
                'Generated log entry ' || i,
                make_date((random()*24+2000)::int, (random()*11+1)::int, (random()*27+1)::int) +
                make_time((random()*22+1)::int, (random()*58+1)::int, (random()*58+1)::int);

            end_time := clock_timestamp() - start_time;

            INSERT INTO results (execution_time, target_table, records)
            VALUES (extract(millisecond FROM end_time), target_table, records);
        END LOOP;

        EXECUTE FORMAT('TRUNCATE TABLE %I', target_table);
    END LOOP;
END;
$$ LANGUAGE plpgsql;

```

## Рисунок 19 – Функция для оценки времени

После запуска для всех отношений получим следующие значения (рисунок 20). Построим по ним график (рисунок 21).

	records integer	target_table text	avg_time numeric
1	10	logs	0.012250000000000000
2	100	logs	0.010666666666666667
3	1000	logs	0.011166666666666667
4	10000	logs	0.010666666666666667
5	100000	logs	0.016166666666666667
6	1000000	logs	0.011166666666666667
7	10	logs_expr	0.013166666666666667
8	100	logs_expr	0.013333333333333333
9	1000	logs_expr	0.014666666666666667
10	10000	logs_expr	0.016333333333333333
11	100000	logs_expr	0.015833333333333333
12	1000000	logs_expr	0.016333333333333333
13	10	logs_func	0.012666666666666667
14	100	logs_func	0.012833333333333333
15	1000	logs_func	0.012666666666666667
16	10000	logs_func	0.013500000000000000
17	100000	logs_func	0.012666666666666667
18	1000000	logs_func	0.013833333333333333
19	10	logs_index	0.012500000000000000
20	100	logs_index	0.018166666666666667
21	1000	logs_index	0.011833333333333333
22	10000	logs_index	0.012500000000000000
Total rows: 30			Query complete 00:00:00.091

Рисунок 20 – Полученное время

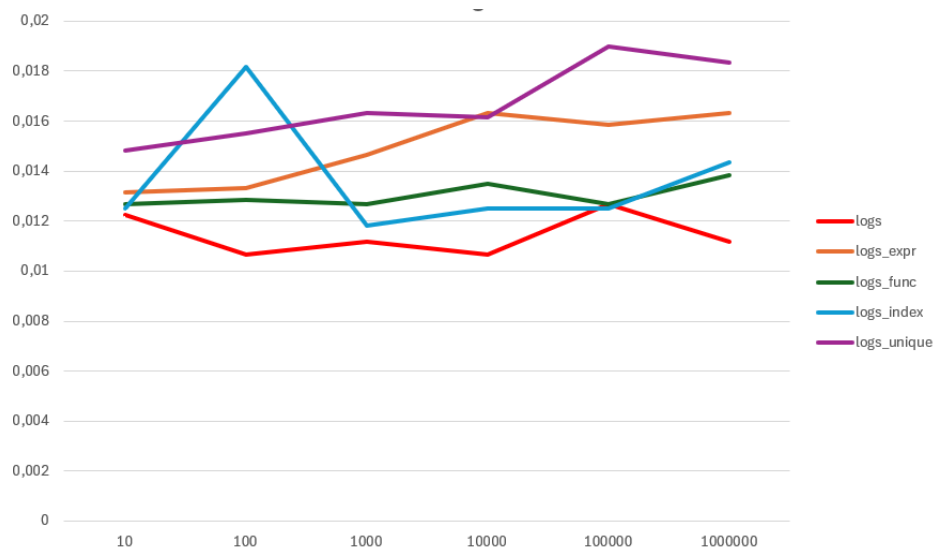


Рисунок 21 – График зависимости времени от операции вставки

## Обновление

Воспользуемся отношениями из предыдущего пункта. Также изменим функцию оценки (рисунок 22). Искать будем по полю changes, а менять поле action, чтобы не было пересечений с индексами.



```
CREATE OR REPLACE FUNCTION random_update(target_table TEXT)
RETURNS VOID AS $$
DECLARE
    record_counts int[] := ARRAY[10, 100, 1000, 10000, 100000, 1000000];
    records int;
    start_time timestamp;
    end_time interval;
    action_types TEXT[] := ARRAY['Insert', 'Update', 'Delete', 'Select'];
    random_log TEXT;
    new_act TEXT;
BEGIN
    FOREACH records IN ARRAY record_counts LOOP
        PERFORM generate_logs(records, target_table);

        FOR i IN 1..6 LOOP
            EXECUTE FORMAT('SELECT changes FROM %I ORDER BY RANDOM() LIMIT 1', target_table) INTO random_log;
            new_act := action_types[1 + (i % array_length(action_types, 1))];

            start_time := clock_timestamp();
            EXECUTE FORMAT('UPDATE %I SET action_type = $1 WHERE changes = $2', target_table) USING new_act, random_log;
            end_time := clock_timestamp() - start_time;

            INSERT INTO results (execution_time, target_table, records)
            VALUES (extract(milliseconds FROM end_time), target_table, records);
        END LOOP;

        EXECUTE FORMAT('TRUNCATE TABLE %I', target_table);
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

Рисунок 22 – Функция оценки

В результате получим следующее время (рисунок 23) и график (рисунок 24).

	records integer	target_table text	avg_time numeric
1	10	logs	0.048333333333333333333333
2	100	logs	0.02616666666666666666667
3	1000	logs	0.088333333333333333333333
4	10000	logs	0.63616666666666666666667
5	100000	logs	6.263166666666666666667
6	1000000	logs	60.069500000000000000000
7	10	logs_expr	0.023333333333333333333333
8	100	logs_expr	0.03216666666666666666667
9	1000	logs_expr	0.086833333333333333333333
10	10000	logs_expr	0.667500000000000000000000
11	100000	logs_expr	6.089666666666666666667
12	1000000	logs_expr	67.541500000000000000000
13	10	logs_func	0.025333333333333333333333
14	100	logs_func	0.03366666666666666666667
15	1000	logs_func	0.086500000000000000000000
16	10000	logs_func	0.639500000000000000000000
17	100000	logs_func	6.028666666666666666667
18	1000000	logs_func	59.479500000000000000000
19	10	logs_index	0.02666666666666666666667
20	100	logs_index	0.030833333333333333333333
21	1000	logs_index	0.09416666666666666666667
22	10000	logs_index	0.64266666666666666666667
Total rows: 30			Query complete 00:01:22.737

### Рисунок 23 – Оценка времени

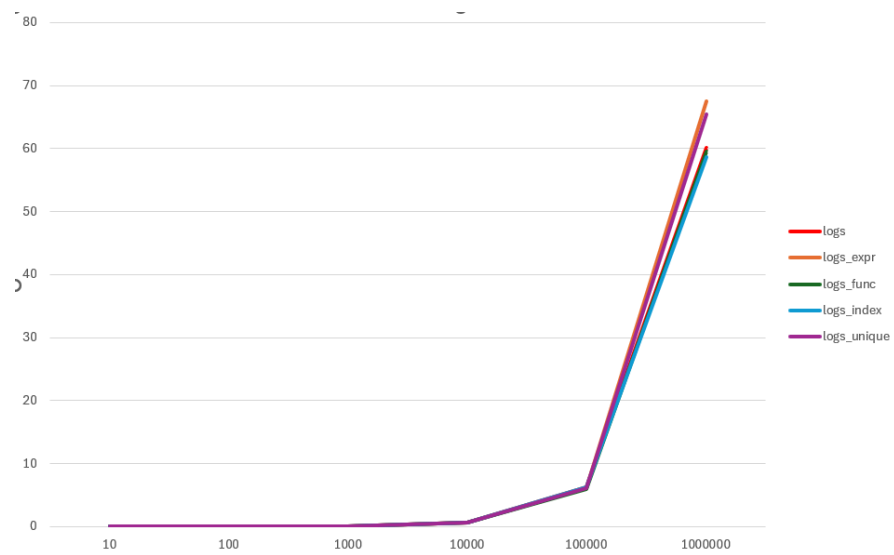


Рисунок 24 – Зависимость времени для update

## **Контрольные вопросы**

### **1. Как организуется физическое хранение данных в рассматриваемом сервере СУБД?**

PostgreSQL хранит данные в файлах на диске, организованных в иерархическую структуру. На самом верхнем уровне находится кластер баз данных, который содержит все базы данных, созданные на сервере. Каждая база данных имеет свой каталог внутри кластера.

Внутри каталога базы данных хранятся файлы, содержащие данные таблиц и индексов. Данные таблиц разбиваются на страницы фиксированного размера (8 КБ), которые хранятся в этих файлах. Каждая запись в таблице называется кортежем и размещается на страницах.

### **2. Какова структура первого блока данных? Последующих блоков?**

Первая страница данных в PostgreSQL, также известная как страница управления таблицей (типа Heap), содержит метаданные о таблице, а также информацию о том, как организованы данные на других страницах.

Содержит информацию о типе страницы (Heap), размере страницы, контрольной сумме и других метаданных. Также хранит уникальный номер, идентифицирующий таблицу в базе данных; ссылки на другие страницы данных, содержащие фактические записи таблицы; данные о том, сколько места на странице доступно для новых записей.

Последующие страницы данных содержат фактические записи таблицы.

### **3. Как секционирование влияет на производительность доступа к данным?**

Идея секционирования заключается в разделении большой таблицы на более мелкие, логически связанные части, называемые секциями. Это делается для того, чтобы запросы к базе данных обрабатывали меньший объем информации, что в свою очередь приводит к ускорению выполнения этих запросов.

#### **4. В каком случае секционирование улучшает характеристики доступа к данным?**

Секционирование становится особенно эффективным, когда речь идет о работе с большими таблицами, содержащими миллионы и миллиарды записей. В таких случаях полное сканирование таблицы для выполнения запроса может занять значительное время. Разделение таблицы на секции позволяет сократить объем обрабатываемых данных, что приводит к ускорению выполнения запросов.

При этом важно, чтобы запросы были ориентированы на определенные диапазоны данных. Тогда они будут обрабатывать только те секции, которые содержат интересующие данные, исключая необходимость сканирования всей таблицы.

#### **5. В каком случае секционирование ухудшает характеристики доступа к данным?**

Если выбрана неоптимальная стратегия секционирования, которая не соответствует характеру запросов к базе данных, это может привести к увеличению времени выполнения запросов. Например, если запросы часто обращаются к данным, которые распределены по разным секциям, системе придется выполнять больше операций для извлечения нужной информации.

Также слишком мелкие секции могут привести к избыточным накладным расходам на управление ими, что замедлит выполнение запросов. Слишком крупные секции, напротив, не дадут должного эффекта, поскольку объем обрабатываемых данных останется большим.

#### **6. Как изменятся результаты тестирования секционирования данных, если таблица будет кластеризована и/или будет иметь индекс, используемый при выполнении запроса?**

Если таблица кластеризована по тому же критерию, что и секционирование, или запрос использует индекс, созданный по столбцу, который также участвует в секционировании, это может привести к значительному улучшению производительности запросов, особенно тех,

которые обращаются к данным в пределах одной секции. Секционирование также позволяет уменьшить размер индекса и увеличит эффективность его применения.

Иначе это может привести к снижению производительности. В таком случае запросы могут потребовать обращения к данным, расположенным в разных секциях и физически удаленных друг от друга, что замедлит выполнение запросов.

### **7. Какой характер имеет зависимость времени выборки одной случайной записи от объема данных различных ситуациях?**

При выборке без индекса для поиска нужной записи СУБД должна просканировать всю таблицу, последовательно проверяя каждую запись.

Время выполнения запроса линейно зависит от размера таблицы. Чем больше записей в таблице, тем больше времени потребуется на поиск. В худшем случае, если нужная запись находится в конце таблицы, СУБД придется просмотреть все записи.

При выборке с простым индексом СУБД может использовать индекс для быстрого поиска нужной записи, не сканируя всю таблицу.

Время выполнения запроса с использованием индекса логарифмически зависит от размера таблицы. Это означает, что чем больше записей в таблице, тем незначительнее увеличивается время поиска.

### **8. Какую сложность имеет эта операция в каждом случае и почему?**

Сложность операции без индекса:  $O(n)$ , где  $n$  - количество записей в таблице.

Сложность операции с индексом:  $O(\log n)$ , где  $n$  - количество записей в таблице.

### **9. Какой характер имеет зависимость времени вставки одной случайной записи от объема данных в различных ситуациях?**

Как видно из полученных ранее графиков, время запроса не имеет четкой зависимости от объема данных, а лишь колеблется в некотором диапазоне. Такую зависимость можно назвать константной.

**10. Какую сложность имеет эта операция в каждом случае и почему? Что влияет на время вставки данных в различных случаях?**

Такую ситуацию, когда время выполнения операций практически не зависит от объема данных, можно назвать константной сложностью или  $O(1)$ . Это означает, что время выполнения остается постоянным или изменяется незначительно при увеличении объема данных.

Операция INSERT в таблицу без индекса имеет относительно низкую сложность, так как при вставке новой записи PostgreSQL просто добавляет данные в конец файла данных, что обычно требует минимальных вычислительных ресурсов.

При наличии индекса операция INSERT становится сложнее, так как кроме добавления данных в таблицу PostgreSQL также должна обновлять индекс, что может потребовать дополнительных ресурсов и времени.

**11. Какой характер имеет зависимость времени обновления одной случайной записи от объема данных различных ситуациях?**

Без индекса характер зависимости линейный или близкий к линейному,  $O(n)$ . БД вынуждена просматривать каждую запись в таблице, чтобы найти нужную для обновления. Чем больше данных в таблице, тем больше времени требуется на поиск.

С индексом характер зависимости в общем случае - логарифмический,  $O(\log n)$ , но может быть и выше в зависимости от типа индекса и количества обновляемых полей. Несмотря на то, что индекс позволяет базе данных быстрее находить нужную запись, не просматривая всю таблицу, дополнительное время тратится на обновление самого индекса.

**12. Из каких операций складывается операция обновления данных?**

Операция обновления данных в БД обычно состоит из следующих шагов:

- Поиск записи. База данных находит запись, которую необходимо обновить.
- Изменение данных. База данных изменяет значения полей в записи.
- Обновление индексов (при наличии). Если для обновляемых полей существуют индексы, база данных обновляет индексы, чтобы они соответствовали измененным данным.

### **13. Какую сложность имеет эта операция в каждом случае и почему? Почему характер временной зависимости для некоторых типов индексов отличается?**

Разные типы индексов имеют разную структуру и алгоритмы поиска и обновления.

Например, индексы по выражению или функции могут требовать дополнительных вычислений при поиске и обновлении, что может увеличить время выполнения операции.

### **14. Что такое кластеризация данных в СУБД и как она влияет на скорости операций с данными?**

Кластеризация данных в СУБД — это метод организации данных, при котором схожие записи физически размещаются рядом друг с другом на диске. Существует несколько подходов к кластеризации данных, но наиболее распространенным является кластеризация на основе индексов. В этом случае создается кластеризованный индекс, который определяет порядок хранения данных на диске. Записи в таблице физически упорядочиваются в соответствии со значениями индексируемого поля.

В отличие от обычных индексов, кластерный индекс для любой таблицы может быть только один. Кластеризация дает преимущество в том случае, если большую часть данных, которые выбираются из таблицы, составляют группы индексированных данных, а не случайно выбранные

отдельные данные, поскольку группы располагаются на одной или следующих друг за другом физических страницах.

Кластеризация по индексу значительно ускоряет поиск данных по этому индексу. Поскольку записи хранятся в упорядоченном виде, СУБД может использовать эффективные алгоритмы поиска, такие как двоичный поиск, для быстрого нахождения нужных записей.

При вставке новых записей в таблицу СУБД старается разместить их рядом с другими записями с близкими значениями индексируемых полей. Это уменьшает фрагментацию данных на диске и ускоряет последующие операции чтения. Однако, если вставка происходит в середину кластера, СУБД может потребоваться перемещение большого количества записей для поддержания порядка в индексе, что может замедлить операцию.

Изменение значений кластеризованного поля может потребовать перемещения записи на другое место на диске. Однако, если изменения невелики, СУБД может оптимизировать этот процесс, минимизируя количество перемещений.

### **15. Какой режим кластеризации надо установить при вставке данных? Как это сделать?**

Кластеризация является одноразовой операцией: последующие изменения в таблице нарушают порядок кластеризации. Другими словами, система не пытается автоматически сохранять порядок новых или изменённых строк в соответствии с индексом. Если для заданной таблицы установить параметр `FILLFACTOR` меньше 100%, это может помочь сохранить порядок кластеризации при изменениях, так как изменяемые строки будут помещаться в ту же страницу, если в ней достаточно места.

При вставке данных в таблицу PostgreSQL не существует специальных режимов кластеризации. Однако, чтобы новые данные сразу же попадали в нужное место на диске в соответствии с индексом, можно выполнить команду `CLUSTER` сразу после вставки данных.



## **16. Какой режим кластеризации надо установить при поиске данных? Как это сделать?**

При поиске данных в таблице PostgreSQL также не существует специальных режимов кластеризации. Однако, при частом использовании запросов, которые сортируют или фильтруют данные по определенной колонке, кластеризация по индексу, созданному на этой колонке, может значительно улучшить производительность.

Для этого перед выполнением запроса необходимо выполнить команду CLUSTER.

## **17. Как кластеризовать таблицу данных по нескольким атрибутам?**

Объединить эти атрибуты в первичный ключ для обеспечения уникальности, после чего создать по ним индекс, а по самому индексу провести кластеризацию таблицы.

## **Вывод**

В ходе данной лабораторной работы были получены навыки по секционированию данных и индексировании, которые могут значительно повысить производительность и улучшить управление большими объемами данных.