Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Прискорене формування результуючої вибірки на основі засобів СКБД та методів процесінгу тексту
(тема)

Виконав:

студент II курсу, групи СПм-21-1

Рудницький С.А.
(прізвище, ініціали)

Спеціальність

123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма

Системне програмування
(повна назва освітньої програми)

Керівник: доц. Барковська О.Ю.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ЕОМ _____ Коваленко А.А.
(підпис) (прізвище, ініціали)

2022 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студенту Рудницькому Сергію Анатолійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Прискорене формування результуючої вибірки на основі засобів СКБД та методів процесінгу тексту

затверджена наказом по університету від “ 07 ” листопада 2022 р. № 1454 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 12 грудня 2022 р.

3. Вхідні дані до роботи загальні відомості про засоби СКБД та методи процесінгу тексту для подальшої його обробки та пришвидшенню отримання результуючої вибірки та методів обробки

4. Перелік питань, що потрібно опрацювати у роботі
Аналіз предметної області засоби СКБД та процесінг тексту;
Формування списку вимог до методів обробки тексту та програми;
Розробка методів та робочої програми і їх налагодження;
Аналіз отриманих результатів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайди презентації – 13 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Формування вимог програми	07.11.22-10.11.22	
2	Пошук та модифікація методів обробки	11.12.22-13.12.22	
3	Розробка та налагодження програми	14.12.22-25.12.22	
4	Тестування та виправлення проблемних	25.12.22-30.12.22	
	місць		
5	Оформлення матеріалів кваліфікаційної	01.12.22-05.12.22	
	роботи		
6	Подання атестаційної роботи керівникові та	06.12.22-08.12.22	
	її попередній захист		
7	Подання роботи на рецензування	09.12.22-12.12.22	

Дата видачі завдання 07 листопада 2022 р.

Студент _____
(підпис)

Керівник роботи _____ доц.Барковська О.Ю.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 91 с., 17 рис., 4 табл., 2 дод., 35 джерел.

PYQT, СТІЛЬНИКОВИЙ-ЗАСТОСУНОК, СКБД, FULL-TEXT SEARCH, INDEXING, POSTGRESQL, MARIADB, PATTERN SEARCH, EXPLAIN ANALYZE.

Метою кваліфікаційної роботи є пошук та покращення методів пришвидшення отримання результуючої вибірки та обробки тексту в великих базах даних, а також створення застосунка для аналізу отриманих результатів.

У ході виконання кваліфікаційної роботи було розроблено модель прискореного формування результуючої вибірки, вдосконалену за рахунок модифікованих методів процесінгу тексту. Працездатність запропонованої моделі протестовано за допомогою розробленого стільникового-застосунку, написаного на мові програмування Python та SQL. Також було модифіковано методи обробки тексту при обробці пошукових запитів до бази даних, а саме повнотекстовий пошук та патерновий. Отримано наступні результати – для малих відношень швидкість обробки виросла у 17 разів, для середніх у 5 разів, для великих відношень вдвічі.

ABSTRACT

Master’s thesis: 91 pages, 17 figures, 4 tables, 2 appendices, 35 sources.

PYQT, СТІЛЬНИКОВИЙ-ЗАСТОСУНОК, СКБД, FULL-TEXT SEARCH, INDEXING, POSTGRESQL, MARIADB, PATTERN SEARCH, EXPLAIN ANALYZE.

The purpose of the qualification work is to find and improve the methods of speeding up the obtaining of the resulting set and text processing in large databases, as well as create of an application for the analysis of the obtained results.

A model of accelerated formation of the resulting set introduced in this work is the result of implaying modified text processing methods. Performance of mentioned model was tested with the help of specially developed desktop application. The application programming languages are Python and SQL. Also modified methods of text processing in the processing of search queries to the database: full-text search and pattern search. Research results as following – for small relations the processing speed increased 17 times, for medium relations it increased 5 times, for large relations it doubled.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних

СКБД – система керування базами даних

ЦП – центральний процесор

CPU – центральний процесор (англ., Central Processing Unit)

CSV – розділені комою значення (англ., Comma-Separeted Values)

FTS – повнотекстовий пошук (англ., Full-Text Search)

GIN – узагальний інвертований індекс (англ., Generalized Inverted iNdex)

I/O – введення/виведення (англ., Input/Output)

PSL – стандартна бібліотека Python (англ., Python Standard Library)

RAID – резервний масив незалежних дисків (англ., Redundant Array of Inexpensive Disks)

RegEx – регулярні вирази (англ., Regular Expression)

TVF – таблична функція (англ., Table Valued Function)

XML – розширювана мова розмітки (англ., eXtensible Markup Language)

ВСТУП

В сучасному світі практично вся буденність людей зав’язана на використанні та експлуатації інформації. Найнадійнішим місце структурованого збереження інформації є бази даних. Сьогодні це надає можливість з легкістю отримати потрібну інформацію за лічені секунди, але погано спроєктована база даних та необдумане використання індексування, мають значний вплив на зниження швидкості отримання результуючої вибірки та неефективну обробку інформації в базі даних. Покращення методів обробки тексту в базі даних це одна із найважливіших задач, яку ставить перед собою інженерія баз даних. Нині більшість реляційних СКБД надають можливість використовувати цілу низку методів та підходів, що можуть вирішити цю задачу

Одним із методів, що надає подібну можливість, є патерновий пошук, але його використання спряжене із низькою продуктивністю. Продумане індексування полів у поєднанні із повнотекстовим пошуком чудово підходить для вирішення такої задачі. Але даний підхід слабко себе показує при оновлені великих відношень, це потребує часу та значних процесорних ресурсів.

Задача полягає у створенні рішення, яке, в свою чергу є дослідженням та вдосконаленням методів процесінгу тексту для збільшення швидкості отримання результуючої вибірки у базах даних великих розмірів, враховуючи особливості індексування таких відношень та їх неоднорідність.

Для вирішення даної задачі було використано мову програмування Python та фреймворк PyQt, що здатний забезпечити створення десктопного застосунку із зрозумілим та зручним інтерфейсом. До того ж застосунки, що написані на Qt легко запускати на різних операційних системах, тобто, створений застосунок буде платформонезалежним. Різні СКБД використовувалися для наочної демонстрації дієвості розробленого підходу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз проблеми

Тенденція сучасного світу полягає в тому, що інформації з кожним роком стає все більше та більше. Пишуться наукові публікації, есеї, книги, до прикладу, в Україні за сім місяців мирного 2021 року було видано понад 5900 книг. Люди все більше користуються месенджерами та соціальними мережами, і всі ці дані десь потрібно зберігати, а найголовніше те, щоб ці сховки із даними були добре організовані та видавали потрібний результат якомого швидше. Тому великі компанії та високорозвинені держави все більше вкладаються у розвиток Big Data та, загалом, в проєктування баз даних, адже хороший дизайн бази даних важливий для забезпечення узгодженості даних, усунення їх надмірності, ефективного виконання запитів, збільшення високопродуктивності застосунка тощо. Звичайні користувачі твіттеру за день “твітять” понад 456 тисяч записів, і, відповідно, бази даних слід проєктувати так, щоб кожен бажаючий міг з легкістю знайти необхідну інформацію, а значить варто заздалегідь подумати про те, як імплементувати засоби обробки та процесінгу тексту у вашу БД задля пришвидшення видачі результуючої вибірки.[1]. Витрачення часу на розробку бази даних економить час та ресурси, а добре спроєктована база даних забезпечує легкість доступу та пошуку інформації, а саме інформація є найголовнішим у сучасному світі.

Більшість реляційних систем керування базами даних дозволяють проводити так званий частковий патерновий пошук, використовуючи предикат LIKE та RegEx(регулярні вирази).[2]. Тим не менш, використання, що першого, що другого підходів спряжене із низькою продуктивністю та зниженням видачі результуючої вибірки, якщо мова йде про великі набори даних. Їм не буде вистачати продуктивності, якщо провести пошук у полях,

котрі не проіндексовано, адже в такому випадку система вимушена буде проаналізувати все відношення цілком, станеться так зване Sequantial Table Scan, це коли система з допомогою команди IO читає всі записи на всіх сторінках збереженої бази даних. У випадку із індексованим полем ситуація може бути трохи кращою, але слід пам’ятати про існування Statistics Collector-a, котрий містить у собі загальну статистику та інформацію по відношеннях, і, зважаючи на це, СКБД вирішить, що доцільніше і швидше буде проаналізувати таблицю повністю аніж використовувати індекси. Те саме станеться, якщо ми спробуємо скористатися регулярними виразами, хоча це і потужний інструмент в умілих руках, але тут теж використовується патерновий підхід, і при їх використанні слід зважати на наслідки перераховані вище. Звідси і приходимо до висновку, що складно писати ефективні запити, коли використовуєш предикат LIKE чи RegEx.[2].

Для пришвидшення отримання результуючої вибірки важливим є загальна оптимізація та прискорення самої бази даних. Задля цього слід дотримуватися низки загальновідомих правил: завжди старатися оптимізовувати свої запити, користуватися секціонуванням, якщо таблиця занадто велика, а також використовувати різного роду індекси, це значно пришвидшить роботу. Але і серед цих правил є певні маловідомі деталі, які не тільки здатні пришвидшити отримання результуючої вибірки, а і можуть покращити обробку тексту в запитах. Ось так ми і знайомимося із повнотекстовим пошуком(Full-Text Search) [3], котрий в якості індексів використовує слова, і можливості якого можна розширити з допомогою семантики. Завдяки різним операторам, які підтримують семантику, пошук можна зробити більш плідним. Також швидкість та ефективність пошукових запитів залежить від грамотного додавання, покращення або просто видалення індексів [4] у відношеннях. Часто бази даних мають доволі витончену структуру та типи даних типу TEXT чи JSONB [5], що унеможливлює використання індексів типу B-Tree, адже в такому випадку система не зможе швидко видавати результат, і тому існує ще один підхід для

вирішення подібної проблеми, а саме створення індексів на основі структури відмінної від звичайного B-tree чи B+tree, і це індекси типу GIN. [6].

Діаграма перерахованих вище підходів представлена нижче на рисунку 1.1:

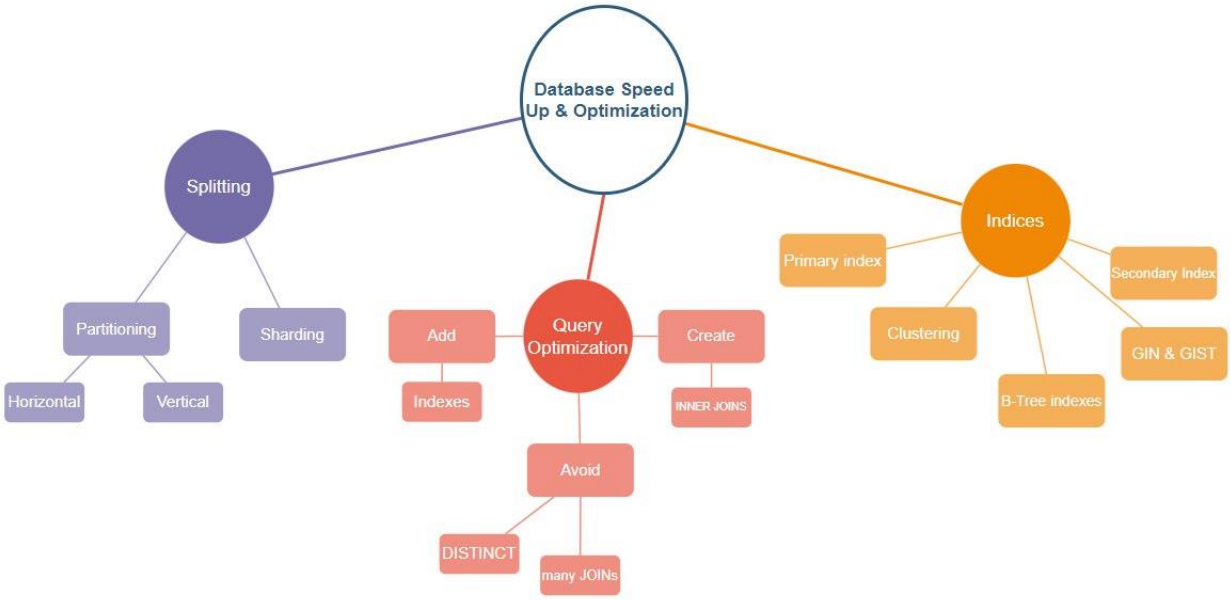

Рисунок 1.1 – Способи оптимізації та пришвидшення БД

Не існує ідеальної технології, алгоритму, чи СКБД, їх використання залежить виключно від ситуації та задачі, котру треба вирішити.[7]. Наведені вище приклади мають свої переваги та недоліки, їх комбінування, нівелювання недоліків, використання переваг та привнесення власних зауважень та методик, і буде розглянуто в цій роботі для прискорення процесінгу тексту.

1.2 Аналіз існуючих рішень

Сучасні проєкти стають все складнішими та витонченішими, а тому виникає потреба у створені допоміжних компонент, що не ускладнюють розробку, а навпаки максимально її полегшують. Інструменти, котрі

дозволяють швидко та якісно реалізувати модулі конче необхідні для сучасних програмних продуктів. Темою цієї роботи є розгляд методів для пришвидшення видачі результуючої вибірки та процесінгу тексту, а також їх покращення та доповнення. Найчастіше згадуваним способом для оптимізації запитів є якісна побудова архітектури майбутньої бази даних та її грамотне індексування.[1]. Чим, власне, є індексування?

Неякісно продумана структура індексів та відсутність взагалі будь-якої індексації, і є основним джерелом вузьких місць у застосуванні баз даних. Проєктування ефективних індексів має першочергове значення для досягнення хорошої продуктивності бази даних і додатків, що їх використовують.

Для загального уявлення про те, що таке індекси достатньо уявити книгу, і в кінці книги, є покажчик, який допомагає швидко знайти потрібну інформацію. Покажчиком є відсортований список ключових слів, а поряд із кожним ключовим словом є набір номерів сторінок, які вказують, де можна знайти кожне ключове слово. Сам індекс зберігається на сторінках, які називаються сторінками індексу. У звичайній книзі, якщо індекс охоплює декілька сторінок і, наприклад, потрібно знайти вказівники на всі сторінки, які містять слово "SQL", доведеться перегортати сторінки, поки не знайдеться індексна сторінка, яка містить ключове слово "SQL".[7]. Звідти вібдуватиметься перехід за вказівниками на всі сторінки книги. Це можна ще більше оптимізувати, якщо на самому початку індексу створити одну сторінку, яка містить алфавітний список того, де можна знайти кожну літеру. Така додаткова сторінка позбавила б від необхідності гортати покажчик, щоб знайти початкове місце. Такої сторінки не існує у звичайних книгах, але вона є в книжковому покажчику. Ця єдина сторінка називається кореневою сторінкою покажчика. Коренева сторінка є початковою сторінкою деревовидної структури, що використовується в покажчику. За аналогією з деревом, кінцеві сторінки, які містять вказівники на фактичні дані, називаються “листовими сторінками” дерева або кінцевими.[7]

Індекс – це структура на диску або в пам'яті, пов'язана з таблицею або розрізом даних(view), яка прискорює пошук записів. Індекс сховища рядків(row_id) містить ключі, що містять одне або декілька полів таблиці чи view. Для індексів сховища рядків ці ключі зберігаються в деревовидній структурі (B-Tree та похідні), що дозволяє рушієві бази даних(Database Engine) швидко і ефективно знаходити рядок або рядки, пов'язані із значеннями ключів.

Row_id зберігає дані, які логічно організовані у вигляді таблиці з рядками та стовпчиками, і що фізично зберігаються у форматі даних по рядках, який називається rowstore, або зберігаються у форматі даних по стовпчиках, який називається columnstore.

Вибір правильних індексів для бази даних та її загального навантаження завжди є балансуванням між швидкістю запитів та вартістю оновлення.[7]. Columnstore індекси або індекси з невеликою кількістю полів у ключі індексу вимагають менше дискового простору та менших витрат, як і ресурсу так і процесорного часу. З іншого боку, індекси що включають в себе багато полів охоплюють більше запитів. Індекси можна додавати, змінювати і видаляти, не змінюючи схему бази даних або архітектуру програми.

Оптимізатор запитів обирає найбільш ефективний індекс [5], в більшості випадків. Загальна стратегія проєктування індексів повинна передбачати різноманітність індексів, щоб оптимізатор запитів міг обирати з них, і робити правильні рішення. Це скорочує час аналізу і забезпечує хорошу продуктивність в різних ситуаціях.

Не слід ототожювати використання індексів з хорошою продуктивністю, а хорошу продуктивність з ефективним використанням індексів. Якби використання індексу завжди допомагало досягти найкращої продуктивності, робота оптимізатора запитів була б невимушеною. Насправді ж, неправильний вибір індексу може призвести до того, що продуктивність буде нижчою за оптимальну. Тому завдання оптимізатора запитів полягає в тому, щоб обирати індекс або комбінацію індексів тільки

тоді, коли це підвищить продуктивність, і уникати індексованого пошуку, коли це буде перешкоджати продуктивності.

Для того щоб виважено підійти до індексування варто зрозуміти наступне:

- особливості самої бази даних;
- характеристики найбільш часто використовуваних запитів;
- характеристики стовпців, що використовуються в запитах;
- які параметри індексу можуть підвищити продуктивність при створенні або обслуговуванні індексу;
- оптимальне місце зберігання індексу.

Загалом, індексування, підбір конкретних індексів, чи типів таблиць, і є основою для ефективного використання методів пришвидшення обробки запитів та покращення процесінгу тексту, що будуть наведені нижче.

Задля отримання бажаної вибірки із сукупністю текстових полів, що відповідають певному патерну зазвичай використовують регулярні вирази, або, як їх ще називають, RegEx.[2]. Нижче розглянемо особливості їх застосування, зважаючи на специфіку даної роботи.

Регулярні вирази, також відомі як RegEx, є критеріями відповідності шаблону, які можуть фільтрувати дані на основі патерну. Цей підхід активно використовується для зіставлення рядкових значень із певним шаблоном, а потім відбувається фільтрування результатів на основі заданої раніше умови.[2].

У комп’ютерній науці часто виникає потреба вирішити наступну задачу: знайти певне слово чи словесну послідовність у документі чи якомусь текстовому кластері, який відповідає фіксованому шаблону. Цей шаблон можна означити за допомогою послідовності символів, які і визначатимуть певний текстовий вираз. RegEx широко використовуються для маніпулювання текстом та його виділення. Найпоширенішою реалізацією регулярних виразів у SQL є оператор LIKE, принаймні з 1989 року , який використовує символи підстановки для відповідності патернам. Стандарт

SQL. Однак, незважаючи популярність і широке застосування, постачальники баз даних надають лише обмежену підтримку індексування для запитів із RegEx, які майже завжди, за невеликими винятками, вимагають так званої Full Table Scan, тобто повного сканування таблиці, що може зайняти кричуще багато часу, особливо, якщо ваш застосунок надсилає базі даних тисячі запитів у хвилину.

Навіть якщо розробники передбачили подібне, і якісно розробили архітектуру БД майбутнього додатку, то все таки можна стикнутися із ситуацією, яку розглянуто в наступному прикладі.

Припустимо, що поле test_name у відношенні test_table1234 є звичайним індексом, створеним з допомогою DDL команди CREATE INDEX. Також дане відношення має первинний ключ id. І тепер для отримання якоїсь результуючої вибірки ми в області condition запиту використовуємо звичайну перевірку на значення з допомогою оператора “=”.

Лістинг 1.1 – Патерновий пошук за значенням

```
SELECT id, test_name FROM test_table1234 WHERE test_name =
‘this_value’
```

У цьому випадку СКБД може навіть не звертатися до купи, бо виконується запит із первинним ключем, що вже є індексом, і в такому випадку, маючи цей ключ, отримати решту полів запита не складе ніяких проблем. Але тепер спробуємо виконати наступний запит:

Лістинг 1.2 – Патерновий пошук за предикатом LIKE

```
SELECT id, test_name FROM test_table1234 WHERE test_name LIKE
‘%this_value%’
```

У даному випадку ми відчуємо, що таке повільний запит, особливо якщо сама таблиця доволі таки велика, адже ми не можемо сканувати та порівнювати індекси. У попередньому запиті ми порівнюємо індекси не із

значенням, а із виразом, і таких виразів може бути багато, а тому БД просто порівнює рядок за рядком. Деякі СКБД у таких ситуаціях використовують паралельний аналіз, але це не надто рятує ситуацію.

Це підводить нас до інших способів використання RegEx, до прикладу, до більш складного оператора зіставлення шаблонів, який називається оператор тільда «~». З його допомогою можна використовувати повноцінний синтаксис регулярних виразів. Також можна скористатися окремим підкласом функцій для роботи із текстом, який призначений конкретно для підрядків та регулярних виразів. Така можливість доступна в більшості найпоширеніших СКБД, у PostgreSQL, наприклад. Ці функції починаються на regex та дозволяють всіляко змінювати значення результуючої вибірки.[9].

Загалом, найкориснішою порадою щодо використання регулярних виразів при обробці тексту в БД є те, що краще не використовувати регулярні вирази, адже це майже завжди буде Full Table Scan та потребуватиме великих часових витрат та певних ресурсів процесора. Але, слід зазначити, що regex – доволі потужний засіб, якщо потрібно знайти окреме слово чи вираз, і доступні вони майже у всіх СКБД, що робить їх портативність більш ніж можливою. Також, серед можливих способів пришвидшення отримання результуючої вибірки є секціонування.

Оскільки величезна кількість даних зберігається в базах даних, продуктивність і масштабування є двома суттєвими факторами при проєктуванні архітектури БД. Із додаванням даних у відношення та із розширенням чи зміною загальної структури БД, збільшується кількість записів на фізичному диску, і, відповідно, кількість дискових сторінок, а із ними і сторінок бази даних, збільшується кількість звернень до пам’яті, тобто кількість операцій IO. За один раз ця операція повертає одну сторінку, слід пам’ятати про те, що результат її роботи залежить від розбивки диску, його форматування, чи того, яка це пам’ять: SSD чи HDD. Дана операція доволі ресурсозатратна, а тому її застосування варто звести до мінімума. Простіше кажучи, запит типу select * буде доволі таки “важким”, і його слід уникати.

Секціонування може бути хорошим рішенням, оскільки воно може допомогти розділити велику таблицю на менші “підтаблиці” і, таким чином, зменшити сканування таблиці та проблеми підкачки пам’яті, що, зрештою, підвищить продуктивність.

При секціонування одразу виникає питання із вибором його типу, адже існує два різновиди: горизонтальне та вертикальне. У першому випадку це ніби рвати листок на менші частини поперек, типу спочатку розрізати його навпіл, тоді ще, і ще. У другому випадку листок рветься не поперек, а вздовж, по стовпчикам. Цей тип секціонування корисний, коли є поля із справді великими типами даних, і їх можна перемістити у повільніші місця для зберігання, що зекономить суттєво час.[8].

У секціонування також є різні класи:

- range, або ж межові, зазвичай використовуються для розбиття відношення по датах чи вікових категоріях;

- list, або ж список, цей клас чудово підходить для дискретно визначених величин, типу назви курсу в університеті, назви міста, країни тощо;

- hash, гешування, основним мінусом є те, що потрібно власноруч задавати величину модуля для обчислення гешу по якому кожен запис буде додаватися у таблицю.

Із поданого вище переліку стає зрозуміло, що найліпше для даної роботи підходить саме другий клас секціонування, а саме list. До прикладу, існує таблиця в якій зібрано список всіх найвизначніших пам’яток архітектури по містах України. І щоб якомога швидше обробляти текст у ньому можна скористатися саме секціонуванням. У кожній партиції можна, на свій розсуд, проіндексувати поля, щоб покращити загальний процесінг та щоб пришвидшити час отримання результуючої вибірки.

Недоліком такого підходу є те, що значення у цьому відношенні мають бути дискретними, а тому, зважаючи на хитке становище історичних пам’яток в країні, слід розуміти, що цей підхід без ніяких допоміжних засобів

ефективним не буде, адже, наприклад, оновлення конкретних записів може бути вельми “важким”. Також слід уникати частого оновлення партицій, бо це може нашкодити пам’яті комп’ютера. Оновлення загальної структури бази даних та програмуння можуть також викликати певні труднощі та проблеми, але, натомість, можна отримати масштабованість та підвищену швидкість для великих баз даних.

Раніше було розглянуто патерновий пошук, котрий є частиною стандарту SQL із самого початку, і доступний для кожної бази даних на основі SQL. Він виглядає наступним чином:

Лістинг 1.3 – Приклад поганої швидкодії предиката LIKE

```
SELECT test_name FROM test_table WHERE test_name LIKE ‘pattern’;
```

Це поверне поля test_name, що відповідають шаблону ‘pattern’. Для розширення можливостей цього пошуку доступні різні байдужі символи(wildcards), типу *, % тощо. Даний підхід обробки текстових полів у базі даних доволі простий та зрозумілий, але, як було зазначено вище, даний підхід є неефективним та громіздким, а тому в комп’ютерній науці для вирішення задачі, розв’язанню котрої і присвячена дана робота, використовують значно витонченіший спосіб – Full-Text Search[6], або скорочено FTS. Повнотекстовий пошук це ефективний і потужний інструмент для пошуку та обробки тексту в базі даних. Дана можливість є все більш важливою функцією сучасних баз даних. Більшість сучасних систем керування базами даних(СКБД) мають FTS, що було вбудовано в них у значно раніших релізах, як ось PostgreSQL у версії 9.6, SQL Server 7.0, MySQL 5.6 тощо. Власне, даний метод охоплює методи пошуку текстових даних і документів, і призначений для того, щоб можна було виконувати лінгвістичний(мовний) пошук в тексті чи документі, які зберігаються в базах даних. Його можна використовувати , змінюючи параметри, такі як слова і фрази, мовні особливості, ранжирування та усунення шумових слів. Також

можна скористатися можливістю індексувати документи в рідних форматах, наприклад, офісні документи та pdf-файли, що зберігаються в базі даних, можна індексувати всіма переліченими засобами, що забезпечить ефективне застосування всіх можливостей повнотекстового пошуку[10].

Загалом, для відчутного пришвидшення отримання результуючої вибірки зазвичай використовують складні структури даних типу індексів, але самі індекси створенні на основі логічних структур, що представленні у вигляді дерев, однією із таких є B-Tree. B-Tree це збалансована структура даних для швидкого обходу всіх елементів, що складається із сторінок розміром 8К [10], причому кожна сторінка в цій структурі називається вузлом індексом. На рисунку 1.2 нижче видно, що дана структура складається з трьох основних рівнів:

- root level, кореневий рівень;
- intermediate level, проміжний рівень;
- leaf level, кінцевий рівень.

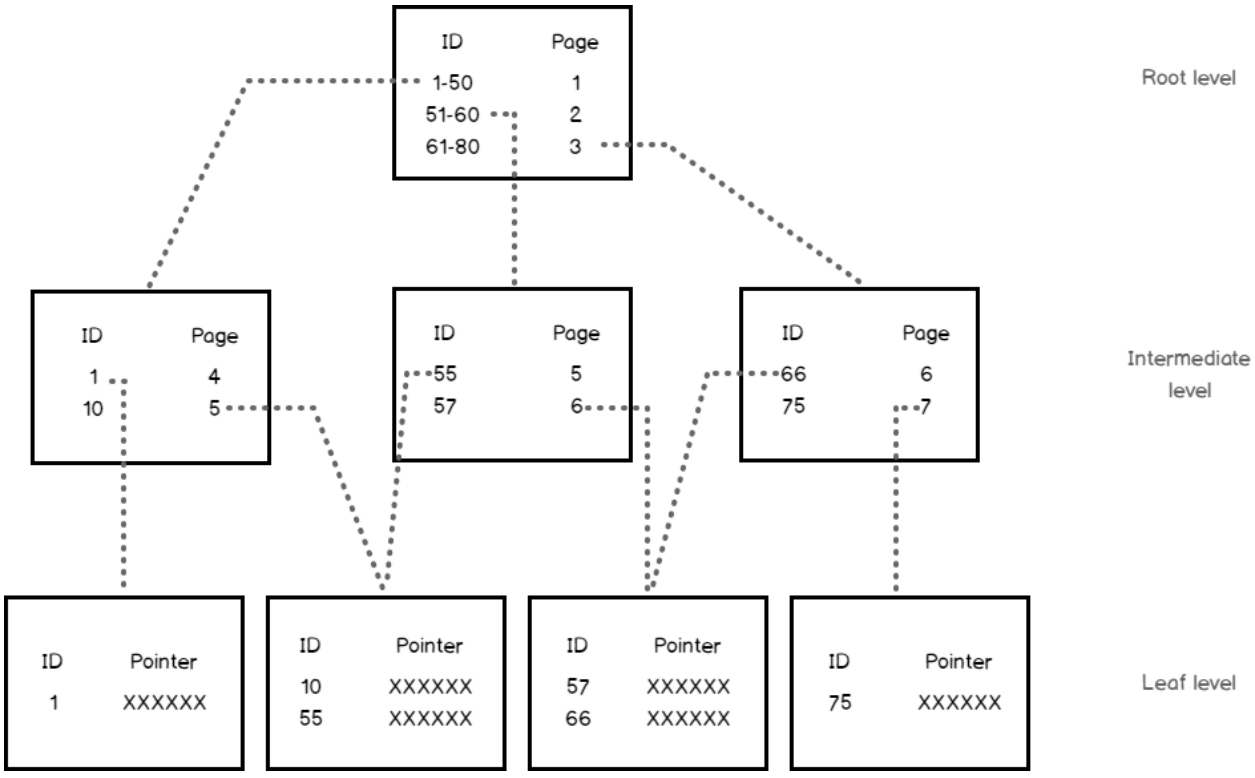

Рисунок 1.2 – Структура B-Tree

Пошук починається із кореневого рівня, що містить в собі один ключ та сторінку індексу. Всі наступні вузли містять k-1 ключів, ключі ще називають елементами. Кожен елемент даної структури має ключ та значення. Значення вказує на запис (x,a), де перше це номер сторінки, а друге рядок, це ж значення чи вказівник на дані може вказувати на первинний ключ. Кінцевий рівень це найнижчий рівень, який містить сторінки даних, що є цілю пошуків, причому кількість кінцевих сторінок залежить від обсягу даних, що зберігаються в індексі. Вставка відбувається на кінцевому рівні. Також є ще проміжний рівень, один або декілька рівнів між кореневим та кінцевим рівнями, які містять значення ключів індексу та вказівники на наступні сторінки проміжного рівня або сторінки даних листів. Кількість проміжних рівнів залежить від обсягу даних, що зберігаються в індексі, а тому слід бути обачними при створенні індексів із великими типами даних, адже це може викликати певні труднощі, до прикладу, менше місця на сторінці, бо, первинний ключ це поле рядкового типу, що автоматично зробить величину індекса дорівнювати 64К, а це забагато, і разом із цим тепер менше значень поміститься на сторінці, і це буде “гальмувати” роботу та повернення результуючої вибірки в яку це поле буде включено. Чим більше сторінок, тим більше буде виконано операцій I/O.

У цієї структури є свої обмеження:

- всі вузли містять у собі і значення і ключ;
- internal nodes потребують більше простору та більше операцій I/O, що гальмує роботу;
- межові запити повільні, бо потрібно проходитися по всьому дереву в пошуках якихось конкретних значень.

B+Tree вирішує всі перераховані вище проблеми. Це не те щоб принципово нова структура даних по відношенню до попередньої, але у неї із B-Tree є відмінність, що робить їх суттєво різними.[11].

Коли йдеться про оновлення, B-Tree значно ефективніші за звичайні бінарні дерева, але деякі операції все одно можуть виявитися доволі

ресурсозатратними, в залежності від того, де в дереві знаходиться вузол, який зберігатиме нові або оновлені дані. Тому і в стандартну структуру B-Tree було введено оптимізацію. Замість того, щоб розглядати всі вузли як однакові, нова структура має два типи вузлів. Вузли кінцевого рівня, містять фактичні дані, а всі інші, включаючи кореневий, містять лише ключові значення та вказівники на наступні вузли. Відповідно, внутрішні вузли, вони ж проміжні, менші, і можуть містити більше інформації та швидше оброблятися. Leaf nodes з'єднанні, тобто, якщо у внітрішніх вузлах було знайдено ключ, то і знайдеться значення. Всі кінцеві вузли пов'язані між собою, і вказівники між ними називаються leaf pointers, один листковий вузол приведе до іншого, що був до нього, і що буде після, також вузли на останньому рівні відсортовано, а це означає, що ця структура чудово підходить для межових запитів. Загальний вигляд B+Tree представлено на рисунку 1.3 нижче:

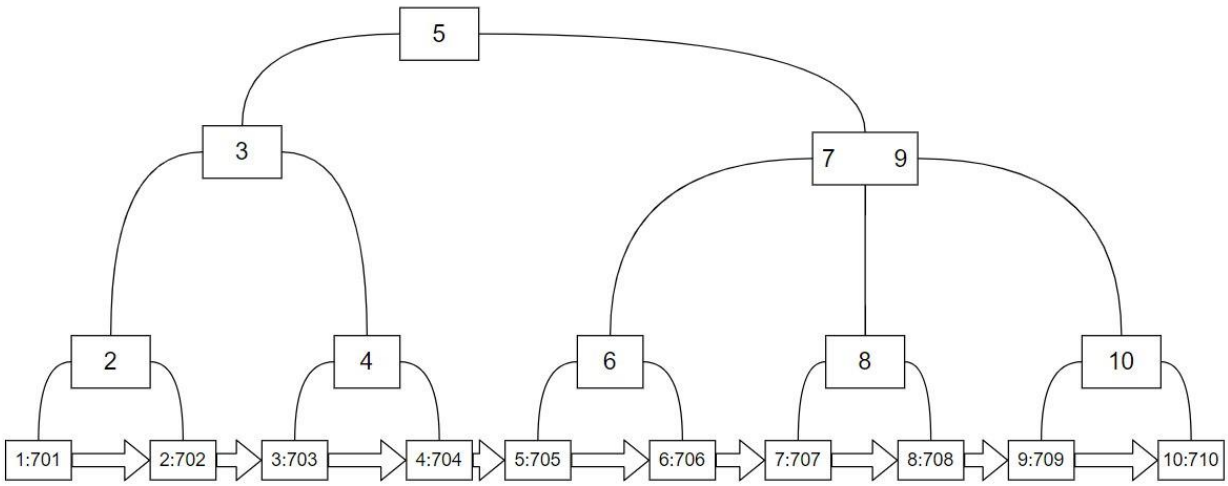

Рисунок 1.3 – Структура B+Tree

Більшість СКБД використовують похідні від B-Tree. Звісно, додавання, налаштування та видалення індексів різних типів є важливою частиною для створення загальної швидкодії роботи застосунку, який використовує базу даних. Часто такі застосунки покладаються на складні функції баз даних, і

типи даних, такі як BLOB, TEXT, чи JSONB, на різні типи масивів, і навіть на розглянутий вище FTS. І логічно, що для більш складних задач простого B-Tree індекса не достатньо, і він не працює в таких ситуаціях, наприклад, для індексування поля типу TEXT чи JSONB. Натомість слід звернути свій погляд на більш витончене рішення, далі до індексів типу GIN.

GIN використовується у випадку, коли елементи, які потрібно індексувати, є складеними значеннями, а запити, які обробляються індексом, потребують пошуку значень елементів, які з'являються в складених елементах. Наприклад, елементи можуть бути документами, а запити можуть бути пошуками документів, що містять певні слова. Тут слово елемент застосовується для позначення складеного значення, яке потрібно проіндексувати, а слово ключ для посилання на значення цього елемента. GIN завжди шукає та зберігає ключі, а не значення елементів.[10].

Індекс GIN зберігає набір пар (ключ, список публікацій), де список публікацій – це набір ідентифікаторів рядків, у яких зустрічається ключ. Той самий ідентифікатор рядка може відображатися в кількох списках публікацій, оскільки елемент може містити більше одного ключа. Кожне значення ключа зберігається лише один раз, тому індекс GIN дуже компактний для випадків, коли той самий ключ з'являється багато разів. GIN є узагальненим в тому сенсі, що код методу доступу GIN не повинен знати конкретні операції, які він прискорює. Замість цього він використовує користувацькі стратегії, визначені для конкретних типів даних. Стратегія буде визначати, як витягуються ключі з індексованих елементів та умов запиту, і як визначити, чи дійсно рядок, який містить деякі з ключових значень у запиті, задовольняє запиту.

Слово інвертований у назві GIN означає спосіб налаштування структури індекса, що полягає в побудові дерева, яке охоплює всі значення стовпчиків, де один рядок може бути представлений у багатьох місцях у дереві. Для порівняння, індекс B-Tree, як правило, має одне місце, де індексний запис вказує на конкретний рядок. Внутрішньо індекс GIN містить

індекс B-Tree, який побудовано над ключами, де кожен ключ є елементом одного або декількох індексованих елементів, наприклад, членом масиву, і де кожен кортеж на сторінці листа містить або вказівник на B-Tree вказівники купи (а “posting tree”), або простий список вказівників купи (а “posting list”), коли список досить малий, щоб поміститися в одному індексному кортежі разом із значенням ключа. Загальна структура GIN представлено нижче на рисунку 1.4:

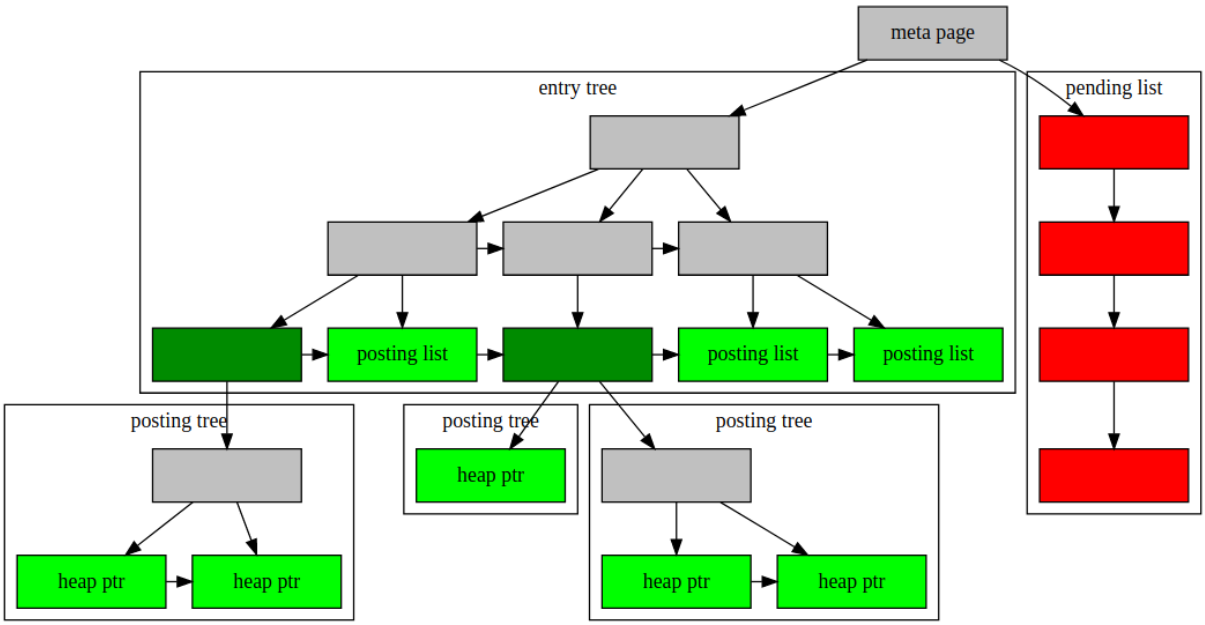

Рисунок 1.4 – Структура GIN-індекса

Загалом, GIN-індекс можна представити як зміст у книзі, де вказівниками на власне таблицю є номери сторінок. Кілька записів можуть бути об'єднані для отримання певного результату.

Як було зазначено вище використання даного типу індексу для великих або складних полів у відношенні є чи не найкращим рішенням.[11]. Припустимо, що у нас є таблиця із назвою test_gin, і в ній є єдине поле t типу TEXT, до того ж не індексоване, заповнимо його тестовими значеннями та виконаємо наступний запит:

Лістинг 1.4 – Часова оцінка виконання запиту на патерновий пошук

```
EXPLAIN ANALYZE SELECT * FROM test_gin WHERE t LIKE '%foo%bar';
```

У даному випадку команда EXPLAIN ANALYZE, фактично, виконає запит і дозволить порівняти те, що, на думку планувальника, повинно було статися, з тим, що сталося насправді. Результат на рисунку 1.5:

```
                                                     QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..42809.35 rows=400 width=8) (actual time=0.334..89.518 rows=72 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on test_gin  (cost=0.00..41769.35 rows=167 width=8) (actual time=55.583..84.209
rows=24 loops=3)
         Filter: (t ~~ '%foo%bar'::text)
         Rows Removed by Filter: 1333503
 Planning Time: 0.118 ms
 Execution Time: 89.545 ms
(8 rows)
```

Рисунок 1.5 – Тестування неіндексованого відношення test_gin

Як можна побачити, час реальної роботи складає майже 90 ms, що доволі таки багато, щоб покращити це скористаємось розширенням pg_trgm та gin_trgm_ops. Створимо індекс GIN:

Лістинг 1.5 – Створення тестового індексу GIN

```
CREATE INDEX text_idx ON test_gin USING GIN (t gin_trgm_ops);
```

Індекси GIN підтримують тільки Bitmap Index Scans (не Index Scan чи Index Only Scan), через те, що вони зберігають тільки частини значень рядків на кожній сторінці індексу. Отриманий результат можна побачити на рисунку 1.6:

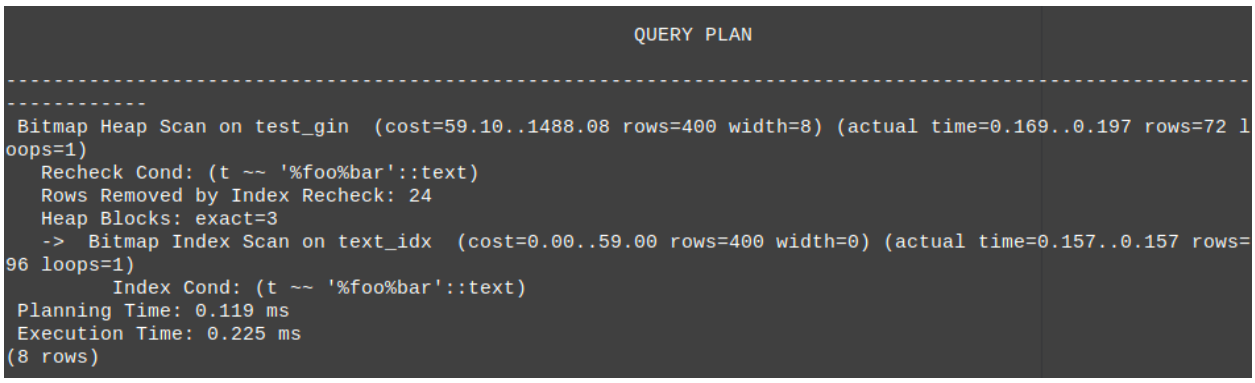

Рисунок 1.6 – Тестування індексованого відношення test_gin, GIN-індекс

Час виконання того самого запиту на вже проіндексованому відношенні test_gin значно менший, менший в цілих 434 рази.

Здатність створювати не одну точку входу для індексу GIN є і перевагою, і недоліком. Індекс часто містить кілька індексних записів на один рядок, що вставляється. Це важливо, щоб врахувати варіанти використання на які розраховано GIN, але спричиняє одну значну проблему: оновлення індексу це доволі “дорого”, і довго. Варто зважати на те, що оновлення одного запису може спричинити оновлення 10 або, в гіршому випадку, 100 індексних записів.

Враховуючи всі перераховані вище способи та методи пришвидшення видачі результуючої вибірки та процесінгу тексту, складемо перелік критеріїв за якими будемо їх між собою порівнювати задля коректного та зваженого підведення підсумків та постановки задачі.

У наведеній роботі в якості існуючих аналогів було наведено низку підходів:

- патерновий пошук;
- секціонування;
- повнотекстовий пошук.

Кожен із них має свої переваги та недоліки. Порівняння проводилось по наступним критеріям:

- швидкодія;

- навантаження на процесор;
- вплив на пам’ять;
- простота використання;
- оновлення структури бази даних;
- незалежність від СКБД;
- масштабованість.

Почнемо із патернового пошуку, який було розглянуто найпершим. Даний підхід виразний тим, що дозволяє чітко визначити та обробити слово чи текстовий набір по заданому патерну пошуку, і це, здавалось би, чи не найкраще рішення, але якщо говорити про його швидкодію, то воно, особливо, якщо робота проводиться на справді великих відношеннях, змушує бажати кращого. І так як швидкодія не дуже, то і процесору теж приходиться не солодко, звісно, різні СКБД по-своєму організують багатопоточність, що може врятувати ситуацію, але не коли мова йде про величезні об’єми інформації. Якщо казати про те, як патерновий пошук впливає на пам’ять комп’ютера, то практично ніяк, адже це просто пошук, процесінг та порівняння існуючих та записаних вже полів, він ніяк їх вміст не змінює. Використання патернового пошуку не можна назвати надто простим, звісно, якщо користуватися предикатами типу LIKE і ILIKE [2], то труднощів вони практично не викликають, щодо складнішого підходу: повноціного використання мови регулярних виразів, то це історія трохи складніша, і потребує певного часу на освоєння. Використання даного підходу ніяк не ускладнить оновлення структури бази даних. Предикат LIKE був доступний у стандарті SQL з самого початку його існування, і він є у всіх найпопулярніших системах керування базами даних. Щодо RegEx, то існує певна різниця в його реалізації для різних СКБД, і перенесення з однієї системи в іншу потребуватиме часу розробника, що не завжди буде дозволено загальними рамками проєкту. Зважаючи на все згадане вище, можна прийти до висновку, що масштабованість патернового пошуку не дозволяє вирішити поставлені у цій роботі задачі.

Наступним було розглянуто Partitioning, або секціонування. Його застосування доцільне, коли є велика таблиця, і її розбиття на партиції може пришвидшити час отримання результуючої вибірки та загалом обробку записів у ній. Швидкодія залежить від правильності індексування полів відношення.[8]. На щастя, нові версії більшості реляційних СКБД дозволяють зробити це майже безболісно. Секціонування не надто навантажує процесор, звісно, це залежить від величини партицій, структури, СКБД тощо. Тобто, даний підхід не дуже сталий та надійний, коли ми говоримо про процесорні навантаження. Використання цього підходу передбачує, що відношення має бути дискретним, адже його оновлення погано впливає на пам’ять комп’ютера, і багаторазове додавання, оновлення чи видалення записів може пошкодити пам’ять. Секціонування дозволяє обробку лише дискретного тексту, що не дуже підходить для проєктів інформація в яких постійно зазнає змін. Оновлення структури спряжене із труднощами, звідси можна зробити висновок, що цей підхід не дуже легкий у використанні. Незалежність від платформи тут присутня. Описане вище не дозволяє назвати даний підхід масштабованим.

Останнім було розглянуто повнотекстовий пошук, він же FTS [5], [6]. Дане рішення показало доволі хорошу швидкодію, як було продемонстровано на реальному прикладі. Навантаження на процесор мінімальні, якщо правильно проіндексувати відношення в БД. Із великими та складними типими даних найкраще себе показує узагальнений інвертований індекс(GIN).[10]. Значним недоліком цього індексу є те, що оновлення, в разі якихось змін у таблиці, можуть бути доволі “дорогими” та довгими. Використання даного підходу є порівняно простим, якщо грамотно користуватися документацією. Підхід до реалізації FTS різниться від одного СКБД до іншого, а тому незалежність від платформи видається трохи примарною, з іншого боку, якщо програмне рішення і розробляють під якусь конкретну СКБД, то перехід від неї до іншої явище рідкісне, і доволі дороговартісне, як і в плані людського ресурсу так і часового.

Виходячи із всього вище сказаного стає зрозумілим, що найкраще для цієї роботи підходить саме повнотекстовий пошук, і разом із ним чудово підходить застосування GIN індексу.

Нижче у таблиці 1.1 наведено порівняння розглянутих рішень:

Таблиця 1.1 – Критерії порівняння існуючих підходів обробки тексту в БД

Критерії порівняння існуючих підходів	Патерновий пошук	Секціонування	Повнотекстовий пошук
Швидкодія	-	±	+
Навантаження на процесор	+	±	-
Вплив на пам’ять комп’ютера	-	+	+
Простота використання	±	+	±
Оновлення структури бази даних	-	-	-
Незалежність від платформи	+	+	-
Масштабованість	-	-	+

1.3 Постановка задачі

Метою кваліфікаційної роботи є пошук та покращення методів пришвидшення отримання результуючої вибірки та обробки тексту в великих базах даних, а також створення застосунка для аналізу отриманих результатів. Дане рішення не повино сильно залежати від оновлення

структури бази даних, чи зміни вмісту записів у ній. Також воно повино бути доволі простим для розуміння та у використанні, щоб кожен, хто не розуміється на тонкощях тематики, міг ним скористатися.

Для досягнення поставленої мети мають бути вирішені такі задачі:

- проаналізувати чинники, що призводять до уповільнення обробки запитів у БД;
- провести порівняльний аналіз існуючих підходів обробки тексту в БД;
- розробити модель прискореного формування результуючої вибірки в БД;
- модифікувати метод повнотекстового пошуку задля пришвидшення пошуку тексту;
- підтвердити працездатність запропонованої моделі, чи комбінування методів повнотекстового пошуку із патерновим та вплив модифікації методу за допомогою існуючих можливостей реляційних СКБД;
- виконати експериментальне дослідження конкурентоспроможності розробленого рішення супроти існуючих;
- проаналізувати отримані результати.

Для реалізації цього рішення буде використано фреймворк PyQt, що дозволить забезпечити його платформонезалежність.Відповідно, весь програмний код буде написаний на мові програмування Python, що надає доступ до величезної кількості різноманітних бібліотек з допомогою яких розробка даного рішення стане ще легшою. Актуальність цього завдання зумовлена високою потребою в програмних рішеннях, що дозволяють швидко отримувати результат на основі текстового запиту, і при цьому не бути дуже складним у використанні.

2 ВИКОРИСТАНІ ТЕХНОЛОГІЇ

2.1 Вибір технологій

Наразі існує ціла низка реляційних систем керування базами даних, котрі дозволяють детально дослідити та розвинути питання, що зазначені у цій роботі, але я хотів би зупинитися на: MySQL, PostgreSQL, MariaDB та SQLite як на найбільш розповсюджених рішеннях. На таблиці 2.1 наведено порівняння цих СКБД:

Таблиця 2.1 – Порівняння існуючих СКБД

MySQL	PostgreSQL	MariaDB	SQLite
Реляційна СКБД	Реляційна СКБД	Реляційна СКБД	Реляційна СКБД
Java, PHP, C++, Python, Ruby, Visual Basic, Delphi, Go, R, Perl, Scheme, Tcl, Haskel and Eiffel	.Net, C, C++, Delphi, Java, JavaScript, (Node.js), Perl, PHP, Python, Tcl	Java, PHP, C++, Python, Ruby, Visual Basic, Delphi, Go, R, Perl, Scheme, Tcl, Haskel, Eiffel and Erlang	Delphi, C, C#, C++, Go, Java, JavaScript, LiveCode, Lue, Objective-C, Perl, PHP, Python, Ruby, Visual Basic and Xojo
Open source with commercial licenses with extended functionality available	Open source	Open source	Open source

Варто зазначити, що всі перелічені СКБД є реляційними, також всі вони підтримують велику кількість інтерфейсів для мов програмування.

MySQL погано себе почуває із великими наборами даних, також дана СКБД неефективно обробляє транзакції. Ще ця СКБД є частково платною, і якщо ви хочете отримати додаткові якісь важливі елементи, то без плати

нічого не вийшло би.

PostgeSQL має широкий набір засобів для роботи із індексами, можна підключати різні бібліотеки, а це означає, що це полегшить розробку та підтримку програмного коду в подальшому.

MariaDB має вищу швидкодію ніж MySQL, до того ж має відкритий вихідний код, SQLite можна не представляти, адже дана СКБД давно чудово себе зарекомендувала.

Всі перераховані СКБД мають добре написані інтерфейси для багатьох мов програмування, але я зупинюсь на Python-і, бо розробка на ньому швидка та відносно легка, що дозволяє більше приділяти увагу ідеям та концепціям, а не всім тонкощам розробки програмного коду. Саме тому у цій роботі буде використано три СКБД: PostgreSQL, MariaDB та SQLite.

Python надає змогу швидко та просто завантажити пакунки, що значно полегшать розробку застосунку для методів. Одним із таких є PyQt та похідні від нього.

2.2 Мова програмування Python

Python – інтерпретована об'єктно-орієнтована мова програмування високого рівня зі строгою динамічною типізацією. У технічному плані Python – це об’єктно-орієнтована мова програмування високого рівня з інтегрованою динамічною семантикою, в першу чергу для веб-розробки та розробки додатків. Дана мова програмування часто використовується у швидкій розробці додатків, оскільки пропонує можливість динамічного набору тексту та динамічну прив'язку.[12].

Мова програмування Python порівняно проста, тому її часто використовують для вивчення новачки, оскільки синтаксис Python, насамперед, фокусується на читабельності. Саме через цю простоту розробка програмного коду відбувається набагато швидше ніж в інших високорівневих мовах програмування. У свою чергу, це зменшує витрати на підтримку та

розробку програми. В свою чергу, це зменшує витрати на підтримку програми та розробку, також це дозволяє працювати різним командам розробників спільно без значних відміностей у досвіді та мові спілкування.

Крім того, Python підтримує використання модулів та пакетів, а це означає, що програми можуть бути розроблені в модульному стилі, а код може бути використаний повторно в різних проектах.[12]. Після розробки необхідного модулю або пакету, їх можна масштабувати для використання в інших проектах, тоді процес імпортизації та експортизації відбувається за простим та зрозумілим сценарієм.[12].

Однією з найбільш вагомих переваг Python є те, що стандартна бібліотека та інтерпретатор доступні безкоштовно, як у двійковій, так і у вихідній формі у вигляді коду. Також не існує ексклюзивності, оскільки Python та всі необхідні інструменти доступні на всіх основних платформах. Тому не потрібно турбуватися про оплату великих витрат на розробку.

Але у Python є і свої недоліки, одна із них – низька швидкодія виконання програми. Хоча, як було вказано вище, цей недолік компенсується зменшенням часу розробки програм. Показники такі, що програма, написана на Python, у 2-4 рази компактніша, ніж її аналог на C ++ або Java. При першому запуску програми відбувається збереження байт-коду (файли .pyc і .pyo), що дозволяє інтерпретатору не витрачати зайвий час на повторну перекомпіляцію модулів при кожному новому запуску, а подібне виконує, до прикладу, інтерпретатор мови Perl. До того всього, для збільшення швидкодії виконання програм існує спеціальна JIT-бібліотека psyco, але слід пам’ятати, що її використання може призвести до збільшення споживання оперативної пам’яті.

Також у Python відсутня статична типізація. Річ в тім, що в Python прийнята так звана «качина типізація». Саме по цій причині типи переданих значень будуть недоступні на етапа компіляції. Через це розробник може часто зіштовхуватися із виключенням типу AttributeError. Відсутність статичної типізації також є однією з основних причин низької швидкодії.

[12].

Також в Python повністю відсутня можливість модифікувати вбудовані стандартні класи: int, str, float, list та інші, щоправда, це забезпечує менше споживання оперативної пам'яті, відповідно і програма буде працювати швидше. Однією із головних причин такого обмеження є необхідність узгодження з модулями розширення. Величезна кількість модулів “кастуватимуть” елементарні типи до відповідних типів мови програмування С замість того, щоб маніпулювати над ними за допомогою С API.

2.3 Бібліотека PyQt для створення за стосунків із графічним інтерфейсом

Насамперед, потрібно дати визначення, що таке, власне, сам Qt. Qt – це набір міжплатформних бібліотек С ++, які реалізують високорівневі API для доступу до багатьох аспектів сучасних настільних та мобільних систем. Сюди входять послуги розташування та позиціонування, мультимедіа, підключення NFC та Bluetooth, веб-браузер на основі Chromium, а також традиційна розробка інтерфейсу користувача. Власне, сам PyQt – це повний набір прив’язок Python для Qt v5. Він реалізований як понад 35 модулів розширення і дозволяє використовувати Python як альтернативну мову розробки додатків для С ++ на всіх підтримуваних платформах, включаючи iOS та Android.

PyQt5 також може бути вбудований в додатки на основі С ++, щоб дозволити користувачам цих програм налаштовувати або покращувати функціональність цих програм. PyQt працює наступним чином: кожен загальнодоступний клас С ++ має wrapper клас в Python. Програміст Python працює з wrapper-ом, і wrapper внутрішньо викликає реальний об'єкт С ++.

PyQt реалізує близько 440 класів і понад 6000 функцій та методів, включаючи:

- значний набір віджетів графічного інтерфейсу;

- класи для доступу до баз даних SQL (ODBC, MySQL, PostgreSQL, Oracle, SQLite);
- QScintilla, віджет текстового редактора на основі Scintilla;
- інформаційні віджети, які автоматично заповнюються з бази даних;
- аналізатор XML;
- підтримка SVG;
- класи для вбудовування елементів керування ActiveX у Windows (лише в комерційній версії) [13], [14].

PyQt використовується для створення графічного інтерфейсу, розробки десктопних додатків за допомогою Python. Прикладами такого програмного забезпечення є: клієнт Dropbox для робочого столу, R Studio, Spyder, робочий стіл KDE (оснований на Qt).

2.4. Мова програмування SQL

SQL – мова запитів, що застосовують у програмах, призначені для управління даними, які зберігаються в реляційній системі керування базами даних (РСКБД), або для обробки потоку в реляційній системі керування потоком даних (РСКПД). Це особливо корисно при обробці структурованих даних, тобто таких, що включають відносини між сутностями та змінними[15].

SQL надає змогу отримати доступ до багатьох записів за допомогою однієї команди[15]. Також він створення SQL позбило розробників необхідності вказувати, як отримати запис, наприклад, з індексом або без нього.

Напочатку дана мова запитів була заснована на реляційній алгебрі та кортежному реляційному числення, SQL складається з багатьох типів виразів, які можуть бути неофіційно класифіковані як підмови, як правило: мова запитів даних (DQL), мова визначення даних (DDL), мова керування даними (DCL) та мова обробки даних (DML). Область SQL включає запит

даних, маніпулювання даними (вставка, оновлення та видалення), визначення даних (створення схеми та модифікація) та контроль доступом до даних. Хоча, SQL по суті є декларативною мовою (4GL), вона також включає процедурні елементи[15].

2.5 Система керування базами даних MySQL та застосунок для адміністрування баз даних phpMyAdmin

MySQL – компактний сервер баз даних із багатьма потоками. Можна охарактеризувати як високошвидкісний, стійкий та простий у використанні. MySQL вважається гарним рішенням для малих та середніх застосунків. Серцеві і коди сервера можуть компілюватися на багатьох платформах. Найповніше можливості сервера виявляються в UNIX-системах, де є підтримка багатопоточності, а це в свою чергу підвищує продуктивність системи в цілому. Можливості сервера MySQL:

- простота у встановленні та використанні;
- підтримується необмежена кількість користувачів, що одночасно працюються з базою даних;
- кількість рядків у таблицях може сягати близько 50 мільйонів;
- висока швидкість виконання команд;
- наявність простої і ефективної системи безпеки.

Так як даний застосунок створювався на машині із дистрибутивом Linux під назвою Manjaro, то, відповідно із Open Source підходом до програмного забезпечення та тим, що Manjaro є Arch-подібним, у роботі було використано відгалуження (форк) MySQL - MariaDB. Це також реляційна система керування базами даних, що поширюється під вільною та відкритою ліцензією GNU GPL.

phpMyAdmin – це безкоштовний програмний інструмент, написаний на PHP, призначений для управління та адміністрування MySQL. phpMyAdmin підтримує широкий спектр операцій на MySQL та MariaDB. Часто

використовувані операції (керування базами даних, таблицями, стовпцями, відношеннями, індексами, користувачами, правами, дозволами тощо).[7].

phpMyAdmin постачається з широким спектром документації, що постійно оновлюється спільнотою та безпосередньо розробниками. phpMyAdmin підтримується як LTR, так і RTL. Це проект із стабільною та гнучкою базою коду.

Застосунок має інтуїтивно зрозумілий веб-інтерфейс. Також підтримує більшість функцій MySQL:

- переглядати та видаляти бази даних, таблиці, представлення даних, поля та індекси; створювати, копіювати, видаляти, перейменувати та змінювати бази даних, таблиці, поля та індекси;
- сервер обслуговування, бази даних та таблиці, з пропозиціями щодо конфігурації сервера;
- виконання, редагування та закладка будь-яких SQL-висловлювань, пакетних запитів;
- керувати обліковими записами та привілеями MySQL;
- керування збереженими процедурами та тригерами.

Крім цього, у phpMyAdmin підтримується імпорт даних з CSV та SQL, експорт даних у різні формати(CSV, SQL, XML, PDF, ISO / IEC 26300), текст та електронна таблиця OpenDocument, адміністрація декількох серверів, створення графіка макета бази даних в різних форматах, створення складних запитів за допомогою запиту за прикладом (QBE), пошук у глобальному масштабі в базі даних або її підмножині, перетворення та збережених даних у будь-який формат за допомогою набору визначених функцій, наприклад, відображення даних BLOB як зображення або посилання для завантаження.

2.6 Система керування базами даних PostgreSQL та застосунок для адміністрування баз даних pgAdmin4

PostgreSQL – об’єктно-реляційна система керування базами даних.

Даний проект не контролюється якоюсь конкретною компанією, його розробка можлива лише завдяки співпраці багатьох людей та компаній. Сервер написано на мові програмування C. Розповсюджується у вигляді набору текстових файлів із вихідним кодом. Процес встановлення залежить від системи, що використовується та детально описаний в документації серверу.[11]. В даній СКБД доступні можливості, що також є в більшості інших систем керування базами даних[11]. Це такі можливості як:

- функції. Дозволяють виконувати деякий код безпосередньо сервером бази даних. Можуть виконуватися із такими ж привілеями, що і в користувача, або з привілеями користувача, що написав;

- індекси. Доступні такі типи: B-дерево, хеш, R-дерево, GIST, GIN. При власній потребі можна створити нові типи індексів;

- багатоверсійність. Дана СКБД здатна підтримувати одночасну модифікацію бази даних кількома користувачами за допомогою механізму Multiversion Concurrency Control. Завдяки цьому виконуються вимоги ACID (атомарність, узгодженість, ізольованість, довговічність), і практично відпадає потреба в блокуванні зчитування;

- типи даних такі, як і в більшості систем керування базами даних. Можливі незначні відмінності;

- об’єкти користувача. Це виключна можливість користувача розширити PostgreSQL задля власних потреб. Є можливість додавати: перетворення типів, нові типи даних, домени, функції, індекси, оператори, процедурні мови;

- успадкування. Таблиці здатні успадковувати характеристики та набори полів від інших таблиць, тих що є батьківськими по відношенню до них;

- тригери. Пов’язані з конкретною таблицею. Визначаються як функції. Є можливість написання різними мовами програмування.

pgAdmin4 – це веб-інструмент для адміністрування PostgreSQL. Даний застосунок написано на PHP[11].

Особливості :

- адміністрація декількох серверів;
- підтримка PostgreSQL 8.4.x, 9.x, 10.x, 11.x;
- здатність керувати всіма аспектами(користувачі та групи, бази даних, схеми, таблиці, покажчики, обмеження, тригери, правила та привілеї, огляди, послідовності та функції, розвинені об'єкти,звіти);
- просте маніпулювання даними(перегляд таблиць, представлення даних та звіти, виконання довільного SQL,вибір, вставка, оновлення та видалення);
- дамп таблиці в різних форматах: SQL, COPY, XML, XHTML, CSV, Tabbed, pg_dump;
- імпорт сценаріїв SQL, даних COPY, XML, CSV та Tabbe;
- простота установки та налаштування;
- розширюється за допомогою плагінів.

2.7 Формат обміну даними JSON

JSON – це легкий формат обміну даними. Легкий для писання та читання людиною, а машиною для аналізування та генерування. Повністю оснований на підмножині стандарту мови програмування JS ECMA-262 третього видання. JSON – це текстовий формат, який повністю не залежить від мови, це робить JSON ідеальною мовою для обміну даними[16].

JSON побудований на двох структурах:

- колекція пар ім’я/значення. У різних мовах це реалізується як об’єкт, запис, структура, словник, хеш-таблиця, список ключів або асоціативний масив;
- впорядкований список значень. У більшості мов це реалізується як масив, вектор, список або послідовність.

Це універсальні структури даних. Практично всі сучасні мови програмування підтримують їх[16]. Має сенс, щоб формат даних, який є

взаємозамінним з мовами програмування, також базувався на цих структурах.

У JSON вони набувають наступних форм:

- об’єкт це невпорядкований набір пар ім’я/значення. Об'єкт починається з {лівою фігурною дужкою і закінчується} правою фігурною дужкою. Кожне ім'я супроводжує: двокрапка, а пари ім'я / значення розділяються комою;

- масив це впорядкована колекція значень. Масив починається з [лівої квадратної дужки і закінчується] правою квадратною дужкою. Значення також розділяються комою;

- значенням може бути: рядок у подвійних лапках, число, true/false, null, об’єкт, масив. Ці структури можуть бути вкладеними;

- рядок - це послідовність, довжиною в нуль або більше символів Unicode, що обмежені подвійними лапками, із використанням екранованих скісних рисок. Символ представлений у вигляді одного символьного рядка. Рядок дуже JSON схожий на рядки в C або Java;

- число дуже нагадує число в C або Java, за винятком того, що восьмеричний та шістнадцятковий формати не використовуються;

- пробіл можна вставити між будь-якою парою лексем. За винятком кількох деталей кодування, що повністю описують мову.

2.8 Система керування базами даних SQLite

SQLite – це бібліотека на мові C, яка реалізує невеликий, швидкий, автономний, високонадійний, повнофункціональний механізм баз даних SQL. SQLite – це найбільш використовуване ядро баз даних у світі. Формат файлу SQLite стабільний, крос-платформний і зворотно сумісний. Файли бази даних SQLite зазвичай використовуються як контейнери для передачі багатого контенту між системами та як довгостроковий архівний формат даних[17].

Вихідний код SQLite знаходиться у загальнодоступному доступі, і

може використовуватися у будь-яких цілях.

SQLite має низку особливостей одна із яких це те що SQLite використовує незвичну систему типів для СКБД, сумісної з SQL: замість присвоєння типу стовпцю, як у більшості систем баз даних SQL, типи присвоюються окремим значенням; в мовному відношенні він динамічно набирається. Також, зазвий, таблиці в SQLite містять прихований стовпець рядового індексу, що забезпечує швидший доступ[17].

Кілька процесів або потоків можуть одночасно отримувати доступ до однієї бази даних. Кілька запитів на доступ для читання можуть бути виконанні паралельно. Запит на запис може бути виконаний лише в тому випадку, якщо наразі не обслуговується жоден інший запит на доступ. Інакше запит поверне відповідний код помилки.

3 РІШЕННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

3.1 Опис недоліків та їх рішення

3.1.1 Апаратна складова

Незважаючи на всі перераховані у другому розділі переваги повнотекстового пошуку він має цілу низку недоліків, які за певних оставин можуть зіграти неостанню роль у знижені загальної продуктивності застосунку.

Загальні причини зниження продуктивності можна поділити на:

- проблема із апаратними ресурсами;
- проблема із пакетуванням повного тексту;
- проблема заповнення індексу для повнотекстового пошуку.

Що стосується першого, то на продуктивність повнотекстового індексування та повнотекстових запитів впливають апаратні ресурси: пам’ять, швидкість диска, швидкість процесора та, власне, сама архітектура машини[18]. Основною причиною зниження продуктивності повнотекстового індексування є обмеження апаратних ресурсів:

- CPU. Якщо центральний процесор використовує, до прикладу, демона фільтрів (fdhost.exe), як у SQL Server, чи самого розміру буфера СКБД недостатньо для нормальної та комфортної роботи, або ж саме використання СКБД процесом наближається до 100 відсотків, то в такому випадку CPU може стати вузьким місцем[18].

- пам'ять комп’ютера. Якщо є дефіцит фізичної пам’яті комп’ютера, і її просто бракує для найпростіших операцій, чи маніпуляцій, то пам’ять може бути вузьким місцем.

- диск комп’ютера. Якщо середня довжина черги очікування диска більш ніж у два рази перевищує кількість голівок диска, то диск також може

стати вузьким місцем. Щоб обійти цю проблему, слід створити повнотекстові каталоги, які відокремлені від файлів і журналів бази даних, також варто створити повнотекстові каталоги на окремих дисках. Встановлення щвидших дисків і використання RAID також може допомогти покращити продуктивність використання індексування[19].

Проблема з пакетуванням повного тексту виникає, якщо система не має вузьких місць у апаратній складовій, ефективність індексування повнотекстового пошуку, здебільшого, залежить від наступного:

- кількість часу за який СКБД буде спроможна створити повнотекстові пакети;
- швидкість із якою демон фільтра може споживати ці пакети.

Проблема заповнення індексу для повнотекстового пошуку полягає в наступному:

- тип заповнення. На відміну від повного заповнення, поетапне, ручне та автоматичне заповнення не призначене для максимізації апаратних ресурсів для досягнення більшої швидкості. Таким чином, намагання налаштувати щось покращити в цій прогалині можуть не підвищити продуктивність для повнотекстового індексування[20], якщо воно використовує інкрементне, ручне або автоматичне відстеження змін;
- контрольне злиття. Після завершення заповнення запускається остаточний процес злиття, який об’єднує фрагменти індексу в один головний повнотекстовий індекс. Це призводить до покращення продуктивності запиту, оскільки потрібно запитувати лише головний індекс, а не кілька фрагментів індексу, а для рейтингу релевантності можна використовувати кращу статистику оцінки. Однак, контрольне злиття може бути інтенсивним введення-виведення, оскільки великі обсяги даних повинні бути записані та прочитані під час об’єднання фрагментів індексу, хоча, це не блокує вхідні запити[21];
- основне об’єднання великого обсягу даних може створити тривалу транзакцію, затримуючи скорочення журналу транзакцій під час контрольної

точки. У цьому випадку за моделі повного відновлення журнал транзакцій може значно збільшитися. Перед реорганізацією великого повнотекстового індексу в базі даних, яка використовує модель повного відновлення, слід переконатися, що журнал транзакцій містить достатньо місця для тривалої транзакції.

Для нівелювання перерахованих вище недоліків слід звернути увагу на те, щоб налаштувати продуктивність повнотекстових індексів. Для цього, насамперед, варто зрозуміти, чи є навантаження на центральний процесор, це можна перевірити кількома способами, і вони різняться в залежності від використовуваного СКБД.

Якщо основною операційною системою на якій стоїть СКБД є Linux, то достатньо скористатися вбудованою командою top, якщо СКБД висить в процесах, і використовує надто багато ресурсу процесора, то однозначно щось не так.

Якщо використовувати виключно можливості СКБД, то для MariaDB можна скористатися простими запитами, щоб зрозуміти, що використовується забагато ресурсу користувучем, цей запит подано нижче на прикладу 3.1:

Лістинг 3.1 – Оновлення статистики

```
SHOW USER_STATISTICS
```

Він поверне всю інформацію про всіх користувачів і дасть загальний огляд використання ресурсів[22]. Результати можуть допомогти помітити неадекватно високе використання одним користувачем ресурсів.

Щоб зрозуміти, що не так у PostgreSQL, слід увімкнути функцію повільного журналу запитів(slow log quiries) бази даних[23]. Це зафіксує всі повільні запити до журналу на основі порогового значення, яке визначає користувач. Увімкнення повільного журналу має дуже низький вплив на базу даних, тому, якщо поріг перевищує 2 секунди або вище, тоді не варто

хвилюватися, що це вплине на загальну продуктивність.

Переконавшись, що у системі існує проблема із використанням ресурсів, і загальним повільним виконанням повнотекстових запитів, варто провести підготовчі дії, які було перевірено та запропоновано мною, щоб максимізувати ефективність повнотекстових індексів, до цих дій входить:

для максимального використання процесорів або ядра ЦП, слід змінити налаштування. Від СКБД до СКБД вони можуть різнитися. Для SQL Server це sp_configure[24], і його параметр 'max full-text crawl range' слід встановити на кількість ЦП у системі. У цій роботі методи пришвидшення отримання результуючої вибірки та обробки тексту розглядаються на PostgreSQL та MariaDB. Щоб досягти максимального використання процесорів або ядра центрально процесора в PostgreSQL слід звернути увагу на shared_buffer[25], зазвичай рекомендується мати принаймні 25% пам’яті в інстансі. Якщо існує велике робоче навантаження, то варто збільшити його розмір до 40% від загальної пам’яті екземпляра, оскільки в самій документації PostgreSQL офіційно зазначається, що понад 40% швидше за все не покращить продуктивність[26]. Також слід звертати увагу на те, що, якщо було змінено значення shared_buffers, то слід також збільшити max_wal_size (журнал запису), щоб допомогти процесам запису використовувати більший кеш. У випадку із MariaDB варто перевірити, чи увімкнено InnoDB для обробки великої кількості одночасних з’єднань, для цього треба перевірити «PROCESSLIST», і якщо там є багато запитів у статусі «LOCK», це означає, що багато запитів призупинено, оскільки таблиці MyISAM обробляють інші транзакції. Щоб виправити це, потрібно перетворити ці таблиці на механізм InnoDB, який підтримує блокування на рівні рядків. Також не забуваймо про те що можна увімкнути постійні з’єднання – якщо у є лише одна програма, яка отримує тисячі з’єднань на годину, увімкнення постійних з’єднань може покращити продуктивність та зменшити навантаження на CPU[27]. Якщо сервер має кілька програм (наприклад, спільний сервер веб-хостингу), це може не працювати.

І спосіб, що може спрацювати для всіх СКБД це загальна оптимізація запитів до баз даних. Надто великі та складні запити потребують часу та процесорних ресурсів на виконання, і це стає причиною перевантаження ЦП. Щоб отримати перелік таких запитів можна переглянути slow query log[28], після цього постаратися зменшити кількість JOIN-ів, патернових пошуків тощо, це значно зменшить навантаження на процесор.

Також слід переконатися, що базова таблиця має кластерний індекс. Для першого поля кластерного індексу відношення цілочисельний типів даних є найоптимальнішим. Варто уникати використання “об’ємних” типів даних типу GUID чи BLOB у першому стовпчику кластерного індексу. Заповнення з кількома діапазонами за кластерним індексом може забезпечити найвищу швидкість заповнення. Цілочисельний тип даних, що служить повнотекстовим ключем матиме найвищу продуктивність, тому слід використовувати саме його.

Оновлення статистики базової таблиці з допомогою оператора UPDATE STATISTICS теж значно покращити роботу індексів Full-Text Search, що ще важливіше, оновлення статистики кластерного індексу або повнотекстового ключа для повної сукупності допоможе групі з кількома діапазонами генерувати “хороші” партиції в таблиці.

Перед повним заповнення на великому багатопроцесорному комп’ютері варто тимчасово обмежити розмір пулу буферів, установивши максимальне значення пам’яті сервера, щоб залишити достатньо пам’яті для процесу fdhost.exe, оптимізаторів, garbage collector-ів, використання операційної системи тощо.

Якщо використовується додаткове заповнення на основі поля з міткою часу, варто створити вторинний індекс у полі з міткою часу, щоб покращити продуктивність додаткового заповнення.

3.1.2 Програмна складова

Було розглянуто можливі методи прискорення видачі результуючої вибірки та обробки тексту у великих базах даних із використанням апаратної складової, але ще існують і різні програмні підходи. І перед їх введенням слід зрозуміти, що основним фактором, який впливає на продуктивність запиту та загальну швидкодію, є кількість даних, які повнотекстовий пошук має обробити перш ніж решта даних надійде в якості результуючої вибірки у реляційну систему. А тому, сам собою напрошується висновок, що для більшості реляційних систем керування базами даних існує можливість покращити продуктивність запиту, якщо відфільтрувати рядки з метою їх зменшення, щоб потім подати їх на вхід до Full-Text Seach системи, і цим самим прискорити обробку.

Індекси повнотекстового пошуку організовані інакше, ніж індекси бази даних, тому вирішення цієї задачі може викликати певні треднощі. Крім того, система повнотекстового пошуку та реляційна система працюють по-різному. На щастя, SQL у своєму арменалі має те, що цілком може задовольнити всі потреби - табличну функцію (TVF)[29]. Саме цей інструмент і буде використовуватися для раннього фільтрування рядків і зменшення кількості рядків, які потрібно обробити пізніше повнотекстовим пошуком для прискорення отримання результуючої вибірки та обробки тексту.

Для роботи із Full-Text Search(повнотекстовим пошуком) в PostgreSQL буде використано модуль pg_trgm[30], що у своєму просторі імен має функції та оператори для визначення літеро-цифрового тексту на основі зіставлення триграми, а також класи операторів індексів, які здатні швидко шукати подібні рядки. Цей модуль є “trusted”, тобто довіреним, це означає, що він може бути встановленим не суперкористувачем, який має привілей CREATE для поточної бази даних.

Щоб встановити його слід спочатку пересвідчитися, що в робочому

середовищі встановлено пакунок postgresql-contrib, тоді в консолі можна скористатися наступною командою, яку представлено у прикладу 3.2:

Лістинг 3.2 – Підключення модулю pg_trgm до PostgreSQL

```
CREATE EXTENSION pg_trgm;
```

3.2 Підведення підсумків аналізу проблеми

Отож, проаналізувавши та провівши дослідження мною було виявлено, вже, що головними перепонами повнотекстового пошуку на шляху до часової та ресурсної оптимізації є:

- кількість записів, що поступають на вхід для обробки;
- апаратна складова, що включає в себе перевантаження центрального процесора, пам’ять та диск комп’ютера, тип заповнення індексів.

Для нівелювання цих недоліків мною було розроблено метод, що адаптується під запити користувача та аналізує вхідні дані.

Як було зазначено вище основним фактором уповільнення роботи повнотекстового пошуку є кількість записів, які мають бути оброблені. Дане питання було вирішено з допомогою використання Text-Valued Function(таблична функція). Перед входженням даних у FTS вони будуть попередньо фільтруватися спеціально розробленим модулем, який також адаптується під запити та використовує ряд заздалегідь створених TVF функцій SQL.

Але також не слід забувати і про фактор уповільнення роботи FTS, що виникає через апаратну складову, і тут модернізація дещо складніше, але від того не менш реальна. Для пришвидшення часу роботи FTS було зроблено наступне:

- переглянуто значення параметра shared_buffer для PostgreSQL. зазвичай рекомендується мати принаймні 25% пам’яті в інстансі. Але якщо

існує велике робоче навантаження, то варто збільшити його розмір до 40% від загальної пам’яті екземпляра, оскільки в самій документації PostgreSQL офіційно зазначається, що понад 40% швидше за все не покращить продуктивність. Також слід звертати увагу на те, що, якщо було змінено значення shared_buffers, то слід також збільшити checkpoint_segments, щоб розподілити процес запису великої кількості нових або змінених даних протягом більш тривалого періоду часу. Але, коли відбуватиметься повне заповнення індексу повнотекстового пошуку слід обмежити значення цього параметру до стандартних 25%, щоб було достатньо пам’яті для системи та garbage collector-ів;

- кожна таблиця повинна мати кластерний індекс. Цілочисельний тип даних, що служить повнотекстовим ключем, матиме найвищу продуктивність, тому і було використано саме його;
- також було впроваджено автоматичний механізм оновлення статистики базової таблиці з допомогою оператора UPDATE STATISTICS. Це допоможе оновлювати статистику кластерного індексу.

3.3 Загальний опис програми

Програма умовно поділяється на три частини: інтерфейс користувача, обробка запитів користувача методами, які надають змогу швидше отримувати результуючу вибірку та обробляти текст та обробка результатів для подальшої їх систематизації та статистичної обробки. Не останню роль в цьому переліку займає саме інтерфейс користувача, адже він має бути зрозумілим і інтуїтивним для будь-кого, хто буде користуватися застосунком. Не менш важливою є робота та обробка запитів користувача, адже чим краще буде продумано структуру та архітектуру методів та програми, тим швидшою буде робота застосунку та тим менше буде потрібно часу на обробку тексту та отримання результуючої вибірки. У третій частині слід розробити відповідні класи для систематизованої та узгодженної акумуляції

статистичних даних з метою подальшої їх обробки та аналізу досліджень.

3.4 Початок роботи

Перед початком потрібно встановити всі необхідні python-пакунки, що знадобляться для розробки програмного продукту. Перед цим, насамперед, потрібно створити віртуальне оточення “venv_passwd_mng” для того, щоб зменшити можливість виникнення конфліктів та колізій між вже встановленими пакунками Python. До того ж використання віртуального оточення дає змогу власноруч вказати, яку саме версію пакунку потребує реалізація конкретно цього проекту, можна, навіть, вказати версію Python. І директорія проєкту не буде переповнена непотрібними python-пакунками. Після налаштування віртуального оточення слід скористатися менеджером пакунків pip для встановлення необхідного програмного забезпечення. Для розробки знадобляться наступні пакунки:

- PyQt5 – пакунок для роботи із PyQt;
- psycopg2 – пакунок для роботи із PostgreSQL;
- mariadb – пакунок для роботи із MariaDB;
- pyqtgraph – пакунок для роботи із статистичними діаграми графічної бібліотеки з відкритим вихідним кодом

3.5 Підключення до БД та головне вікно застосунку

У цьому розділі було розроблено модуль який дозволяє безперешкодно підключатися, що до PostgreSQL та MariaDB, і робити це не захаращуючи програмний код. Ідея полягає в тому, щоб створити єдину точку входу через яку можна буде без проблем підключатися та перепідключатися до різних БД, при цьому, щоб ці підключення були доступні із будь-якого місця програми та через один єдиний екземпляр, що завжди буде знаходитися в пам’яті програми. Це було досягнуто реалізацією класу під назвою

DatabaseConn, даний клас представляє собою один із патернів проектування під назвою Singleton[31]. Цей патерн обмежує створення класу одним єдиним об'єктом. Це тип патерну створення і включає лише один клас для створення методів та зазначених об’єктів. Він забезпечує глобальну точку доступу до створеного екземпляра, графічно цей підхід представлено на рисунку 3.1 :

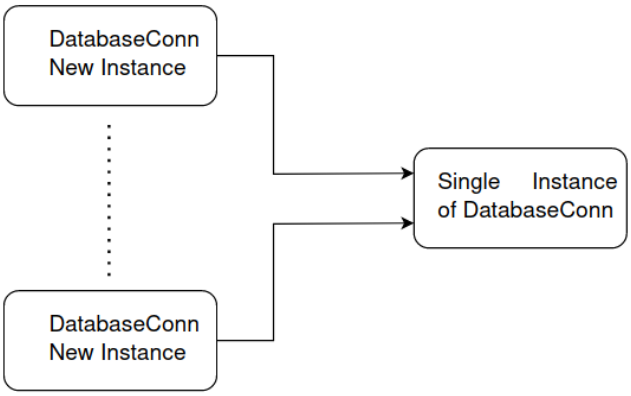

Рисунок 3.1 – Патерн проєктування Singleton

Це один із найпростіших патернів, але один із найбільш вживаніших, адже забезпечує менше використання ресурсів комп’ютера за рахунок зменшення використання пам’яті програмою (приклад 3.3).

Лістинг 3.3 – Програмний код для патерну проєктування Singleton

```
class DatabaseConn:
    __instance = None
    @staticmethod
    def getInstance():
        if DatabaseConn.__instance is None:
            DatabaseConn()
        return DatabaseConn.__instance

    def __init__(self):
        if DatabaseConn.__instance is not None:
            raise Exception(f'This class is Singleton!')
          else:
               DatabaseConn.__instance = self
               self.__which_db = db_name
               self.__params_dict = db_params
               self.db_manipulators()
```

Здатність програми до серіалізації курсорів різних СКБД було досягнуто з допомогою патерну проєктування під назвою Фабрика. Кожен шаблон Фабрика містить: конкретну реалізацію та створювач, що і ініціалізує цю конкретну реалізацію. Даний шаблон використовується для створення конкретних реалізацій загального інтерфейсу. Він відокремлює процес створення об’єкта від коду, який залежить від інтерфейсу об’єкта[32]. Наприклад, програмі для виконання своїх завдань потрібен об’єкт із певним інтерфейсом. Конкретна реалізація інтерфейсу визначається деяким параметром. Замість використання складної умовної структури if/elif/else для визначення конкретної реалізації програма делегує це рішення окремому компоненту, який створює конкретний об’єкт. Завдяки такому підходу код програми спрощується, що робить його зручнішим для повторного використання та зручнішим у розробці та підтримці. Перетворення об'єкта в інше представлення часто називається серіалізацією, рисунок 3.2.

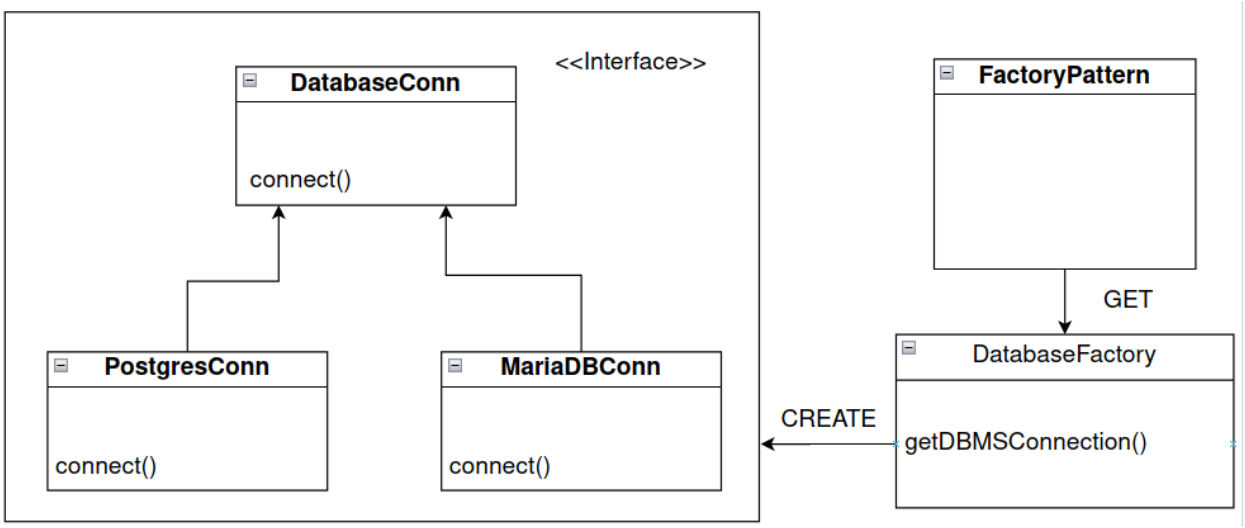

Рисунок 3.2 – Патерн проєктування Manufacture для створення екземпляра

Для кожної реалізації було написано окремий клас, так простіше керувати поведінкою кожного із типів підключення. Тоді, при потребі, кожен раз викликається метод класу-серіалізатора, який повертає реалізацію в залежності від параметра переданого методу. Лістинг 3.4 нижче:

Лістинг 3.4 – Метод серіалізатор головного класу шаблону проеєтування Manufacture

```
def make_serialization(self):
     if self.connection_mode == 0:
          temp_conn = PostgreSerialization()
          return  temp_conn
     elif self. connection_mode == 1:
          temp_conn = MariaDBSerialization()
          return  temp_conn
```

Головне вікно представляє собою клас, що має наслідування від класу QMainWindow PyQt, і є нічим іншим як головною областю звідки можна виконувати пошук тексту в таблицях бази даних до якої підключився користувач. Тут розташована область для редагування тексту, де можна ввести та виконати запит до самої БД.

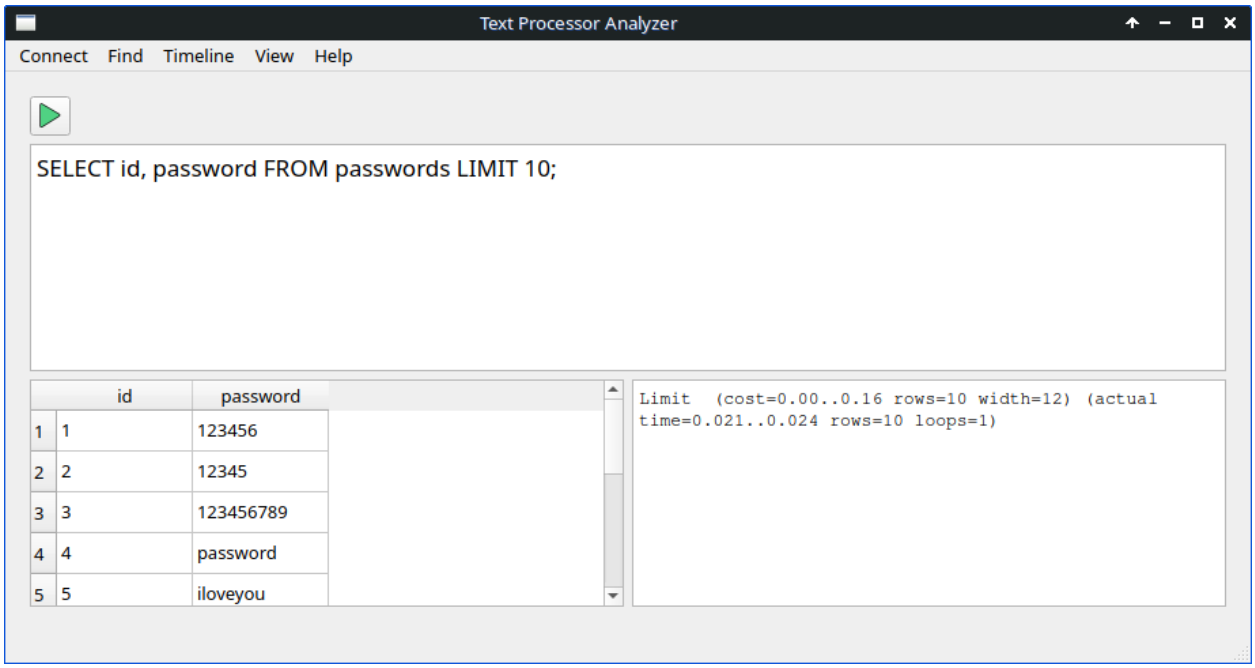

Рисунок 3.3 – Патерн проєктування Manufacture для створення екземпляра

Після запуску на виконання з допомогою відповідної кнопки, що знаходиться вище ліворуч від самої області, можна побачити або виключення від самої СКБД, або результат виконання у двох областях, що розташовані нижче. Це область результату запиту та область із виконаним предикатом

EXPLAIN ANALYZE[33]. Перша представляє собою QTableView, і дозволяє вивести у собі результуючу вибірку, а друга є звичайним QPlainText, і дозволяє виводи текст із поясненням щодо виконаного запиту. Загальний вигляд головного вікна представлено на рисунку 3.3.

Область результуючої вибірки реалізовано з допомогою QTableView. QTableView реалізує загальний вигляд таблиці, який відображає елементи з моделі. У цьому випадку моделі QabstractTableModel. Цей клас використовується для стандартних таблиць, які раніше були класом QTable, але з використанням більш гнучкого підходу.

QTableView реалізує інтерфейси, визначені класом QAbstractItemView, щоб дозволити йому відображати дані, надані моделями, похідними від класу QAbstractItemModel. Код хедера таблиці наведено нижче в лістингу 3.5:

Лістинг 3.5 – Вивід назви полів таблиці

```
 def headerData(self, section, orientation,
role=Qt.DisplayRole):
        self.header_labels = list()
        for i in self._data[0]:
                self.header_labels.append(i[0])
        if role == Qt.DisplayRole and orientation ==
Qt.Horizontal:
            return self.header_labels[section]
        return QtCore.QAbstractTableModel.headerData(self,
section, orientation, role)
```

3.6 Методи обробки тексту

Реалізація методів, які використовує програма, щоб адаптуватися під запити користувача та дати максимально задовільний результат є не менш важливою частиною даної роботи.

Як і було зазначено вище підхід до реалізації цих методів залежить від розміру самих віднощень. Їх можна розділити на 3 класи:

- мала таблиця, відношення величиною менше 100000;
- середня таблиця, відношення величиною більше 100000 та менше

1000000;

- велика таблиця, відношення величиною 1000000 та більше.

Для більш зручного процесу розробки та підтримки коду було використано патерн проєктування під назвою Фабрика. Даний патерн дозволив якнайліпше підійти до розробки цього модуля застосунка. Існує один клас серіалізатор під назвою QueryAnalyzer, у собі від містить метод get_analyzed_data, що перевіряє розмір відношення та створює екземпляри класів для кожного із методів, його код представлено нижче в лістингу 3.6:

Лістинг 3.6 – Метод серіалізатор методів обробки тексту

```
 def get_analyzed_data(self):
        temp_result = None
        if self.table_records_num in range(0, 100000):
            temp_result = SmallTableAnalyze(self.query,
self.table, self.table_records_num)
        elif self.table_records_num in range(100000, 1000000):
            temp_result = MediumTableAnalyze(self.query,
self.table)
        elif self.table_records_num >= 1000000:
            temp_result = LargeTableAnalyze(self.query,
self.table)

        return temp_result
```

Даний метод повертає аналізатор для кожного із запропонованих у цій роботі методів:

- малий аналізатор, аналізує записи, використовуючи звичайну окрему структуру INDEX у відношенні, не потребує додаткових підготувань, адже розмір відношення зовсім малий для відчутної різниці між методами для більших таблиць;

- середній аналізатор, використовує окрему структуру БД GIN індекс;

- великий аналізатор, як і середній використовує GIN, але також і Table-Valued Function, для фільтрації вхідної вибірки.

Для третього методу було спеціально розроблено Table-Valued Function, яка спрацьовує при обробці великих відношень. SQL код для цього наведено нижче у лістингу 3.7:

Лістинг 3.7 – Метод серіалізатор методів обробки тексту

```
create or replace function get_dataset (user_pattern VARCHAR)
returns table(record_value VARCHAR)
as $$
begin
     return query
     select
          {self.user_field}
     from
          {self.user_table} where
     {self.user_field} like '%' || user_pattern || '%';
end
$$ language plpgsql;
```

Де table_field та user_table це поле таблиці та сама таблиця відповідно, яку передає користувач на вхід. Далі результуюча таблиця поступає на вхід до повнотекстового пошуку.

Середні відношення обробляються GIN для PostgreSQL та FULL INDEX для MariaDB структурами. Які також при спрацюванні методу або просто відпрацьовують, або створюються як нові структури, і тільки після цього спрацьовують.

Розробка цих методів була простішою ще по тій причині що однією із найважливіших особливостей Qt є так звані сигнали та слоти[34], у даному проєкті вони активно використовуються. Це дуже зручний та зрозумілий механізм. Полягає він у тому, що сигнали та слоти використовуються для зв'язку між об'єктами. Сигнал спрацьовує, коли відбувається певна подія. Віджети Qt мають багато заздалегідь визначених сигналів, але завжди можна створити підклас віджетів, щоб додати до них власні сигнали. Слот – це функція, яка викликається у відповідь на певний сигнал. Віджети Qt мають багато заздалегідь визначених слотів, але можна створити підклас віджетів та додати власних слотів, щоб можна було обробляти кастомні сигнали та події. При створенні з’єднання із базою даних СКБД спрацьовує сигнал для ініціалізації Singleton та параметрів БД.

Для того, щоб перевірити ці методи на практиці було створено спеціальне пошукове вікно типу QDialog, в ньому можна обрати таблицю в

якій буде проведено пошук потрібного значення, а також можна задати наступні параметри пошуку:

- враховувати реєстр тексту;
- пошук ідентичного тексту в полях.

Загальний вигляд вікна представлено нижче на рисунку 3.4 :

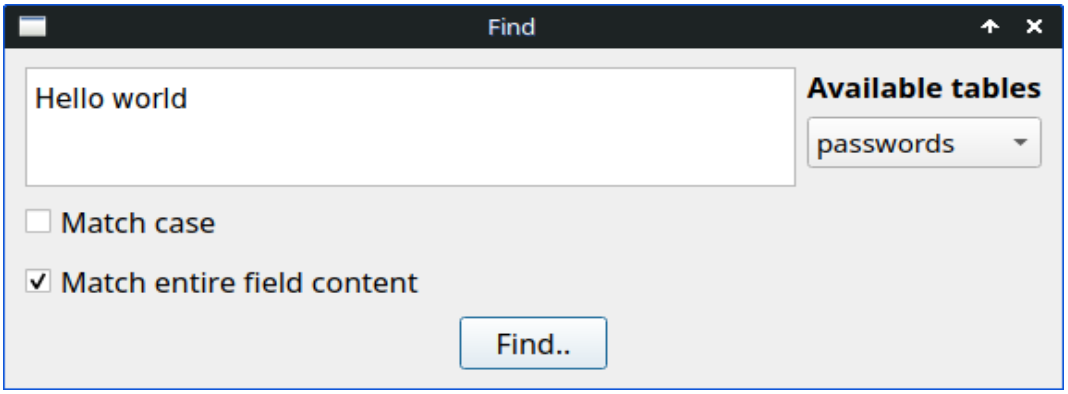

Рисунок 3.4 – Вікно обробки запиту користувача

Після вибору цільового відношення та параметрів пошуку слід натиснути кнопку пошуку, після цього відбувається аналіз запиту та його серіалізація між методами пошуку. Загальний вигляд патерна Manufacture для QueryAnalyze представлено нижче на рисунку 3.5 нижче:

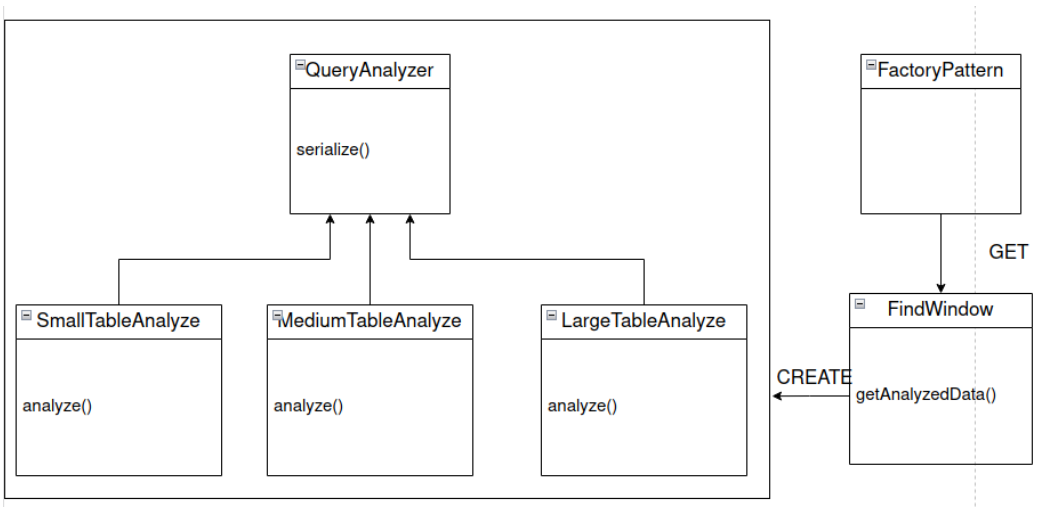

Рисунок 3.5 – Патерн проєктування Manufacture методів обробки тексту

Реалізація вище вказаних методів для MariaDB мало чим відрізняється від такої реалізації для PostgreSQL. MariaDB також має спеціальний індекс, що відрізняється від B-Tree індекса за структурою, і призначений для повнотекстового пошуку, це FULLTEXT INDEX[35]. Повнотекстовий індекс у MariaDB – це індекс типу FULLTEXT, і він надає такі можливості як:

- повнотекстові індекси можна використовувати лише з таблицями MyISAM, Aria, InnoDB і Mroonga, і їх можна створювати лише для полів CHAR, VARCHAR або TEXT;

- партиції не можуть містити повнотекстові індекси, навіть якщо система зберігання їх підтримує;

- для великих наборів даних набагато швидше завантажити дані в таблицю, яка не має індексу FULLTEXT, а потім створити індекс після цього, ніж завантажувати дані в таблицю, яка має індекс FULLTEXT.

Загалом, керування двома СКБД в програмі відбувається за рахунок використання створеного Singleton, в цьому класі містяться кредити входу, і, власне, назва самої СКБД. Виходячи із цих даних, всі наступні класи програми, які використовують відношення бази даних, спираються на ці дані та далі обробляють дані у відповідності із робробленими для кожної СКБД методами.

3.7 Обробка запитів та статистика

Обробка запитів користувача відбувається з допомогою класу ResultParser. Даний клас отримує на вхід запит, ім’я таблиці та результат виконання оператора EXPLAIN ANALYZE. Після цього результат статистичний результат роботи запиту записується в SQLite таблицю, яка і слугує статистичним акумулятором застосунку. Реалізація обробки запиту користувача наведено нижче у лістингу 3.8:

Лістинг 3.8 – Методи парсингу запиту користувача

```
def execute_time(self):
        temp = self.explain_query[-1][0].split(': ')[-1]

        return temp

    def execute_plan(self):
        temp = self.explain_query[0][0].split('  ')[0]
```

Вставку у таблицю наведно у лістингу 3.9:

Лістинг 3.9 – Вставка запису у статичтичну таблицю

```
 def write_statistics(self, data_set):
        try:
            placeholders = ', '.join(['%s'] *
len(self.data_set))
            columns = ', '.join(data_set.keys())
            placeholders = ', '.join(['?'] * len(data_set))
            sql = "INSERT INTO %s ( %s ) VALUES (%s)" %
(self.table_name, columns, placeholders)

            self.db_cursor.execute(sql, list(data_set.values()))
            self.db_connector.commit()
            self.db_connector.close()
        except Exception as error:
            print(f'Error: {error}', sql)
```

Таблиця, якій відведена роль записувати та зберігати статистичні дані програми має наступну структуру:

- ID, первинний ключ та унікальний ідентифікатор статистичного запису;
- query, сам запит користувача;
- table_size, розмір таблиці бази даних;
- dbms, вказує до якої СКБД належала таблиця на момент виконання запиту;
- exec_time, час виконання запиту;
- analyze_plan, план з допомогою якого планувальник виконав би вибірку із таблиці;
- exec_date, дата та час виконання запиту.

З допомогою цієї таблиці програма читає та записує статистичні дані застосунку.

3.8 Статистичне представлення даних

Статистично дані про обробку запитів користувача програмою можна представити з допомогою функції Timeline > Graph. Після вибору цього підменю відкриється вікно із можливими часовими рамками за які можна подивитися статистику користувача. Доступні опції: день, вчора, тиждень тому, місяць тому, весь час. Після вибору періоду достатньо натиснути на кнопку підтвердити, після цього користувачеві відкриється вікно із статистикою, і він зможе її переглянути. Якщо у таблиці нічого немає, то буде виведено просто білий графік.

Метод для читання записів та їх обробки представляє собою звичайний запит до БД, його виконання та отримання результату, приклад. Для створення вікна із статистичними графіками було використано бібліотеку pyqtgraph. Його програмна реалізація представляє собою написання ініціалізатора для головних елементів та стилів, і безпосередню імплементацію самого об’єкта в програмі, лістинг 3.10

Лістинг 3.10 – Метод ініціалізації вікна статистики

```
 x = [1, 2, 3,4 ,5]
        xdict = {1: '7000 kB', 2: '28 MB', 3: '39 MB', 4: '50
MB', 5: '790 MB'}
        pen_1 = pg.mkPen(color=(255, 0, 0))
        pen_2 = pg.mkPen(color=(-0, 255, 0))
        name_1 = "PostgreSQL"
        name_2 = 'MariaDB'
        stringaxis = pg.AxisItem(orientation='bottom')
        stringaxis.setTicks([xdict.items()])
        self.graphWidget.setAxisItems(axisItems = {'bottom':
stringaxis})
        self.graphWidget.addLegend()
        self.graphWidget.plot(x, mean_time_postgres,
name=name_1,  pen=pen_1, symbol='+', symbolBrush=('r'))
        self.graphWidget.plot(x, mean_time_maria, name=name_2,
pen=pen_2, symbol='+', symbolBrush=('b'))
```

Вікно вибору часового проміжку представлено нижче на рисунку 3.6:

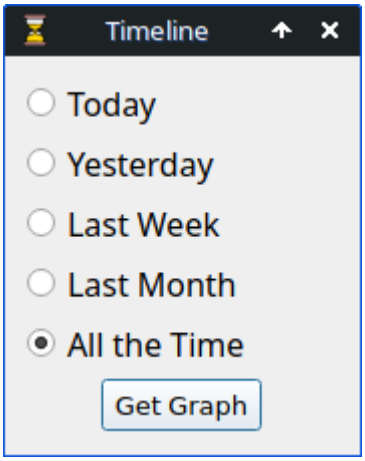

Рисунок 3.6 – Вибір часового проміжку програми

Можвилий вигляд вікна статистики без вхідних даних наведено нижче на рисунку 3.7:

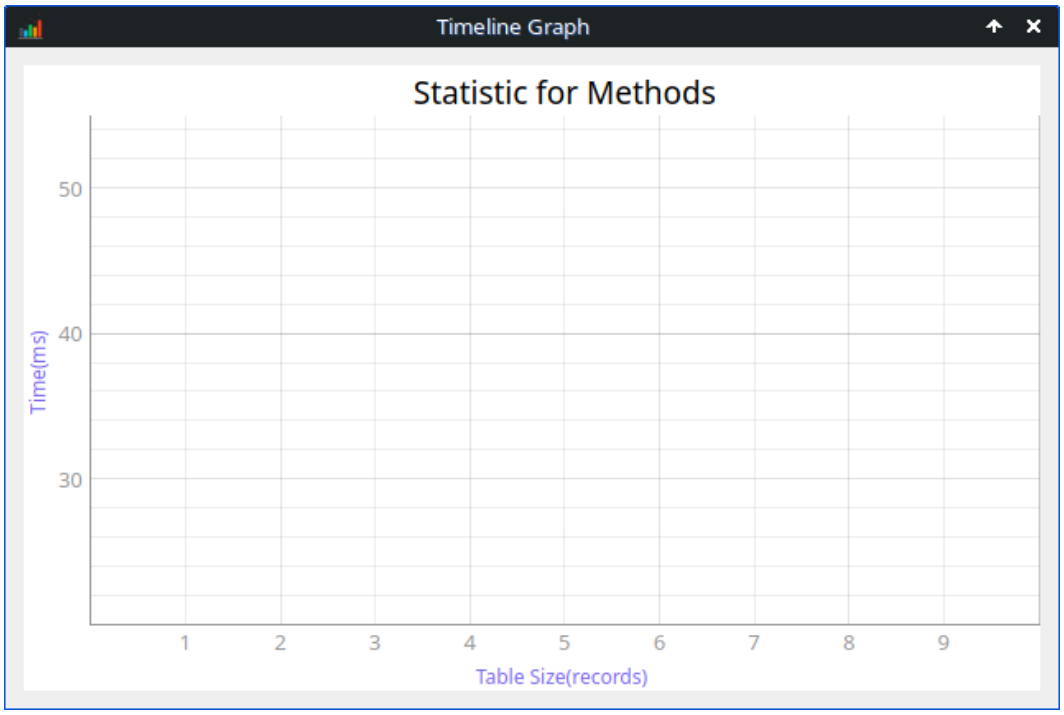

Рисунок 3.7 – Патерн проєктування Manufacture методів обробки тексту

Зважаючи на все, було б логічно представити цю роботу у вигляді IDEF0 діаграми. Дану діаграму наведено нижче на рисунку 3.8:

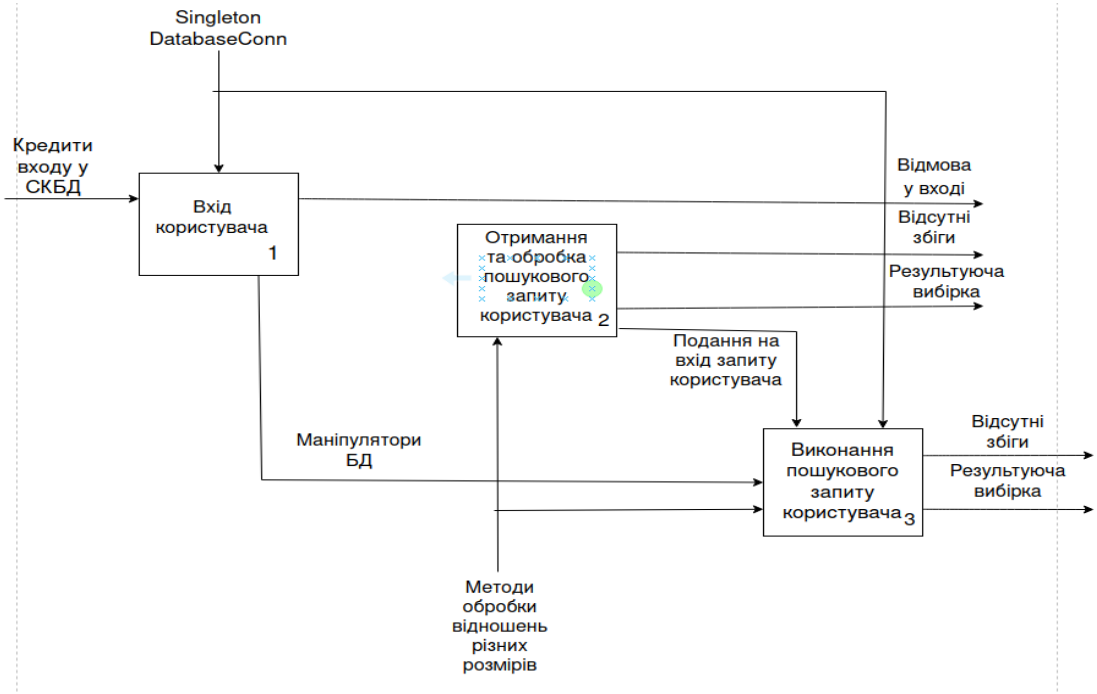

Рисунок 3.8 – IDEF0 діаграма застосунку

Даний підхід вирізняється простотою реалізації та простотою підтримки програмного коду, адже всі ключові складові та модулі ієрархічно розбито, що дозволяє:

- швидко при потребі змінювати структуру;
- переписувати або замінювати наявні рішення програми;
- робити зміну у методології підходів обробки тексту програми.

Мною було запропоновано поєднати патерновий пошук із повнотекстовим, при цьому значно модернізувавши, що перший, що другий підходи. Також простота полягає в тому, що всі отриманні данні одразу конфертуються у статистичні дані, а потім вже обробляються програмою.

4 АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

4.1 Опис гіпотези

Для наочної демонстрації дієвості запропонованого метода слід розглянути існуючі методи дослідження та обрати, який саме якнайліпше підходить для цієї цілі.

Одним із найпоширеніших методів наукового дослідження є емпіричний метод, він полягає в:

- спостереженні;
- вимірюванні;
- моделюванні;
- прогнозуванні;
- перевірці прогнозу.

Його основними формами є спостереження та експеримент. До них відносяться також численні вимірювальні процедури, ці процедури тісно пов’язанні із теорією, хоча і здійснюються в парках емпіричного пізнання, а особливо в рамках експерименту.

Основною відмінністю спостереження від експеримента є те, що спостереження це фіксація з допомою різних приладь, органів чуттів тощо природнього перебігу якихось процесів, а експеримент це штучне творення, активне та цілеспрямоване втручання у перебіг досліджуваного процесу. У ході дослідження сам об’єкт повністю ізолюється від зовнішнього впливу, і представляється у чистому вигляді. При цьому експериментатор задає конкретні умови, контролює їх, модернізує та багаторазово відтворює. І завжди будь-який експеримент керується на направляється якоюсь концепцією, гіпотезою або ідеєю. Саме шляхом ізоляціє об’єкта, його активна зміна та перетворення, саме це дозволяє висовувати, підкріпляти та доводити наукові гіпотези.

Врахочуючи все вище сказане ми приходимо до висновку, що найкраще для цього дослідження підходить саме метод наукового дослідження експеримент, адже він дозволяє багаторазово підтворювати та ізолювати систему для перевірки ідеї, гіпотези тощо.

Саме тому наступним, що ми зробимо, буде формулювання та висування гіпотези, яку ми доведемо провівши експерименти над створенною у попередніх розділах програмою та методами.

Гіпотеза: застосування запропонованих мною методів пришвидшення видачі результуючої вибірки та обробки тексту, і правда пришвидшує обробку та видачу результуючої вибірки у великих базах даних.

4.2 Опис експерименту

Для доведення висунутої у попередньому підрозділі гіпотези ми проведемо експеримент, який полягатиме у тому, що на вхід бази даних буде поступати запит. СКБД має ж обробити його, вирішити до якої категорії відноситься відношення, яке перевіряє користувач, та застосувати до нього відповідний розроблений заздалегідь метод пришвидшення отримання результуючої вибірки та обробки тексту. Для того, щоб переконатися, що ці методи працюють коректно та правильно спочатку слід провести експерименти із використанням не покращених методів та функцій бази даних. У випадку із патерновим пошуком це звичайнісінький пошук по оператору LIKE/ILIKE та більше нічого. Для повнотекстовго пошуку це не оновленна статистика користувача та абсолютне ігнорування Table-Valued Fucntions, які непогано покращують кінцевий результат обробки.

Щоб результати експерименту були не надто однобокими було вирішено скористатися як мінімум п’ятьма відношеннями різного розміру:

goodreads_books містить опис та назву книг, що були додані на читацький інтернет ресурс Goodreads;

- twitter_twits твітти звичайних користувачів соціальної мережі

Твіттер;

- reddit дописи користувачів соціальної мережі Reddit;
- passwords найрозповсюдженіші паролі в світі;
- mailbox величезна колекція електронних листів.

Як можна було зрозуміти по контексту, всі ці відношення йдуть від найменшого до найбільшого. У найменшому 10000 записів, а у найбільшому більше шести мільйонів, всі вони містять у собі текстове поле швидкість обробки якого ми і будемо досліджувати.

Кожне відношення “підлягає” перевірці не менше 20 разів, далі для конкретної табличної розмірності обчислюється середнє арифметичне, і на основі розміру таблиці та швидкості її обробки і будується графік. Також цей експеримент буде проводитися на двох системах керування базами даних: PostgreSQL та MariaDB, і виходячи із цього можна буде з точністю сказати, що даний підхід та методи релевантні для кількох різних систем, і що він є переносним.

Набори даних було взято із веб-ресурсу Kaggle, він надає широкий доступ до бази із величезними наборами даних прелсиавлених у різних форматах файлів.

4.3 Проведення експерименту

Для полегшення проведення експерименту було сформовано вхідні набори даних для кожного відношення. Спочатку серію експериментів було проведено без удосконалення системи. Відповідно, і результати змушують бажати кращого. У більшості ітерацій експерименту СКБД використовує Seq Scan за неможливості скористатися чимось іншим, звідси і низька швидкість видачі результуючої вибірки та обробки тексту, і чим більшим стає відношення, тим більш це відчутно.

Експеримент проводився на двох СКБД. Для виведення графіка результату було використано timeline_graph модуль, що дозволяє на основі

часового проміжку отримувати статистичні дані користувача. Результат проведення цього експерименту наведено нижче на рисунку 4.1:

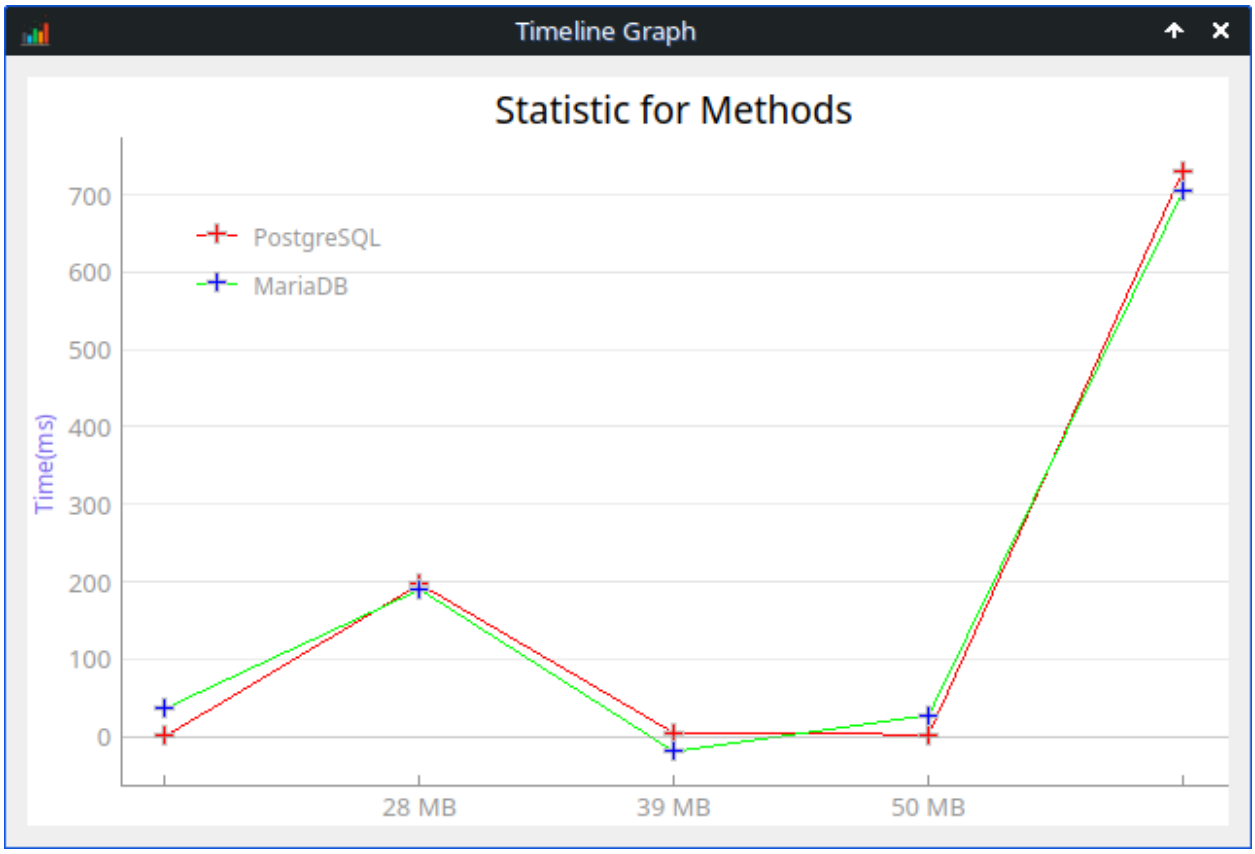

Рисунок 4.1 – Результат першого експерименту з використанням не удосконалених методів

Перші дві таблиці входять у категорію малі відношення, наступні дві це середні, а остання це велике відношення. Відповідно, до перших двох було застосовано неудосконалений метод патернового пошуку, до наступних повнотекстовий пошук.

Як бачимо із графіка, розмір має значення, і чим більше відношення, тим більше потрібно часу на обробку запиту та формулювання результуючої вибірки. Першу відношення практично не потребує ніяких корективів, адже часу його обробки на досить високому рівні. Питання тільки, а чому друге відношення twitter_twits потребує стільки часу на отримання результату. Все дуже просто, дане відношення розміром майже сто тисяч, тоюбто, воно

майже підпадає під визначення середнього в рамках даного експерименту, але ні, і саме тому воно не використовує більш ефективну структуру даних БД, яка допомогла б йому швидше повертати результуючу вибірку. Що в PostgreSQL, що в MariaDB тут є проблема. Наступних два запити також не потребують багато процесорних ресурсів та часу. А ось час роботи третього летить в небеса, адже його середнє арифметичне перевищує 700ms, що є дуже і дуже багато. Ці результати зумовлено тим, що у старих методах не враховується широкої роботиіз індексуванням, що значно може покращити продуктивність роботи БД. Також це зумовлено не до кінця продуманою роботою із апаратурою.

Результат вимірювання часу обробки відношень наведено нижче на таблиці 4.1:

Таблиця 4.1 – Результати першого експерименту

Результати вимірювання	small_1(мала)	small_2(мала)	medium_1(середня)	medium_2(середня)	large_1
Середній час обробки відношення (PostgreSQL)	0.80695	196.99	4.367	2.4	731.3288
Середній час обробки відношення (MariaDB)	1.04695	180.84495	5.798	2.7349	755.0453

MariaDB працює практично так само, як і PostgreSQL, але в деяких місцях із невеликим відставанням, що може бути пов'язано із низкою факторів:

- MariaDB підтримує курсори, але вони в режимі read-only, що може потребувати додаткового часу на обробку результату;
- Повнотекстовий пошук краще парцює в PostgreSQL ніж в MariaDB.

Тепер проведемо наступний експеримент тільки вже із покращеними методами обробки тексту та пришвидшеного отримання результуючої вибірки. Цей експеримент проводився за аналогією, як і другий. Результат його виконання можна побачити на рисунку 4.2 нижче:

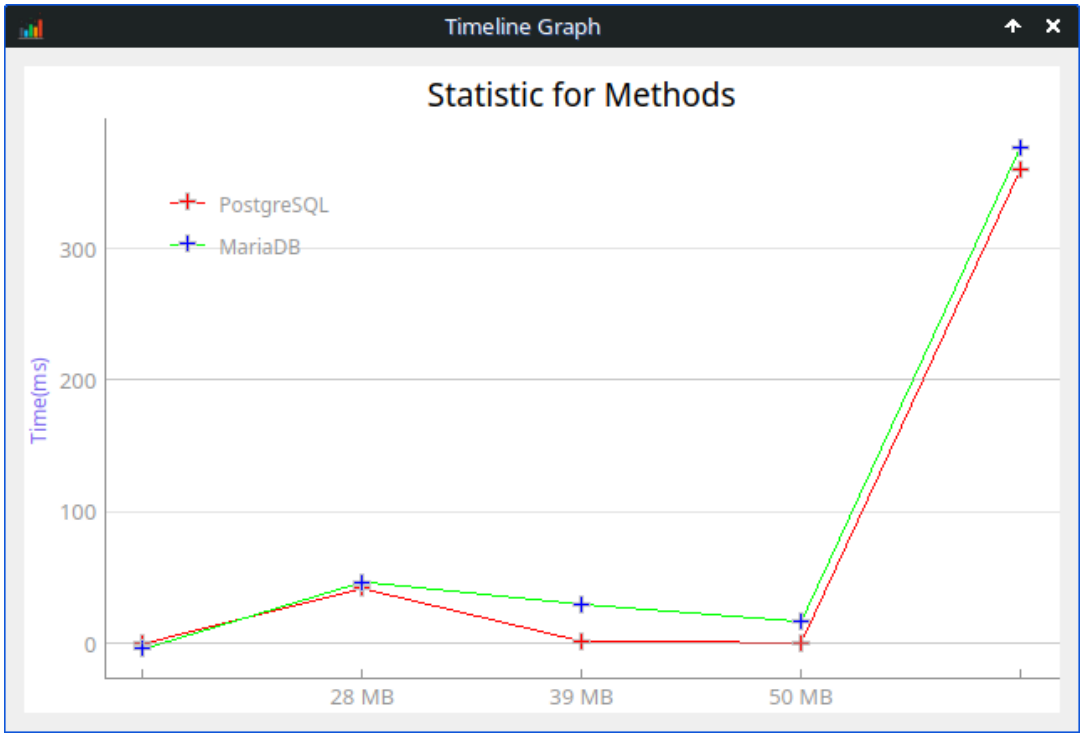

Рисунок 4.2 – Результат другого експерименту з використанням модифікованих методів

Результат модифікованих та покращених методів видно на лице. Час обробки найбільшого відношення скоротився практично вдвічі, те саме можна сказати, і про всі інші таблиці. Методи, які було модифіковано мною, і правда дають значущий результат та пришвидщують час отримання результуючої вибірки та обробки тексту у великих базах даних та таблицях.

Середнє арифметичне від розрохунків подано нижче на таблиці 4.2:

Швидкість обробки запитів системою у другому експерименті значно виросла порівняно із першим. Для першої таблиці час обробки зменшився у цілих 17 разів(171.87%), для другої цей показник упав майже у 5

разів(470.7%), для третьої таблиці він зменшився у майже 3 рази(242.8%), для четвертої у понад 3 рази(326.5%), і нарешті для п’ятої вдвічі(203.1). Тобто, найкращий результат показала СКБД PostgreSQL, адже значення, що повертає ця система керування БД вища за MariaDB, як і було зазначено в другому розділі, PostgreSQL надає змогу широко використовувати індексування та вільно розширювати власний функціонал за рахунок зовнішніх модулів, а це, в свою чергу, дозволяє створити ряд модулів, чи методів, що гарантовано покращать обробку тексту та зменшать час отримання результуючої вибірки.

Таблиця 4.2 – Результати другого експерименту

Результати вимірювання	small_1(мала)	small_2(мала)	medium_1(мала)	medium_2(мала)	large_1
Середній час обробки відношення (PostgreSQL)	0.04695	41.84495	1.798	0.7349	360.0453
Середній час обробки відношення (MariaDB)	0.0574	45.986	5.452	2.156	365.991

Це дуже хороші результати, але під час експерименту я помітив низку дуже цікавих моментів:

- використання GIN на словах, що за довжиною менші за 3 символи є неефективним, ресурсозатратним та недоцільним;
- слова на початку аналізуються швидше ніж ті, що далеко в середині, але після оновлення статистики показники покращуються;
- якщо таблиця по розміру практично виходить за межі своєї категорії, то є можливість що обробка записів там буде дещо повільнішою ніж таблиць, що у своїй категорії;

- пошук цілої фрази у полі це напевно найповільніша операція, адже СКБД робить перевірку не на подібність, а проходиться по всій структурі індексу повністю, і звіряється із кожним записом окремо.

Добре продумана структура та архітектура бази даних може з легкістю компенсувати всі ці недоліки. Не слід також забувати, що чим гірше побудовано структуру та архітектуру БД, тим більше операцій I/O СКБД буде виконувати для того, щоб повернути результуючу вибірку. Навіть добре побудована структура та архітектура не гарантує вам високу швидкість роботи, адже це ще і правильність написання запитів до БД тощо.

ВИСНОВКИ

У результаті виконаної роботи було створено програму та методи, що повністю задовольняють поставлені вимоги, а програма має необхідний функціонал. Для вирішення поставлених задач було використано фреймворк PyQt, мови програмування Python та SQL, СКБД SQLite, PostgreSQL, MariaDB з огляду на те, що з допомогою PostgreSQL інженерія баз даних трохи легша та і робота із індексами там простіша ніж, скажімо, в Oracle. Всі ці засоби було обрано з огляду на те, що це достатньо прості та зручні засоби для розробки стільникових-застосунків. Також, зважаючи на те, що PyQt надає зручні та доступні засоби розробки для стільникових-застосунків, а SQLite надає змогу розгортати БД без використання додаткових серверів, що значно полегшило імплементацію статистичного модуля.

Під час виконання даної роботи було зроблено наступні зауваження:

- деякі відношення не надто великого розміру не варті того, щоб для них створювати окрему структуру в БД, адже це зайва витрата ресурсу, і більше того, такий крок здатен навпаки уповільнити обробку запитів;
- створення індексів доцільне лице тоді, коли відомо, що у результуючу вибірку буде входити лише обмежене число записів, інакше це не матиме сенсу;
- «гортати сторінки» БД це дуже ресурсозатратна операція, а тому таку і подібні ситуації слід уникати;
- від правильно підібраного типу індексу залежить швидкість обробки запитів та взагалі нормальна робота БД;
- при умілому використанні індекси типу GIN в PostgreSQL та FULLTEXT INDEX в MariaDB здатні дати доволі таки хороший результат;
- слова на початку речення будуть аналізуватися швидше ніж ціла фраза у запиті.

В ході виконання даної роботи було отримано додаткові навички з

підтримки, розробки та інженерії реляційних баз даних, написання сценаріїв на SQL, розробки користувацького інтерфейсу. Набуто просунуті вміння роботи з СКБД PostgreSQL та MariaDB. Покращено знання та навички з Python, PyQt та Qt. Також покращено навички з проєктування програмного проєкту, стільникових-застосунків, моделювання вимог та складання документації до програмного забезпечення.

Існують перспективи для покращення та вдосконалення застосунку: додавання більш привабливого дизайну, переробка його архітектури на багатопоточну, додавання можливості підключатися до більшої кількості СКБД, шифрування статистики програми та перенесення її у Cloud.

На даному етапі можна вважати достатньо функціональним продуктом, який можна використовувати для швидкого та надійного пошуку потрібного тексту у великій базі даних.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Рудницький С.А. Методи пришвидшення процесінгу тексту в базах даних Рудницький С.А. Десята міжнародна науково-технічна конференція ПРОБЛЕМИ ІНФОРМАТИЗАЦІЇ. 2022. С. 1.

2. Dominic Tsang, Sanjay Chawla An index for regular expression queries: Design and implementation. 2011. С 1-2. Посилання: https://bit.ly/3U7aDk7

3. Michael Coles, Hilary Cotter. Pro Full-Text Search in SQL Server 2008. [Текст] / New York: Apress, 2008. 287 с.

4. B.Sri Sai Krishna Chaitanya, D.Ajay Kumar Reddy, B.Pavan Sai Eshwar Chandra, A.Bala Krishna, Remya R.K.Menon Full-text Search Using Database Index. *5th International Conference on Computing Communication Control and Automation (ICCUBEA)*. 2019. С. 1-2. IEEE: https://bit.ly/3DFQuev

5. Jason Strate, Grant Flitchey Expert Performance Indexing in SQL Server. [Текст] / New York: Apress, 2015. 415 с.

6. Zain Ul Hassan, Muhammad Naeem, Muhammad Khalid Proposed Generic Full Text Searching Algorithm: A Database Approach. *Article in International Journal of Computer & Organization Trends*. 2015. С 1. ISSN 2249-2593, Посилання: https://bit.ly/3DnOQ0Q

7. Learn Microsoft | SQL Server and Azure SQL index architecture and design guide – Режим доступа: www/ URL: https://learn.microsoft.com/en-us/sql/relational-databases/sql-server-index-design-guide?view=sql-server-ver15 – Загл. з екрану.

8. PostgreSQL Wiki | ColumnOrientedSTorage – Режим доступа: www/ URL: https://wiki.postgresql.org/wiki/ColumnOrientedSTorage – Загл. з екрану.

9. MySQL | Full-Text Search Functions – Режим доступа: www/ URL: https://dev.mysql.com/doc/refman/8.0/en/fulltext-search.html – Загл. з екрану.

10. Pganalyze | Understanding Postgres GIN Indexes: The Good and the Bad – Режим доступа: www/ URL: https://pganalyze.com/blog/gin-index – Загл. з екрану.

11. PostgreSQL | Preferred Index Types for Text Search – Режим доступа: www/ URL: https://www.postgresql.org/docs/14/textsearch-indexes.html – Загл. з екрану.

12. Learn Python ı What is Python[Електронний ресурс] – Режим доступа: www/ URL: https://www.pythonforbeginners.com/learn-python/what-is-python – Загл. з екрану.

13. PyQt | Documentation [Електронний ресурс] – Режим доступа: www/ URL: https://wiki.python.org/moin/PyQt – Загл. з екрану.

14. PyQt | PyQt Memory Management [Електронний ресурс] – Режим доступа: www/ URL: http://enki-editor.org/2014/08/23/Pyqt_mem_mgmt.html – Загл. з екрану.

15. SQL | SQL Syntax [Електронний ресурс] – Режим доступа: www/ URL: https://www.sqltutorial.org/sql-syntax/ – Загл. з екрану.

16. JSON | Introducing JSON [Електронний ресурс] – Режим доступа: www/ URL: https://www.json.org/json-en.html – Загл. з екрану.

17. SQLite | Welcome to SQLite Documentation [Електронний ресурс] – Режим доступа: www/ URL: https://sqlite.org/index.html – Загл. з екрану.

18. Learn Microsoft | Improve the Performance of Full-Text Indexes [Електронний ресурс] – Режим доступа: www/ URL: https://learn.microsoft.com/en-us/sql/relational-databases/search/improve-the-performance-of-full-text-indexes?view=sql-server-ver16 – Загл. з екрану.

19. RAID | Understanding RAID Performance at Various Levels [Електронний ресурс] – Режим доступа: www/ URL: https://www.arcserve.com/blog/understanding-raid-performance-various-levels – Загл. з екрану.

20. Full-Text Search | Improve Your Query Performance [Електронний ресурс] – Режим доступа: www/ URL: https://www.couchbase.com/blog/full-

text-search-tips-for-query-performance/ – Загл. з екрану.

21. SQL | Merging and Resolving Conflicts Programmatically with SQL [Електронний ресурс] – Режим доступа: www/ URL: https://www.dolthub.com/blog/2021-03-15-programmatic-merge-and-resolve/ – Загл. з екрану.

22. MariaDB | SHOW USER_STATISTICS [Електронний ресурс] – Режим доступа: www/ URL: https://mariadb.com/kb/en/show-user-statistics/ – Загл. з екрану.

23. PostgreSQL | Slow Query Questions [Електронний ресурс] – Режим доступа: www/ URL: https://wiki.postgresql.org/wiki/Slow_Query_Questions – Загл. з екрану.

24. Learn Microsoft | sp_configure (Transact-SQL) [Електронний ресурс] – Режим доступа: www/ URL: https://learn.microsoft.com/en-us/sql/relational-databases/system-stored-procedures/sp-configure-transact-sql?view=sql-server-ver16 – Загл. з екрану.

25. PostgreSQL | Resource Consumption [Електронний ресурс] – Режим доступа: www/ URL: https://www.postgresql.org/docs/9.0/runtime-config-resource.html#GUC-SHARED-BUFFERS – Загл. з екрану.

26. PostgreSQL | Optimizing PostgreSQL shared_buffers – PostgreSQL High Performance Guide Consumption [Електронний ресурс] – Режим доступа: www/ URL: https://distributedsystemsauthority.com/optimizing-postgresql-shared-buffers/ – Загл. з екрану.

27. PostgreSQL | High CPU Usage [Електронний ресурс] – Режим доступа: www/ URL: https://www.eversql.com/how-to-fix-postgresql-high-cpu-usage/ – Загл. з екрану.

28. PostgreSQL | Enable slow query log in PostgreSQL [Електронний ресурс] – Режим доступа: www/ URL: https://www.eversql.com/enable-slow-query-log-postgresql/ – Загл. з екрану.

29. PostgreSQL | Table Functions [Електронний ресурс] – Режим доступа: www/ URL: https://www.postgresql.org/docs/7.3/xfunc-

tablefunctions.html – Загл. з екрану.

30. PostgreSQL | pg_trgm [Електронний ресурс] – Режим доступа: www/ URL: https://www.postgresql.org/docs/current/pgtrgm.html – Загл. з екрану.

31. Python | Singleton [Електронний ресурс] – Режим доступа: www/ URL: https://realpython.com/primer-on-python-decorators/#creating-singletons – Загл. з екрану.

32. Python | The Factory Method Pattern and Its Implementation in Python [Електронний ресурс] – Режим доступа: www/ URL: https://realpython.com/factory-method-python/ – Загл. з екрану.

33. PostgreSQL | Using EXPLAIN [Електронний ресурс] – Режим доступа: www/ URL: https://wiki.postgresql.org/wiki/Using_EXPLAIN – Загл. з екрану.

34. PyQt | Qt for Python Signals and Slots [Електронний ресурс] – Режим доступа: www/ URL: https://wiki.qt.io/Qt_for_Python_Signals_and_Slots – Загл. з екрану.

35. MariaDB | Full-Text Index Overview [Електронний ресурс] – Режим доступа: www/ URL: https://mariadb.com/kb/en/full-text-index-overview/ – Загл. з екрану.