МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

МАГІСТЕРСЬКА РОБОТА

на тему:

«Миттєвий нечіткий пошук для мови запитів GraphQL з використанням ймовірносно-кореляційного рейтингу»

Завідувач
випускаючої кафедри **Довбиш А.С.**

Керівник роботи **Проценко О.Б.**

Студента групи ІНм – 81н **Мажуга Б.В**

СУМИ 2020

Сумський державний університет

(назва вузу)

Факультет *ЕлІТ* Кафедра *Комп'ютерних наук*

Спеціальність *«Інформатика»*

Затверджую:

зав.кафедрою ______________

____

"____" ______________ 20___р.

ЗАВДАННЯ

НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

Мажузі Богдану Вікторовичу

(прізвище, ім'я, по батькові)

1. Тема проекту (роботи) *Миттєвий нечіткий пошук для мови запитів GraphQL з використанням ймовірносно-кореляційного рейтингу*

затверджую наказом по інституту від "____" ______________ 20 ___ р. №
____

2. Термін здачі студентом закінченого проекту (роботи)______________

3. Вхідні данні до проекту (роботи) ______________

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

1) Аналіз проблеми та огляд літератури за темою; 2) Постановка завдання й формування завдань дослідження; 3) Аналіз методів нечіткого пошуку; 4) Опис алгоритму миттєвого нечіткого пошуку; 5) Розробка програми для мови запитів GraphQL; 5) Аналіз отриманих результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) ______________

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання ____

Керівник ______________________
(підпис)

Завдання прийняв до виконання ______________________
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка
1.	Аналіз алгоритмів нечіткого пошуку.		
2.	Постановка задачі та формування завдань дослідження.		
3.	Опис алгоритму миттєвого нечіткого пошуку.		
4.	Розробка програми для мови запитів GraphQL		
5.	Оформлення пояснювальної записки до кваліфікаційної магістерської роботи		

Студент – дипломник ______________________
(підпис)

Керівник проекту ______________________
(підпис)

РЕФЕРАТ

Записка: 43 стор., 14 рис., 2 таблиці, 2 додатки, 15 літературних джерел.

Об’єкт дослідження — алгоритми миттєвого нечіткого пошуку

Мета роботи — розробити і програмно реалізувати гнучку систему миттєвого нечіткого пошуку для мови запитів GraphQL з використанням ймовірносно-кореляційного рейтингу.

Результати — проведено аналіз та виконано програмну реалізацію алгоритмів миттєвого нечіткого пошуку у вигляді програми з використанням технологій NestJS, GraphQL, Typescript, з підключеною до ней базою даних PostgreSQL.

НЕЧІТКИЙ ПОШУК, МИТТЄВИЙ НЕЧІТКИЙ ПОШУК, MULTI-KEYWORD SEARCH, FUZZY SEARCH, PROBABILISTIC CORRELATION, APOLLO, GRAPHQL, NESTJS, POSTGRESQL

ЗМІСТ

ВСТУП 4
1 ІНФОРМАЦІЙНИЙ ОГЛЯД 5
1.1 Алгоритми пошуку текстів та нечіткий пошук 5
1.2 Загальна інформація про GraphQL 6
1.4 Загальна інформація про Nest.JS 8
1.5 Миттєвий нечіткий пошук та сучасні алгоритми нечіткого пошуку 9
1.7 Постановка задачі 13
2 ВИБІР МЕТОДУ РІШЕННЯ 15
2.1 Обґрунтування методу та вхідні дані 15
2.2 Миттєвий нечіткий пошук ключових слів 16
2.3 Миттєвий нечіткий пошук з багатьма ключовими словами 17
2.4 Вибір технологій та мов програмування 18
2.7 Вибір СКБД 20
3 ПРОГРАМНА РЕАЛІЗАЦІЯ 22
3.1 Підготовка бази даних 22
3.2 Побудова сервера GraphQL на Nest.js 24
3.3 Реалізація пошуку за рейтингом релевантності 27
3.4 Висновки рейтингу ймовірнісно-кореляційної релевантності та тестування результатів 30
ВИСНОВКИ 37
СПИСОК ЛІТЕРАТУРИ 38
ДОДАТКИ 40
Додаток А 40
Додаток В 41

ВСТУП

Дослідження проблем пошукових систем дуже поширене в інформатиці. Ми шукаємо файл, ми шукаємо каталог, шукаємо символ у файлі, шукаємо тексти пісень, ми шукаємо елемент у списку посилань тощо. І це лише невелика кількість нескінченного пошуку, яку люди звикли робити за допомогою комп'ютера. Існують різні методи пошуку, такі як пошук по глибині (DFS), пошук по ширині (BFS), сходження на гору, пошук променя тощо. Кожен метод має свою перевагу для вирішення різних типів пошуку в різних ситуаціях. Усі існуючі методи пошуку не завжди базуються на точних даних. Отже, боротьба з невизначеністю в пошуку, нечітка логіка та інтуїтивна нечітка логіка будуть відповідним інструментом.

Ні для кого не секрет, що в наш час швидкість та точність виконання пошуку є критично важливим, тому так званий «миттєвий» пошук рекомендує виконати запит "на льоту" і миттєво відображає результати при кожному натисканні клавіші. Бажано, щоб результати запитів були надійними щодо типографічних помилок, які з'являються не тільки в запиті, але і в документах. Крім того, миттєвий пошук вимагає миттєвого часу відповіді та ранжирування результатів, щоб зосередитись на найбільш важливих відповідях.

Для створення запитів на сьогодні не має іншого способу, як використання мови запитів GraphQL, адже дозволяє отримувати той об'єм інформації, який запитується і також значно швидше ніж REST.

У цій роботі створюються та аналізуються прості та ефективні методи миттєвого нечіткого пошуку одного ключового слова та багаторечових ключових слів, стійких до типографічних помилок і які використовують не більше, ніж перевернуті та спрямовані індекси.

1 ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1 Алгоритми пошуку текстів та нечіткий пошук

Дослідження алгоритмів пошуку тексту досягли значних успіхів за останні десятиліття. Хоча в літературі є декілька персоналізованих пошукових систем, розуміння намірів користувачів залишається складною проблемою. Наступні недоліки в звичайній текстовій пошуковій системі ускладнюють задоволення реальних потреб користувачів.[1]

По-перше, обчислювальна вартість широкомасштабного пошуку тексту все ще висока. При швидкому збільшенні Інтернет-бази даних обчислювальне навантаження на вбудовані системи занадто велике. Якщо стабільне бездротове з'єднання недоступне, витрати на обчислення алгоритмів пошуку залежать від локального мобільного пристрою. Для підвищення якості та швидкості роботи пошукової системи ефективність звичайних алгоритмів пошуку потребує подальшого підвищення.[1]

По-друге, інтереси користувачів не повністю враховуються в процесі пошуку. Алгоритми відповідності тексту на основі ключових слів можуть не повністю зрозуміти намір користувача. У процесі пошуку слід враховувати індивідуальні персонажі, особистість, контекст пошуку та інтереси. Пошук - це пасивний спосіб рекомендації, результати відповідності тексту повинні відповідати реальним потребам кожного окремого користувача.[1]

По-третє, смислове значення великих текстових даних недостатньо добре представлено в алгоритмах пошуку. Методи відповідності тексту на основі ключових слів часто не мають семантичного розуміння. Користувачам важко представити кожен можливий вираз із тим самим значенням, який вони намагаються шукати в текстовій базі даних.[1]

Сьогодні алгоритми нечіткого пошуку існують як основа всіх систем перевірки орфографії та повноцінних пошукових систем, наприклад як Google або Yahoo[13].

Подібні алгоритми використовуються для функцій на зразок «Можливо ви мали на увазі ...» в пошукових системах.

Нечіткий пошук є важливою функцією пошукових систем. Також його реалізація є набагато складнішою, ніж, наприклад, реалізація пошуку простого за точним збігом.

Завдання нечіткого пошуку можна сформулювати так: «По заданому слову знайти в тексті або словнику розміру n всі слова, що збігаються з цим словом (або починаються з цього слова) з урахуванням k можливих відмінностей».

1.2 Загальна інформація про GraphQL

GraphQL - це мова запиту даних та специфікація, розроблена внутрішньо Facebook у 2012 році до публічного відкриття у 2015 році. Вона пропонує альтернативу архітектурам на основі REST з метою підвищення продуктивності розробників та мінімізації кількості переданих даних. GraphQL використовується у виробництві сотнями організацій усіх розмірів, включаючи Facebook, Credit Karma, GitHub, Intuit, PayPal, New York Times та багато інших.[8]

GraphQL використовується різними компаніями для живлення своїх додатків, веб-сайтів та API. Одним із найпомітніших ранніх прийнятих GraphQL був GitHub. Його REST API пройшов три ітерації, а версія 4 його публічного API використовує GraphQL.

Як згадується на веб-сайті, GitHub виявив, що "можливість точно визначати потрібні вам дані – і лише ті дані, які ви хочете - є вагомою перевагою перед кінцевими точками REST API v3".

Існують щонайменше три конференції, присвячені саме GraphQL: GraphQL Summit у Сан-Франциско, GraphQL Фінляндія в Хельсінкі та GraphQL Europe у Берліні. Спільнота продовжує зростати за допомогою місцевих зустрічей та різноманітних програмних конференцій.

GraphQL зараз, без перебільшення, це - останній писк IT-моди. Це мова запитів, яка використовується клієнтськими додатками для роботи з даними. З GraphQL пов'язане таке поняття, як «схема» – це те, що дозволяє організовувати створення, читання, оновлення та видалення даних в вашому додатку (тобто – перед нами чотири базові функції, які використовуються при роботі зі сховищами даних, які зазвичай позначають акронімом CRUD - create, read, update, delete).

GraphQL – це також час виконання запитів зі своїми даними. Послуга GraphQL є транспортно-агностичною, але зазвичай обслуговується через HTTP. Запит GraphQL запитує лише ті дані, які йому потрібні. Рисунок 1.1 - приклад запиту GraphQL.[2]

```
1 query {
2   person(personID:5) {
3     name
4     birthYear
5     created
6   }
7 }
```

```
{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "19BBY",
      "created": "2014-12-10T15:20:09.791000Z"
    }
  }
}
```

Рисунок 1.1 – Приклад запиту GraphQL

Принципи дизайну GraphQL:

1) Продукт орієнтований – GraphQL керується потребами клієнта в даних та мовою та часом виконання, які підтримують клієнта.
2) Сильне введення тексту – сервер GraphQL підтримується системою типу GraphQL. У схемі кожна точка даних має певний тип, щодо якого вона буде перевірена.
3) Ієрархічна – запит GraphQL є ієрархічним. Поля вкладені в інші поля, а запит має форму даних, які він повертає.

4) Інтроспектива – мова GraphQL здатна запитувати систему типів сервера GraphQL.
5) Запити, визначені клієнтом – графічний сервер GraphQL надає можливості, які клієнтам дозволено споживати.

1.4 Загальна інформація про Nest.JS

Сьогодні у сучасних розробників є багато альтернатив, коли мова заходить про створення веб-сервісів і інших серверних додатків. Node JS є популярник, проте багато розробників віддали перевагу б більш надійній мові, аніж JavaScript. Фреймворк NestJS виводить NodeJS на новий рівень, надаючи сучасні інструменти для створення високопродуктивних додатків з використанням компонентів, провайдерів, модулів і інших корисних високорівневих абстракцій.

Nest.js - це серверний фреймворк Node.js для створення ефективних, надійних та масштабованих програм. Він надає додаткам модульну структуру для організації коду в окремі модулі. Він був побудований для усунення неорганізованих баз кодів.[3]

Автор фреймворку був натхненний ідеями Angular, і NestJS вийшов ну дуже схожим на Angular, особливо в ранніх версіях. Тому Nest.js складається з:

1) Controllers

Шар контролерів відповідає за обробку, що знаходиться у запрошених і возить від клієнта клієнта.

2) Providers

Майже всі є Providers - Service, Repository, Factory, Helper і т.д. Вони можуть бути впроваджені в контролери та інші провайдери.

3) Modules

Модуль - це клас з декоратором *@Module ()*. Декоратор *@Module ()* надає метадані, які Nest використовує для організації структури програми. Кожна

програма Nest має як мінімум один модуль, кореневий модуль. Кореневий модуль - це місце, де Nest починає впорядковувати дерево додатків. Фактично, кореневий модуль може бути єдиним модулем у вашому додатку, особливо коли додаток маленький. У більшості випадків у вас буде кілька модулів, кожен з яких має тісно пов'язаний опції. У Nest модулі за замовчуванням є Синглетон, тому ви можете без праці ділити один і той же екземпляр компонента між двома і більше модулями.

4) Middlewares

Middlewares - це функція, яка викликається перед обробником роута. Вони мають доступ до request і response.[11]

1.5 Миттєвий нечіткий пошук та сучасні алгоритми нечіткого пошуку

Під час миттєвого пошуку для кожного ключового слова або ключового слова, яке зараз вводиться, пошукова система повинна повертати результати не лише для подібних слів, але і для слів, префікс яких подібний до ключового слова.

Одним з найбільш ранніх прикладів миттєвого пошуку є оболонка Unix, яка представляє список усіх імен файлів, починаючи з літери, введеної в командному рядку. Хоча мета миттєвого пошуку – пошук інформації, вона також використовується в текстових редакторах для прогнозування введення користувача. У контексті систем пошуку інформації існують алгоритми, що пропонують метод індексації ключових слів, а потім запитів для миттєвого пошуку. Однак ці методи вимагають великих витрат на обробку і багато місця[15].

Існують також реалізації нечіткого миттєвого пошуку, які не толерантні до друкарських помилок і порядку слів. Вже давно такі алгоритми інтегровані у ряд пошукових систем, наприклад, Google Instant для пошуку в Інтернеті, Facebook, який шукає відповідних людей, Інтернет-кінотеатри, що

рекомендують фільми, YouTube, що пропонує відео, інтерактивні пропозиції запитів у система електронної пошти, тощо. Однак ці пошукові системи використовують масивні журнали попередніх запитів і не можуть генерувати потрібні відповіді, якщо поставлений запит відсутній у журналі.

Додатковим викликом у пошукових системах є робота з великою кількостю інформації, що знаходяться в базі даних. Одним із рішень у роботі з великою кількістю даних є ранжування результатів запитів; оскільки ранжування спрямовує увагу лише на найбільш відповідні відповіді. Алгоритми ранжування широко вивчені в базах даних та пошуку інформації.

У цій роботі вивчаються прості та ефективні методи нечіткого миттєвого пошуку, які відповідають на одне ключове слово та запити на кілька ключових слів. Крім того, методи обчислюють відповіді поступово, використовуючи кешовані результати, та класифікують відповіді, виходячи з їх відповідності запиту, використовуючи імовірнісний кореляційний коефіцієнт.

Миттєвий пошук є ефективним, коли він дає швидше отримання набору відповідей з мінімальним часом обчислення. Нечіткий пошук, необхідний для запитів, які є помилками з декількох причин. Нечіткий пошук, що використовується для поліпшення досвіду пошуку користувачів шляхом пошуку відповідних відповідей із ключовими словами, схожими на ключові слова.

Ми використовуємо значення порогової фрази, яке використовується для обмеження набору відповідей, створеного миттєвим нечітким пошуком. Тому головне завдання полягає в тому, щоб підвищити швидкість виконання роботи, а також мінімізувати набір відповідей для пошуку бажаних документів для запиту користувача. У той же час нам також потрібні кращі функції ранжування, які враховують близькість ключових слів для обчислення релевантних балів.

Розвиток сучасної науки, техніки і суспільства втілило в життя такої великої потік науково-технічної та соціальної інформації, що традиційних

методів її накопичення, систематизації і переробки стає явно недостатньо. Ефективним вирішенням цієї важливої проблеми є автоматизація процесів обробки великих масивів текстової інформації, що передбачає більш широке використання автоматичних процедур обробки текстів. Це дозволить значно підвищити якість науково-дослідних і експериментальних робіт і скоротити терміни їх виконання.[12]

Сьогодні існують різні алгоритми нечіткого пошуку:

1) Відстань Левенштейна.

Відстань Левенштейна (також функція Левенштейна, алгоритм Левенштейна або відстань редагування) у теорії інформації і комп'ютерній лінгвістиці міра відмінності двох послідовностей символів (рядків). Обчислюється як мінімальна кількість операцій вставки, видалення і заміни, необхідних для перетворення одної послідовності в іншу. [4]

2) Відстань Дамерау-Левенштейна

До віднстані Левенштейна додається це одне правило — транспозиція (перестановка) двох сусідніх букв також враховується як одна операція, поряд зі вставками, вилученнями і замінами.

Відстань Дамерау-Левенштейна між двома рядками a та b визначається функцією $d\ (|a|,|b|)\ d_{a,b}(|a|,|b|)$ як зображено на рисунку 1.2.

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if} \min(i,j) = 0, \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ d_{a,b}(i-2,j-2)+1 \end{cases} & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j \\ \min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} & \text{otherwise,} \end{cases}$$

Рисунок 1.2 – Відстань Дамерау-Левенштейна

Щоб обчислювати таку відстань, досить трохи модифікувати алгоритм

знаходження звичайного відстані Левенштейна наступним чином: зберігати не дві, а три останні рядки матриці, а також додати відповідну додаткову умову — в разі виявлення транспозиції при розрахунку відстані також враховувати і її вартість.[5]

3) Алгоритм Bitap з модифікаціями від Wu і Manber

Також буває необхідно обчислювати відстань між префіксом-зразком і рядком – тобто знайти відстань між заданим префіксом і найближчим префіксом рядки. У цьому випадку необхідно взяти найменшу з відстаней від префікса-зразка до всіх префіксів рядки.

Найчастіше при нечіткому пошуку важливо не стільки саме значення відстані, скільки факт того, перевищує воно чи ні певну величину.

4) Алгоритм розширення вибірки

Цей алгоритм часто застосовується в системах перевірки орфографії, там, де розмір словника невеликий, або ж де швидкість роботи не є основним критерієм. Він заснований на зведенні задачі про нечіткому пошуку до задачі про точний пошуку.

5) Метод N-грам

Цей метод є достатньо старим, і є найбільш широко використовуваним, так як його реалізація дуже проста, і при цьому він забезпечує досить хорошу продуктивність. Алгоритм ґрунтується на принципі: «Якщо слово А збігається зі словом Б з урахуванням кількох помилок, то з великою часткою ймовірності у них буде хоча б один загальний підрядок довжини N». Ці підрядки довжини N і називаються N-грамами.

Під час індексації слово розбивається на такі N-грами, а потім це слово потрапляє в списки для кожної з цих N-грам. Під час пошуку запит також розбивається на N-грами, і для кожної з них виробляється послідовний перебір списку слів, що містять такий підрядок.

6) Гешування за сигнатурі

Алгоритм гешування по сигналу (ХС) спочатку був описаний Бойцовим Л.М і до сьогодні залишається найчастіше розповсюдженим методом пошуку, вимагаючи неточного завдання термінових запитів. Цей простий і дуже ефективний алгоритм для роботи в більших колекційних строках.[10]

Алгоритм базується на досить очевидному поданні «структури» слова у вигляді бітових розрядів, яку використовують як гешу (сигнатури) в геш-таблиці.

При індексації такі геші обчислюються для кожного зі слів, і в таблицю заноситься відповідність списку словникових слів цього хешу. Потім, під час пошуку, для запиту обчислюється геш і перебираються всі сусідні геші, що відрізняються від вихідного не більше ніж в *k* бітах. Для кожного з таких гешів проводиться перебір списку відповідних йому слів

Процес обчислення гешу – кожному біту гешу зіставляється група символів з алфавіту. Біт 1 на позиції і в геші означає, що в початковому слові присутній символ з *i*-ої групи алфавіту. Порядок букв в слові абсолютно ніякого значення не має.

Цьому алгоритму присутній один досить суттєвий недолік: він повільно працює, якщо індекс фрагментований: тобто в тому випадку, якщо списки слік з однаковими з однаковими сигнатурами розкидані по несуміжним секторам на диску. На сьогодні це не є великою проблемою, після чого, з однієї сторони, розміри пам’яті комп'ютерів часто включають завантаження словника цілком, а з іншої сторони, дефрагментація словника, як правило, виконується за декілька хвилин.[10]

1.7 Постановка задачі

Базуючись на відомих алгоритмах нечіткого пошуку, розробити та програмно реалізувати алгоритм миттєвого нечіткого пошуку для одного слова та декількох слів для мови запитів GraphQL з використанням ймовірносно-кореляційного рейтингу.

При розробленні алгоритму перевірити достовірність отриманих результатів. Для цього відобразити графіки роботи алгоритму та обрати оптимальний алгоритм. Протестувати роботу алгоритму.

Програмну реалізацію виконати на мові програмування Node.JS, а саме використати фреймворк NestJS. При створенні запитів використати типізування Typescript. Базу даних слів створити використовуючи об'єктно-реляційну базу даних PostgreSQL.

2 ВИБІР МЕТОДУ РІШЕННЯ

2.1 Обґрунтування методу та вхідні дані

Нехай $R = \{r_1, r_2, \ldots\}$ - це набір записів, а $D = \{w_1, w_2, \ldots\}$ - словник, що містить набір ключових слів Р. У таблиці 1 наведено приклад набору записів, у таблиці 2 наведено перевернутий індекс набору записів.

Таблиця 1 – Приклад набору даних

Ідентифікатор запису	Назва
R_1	Пошук інформації: структури даних та алгоритми
R_2	Зберігання та пошук інформації
R_3	Інтелектуальний пошук інформації
R_4	Підходи до інтелектуального пошуку інформації
R_5	Розподілена система зберігання та пошуку інформації
R_6	Система зберігання інформації

Таблиця 2 – Інвертований індекс

1	пошук	$R_1 R_2 R_3 R_4 R_5$
2	інформації	$R_1 R_2 R_3 R_4 R_5 R_6$
3	структури	R_1
4	даних	R_1
5	та	$R_1 R_2 R_5$
6	алгоритми	R_1
7	зберігання	$R_2 R_5 R_6$
8	інтелектуальний	R_3
9	підходи	R_4
10	до	R_4

Продовження таблиці 2 – Інвертований індекс

11	інтелектуального	R_4
12	розподілена	R_5
13	система	$R_5 R_6$

У роботі використовується відстань редагування для вимірювання різниці між двома ключовими словами, яка визначається як мінімальна кількість операцій редагування (вставка, видалення, заміна окремих символів, необхідних для перетворення одного ключового слова в інше).

Позначаємо відстань редагування між двома ключовими словами S_1 та S_2. Два ключових слова схожі, якщо $(S_1, S_2) \leq \tau$ де τ – поріг відстані редагування.

2.2 Миттєвий нечіткий пошук ключових слів

Миттєвий нечіткий пошук ключових слів складається з пошуку ключових слів, які мають префікс, близький до рядка запиту. Більш формально, задавши рядок запиту $q = c_1, c_2 \ldots$, миттєвий нечіткий пошук ключових слів генерує список ранжуваних пар (w_i, r_j) таким чином, що префікс v в w_i схожий на q, а – ключове слово $r_j \in R$.

У даній роботі представлений простий та ефективний алгоритм миттєвого нечіткого пошуку ключових слів, де інформаційна система приймає послідовність запитів від користувача, який вводить текст за символом, та обчислює відповіді на запит із відповідей, сформованих у попередньому запиті в послідовності. Отримані відповіді класифікуються залежно від їх відповідності.

Нехай $q = c_1 c_2 \ldots$ – запит, який вводиться символ за символом, τ - поріг відстані редагування. Для підрядків $q_1 = c_1$, $q_2 = c_1c_2, \ldots, q_\tau = c_1c_2 \ldots c_\tau$ відповідь встановлюється з усіх пар (w_i, r_j) таких, що $w_i \in D$ – це ключове слово запису $r_i \in R$.

Для підзапиту $q_{\tau+1} = c_1\ c_2\ ...\ c_{\tau+1}$ весь перевернутий індекс сканується з метою генерування відповіді. Нехай $\phi_{\tau+1}$ - відповідь на запит $q_{\tau+1}$, а $v_{i,\tau+1}$ – префікс w_i, довжина якого дорівнює $\tau + 1$, тоді для кожного $w_i \in D$ префікс $V_{i,\tau+1}$ порівнюється з запитом q_{r+1}, і пара (w_i, r_j) включена у набір відповідей запиту q_{t+1} тоді і лише тоді, коли $V_{i,\tau+1}$ схожий на q_{r+1}. Іншими словами, відповідь на запит q_{t+1} – це набір пар, визначений так, як показано на рисунку 2.1.

$$\varphi_{t+1} = \{(w_i, r_j)\,|\mathrm{edist}(v_{i,\tau+1}, q_{\tau+1}) \leq \tau\}$$

Рисунок 2.1 – Відповідь на запит $\phi_{\tau+1}$

Для наступних запитів $q_x = c_1c_2...c_x$ таким, що $x > \tau + 1$, відповідь ϕ_x обчислюється з ϕ_{x-1} наступним чином. Замість сканування всього інвертованого індексу розглядаються лише ключові слова пар, що входять до ϕ_{x-1}. Спочатку ϕ_x ініціалізується порожнім, потім для кожного $(w_i, r_j) \in \phi_{x-1}$, відстань редагування префіксу $v_{i,x}\, w_i$, довжина якої дорівнює x, обчислюється проти запиту q_x, і пара (w_i, r_j) включається в ϕ_x тоді і тільки якщо edist $(v_{i,x}, q_x) \leq \tau$. Формально відповідь на запит q_x – це набір пар, визначений як показано на рисунку 2.2.

$$\phi_x = \{(w_i, r_j)\,|\,(w_i, r_j) \in \phi_{x-1} \wedge \text{ edist } (v_{i,x}, q_x) \leq \tau\}$$

Рисунок 2.2 – Відповідь на запит q_x

Алгоритм є стійким до друкарських помилок, простим та ефективним. В алгоритмі використовується інвертований індекс, який не сканується повністю для всіх підзапитів. Крім того, під час обчислення відстані редагування між префіксом ключового слова та заданим запитом враховується лише префікс, довжина якого дорівнює довжині запиту.

2.3 Миттєвий нечіткий пошук з багатьма ключовими словами

Запит на кілька ключових слів q_l складається з послідовності ключових слів $(w_1, w_2, ..., w_l)$. Під час миттєвого пошуку генерується запит для кожного

символу, набраного користувачем, і миттєвий нечіткий пошук у кількох ключових словах виконує пошук записів $r_j \in R$ таким чином, щоб виконувались наступні дві умови:

- запис r_j має ключове слово, схоже на w_i для $1 \leq i \leq l-1$.

- запис r_j має ключове слово з префіксом, схожим на w.

Спочатку, коли користувач вводить запит символ за символом, відповідь на перше ключове слово w_1 обчислюється за допомогою перевернутого індексу. Після завершення першого ключового слова гешуються записи, які відповідають першому ключовому слову. Згодом, коли користувач вводить послідовність символів, що генерують запити з декількома ключовими словами, для кожного введеного символу генерується послідовність підзапитів, а результати попереднього запиту гешуються.

Нехай w_l – це ключове слово, яке вводиться, де $l > 1$, тоді префікси, подібні до ключового слова w_l, отримуются із записів результатів гешування попереднього запиту, а не в повному наборі ключових слів. Нехай R_{i-1} – це набір записів, гешованих для підзапиту q_{i-1}, тоді набір записів, що містять ключові слова, схожі на $w_1, w_2, \ldots, w_l$ та ключове слово з префіксом, подібним w_l до обчислюється для q_i наступним чином: для кожного $r_i \in R_{i-1}$ попередній індекс використовується для визначення, чи існує ключове слово з префіксом, аналогічним w_i в r_i Якщо ключове слово з префіксом, подібним w_l, існує в r_i, то r_i включається в R_i.

Алгоритм використовує інвертований та попередній індекс. Крім того, алгоритм працює з помилками вводу та не залежить від порядку слів. Алгоритм ефективний, оскільки він не сканує всі записи для кожного підзапиту.

2.4 Вибір технологій та мов програмування

1) NestJS

Nest (NestJS) - це основа для створення ефективних масштабованих додатків на сервері Node.js. Він використовує прогресивний JavaScript,

створений і повністю підтримує TypeScript (але все ще дозволяє розробникам писати на чистому JavaScript) і поєднує елементи OOP (об'єктно-орієнтоване програмування), FP (функціональне програмування) та FRP (функціональне реактивне програмування).

Nest використовує надійні рамки HTTP-сервера, такі як Express (за замовчуванням), і, за бажанням, можна налаштувати також використовувати Fastify.

Nest забезпечує рівень абстракції над цими загальними рамками Node.js (Express / Fastify), але також піддає свої API безпосередньо розробнику. Це дозволяє розробникам використовувати безліч сторонніх модулів, доступних для базової платформи.

Nest - дозволяє легко інтегруватися з будь-якою базою даних SQL або NoSQL. Взагалі підключення Nest до бази даних - це просто питання завантаження відповідного драйвера Node.js для бази даних.

2) GraphQL

GraphQL - це потужна мова запитів для API. Це елегантний підхід, який вирішує багато проблем, які зазвичай зустрічаються з API REST. GraphQL у поєднанні з TypeScript допомагає розвивати кращу безпеку типів за допомогою запитів GraphQL, надаючи цільове введення тексту.

GraphQLModule – це обгортка навколо сервера Apollo. У роботі використовуєтсья перевірений пакет GraphQL, щоб забезпечити спосіб використання GraphQL з Nest.

Nest пропонує два способи побудови додатків GraphQL, перший код та методи першої схеми. При першому підході до коду ви використовуєте декоратори та класи TypeScript для створення відповідної схеми GraphQL. Цей підхід корисний, якщо віддавати перевагу працювати виключно з TypeScript і уникати переключення контексту між синтаксисами мови.

У схемі першого підходу джерелом істини є файли GraphQL SDL (Мова визначення схеми). SDL - це мовно-агностичний спосіб ділитися файлами

схем між різними платформами. Nest автоматично генерує визначення TypeScript (використовуючи класи або інтерфейси) на основі схем GraphQL, щоб зменшити необхідність запису зайвого кодового шаблону.

3) Typescript

TypeScript є строго типізований надмножиною JavaScript, що означає, що він додає деякі синтаксичні переваги для мови, але при цьому дозволяє писати звичайний JavaScript. Він заохочує більш декларативний стиль програмування через такі речі, як інтерфейси і статичну типізацію, пропонує модулі і класи, а найголовніше, досить добре інтегрується з популярними бібліотеками та кодом JavaScript.[6]

4) Розпізнавачі (resolvers)

Розпізнавачі надають інструкції щодо перетворення операції GraphQL (запиту, мутації чи підписки) у дані. Вони повертають ту саму форму даних, яку ми визначаємо в нашій схемі - або синхронно, або як обіцянку, яка відповідає результату цієї форми. Як правило, вручну створюється карта роздільної здатності. Пакет @ *nestjs/graphql*, з іншого боку, автоматично генерує карту роздільної здатності, використовуючи метадані, надані декораторами, які використовуются для анотування класів.

5) GraphQL playground (тестування GraphQL)

GraphQL playground – це графічний, інтерактивний, браузерний GraphQL IDE, доступний за замовчуванням за тією ж URL-адресою, що і сам сервер GraphQL. Для доступу до playground потрібен базовий сервер GraphQL[14].

2.7 Вибір СКБД

PostgreSQL - це потужна об'єктно-реляційна база даних із відкритим кодом з більш ніж 30-річним активним розвитком, яка заслужила їй високу репутацію надійності та продуктивності.[7]

PostgreSQL — широко розповсюджена система керування базами даних

з відкритим сирцевим кодом. Прототип був розроблений в Каліфорнійському університеті Берклі в 1987 році під назвою POSTGRES, після чого активно розвивався і доповнювався. В червні 1990 року з'явилась друга версія із переробленою системою правил маніпулювання та роботи з таблицями, у 1991 році — третя версія, із доданою підтримкою одночасної роботи кількох менеджерів збереження, покращеним механізмом запитів і доповненою системою внутрішніх правил. В цей час POSTGRES використовувався для реалізації великих систем, таких як: система аналізу фінансових даних, пакет моніторингу функціональності потоків, база даних відстеження астероїдів, система медичної інформації, кілька географічних систем. POSTGRES також використовувався як навчальний інструмент в кількох університетах. 1992 року POSTGRES став головною СКБД наукового комп'ютерного проекту Sequoia 2000. 1993 року кількість користувачів подвоїлась. Стало зрозуміло, що для підтримки й подальшого розвитку необхідні великі витрати часу на дослідження баз даних, тому офіційно проект Берклі було зупинено на версії 4.2. 1994 року Andrew Yu і Jolly Chen додали інтерпретатор мови SQL, вдосконалили сирцевий код і виклали в Інтернеті свою реалізацію під назвою Postgres95. 1996 року програмний продукт було перейменовано на PostgreSQL із початковою версією 6.0. Подальшою підтримкою й розробкою займається група спеціалістів у галузі баз даних, які добровільно приєднались до цього проекту.[9]

PostgreSQL базується на мові SQL і підтримує численні можливості, такі як: підтримка БД необмеженого доступу, потужні і надійні механізми транзакцій і реплікації, розширювана система вбудованих мов програмування і підтримка завантаження C-сумісних модулів, успадкування та легка розширюваність.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Підготовка бази даних

Для пошуку даних серед великої кількості інформації, необхідно створити таку базу даних, яка містить мільйони рядків речень або слів. Тому створений простий скрипт SQL для PostgreSQL для наповняння таблиці необхідними данними.

Усі команди виконуються безпосередньо в інтерфейсі командного рядка psql:

```
psql –h localhost –U postgres
```

Створення бази даних PostgreSQL для таблиць:

```
CREATE DATABASE test_db;
\c test_db
```

Для цієї роботи створена дуже проста модель даних, яка складається з двох ключів *id* та *title*. Скрипт для створення таблиці виглядає наступним чином:

```
CREATE TABLE titles(
  id    SERIAL PRIMARY KEY,
  title VARCHAR(80) NOT NULL
);
```

Усі таблиці мають стовпець `SERIAL PRIMARY KEY`, тому PostgreSQL піклується про створення ідентифікаторів.

Заповнити таку таблицю просто, адже необхідно створити лише одне поле з типом *VARCHAR*. Спочатку додали функцію для генерування випадкового рядка:

```
Create or replace function random_generating_string(lengh
integer) returns randomTextStringSumdu as
```

```
$$
declare
  chars randomTextStringSumdu[] := '{0,1,2,3,4,5,6,7,8,
,W,X,Y,Z,a,b,c,d,e,f,g,9,A,B,C,D, f,g,h,i,j,k,l
E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z,a,1,2,3,4,5,6,7,8,,b
,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z}';
  result randomTextStringSumdu:= '';
  i integer := 0;
begin
  if length < 0 then
    raise exception 'Given length cannot be less than 0';
  end if;
  for i in 1..length loop
    randomTextStringSumdu:= randomTextStringSumdu ||
chars[1+random()*(array_length(chars, 1)-1)];
  end loop;
  return result;
end;
$$ language plpgsql;
```

Далі запит GENERATE_SERIES (1, 100000000), як простий спосіб генерувати 100000000 рядків. Зараз дуже легко генерувати тисячу чи мільйон рядків, просто змінюючи одне число.

```
INSERT INTO titles (title)
SELECT
  random_generating_string (RANDOM() * 10)
 AS titile
FROM GENERATE_SERIES(1, 10) seq;

SELECT * FROM title;
```

На рисунку 3.1 зображено як виглядає заповнена таблиця в БД.

```
id |       title
----+--------------------
  1 | kdjui ug8 osaj
  2 | rgsdv e
  3 | chu8ij n d4
  4 | sjiv3847gvj s
  5 | jn sdkf kdfj 3f
  6 | yssj jn5 oisa
  7 | ion  au3fp03 45
  8 | ijshuw jrnfo8p2
  9 | as adfg h35v y52
 10 | feg r 43t1g1 g3g5
...
```

Рисунок 3.1 – Згенерована таблиця в БД

3.2 Побудова сервера GraphQL на Nest.js

Наступним кроком є побудова GraphQL сервера, який матиме архітектуру, яка зображена на рисунку 3.2.

Для побудови сервера були встановлені на необхідні пакети:

```
$ npm i --save @nestjs/graphql graphql-tools graphql
```

Після встановлення пакетів імпортували *GraphQLModule* і налаштували його статичним методом *forRoot ()*:

```
import { Module } from '@nestjs/common';
import { GraphQLModule } from '@nestjs/graphql';

@Module({
    imports: [
        GraphQLModule.forRoot({}),
```

```
    ],
})

export class AppModule {}
```

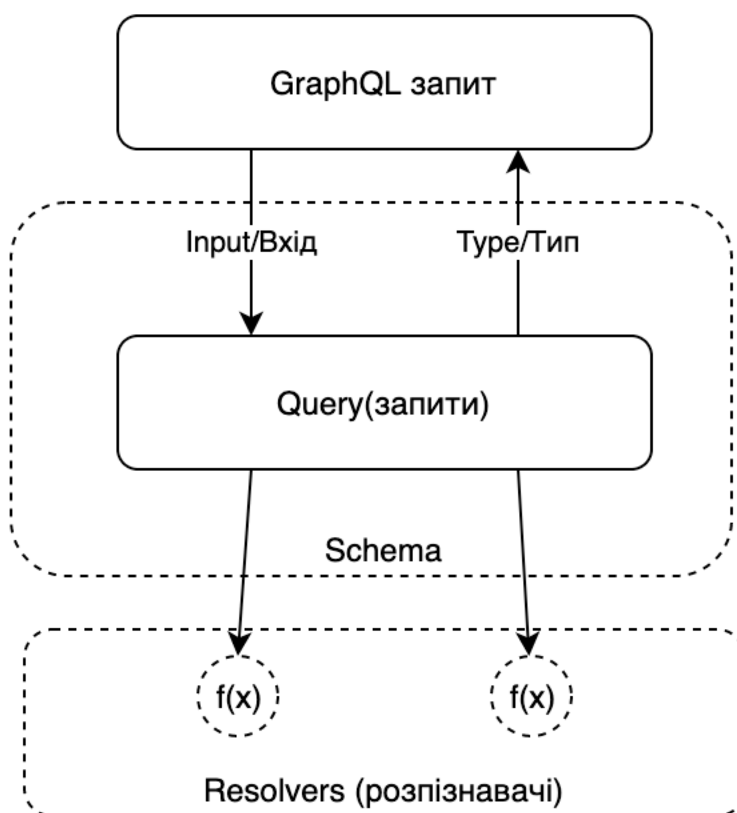

Рисунок 3.2 – Загальна схема сервера GraphQL на Nest.js

Наступним кроком є створення розпізнавача и типу для нього. Більшість визначень у схемі GraphQL – це типи об'єктів. Кожен тип об'єкта, який визначається, повинен представляти доменний об’єкт, з яким в подальшому буде взаємодіяти:

```
type Titles {
  id: Int!
  title: String
```

```
}
```

Визначили об’єкти (тип данних), які можуть існувати в нашому графі, але ще не маємо способу взаємодії з цими об'єктами. Щоб вирішити це, створили клас розпізнавача:

```
@Resolver(of => Title)
export class TitleResolver {
    constructor(
        private titleService: TitleService,
            ) {}

@Query(returns => Title)
    async title(@Args('id', { type: () => Int }) id: number) {
        return this.titleService.findOneById(id);
    }
```

Далі необхідно створити розпізнавач пошуку, він і буде викликати алгоритм пошуку:

```
@Resolver(of => SearhEngine)
export class SearhEngine Resolver {
    constructor(
        private searhEngineService: SearhEngineService,
            ) {}

@Query(returns => Title)
    async title(@Args('title', { type: () => String }) title:
string) {
        return SEARCH_ENGINE.search(title);
```

```
}
```

Використовується також метод *search* об’єкт *SEARCH_ENGINE*, реалізація якого описана в наступному пункті.

3.3 Реалізація пошуку за рейтингом релевантності

Існує багато способів співвідносити один текст з іншим, але потрібно створити такий спосіб, що буде заснованим на статистиці, який не потребує розуміння самої мови, а скоріше розглядає статистику використання слів та збігів і зважує документи на основі поширеності їх унікальних слів. Для цього в проект NestJS необхідно встановити бібліотеку, що реалізовує один з найсучасніших алгоритмів ранжирування Okapi BM25:

```
$ npm install wink-bm25-text-search —save
```

Визначимо простий статичний метод *Tokenize()*, метою якого є розбір рядка в масив лексем:

```
SEARCH_ENGINE.Tokenize = (someTest) => {
   const fixedText = someText
    .toLowerCase()
    .replace(/\s+/g, ' ')
    .replace(/\W/g, ' ')
    .trim()
    .split(' ')
    .map(world => stemmer(world));

    // Фільтр

   const result = [];
for (let i = 0, len = fixedText.length; i &lt; len; i++) {
        if (stopStems.includes(fixedText [i])) {
```

```
            result.push(fixedText [i]);
        }
    }
return result;
};
```

У цьому методі зменшуются великі регістри всіх лексем (щоб зменшити ентропію), запускається алгоритм для зменшення ентропії цілого рядка, а також для поліпшення відповідності (щоб слова «скло» та «скляний» збігалися), і фультруються стоп-слова (дуже поширені слова) для подальшого зменшення ентропії.

Наступним кроком є створення важливого методу *addSearchDoc().* По суті, будується дві подібні структури даних: *this.docs* та *this.terms*:

```
     SEARCH_ENGINE.prototype.addSearchDoc = (searchDoc)  => {
            if (typeof searchDoc.id === 'undefined') {
                throw new Error(1, 'Відсутній ID .');
              };
    if (typeof searchDoc.body === 'undefined') {
        throw new Error(2, 'Пилка зчитування body.');
     };

     // Токенізований список слів
     const tokensArray = BM25.Tokenize(searchDoc.body);
      const privateTerms = {};
      const  docObj =  id: searchDoc.id, tokensArray, body:
doc.body};
     // кількість термів
    docObj.termCount = tokensArray.length;
    // Increment totalDocs
    this.totalDocs++;
    // Підрахунок середньої довжини документа (averageDocLength)
    this.totalDocumentPrivatTermLength += docObj.termCount;
```

```
    this.averageDocLength = this. totalDocumentPrivatTermLength
/ this.totalDocs;

}
```

this.docs – це база даних про окремі документи, але поряд із збереженням повного оригінального тексту документа, також зберігається довжина документа та список усіх лексем у документі разом із їх кількістю та частотою. *this.terms* – представляє всі терміни в базі даних.

Також використовується метод, що подбає про індентифікатори термів:

```
SEARCH_ENGINE.prototype.updateIdf = () => {
    const keys = Object.keys(this.terms);
    for (let index = 0, len = keys.length; i &lt; len; index ++)
{
        const term = keys[index];
        const num = (this.totalDocuments − this.terms[term].n +
0.5);
        const denomValur = (this.terms[term].n + 0.5);
        this.terms[term].id = Math.max(Math.log10(num /
denomValur), 0.01);
    }
};
```

Далі метод *search().* Він перебирає всі документи та присвоює бал відповідності BM25 кожному, сортуючи найвищі бали:

```
SEARCH_ENGINE.prototype.search = (string) => {
  const currentTerms = BM25.Tokenize(string);
  const results = [];

  // Перебір кожного документа по черзі.
  const keys = Object.keys(this.docs);
```

```
  for (
    let internalIndex = 0, nDocs = keys.length;
    internalIndex < nDocs;
    internalIndex++
  ) {
    const currentId = keys[internalIndex];
    // Оцінка релевантності для документа.
    this.docs[currentId].privateScore = 0;

    // Обчислення балу за кожний термін
    for (let index = 0, len = currentTerms.length; index < len;
index++) {
      const currentTerm = currentTerms[index];

      //Ніколи не бачили цього терміна, тому IDF буде 0.
      // Значить, можемо пропустити весь термін, він нічого не
додає до оцінки
      // і його немає в жодному документі.
      if (typeof this.terms[currentTerm] === "undefined") {
        continue;
      }

}
```

Суть коду, описаного вище, полягає в наступному: для кожного документа та кожного терміна запиту обчислюється бал. Оцінка для кожного терміна запиту заздалегідь розраховується на основі простого пошуку, частота терміна залежить від документа, але вона також попередньо обчислюється, а решта роботи - це просто множення та ділення. До кожного документу додається тимчасова змінна під назвою *privateScore*, а потім сортуються результати за балом (у зменшенні) та повертається топ 10.

3.4 Висновки рейтингу ймовірнісно-кореляційної релевантності та тестування результатів

Кореляція між ключовими словами є мірою взаємозалежності і може бути обчислена у програми на основі умовної ймовірності як показано на рисунку 3.3.

$$\operatorname{cor}(w_i, w_j) = \frac{P(w_i|w_j)\,P(w_j|w_i)}{P(w_i)\,P(w_j)} = \frac{P(w_i \wedge w_j)^2}{P(w_i)\,P(w_j)}$$

Рисунок 3.3 – Формула обчислення кореляції між ключовими словами

Можна помітити, що $cor(w_i, w_j) = cor(w_j, w_i)$. Більше того, $cor(w_i, w_j) = 1$ означає, що ключові слова W_i *і* W_j взаємозалежні один від одного і завжди відображаються разом у записах; і навпаки, cor $(w_j, w_i) = 0$ означає, що ключові слова W_i *і* W_j не залежать один від одного і ніколи не з'являються разом.

Більш загальну кореляцію ключового слова w_i з одним або декількома ключовими словами одночасно обчислюється як показано на рисунку 3.4.

$$\operatorname{cor}(w_i, W) = \frac{P(w_i|W)\,P(W|w_i)}{P(w_i)\,P(W)} \quad \frac{P\left(w_i \bigwedge_{w_j \in W} w_j\right)^2}{P(w_i)\,P\left(\bigwedge_{w_j \in W} w_j\right)}$$

Рисунок 3.4 – Формула обчислення загальної кореляції w_i з одним або декількома ключовими словами одночасно

Нехай $R_l \in 2^R$ - сукупність гешованих результатів, а $W_l \in 2^D$ – набір усіх різних ключових слів у гешованих результатах R_l. Якщо ключове слово $w_j \in W_l$, то кореляція Wi, що задається набором ключових слів $W \subseteq W_l$ над гешованими результатами, R_l позначається як $cor_l(w_i, W)$, і обчислюється з гешованих результатів замість всього сховища даних R.

Нехай $r \in R_l$ - запис у гешованих результатах R_l, що складається з ключових слів $r = \{k_1, k_2, \ldots, k_n\}$ де $k_i \in W_l$ при $1 \leq i \leq n$; то релевантність або

значення ключового слова K і в записі r можна визначити як показано на рисунку 3.5.

$$\mathrm{rel}\left(k_i, r\right) = \frac{\sum_{W \in 2^r \wedge k_i \notin W} \mathrm{freq}(W)^* \,\mathrm{cor}_i\left(k_i, W\right)}{2^{|r|-1} - 1}$$

Рисунок 3.5 – Формула визначення релевантності та значення ключового слова K і в записі r

Можна зазначити, що $0 \leq rel\ (k, r) \leq 1$; а $rel\ (k, r) = 1$ означає, що ключове слово k має більшу актуальність у r, тоді як $rel\ (k, r) = 0$ не означає, що ключове слово k має r.

Тестування

Коли програма працює у фоновому режимі, відкриваємо веб-браузер і переходимо до http://localhost:3001/graphql (хост і порт можуть змінюватися, така конфігурація для розробки локально). Відкриється GraphQL playground, як показано на рисунку 3.6.

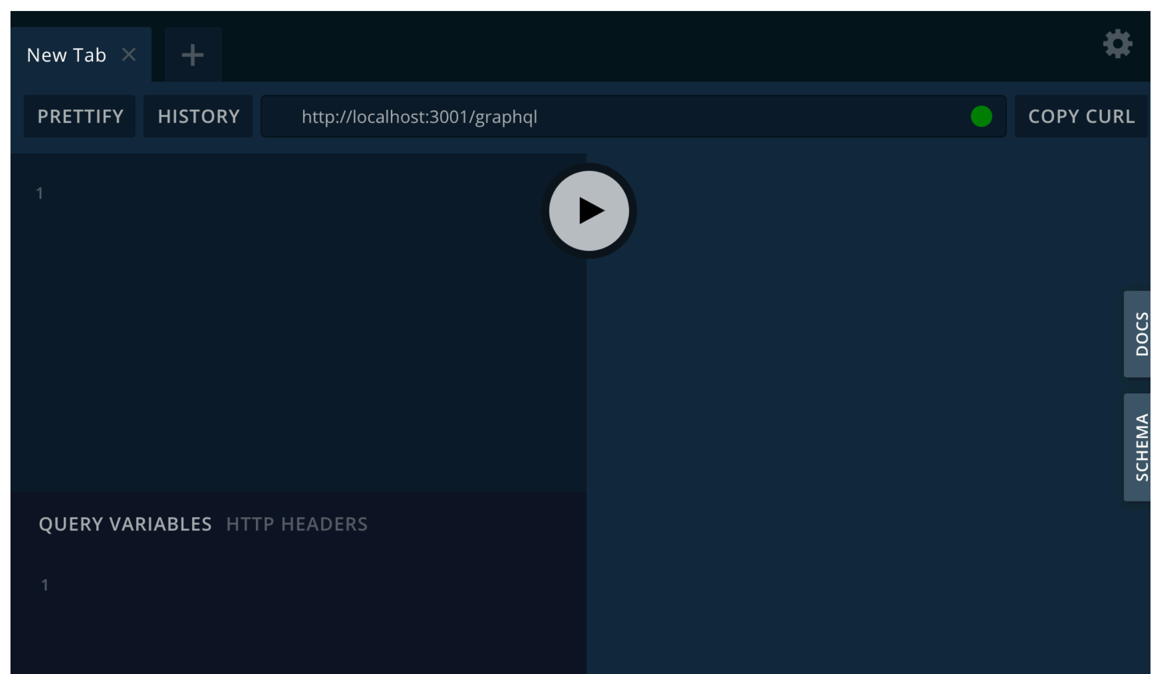

Рисунок 3.6 – GraphQL playground

Виконуємо перший запит. Для цього необхідно заповнити поле запиту, вказати змінні пошуку та натиснути кнопку, що посередині. Маємо результат, який показано на рисунку 3.7.

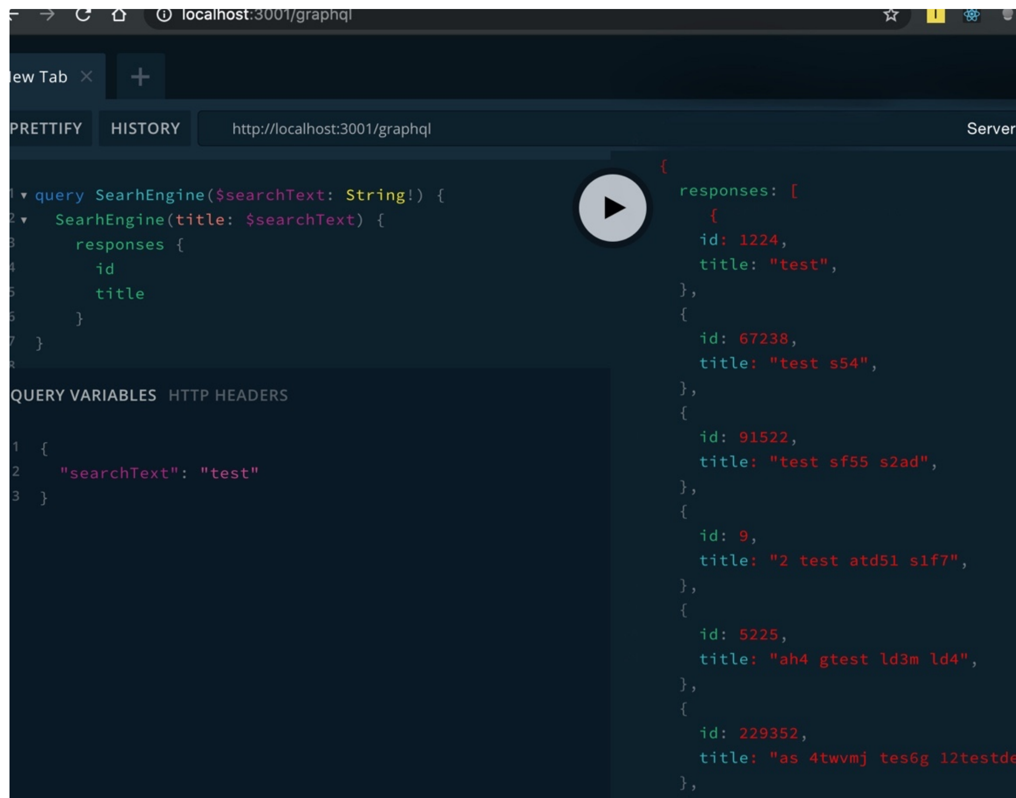

Рисунок 3.7 – Результат виконання пошукового запиту

Як можна побачити з рисунка 3.7, запит виконався успішно, адже справа від кнопки є результат – масив об’єктів, який складається з поля *id* та *title*. Також можна помітити, що масив відсортований та першими елементами в масиві є об’єкти чий *title* найкраще співпадає зі змінною *searchText*, яку вказували при запиті.

Продуктивність запропонованих алгоритмів оцінюється на двох згенерованих скриптом наборах даних, а саме на 100000 та 10000000 рядків з текстом. Експерименти проводяться на процесорі Intel Atom X3 при 1,10 ГГц та 2 ГБ пам’яті, оскільки на більш потужному обладнані різницю на великих даних помітити складніше. Середній час для отримання відповідей на

миттєвий нечіткий пошук за ключовим словом, реєструється для різної довжини префіксу та подається на рисунку 3.8. Аналогічно, для оцінки миттєвого нечіткого пошуку за кількома ключовими словами, середній час роботи для генерування відповідей на підзапити двох ключових слів та трьох ключових слів записується для різних довжин префіксу на рисунку 3.9 та рисунку 3.10 відповідно. З рисунків 3.8 – 3.10 можна зазначити, що запропонований алгоритм постійно добре працює для різної довжини префіксу.

Щоб оцінити якість відповідей, що формуються запропонованими алгоритмами та методом ранжирування, точність вимірюється шляхом визначення відсотка очікуваних результатів, отриманих цими підходами. Виходячи з дослідження, точність миттєвого нечіткого пошуку ключових слів становить 85%, а миттєвого пошуку кількох ключових слів - 90%.

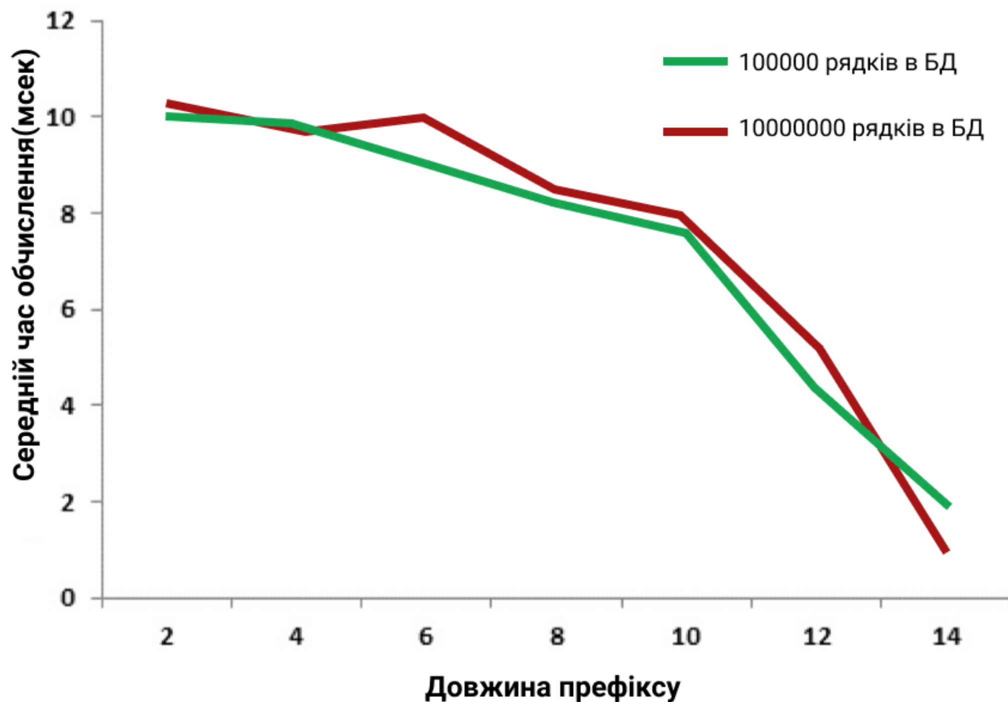

Рисунок 3.8 – Середній час обчислення миттєвого нечіткого пошуку одного ключового слова при різній довжині префіксу

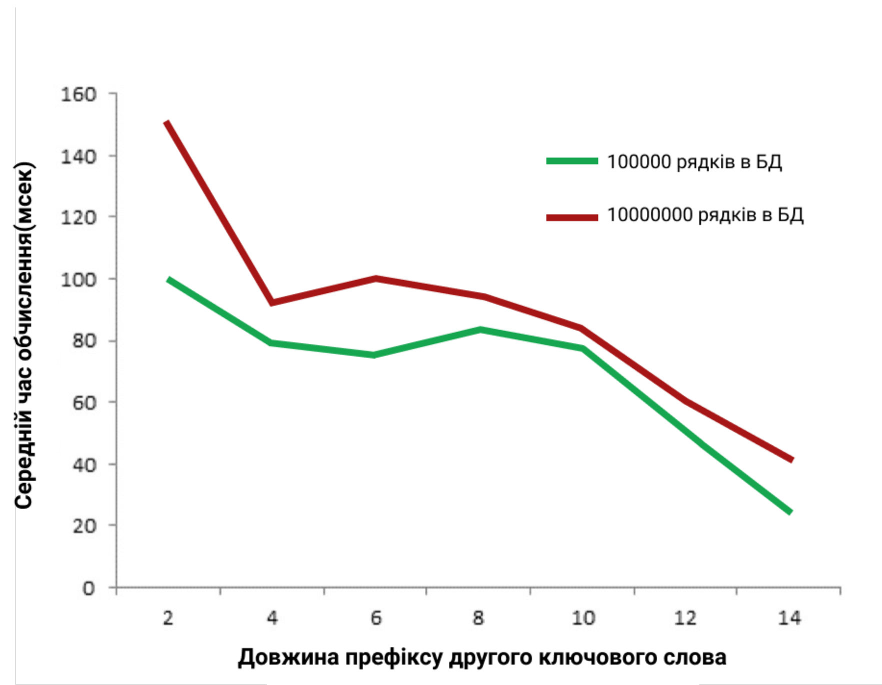

Рисунок 3.9 – Середній час обчислення миттєвого нечіткого пошуку для двох слів

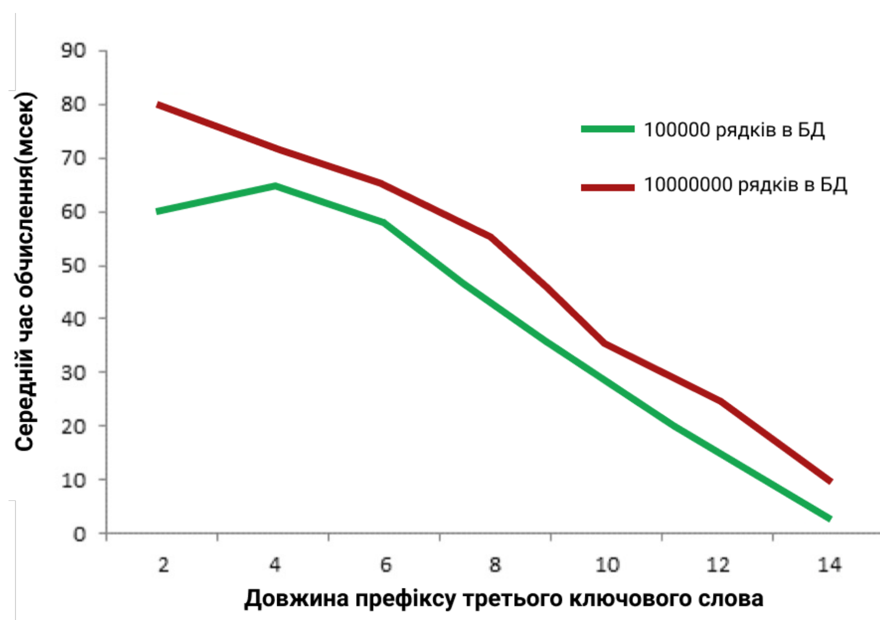

Рисунок 3.10 – Час обчислення миттєвого нечіткого пошуку для трьох слів

З рисунка 3.8 можна побачити, що середній час обчислення миттєвого нечіткого пошуку одного слова для різної довжини префіксу не залежить від кількості рядків в базі даних. Але аналізуючи рисунок 3.9, можна помітити, що швидкість виконання запиту двох слів в базі даних на 100000 в середньому на 15мс швидша ніж аналогічний пошук на 10000000 елементів, приблизно такий самий результат можна помітити і на рисунку 3.10, де пошук проводився за трьома словами.

ВИСНОВКИ

В процесі роботи було розроблено гнучку систему миттєвого нечіткого пошуку в базі даних по одному ключовому слову та при декількох ключових словах, що може видавати точні результати в залежності від заданих параметрів.

У роботі було проаналізовано алгоритми пошуку, які використовуються сьогодні на сучасних веб-ресурсах та був обраний найбільш оптимальний та швидкий алгоритм.

У цій роботі є аналізом нечіткі алгоритми миттєвого пошуку для відповіді на окремі ключові слова та запити з декількома ключовими словами. Алгоритми обчислюють відповіді поступово, гешуючи результати попереднього запиту. Крім того, наведені результати ймовірнісно-кореляційного рейтингу для визначення відповідності отриманої відповіді. Експерименти проводяться на реальних даних, які підтверджують ефективність обраного алгоритму.

Мета роботи розробити сучасну та швидку програму було виконано. Програма була розроблена з використанням технологій NestJS, GraphQL, Typescript та PostgreSQL.

У ході роботи були проведені перевірки коректності роботи програми миттєвого нечіткого пошуку при різних запитах. Результати були задовільні та показали, що програма відповідає умовам поставлений в завданні.

СПИСОК ЛІТЕРАТУРИ

1. Yun Liu, Tianmeng Gao, Baolin Song, Chengwei Huang – Personalized Fuzzy Text Search Using Interest Prediction and Word Vectorization , China, Honk Kong, 2017. - 2-3с.

2. Alex Banks, Eve Porcello. Learning GraphQL. Declarative Data Fetching for Modern Web Apps. O'Reilly Media, 2018. - 200с

3. Getting Started with NestJS [Електронний ресурс] – Режим доступу: https://www.digitalocean.com/community/tutorials/getting-started-with-nestjs

4. Нечеткий поиск в словаре с универсальным автоматом Левенштейна. [Електронний ресурс] – Режим доступу: https://habrahabr.ru/post/275937/

5. Мажуга Б В. Система нечіткого пошуку в базі даних [Текст]: робота на здобуття кваліфікаційного рівня бакалавр; спец.: 122 - комп`ютерні науки / Б.В. Мажуга; кер. В.В.Ємельяненко. - Суми: СумДУ, 2018. - 59 с.

6. Что такое TypeScript: статическая типизация для Интернета [Електронний ресурс] – Режим доступу: https://webformyself.com/chto-takoe-typescript-staticheskaya-tipizaciya-dlya-interneta/

7. PostgreSQL: The World's Most Advanced Open Source Relational Database [Електронний ресурс] – Режим доступу: https://www.postgresql.org/

8. GraphQL Foundation [Електронний ресурс] – Режим доступу: https://foundation.graphql.org/

9. PostgreSQL [Електронний ресурс] – Режим доступу: https://uk.wikipedia.org/wiki/PostgreSQL

10. Нгуен, Ной Хыу. Обзор некоторых алгоритмов нестрогого сопоставления записей применительно к задаче исключения дублирования персональных данных / Ной Хыу Нгуен. — Текст : непосредственный // Молодой ученый. — 2013. — № 5 (52). — С. 163-166.

11. NestJS — тот самый, настоящий бэкенд на nodejs. [Електронний ресурс] – Режим доступу: https://medium.com/@irustm/nestjs-%D1%82%D0%BE%D1%82-%D1%81%D0%B0%D0%BC%D1%8B%D0%B9-%D0%BD%D0%B0%D1%81%D1%82%D0%BE%D1%8F%D1%89%D0%B8%D0%B9-%D0%B1%D1%8D%D0%BA%D0%B5%D0%BD%D0%B4-%D0%BD%D0%B0-nodejs-2985772b585

12. Мазов Н.А. N-граммные методы обработки текстовой информации / Н.А Мазов — Текст: Объединенный институт геологии, геофизики и минералогии Сибирского отделения РАН, Новосибирск, России. 2016. 17с.

13. Девід Б, Шанкур В.Г., Срівес С, Срівастава ДК.: Модель відбору вибірок для ділової та історичної аналітики даних. Управління InData, аналітика та інновації. Сінгапур. Спрингер. 2020. 400с

14. GraphQL Playground Visually explore Apollo Server [Електронний ресурс] – Режим доступу: https://www.apollographql.com/docs/apollo-server/testing/graphql-playground/

15. Bast H, Weber I. Повна пошукова система: інтерактивна, ефективна та інтеграційна IP та БД. Третя дворічна конференція з інноваційних систем даних. Сан Франциско VLDB Journal. 2007. 88–95.

ДОДАТКИ

Додаток А

Код метода addSearchDoc

```
SEARCH_ENGINE.prototype.addSearchDoc = (searchDoc)  => {
            if (typeof searchDoc.id === 'undefined') {
                throw new Error(1, 'Відсутній ID .');
              };
    if (typeof searchDoc.body === 'undefined') {
        throw new Error(2, 'Пилка зчитування body.');
     };

     // Токенізований список слів
     const tokensArray = BM25.Tokenize(searchDoc.body);
      const privateTerms = {};
      const  docObj =  id: searchDoc.id, tokensArray, body:
doc.body};
     // кількість термів
    docObj.termCount = tokensArray.length;
    // Increment totalDocs
    this.totalDocs++;
    // Підрахунок середньої довжини документа (averageDocLength)
    this.totalDocumentPrivatTermLength += docObj.termCount;
    this.averageDocLength = this. totalDocumentPrivatTermLength
/ this.totalDocs;
 // Підрахунок частоти терміна
    for (let i = 0, len = tokensArray.length; i &lt; len; i++) {
        const term = tokensArray [i];
        if (!privateTerms[term]) {
            privateTerms[term] = {
                count: 0,
                freq: 0
```

```
            };
        };
        privateTerms[term].count++;
    const keys = Object.keys(privateTerms);
    for (let i = 0, len = keys.length; i &lt; len; i++) {
        const term = keys[i];
        privateTerms[term].freq = privateTerms [term].count /
docObj.termCount;
        if (!this.publicTerms[term]) {
            this. publicTerms [term] = {
                n: 0,
                idf: 0
            };
        }
        this.terms[term].n++;
    }
    this.updateIdf();
    docObj.terms = privateTerms;
    this.documents[docObj.id] = docObj;

     };
```

Додаток В

Код метода search

```
SEARCH_ENGINE.prototype.search = (string) => {
  const currentTerms = BM25.Tokenize(string);
  const results = [];

  // Перебір коженого документа по черзі.
  const keys = Object.keys(this.docs);
  for (
    let internalIndex = 0, nDocs = keys.length;
    internalIndex < nDocs;
    internalIndex++
  ) {
```

```
    const currentId = keys[internalIndex];
    // Оцінка релевантності для документа.
    this.docs[currentId].privateScore = 0;

    // Обчислення балу за кожний термін
    for (let index = 0, len = currentTerms.length; index < len;
index++) {
      const currentTerm = currentTerms[index];

      // Ми ніколи не бачили цього терміна, тому IDF буде 0.
      // Значить, ми можемо пропустити весь термін, він нічого
не додає до оцінки
      // і його немає в жодному документі.
      if (typeof this.terms[currentTerm] === "undefined") {
        continue;
      }

      // Цей термін відсутній у документі, тому частина 0.
      // терм не впливає на результату пошуку.
      if (typeof this.docs[currentId].terms[currentTerm] ===
"undefined") {
        continue;
      }

      // Термін є в документі!
      // Весь терм :
      // IDF * (TF * (k1 + 1)) / (TF + k1 * (1 - b + b *
docLength / avgDocLength))

      // IDF попередньо обчислюється для всього документа.
      const identificator = this.terms[currentTerm].idf;
      // Numerator of the TF portion.
      const num = this.docs[currentId].terms[currentTerm].count
* (this.k1 + 1);
      // Знаменник частини TF.
      const denom =
        this.docs[currentId].terms[currentTerm].count +
        this.k1 *
          (1 -
            this.b +
            (this.b * this.docs[currentId].termCount) /
              this.averageDocumentLength);
```

```
        // Додаємо до запиту цей термін запиту
        this.docs[currentId].privateScore += (identificator * num)
/ denom;
      }

      if (
        !isNaN(this.docs[currentId].privateScore) &&
        this.docs[currentId].privateScore > 0
      ) {
        results.push(this.docs[currentId]);
      }
    }

    results.sort((a, b) => b.privateScore - a.privateScore);
    return results.slice(0, 10);};
```