НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

«На правах рукопису»
УДК 004.62

До захисту допущено:

Завідувач кафедри

________ Олександр РОЛІК

«___»____________2024 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Інтегровані інформаційні системи»
зі спеціальності 126 «Інформаційні системи та технології»
на тему: «Підсистема оптимізації SQL скриптів для вирішення аналітичних задач кол-центру»

Виконала:
студентка 2 курсу, групи ІА-з21мп
Коломієць Уляна Дмитрівна ___________

Керівник:
доцент кафедри ІСТ, к.т.н., доцент
Бойко Олександра Володимирівна ___________

Рецензент:
доцент кафедри ОТ, к.т.н, заступник декана
з міжнародної діяльності
та перспективного розвитку
Волокита Артем Миколайович ___________

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студентка Коломієць У.Д. ______

Київ – 2024 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інтегровані інформаційні системи»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_______ Олександр РОЛІК

«___»____________2024 р.

ЗАВДАННЯ

на магістерську дисертацію студентці

Коломієць Уляні Дмитрівні

1. Тема дисертації «Підсистема оптимізації SQL скриптів для вирішення аналітичних задач кол-центру», науковий керівник дисертації Бойко Олександра Володимирівна, доцент кафедри ІСТ, к.т.н., доцент, затверджені наказом по університету від «03» 11 2023 р. № 5135-с.

2. Термін подання студентом дисертації «08» 01 2024 р.

3. Об’єкт дослідження: процес оптимізації SQL скриптів.

4. Предмет дослідження: методики оптимізації SQL скриптів для вирішення аналітичних задач кол-центру.

5. Перелік завдань, які потрібно виконати: вивчити та проаналізувати роботу кол-центру для визначення особливостей аналітичних задач, вирішення яких у подальшому вимагає оптимізації SQL скриптів; розглянути та проаналізувати основні відмінності SQL від інших мов для визначення особливостей їх оптимізації; розглянути та проаналізувати основні методи та методики оптимізації SQL скриптів; розробити підсистему оптимізації SQL скриптів та надати рекомендації щодо її впровадження в роботу кол-центру для підвищення продуктивності обробки даних та зменшення оцінки плану та кардинальності SQL скриптів.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу : діаграма прецедентів, діаграма компонентів, діаграма вимог, діаграма послідовностей, схема роботи оптимізатора, діаграма класів, архітектура підсистеми та системи в цілому, ілюстрації результатів досліджень.

7. Орієнтовний перелік публікацій : Oleksandr Rolik, Kseniia Ulianytska, Maryna Khmeliuk, Volodymyr Khmeliuk, Uliana Kolomiiets. Increase Efficiency of Relational Databases Using Instruments of Second Normal Form.

8. Дата видачі завдання 01.09.2023 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Пошук існуючих рішень та написання першого розділу магістерської дисертації	5-15 вересня	
2	Розробка підсистеми оптимізації	22-29 вересня	
3	Тестування підсистеми оптимізації	2-22 жовтня	
4	Дослідження отриманих результатів	27 жовтня – 13 листопада	
5	Написання та оформлення 2-5 розділів МД	17 листопада- 15 грудня	

Студентка Уляна Коломієць

Науковий керівник Олександра Бойко

РЕФЕРАТ

«Підсистема оптимізації SQL скриптів для вирішення аналітичних задач кол-центру»: 129 с., 9 табл., 35 рис., 4 дод., 27 джерел.

ОПТИМІЗАЦІЯ, STRUCTURED QUERY LANGUAGE, ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ, РЕЛЯЦІЙНІ БАЗИ ДАНИХ, АНАЛІТИЧНІ ЗАДАЧІ КОЛ-ЦЕНТРУ, ІНСТРУМЕНТИ ОПТИМІЗАЦІЇ.

Об'єктом дослідження представленої магістерської дисертації є процес оптимізації SQL скриптів.

Предмет дослідження: методики оптимізації SQL скриптів для вирішення аналітичних завдань кол-центру.

Метою роботи є підвищення продуктивності обробки даних та зменшення кардинальності та оцінки плану виконання SQL скриптів, що створюються для вирішення задач кол-центру за рахунок оптимізації відповідних SQL скриптів. Для досягнення мети розв'язано низку взаємозв'язаних окремих завдань. А саме, вивчено та проаналізовано роботу кол-центру для визначення особливостей аналітичних задач, вирішення яких у подальшому вимагає оптимізацію SQL скриптів; розглянуто та проаналізовано основні відмінності SQL від інших мов для визначення особливостей їх оптимізації; розглянуто та проаналізовано основні методи та методики оптимізації; розроблено підсистему оптимізації SQL скриптів та надано рекомендації щодо її впровадження в роботу кол-центру для покращення процесу аналізу даних. У роботі, зокрема першому розділі проводиться огляд аналітичних задач та визначення ролі SQL в аналітиці даних. Визначається вплив оптимізації на продуктивність через безпосередньо проведення експерименту. Подальші розділи присвячені аналізу існуючих методик, порівняльному аналізу сучасних оптимізаторів та визначенню вимог та розробці підсистеми, включаючи вибір методів машинного навчання для оптимізації SQL скриптів. Результатом дослідження є створена підсистема оптимізації SQL скриптів, яка зменшує кардинальність та оцінку плану SQL скриптів та підвищує продуктивність операцій, що виконуються для надання

оперативної інформації у кол-центр. У роботі також розглядається можливість впровадження розробленої системи в практику роботи кол-центру з метою покращення процесу аналізу даних. Четвертий розділ цілком присвячений розробленню стартап-проєкту для теми даної роботи з подальшим описом плану дій.

ABSTRACT

"SQL script optimization subsystem for solving analytical problems of the call center": 129 p., 9 tab., 35 draw., 4 app., 27 sources.

OPTIMIZATION, STRUCTURED QUERY LANGUAGE, INCREASE OF PRODUCTIVITY, RELATIONAL DATABASES, ANALYTICAL TASKS OF THE CALL CENTER, OPTIMIZATION TOOLS.

The object of research of the presented master's thesis is the process of optimizing SQL scripts.

The subject of research: methods of optimizing SQL scripts for solving analytical tasks of the call center.

The purpose of the work is to increase the productivity of data processing and reduce the cardinality and evaluation of the execution plan of SQL scripts created to solve the problems of the call center due to the optimization of the corresponding SQL scripts. To achieve the goal, a number of interconnected individual tasks were solved. Namely, the work of the call center was studied and analyzed to determine the specifics of analytical problems, the solution of which in the future requires the optimization of SQL scripts; considered and analyzed the main differences of SQL from other languages to determine the features of their optimization; the main optimization methods and techniques were considered and analyzed; a SQL script optimization subsystem was developed and recommendations were provided for its implementation in the work of the call center to improve the data analysis process. In the work, in particular, the first chapter provides an overview of analytical tasks and the definition of the role of SQL in data analytics. The effect of optimization on performance is determined by directly conducting an experiment. The following sections are devoted to the analysis of existing techniques, comparative analysis of modern optimizers and definition of requirements and subsystem development, including the selection of machine learning methods for optimizing SQL scripts. The result of the research is the creation of an optimization subsystem of SQL scripts, which reduces the cardinality and evaluation of the SQL script plan and increases the productivity of

operations performed to provide operational information to the call center. The work also considers the possibility of introducing the developed system into the practice of call center work in order to improve the process of data analysis. The fourth chapter is entirely devoted to the development of a startup project for the topic of this work with a further description of the action plan.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ.. 10
ВСТУП.. 12
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ................ 16
1.1 Огляд аналітичних задач кол-центру.. 16
1.2 Роль SQL у вирішенні аналітичних задачах кол-центру 19
1.3 Проблеми при вирішенні задач кол-центру 32
1.4 Вплив оптимізації на продуктивність системи... 35
1.5 Постановка задачі 40
Висновки до розділу 1... 41
2 АНАЛІЗ СУЧАСНИХ СИСТЕМ ТА МЕТОДИК ОПТИМІЗАЦІЇ 43
2.1 Аналіз існуючих методик та їх обмежень 43
2.2 Порівняльний аналіз сучасних оптимізаторів 47
2.3 Вибір технологій проектування. Переваги RUST. 54
2.4 Аналіз можливостей RUST та EGG LIBRARY... 59
Висновки до розділу 2... 65
3 РОЗРОБКА ПІДСИСТЕМИ ОПТИМІЗАЦІЇ СКРИПТІВ ДЛЯ ВИРІШЕННЯ АНАЛІТИЧНИХ ЗАДАЧ КОЛ-ЦЕНТРУ 67
3.1 Аналіз основних правил оптимізації SQL запитів 67
3.2 Розробка алгоритму оптимізації SQL скриптів 82
3.3 Визначення вимог та специфікацій. Діаграма прецедентів............................ 87
3.4 Архітектура програмного забезпечення. Діаграма класів.............................. 91
3.5 Дослідження ефективності. Опис метрик 100
Висновки до розділу 3... 104
4 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ 105
4.1 Опис ідеї проєкту 105
4.2 Технологічний аудит 105
4.3 Аналіз ринкових можливостей запуску стартап-проєкту............................. 105
4.4 Розроблення ринкової стратегії проєкту 106

Висновки та перспектива подальших досліджень 107
ВИСНОВКИ 109
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 111
ДОДАТОК А .... **Error! Bookmark not defined.**
ДОДАТОК Б .... **Error! Bookmark not defined.**
ДОДАТОК В .... **Error! Bookmark not defined.**
ДОДАТОК Г .... **Error! Bookmark not defined.**

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

БД – база даних;

SQL (STRUCTURED QUERY LANGUAGE) – декларативна мова програмування для взаємодії користувача з базами даних;

СУБД (СКБД) – системи управління (керування) базами даних;

CODASYL – (англ. COnference on DAta SYstems Languages) — Конференція по мовам систем обробки даних;

XML – (англ. EXtensible Markup Language, скорочено XML) - мова розмітки;

ANSI – (англ. American National Standards Institute) - Американський інститут національних стандартів;

FIPS – (англ. FIPS, Federal Information Processing Standards) — стандарти з публічним доступом;

OLAP – (англ. online analytical processing, аналітична обробка у реальному часі) — інтерактивна система, що дозволяє переглядати різні підсумки по багатовимірних даних;

JDBC – (англ. Java DataBase Connectivity) — з'єднання з базами даних на Java;

GIL – (англ. Global Interpreter Lock, GIL) — механізм інтерпретатора, який гарантує що в кожен момент часу виконується код лише одного потоку, для уникнення конкурентного доступу до спільних структур даних;

ETL – (Extract, Transform, Load (ETL) або витяг, перетворення та завантаження) — процес, який використовується в базах даних для забезпечення їх роботи для підтримки прийняття рішень;

ООП – (англ. Object-oriented programming, OOP) — одна з парадигм програмування, яка розглядає програму як множину «об'єктів», що взаємодіють між собою;

UML – (англ. Unified Modeling Language) — уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування;

ACID – (англ. Atomicity, Consistency, Isolation, Durability) — набір властивостей, що гарантують надійну роботу транзакцій бази даних: атомарність, узгодженість, ізольованість, довговічність.

ВСТУП

Актуальність обраної теми обумовлена декількома факторами, зокрема рядом факторів та потреб у сучасних кол-центрах. В цей перелік можна включити зростання обсягів даних, нові вимоги до продуктивності, підвищення конкурентоспроможності, зростаючі обсяги даних для аналізу. Запити на аналітику стають більш складними, а користувачі очікують оперативних відповідей у реальному часі. Оптимізація SQL скриптів стає ключовим чинником для забезпечення швидкості обробки даних, ефективного використання ресурсів та підтримання конкурентоспроможності кол-центру в сучасному бізнес-середовищі. У сучасному світі генерується надзвичайно велика кількість даних, особливо в галузі обслуговування клієнтів. Це включає в себе інформацію про виклики клієнтів, наявні офери, наявні продукти у клієнтів, їх рахунки та інші аспекти взаємодії. Обробка та аналіз цих даних стає важливою для прийняття управлінських рішень і забезпечення якості обслуговування клієнтів. Кол-центри вимагають максимально ефективних систем для обробки даних та аналітики. Швидкість виконання SQL скриптів важлива для надання вчасної інформації та реагування на вимоги клієнтів. В умовах загостреної конкуренції кол-центри шукають способи підвищення якості своїх послуг і зниження витрат. Оптимізація SQL скриптів може допомогти досягти цих цілей зменшуючи час та ресурси, витрачені на аналітичну обробку. Аналітика грає ключову роль у виборі стратегічних рішень кол-центру. Швидкий доступ до точної та релевантної інформації може сприяти кращим рішенням щодо управління ресурсами та покращення якості обслуговування клієнтів. Кол-центри накопичують все більше даних про клієнтів та їхні вимоги. Оптимізація SQL скриптів стає ключовою для ефективного аналізу цих даних та виявлення основних патернів. Отже, розробка підсистеми оптимізації SQL скриптів для аналітичних задач кол-центру має значиму актуальність, оскільки вона відповідає потребам сучасного бізнесу у швидкому та ефективному аналізі даних для прийняття обґрунтованих рішень та підвищення якості обслуговування клієнтів. Ця тема має великий практичний і науковий потенціал для оптимізації роботи кол-центрів.

На даний момент для оптимізації SQL скриптів використовують різноманітні оптимізатори, проте ці рішення мають свої недоліки. Зокрема багато з систем оптимізації можуть зреагувати непередбачувано до раптового зростання обсягів даних, що ускладнює їх ефективне використання в динамічних середовищах кол-центрів. Деякі оптимізатори також можуть бути недостатньо швидкими для роботи в режимі реального часу, не забезпечують достатню адаптабельність до різних типів завдань, та виявляються обмеженими в оптимізації складних SQL скриптів. Крім того, їхня ефективність може залежати від постійного моніторингу та налаштувань, що ускладнює їх практичне використання без спеціалізованого підходу певного співробітника.

Метою роботи є підвищення продуктивності обробки даних та зменшення кардинальності та оцінки плану виконання SQL скриптів, що створюються для вирішення задач кол-центру за рахунок оптимізації відповідних SQL скриптів. Для досягнення мети розв'язано низку взаємозв’язаних окремих завдань. А саме, вивчено та проаналізовано роботу кол-центру для визначення особливостей аналітичних задач, вирішення яких у подальшому вимагає оптимізацію SQL скриптів; розглянуто основні відмінності SQL від інших мов для визначення особливостей їх оптимізації; проаналізовано основні методики оптимізації; розроблено підсистему оптимізації SQL скриптів та надано рекомендації щодо її впровадження в роботу кол-центру для покращення процесу аналізу даних.

Об'єктом дослідження представленої магістерської дисертації є процес оптимізації SQL скриптів. Дослідження було спрямоване на розробку підсистеми, яка забезпечить оптимальну обробку та аналіз великого обсягу даних для підтримки ефективної роботи кол-центру в умовах сучасного бізнес-середовища.

Предметом дослідження є методики оптимізації SQL скриптів для вирішення аналітичних завдань кол-центру.

В магістерській дисертації використовуються різні методики дослідження, спрямовані на глибоке розуміння та оптимізацію SQL скриптів для вирішення аналітичних завдань кол-центру. Перелік методів обрано таким чином, щоб вони логічно доповнювали один одного та забезпечували повноцінний аналіз предмету

дослідження. Початковий етап дослідження передбачає докладний аналіз наукових та технічних джерел для отримання глибокого розуміння сучасних тенденцій у сфері оптимізації SQL скриптів та їх впливу на вирішення аналітичних завдань кол-центрів. Також виконано експериментальні дослідження. Здійснено серії експериментів для об'єктивного вивчення ефективності різних методів оптимізації на конкретних аналітичних завданнях, що реалізуються у кол-центрах. Також використано case study - це детальне дослідження конкретного випадку, ситуації, події чи процесу в реальних умовах. У даному випадку дослідження оптимізації SQL скриптів для аналітичних завдань кол-центру. Case study включає в себе ретельний аналіз впровадження оптимізованих скриптів. Реалізація case study виконана для глибокого розгляду впливу впроваджених оптимізацій на продуктивність та ефективність роботи бази даних в реальних умовах. Виконано аудит бази даних для виявлення можливостей оптимізації та аналізу поточного стану системи. Ці методи дозволяють систематично дослідити, обґрунтувати та представити оптимальні рішення для оптимізації SQL скриптів у контексті аналітичних завдань кол-центру, що відповідає загальній меті магістерської дисертації.

Наукова новизна роботи полягає у створенні підсистеми оптимізації SQL скриптів для ефективного вирішення аналітичних задач кол-центру зі здатністю розпізнавати складові SQL коду з наступним розподіленням його на окремі блоки та застосуванням алгоритмів оптимізації, перетворенням або заміщенням певних ділянок коду з метою підвищення продуктивності та ефективності скриптів. Ця інноваційна підсистема сприяє автоматизації процесу оптимізації SQL скриптів, що дозволяє значно збільшити швидкодію та ресурсозбереження в області обробки аналітичної інформації

Практичне значення: ефективність роботи з базою даних безпосередньо впливає на продуктивність системи та час відповіді для користувачів. SQL Optimizer Plus може використовуватися в аналітичних системах, CRM-системах та інших додатках, де використовуються складні SQL скрипти для отримання даних. Наприклад, кол-центр може застосовувати цей інструмент для оптимізації запитів, які використовуються для аналізу клієнтської інформації, статистики дзвінків та інших

операцій. Це дозволяє зменшити час виконання запитів, підвищити загальну продуктивність та забезпечити кращий користувацький досвід.

Публікація: Oleksandr Rolik, Kseniia Ulianytska, Maryna Khmeliuk, Volodymyr Khmeliuk, Uliana Kolomiiets. Increase Efficiency of Relational Databases Using Instruments of Second Normal Form. [1]

Також при розвантаженні бази даних від важких запитів аналітики зможуть покращити швидкість виконання завдань у межах своїх професійних обов'язків, а отже значно збільшити кількість виконуваних задач кожного співробітника, який був задіяний у процесах оптимізації. Так як при сильному навантаженні бази даних важко отримувати бажані результати за лічені хвилини. Тому даний оптимізатор буде використовуватись як незамінний додаток у кожному аналітичному відділі.

Дана підсистема має бути зручним інструментом у руках кожного аналітика кол-центру, збільшивши швидкість виконання кожного окремого запиту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ І ПОСТАНОВКА ЗАВДАННЯ

1.1 Огляд аналітичних задач кол-центру

У світі, де конкуренція на ринку стає все більш інтенсивною, підприємства шукають нові та ефективні способи взаємодії з клієнтами. Спрямованість на клієнта та його задоволення стали визначальними факторами для успіху. У цьому контексті виникає необхідність в інноваційних методах управління клієнтським досвідом. Кол-центр, як ключовий елемент стратегії взаємодії з клієнтами, виступає як надійний посередник між компанією та її клієнтами. Організація володіє потужними можливостями для обробки великого обсягу вхідних та вихідних дзвінків, а також для забезпечення інших форм комунікації, таких як чати, колбеки. Кол-центр визначається не лише як платформа для вирішення проблем та надання інформації, але і як стратегічний інструмент для покращення клієнтського досвіду. Забезпечення якості в цьому контексті означає надання оперативної та точної інформації, а також ефективного вирішення проблем клієнтів. Одним із ключових аспектів розвитку кол-центру є перехід до мультиканального підходу до комунікації. Крім традиційних дзвінків, часто використовуються чати та інші електронні форми комунікації. Це не лише розширює можливості взаємодії, але й ставить нові вимоги до ефективності та оптимізації процесів. Розвиток технологій, зокрема штучного інтелекту та аналітики даних, стає ключовим фактором для оптимізації функцій кол-центру. Використання аналітичних інструментів дозволяє не тільки вдосконалити роботу операторів, але і надати персоналізовані та ефективні рішення клієнтам. Отже, кол-центр, у контексті сучасного бізнес-середовища, виступає як стратегічний інструмент для ефективної взаємодії з клієнтами. Використання популярних комунікаційних каналів, застосування інноваційних технологій та підвищення якості обслуговування — це ключові аспекти, які визначають роль та значення кол-центру в сучасному бізнес-середовищі.

Аналітичні завдання, які виникають перед кол-центром у сфері аналітики баз даних представляють собою важливий виклик, оскільки вони вимагають глибокого

розуміння інформації, що зберігається у базі, а також вміння використовувати аналітичні інструменти для отримання цінних інсайтів. Важливою задачею є аналіз ефективності операцій кол-центру. Однією з ключових аналітичних задач є визначення ефективності операцій кол-центру на основі даних, що зберігаються у базі. Це включає аналіз часу відповідей на дзвінки, швидкості вирішення проблем клієнтів, а також використання ресурсів. Другою важливою задачею є аналіз поведінки клієнтів та тенденцій. Аналіз баз даних дозволяє кол-центру відстежувати та аналізувати поведінку клієнтів. Це включає в себе ідентифікацію та вивчення тенденцій, взаємодію з продуктами та послугами, а також визначення факторів, які впливають на задоволеність клієнтів. Третьою задачею є прогнозування обсягу роботи та навантаження. Аналіз баз даних дозволяє кол-центру прогнозувати обсяг роботи та навантаження на основі історичних даних. Це сприяє оптимальному розподілу ресурсів, забезпеченню високої доступності та ефективній роботі кол-центру. Не менш важливою задачею є виявлення точок оптимізації та покращення процесів. Аналітика баз даних виявляє можливості оптимізації та покращення процесів кол-центру. Це включає в себе визначення ефективності використання ресурсів, виявлення слабких місць у системі та визначенні стратегій оптимізації. Аналіз аналітичних задач кол-центру в контексті аналітики баз даних виявляється критично важливим для його функціонування. Від оптимізації операцій до прогнозування та виявлення можливостей покращення, аналітика даних є ключовою для надання ефективних та якісних послуг клієнтам.

У сучасному світі, де взаємодія з клієнтами визначає успіх бізнесу, кол-центр відіграє важливу роль у забезпеченні якісного та ефективного обслуговування клієнтів. Зростаюча конкуренція та високі очікування споживачів ставлять завдання перед кол-центрами щодо постійного покращення своєї роботи та адаптації до змін у вимогах ринку. Впровадження системи метрик є важливим етапом у підвищенні якості обслуговування клієнтів та оптимізації ресурсів. Аналітики кол-центру звикли оперувати наступними основними метриками (див. таблицю 1.1). Найчастіше наступні метрики використовують.

Таблиця 1.1 – Основні метрики кол-центру

Назва	Уніфікована назва	Опис
Час відповіді	Average Response Time	Визначає середній час за який оператор кол-центру реагує на вхідний запит чи дзвінок. Низький час відповіді є показником ефективності та здатності операторів оперативно реагувати на потреби клієнтів
Час розмови	Average Talk Time	Визначає середній час, який оператор витрачає на розмову з клієнтом. Ефективне управління часом розмови важливе для забезпечення якісного обслуговування та максимальної ефективності роботи кол-центру
Кількість чатів за годину	Chat Volume per Hour	Вказує на кількість чатів, які обслуговує кол-центр за годину. Визначення пікових годин та навантаження дозволяє оптимізувати ресурси та забезпечити швидке вирішення питань клієнтів
Ефективність вирішення запитів	Query Resolution Efficiency	Визначає відсоток успішно вирішених запитів відносно загальної кількості отриманих. Дозволяє визначити наскільки ефективно кол-центр вирішує проблеми клієнтів

Назва	Уніфікована назва	Опис
FTE	Full-Time Equivalent (FTE)	Визначає кількість повних робочих годин, які витрачаються на обслуговування одного оператора. Розрахунок включає в себе не лише робочий час, але й інші фактори, такі як перерви, тренування та адміністративні завдання. Дозволяє ефективно використовувати час оператора та визначати оптимальну кількість ресурсів для підтримки плаваючого обсягу роботи.
FCR	First Contact Resolution	Визначає відсоток випадків, які вирішуються під час першого контакту з клієнтом. Високий FCR свідчить про ефективність кол-центру в розв'язанні проблем без необхідності повторних звернень. Вимірювання цієї метрики дозволяє оцінити якість обслуговування та ефективність вирішення проблем з першого разу
CSAT	Customer Satisfaction Score	Визначає рівень задоволеності клієнтів і є однією з ключових метрик для вимірювання якості обслуговування.

1.2 Роль SQL у вирішенні аналітичних задачах кол-центру

Перш за все варто з'ясувати що таке Structured Query Language та в яких випадках його застосовують спеціалісти, а також які основні ознаки даної мови.

Фахівці обробки великих обсягів даних чи аналітики дають коротке визначення даному інструменту, SQL – це інструмент аналізу даних. Простіше кажучи, SQL - це мова програмування за допомогою якої ми можемо обирати необхідні за технічним

завданням дані, що зберігаються в реляційній базі даних. SQL - це мова, яку фахівці використовують для взаємодії з базами даних.

Звертаючись до визначення за Аланом Больє, SQL – мова для формування, маніпулювання та вилучення даних з реляційної БД. Однією з причин популярності реляційних баз даних є їх здатність опрацьовувати великі обсяги інформації при правильному проектуванні. У взаємодії з обширними даними SQL можна порівняти з сучасним цифровим фотоапаратом із потужним об'єктивом: він дозволяє переглядати об'ємні дані або здійснювати "глибокий фокус", тобто фокусуватися на конкретних деталях. [2]

Оптимізація запитів SQL є важливою навичкою для розробників, адміністраторів та аналітиків, оскільки вона може значно підвищити ефективність і швидкість реагування програм баз даних.

Розпочати дослідження історії розвитку SQL варто з історії розвитку тих інструментів, для яких було винайдено цю мову, а саме реляційних баз даних. Історія розвитку систем керування базами даних розпочалась у 1960-ті роки. У цей період народжуються перші системи керування базами даних (СКБД). Виникає мережева модель даних CODASYL і компанія North American Rockwell розробляє ієрархічну БД, яка пізніше лягла в основу створення системи IMS від IBM. У 1970-му році Едгар Ф. Кодд формулює теоретичні засади реляційної моделі даних, яка спочатку цікавить лише академічне середовище. Далі поява експериментальних реляційних СКБД, таких як Ingres (в Берклі) та System R (IBM), що були анонсовані у 1976 році. У 1980-х роках з'являються перші комерційні версії реляційних СКБД, такі як Oracle та DB2. Реляційні СКБД починають успішно витісняти мережеві та ієрархічні системи. Ведеться дослідження розподілених СКБД, але вони не отримують широкого застосування на ринку. Увага науковців у 1990-х роках спрямовується на об'єктно-орієнтовані СКБД, особливо в тих галузях, де використовуються складні дані, такі як інженерні та мультимедійні БД. Основним нововведенням у 2000-х роках стає підтримка та використання формату XML в СКБД. Розвиток вільного програмного забезпечення створює конкуренцію для комерційних СКБД. [3]

Кодд запропонував мову для роботи з даними в реляційних таблицях, названий DSL/Alpha. Незабаром після публікації статті Кодда в IBM була організована група для створення прототипу мови на базі його ідей. Ця група розробила спрощену версію DSL/Alpha, яку назвали SQUARE. У результаті вдосконалення SQUARE з'явилася мова SEQUEL (цей проект включав в себе створення системи управління реляційними базами даних та мови запитів з назвою SEQUEL (Structured English Query Language — англійська мова структурованих запитів)), яка зрештою отримала ім'я SQL. Початкова назва мови була SEQUEL, але з часом слово "English" втратилося, і абревіатура прийняла той вигляд, який став загальноприйнятим. SQL спрямований на зручне та зрозуміле формулювання запитів до реляційних баз даних для користувачів. Також від самого початку він отримав визнання як "повна мова баз даних". Це означає, що SQL включає інструменти для визначення та маніпулювання схемою бази даних, встановлення обмежень цілісності та тригерів, визначення структур фізичного рівня для ефективного виконання запитів, авторизації доступу до відносин та їх полів, а також визначення точок збереження транзакцій та виконання фіксації та відкату транзакцій.

У середині 1980-х розпочався процес розробки першого стандарту мови SQL Національним інститутом стандартизації США (American National Standards Institute, ANSI), який був опублікований у 1986 році. Подальші вдосконалення були анонсовані у наступних версіях стандарту SQL (1989, 1992, 1992 роки). [4]

Для наочного відображення основних змін в мові звернемось до таблиці 1.2.

Таблиця 1.2 – Стандарти SQL

Рік	Назва	Синонім	Зміни
1986	SQL-86	SQL-87	Перший стандарт погоджено за ANSI, а також погоджений ISO в 1987 році.

Рік	Назва	Синонім	Зміни
1989	SQL-89	FIPS 127-1	Попередня версія поліпшена.
1992	SQL-92	SQL2, FIPS 127-2	Видимі покращення (ISO 9075). Стандарт FIPS 127-2 прийняв рівень Entry Level
1999	SQL:1999	SQL3	Розширено функціонал шляхом додавання нескалярних типів даних, об'єктно-орієнтованих можливостей регулярних виразів, рекурсивних запитів, підтримки тригерів та базових процедурних розширень.
2003	SQL:2003		Введено розширення для обробки XML даних, віконні функції (застосовуємо для OLAP- баз даних), генератори послідовностей і засновані на них типи даних.
2006	SQL:2006		Функціонал роботи з XML-даними значно розширено. Поява можливості спільного використання SQL та XQuery в запитах.
2008	SQL:2008		Покращено можливості віконних функцій, виправлені неоднозначності стандарту SQL:2003.

SQL має свої власні цілі та сферу застосування, і він беззаперечно не є заміною для мов програмування загального призначення. Він налаштований для роботи з даними у контексті реляційних баз даних. Тим не менш, його можна порівнювати з деякими іншими мовами, але з урахуванням відмінностей в цілях і сферах застосування. Наприклад, SQL та Python можуть порівнюватися в контексті роботи з даними, оскільки Python надає потужні бібліотеки для роботи з базами даних, такі як SQLAlchemy та Django ORM. Однак Python також є універсальною мовою програмування, яка може виконувати безліч інших завдань, в той час як SQL спеціалізований для роботи з даними в базах даних. SQL і Java можуть порівнюватися, коли йдеться про розробку додатків, які використовують реляційні бази даних. Java часто використовується для створення додатків, які взаємодіють з базами даних з використанням JDBC (Java Database Connectivity). SQL і C# також можуть порівнюватися особливо при розробці додатків під платформу Microsoft, оскільки C# має інтеграцію з Microsoft SQL Server та іншими СУБД. Однак важливо розуміти, що SQL та багато інших мов програмування мають різні цілі та застосування і їх порівняння залежить від контексту. SQL призначений для роботи з даними в реляційних базах даних, у той час як інші мови можуть мати ширший спектр завдань та можливостей.

Python і SQL є популярними мовами у світі програмування. Критична відмінність між ними полягає в тому, що в той час як Python є мовою програмування високого рівня, яка використовується для створення програм і дослідження даних, SQL є високопродуктивною мовою, яка використовується для спілкування з базами даних. (див. таблицю 1.3)

Таблиця 1.3 – Порівняння Python та SQL за основними критеріями

Категорія	Python	SQL
Продуктивність	Повільніше для великих обчислень	Швидша продуктивність для простих запитів і агрегацій

Категорія	Python	SQL
Функціональні можливості спричинити блокування	Широкі, завдяки інтеграції з великою кількістю бібліотек	Обмежені, оскільки бібліотеки сторонніх розробників не такі широкі і інтеграція з цими бібліотеками може спричинити блокування
Тестування	Широке тестування модулів і інтеграції через процес конвеєра коду	Тестування зазвичай відбувається під час виробництва, і немає розширених модульних тестів
Масштабованість	Використовує GIL (Global Interpreter Lock), який обмежує швидкість і продуктивність, коли система потребує розширення	SQL може збільшити/зменшити масштаб шляхом додавання/видалення таблиць із бази даних.
Простота використання	Простий у використанні синтаксис, однак існує кілька концепцій, які потрібно вивчити, що може збільшити труднощі	Дуже зручний для початківців, з меншою кількістю понять для вивчення
Налагодження	Налагодження в Python легше завдяки точкам зупинки	Розділяє моделі SQL на кілька файлів, щоб допомогти з налагодженням

Категорія	Python	SQL
Ролі	Python має вирішальне значення для таких людей, як спеціалісти з обробки даних, оскільки він містить ряд бібліотек, необхідних для виконання багатьох завдань, таких як маніпулювання даними, суперечки та дослідження	Інженерам даних потрібні значні навички SQL для моделювання даних і ETL.

Щоб підкреслити основні характеристики мови SQL слід провести паралелі з Java. SQL є стандартною предметно-спеціальною мовою програмування, яка полегшує зберігання, обробку та вилучення даних у реляційних базах даних, Java є високорівневою мовою програмування для різноманітних веб-додатків загального кодування. Основні відмінності наведено в таблиці 1.4.

Таблиця 1.4 - Порівняння Python та Java за основними критеріями

Категорія	SQL	Java
Тип мови	Стандартна мова запитів (декларативна, предметно-залежна)	Об’єктно-орієнтована мова програмування (ООП) (загального призначення, імперативна, скриптова, платформо-незалежна)

Категорія	SQL	Java
Використання	Для спілкування з базами даних, оцінювання та маніпулювання даними	Для створення додатків консолей, ПК, телефонів, ЦОД та інших пристроїв
Швидкість	Обробка даних у SQL є швидшою, ніж написання коду в Java	Java повільніше, ніж SQL
Популярність	Третя за популярністю мова програмування у 2023 році	П’ята за популярністю мова програмування у 2023 році

У 2020 році Stack Overflow — популярна система питань і відповідей для професійних програмістів провела опитування серед усіх користувачів та професіоналів щодо найпоширенішої мови програмування. Дуже цікавим є той факт, що серед усіх користувачів, а саме 54.7% (див. рисунок 1.1) та серед професійних розробників, а саме 56.9% (див. рисунок 1.2) SQL займає третє місце за частотою використання. [2]

Однією з головних причин такої поширеності є простота використання. SQL має простий синтаксис, який дозволяє легко створювати запити для отримання, вставлення, оновлення та видалення даних з бази даних. Оскільки SQL є давно використовуваною та поширеною мовою, існує велика спільнота користувачів та безліч ресурсів, таких як документація, форуми та навчальні матеріали, які полегшують вивчення та роботу з нею. SQL використовується у різних областях, включаючи веб-розробку, бізнес-аналітику, наукові дослідження та інші. Його універсальність робить його корисним для різних завдань. Також SQL підтримує транзакційні операції, що дозволяє виконувати групу операцій атомарно, тобто або всі вони виконуються, або жодна з них. Загалом, SQL є потужним та універсальним

інструментом для роботи з базами даних, що робить його надзвичайно популярним у сфері розробки програмного забезпечення та управління даними.

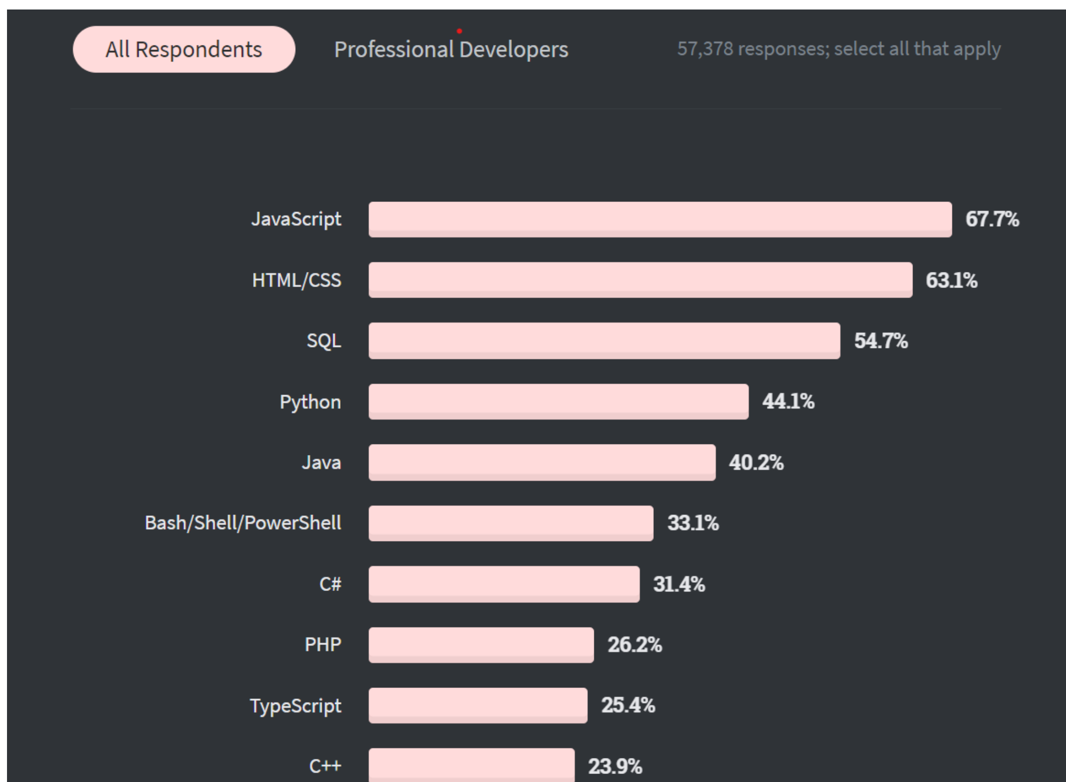


Рисунок 1.1 – Поширеність мов програмування (усі користувачі)

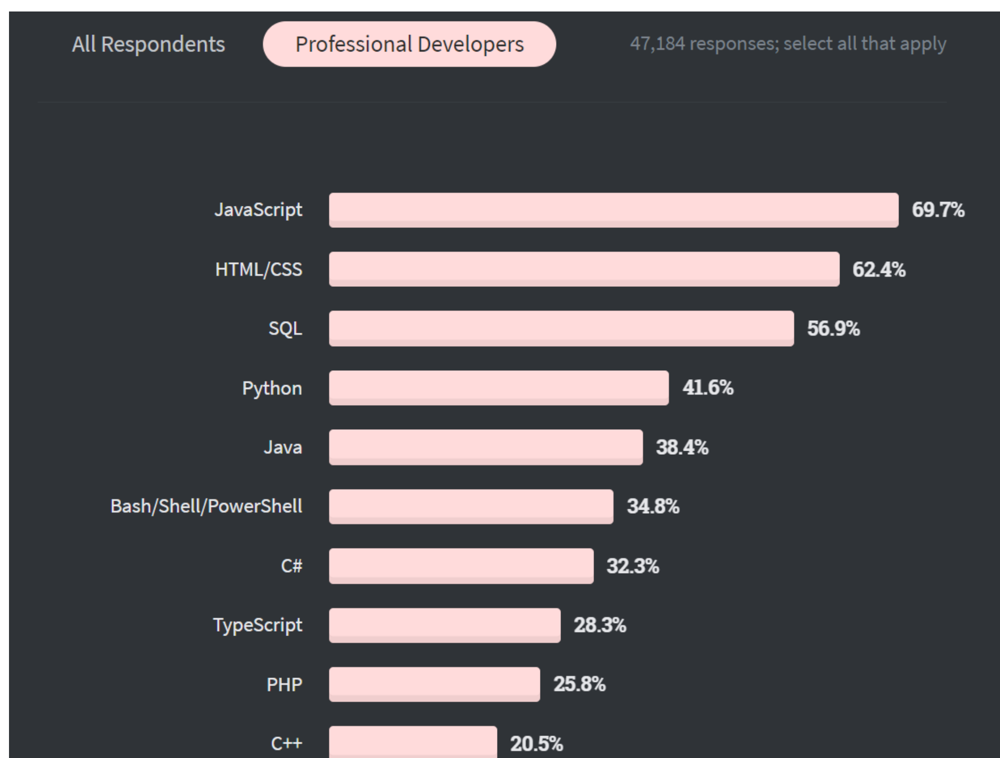


Рисунок 1.2 - Поширеність мов програмування (розробники)

Отже, можна зробити висновок про те, що мова SQL є дуже популярною і широко використовуваною в сфері реляційних баз даних. SQL застосовують для створення, редагування і видалення даних у базах. Вона має багато різних реалізацій (наприклад, Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL, SQLite і багато інших), вона використовується в різних галузях, включаючи веб-розробку, бізнес-аналітику, адміністрування баз даних та інші галузі. SQL є стандартом для взаємодії з реляційними базами даних, і багато сучасних програмних продуктів та систем підтримують її. SQL також є однією з основних навичок, які часто шукають роботодавці в інформаційних технологіях і базах даних. Тому знання SQL може бути корисною навичкою для інженерів програмного забезпечення, аналітиків даних та інших фахівців у сфері обробки даних. [5]

NoSQL (англ. «не тільки SQL») — це широкий клас баз даних, який спрямований на забезпечення максимальної масштабованості і доступності, у порівнянні з традиційними реляційними базами даних. Системи NoSQL мають наступні основні властивості: зазвичай використовують розподілену та відмовостійку архітектуру, в якій можна легко додати більшу кількість комп'ютерів; дані копіюються для забезпечення відмовостійкості; підтримують спрощену транзакційну модель. Спрощена транзакційна модель в основному полягає в автоматичному управлінні транзакціями, тобто спрощена модель може передбачати автоматичне управління транзакціями без необхідності явного вказування команд початку (BEGIN TRANSACTION), фіксації (COMMIT), або відміни (ROLLBACK). Система може автоматично визначати та керувати границями транзакцій. Спрощена модель може використовувати певний рівень ізоляції транзакцій за замовчуванням без великої кількості конфігураційних параметрів. Рівень ізоляції визначає, наскільки одна транзакція ізолюється від інших. Спрощена транзакційна модель може включати в себе автоматичне відновлення транзакцій після помилок, спрощуючи таким чином відновлення консистентності бази даних. Спрощена модель використовує оптимізації для мінімізації використання блокування та забезпечення більшої продуктивності в умовах конкурентного доступу до даних. [6]

Існують різні типи NoSQL (див. рисунок 1.3).

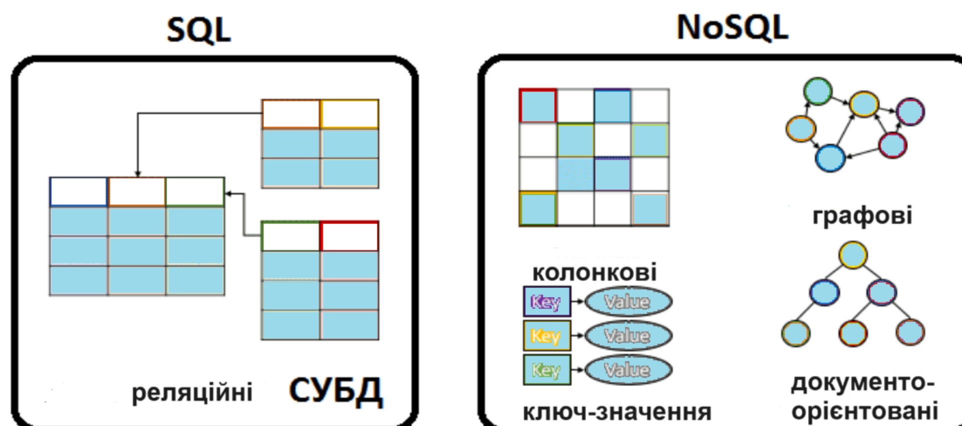


Рисунок 1.3 – Види NoSQL баз даних

NoSQL бази даних - це клас баз даних, які відрізняються від традиційних реляційних систем керування базами даних (СКБД). Основна характеристика NoSQL баз це гнучкість схеми, що дозволяє зберігати дані різної структури без строгих обмежень на відношення між даними. Вони також славляться масштабованістю, здатністю обробляти великі обсяги даних та розподіленістю.

NoSQL бази даних можна розподілити на чотири основні категорії, як вказано в таблиці 1.5. Зазвичай це включає бази даних документів, які легко масштабуються і можуть змінювати свою структуру за потреби; стовпчасті бази даних, які активно використовуються в аналітичних системах і сервісах, що опрацьовують великі обсяги даних, і де групування властивостей колонок при запиті індексує менший обсяг даних, що забезпечує високу швидкість виконання; графові бази данихзазвичай налаштовані на проектування складних віднощень за допомогою теорії графів, де ребра графа видаються зв'язками, а вузли - об'єктами; і бази даних ключ-значення, які є найпростішим видом нереляційних баз даних і відрізняються високою швидкістю доступу до даних завдяки адресному зберіганню. Таблиця 1.5 відображає класифікацію усіх NoSQL баз даних і допомагає зрозуміти їх тип з детальним описом і наведенням прикладів.

Таблиця 1.5 – Класифікація NoSQL баз даних

Тип	Опис	Приклад
База даних документів	Ці бази даних використовують документи, такі як JSON або XML, для зберігання даних.	MongoDB і Couchbase
Стовпчаста база даних	Дані зберігаються у вигляді стовпців	Apache Cassandra та HBase
Графові бази даних	Вони призначені для зберігання та операцій над графовими даними	Neo4j та Amazon Neptune
Ключ-значення бази даних	Ці бази даних зберігають дані у вигляді ключ-значення	Redis і Riak

NoSQL бази даних стали популярними у веб-додатках та соціальних мережах завдяки їх здатності зберігати та обробляти великі обсяги даних користувачів. Наприклад, MongoDB використовується для зберігання профілів користувачів, текстових повідомлень та інших даних, пов'язаних з соціальними мережами. NoSQL бази даних грають важливу роль у зборі, зберіганні та аналізі даних, зібраних з різних IoT-пристроїв. Наприклад, Apache CouchDB може бути використаний для зберігання та реплікації даних, зібраних від сенсорів та пристроїв IoT.

Сфера застосування у цього виду баз даних різноманітна. NoSQL бази даних можуть бути використані для обробки великих обсягів даних (Big Data) та аналітики. Наприклад, Apache HBase може бути використаний для зберігання великих обсягів структурованих даних, які застосовуються в аналітичних операціях. Графові бази даних, такі як Neo4j, дозволяють зберігати та запитувати графові дані, які використовуються в різних галузях, включаючи соціальні мережі, рекомендації та аналітику мереж.

Отже, NoSQL бази даних є важливим інструментом для зберігання, обробки і аналізу різних типів даних в сучасному світі. Вони використовуються в різних

галузях, де потрібно ефективно опрацьовувати нереляційні дані та масштабувати системи для забезпечення швидкодії.

Для наочного розуміння основних відмінностей реляційних та нереляційних баз даних звернемось до порівняльної таблиці 1.6 (див. таблицю 1.6). Головна перевага реляційних баз даних полягає в стабільності та надійності зберігання інформації, мінімізації ризику втрати даних. У порівнянні з нереляційними базами даних, реляційні відповідають стандартам ACID - набору вимог до транзакційних систем. Їхнє відповідання забезпечує цілісність, збереження і передбачуваність роботи бази даних. Атомарність (Atomicity) гарантує, що жодна транзакція не буде зафіксована частково в системі. Непротирічність (Consistency) вимагає фіксації лише припустимих результатів транзакцій. Ізольованість (Isolation) забезпечує незалежність результатів однієї транзакції від паралельних транзакцій. Довговічність (Durability) гарантує збереження змін в базі даних незалежно від збоїв чи впливу користувачів.

Таблиця 1.6 – Порівняння реляційних та нереляційних баз даних

Аспект	Реляційні	Нереляційні
Модель даних	Використовують табличну модель даних, де дані організовані у вигляді таблиць з рядками і стовпцями	Використовують різні моделі даних, такі як документальні (JSON або XML), стовпчасті, графові, ключ-значення та колонкові
Схема даних	Вимагають строго визначеної схеми даних, яка описує структуру таблиць, типи даних. Зміна схеми може бути складною та вимагати планування а також попереднього розгляду схему для подальшої переробки задля коректних зв’язків	Не вимагають фіксованої схеми. Додавання, зміна та видалення полів може виконуватися без значних зусиль. Це називається "гнучкою схемою" (schema-less)

Аспект	Реляційні	Нереляційні
Мова запитів	Використовують мову SQL	Мають свої власні мови запитів або можуть використовувати інші інтерфейси для взаємодії з даними, такі як API
Масштабованість	Складніше масштабувати горизонтально через строгу схему	Підходять для горизонтального масштабування, тобто додавання нових серверів для розподілення навантаження та збільшення продуктивності
Додаткові операції	Мають стандартний набір SQL операцій, які використовуються для роботи з таблицями та даними	Можуть надавати специфічні операції, призначені для конкретних моделей даних, такі як пошук по графових структурах чи операції з геоданими
Використання	Використовуються для сховищ даних, де важлива структура та цілісність даних	Застосовуються там, де необхідно оперувати неструктурованими даними

Отже, вибір між реляційними і NoSQL базами даних залежить від конкретних вимог проєкту та характеристик даних, які необхідно зберігати та обробляти. [7]

1.3 Проблеми при вирішенні задач кол-центру

З кожним роком з космічною швидкістю зростає кількість оброблюваних цифрових даних у світі. Кількість даних, що генерується щорічно, зростає з року в рік починаючи з 2010 року. Фактично вважається, що 90% світових даних було згенеровано лише за останні два роки. За 13 років цей показник зріс у 57 разів, починаючи з 2 зеттабайтів у 2010 році. Очікується, що обсяг створених цифрових даних 2023 році зросте на понад 150% до 2025 року (див рисунок 1.4). [8]

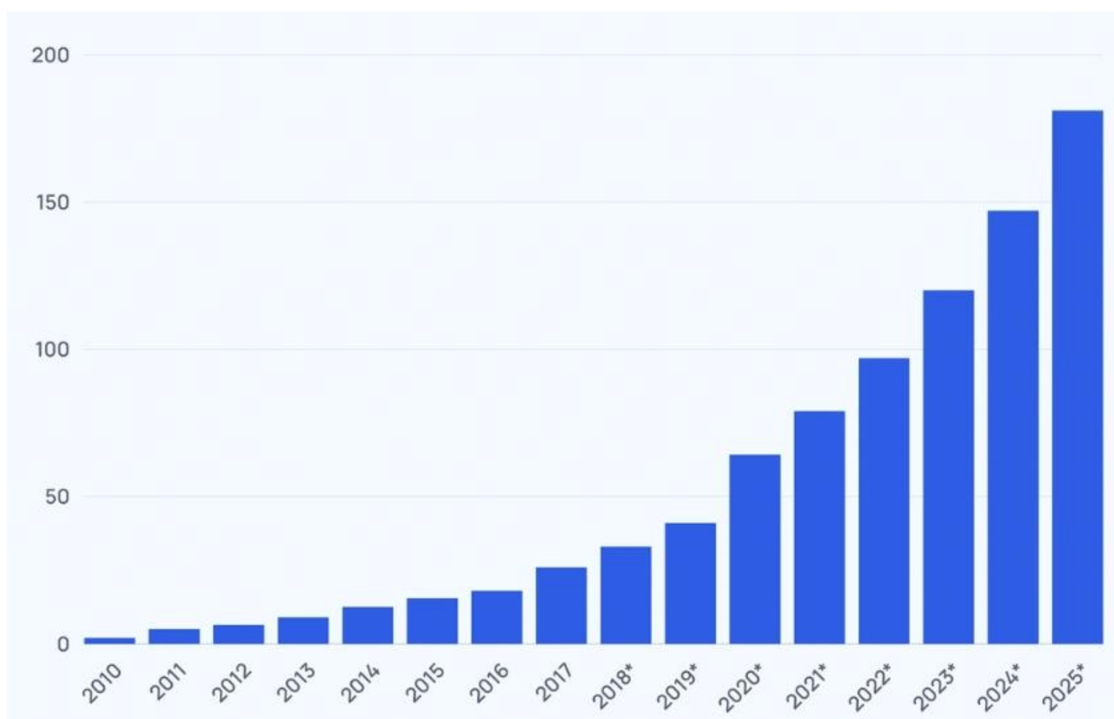


Рисунок 1.4 – Обсяги зростання цифрових даних

Отже, генерація цифрових даних наразі стрімко зростає. З кожним роком ми спостерігаємо як ці цифри досягають нових піків. Цифрові дані кол-центрів не є вийнятком. Обсяги інформації в кол-центрах зростають у зв'язку з рядом факторів, включаючи технологічний прогрес, збільшення обсягів взаємодії з клієнтами та зміни в підходах до обробки даних. З розвитком технологій і зі зростанням кількості каналів зв'язку (телефон, чат, електронна пошта, соціальні мережі), кол-центри стикаються з більшим обсягом вхідних даних від клієнтів, ніж у попередні роки. Запровадження нових сервісів та розширення функціоналу може включати в себе додаткові дані, що обробляються кол-центрами. Наприклад, впровадження системи віртуальних асистентів, розпізнавання мови, чи інших інновацій. Кол-центри використовують аналітику для покращення ефективності і якості обслуговування. Це включає в себе аналіз даних про взаємодію з клієнтами та оптимізацію роботи агентів. Високий попит на точну та вчасну інформацію призводить до зростання обсягів даних для аналізу. Зростання потужності обчислювальних систем та технологій Big Data дозволяє кол-центрам зберігати та обробляти величезні обсяги даних. Це стимулює збільшення обсягів інформації для аналізу та використання. З плином часу

користувачі стають більш вимогливими і їхні запити часто вимагають більш глибокого аналізу та обробки даних. Це призводить до зростання обсягів інформації, яку необхідно обробляти у кол-центрах. З поширенням Інтернету речей (IoT) та підключення все більше пристроїв до мережі, кол-центри отримують додаткові дані від різних джерел, що призводить до збільшення обсягів. Ці фактори разом створюють ситуацію, де кол-центри повинні ефективно працювати з величезними обсягами даних, забезпечуючи при цьому швидку та якісну обробку для задоволення потреб бізнесу та клієнтів.

Сучасний кол-центр, що обслуговує великий обсяг клієнтських запитів та оперативно вирішує аналітичні задачі, стикається з рядом викликів, пов'язаних із складністю та об'ємом даних що обробляються. Аналітики, відповідальні за оптимізацію та вирішення аналітичних задач, повинні вирішувати ряд проблем, щоб забезпечити ефективну роботу кол-центру та забезпечити високу якість обслуговування. Обробка великої кількості даних та складність аналітичних запитів можуть впливати на продуктивність кол-центру. Підходи до оптимізації та вдосконалення роботи з базами даних стають стратегічно важливими для забезпечення швидкого та ефективного реагування на виклики клієнтів. Складність SQL запитів або скриптів (у різних джерелах зазначається по-різному) та їх низька оптимізація можуть стати перешкодою для аналітиків у оперативному та ефективному вирішенні задач. Розуміння оптимальних методів написання запитів та використання інструментів для їх оптимізації визначає успішність аналітичного процесу. Наявність дублювання та різноманітність форматів даних може впливати на точність аналітичних результатів. Вирішення цієї проблеми вимагає стандартизації та уніфікації даних з різних джерел. Забезпечення високого рівня безпеки даних є однією з ключових задач у роботі кол-центру. Аналітики повинні враховувати та вдосконалювати методи захисту конфіденційної інформації від несанкціонованого доступу. Різний рівень знань серед аналітиків може призводити до нерівномірності у якості вирішення завдань. Забезпечення доступу до навчання та розвитку навичок є важливим аспектом управління аналітичним потенціалом. Невідомість та складність взаємодії з іншими системами може стати проблемою для аналітиків. Інтеграція

різних систем та джерел даних сприяє зручній роботі та забезпеченню консистентності результатів аналізу.

Існують значні виклики, пов'язані з оптимізацією SQL запитів для досягнення аналітичних цілей, що спричиняють зниження продуктивності та ускладнення обробки даних. Однією з головних проблем є великий обсяг даних, що призводить до повільних виконань SQL запитів, особливо у випадку складних аналітичних завдань. Складність SQL запитів, вимагаючих об'єднань, фільтрації та групування даних, також впливає на їхню ефективність. Погано оптимізовані SQL запити можуть збільшувати час виконання та завантажувати базу даних, порушуючи продуктивність. Неоптимальні структури даних також можуть впливати на ефективність виконання SQL запитів. Створення оптимізатора нового покоління виявляється доцільним для покращення продуктивності та оптимізації SQL запитів у кол-центрах. Такий оптимізатор повинен використовувати передові методи для зменшення оцінки планів виконання, автоматизації оптимізацій та адаптації до змін у структурах даних. Його впровадження спростить роботу з базою даних та забезпечить стабільну продуктивність, що важливо для ефективної аналітики в реальному часі. Враховуючи автоматизований підхід до оптимізації SQL запитів у нової підсистеми, можна буде говорити про збільшення продуктивності та ефективне вирішення труднощів. Це натомість спричинить зменшення витрат робочих годин аналітиків будь-якого відділу на оптимізацію власних скриптів.

1.4 Вплив оптимізації на продуктивність системи

Вплив оптимізації на продуктивність системи можна простежити безпосередньо виконавши фізичний експеримент. Для цього використаємо базу даних Postgresql та IDE Pg4Admin. Для початку створимо таблицю з користувачами, що міститиме унікальний первинний ключ (id), логін, ім’я та фамілію, а також дату. А також внесемо в неї 10 мільйонів записів (див. рисунок 1.5).

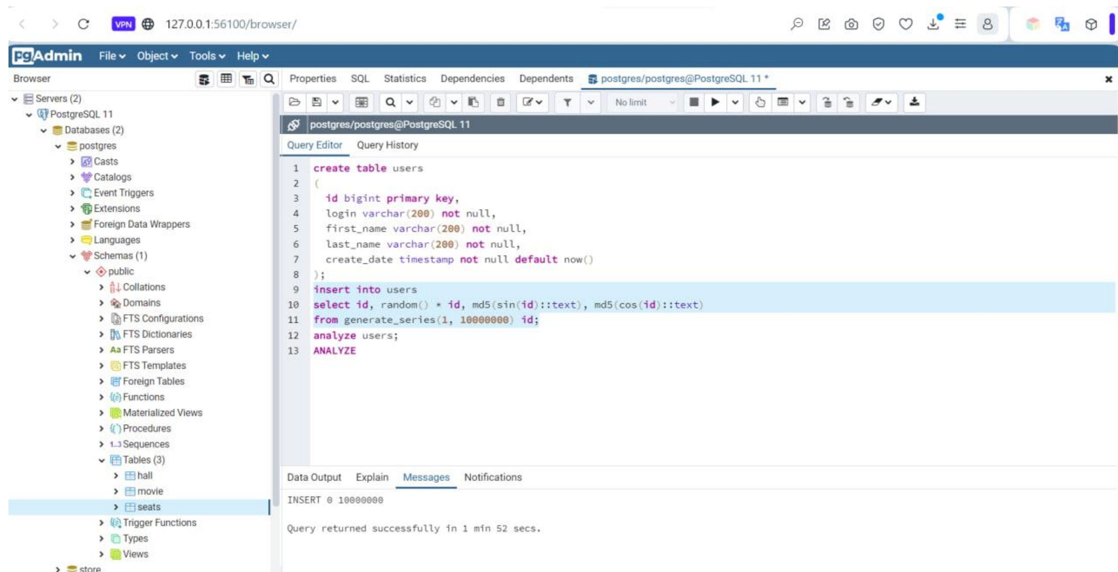


Рисунок 1.5 – Створення таблиці

Виконаємо запит з ключовими словами analyze, explain. Explain показує, як буде працювати запит, але це не реальний план виконання запиту. Щоб це виправити варто додати analyze (див. рисунок 1.6).

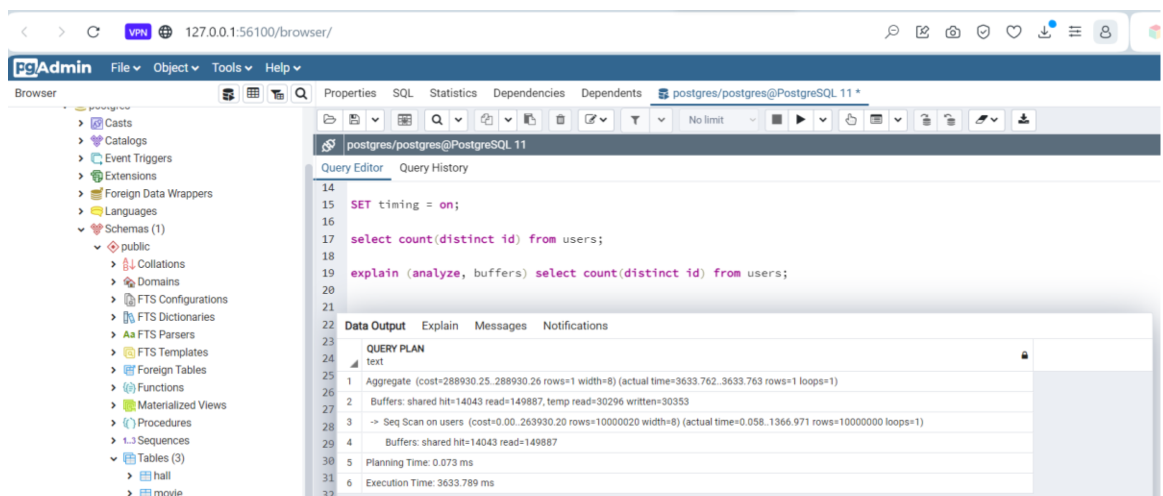


Рисунок 1.6 – Виконання запиту

Якщо скласти всі значення у рядку buffers, ми отримуємо обсяг даних, який буде використаний для виконання даного запиту. Загалом, цей обсяг може становити від 600 до 700 сторінок, що є дуже великим значенням, особливо враховуючи, що розмір однієї сторінки даних у PostgreSQL дорівнює 8 кілобайтам. Використання такого

підходу дозволяє не лише визначити обсяг даних, але й зрозуміти, яким чином PostgreSQL витрачає ресурс під час виконання запиту. Отже, shared hit - це дані, які вже містилися в оперативній пам'яті та були використані з неї. Read - це дані, які довелося зчитувати із диска, оскільки їх не було в оперативній пам'яті. Dirtied - кількість "брудних" даних, які PostgreSQL виявив під час виконання запиту. Одним з цікавих показників є temp_write. Він вказує на те, що оперативної пам'яті для виконання запиту не вистачило, і, отже, в певний момент база даних створила файл на диску обсягом 30 сторінок, який потім зчитувала. Збільшення значення work_mem може допомогти вирішити цю проблему.

Спочатку запит виглядав так count(distinct id). Тепер розглянемо count(*) та його план виконання (див. рисунок 1.7).

```
explain (analyze, buffers) select count(*) from users;
```

```
QUERY PLAN
text
1  Finalize Aggregate  (cost=217013.65..217013.66 rows=1 width=8) (actual time=972.169..980.499 rows=1 loops=1)
2    Buffers: shared hit=14075 read=149855
3    -> Gather  (cost=217013.44..217013.65 rows=2 width=8) (actual time=971.758..980.491 rows=3 loops=1)
4         Workers Planned: 2
5         Workers Launched: 2
6         Buffers: shared hit=14075 read=149855
7         -> Partial Aggregate  (cost=216013.44..216013.45 rows=1 width=8) (actual time=912.535..912.536 rows=1 loops=3)
8              Buffers: shared hit=14075 read=149855
9              -> Parallel Seq Scan on users  (cost=0.00..205596.75 rows=4166675 width=0) (actual time=0.266..745.971 rows=3333333 loops=3)
10                  Buffers: shared hit=14075 read=149855
11 Planning Time: 0.258 ms
12 Execution Time: 980.548 ms
```

Рисунок 1.7 – План виконання count(*)

Раніше був показник у 225 тисяч буферів, а в даному запиті стало 160 тисяч - тобто запит виконався практично в 1.5 разів швидше. За рахунок чого досягається таке прискорення? Тут видно, що запит виконується буквально за одну секунду, тому що одночасно три воркери починають читати дані щодо індексу.

Тепер звернемось до таблиці з користувачами. Також виконаємо створення обліковго запису клієнта кол-центру, робимо посилання на таблицю та створюємо лог акаунта, з посиланням на таблицю унікального ідентифікатора акаунта (див. рисунок 1.8).

```
postgres/postgres@PostgreSQL 11
Query Editor   Query History
21  explain (analyze, buffers) select count(1) from users;
22
23
24
25  create table account(id bigint primary key,
26                      user_id bigint not null references users, name varchar(200),
27                      create_date timestamp not null default now());
28
29  create table account_log (id bigint primary key, account_id bigint not null references account,
30                           login_date timestamp not null default now());
31
32
33  insert into account select id, (random()*10000000) :: bigint, md5(id::text), now() from generate_series(1,1000000) id;
34
35  insert into account_log select id,
36  (random()*10000000) :: bigint, '2023-01-01' :: timestamp + ((random() * 365)::int:: text || ' days') ::interval
37  from generate_series(1,20000000) id;
38
39
40
41
42
43
44  Data Output   Explain   Messages   Notifications
45  Query returned successfully in 2 min 18 secs.
46
47
```

Рисунок 1.8 – Маніпуляції з таблицями

Від бізнесової складової кол-центру на департамент аналітики надходить задача на підрахунок кількості клієнтів, які зайшли в систему в період з 01.04.2023 по 30.04.2023. Для цього створюємо наступну вибірку (див. рисунок 1.9, 1.10).

```
44  explain(buffers,analyze)
45  select count(*), min_login_date::date
46  from
47  (
48    select min(login_date) min_login_date
49    from users u
50      left join account a on u.id = a.user_id
51      left join account_log al on al.account_id = a.id
52    group by u.id
53  ) t
54  where min_login_date between '2023-04-01' and '2023-04-30'
55  group by min_login_date::date
56  order by min_login_date::date
57
58
```

Рисунок 1.9 – Реалізація задачі від кол-центру

А також отримаємо план виконання. Час планування та виконання достатньо великий для даної задачі, так як замовники запланували виконання даного скрипту не одноразово, а за щоденним регламентом. Якщо даний скрипт буде виконуватись щоденно він може значно погіршити продуктивність системи.

```
postgres/postgres@PostgreSQL 11
Query Editor   Query History
Data Output   Explain   Messages   Notifications
QUERY PLAN
text
15            -> Merge Left Join  (cost=0.99..550060.76 rows=10000020 width=16) (actual time=0.032..5334.003 rows=10048024 loops=1)
16                 Merge Cond: (u.id = a.user_id)
17                 Buffers: shared hit=970813 read=223089 written=428
18                 -> Index Only Scan using users_pkey on users u  (cost=0.43..423617.73 rows=10000020 width=8) (actual time=0.013..2697.031 rows=10000000 loops=1)
19                      Heap Fetches: 10000000
20                      Buffers: shared hit=4 read=191251 written=370
21                 -> Materialize  (cost=0.55..88942.97 rows=1000000 width=16) (actual time=0.016..1485.136 rows=1000000 loops=1)
22                      Buffers: shared hit=970809 read=31838 written=58
23                      -> Nested Loop Left Join  (cost=0.55..86442.97 rows=1000000 width=16) (actual time=0.012..1370.956 rows=1000000 loops=1)
24                           Join Filter: (al.account_id = a.id)
25                           Buffers: shared hit=970809 read=31838 written=58
26                           -> Index Scan using ix__account__user_id on account a  (cost=0.42..71434.83 rows=1000000 width=16) (actual time=0.006..1201.071 rows=1000000...
27                                Buffers: shared hit=970808 read=31838 written=58
28                           -> Materialize  (cost=0.13..8.14 rows=1 width=16) (actual time=0.000..0.000 rows=0 loops=1000000)
29                                Buffers: shared hit=1
30                                -> Index Scan using ix__account_log__account_id on account_log al  (cost=0.13..8.14 rows=1 width=16) (actual time=0.002..0.002 rows=0 loops=1)
31                                     Buffers: shared hit=1
32 Planning Time: 0.622 ms
33 Execution Time: 7168.892 ms
```

Рисунок 1.10 – План виконання скрипта для задачі від кол-центру

Для оптимізації запиту насправді просто достатньо прибрати таблицю з користувачами. У першому запиті по ньому відбувається групування, але все необхідне міститься в унікальному ідентифікаторі користувача. І відмова від таблиці Користувачі дасть нам необхідну оптимізацію, а також додамо фільтр, зберігши шукані рядки. Отримаємо наступний скрипт та результат виконання (див. рисунок 1.11 та 1.12).

```
postgres/postgres@PostgreSQL 11
Query Editor   Query History
61 with cte
62 as
63 (
64  select min(login_date :: date) min_login_date, al.account_id
65   from  account_log al
66   where al.login_date between '2023-04-01' and '2023-04-30'
67   group by al.account_id
68 ),
69 cte_ac
70 as
71 (
72   select min_login_date, a.user_id,
73             exists(select * from account_log al
74             join account au on au.id = al.account_id where au.user_id = a.user_id and al.login_date < '2023-04-01' ) is_exists
75   from cte
76     join account a on a. id = cte.account_id
77 )
78 select count(*), min_login_date
79 from
80 (
81   select a.user_id, min(min_login_date) min_login_date
82   from cte_ac a
83   where a.user_id not in (select user_id from cte_ac where is_exists
84   group by a.user_id
85 ) t
86 group by min_login_date
87 order by min login date
```

```
Data Output   Explain   Messages   Notifications
QUERY PLAN
text
1 GroupAggregate  (cost=33.30..33.32 rows=1 width=12) (actual time=0.009..0.011 rows=
```

Рисунок 1.11 – Оптимізований скрипт

```
QUERY PLAN
text
25        SubPlan 3
26          ->  Nested Loop  (cost=0.55..16.59 rows=1 width=8) (never executed)
27                ->  Index Scan using ix__account_log__account_id on account_log al_2  (cost=0.13..8.14 rows=1 width=8) (never executed)
28                      Filter: (login_date < '2023-04-01 00:00:00'::timestamp without time zone)
29                ->  Index Scan using account_pkey on account au_1  (cost=0.42..8.44 rows=1 width=16) (never executed)
30                      Index Cond: (id = al_2.account_id)
31  ->  Sort  (cost=0.08..0.09 rows=1 width=4) (actual time=0.018..0.019 rows=0 loops=1)
32        Sort Key: t.min_login_date
33        Sort Method: quicksort  Memory: 25kB
34        Buffers: shared hit=1
35        ->  Subquery Scan on t  (cost=0.05..0.07 rows=1 width=4) (actual time=0.010..0.011 rows=0 loops=1)
36              Buffers: shared hit=1
37              ->  HashAggregate  (cost=0.05..0.06 rows=1 width=12) (actual time=0.010..0.010 rows=0 loops=1)
38                    Group Key: a.user_id
39                    Buffers: shared hit=1
40                    ->  CTE Scan on cte_ac a  (cost=0.02..0.04 rows=1 width=12) (actual time=0.010..0.010 rows=0 loops=1)
41                          Filter: (NOT (hashed SubPlan 5))
42                          Buffers: shared hit=1
43                          SubPlan 5
44                            ->  CTE Scan on cte_ac  (cost=0.00..0.02 rows=1 width=8) (never executed)
45                                  Filter: is_exists
46  Planning Time: 0.521 ms
47  Execution Time: 0.153 ms
```

Рисунок 1.12 – План запиту оптимізованого скрипта

Отже, по буферах і обсягу даних було отримано незначний запас, зате значно оптимізовано роботу запиту, що якраз і було важливо для аналітиків. У першому запиті Planning Time = 0.6 мс та Execution Time = 7168.1 мс, у останньому Planning Time = 0.5 мс та Execution Time = 0.153 мс.

1.5 Постановка задачі

Для підвищення продуктивності обробки даних SQL скриптів, які використовуються для розв'язання аналітичних задач кол-центру, необхідно розробити підсистему оптимізації SQL скриптів.

Основною задачею є розробка підсистеми для оптимізації SQL скриптів, які використовуються для вирішення аналітичних завдань кол-центру. Метою програми є покращення продуктивності обробки даних та зменшення оцінки плану виконання та кардинальності SQL скриптів. Підсистема повинна виявляти та застосовувати

оптимізаційні заходи до SQL скриптів, що в подальшому допоможе зменшити витрати ресурсів. Аналіз та оптимізація повинні бути спрямовані на забезпечення оперативної відповіді на аналітичні запити в реальному часі. Окрім цього, підсистема повинна брати до уваги різноманітні фактори, такі як зростання обсягів даних та зміни у вимогах до кол-центрів і забезпечувати адаптацію до сучасного бізнес-середовища. Результатом її роботи повинен бути оптимізований SQL скрипт, який ефективно вирішує аналітичні завдання кол-центру та сприяє його конкурентоспроможності. Підсистема повинна працювати в умовах посиленої конкуренції, забезпечуючи якість обслуговування клієнтів та оптимізацію витрат. Враховуючи величезний обсяг даних, які накопичуються кол-центрами, оптимізація SQL скриптів стає стратегічним інструментом для аналізу цих даних та виявлення цінних патернів.

Висновки до розділу 1

Отже, для підвищення продуктивності обробки даних та зменшення кардинальності та оцінки плану виконання SQL скриптів, що створюються для вирішення задач кол-центру за рахунок оптимізації відповідних SQL скриптів проаналізовано роботу кол-центру з застосуванням глибинного аналізу задач, які стоврюються кол-центром на аналітиків. А саме виконано фізичний експеримент та визначено вплив оптимізації на продуктивність системи. Обрана тема є досить актуальною у зв'язку з різноманітними факторами, зокрема, зростанням обсягів даних та змінами у вимогах до сучасних кол-центрів. Нові вимоги до продуктивності, підвищення конкурентоспроможності та необхідність обробки зростаючих обсягів даних для аналізу визначають актуальність дослідження. Сучасний бізнес вимагає від кол-центрів ефективних рішень для оптимізації обробки даних та підтримки конкурентоспроможності. Запити на аналітику стають більш складними, а користувачі очікують оперативних відповідей у реальному часі. В цьому контексті оптимізація SQL скриптів стає ключовим фактором для забезпечення оперативної обробки даних та ефективного управління ресурсами та підтримання конкурентоспроможності кол-центру в динамічному бізнес-середовищі. Сучасний

обсяг генерованих даних, особливо у сфері обслуговування клієнтів, вимагає систематичної обробки та аналізу. Інформація про виклики клієнтів, їхні вимоги та інші аспекти взаємодії стають ключовими для прийняття управлінських рішень та забезпечення високої якості обслуговування клієнтів. Оптимізація SQL скриптів виявляється ефективним інструментом для забезпечення вчасної інформації та реагування на вимоги клієнтів у стрімкому темпі конкурентної гри. В умовах посиленої конкуренції кол-центри, спрямовані на підвищення якості послуг та оптимізацію витрат, розглядають оптимізацію SQL скриптів як стратегічний напрямок. Швидкість обробки SQL запитів стає критичною для надання оперативної інформації та ефективної взаємодії з клієнтами. У сучасному світі кол-центри накопичують величезний обсяг даних про клієнтів та їх вимоги, оптимізація SQL скриптів стає необхідною для ефективного аналізу даних та виявлення певних шаблонів. Вірне використання аналітики визначає стратегічні рішення кол-центру, а швидкий доступ до точної та релевантної інформації стає ключем до успіху в управлінні ресурсами та поліпшенні обслуговування клієнтів.

2 АНАЛІЗ СУЧАСНИХ СИСТЕМ ТА МЕТОДИК ОПТИМІЗАЦІЇ

2.1 Аналіз існуючих методик та їх обмежень

Основною задачею сучасних оптимізаторів в діючих реляційних системах є мінімізація вартості обробки запиту. Вартість послідовної обробки комплекту n запитів – це просто сума вартості обробки кожного з n запитів. Послідовна обробка набору запитів зазвичай підходить для онлайн використання системи баз даних. Однак такий підхід може бути дуже неефективним для пакетної обробки запитів, яка вбудована у звичайні високорівневі алгоритмічні мови програмування. Якщо вартість обробки набору запитів має бути значно нижчою, ніж сума вартості обробки кожного із запитів, то варто звернути увагу на стратегію, яка дозволяє одночасну обробку набору запитів, що насамперед вимагає глобального знання характеристик запитів. Знання розміру відношень та доступних шляхів доступу і характеристик запиту дозволяє визначити не тільки підмножини даного набору для одночасної обробки, але також майже оптимальний набір вторинних індексів і відсортовані копії відношень для обробки заданого набору запитів. Двоетапна стратегія оптимізації набору запитів і стану обробки даних, заснованих на попередній обробці запитів є першим рішенням при вирішенні цієї проблеми. Перший етап відбувається під час компіляції, а саме аналіз характеристик запиту та бази даних. Він складається з двох фаз. Перша визначає оптимальний набір індексів і сортує копії відношень, наступна фаза виводить оптимальну послідовність груп і операторів обробки даних. Другий етап виконує програму, яка містить набір звітів про маніпулювання даними. Під час цього етапу буфер основної пам'яті розділяється на три області: область для зберігання сторінок даних, які містять кортежі відношення; програмна зона для проведення всіх необхідних процедур одночасної обробки заданого набору запитів; вихідну область для зберігання результатів запитів, поки вони не будуть записані на диск. Цей підхід може бути ефективним, коли заданий набір складається із запитів, які можна обробити за результатами обробки інших запитів у наборі.

Наступним методом є оптимальне використання простору буфера оперативної пам'яті. Цей метод, відомий як метод вкладеного блоку, виявляється кращим за метод вкладеного ітерування, а також цей метод перевершує існуючий метод обчислення ділення одного відношення на інше. Метод вкладеного блоку (Nested-Block Nested-Loop Join) є одним із методів оптимізації запитів в базах даних. Він використовується для виконання операції об'єднання (join) двох таблиць. Основна ідея полягає в тому, щоб внутрішній цикл (loop) був вбудований у зовнішній цикл, що дозволяє ефективніше виконувати операцію об'єднання. Спочатку обираються дві таблиці, які необхідно об'єднати. Одна з них визначається як зовнішня таблиця, а інша як внутрішня. Далі внутрішню таблицю поділяють на блоки (chunks) за допомогою певної стратегії. Наприклад, можна використовувати хеш-функцію для розподілу рядків внутрішньої таблиці між різними блоками. Натомість для кожного блоку внутрішньої таблиці виконується зовнішній цикл, де перевіряються умови об'єднання з рядками зовнішньої таблиці. Цей процес повторюється для кожного блоку внутрішньої таблиці. Результати кожного зовнішнього циклу об'єднуються для отримання остаточного результату операції об'єднання. Метод вкладеного блоку може бути оптимізований за допомогою різних технік, таких як кешування, використання індексів та інші стратегії. Метод вкладеного блоку ефективний в тих випадках, коли одна з таблиць помітно менша за іншу або коли можливо використати певні властивості даних для поліпшення продуктивності операції об'єднання. Однак він може бути менш ефективним в ситуаціях, де таблиці мають схожий розмір та не мають виразних властивостей для оптимізації. [9]

Відносно новою методикою оптимізації скриптів є застосування генетичних алгоритмів. Натхненні природнім відбором, генетичні алгоритми є абстракцією біологічної еволюції і, отже, є методом для переходу від однієї популяції хромосом до нової популяції за допомогою своєрідного природного відбору разом із генетично натхненими операторами рекомбінації, мутації та інверсії. [10]

У генетичному алгоритмі рішення називаються індивідами. Після того, як початкова популяція генерується випадковим чином, функція вибору та варіації виконується в циклі до досягнення критерію завершення. Кожен цикл називається

поколінням. Оператор вибору призначений для покращення середньої якості популяції, надаючи індивідам вищої якості вищу ймовірність бути скопійованими в наступне покоління. Якість індивіда вимірюється функцією придатності. Функція придатності визначає оптимальність рішення (тобто хромосоми) в генетичному алгоритмі, щоб конкретну хромосому можна було порівняти з усіма іншими хромосомами. Схрещування та мутація є важливими операторами генетичного алгоритму. Схрещування вибирає гени від батьківських хромосом і створює нового нащадка. Після виконання схрещування відбувається мутація.

Коли генетичні алгоритми застосовуються до оптимізації запитів ініціальна популяція генерується випадково. Для кожного покоління виконуються генетичні операції, і таким чином популяція еволюціонує, зазвичай зменшуючи середні витрати своїх індивідів. Коли отримано найбільш оптимальний план, він передається до бази даних для виконання запиту. Оптимізацію запитів можна провести двома підходами алгебраїчні маніпуляції чи трансформації. Підхід, заснований на алебраїчних виразаї, полягає в тому, щоб спочатку представити кожний реляційний запит як вираз реляційної алгебри, а потім трансформувати його в еквівалентний, але ефективніший вираз реляційної алгебри. Трансформація керується евристичними правилами оптимізації. Основна ідея підходу, заснованого на оцінці вартості, полягає в тому, що для кожного запиту перераховуються всі можливі плани виконання. Основна проблема в оптимізації запитів полягає в тому, що простір пошуку ускладнений, а генетичні алгоритми теоретично та емпірично доведені надійно шукати рішення в складних просторах. Ці алгоритми є обчислювально простим, але потужними рішень у даній області застосування. [11]

Також одним з сучасних методів є використання динамічного програмування - це метод оптимізації, який використовується для розв'язання проблем з оптимізацією шляхом розбиття їх на менші підзадачі та збереження оптимальних рішень цих підзадач. У контексті оптимізації планування запитів в базах даних, динамічне програмування може бути використане для знаходження оптимальних планів виконання запитів, зменшуючи кількість дублювання обчислень і забезпечуючи ефективні рішення. Загалом, для вирішення задачі потрібно вирішити різні частини

проблеми, а потім об'єднати рішення підзадач для досягнення загального рішення. Часто багато з цих підзадач насправді є однаковими. Підхід динамічного програмування спрямований на вирішення кожної підзадачі лише один раз, тим самим зменшуючи кількість обчислень. Якщо рішення для заданої підзадачі вже було обчислено, воно зберігається або запам'ятовується програмно. Наступного разу, коли потрібно те саме рішення, його просто шукають. Цей підхід особливо корисний, коли кількість повторюваних підзадач зростає експоненційно як функція величини введення. Спочатку проблема виявляє оптимальну підструктуру, якщо оптимальне рішення задачі містить в собі оптимальні рішення підзадач. Відбувається розподіл проблеми на підзадачі. Алгоритми динамічного програмування зазвичай використовують перевагу підзадач, що перетинаються, а потім зберігають рішення в таблиці, де його можна витягти, коли воно потрібно. [12]

Отже, незважаючи на те, що сучасні методики оптимізації SQL скриптів можуть значно поліпшити продуктивність запитів та взаємодії з базою даних є кілька недоліків та обмежень, які можуть виникнути. Оптимізація SQL скриптів може бути трудомісткою та вимагати глибокого розуміння структури бази даних, індексації та алгоритмів обробки запитів. Також деякі оптимізації можуть бути ефективними тільки для конкретних типів баз даних або систем управління базами даних (СУБД). Оптимізації, які ефективні на одній базі даних, можуть не бути такими ефективними на інших. Оптимальні стратегії можуть змінюватися з часом, особливо коли обсяги даних або характеристики запитів змінюються. Це може потребувати постійного моніторингу та адаптації оптимізацій. Невірно застосовані оптимізації можуть призвести до втрати стабільності системи або навіть до втрати даних. Наприклад, неправильно вибраний індекс може призвести до збільшення часу вставки або оновлення даних. Іноді в оптимізації можуть зустрічатися ситуації, коли вони не є ефективними або навіть погіршують продуктивність. Наприклад, використання індексу для дуже маленької таблиці може призвести до додаткового навантаження без суттєвого покращення продуктивності. Деякі оптимізаційні заходи можуть вимагати додаткових системних ресурсів, таких як пам'ять або обчислювальна потужність. Це може впливати на загальну продуктивність системи або обмежувати їх

масштабованість. Для ефективної оптимізації потрібно мати глибокі знання в області роботи з базами даних, їх внутрішньою структурою та механізмами оптимізації. Важливо збалансувати переваги від оптимізації із зазначеними недоліками та розглядати кожен випадок індивідуально для досягнення найкращої продуктивності в конкретному контексті вирішуваної задачі. [13]

2.2 Порівняльний аналіз сучасних оптимізаторів

В світі реляційних баз даних, де швидкість та ефективність є критичними чинниками, оптимізація SQL запитів відіграє важливу роль у забезпеченні оптимальної продуктивності систем. З розвитком технологій та розширенням функціональних можливостей баз даних на ринку з'являється все більше інструментів для автоматизації та покращення оптимізації SQL запитів. У даному розділі проведено глибокий аналіз та порівняння сучасних оптимізаторів SQL запитів. З’ясовано їх функції, можливості та обмеження, визначено сценарії, в яких кожен оптимізатор особливо ефективний у застосуванні. В сучасному інформаційному середовищі, де обсяги даних постійно зростають, а вимоги до продуктивності стають все вищими, вибір оптимального оптимізатора може мати вирішальне значення для успішної роботи з базами даних. Нові оптимізатори стають лідерами в цій конкурентній області та відповідають на сучасні виклики у сфері обробки та оптимізації SQL запитів.

Зазвичай оптимізатори звикли розділяти на дві категорії. Оптимізатор на основі правил (RBO) відрізняється від оптимізатора на основі вартості (CBO). Наприклад, маємо SQL запит:

```
SELECT *
FROM Table1
WHERE column2 = 55;
```

Припустимо, що column2 це індексований неунікальний стовпець. Оптимізатор на основі правил знаходить column2 у системному каталозі та встановлює, що він індексований, але не унікально. Далі RBO поєднує ці дані з інформацією після знака

рівності. Зазвичай припускається, що умови пошуку "= <літерал>" виведуть 5% усіх рядків. Це вузький пошук, і, як правило, швидше виконувати його за допомогою бінарного дерева, ніж просканувати всі рядки у таблиці. Таким чином, оптимізатор на основі правил розробляє певний план, а саме знайти відповідні рядки за допомогою індексу на column2. Він використовує факт наявності індексу та деяке фіксоване припущення. Оптимізатор на основі вартості може спланувати подальшу оптимізацію. Припустимо, що системний каталог містить три додаткові факти: (1) що у Table1 є 100 рядків, (2) що у Table1 є дві сторінки, і (3) що значення 55 зустрічається 60 разів у індексі для column2. Ці факти змінюють все. Операція рівності відповідатиме 60% рядків, тож це не є вузьким пошуком. Всю таблицю можна просканувати двічі, тоді як пошук за індексом вимагатиме трьох прочитаних сторінок (одне для пошуку в індексі, два інших для отримання даних пізніше). Таким чином, оптимізатор на основі вартості розробляє інший план: знайти відповідні рядки за допомогою перегляду таблиці. Оптимізатор на основі вартості використовує волатильні дані (значення рядків та стовпців, які були вставлені) та перевизначення (що вміст важливіший, ніж фіксовані припущення). Іншими словами, оптимізатор на основі вартості є оптимізатором на основі правил, який має додаткову, волатильну інформацію (непередбачувані та часто змінювані дані), щоб перевизначити фіксовані припущення, які інакше б керували його рішеннями. Більшість компаній стверджують, що їхні СУБД мають оптимізатори на основі вартості. Але важливим є те чи оптимізатор правильно оцінює вартість, і як він діє при безпосередньому оцінюванні. [14]

Насамперед визначимось з основними критеріями, яким варто приділити особливу увагу при аналізі сучасних оптимізаторів.

Досконалість оптимізації запитів складається з якості та ефективності планів виконання, які генерує оптимізатор та здатність вибирати оптимальні індекси для швидкого доступу до даних, а також обробки та оптимізації різних типів JOIN-операцій. Наступним критерієм є підтримка стандартів SQL, а саме ступінь відповідності оптимізатора стандартам SQL, таким як ANSI SQL. Третім не менш важливим критерієм є можливість оптимізації пам'яті, тобто ефективне використання

пам'яті для обробки та збереження проміжних результатів запитів. Також оптимізація для конкретних типів запитів, а саме здатність оптимізувати різні типи запитів, такі як SELECT, INSERT, UPDATE, DELETE і підтримка паралельного виконання, що пояснює можливості оптимізатора використовувати паралельну обробку для підвищення продуктивності. При порівнянні слід звертати увагу на здатність адаптуватися до змін у навантаженні та характеристиках даних для підтримки динамічної оптимізації і вплив оптимізатора на системні ресурси, такі як процесор, пам'ять та диск. Останнім є можливості профілювання та аналізу виконання, тобто інструменти для моніторингу та аналізу продуктивності SQL запитів.

Отже, при аналізі існуючих рішень варто звертати увагу на такі ключові показники як збільшення продуктивності системи, зменшення споживання пам’яті та процесорного часу, зменшення використання простору зберігання даних.

Опрацювавши дані з інтернет-джерел можна виокремити кілька популярних програм та інструментів для оптимізації запитів SQL, які частіше за інші використовуються у світі аналітики даних:

- оптимізатор MySQL Query Optimizer. Вбудований оптимізатор запитів для MySQL баз даних. Він автоматично аналізує та оптимізує SQL запити, використовуючи статистику таблиць, наявні індекси та інші фактори;
- оптимізатор SQL Server Query Optimizer. Оптимізатор запитів для Microsoft SQL Server. Він використовує різні техніки, такі як статистика таблиць, план виконання запиту та інші фактори для покращення продуктивності запитів;
- оптимізатор Oracle SQL Optimizer. Інструмент розроблений для оптимізації запитів до баз даних Oracle. Він пропонує аналіз та рекомендації щодо використання індексів, підключення таблиць та інших оптимізаційних можливостей;
- оптимізатор PostgreSQL Query Optimizer. Оптимізатор запитів для PostgreSQL баз даних. Він використовує різні методи, такі як генетичні алгоритми, план виконання та інші стратегії, щоб знайти оптимальний спосіб виконання запиту;

– інструменти від сторонніх постачальників. Крім вбудованих оптимізаторів, існують різні комерційні та безкоштовні інструменти від сторонніх постачальників, які надають додаткові можливості для аналізу та оптимізації запитів. Наприклад, SQL Sentry Plan Explorer, Quest Toad, dbForge Studio, Navicat та багато інших.

Ці інструменти надають зручний інтерфейс для аналізу запитів, візуалізації планів виконання, а також можливості експериментувати з різними оптимізаційними стратегіями з метою підвищення продуктивності запитів.

Популярні додатки для оптимізації запитів до баз даних можуть надавати різні показники та статистику для оцінки продуктивності та ефективності запитів.

Деякі загальні показники, які часто залучені в ці додатки, включають:

– час виконання запиту. Показує загальний час, який потрібно для виконання запиту до бази даних. Чим менше час виконання, тим ефективніше запит;
– план виконання запиту. Візуалізоване представлення оптимізованого плану виконання запиту. Він показує порядок виконання операцій, використані індекси, методи з'єднання таблиць та інші деталі;
– кількість прочитаних/записаних рядків. Показує, скільки рядків було прочитано або записано під час виконання запиту. Менша кількість операцій зчитування/запису може свідчити про більш ефективне виконання запиту;
– використання індексів. Цей показник показує, які індекси були використані під час виконання запиту. Використання підходящих індексів може покращити продуктивність запиту;
– обсяг пам'яті та ресурсів. Деякі додатки можуть показувати, скільки пам'яті або інших ресурсів (наприклад, CPU) було використано під час виконання запиту. Це може допомогти виявити проблеми з використанням ресурсів та встановити оптимальні значення;
– статистика запиту. Деякі додатки надають детальну статистику щодо часу виконання певних операцій у запиті, використання індексів, кількості

зчитаних/записаних рядків та інших параметрів. Це може допомогти виявити слабкі місця та виконати точкову оптимізацію.

Варто зазначити, що конкретні показники можуть варіюватись в залежності від обраного додатку для оптимізації запитів та його функціональності.

Хоча сучасні оптимізатори запитів до баз даних мають значний потенціал і надають багато корисних функцій, все ж існують певні аспекти, в яких вони можуть не відповідати очікуванням. А саме, комплексність запитів, тобто оптимізатори запитів мають обмежену здатність до ефективної оптимізації складних або дуже складних запитів. При наявності багатьох таблиць, складних умов, підзапитів і функцій можуть виникати виклики для точної та оптимальної оптимізації. Застарілі статистичні дані, тобто оптимізатори базуються на статистичних даних про дані таблиць, таких як розподіл значень, кількість рядків тощо. Проблема виникає, коли ці статистичні дані застарілі або неправильні, що може призвести до генерування неоптимальних планів виконання запитів. Брак можливостей адаптивної оптимізації, а саме багато оптимізаторів запитів не мають вбудованої підтримки адаптивної оптимізації. Це означає, що вони не можуть автоматично адаптуватись до змін у навантаженні або конкретних умов виконання запиту. Як результат, вони можуть не завжди найкращим чином пристосовуватись до змінних умов та динамічних вимог. Відсутність підтримки для специфічних баз даних або мов запитів, тобто оптимізатори призначені для конкретних типів баз даних або мов запитів, що обмежує їхню універсальність. Це є перешкодою, якщо користувач працює з різними базами даних або мовами запитів. Обмежена підтримка для аналізу структури даних, а саме оптимізатори запитів часто фокусуються на оптимізації самого запиту, але не надають глибокого аналізу структури даних. Це означає, що вони припускають можливість покращити продуктивність шляхом оптимізації схеми бази даних або перегляду індексів.

Ці аспекти необхідно враховувати, тому слід вдатись до написання більш спеціалізованих інструментів для досягнення найкращої продуктивності бази даних. Щоб узагальнити усі переваги та недоліки існуючих рішень перейдемо до таблиці 2.1

та більш детальної таблиці 2.2, яка зможе на конкретних прикладах продемонструвати головні відмінності сучасних оптимізаторів.

Таблиця 2.1 – Порівняльна таблиця існуючих оптимізаторів

Програмне забезпечення	Переваги	Недоліки
Планувальник SQL Server	Ефективна оптимізація запитів	Обмежена підтримка розподілених систем
	Підтримка статистики та індексів	Погана сумісність з деякими старими версіями СУБД
	Підтримка оптимізації на рівні пакету	Вимагає значних ресурсів для роботи
	Можливість зміни стратегії оптимізації	
PostgreSQL Optimizer	Розширена підтримка виразів та функцій	Може потребувати додаткового налаштування
	Підтримка кількох методів з'єднання таблиць	Може бути складним у використанні для початківців
	Підтримка аналізу пріорітету запитів	Не підтримує автоматичне створення індексів
	Можливість налаштування параметрів оптимізатора	
Oracle Optimizer	Розширена підтримка аналізу запитів	Ліцензійна плата за використання, тобто немає безкоштовної версії для використання

Таблиця 2.2 – Порівняння інших оптимізаторів

Параметр	MySQL Query Optimizer	SQL Server Query Optimizer	Oracle SQL Optimizer	PostgreSQL Query Optimizer	Query Optimizer (загальний)
Тип	Вартісний та правиловий	Вартісний	Вартісний	Вартісний	Вартісний та правиловий
Якість оптимізації запитів	Залежить від версії та налаштувань	Висока	Висока	Висока	Різні рівні якості в залежності від СУБД
Підтримка стандартів SQL	Так	Так	Так	Так	Так
Можливості оптимізації пам'яті	Так	Так	Так	Так	Так
Оптимізація	Так	Так	Так	Так	Так
Паралельне виконання	Залежить від версії та налаштувань	Так	Так	Так	Залежить від СУБД
Автоматична регуляція	Залежить від версії	Так	Так	Так	Так
Взаємодія з системними ресурсами	Залежить від версії та налаштувань	Так	Так	Так	Залежить від СУБД
Можливості профілювання	Так	Так	Так	Так	Залежить від СУБД

Отже, існуючі оптимізатори, такі як MySQL Query Optimizer, SQL Server Query Optimizer, Oracle SQL Optimizer та PostgreSQL Query Optimizer, володіють певними сильними сторонами, проте їх недоліки можуть включати обмежену ефективність в адаптації до змін у структурі даних та обсягу інформації, обчислювальні витрати під час генерації оптимальних планів виконання, недостатню підтримку для паралельного та розподіленого виконання, а також можливість виявлення слабких місць при оптимізації конкретних типів запитів. Створення нового оптимізатора може бути обгрунтовано необхідністю вирішення цих недоліків та надання оптимального та адаптивного підходу до оптимізації запитів. Новий оптимізатор може зосередити свої ресурси на забезпеченні ефективної реакції на зміни в даних, використанні обчислювальних ресурсів для генерації швидких та оптимальних планів виконання, а також максимізації використання паралельного та розподіленого виконання для забезпечення високої продуктивності в різноманітних середовищах роботи з даними.

2.3 Вибір технологій проектування. Переваги RUST.

RUST — сучасна мультипарадигмальна мова програмування загального призначення.

Термін "мультпарадигмальна мова програмування" вказує на те, що мова підтримує та дозволяє використання кількох парадигм програмування. Парадигми програмування це основні стилі та методології, які визначають способи створення програмного коду. Існують різні парадигми, такі як імперативна, функціональна, об'єктно-орієнтована, логічна, паралельна, тощо. У випадку Rust це означає що вона дозволяє програмістам використовувати різні стилі програмування в залежності від конкретного завдання чи власних вподобань. Rust охоплює кілька парадигм програмування, зокрема, імперативна парадигма передбачає реалізацію завдань через послідовні команди, які змінюють стан програми, процедурна парадигма передбачає організацію коду у функції, які можуть викликатися з інших частин програми. Також об'єктно-орієнтована парадигма (ООП) передбачає використання об'єктів і класів для

структуризації коду та роботи з об'єктами. Функціональна парадигма охоплює використання функцій, які не мають стану і здатні до композиції для розв'язання задач. Також існує модель акторів. Використання концепції акторів існує для створення паралельних програм і обробки великої кількості невеликих обчислень. У Rust передбачено використання засобів для створення загальних та параметризованих структур коду, а також для маніпулювання програмним кодом під час компіляції. Такий різноманітний набір парадигм дозволяє програмістам використовувати Rust для вирішення різноманітних завдань в контексті окремих стилів програмування.

Мова відзначається строгою типізацією та акцентом на безпечній роботі з пам'яттю і забезпеченні високої швидкодії виконання завдань, здатність створювати мільярди процесів та їх підпроцесів. У своїй структурі, Rust нагадує C++, але відрізняється в деяких аспектах синтаксису та семантики, а також спрямований на блочну організацію структури коду, що дозволяє реалізовувати завдання у вигляді певних підпрограм. Автоматичне керування пам'яттю звільняє розробника від необхідності маніпулювання операндами і захищає від проблем, пов'язаних з низькорівневою роботою з пам'яттю, такими як доступ до вільної пам'яті, переіменування нульових вказівників, перевищення меж буфера та інше. Rust підтримує поєднання імперативних, процедурних і об'єктно-орієнтованих методів, а також парадигми функціонального програмування і моделі акторів, а також узагальненого і мета-програмування в статичних і динамічних стилях.

В теперішній час Rust представляє собою універсальну мову програмування, придатну для вирішення завдань будь-якої складності. Незалежно від того, чи йдеться про написання консольної утиліти, чи розробку прошивки для вбудованої електроніки, чи реалізацію високонавантаженого бекенду, який обробляє сотні бізнес-сценаріїв. Серед ключових переваг Rust слід відзначити те, що Rust виник як відповідь на вимоги надійності веб-браузера Servo, але на даний момент виходить за рамки вхідної мети і в основному застосовується для розробки вбудованих систем, операційних систем та високонавантажених сервісів. У порівнянні з C та Go, Rust вирізняється відсутністю багатьох їх недоліків і продемонструвавши вищу ефективність в обробці завдань, які зазвичай виконуються Go, C чи C++. Зокрема,

швидкість C++ досягається завдяки концепції «Zero-cost abstraction» та нативному виконанню, а надійність підтримується через простоту та перевірки володіння ресурсами на етапі компіляції. Вбудована сучасна система управління пам'яттю Rust, що включає в себе Ownership, Lifetime та Borrow Checker, дозволяє розробникам уникнути непередбачуваних проблем, пов'язаних з некоректною роботою з пам'яттю. [15]

Ownership, Lifetime і Borrow Checker є ключовими концепціями в мові програмування Rust і використовуються для управління пам'яттю та уникненням проблем. (див. таблицю 2.3)

Таблиця 2.3 – Основні концепції RUST

Концепція	Основна ідея	Принципи
Ownership (власність)	Визначає, як об'єкти взаємодіють з пам'яттю та як передаються між функціями	Кожен об'єкт має свого "власника". У власника є право визначати коли пам'ять звільнити. Якщо власник закінчує своє існування, пам'ять автоматично звільняється. Передача об'єкта іншому власнику може здійснюватися через механізм, але не через просте копіювання пам'яті
Lifetime (час життя)	Визначає, скільки часу об'єкт або посилання на об'єкт залишається дійсними, тобто може спрогнозувати і зорієнтувати з приводу часу життя об’єкта	Всі посилання мають свій власний "час життя", який вказує, до якого моменту вони гарантовано залишаються дійсними.

Концепція	Основна ідея	Принципи
Borrow Checker (перевірка посилань)	Гарантує відсутність помилок, пов'язаних з використанням пам'яті	Вказівки на пам'ять (посилання) перевіряються на валідність під час компіляції. Головна мета це уникнення конфліктів при доступі до об'єктів з багатьох частин програми. Забезпечується безпека використання пам'яті без необхідності великого обсягу вартісних вказівок в коді

Отже, ці концепції в сукупності створюють систему, яка робить Rust дуже потужною і безпечною мовою програмування, здатною уникати багатьох типових помилок, пов'язаних з роботою пам'яті. Rust представляє собою вже готову технологію, яка успішно використовується розробниками. Це мова програмування системного рівня, яка надає контроль над деталями низького рівня. У Rust є можливість вибирати чи зберігати дані в стеку (для статичного виділення пам'яті), чи в купі (для динамічного виділення пам'яті). Важливо також зазначити принцип RAII (Resource Acquisition Is Initialization), який, хоча базово пов'язаний з C++, також притаманний Rust. При виході об'єкта за межі області видимості викликається його деструктор, відбувається звільнення призначених йому ресурсів. У мові програмування Rust не потрібно вручну керувати ресурсами, тобто програмісти повністю захищені від багів, які виникають коли є певний витік даних, що сприяє більш ефективному використанню пам'яті. Оскільки Rust не використовує постійного збирача сміття (наприклад, Garbage collector), як це часто буває в високорівневих мовах, таких як Java, проекти на Rust можуть виступати у ролі бібліотек для інших мов програмування завдяки інтерфейсам зовнішніх функцій. Це досить актуально для

існуючих проектів, де необхідно забезпечити високу ефективність при одночасному збереженні безпеки пам'яті. У подібних ситуаціях програма на Rust має здатність заміщувати індивідуальні компоненти програмного забезпечення, де продуктивність є надважливою, не вимагаючи повної переробки усього продукту. Мова програмування Rust, яка володіє можливістю прямого доступу до обладнання та пам'яті, вважається оптимальним вибором для створення вбудованих систем через свій низький рівень абстракції. Крім того, Mozilla, де і почалася розробка цієї мови, використовує Rust у своїх браузерних двигунах. Основними перевагами використання Rust є висока продуктивність і безпека пам'яті, що робить його привабливим для науковців, які часто використовують для обробки великих об'ємів даних. Rust демонструє вражаючу швидкість, тобто є ідеальним вибором для машинного навчання, де потрібно швидко опрацьовувати великі об'єми даних. Загалом, Rust визначається високою продуктивністю при забезпеченні безпеки пам'яті, підтримкою паралельного програмування, зростаючою кількістю пакетів у репозиторії crates.io та активним співтовариством.

Використання мови програмування Rust для написання оптимізатора має кілька обґрунтованих причин, по-перше це продуктивність та швидкість. Rust відомий своєю високою продуктивністю і швидкістю виконання запитів. Вбудовані механізми контролю пам'яті дозволяють писати ефективний код без зайвого переносу витрат часу на збирання сміття. По-друге, безпека пам'яті. Забезпечення безпеки пам'яті є ключовим аспектом Rust. Одна з основних переваг це відсутність витоку пам'яті та руйнації пам'яті, що робить Rust відмінним вибором для написання оптимізаторів, де важлива стійкість та надійність роботи. По-третє, Rust надає прямий доступ до обладнання та низькорівневий контроль, що особливо корисно при розробці систем, які можуть взаємодіяти з конкретними аспектами архітектури апаратного забезпечення. По-четверте, підтримує паралельне програмування і безпечно дозволяє взаємодіяти з багатьма потоками. Це важливо для програмних систем, які можуть використовувати паралельні обчислення для прискорення обробки даних. Також дана мова має ряд сучасних функцій та конструкцій мови, які роблять код більш легким у розумінні. Це дозволяє чітко виражати ідеї та легко взаємодіяти з кодом оптимізатора.

Не менш важливим є те, що Rust має активну та розширену екосистему бібліотек і пакетів, яка полегшує процес розробки підсистеми чи системи. Також можна використовувати існуючі бібліотеки та інструменти для спрощення завдань. [16]

Загалом, Rust вигідно виділяється через свою комбінацію продуктивності, безпеки, контролю над пам'яттю та простоти використання. Це робить його привабливим вибором для написання високоефективних оптимізаторів, які вимагають від програмістів низькорівневого контролю надійності.

2.4 Аналіз можливостей RUST та EGG LIBRARY

Egg - це фреймворк для оптимізації програм, написаний на Rust. Його основна технологія базується на методі, відомому як метод насичення рівності. Ідея полягає в тому, щоб поступово переписувати вирази, знаходячи всі еквівалентні форми, а потім визначити оптимальне рішення серед них. Під час цього процесу Egg використовує структуру даних e-graph для ефективного запиту та утримання класів еквівалентності під час виконання програми, зменшуючи витрати часу та простору для оптимізації програми. У контексті математики та теорії виразів "насичення рівня" (saturation) використовується для позначення процесу, коли множина об'єктів або умов стає повною чи насиченою, тобто вже не може бути розширена за певними правилами чи умовами. Наприклад, нехай S – множина диз'юнктів. Припустимо, $S_0 = S$ і для $n \in \mathbb{N}$ створюємо множину S_n за наступним правилом : $S_n = \{R(C_1, C_2) | \, C_1 \in \bigcup^{n-1} S_i \, , C_2 \in S_{n-1}\}$. Множина S_n називається множиною диз'юнктів n-го рівня. Нехай $S = \{P \vee Q, P \vee \neg Q, \neg P \vee Q, \neg P \vee \neg Q\}$.

S_0	1.	$P \vee Q$
	2.	$P \vee \neg Q$
	3.	$\neg P \vee Q$
	4.	$\neg P \vee \neg Q$

S_1	5.	P	$R(1, 2)$
	6.	Q	$R(1, 3)$
	7.	$Q \vee \neg Q$	$R(1, 4)$
	8.	$P \vee \neg P$	$R(1, 4)$
	9.	$Q \vee \neg Q$	$R(2, 3)$
	10.	$P \vee \neg P$	$R(2, 3)$
	11.	$\neg Q$	$R(2, 4)$
	12.	$\neg P$	$R(3, 4)$
S_2	13 – 16.	$P \vee Q$	$R(1, 7) = R(1, 8) = R(1, 9) = R(1, 10)$
	17.	P	$R(1, 11)$
	18.	Q	$R(1, 12)$
	19.	P	$R(2, 6)$
	20 – 23.	$P \vee \neg Q$	$R(2, 7) = R(2, 8) = R(2, 9) = R(2, 10)$
	24.	$\neg Q$	$R(2, 12)$
	25.	Q	$R(3, 5)$
	26 – 29.	$\neg P \vee Q$	$R(3, 7) = R(3, 8) = R(3, 9) = R(3, 10)$
	30.	$\neg P$	$R(3, 11)$
	31.	$\neg Q$	$R(4, 5)$
	32.	$\neg P$	$R(4, 6)$
	33 – 36.	$\neg P \vee \neg Q$	$R(4, 7) = R(4, 8) = R(4, 9) = R(4, 10)$
	37.	P	$R(5, 8)$
	38.	P	$R(5, 10)$
	39.	X	$R(5, 12)$

Як бачимо, x отримано лише на 39-му кроці. Крім того, диз’юнкти 7 – 10 – тавтології, або тавтологічні диз’юнкти, а також диз’юнкти 13 – 38 повторюють вище написані в S_0 та S_1. [17]

У випадку стратегії насичення рівня, використовуваної в оптимізації програм, основна ідея полягає в пошуку всіх можливих еквівалентних форм виразів та виборі оптимального рішення серед них. Процес насичення рівня включає в себе поетапне додавання нових об'єктів та умов до множини існуючих, а також перевірку еквівалентності та спроби визначити оптимальні представлення. Щодо зв'язку з методом, відомим як "Насичення Рівності" (Equality Saturation), цей метод використовує стратегію насичення рівня для оптимізації програм. У випадку бібліотеки Egg, яка використовує цей метод, стратегія насичення рівня впроваджується через структуру даних, відому як ефективний граф (E-graph), що дозволяє динамічно переписувати та оптимізовувати вирази, визначаючи їх еквівалентні форми.

Метод, відомий як метод насичення рівності є підходом до оптимізації програм, який використовується в бібліотеці Egg. Задача оптимізації полягає в тому, щоб знайти оптимальний вираз або представлення для певного обчислення. Проте існує багато еквівалентних форм виразів, які можуть виконувати ту ж саму функцію. Equality Saturation використовує метод поетапного переписування виразів. Процес полягає в тому, щоб поступово переписувати вирази, шукаючи всі можливі еквівалентні форми. Для ефективної реалізації Equality Saturation використовується структура даних, відома як e-graph (ефективний граф). E-graph дозволяє ефективно визначати та утримувати класи еквівалентності для вузлів виразів. [18]

В теорії графів визначено, що граф є абстрактним комбінаторним об'єктом, який складається із двох множин: V - множина вершин, і E - множина з'єднань між цими вершинами. (див. рисунок 2.1).

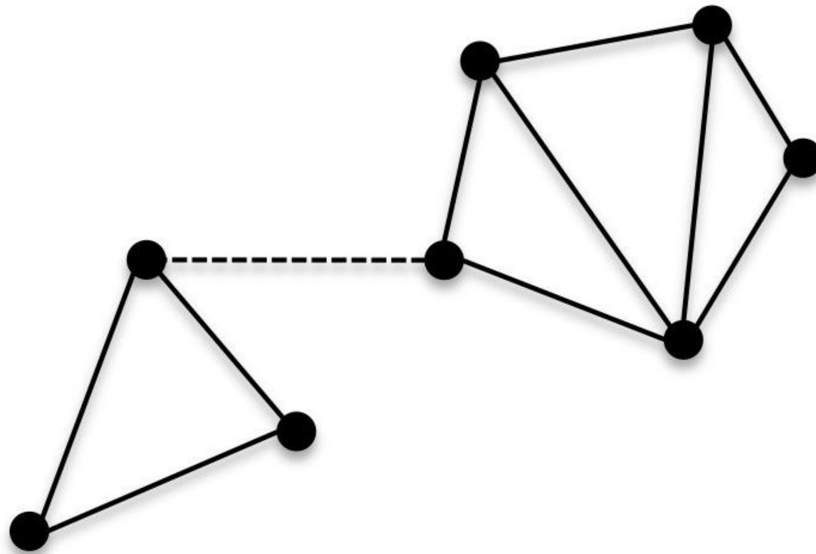


Рисунок 2.1 - Зв'язний граф

Ефективний граф (E-graph) (див. рисунок 2.2) відрізняється від звичайного графа (див. рисунок 2.1) тим, що він спеціально призначений для представлення та оптимізації виразів чи структурних даних в контексті оптимізації програм.

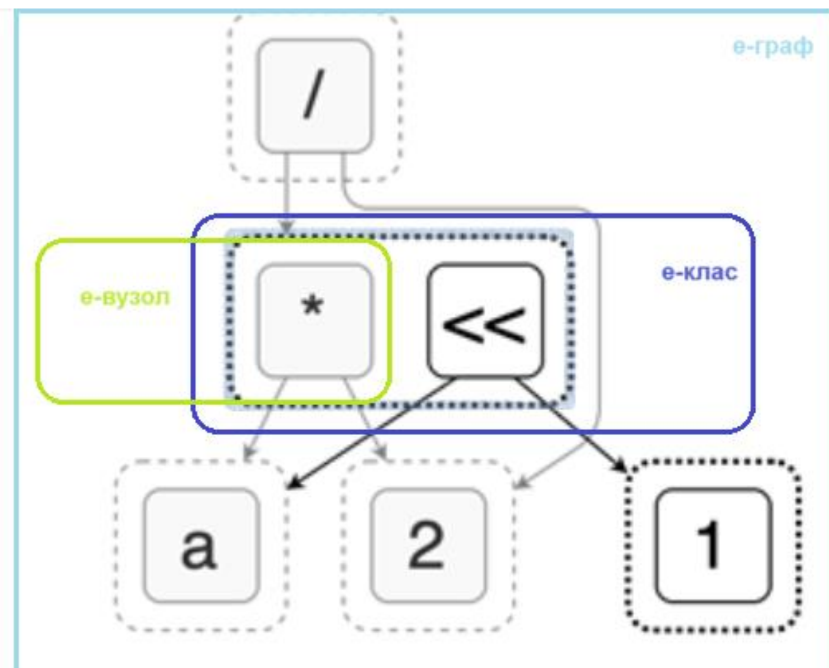


Рисунок 2.2 – Складові е-графу

У ефективному графі кілька вузлів може утворювати еквівалентний клас, що означає, що ці вузли можуть бути замінені один одним без втрати семантики. Це дає можливість ефективно оптимізувати вирази, знаходячи оптимальні варіанти представлення. Ефективний граф дозволяє динамічно вставляти нові вузли та об'єднувати класи, що дозволяє гнучко оптимізувати та переписувати вирази під час виконання програми. Використання ефективного графа часто пов'язане з оптимізацією виразів чи структурних даних, замість того, щоб створювати нові структури. Ефективний граф може визначати еквівалентність та замінюваність вузлів для результативного використання пам'яті та ресурсів. Він також може представляти велику кількість можливих комбінацій виразів чи структурних даних за допомогою обмеженого обсягу пам'яті, що є корисним для оптимізації програм. Загалом, ефективний граф спрощує процес оптимізації виразів, дозволяючи продуктивно виявляти та використовувати еквівалентності між частинами програмного коду.

Всі елементи виразу представлені як е-вузли, які можуть представляти змінні, константи або операції. Е-класи формуються з еквівалентних е-вузлів, що дозволяє їм бути заміненими один одним. Під час виконання програми, E-graph дозволяє динамічно вставляти нові е-вузли та об'єднувати в е-класи, що відкриває можливість динамічного переписування виразів. Мета полягає в тому, щоб, знаючи всі можливі еквівалентні форми виразу, вибрати оптимальний варіант, який може бути більш ефективним або зручним для подальшого використання. Бібліотека Egg використовує цей метод для оптимізації програм, зокрема у контексті покращення виразів та пошуку найкращих представлень. Equality Saturation виявляється корисним в ситуаціях, де існує багато можливих оптимізацій і важко заздалегідь визначити, яка з них буде найрезультативнішою. Завдяки цьому методу програми можуть динамічно переписуватись та адаптуватись до змінних умов і вимог.

Наступне зображення показує етап за етапом процес оптимізації виразу a * 2 / 2 до просто a. (див. рисунок 2.3) [19]

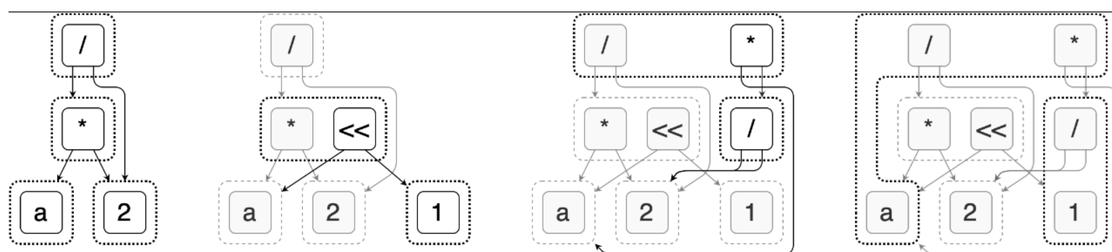


Рисунок 2.3 – Оптимізація виразу

Якщо звернути увагу на другий граф з рисунку 2.3, то можна помітити усі вищеописані складові e-графу, а саме e-граф, e-клас, e-вузол. (див. рисунок 2.2)

Кожен вузол на графі є e-вузлом, що може відображати змінну, константу або операцію. Група e-вузлів може утворювати e-клас, який представляє собою сукупність еквівалентних вузлів із тією самою семантикою, які можуть взаємозамінно використовуватись. Дочірні вузли кожного e-вузла є e-класами, які спільно утворюють ефективний граф. Таким чином, ця структура даних може відображати велику кількість утворюваних комбінацій, займаючи обмежений обсяг простору.

Отже, бібліотека Egg у Rust є незамінним інструментом для написання власного оптимізатора скриптів. Основні переваги включають в себе використання концепції насичення, що дозволяє системі ефективно оптимізувати вирази та вибирати оптимальні варіанти. Інша важлива особливість це можливість динамічних оптимізацій, що дозволяє вносити зміни в граф виразів під час виконання програми. Це забезпечує гнучкість та можливість поліпшення коду на льоту. Ефективне використання пам'яті та ресурсів є ще однією перевагою, оскільки бібліотека представляє багато комбінацій виразів за обмеженого обсягу пам'яті. Важливо відзначити, що Egg використовує можливості низькорівневого програмування в Rust, що робить його продуктивним інструментом для оптимізації коду. Крім того, активна та підтримувана спільнота Rust гарантує надійну підтримку та постійне вдосконалення інструментів, таких як Egg. Загалом, використання бібліотеки Egg у Rust дає змогу створювати потужне програмне забезпечення, надаючи високу гнучкість у програмуванні.

На момент написання роботи бібліотека Egg у Rust була популярною та визнаною своєю результативнісю у виразовій оптимізації з використанням насичення. Проте, у світі програмування розвиваються нові інструменти, і можливо, виникли нові аналоги або альтернативи. Декілька бібліотек та інструментів, які також можуть бути корисними для виразової оптимізації або мають певні схожі концепції це Simplr (Rust). Це ще одна бібліотека у Rust для роботи з графами та зміни вхідного коду. Rustc це компілятор мови програмування Rust, що також використовує певні концепції насичення для оптимізації внутрішнього представлення коду. Z3 (зазвичай використовується для SMT розв'язування). Хоча це не є прямим аналогом, Z3 може бути використаний для розв'язування різних задач у схожих областях, таких як оптимізація та розробка аналізаторів. LLVM (зі спеціалізованими оптимізаціями) - це набір компіляторних інструментів, які також мають спеціалізовані інструменти для покращення продуктивності коду.

Проте вибір між різними бібліотеками також залежав від конкретних потреб проекту, ступеня зручності використання та особистих вподобань. Перед вибором було оцінено вищеописані конкретні характеристики та можливості кожної бібліотеки і в ході обґрунтованого аналізу за основу обрано EGG Library.

Висновки до розділу 2

Отже, сучасні методи оптимізації запитів у реляційних системах, такі як компіляційна оптимізація, метод вкладеного блоку та генетичні, еволюційні алгоритми, сприяють ефективній обробці запитів і покращують продуктивність баз даних. Обраний еволюційний підхід до оптимізації виконання SQL запитів використовує гнучкість генетичних алгоритмів для автоматизації пошуку оптимальних рішень. Його переваги включають гнучкість, автоматизацію, здатність уникнути локальних максимумів та миттєву адаптацію до змін. Процес еволюційної оптимізації включає формування популяції планів виконання, визначення вартості за допомогою функції вартості та ітеративні вдосконалення через правила

переписування. Цей підхід може бути ефективним для задач оптимізації виконання SQL запитів у реляційних системах баз даних.

Еволюційний підхід до оптимізації виконання SQL запитів включає постійне вдосконалення та адаптацію стратегій оптимізації відповідно до змін у навколишньому середовищі, обсягу даних та вимог до продуктивності. Цей підхід базується на постійному вивченні та аналізі роботи системи з метою виявлення можливостей для поліпшення продуктивності виконання SQL запитів. Цей підхід включає багато етапів, а основний це систематичний моніторинг роботи бази даних для виявлення точок оптимізації та аналізу використання ресурсів. Також визначення та адаптація оптимізаційних стратегій відповідно до змін у схемі бази даних, обсягу даних, частоти запитів та інших факторів. Включає послідовні цикли вдосконалення з подальшим тестуванням нових стратегій, аналізом їх ефективності та впровадженням найкращих практик у продуктивне відпрацювання скриптів. Використання спеціальних інструментів для вимірювання часу виконання запитів, виявлення точок затримок та ідентифікації можливих шляхів оптимізації.

3 РОЗРОБКА ПІДСИСТЕМИ ОПТИМІЗАЦІЇ СКРИПТІВ ДЛЯ ВИРІШЕННЯ АНАЛІТИЧНИХ ЗАДАЧ КОЛ-ЦЕНТРУ

3.1 Аналіз основних правил оптимізації SQL запитів

SQL запит - це синтаксична конструкція для взаємодії з базою даних, яка використовується для виконання різних операцій таких як додавання, зчитування, оновлення та видалення даних. SQL запити дозволяють користувачам отримувати доступ до інформації з баз даних, фільтрувати дані, виконувати аналітику, а також здійснювати адміністративні дії. SQL запити складаються з різних частин, таких як SELECT (для вибору даних), INSERT (для додавання даних), UPDATE (для оновлення даних) та DELETE (для видалення даних) і вони використовуються для виконання конкретних завдань в контексті роботи з базою даних.

Існує три фази, через які проходить запит під час обробки його власне СУБД. По-перше, це парсинг (англ. “parsing” - автоматизований збір та структурування даних з джерела) і транслітерація (дослівно переклад). По-друге, це оптимізація (англ. “optimization” - комплекс заходів, спрямованих на удосконалення та оновлення існуючих методів для досягнення мети). По-третє, оцінювання. Більшість запитів, надісланих до СУБД, пишуться на високорівневій мові, такій як SQL. Під час транскрибування скрипта наявний етап перекладу, тобто зрозуміла для людини форма запиту перекладається у форми, придатні для використання СУБД. Це може бути відображено у формі виразу реляційної алгебри, дерева запитів чи графу запитів. Наприклад, напишемо елементарний запит.

```
SELECT create
FROM films
WHERE create = ‘Nolan’;
```

Даний запит може бути перекладено (транскрибовано) будь-яким із наведених нижче виразів реляційної алгебри, наприклад настпуним виразом, який пояснює певні відношення для запиту до даних, який описано вище за допомогою мови скриптів (див. формулу 3.1):

$\delta_{create=\text{"Nolan"}}(\pi_{create}(films))$ або $\pi_{create}=(\boldsymbol{\delta}_{create=\text{"Nolan"}}(films))$, (3.1)

де δ – певна вибірка, узагальнена — це одиничний оператор, що записується як $\boldsymbol{\delta}_{Ч}(R)$; Ч - формула логіки висловлювань, який в свою чергу містить деякі атоми, які прийняті в стандартній вибірці та відповідних операторах заперечення, кон'юнкції, диз'юнкції. Така вибірка вибирає повністю всі кортежі із R, для яких Ч – істина;

π – проєкція — це також унарна операція, що записується як $\pi_{a1,\dots,an}(R)$;

a_1-a_n це множина назв атрибутів. Результатом зазвичай є множина, що отримується із всіх кортежів та обмежується a_1-a_n.

Загалом, реляційна алгебра Кодда - це формальна система для операцій над реляційними даними, яка була розроблена Едгаром Ф. Коддом, одним із початківців реляційних баз даних. Реляційна алгебра Кодда надає мову для виразу операцій, які можна виконати над реляційними таблицями, такими як вибірка, проекція, об'єднання, різниця, декартовий добуток та інші. Ці операції дозволяють виконувати запити до бази даних, фільтрувати дані, отримувати підмножини рядків або стовпців і виконувати інші операції для обробки та аналізу даних в реляційних таблицях. Реляційна алгебра Кодда є однією з теоретичних основ реляційних баз даних і допомагає визначати, які операції можна виконувати з даними і як це робити. Вона є важливою для розробки SQL запитів та реляційних систем керування базами даних, оскільки SQL використовує реляційну алгебру Кодда як основу для своїх операцій.

Відповідно цей запит може бути представлений у вигляді графового дерева (див. рисунок 3.1).

Рисунок 3.1 – Запит у вигляді дерева

Після парсингу та транслітерації у вираз з реляційної алгебри Кодда запит перетворюється у читабельну для нижчого рівня мови форму, зазвичай це дерево запитів або граф, які обробляє конкретна система або підсистема. Тоді оптимізатор виконує аналіз запиту та створює ряд дійсних планів оцінювання. Відтам він визначає найбільш відповідну оцінку та сплановує виконання. Після вибору плану він передається у механізм виконання запитів DMBS (англ. “DBMS” - це програмне забезпечення або система, яка дозволяє створювати, керувати та взаємодіяти з базами даних. Головною функцією DBMS є забезпечення надійного зберігання даних, доступ до даних, забезпечення безпеки і можливість виконання різних операцій з даними, таких як додавання, оновлення, видалення і зчитування. DBMS також допомагає забезпечити цілісність даних і забезпечити одночасний доступ до даних для кількох користувачів), де виконується план і повертаються результати.

Першим кроком у обробці запиту, надісланого до СУБД є перетвоення запиту в форму придатну для використання механізмами обробки запитів. Мова запитів високого рівня представляє запит у вигляді рядка або послідовності символів. Певні послідовності символів представляють різні типи токенів, наприклад ключові слова, оператори, операнди, літеральні рядки тощо. Загальновідомо, що в SQL, як і в інших високорівневих мовах програмування існують правила (синтаксис і граматика), які власне і керують яким чином слід об’єднувати дійсні твердження. Основним завданням синтаксичного аналізатора є вилучення токенів з необробленого рядка символів і переклад їх у відповідні внутрішні елементи даних, а саме операції та операнди реляційної алгебри, а також структури, це може бути дерево запитів чи граф запитів. Останнім завданням синтаксичного аналізатора є перевірка коректності синтаксису вихідного рядка запиту. (див кресленник К). На етапі оптимізації запиту процесор запитів застосовує внутрішні правила реструктуризації запиту для перетворення їх структури в еквівалентні, але більш ефективні форми. Вибір найбільш релевантних правил для часу та місця застосування є основною функцією двигуна оптимізації.

Останнім етапом обробки запиту є фаза оцінки. Вибирається найефективніший план оптимізації і безпосередньо застосовується до запиту.   Також може існувати

декілька планів виконання у одного запиту. Окрім обробки запиту простим послідовним способом, деякі індивідуальні операції запитів можуть оброблятися паралельно або як незалежні процеси, або як взаємозалежні конвеєрні процеси чи потоки. Незалежно від обраного методу, фактичні результати повинні бути однаковими. [20]

Загальновідомо, що саме читання даних є першою складовою data literacy (інформаційної грамотності). Що означає читати дані? У звичайному розумінні «читати» означає «сприймати і розуміти щось». Основний зміст, який можна вкласти в поняття «дата-грамотність» це приймати за допомогою органів чуття деякі дані, розуміючи їх. [16]

Отже, читання даних є фундаментом аналізу даних. Якісний і безпомилковий аналіз може створити аналітик, який впевнено володіє читанням даних, а саме створенням скриптів. Правильна інтерпретація запиту, а згодом його графічна прорисовка надає загальну картинку стану усіх бізнес-процесів кол-центру.

Кол-центр - це місце, де оператори, що пройшли адаптацію, обробляють великий обсяг вхідних і вихідних телефонних дзвінків, а також здійснюють інші форми комунікації з клієнтами. Важливим аспектом є те, що велика кількість задач, спрямованих на вдосконалення продуктивності, підвищення якості обслуговування та оптимізацію бізнес-процесів присутні в аналітиці кол-центру. Основні задачі аналітика спрямовані на відстеження кількості дзвінків на оператора, аналіз тривалості обслуговування кожного дзвінка, визначення часу очікування клієнтів у черзі, моніторинг рішення проблем клієнтів, аналіз якості спілкування операторів, визначення показників клієнтської задоволеності (NPS, CSAT), аналіз історичних даних для прогнозування кількості дзвінків, визначення пікових годин та днів навантаження, аналіз навантаження для оптимізації розподілу операторів за робочим графіком, вимірювання результативності маркетингових та рекламних кампаній, визначення конверсії дзвінків у продажі, аналіз результатів навчання та тренінгів операторів, визначення індивідуальних слабких місць та можливостей покращення, боротьба з обманом та шахрайством, виявлення аномалій у зверненнях клієнтів, моніторинг ситуацій, що можуть свідчити про шахрайство. Ці аналітичні задачі

допомагають підвищити ефективність кол-центру, забезпечити високий рівень обслуговування та підтримувати позитивний досвід клієнтів. А отже для відпрацювання ефективних рішень для усіх задач має бути написано і протестовано SQL скрипт.

На прикладі наступної задачі кол-центру було створено скрипт, а також вжито оптимізаційних заходів, щодо зменшення планів виконання скриптів. Одна з основних задач аналітика спрямована на виведення фінансових показників для банку за певний період часу, зокрема, пов’язаних з переказами грошей через різні системи та валютні операції, які допомогли здійснити оператори кол-центру. Наступний код (див. Додаток К) є основною частиною скрипту для обробки та аналізу даних з бази, пов'язаних із фінансовими операціями банку. Його було розроблено для отримання фінансових показників та статистики за певний період часу. Загальний план запиту виглядає настпуним чином: код визначає змінні та виконує запит до таблиці з курсами валют за певний період (починаючи з @month_beg до @rep_date). Розрахунок курсу долара проводиться для кожної дати. Аналіз доходу банку в різних валютах. Далі виконуємо аналіз даних про перекази, проведені через певну систему. Розрахунок доходу банку в валюті переказу та в гривнях виконується на основі певного відсотку від комісії. Аналіз даних про відділення та перекази (MG, Ria, WU). Далі отримуємо інформацію про відділення банку, а також аналіз переказів через системи MG, Ria та WU. Це включає як відправку, так і отримання грошей через ці системи. Також є блок коду, який визначає кількість операцій для кожного клієнта за конкретний період часу та тип операції. Результат об'єднується в окрему таблицю. На кінцевому етапі створюється основна таблиця (#mg), яка об'єднує дані з різних джерел, таких як курси валют, доход банку, інформація про відділення кол-центру та кількість операцій.

Було проведено аналіз плану виконання скрипта і зроблено наступні висновки. У даному запиті часто відслідковуються складні конструкції у процесі обробки даних у тимчасових таблицях. (див. рисунок 3.2).

```
UNION

select
date(TransferDate)
'Ria'
TransferNumber
i.CurrencyTag
--CurrencyAmount,
case when Direction = 0 then ROUND(ISNULL(TransferTotalFeeAmount, 0) * 0.25, 2) e
sum(CurrencyAmount*(case when Direction = 0 then ROUND(ISNULL(TransferTotalFeeAmc
FROM dfc.tb_cnb_mtProfixTransfers i
left join #R r on r.cdate=date(TransferDate) and r.CurrencyTag=i.CurrencyTag
WHERE SystemId = 14
and Status=40
and PermitFlag!=255 and date(TransferDate)>=@month_beg and date(TransferDate)<=@r
GROUP BY TransferDate, TransferNumber, i.CurrencyTag, CurrencyAmount, TransferTot

UNION

SELECT
date(bls.DayDate_fuib)
'Ria'
info
bls.CurrencyTag
--cast(cast(CrncyAmount as NUMERIC(15,2)) as char)
cast(CrncyAmount as NUMERIC(15,2))
cast(MainAmount as NUMERIC(15,2))
FROM DFC.tb_sc_vBills bls
JOIN DFC.tb_sc_vAccounts acc on bls.DebitId=acc.Id and (acc.Moniker =
WHERE bls.DayDate_fuib > @month_beg
and info<>''
and PermitFlag=0

UNION
SELECT
date(bls.DayDate_fuib)
'MG'
info
bls.CurrencyTag
cast(CrncyAmount as NUMERIC(15,2))
cast(MainAmount as NUMERIC(15,2))
FROM DFC.tb_sc_vBills bls
```

Рисунок 3.2 – Вихідний скрипт

Варто розподіляти ітерації за рахунок Update, а не за допомогою Union/Join. (див. рисунок 3.3).

```
UPDATE
ltb_Profix_Transfers a
SET
a.Sender_issue_country     = b.IssueCountry,
a.Sender_passport_series   = b.Series,
a.Sender_passport_number   = b.[Number],
a.Sender_doc_type          = c.[Name]
FROM
dfc.tb_cnb_mtClientDocuments b
LEFT JOIN dfc.tb_cnb_mtDocTypes c ON c.Id=b.DocTypeId
WHERE
 a.Sender_Doc_Id = b.Id;
```

Рисунок 3.3 – Оптимізація за рахунок розділення ітерацій

План запитів суттєво ускладнюється при перенасиченні однієї ітерації конструкціями Union та Join. За можливості, альтернативою буде використання

UPDATE. Використання UPDATE дозволило суттєво спростити план запиту, розділивши завантаження даних та їх обробку у тимчасовій таблиці на кілька ітерацій. При цьому ефективність та швидкість виконання запиту підвищилася, а саме у додатку Г відображено INSERT + UNION|JOIN, а на рисунках 3.4 та 3.5 UPDATE (див. рисунок 3.4, 3.5, додаток Г).

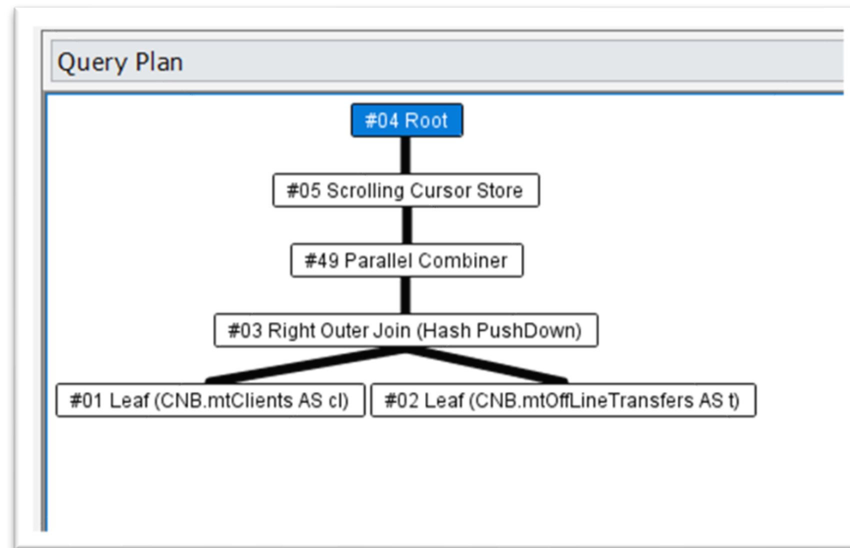


Рисунок 3.4 - Update

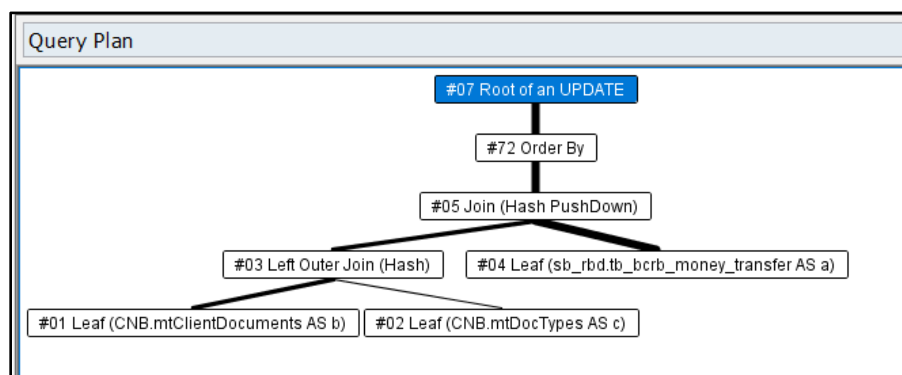
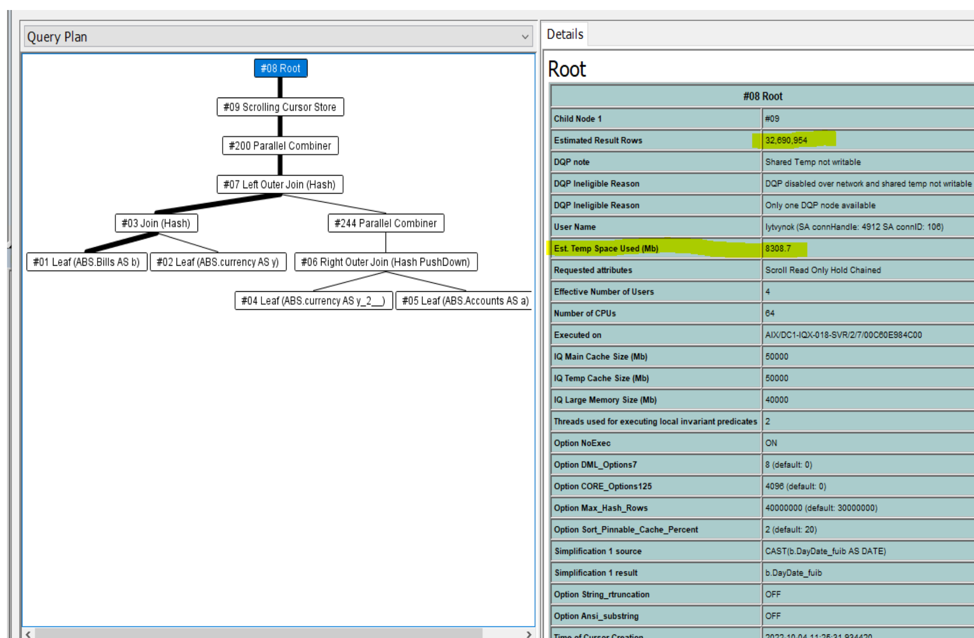


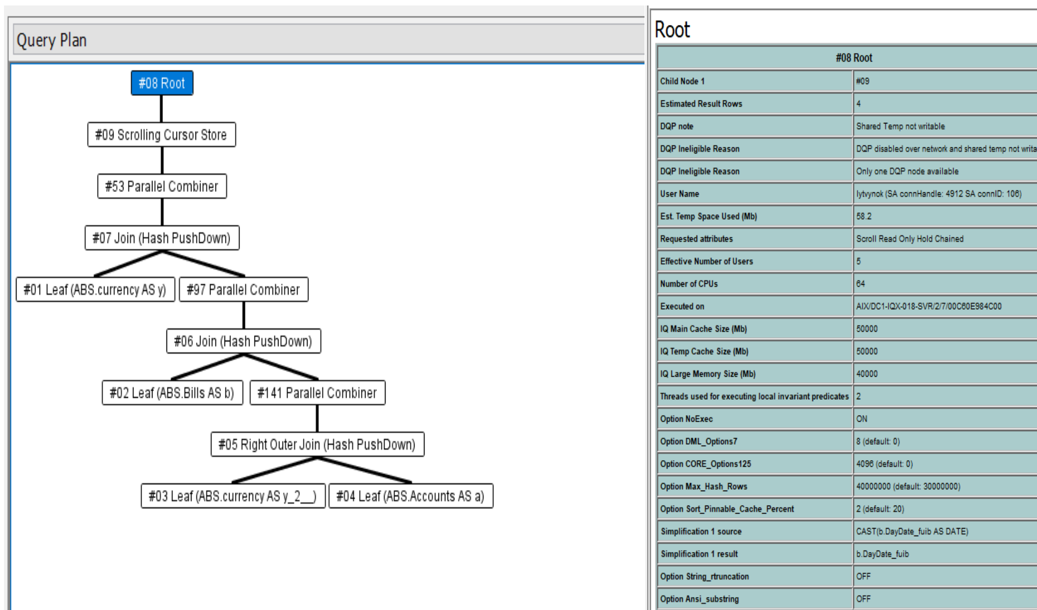
Рисунок 3.5 – Update

Також план запитів за умови поєднання OUTER JOIN (LEFT|RIGHT) (див рисунок 3.6) суттєво збільшує потребу ресурсу пам’яті, оскільки проводить повне сканування таблиці, з якою ведеться об’єднання. По можливості варто застосовувати INNER JOIN (див. рисунок 3.7).



#08 Root	
Child Node 1	#09
Estimated Result Rows	32,690,954
DQP note	Shared Temp not writable
DQP Ineligible Reason	DQP disabled over network and shared temp not writable
DQP Ineligible Reason	Only one DQP node available
User Name	lytvynok (SA connHandle: 4912 SA connID: 106)
Est. Temp Space Used (Mb)	8308.7
Requested attributes	Scroll Read Only Hold Chained
Effective Number of Users	4
Number of CPUs	64
Executed on	AIX/DC1-IQX-018-SVR/2/7/00C60E984C00
IQ Main Cache Size (Mb)	50000
IQ Temp Cache Size (Mb)	50000
IQ Large Memory Size (Mb)	40000
Threads used for executing local invariant predicates	2
Option NoExec	ON
Option DML_Options7	8 (default: 0)
Option CORE_Options125	4096 (default: 0)
Option Max_Hash_Rows	40000000 (default: 30000000)
Option Sort_Pinnable_Cache_Percent	2 (default: 20)
Simplification 1 source	CAST(b.DayDate_fuib AS DATE)
Simplification 1 result	b.DayDate_fuib
Option String_rtruncation	OFF
Option Ansi_substring	OFF
Time of Cursor Creation	2022-10-04 11:25:31.934420

Рисунок 3.6 - LEFT JOIN



#08 Root	
Child Node 1	#09
Estimated Result Rows	4
DQP note	Shared Temp not writable
DQP Ineligible Reason	DQP disabled over network and shared temp not writable
DQP Ineligible Reason	Only one DQP node available
User Name	lytvynok (SA connHandle: 4912 SA connID: 106)
Est. Temp Space Used (Mb)	58.2
Requested attributes	Scroll Read Only Hold Chained
Effective Number of Users	5
Number of CPUs	64
Executed on	AIX/DC1-IQX-018-SVR/2/7/00C60E984C00
IQ Main Cache Size (Mb)	50000
IQ Temp Cache Size (Mb)	50000
IQ Large Memory Size (Mb)	40000
Threads used for executing local invariant predicates	2
Option NoExec	ON
Option DML_Options7	8 (default: 0)
Option CORE_Options125	4096 (default: 0)
Option Max_Hash_Rows	40000000 (default: 30000000)
Option Sort_Pinnable_Cache_Percent	2 (default: 20)
Simplification 1 source	CAST(b.DayDate_fuib AS DATE)
Simplification 1 result	b.DayDate_fuib
Option String_rtruncation	OFF
Option Ansi_substring	OFF

Рисунок 3.7 - Inner JOIN

Також більшість аналітиків кол-центру для того аби згадати структуру таблиці, або ж для обмеження вибірки даних і тестування цільового запиту застосовують Select top N from table. Часом, особливо на великих масивах даних, ця функція забирає суттєвий ресурс на обробку запиту. Для того, щоб мінімізувати цей вплив на рівні коду можна використовувати системні процедури на рівні клієнта варто обмежити

властивість виводу даних (fetch) до мінімуму і застосувати select * from (див. рисунок 3.8).

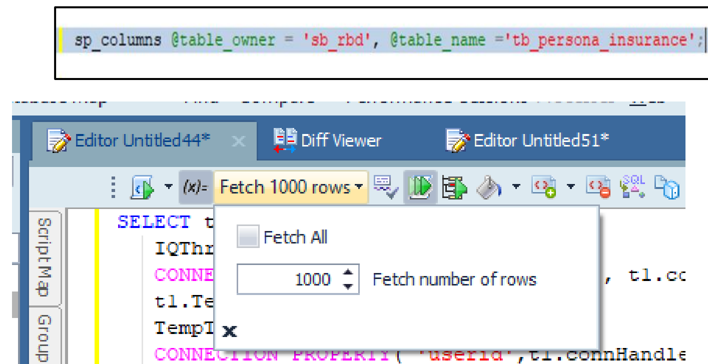


Рисунок 3.8 – Використання системних процедур

Ще одним аспектом на який варто звернути увагу є дуже поширена проблема – нерелевантність результату обсягам оброблюваних даних.

На прикладі скриптів (див. Додаток К, Додаток С) можна зробити висновок, що результат у кінцевій вибірці змінюється протягом місяця, ретроспектива не змінюється, але запит щоразу проводить обробку і вивантаження даних на інтервалі трьох років. Чим більше даних аналітик кол центру забирає для обробки – тим більшим буде мультиплікатор усіх неявних проблем у скрипті – тим довше і важче він працюватиме. Отже, виходом є обмеження обсягу вибірок потрібним періодом ще на початку написання коду та створення правильних предагрегатів.

Оптимізація SQL запитів є ключовим етапом у розробці та управлінні базами даних, спрямованим на забезпечення максимальної продуктивності та швидкодії операцій обробки даних. Забезпечення ефективності SQL запитів має прямий вплив на продуктивність веб-додатків, систем та інших інформаційних проектів. У цьому контексті важливо розглянути основні принципи та методи оптимізації SQL запитів, серед яких важливе місце відводиться таким аспектам, як хеш-з'єднання, використання тимчасових таблиць та оптимізація операторів SQL:

– хеш-з'єднання (Hash Join). Оптимізація SQL запитів з використанням хеш-з'єднання є важливим елементом для підвищення ефективності об'єднання даних з різних таблиць. Використання хеш-функцій для швидкого

знаходження відповідних записів дозволяє значно зменшити час виконання складних запитів. Оптимальний вибір індексів та ретельне налаштування хеш-з'єднань можуть істотно покращити продуктивність SQL запитів;

- тимчасові таблиці. Використання тимчасових таблиць є ефективним методом для збереження проміжних результатів обчислень та фільтрації даних. Цей підхід сприяє спрощенню завдань та зменшенню навантаження на базу даних, а також може бути використаний для попереднього аналізу та оптимізації даних перед виконанням складних операцій;
- оптимізація операторів SQL. На важливому етапі оптимізації SQL запитів стоїть вибір та належне використання операторів SQL. Використання оптимальних методів фільтрації, групування та вибору даних може значно покращити читання та запис до бази даних, а також зменшити час виконання запитів.

Загальний підхід до оптимізації SQL запитів допомагає забезпечити оптимальну продуктивність та ефективність роботи з базою даних. Це важливий аспект розробки програмного забезпечення, спрямований на оптимізацію роботи з великими обсягами даних та забезпечення швидкодії операцій.

Загальновідомо, що на етапі планування будь-якого програмного додатку, розробники створюють модель бази даних, в якій будуть зберігатись усі дані, що транслюються в програмі. Тобто аналітики обирають тип бази даних в залежності від потреб. Розробники роблять свій вибір бази даних (реляційна модель або модель NoSQL) і вибір системи управління базами даних (СУБД). Надалі має бути готова UML- модель системи, що включає сценарій використання та інші діаграми. Відповідно має бути написаний SQL запит для доступу до необхідних даних. Будь-який SQL запит можна написати різними способами, але загальноприйнято дотримуватися найкращих практик, щоб отримати найбільш продуктивний запит. Зазвичай їх дотримуються при створенні власних скриптів.

Отже, на основі джерел зібрано і виведено найкращі практики для написання SQL запитів. А саме:

- варто використовувати EXISTS замість IN, щоб перевірити чи існують запитувані дані в таблиці;
- уникати * в операторі SELECT, краще замість * прописувати назви стовпців, які необхідно вивести;
- обирати відповідний тип даних під конкретні випадки, наприклад замість типу text, що може зберігати до двух гігабайт даних тип varchar (20) – в дужках якого можна вказати кількість символів, що буде передаватись. Відповідно буде виділено під стовпець менше пам’яті і ефективніше використано ресурс бази даних;
- уникати NCHAR і NVARCHAR, оскільки обидва типи даних займають лише помножену на два пам’ять типів CHAR і VARCHAR;
- уникати NULL у полі фіксованої довжини. Уникання значення NULL у полі фіксованої довжини має кілька причин. Це ефективність використання пам'яті, так як поля фіксованої довжини зазвичай використовують фіксовану кількість пам'яті незалежно від фактичного обсягу даних. Введення NULL у таке поле не впорядковане і може призвести до неефективного використання пам'яті. Деякі системи керування базами даних (СКБД) можуть використовувати спеціальні оптимізації для полів фіксованої довжини, які не передбачають обробку NULL значень. Введення NULL у фіксоване поле може зробити код для обробки цих даних більш складним, оскільки потрібно буде робити додаткові перевірки на NULL перед роботою з даними. Зберігання NULL у фіксованому полі може призвести до неочікуваних результатів, особливо якщо це не передбачено в бізнес-логіці. Обробка NULL значень в фіксованих полях може бути менш ефективною при пошуку та фільтрації даних. Звісно, в деяких випадках використання NULL у фіксованих полях може бути обґрунтованим, зокрема, якщо вам потрібно відокремлювати значення, які відсутні від тих, що мають конкретне значення. Однак, взагалі кажучи, використання NULL у фіксованих

полях може викликати певні труднощі, і перед використанням його слід ретельно обдумати.

- Варто уникати HAVING. У більшості випадків HAVING застосовується після GROUP BY та WHERE, що означає, що всі дані будуть згруповані та фільтровані перед застосуванням умови HAVING. Це може призвести до зайвого обчислення та збільшення обсягу даних, що обробляється. Там, де можливо, краще використовувати WHERE для фільтрації даних, оскільки це буде зроблено перед групуванням, що полегшить оптимізацію запиту. HAVING може призвести до оптимізаційних проблем, оскільки СКБД може мати обмежені можливості для оптимізації цього типу умов. Використання HAVING може робити SQL запит менш зрозумілим та читабельним. Враховуючи, що HAVING застосовується після групування, може бути важко зрозуміти, як саме фільтрація відбувається. У деяких випадках є можливість використати підзапит або інші конструкції SQL для досягнення того ж ефекту без використання HAVING. Необхідно враховувати, що в окремих ситуаціях використання HAVING є необхідним і логічним. Однак, краще використовувати інші конструкції SQL для фільтрації даних;
- видалення невикористаних індексів. Невикористані індекси займають дисковий простір. Видалення їх дозволяє звільнити місце на диску та ефективніше використовувати ресурси. Під час вставки або оновлення даних база даних повинна підтримувати індекси. Невикористані індекси збільшують час, потрібний для цих операцій. Видалення непотрібних індексів може покращити продуктивність при вставці та оновленні даних. Зайві індекси можуть уповільнювати вибірку даних, оскільки системі потрібно підтримувати декілька індексів під час виконання запитів. Видалення невикористаних індексів може поліпшити швидкість вибірки даних. Кожен індекс вимагає деякої кількості пам'яті для зберігання. Зайві індекси використовують пам'ять і видалення їх зменшує витрати пам'яті бази даних. Якщо індекс не використовується, його існування викликає плутанину та ризики

консистентності даних. Видалення невикористаних індексів сприяє збереженню чистоти та порядку в базі даних.

– використання JOIN замість підзапитів. Запити, які використовують JOIN, мають бути оптимізовані більш ефективно, оскільки СКБД може використовувати різні методи з'єднання (наприклад, з'єднання по індексу) для швидкого отримання результатів. JOIN може бути більш масштабованим для обробки складних запитів та з'єднань декількох таблиць, оскільки він дозволяє використовувати різні типи з'єднань (наприклад INNER JOIN, LEFT JOIN, RIGHT JOIN) для задоволення конкретних потреб. SQL запити з використанням JOIN є більш читабельними та зрозумілими, особливо для тих, хто читає та редагує код. JOIN визначає відношення між таблицями прямо у частині FROM, що полегшує зрозуміння, як дані з'єднуються. Системі керування базами даних легше оптимізувати та кешувати результати запитів, які використовують JOIN, що натомість призводить до покращення продуктивності. Використання JOIN дозволяє легко додавати додаткові умови та об'єднувати багато таблиць, що робить його гнучким та придатним для складних структур даних. Хоча підзапити також корисні у певних сценаріях, використання JOIN є загальною та часто більш ефективною практикою для з'єднання даних з різних таблиць;
– використання виразу WHERE. Зазвичай фільтрація значно зменшує обсяг даних;
– використання WITH (NOLOCK) під час запиту. Означає що БД не повинна блокувати ресурси для читання даної таблиці впродовж часу виконання запиту. Це відноситься до рівня ізоляції транзакцій в базі даних;
– використання SET NOCOUNT ON та TRY CATCH, щоб уникнути тупикової ситуації. Коли SET NOCOUNT ON встановлено, SQL Server не повертає кількість змінених або вставлених рядків під час виконання кожної команди. Це дозволяє зменшити об'єм даних, який повертається клієнту і поліпшити продуктивність особливо для великих операцій. Конструкція TRY CATCH дозволяє обробляти виняткові ситуації в коді. Це особливо корисно при

взаємодії з базою даних, де можуть виникати помилки під час виконання операцій. Використання TRY CATCH дозволяє перехоплювати помилки та виконувати обробку помилок, уникнувши тим самим тупикових ситуацій, коли виконання транзакції припиняється через неочікувану помилку. Конструкція TRY CATCH дозволяє коректно відкатувати транзакцію у випадку виникнення помилки, що дозволяє уникнути залишкових змін, які можуть виникнути в результаті непередбачених помилок. За допомогою TRY CATCH можна здійснювати логування та відстеження помилок, щоб швидше виявляти та усувати проблеми;

– уникнення Cursor. Використання курсорів призводить до низької продуктивності, оскільки обробка записів один за одним дуже повільна, особливо при роботі з великими обсягами даних. Курсори створюють високе навантаження на сервер бази даних, якщо вони використовуються для ітерації через велику кількість записів. Використання курсорів викликає блокування ресурсів в базі даних при використанні курсорів для оновлення або видалення записів. Це впливає на конкурентність та продуктивність системи. Більшість операцій, які використовують курсори оптимізовують за допомогою масивних операцій набору даних (set-based operations), що зазвичай працюють ефективніше. У багатьох випадках можна використовувати мови програмування, такі як C#, Python, або Java, для обробки даних на рівні клієнта, замість використання курсорів на рівні бази даних;

– використання UNION ALL замість UNION. Оператор UNION використовується для об'єднання результатів двох або більше запитів та видаляє дублікати. Однак це вимагає використання додаткових ресурсів для визначення та видалення дублікатів. Оператор UNION ALL також об'єднує результати, але не видаляє дублікати. Таким чином, він зазвичай працює швидше, оскільки не вимагає витрат на видалення дублікатів;

– використання назви схеми перед іменем об’єктів SQL. Використання схеми перед іменем об'єкта допомагає уникнути конфліктів імен, особливо в великих

базах даних, де можуть існувати об'єкти з однаковими іменами в різних схемах. Використання схем допомагає в структуруванні та організації бази даних. Використання назви схеми дозволяє реалізувати багаторівневу організацію об'єктів в базі даних, що полегшує розуміння структури та логіки бази даних. Аналітикам легше розуміти, до якої схеми вони звертаються, особливо в складних запитах чи збережених процедурах, де використовуються об'єкти з різних схем. Схеми також можуть використовуватися для управління правами доступу до об'єктів;

– використання збережених процедур для часто використовуваних складних запитів. Збережені процедури дозволяють централізувати та утримувати бізнес-логіку на рівні бази даних. Це означає, що весь код, пов'язаний з операціями бази даних, може бути розташований на одному рівні, що полегшує управління та зміну цієї логіки. Збережені процедури можуть включати операції транзакцій, які забезпечують консистентність бази даних. Це дозволяє гарантувати атомарність, консистентність, ізоляцію та довершеність (ACID) для операцій. Збережені процедури можуть бути збережені та скомпільовані в базі даних, що дозволяє покращити продуктивність. Однак це може залежати від конкретного СКБД та його оптимізацій. Збережені процедури обробляють масові операції та множинні рядки даних більш ефективно, ніж окремі SQL запити. Це призводить до зменшення витрат на передачу даних між сервером та клієнтом. Збережені процедури можуть надавати додатковий захист від SQL ін'єкцій, оскільки вони дозволяють використовувати параметри та передавати значення безпосередньо через інтерфейс бази даних. Збережені процедури дозволяють використовувати параметри, що полегшує повторне використання коду та забезпечує більшу гнучкість при виклику процедур;

– уникнення функцій таблиці з кількома операторами (TVFs). Функції таблиці з кількома операторами призводять до втрати продуктивності через велику кількість обчислень та операцій, які вони виконують. Оптимізатор запитів може мати обмежені можливості оптимізації функцій таблиці з кількома операторами, оскільки ці функції виконуються для кожного рядка даних

окремо, що ускладнює використання індексів та інших оптимізацій. Функції таблиці з кількома операторами часто ускладнюють можливість використання індексів для прискорення запитів, оскільки обчислення відбуваються на кожному рядку. Використання функцій таблиці з кількома операторами робить код більш складним та важким для розуміння. Запити, які використовують функції таблиці з кількома операторами, стають складнішими для ефективного планування виконання, що призводить до втрати продуктивності. [17]

Отже, оптимізація скриптів в базах даних - важливий етап розробки, спрямований на поліпшення продуктивності, ефективності та читабельності коду. Зазначені правила вказують на кілька ключових аспектів, які сприятимуть оптимізації та покращенню виконання SQL запитів. [18]

3.2 Розробка алгоритму оптимізації SQL скриптів

Еволюційний підхід до оптимізації планів виконання SQL запитів має кілька переваг і особливостей порівняно з іншими методиками оптимізацій. Еволюційні алгоритми є досить гнучкими, оскільки їх можна налаштовувати для вирішення конкретних проблем оптимізації. За допомогою правил переписування та функції вартості є можливість налаштувати перетворення та оцінку алгоритмом планів виконання. Еволюційні методи дозволяють автоматизувати процес пошуку оптимальних рішень. Алгоритм може визначати, які перетворення потрібні для покращення вартості плану без прямого втручання програміста. Еволюційні алгоритми можуть допомагати уникнути локальних максимумів за допомогою пошуку глобально оптимальних рішень. У контексті оптимізації термін "локальний максимум" вказує на ситуацію, коли значення функції чи показника досягає свого найвищого значення в околиці певної точки, але це значення може бути менше, ніж максимальне значення в інших частинах простору. Тобто це точка, де функція має максимум, але не обов'язково глобальний (абсолютний) максимум. У процесі оптимізації, особливо коли використовуються методи локального пошуку чи оптимізації, можливість застрягання в локальному максимумі є частоою проблемою.

Оптимізаційний алгоритм може визначити краще значення функції в порівнянні з поточною точкою, але не виявити глобальний максимум через обмежений обсяг перегляду. Такий алгоритм може вважати знайдену локальну точку максимуму оптимальною, хоча існує краща точка в іншій частині простору. З урахуванням цього еволюційні алгоритми, які здатні уникати локальних максимумів, можуть бути корисними в задачах оптимізації, де важливо знайти глобально оптимальні рішення. Це особливо важливо в задачах оптимізації, де розглядається велика кількість можливих рішень. Еволюційні алгоритми можуть адаптуватися до змінних умов або оточення, реагуючи на нові обставини та навчаючись в процесі оптимізації. Такий підхід може бути застосований до різноманітних задач оптимізації від планування ресурсів до структурної оптимізації. [24]

Еволюційний підхід до оптимізації виразів плану виконання SQL запитів застосовує ідеї з еволюційних алгоритмів для покращення якості виразу через ітеративний процес. Основні етапи еволюційного підходу в даному контексті наступні. Спочатку формується початкова популяція виразів плану виконання SQL запитів. Кожен вираз представляє можливий спосіб виконання запиту. Для кожного виразу обчислюється вартість за допомогою обчислення функції вартості. Функція вартості оцінює ефективність виразу виконання запиту з урахуванням таких факторів, як кардинальність, швидкодія, обсяги даних і т.д. Визначаються правила переписування, які можуть застосовуватися до виразів плану. Ці правила описують оптимізаційні трансформації, які можна застосовувати до виразів для отримання більш ефективних планів виконання. Запускається ітеративний процес оптимізації, де на кожній ітерації вибираються певні вирази плану для оптимізації. Вибір виразів базується на їх поточній вартості та можливості застосування правил переписування. Вибрані вирази піддаються правилам переписування. Це включає зміну порядку операцій, об'єднання та перетворення операцій для поліпшення швидкодії. Після застосування правил переписування обчислюється нова вартість для отриманого виразу. Якщо нова вартість краща за попередню, то новий вираз приймається, інакше залишається попереднім. Процес продовжується до досягнення критерію зупинки, такого як фіксована кількість ітерацій, зміна вартості менше певного порогу або інші

критерії. Оптимізовані вирази та плани виконання, які досягли найменшої вартості, визначаються як результат оптимізації. Також бібліотека egg використовується для представлення та оптимізації виразів у формі еволюційного графа. Функція вартості (PlanCostFunction) обчислює вартість виразу, а правила переписування визначають оптимізаційні трансформації. (див кресленник Ц Алгоритм методики оптимізації запитів).

Загалом, основний алгоритм будь-якого оптимізатора SQL запитів розділений на кілька етапів, кожен з яких виконує певну функцію (див. рисунок 3.9).

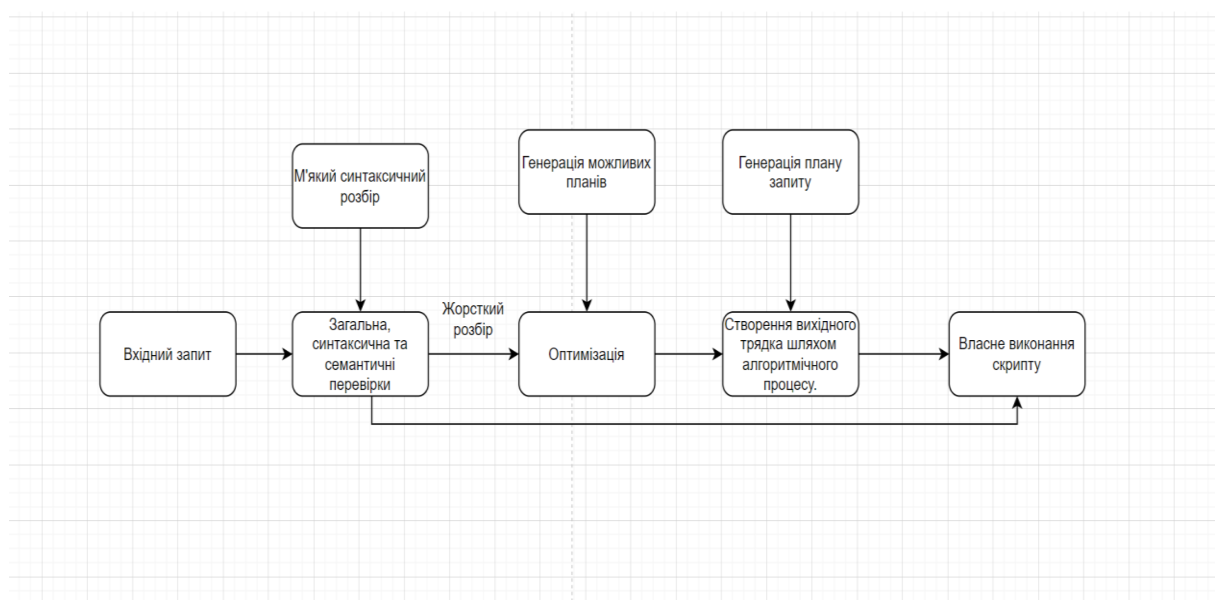


Рисунок 3.9 – Основні блоки при написанні оптимізатора

Користувач подає SQL запит для виконання. Далі відбуваєтсья отримання вхідного SQL запиту від користувача. Потім виконується перевірка правильності синтаксису та відповідність запиту семантичним правилам. Також застосовується використання лексичного та синтаксичного аналізатора для перевірки правильності структури запиту та визначення його семантики. Наступним етапом є побудова абстрактного синтаксичного дерева (AST) для представлення структури запиту. Також обов’язковим є етап використання аналізатора для перетворення вхідного SQL запиту в абстрактне синтаксичне дерево. Надалі виконується вибір оптимального плану виконання запиту з числа можливих альтернатив, а також відбувається

застосування правил переписування та евристик для оптимізації абстрактного синтаксичного дерева. Програма створює кілька різних варіантів планів виконання на основі оптимізованого дерева. В кінці відбувається формування результуючого SQL запиту на основі обраного оптимального плану. Отже, вихідний запит служить вхідними даними для всього процесу. Загальна синтаксична та семантична перевірка готує вхідний запит для подальших етапів, перевіряючи його на відповідність стандартам та виключаючи помилки. М'який синтаксичний розбір конвертує вхідний SQL запит у структуроване абстрактне синтаксичне дерево для подальших обчислень. Оптимізація визначає найкращий план виконання для абстрактного синтаксичного дерева, вибираючи меншовартісні операції та порядок їх виконання. Пісоя цього створюються різні варіанти планів виконання з урахуванням оптимізованого дерева та вибраного оптимального плану. Створення вихідного рядка використовує інформацію про оптимальний план для створення фінального оптимізованого SQL запиту.

Виходячи з загальноприйнтих блоків до написання оптимізаторів було створено власний з певними нововведеннями. Присутній ввод запиту користувачем. Вхідний SQL запит отримується від користувача з файлу та десеріалізується за допомогою бібліотеки json. Отримання вхідного SQL запиту відбувається в основній функції main після зчитування файлу. М'який синтаксичний розбір також присутній через створення абстрактного синтаксичного дерева (AST), що представлено за допомогою мови egg (Evolutionary Algorithms in Rust Grammar). Визначена мова використовується для побудови AST. Порівнюючи з основним блоком аналізаторів оптимізація також відбувається за допомогою правил переписування та евристик, які визначені вектором rules у функції runner. Використовується бібліотека egg для реалізації власне оптимізації. Також генерація планів виконання відбувається за допомогою правил переписування та евристик. Правила, такі як порядок з'єднань, операції з'єднань та інші, визначені в векторі rules у функції runner. Накінець створення SQL запиту представлено у виведенні результатів виконання оптимізованого запиту у консоль. Формування результуючого SQL запиту відбувається в функції main після оптимізації та виведення результатів. Отже, у коді

присутні основні етапи від вхідного запиту до виведення результатів, включаючи м'який синтаксичний розбір, оптимізацію та генерацію можливих планів виконання.

Основними відмінностями в порівнянні с іншими оптимізаторами є наступне. Даний оптимізатор використовує бібліотеку Evolutionary Algorithms in Rust Grammar для побудови та оптимізації абстрактного синтаксичного дерева (AST). Evolutionary Algorithms in Rust Grammar надає механізм правил переписування та інструменти для автоматичної оптимізації AST, що може спростити процес створення оптимізатора. Введена глобальна змінна, яка зберігає введені метадані таблиць для подальшого використання в різних частинах програми. Це корисно для оптимізації та визначення характеристик таблиць на різних етапах оптимізації. Додані структури Meta та Table для представлення введених метаданих та властивостей таблиць у форматі JSON. Ці структури дозволяють зручно представляти та обробляти інформацію про таблиці у вхідних даних. Макрос def_lang визначає мову для спрощення планів виконання SQL запитів, а макрос rewr використовується для задання правил переписування AST. Ці макроси спрощують визначення та роботу з мовою та правилами переписування. Доданий клас CostFunc для обчислення вартості виразу плану визначає, які правила переписування слід застосовувати до вхідного запиту. Використовується для визначення, які оптимізаційні правила застосовувати для кращого вибору плану. Доданий клас Analyze для аналізу плану та обчислення кардинальності, розмірів та інших характеристик вузлів використовується для аналізу та обчислення характеристик плану для функції вартості. Ці нововведення роблять оптимізатор більш гнучким та розширюваним, а використання бібліотеки спрощує реалізацію оптимізаційних правил та аналізу планів виконання (див. рисунок 3.10).

Рисунок 3.10 – Основні блоки програми оптимізатора

3.3 Визначення вимог та специфікацій. Діаграма прецедентів

В процесі визначення вимог та створення специфікацій для системи прийнято використовувати різні типи діаграм для ретельного розгляду функціональних, структурних та поведінкових аспектів системи. Зокрема, важливі наступні діаграми. Діаграма прецедентів (Use Case Diagram) дозволяє ідентифікувати та моделювати взаємодію системи з зовнішніми сутностями для досягнення конкретних цілей. Визначає прецеденти (функціональні можливості) та акторів (учасники взаємодії). [25]

Діаграма активностей (Activity Diagram) моделює процеси та дії в системі, допомагаючи відобразити послідовність дій та взаємодій учасників. Діаграма послідовностей (Sequence Diagram) показує взаємодії та обмін повідомленнями між об'єктами чи компонентами системи в конкретних сценаріях. Діаграма класів (Class Diagram) визначає структуру системи через класи, їх атрибути та методи, а також взаємозв'язки між класами. Діаграма варіантів використання (Use Case Diagram) ілюструє, як різні актори взаємодіють з системою для виконання функціональних можливостей. Діаграма станів моделює різні стани об'єктів чи системи та переходи між ними. Діаграма взаємодії надає загальний огляд взаємодій між об'єктами та системою. Ці діаграми допомагають висвітлити різні аспекти системи, враховуючи її функціональність, структуру та поведінку, що важливо для ретельного визначення та документування вимог до системи чи підсистеми.

Діаграма прецедентів є абстрактним відображенням функціонального призначення системи та відповідає на ключове питання моделювання: які конкретні дії виконує система у зовнішньому середовищі? На цій діаграмі використовуються два основних типи об'єктів: варіанти використання (прецеденти), які представляють конкретні функціональні можливості, і діючі особи (актори), які взаємодіють із системою. Між ними встановлюються такі типи відношень: асоціації між актором і прецедентом, узагальнення між акторами, узагальнення між прецедентами, а також різні типи залежностей чи асоціацій між прецедентами. Під час формулювання прецедентів акцентується увага на взаємодії системи з конкретним актором. Діаграму прецедентів підсистеми оптимізації можна відобразити наступним чином (див. рисунок 3.11).

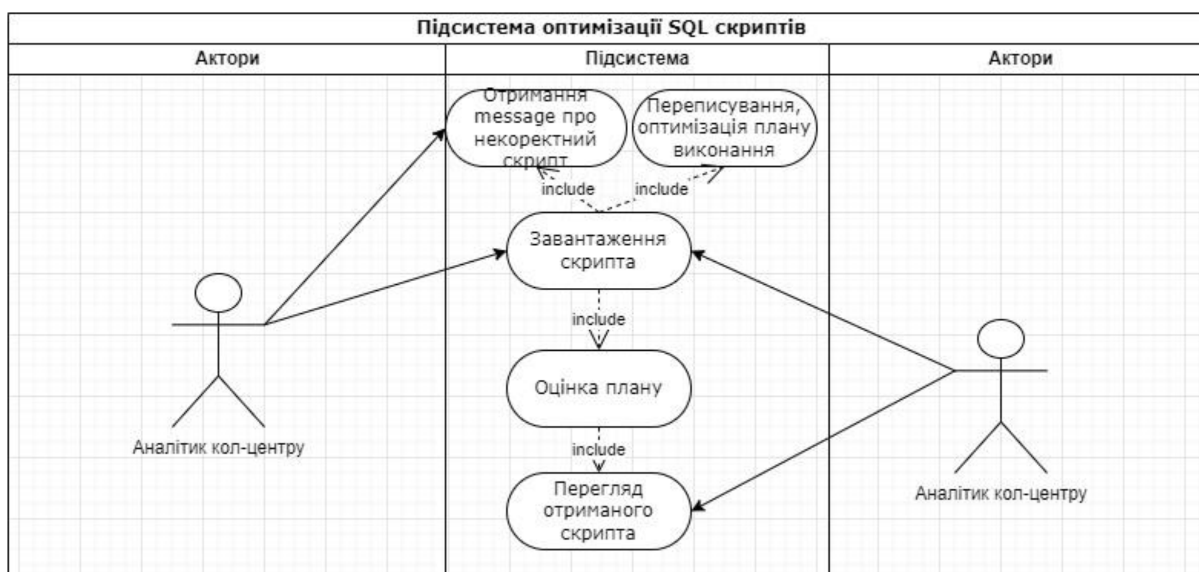


Рисунок 3.11 – Діаграма прецедентів

Прецеденти визначаються згідно зі специфікацією вимог до системи. Актором може виступати будь-який зовнішній суб'єкт, що взаємодіє з системою (наприклад, викладач), підсистема (наприклад, програма для оптимізації будь-чого) або пристрій (наприклад, барометр). Усі взаємодії між цими суб'єктами та системою розглядаються як прецеденти. Діаграми прецедентів призначені для відображення статичних аспектів системи через призму користувачів. Ця перспектива, головним чином,

охоплює поведінку системи – сервіси що доступні ззовні та надаються системою. На початкових етапах створення моделі статичних характеристик системи, діаграми прецедентів використовуються двома способами: для моделювання контексту системи, де умовно відокремлюється система та виявляються взаємодіючі особи, та для відтворення вимог до системи в моделі, де визначається які функціональності система повинна забезпечити для зовнішнього користувача без врахування конкретного методу реалізації. Кожна система має внутрішні та зовнішні сутності. Сутності, які перебувають за межами системи і взаємодіють з нею, формують її контекст. Мова моделювання UML надає можливість моделювати контекст за допомогою діаграм прецедентів, де акцент робиться на зовнішніх акторах. Визначення акторів є ключовим для опису класів сутностей, що взаємодіють із системою, і також важливо визначити, хто не є актором, оскільки це обмежує область системи. [26]

Створення діаграм прецедентів для системи спрямоване на кілька важливих цілей і має свої переваги. Діаграми прецедентів дозволяють ідентифікувати, моделювати та візуалізувати різні сценарії взаємодії користувачів з системою. Це допомагає зрозуміти, як будуть взаємодіяти з системою користувачі і як система відповідатиме на їхні вимоги. Діаграми прецедентів допомагають визначити основних акторів (суб'єктів, які взаємодіють із системою) та їхні ролі в контексті системи. Це важливо для подальшого визначення функціональних вимог до системи. Діаграми прецедентів описують конкретні сценарії взаємодії між акторами та системою. Це допомагає виявити різні шляхи взаємодії та визначити, як система повинна реагувати на різні вхідні події. Діаграми прецедентів є корисним інструментом для визначення функціональних вимог до системи. Вони служать основою для створення більш конкретних специфікацій та планування роботи над проектом. Діаграми прецедентів допомагають визначити границі системи та її взаємодію з зовнішніми суб'єктами (акторами). Це важливо для визначення, як система вписується в контекст і взаємодіє з навколишнім середовищем. Діаграми прецедентів можуть використовуватися як ефективний інструмент комунікації між

членами розробницької команди, замовниками та іншими зацікавленими сторонами. Вони надають чітке та візуальне представлення вимог та функціональності системи.

Підсистема оптимізації SQL скриптів для вирішення аналітичних задач кол-центру матиме наступну діаграму прецедентів (див. додаток С). Адміністратор системи відповідає за управління та конфігурацію підсистеми оптимізації SQL запитів. Аналітик кол-центру використовує підсистему для виконання аналітичних запитів та отримання необхідної інформації. Адміністратор може налаштовувати параметри оптимізації SQL запитів, такі як індексація таблиць, використання кешу та інші оптимізації для покращення продуктивності. Аналітик може використовувати підсистему для виконання аналітичних SQL запитів та отримання результатів оптимізованою шляхом. Підсистема оптимізації SQL - це основна частина системи, яка виконує оптимізацію SQL запитів з метою підвищення продуктивності та ефективності вирішення аналітичних задач. Взаємодія з базою даних вказує на те, що підсистема оптимізації SQL взаємодіє з базою даних для виконання запитів. Керування конфігурацією показує, що адміністратор може змінювати параметри оптимізації для відповіді на змінні умови. Результатом роботи підсистеми є оптимізований SQL запит, який може бути використаний для отримання швидких та ефективних відповідей на аналітичні запити. Аналітик може ініціювати процес оптимізації для конкретного SQL запиту. Адміністратор може змінювати параметри оптимізації для адаптації до змінних вимог аналітичних задач. Діаграма прецедентів має допомагати у розумінні взаємодії між різними компонентами системи та їх роллю у вирішенні конкретних завдань.

Отже, розглянута діаграма прецедентів для підсистеми оптимізації SQL запитів аналітичних задач кол-центру виявляється важливим інструментом для визначення взаємодії різних складових системи. Вона визначає акторів, прецеденти, систему, зв'язки та сценарії, надаючи зрозумілу картину того, як різні частини взаємодіють між собою. Актори, такі як адміністратор системи та аналітик кол-центру, чітко визначаються за їхніми ролями та обов'язками в системі. Прецеденти, такі як налаштування оптимізації та виконання аналітичних запитів, конкретизують можливі дії акторів та систему. Система представлена підсистемою оптимізації SQL, яка

відповідає за оптимізацію запитів для досягнення підвищеної продуктивності та ефективності. Зв'язки, такі як взаємодія з базою даних та керування конфігурацією, розкривають ключові аспекти взаємодії підсистеми з іншими компонентами. Отримані результати у вигляді оптимізованого SQL запиту вказують на практичну користь роботи підсистеми та її значення для задоволення аналітичних потреб користувачів. Сценарії, такі як запуск оптимізації для конкретного запиту та зміна параметрів оптимізації, підкреслюють гнучкість та адаптабельність системи до змінних умов. У цілому, діаграма прецедентів виявляється ефективним інструментом для розуміння та визначення вимог до системи, комунікації між розробниками та замовниками, а також для планування роботи над проектом. [27]

3.4 Архітектура програмного забезпечення. Діаграма класів

Архітектура програмного забезпечення — це фундаментальний елемент у розробці програмних продуктів, який визначає структуру, взаємодію та організацію компонентів системи. У цьому розділі розглядається архітектура оптимізатора SQL запитів для аналітичних задач кол-центру, що є важливим компонентом для ефективного використання та аналізу даних у великих обсягах. Архітектура ПЗ зазвичай визначає організаційну структуру системи, що натомість включає і її компоненти, їх взаємодію та властивості. При створенні шаблону архітектури програмного застосунку зазвичай звертаються до діаграми вимог (див. додаток Діаграма вимог). Діаграма вимог визначає та візуалізує вимоги до програмного продукту, допомагаючи у розумінні потреб користувачів та зацікавлених сторін. Вона описує функціональність, системні властивості, вимоги до продукту, умови змін, та взаємодію з іншими системами. Цей вид діаграми є основою для розробки програмного продукту, визначення його характеристик та функціональності. Вона також використовується для керування змінами та ризиками під час розробки, а також для уникнення непорозумінь між різними зацікавленими сторонами. У зазначеній діаграмі вимог було вказано важливість досягнення ефективного виконання SQL запитів для оперативного аналізу даних. Здатність системи адаптуватися до зростання

обсягу даних та завдань аналітики. Забезпечення неперервної доступності системи та відновлення в разі помилок. А також відповідні компоненти, які відповідають за реалізацію даних вимог.

Діаграма вимог та діаграма послідовності в програмному забезпеченні взаємопов'язані елементи, які використовуються на різних етапах розробки програмного продукту. Діаграма вимог визначає та візуалізує вимоги до системи чи програмного продукту. Вона служить для збору та аналізу потреб користувачів та зацікавлених сторін. Діаграма вимог допомагає визначити функціональність, системні властивості та вимоги до продукту. Діаграма послідовності, у свою чергу, використовується для проектування взаємодії різних компонентів системи. Вона конкретизує, як ці компоненти взаємодіють для виконання певних функцій чи завдань, визначаючи послідовність подій та обмін даними. На етапі розробки, інформація, зібрана за допомогою діаграми вимог, може використовуватися для створення діаграми послідовності. Діаграма послідовності визначає конкретні етапи взаємодії між об'єктами та компонентами системи, враховуючи вимоги та функціональність, визначені на попередньому етапі. Таким чином ці дві діаграми взаємодіють у процесі розробки для забезпечення відповідності продукту визначеним вимогам та ефективної взаємодії його компонентів.

Інструкція користувача передбачає декілька послідовних кроків. Для того аби в мові програмування Rust встановити залежності з зовіншніми бібліотеками варто додати залежності у файл Cargo.toml. У корені усіх проєктів знаходиться файл Cargo.toml з певними секціями, один з них це dependencies (від англ. – «залежності»). Далі виконається команда у cmd, а саме cargo build. Cargo автоматично завантажить та збере вказані залежності. Після успішної установки залежностей їх можна використовувати у коді. Cargo підключить бібліотеки до проекту, тож після цього з’явиться можливість імпортувати необхідні модулі та використовувати функції та типи з цих бібліотек (див. рисунок 3.12, 3.13).

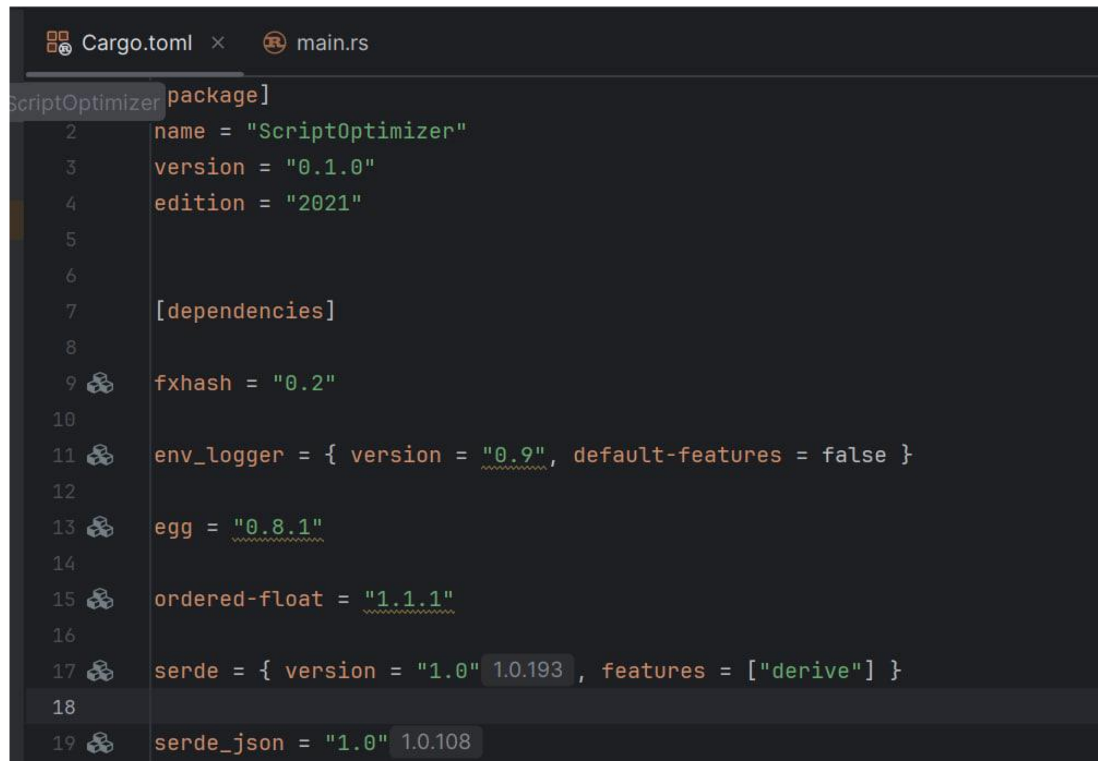


Рисунок 3.12 – Додавання залежностей у проєкт

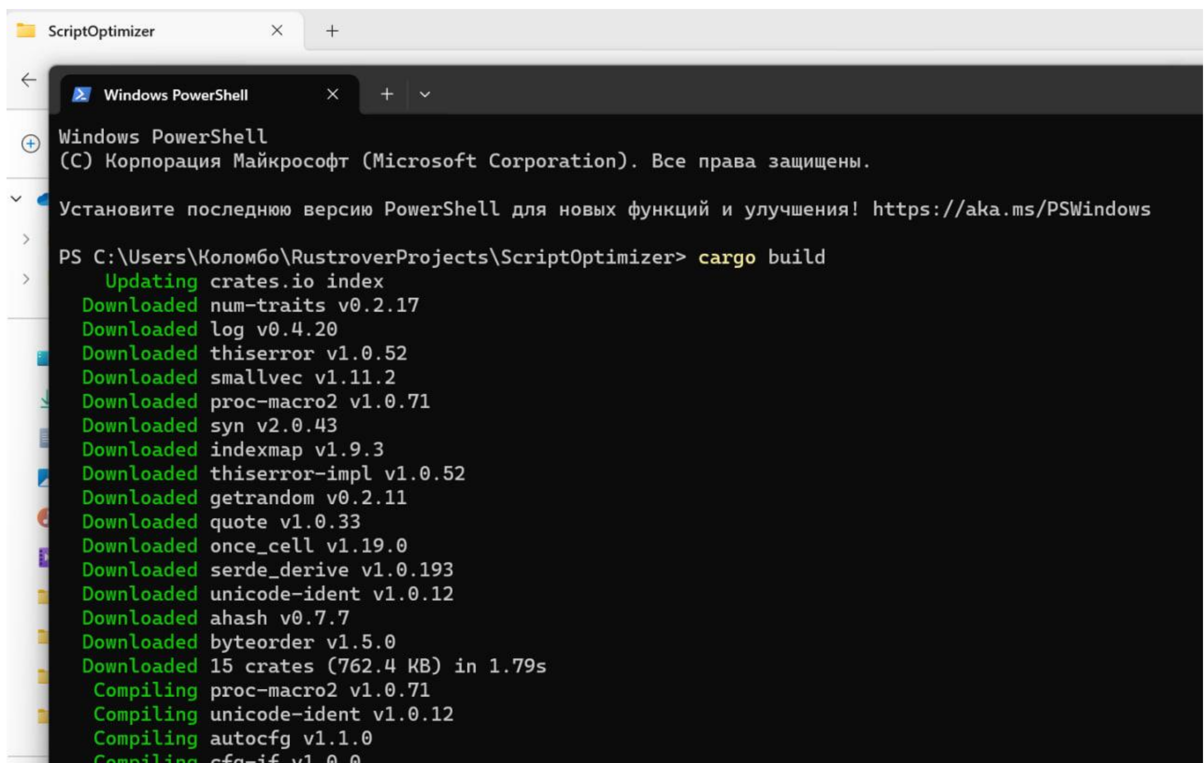


Рисунок 3.13 – Виконання команди build

Далі для продовження роботи з проєктом варто відкрити cmd від імені адміністратора та перейти в папку проєкту (див. рисунок 3.14).

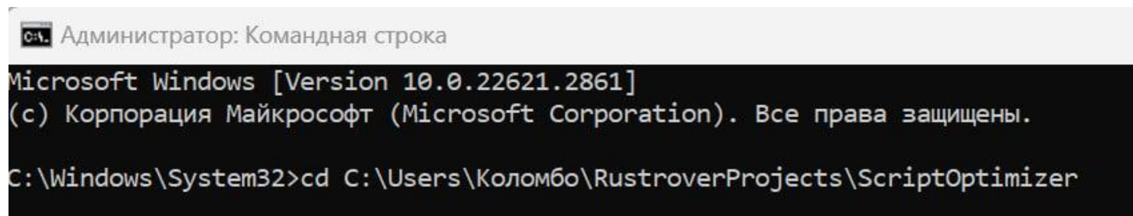


Рисунок 3.14 – Робота у cmd

Діаграма послідовностей (див. додаток Діаграма послідовності) передбачає наступні кроки. Спочатку користувач має підготувати власний запит для вирішення поставленої аналітичної задачі від кол-центру. Наприклад, оберемо приклад коли необхідно відтворити події, а саме поєднати усі дзвінки операторів і успішні офери клієнтів. (див. рисунок 3.15)

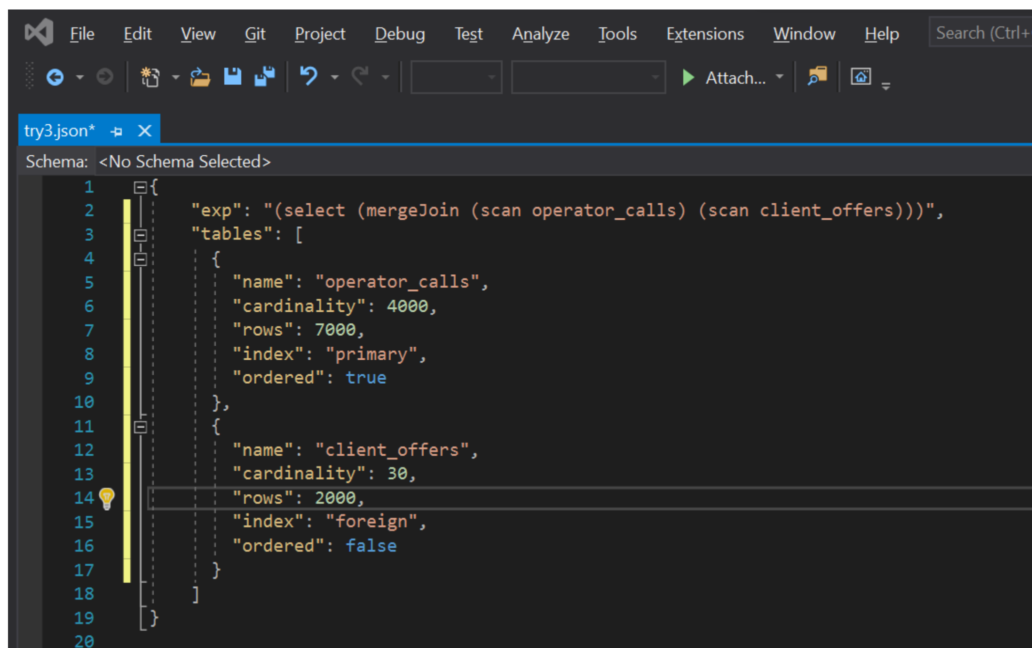


Рисунок 3.15 – Приклад тесту

Тобто при продзвоні оператором усіх клієнтів він пропонує кожному персональний офер, наприклад підключити moneybox (система для накопичення коштів на певні цілі), успішною активацією для оператора (за яку є можливість

отримати боунс) вважається та, що була підтвердежна клієнтом, а саме була виконана цільова дія. Далі слід завантажити скрипт і зберегти його за відповідним шляхом, вказавши його в додатку. Наприклад можна завантажити файл і зберегти у форматі json (див. рисунок 3.16).

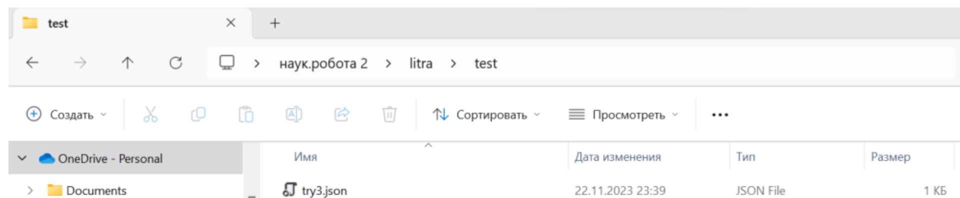


Рисунок 3.16 – Завантаження файлу

У наступному кроці парсер виконує перетворення файлу формату json у читабельний. Також створюється ексемпляр підпрограми Runner. (див. рисунок 3.17)

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
     Running `target\debug\eggplant.exe C:\\Users\\Коломбо\\Desktop\\наук.робота 2\\litra\\test\\try3.json`
Оптимізатор SQL запитів
Вхідна кардинальність = 4000
Оцінка вхідного плану = 4030
Вхідний запит:    (select (mergeJoin (scan operator_calls) (scan client_offers)))
Отримана кардинальність = 4000
Оцінка оптимізованого запиту = 4030
Оптимізований запит:   (select (hashJoin (scan operator_calls) (seek client_offers)))
```

Рисунок 3.17 – Результат роботи програми

Далі відбувається аналіз наявного плану запиту та часу його виконання і побудова нового оптимізованого плану запиту. Наприкінці користувачеві виводиться результат у консоль.

Тобто процес оптимізації скрипта проходить повний цикл (алгоритм) наведений у кресленику. А саме, спочатку користувач обирає необхідний файл, потім прописує шлях до нього у підсистемі, після чого відбувається безпосередньо зчитування файлу. Далі функція, яка шукає метадані для конкретної таблиці, що ввів користувач визначає мову та операнди що ввів користувач. Натомість підсистема визначає чи є додаткові дані, якщо так товони прикріплюються до вузлів графа з послідовним

визначенням характеристик вузлів на основі виразу мови. Потім функції для отримання кардинальності та оцінки планів визначають вхідні дані запиту. Далі відбуваються функціональні обчислення, а саме визначення вартості символу та вартості операцій, обчислення вартості плану, отримання вартості класів. Потім підключається планувальник з правилами переписування запитів, свторює новий запит, потім порівнює їх плани виконання і у разі успішної оцінки, тобто зменшегої оцінки плану та кардинальності виводить результат користувачеві у консоль (див. рисунок 3.18). Тобто дана підсистема також містить усі основні блоки для побудови відповідно інших оптимізаторів, але застосовує власні функції визначення оцінки вартості, правила переписування та виведення даних для відображення їх користувачеві.

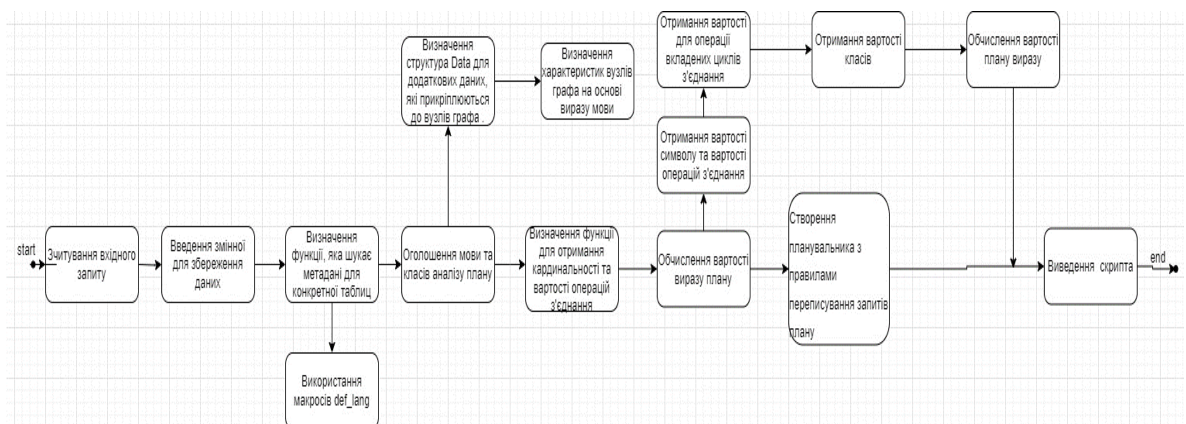


Рисунок 3.18 – Структурно-логічна схема роботи підсистеми

Діаграма класів є важливим інструментом в об'єктно-орієнтованому програмуванні, який візуалізує структуру класів та взаємодії між ними в системі. Діаграма класів дозволяє візуалізувати структуру системи шляхом показу класів та їх взаємозв'язків. Дуже зручна функція швидкого перегляду, як класи пов'язані між собою і як вони організовані. Також діаграма класів вказує на взаємодії між класами підпрограми. Це полегшує розуміння того, як класи обмінюються даними і взаємодіють між собою в рамках системи. Під час розробки програми діаграма класів допомагає у визначенні, як краще організувати код. Є невід’ємна можливість визначити без втрачання часу, які класи потрібні, як вони будуть взаємодіяти, та які

саме будуть відповідати за конкретні частини функціоналу. Діаграма класів є корисним інструментом для комунікації між членами розробницької команди. Вона допомагає всім сторонам розуміти архітектурні аспекти системи та організацію коду. Під час фази проектування діаграма класів дозволяє ретельно визначити структуру системи перед тим, як розпочати реалізацію. Це сприяє створенню більш гнучкої та розширюваної системи. Виходячи з діаграми класів, розробник може легко визначити, де можна внести зміни, покращити або оптимізувати код. Це особливо важливо під час рефакторингу програмного коду. Отже, діаграма класів є потужним інструментом для управління складністю системи та полегшення розробки та обслуговування програмного забезпечення. Саме тому, для підсистеми оптимізації скриптів було виокремлено основні елементи та покладено їх в основу діаграми класів (див. кресленник).

Проаналізувавши дану діаграму можна зробити наступні висновки. Основними класами є клас Meta, що представляє метадані для введених файлів у форматі JSON, включаючи вираз та таблиці. Клас Table представляє таблицю з метаданими, такими як ім'я, індекс, кардинальність, кількість рядків та інші властивості. Клас OptPlan визначає мову для спрощення планів виконання SQL запитів. Містить різні типи операцій, такі як SELECT, JOIN і т. д. Клас Analyze це клас аналізу плану, який визначає, як обчислювати атрибути для вузлів плану. А також видповідні їм методи, а саме функція для отримання метаданих таблиці за ім'ям, методи для обчислення кардинальності та інших параметрів для різних видів операцій, методи для обчислення вартості операцій JOIN залежно від різних умов. Методи класу CostFunc та відповідний код для обчислення вартості виразу плану. Основні структури даних це data - структура, яка містить додаткові дані, прикріплені до вузлів egraph, такі як кардинальність, кількість рядків, чи є введені дані таблицею та інші властивості. Також представлена глобальна змінна для зберігання введених метаданих. Введено основні функції для обчислення вартості різних видів операцій та символів, функція для обчислення вартості виразу плану. Присутня головна функція main, що зчитує вхідний файл, десеріалізує його та використовує отримані метадані для побудови та оптимізації плану виконання SQL запиту. Тобто якщо розглядати кожен клас та його

функцію окермо ми отримаємо наступне. Клас Meta містить атрибут tables, який представляє список об'єктів Table. Отже, між цими класами існує асоціація "1 - багато" (One-to-Many). Кожний об'єкт Table містить посилання на Meta, таким чином визначається агрегація. OptPlan використовується як перерахування для представлення мови планів виконання. Клас Analyze включає в себе аналіз плану та використовує OptPlan для визначення виду операцій. EfectiveGraph є основним представленням еквівалентних класів планів. Клас Data є додатковою інформацією, яка додається до вузлів EfectiveGraph. Analyze використовує об'єкти Data для аналізу та зберігання характеристик вузлів. EfectiveGraph використовує OptPlan для представлення еквівалентних класів у графі. CostFunc отримує EfectiveGraph для визначення вартості операцій в плані. CostFunc використовується при створенні запуску для визначення вартості правил переписування. Розширена діаграма, яка представляє схему роботи представлена на кресленнику.

У даному оптимізаторі використовуються різні алгоритми та структури даних для перетворення SQL запитів. Головним алгоритмом, який покладено в основу створення оптимізатора є генетичний алгоритм. Використовуються генетичні алгоритми для оптимізації та переписування SQL запитів. Визначена мова для представлення операторів запиту та їх аналізу. Створені правила переписування для оптимізації порядку та структури запитів. Використовується генетичний алгоритм для побудови плану виконання запиту. Генетичні алгоритми є популярним і ефективним методом оптимізації, який був інспірований еволюційним процесом в природі. Вони знаходять широке застосування в різних областях через свою універсальність та можливість роботи зі складними просторами параметрів. Генетичні алгоритми досить популярні з декількох причин, загалом тому що можуть бути застосовані до широкого спектру задач оптимізації та пошуку рішень, включаючи функціональну оптимізацію, навчання з підкріпленням, розробку графів, планування, розподілення задач та інше. Генетичні алгоритми добре справляються з оптимізацією в просторах параметрів великої розмірності та складності, де інші методи можуть виявитися менш ефективними. Велика кількість варіантів може оброблятися паралельно, що робить генетичні алгоритми ефективними в умовах, де

доступні обчислювальні ресурси для паралельної обробки. Механізми випадковості та рекомбінації дозволяють генетичним алгоритмам уникнути застрягання в локальних оптимумах та забезпечити рух по простору рішень. Генетичні алгоритми мають просту структуру та легко реалізуються. Це забезпечує їх широку доступність та застосування в різних галузях. Генетичні алгоритми можуть використовуватися для оптимізації параметрів моделей машинного навчання, вибору архітектур та гіперпараметрів. Дані алгоритми здатні здійснювати глобальний пошук, що є важливим для задач, де необхідно знайти оптимальне рішення в широкому просторі. У даному оптимізаторі генетичні алгоритми використовуються для оптимізації плану виконання SQL запитів, дозволяючи підбирати оптимальні комбінації операцій та порядок їх виконання. Генетичні алгоритми використовують концепції селекції та мутації щоб привести до покращення рішення. Такі властивості роблять їх сильним інструментом для рішення різноманітних завдань у цій сфері.

У даній підсистемі використання генетичних алгоритмів обгрунтовується наступним чином. Генетичні алгоритми мають просту структуру, що дозволяє легко реалізувати їх та модифікувати залежно від конкретних потреб. Дана програма містить визначення мови, аналіз плану та функції вартості, що робить його зрозумілим та доступним для подальших модифікацій. Також вони є універсальним методом, який може бути успішно використаний для рішення різноманітних задач оптимізації, можуть адаптуватися до складних структур даних та враховувати багато параметрів при пошуку оптимального рішення. Важливо що вони мають вбудовані механізми для глобального пошуку, що є важливим для задач оптимізації в просторі планів виконання SQL запитів. Також використовують випадковість та рекомбінацію, щоб уникнути застрягання в локальних оптимумах. Оскільки генетичні алгоритми можуть пристосовуватися до різних сфер, вони вдало використовуються для оптимізації планів виконання SQL запитів, де потрібно вибрати оптимальні комбінації операцій та порядок їх виконання. Загалом, вибір генетичного алгоритму в основу підсистеми спростить пошук оптимального рішення.

3.5 Дослідження ефективності. Опис метрик

У сфері оптимізації SQL запитів та виконання скриптів, важливим аспектом є вимірювання ефективності та якості виконання запитів. Це досягається за допомогою різноманітних метрик та параметрів систем, які оцінюють якість та продуктивність виконання запитів. Введення метрик дозволяє здійснювати об'єктивний аналіз ефективності запитів та вживання оптимізаційних заходів. Наприклад, шляхом вимірювання часу виконання, кількості оброблених записів та інших параметрів, можна чітко оцінити безпосередній вплив на продуктивність системи. Метрики стають ключовим інструментом для визначення якості оптимізації. Вони дозволяють порівнювати вихідні та отримані значення, визначаючи наскільки ефективно оптимізатор впливає на продуктивність та якість обробки запитів. Метрики допомагають приймати обгрунтовані рішення з точки зору оптимізації. Аналіз вхідних та вихідних параметрів дозволяє інженерам баз даних та розробникам чітко розуміти, які аспекти системи потребують уваги та вдосконалення. Деякі найпоказовіші метрики, такі як час виконання оптимізованого запиту, зменшення вхідної кардинальності та поліпшення оцінки вхідного плану, слугують ключовими показниками ефективності оптимізації. Їх використання полегшує не лише вимірювання результатів, але і забезпечує можливість швидкого виявлення найбільш вигідних стратегій оптимізації. Загалом, введення метрик у контексті оптимізації запитів стає кроком до більш ефективної, якісної та швидкої обробки даних.

Загальноприйнято вважати основними метриками у сфері оптимізації запитів наступні. Час виконання вимірюється для оцінки продуктивності оптимізованого запиту. Зменшення часу виконання свідчить про покращення продуктивності. Час виконання вимірюється в мікросекундах (мкс). Наступним параметром є кардинальність - кількість записів, яку оптимізатор передбачає вихідним результатом виконання SQL запиту без оптимізації. Також не менш важливим параметром є оцінка вхідного плану. Одиниці виміру безрозмірні. Ця метрика оцінює якість вхідного плану виконання запиту перед обробкою. Використовується для порівняння якості планів перед та після обробки. Зазначені вище метрики також використовуються в

контексті обробки запитів у даній роботі. Наприклад, порівнюючи вхідну кардинальність та оцінку вхідного плану з отриманими після оптимізації метриками, можна визначити ефективність та якість роботи підпрограми. Час виконання отриманого запиту слугує показником продуктивності, а отримана кардинальність вказує на обсяг даних у вихідному результаті оптимізованого запиту. Вимірювання та аналіз вказаних метрик дозволяють здійснити ефективну обробку запитів від аналітиків, забезпечуючи одночасне вдосконалення продуктивності та якості виконання.

Значну увагу варто приділити правилам, які були використані під час написання підсистеми.

В основному це правила переписування (rewrite rules), які грають важливу роль у визначенні стратегії оптимізації планів виконання. Правила переписування визначають трансформації, які можна застосовувати до виразів плану з метою поліпшення їх продуктивності або інших характеристик. Присутні правила переписування порядку з'єднань: (заміна (hashJoin ?a ?b) на (hashJoin ?b ?a)) і т.д. Ці правила спрямовані на зміну порядку виконання операцій для поліпшення продуктивності та вибору більш ефективних методів обробки даних. Правила переписування операцій з'єднань: hash-join-merge-join і т.д. Ці правила спрямовані на вибір найліпшого методу з'єднання таблиць, що може впливати на продуктивність запиту. Застосовано різноманітні функції для визначення вартості операцій з'єднання. Вартість операцій використовується для визначення, наскільки ефективним буде використання певного методу в конкретному контексті, що допомагає вибрати найкращий варіант. Правила переписування є ключовим елементом у здійсненні автоматичної оптимізації, дозволяючи системі вибирати певне представлення запиту для виконання. Вони важливі для підвищення ефективності виконання скриптів, зменшення часу виконання та забезпечення оптимального вибору планів виконання.

Експериментальне дослідження проводилось на персональному комп’ютері з наступними характеристиками : Intel Cor i7 8th Gen CPU @ 3.20GHz.

На рисунку нижче (див.рисунок 3.19) подано код скрипта у форматі json. Замовник у вигляді кол-центру поставив технічне завдання на аналітиків з метою

впровадження великого проєкту по зворотньому зв'язку з клієнтами, а саме чатам. Отже, надано три таблиці з даними про кожен чат, операторів, що консультували клієнтів та власне кожне повідомлення в чаті. У кожній з таблиць вказано кардинальність, кількість рядків та чи впорядковані дані. У базі даних впорядковані дані — це дані, які зберігаються в певному порядку або мають якийсь внутрішній логічний порядок. Цей порядок може бути встановлений за певними правилами або визначатися значенням конкретного атрибуту. Впорядковані дані можуть бути важливі в різних сценаріях, таких як оптимізація запитів, підтримка певного порядку виведення результатів, полегшення виконання певних операцій тощо.

```json
{
  "expression": "(select (hashJoin (hashJoin (scan chat_messages) (scan chat_conversations)) (chat_agents)))",
  "tables": [
    {
      "name": "chat_conversations",
      "cardinality": 880,
      "rows": 22000,
      "index": "primary",
      "ordered": true
    },
    {
      "name": "chat_messages",
      "cardinality": 4050,
      "rows": 190000,
      "index": "foreign",
      "ordered": false
    },
    {
      "name": "chat_agents",
      "cardinality": 120,
      "rows": 122000,
      "index": "foreign",
      "ordered": false
    }
  ]
}
```

Рисунок 3.19 – Фрагмент запиту

Цей фрагмент представляє операцію сканування таблиці трьох таблиць. Тобто, відбувається проходження всіх записів у зазначених таблицях. Далі відбувається операція хеш-з'єднання (hash join) між результатами двох попередніх сканувань таблиць. Хеш-з'єднання - це один з методів об'єднання двох таблиць, використовуючи хеш-функції для швидкого пошуку співпадінь між значеннями в об'єднуваних стовпцях. Далі відбуваєтсья останній етап, де результати попереднього хеш-з'єднання об'єднуються з таблицею за допомогою ще одного хеш-з'єднання. В результаті

аналітики отримують об'єднану таблицю, яка містить дані з усіх трьох вихідних таблиць, з'єднаних відповідно до логіки хеш-з'єднань і результат фільтрується за допомогою операції select.

Після цього було запущено оптимізатор, який виконанав зчитування скрипта і вивів дані з результатами для аналітиків у консоль (див. рисунок 3.20).

```
nt -- C:\\Users\\Коломбо\\Desktop\\наук.робота 2\\litra\\test\\test.json
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target\debug\eggplant.exe C:\\Users\\Коломбо\\Desktop\\наук.робота 2\\litra\\test\\test.json`
Оптимізатор SQL запитів
Вхідна кардинальність = 8100
Оцінка вхідного плану = 9112
Вхідний запит:    (select (hashJoin (hashJoin (scan chat_messages) (scan chat_conversations)) chat_agents))
Отримана кардинальність = 4930
Оцінка оптимізованого запиту = 5930
Оптимізований запит:   (select (mergeJoin (nestedLoopsJoin (seek chat_conversations) chat_agents) (seek chat_messages)))
```

Рисунок 3.20 – Результат роботи

З результату роботи програми видно, що оригінальний запит має вхідну кардинальність 8100 та оцінку вхідного плану 9112. Власне вхідний запит це (select (hashJoin (hashJoin (scan chat_messages) (scan chat_conversations)) chat_agents)) Також оптимізований запит має отриману кардинальність 4930 та оцінку оптимізованого запиту 5930. На виході маємо переписаний оптимізований запит, а саме (select (mergeJoin (nestedLoopsJoin (seek chat_conversations) chat_agents) (seek chat_messages))) Отже, оптимізатор використовує різні стратегії для об'єднання таблиць, такі як hashJoin, nestedLoopsJoin, і mergeJoin. Конкретно в цьому випадку оптимізатор вирішив застосувати mergeJoin замість двох вкладених hashJoin. Кардинальність оптимізованого запиту (4930) є значно меншою, ніж у вхідному запиті (8100). Це свідчить про те, що оптимізований план більш ефективний, оскільки він обробляє менше записів для отримання результатів. Оцінка плану вказує на те, що оптимізований план вважається менш витратним з точки зору ресурсів в порівнянні з вихідним планом. Загалом можна зробити висновок, що оптимізатор виявив більш оптимальний план виконання запиту, який за його оцінкою, повинен працювати швидше та ефективніше.

Висновки до розділу 3

Даний розділ присвячений розробці підсистеми оптимізації скриптів для вирішення аналітичних задач кол-центру і представляє собою ключовий розділ дослідження, який визначає методологічний та технічний фундамент оптимізації даних. Було розглянуто процес розробки алгоритму для оптимізації скриптів, спеціально адаптованих для розв'язання аналітичних завдань кол-центру. Цей алгоритм став ключовим компонентом підсистеми, що дозволяє покращити ефективність обробки даних та оптимізувати час виконання скриптів. Виконано визначення вимог та специфікацій, а саме було враховано специфічні потреби та вимоги кол-центру. Застосування діаграми прецедентів дозволило чітко визначити функціональність підсистеми та її взаємодію з іншими компонентами. Також було визначено архітектуру програмного забезпечення, де за допомогою діаграми класів була окреслена загальна структура системи. Це дозволило створити гнучку та розширювану архітектуру, яка забезпечує легкість утримання та можливість майбутнього розширення. Далі проведено дослідження ефективності розробленої підсистеми. Результати цього дослідження демонструють позитивні зміни у продуктивності, що підтверджується аналізом виконаних запитів та їх часовою ефективністю. Наприкінці підсумовано досягнуті результати. Розроблена підсистема оптимізації скриптів виявилася ефективною та відповідає вимогам кол-центру. Результати її роботи вказують на значний прогрес у покращенні продуктивності аналітичних завдань, що є важливим внеском у сферу обробки даних у контексті кол-центру.

4 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

4.1 Опис ідеї проєкту

Назва проєкту: SQL Optimizer Plus

SQL Optimizer Plus - це стартап, який надає підсистему оптимізації SQL запитів для ефективного вирішення аналітичних задач кол-центрів. Даний продукт використовує передові технології оптимізації, базуючись на інтегрованому рішенні, що використовує мову програмування Rust та фреймворк egg. Введена автоматична оптимізація SQL запитів, тобто використання передових алгоритмів оптимізації для покращення продуктивності виконання SQL запитів. Також використовується аналіз та використання метаданих для ефективної оптимізації запитів в реальному часі. Підтримка популярних систем управління базами даних, таких як PostgreSQL, MySQL, та Microsoft SQL Server. Забезпечення інструментів моніторингу для визначення та вирішення ефективності SQL запитів.

4.2 Технологічний аудит

Мови програмування: Rust, SQL.

Фреймворки та бібліотеки: egg, serde_json.

Інструменти розробки: std::fs, std::env.

Безпека: Застосування стандартів безпеки та шифрування для збереження конфіденційності даних.

4.3 Аналіз ринкових можливостей запуску стартап-проєкту

Цільовий ринок: організації з кол-центрами, що активно використовують аналітику для оптимізації роботи та прийняття рішень.

Конкурентні переваги. Висока продуктивність, а саме використання передових алгоритмів та метаданих для досягнення оптимальної продуктивності. Гнучкість і адаптивність, а саме здатність реагувати на зміни в аналітичних завданнях та впроваджувати нові методи оптимізації. Інтеграція з різними СУБД, а саме забезпечення сумісності з різними системами управління базами даних. Монетизація, а саме модель підписки, тобто збір плати за доступ до підсистеми оптимізації та покращених функцій. Навчання та консультації, тобто надання навчання та консультацій з оптимізації SQL запитів для підприємств.

4.4 Розроблення ринкової стратегії проєкту

Етапи впровадження. Локальний ринок (перші 3 місяці). Залучення перших клієнтів та отримання відгуків для вдосконалення продукту. Розширення функціональності (наступні 6 місяців). Додавання нових функцій та опцій для розширення здатностей підсистеми оптимізації. Глобальний ринок (1-2 роки). Вихід на міжнародний ринок, укладання партнерських угод та адаптація до різних мов та регіональних особливостей.

Маркетинг та реклама. Реклама через соціальні мережі та інтернет-ресурси. Участь у конференціях. Активна участь у відповідних галузевих заходах для залучення уваги та партнерів.

Партнерства. Укладання партнерських угод з постачальниками та іншими підприємствами, щоб розширити обсяг користувачів.

Підтримка користувачів. Забезпечення цілодобової підтримки для користувачів з різних часових поясів.

Навчання та ресурси. Надання навчальних матеріалів та ресурсів для оптимального використання підсистеми оптимізації.

Аналіз відгуків. Регулярний аналіз відгуків користувачів для вдосконалення продукту.

Метрики продуктивності. Систематичне відстеження метрик ефективності оптимізації та виправлення недоліків.

SQL Optimizer Plus розвиватиметься як інноваційне рішення для ефективної оптимізації SQL запитів, спрямоване на забезпечення високої продуктивності та задоволення потреб кол-центрів. А також аналітики матимуть змогу придбати зручний додаток для оптимізації скриптів, який зменшуватиме навантаження на співробітників .

Висновки та перспектива подальших досліджень

Перспективу подальших досліджень у даній галузі можна оцінити наступним чином. Зорієнтування подальшої розробки має бути у напрямку розширення спектру підтримуваних мов та граматик, що дозволить оптимізатору більш гнучко пристосовуватися до різноманітних вимог різних програм і середовищ. Використання ефективних методів оптимізації для специфічних мов може призвести до значних покращень у продуктивності. Впровадження та розвиток методів машинного навчання в області оптимізації може відкрити нові горизонти. Використання алгоритмів навчання з підкріпленням для автоматичного вибору оптимальних стратегій та адаптації до змінних умов зробить оптимізатор більш гнучким та ефективним. Оптимізатор, який розуміє та ефективно працює в розподілених системах, виявиться критичним для застосувань, які використовують хмарні обчислення або в розподілених мережах. Розробка алгоритмів, спрямованих на оптимізацію в умовах розподіленості, підійме продуктивність на новий рівень. Створення інтерактивних інтерфейсів для користувачів, які дозволяють маніпулювати та контролювати процес оптимізації в реальному часі, зробить оптимізацію більш доступною та інтуїтивно зрозумілою для широкого кола користувачів. Розробка нових методів обчислення вартості виразу плану та аналізу його характеристик підвищить точність оптимізації. Інтеграція алгоритмів для визначення кардинальності, розмірів та інших параметрів зробить оптимізатор більш вдосконаленим та адаптованим до різних умов. Ці напрями досліджень визначать новий етап в розвитку оптимізатора та його потенціалу для вдосконалення продуктивності та швидкодії виконання запитів.

Створення високоефективного та інноваційного оптимізатора для ринку даних має значний вплив на декілька аспектів індустрії. Тобто застосування інноваційного оптимізатора призведе до значного збільшення продуктивності виконання даних, що особливо важливо в умовах великих обсягів інформації. Оптимізатор, що добре пристосовується до різних умов та розподілених середовищ, може допомогти економити ресурси, такі як час обчислення та потужність обчислювальних систем. Застосування оптимізатора може полегшити роботу з великими обсягами даних та забезпечити ефективну обробку інформації в реальному часі, навіть в умовах високих навантажень. Високоефективний оптимізатор стане стимулом для розробки нових застосунків та сервісів, що використовують дані, включаючи штучний інтелект, та інші області. Компанії, які використовують передові технології оптимізації, можуть здобути конкурентну перевагу на ринку, надаючи клієнтам швидкі та ефективні рішення для обробки та аналізу даних. Успіх розробки сприятиме розвитку екосистеми інструментів для роботи з даними, забезпечуючи нові можливості та інтеграції. Успішна реалізація та впровадження такого оптимізатора збільшить зацікавленість інвесторів у проекти, пов'язані з оптимізацією та обробкою даних. Загалом, створення оптимізатора та вищеописані подальші впровадження визначать нові стандарти ефективності та продуктивності в обробці та аналізі даних, що відкриє шлях для розвитку в цій стратегічно важливій області.

ВИСНОВКИ

Дана робота присвячена розробці підсистеми оптимізації SQL скриптів для вирішення аналітичних задач кол-центру для підвищення продуктивності обробки даних та зменшення кардинальності та оцінки плану виконання SQL скриптів. Під час виконання роботи було вивчено та проаналізовано роботу кол-центту для визначення особливостей аналітичних задач. Отже, особливості аналітичних задач кол-центру полягають у вивченні та оптимізації великого обсягу даних, спрямованих на вдосконалення ефективності обслуговування клієнтів. Це включає аналіз тенденцій дзвінків, розпізнавання проблемних ситуацій, вдосконалення роботи операторів та стратегій взаємодії з клієнтами для підвищення якості обслуговування. Виокремлення ролі SQL у вирішенні аналітичних задач, ідентифікація проблем та визначення впливу оптимізації на продуктивність системи стали важливою основою для подальших досліджень. Розглянуто та проаналізовано основні відмінності SQL від інших мов для визначення особливостей їх оптимізації. Таким чином, скрипти функціонують шляхом інтерпретації коду без попередньої компіляції, виконуючи інструкції послідовно зверху вниз, використовуючи змінні та типи даних, керуючи потоком за допомогою конструкцій умов та циклів, розділяючи код на функції для полегшення читання та повторного використання, і забезпечуючи обробку помилок для забезпечення стабільності програми. Розглянуто основні методи та методики оптимізації SQL скриптів. Оптимізація SQL скриптів включає в себе використання ефективних методів та методик, таких як індексація, оптимізація запитів, використання правильних типів даних, нормалізація бази даних та кешування для підвищення продуктивності та швидкодії. Розробка підсистеми оптимізації виявилася ключовим етапом роботи, де розробка алгоритму для оптимізації SQL скриптів та архітектура програмного забезпечення були уважно вивчені та описані. Використання мови програмування RUST та бібліотеки EGG стали обґрунтованим вибором. Стартап-проєкт дозволив оцінити роботу комплексним поглядом на можливості впровадження розробленої системи. Опис ідеї проєкту, технологічний аудит та аналіз ринкових можливостей забезпечили раціональне планування запуску стартапу, а

розроблення ринкової стратегії забезпечило підготовку проєкту до успішної реалізації. Отже, розроблено підсистему оптимізації SQL скриптів та надано рекомендації щодо її впровадження в роботу кол-центру для підвищення продуктивності обробки даних та зменшення кардинальності та оцінки плану виконання SQL скриптів. Результати досліджень є внеском у розвиток сфери обробки даних та управління кол-центрами.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Oleksandr Rolik, Kseniia Ulianytska, Maryna Khmeliuk, Volodymyr Khmeliuk, Uliana Kolomiiets. Increase Efficiency of Relational Databases Using Instruments of Second Normal Form. URL: https://ieeexplore.ieee.org/document/9678605.
2. Stack Exchange для професійних програмістів. URL: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-all-respondents (дата звернення 12.11.2023).
3. Кривий Р.А. Методична розробка за темою: «Побудова SQL запитів». Дніпропетровськ: Державний вищий навчальний заклад «Дніпропетровський транспортно-економічний коледж», 2016.
4. Allen G. Taylor. SQL For Dummies 8th Edition. E-Book: 8th edition, 2013. 480p.
5. Scott L. Sql vs. Java what's the difference, and which is better. : Article, 2023.
6. Yasin N. Silva, Isadora Almeida, Michell Queiroz. SQL: From Traditional Databases to Big data. Conference: the 47th ACM Technical Symposium, 2016.
7. A. Sattar, T. Lorenzen, and K. Nallamaddi. Incorporating nosql into a database course. ACM Inroads, 4(2):50–53, June 2013.
8. Fabio Duarte. Amount of Data Created Daily. URL: https://explodingtopics.com/blog/data-generated-per-day.
9. Rosenthal, D. Reiner. An architecture for query optimization. ACM SI: ~'1OD Conf., Orlando, Florida, June 1982. ROSE82b. [47-48]
10. Mitchell, M. An introduction to genetic algorithms. USA: The MIT Press, 1998. 221 p.
11. M. Sinha and S.V. Chande. Query optimization using genetic algorithms. Journal: Research Journal of Information Technology, 2, 2010. 139-144p.
12. Pandey A. Dynamic programming solution for query optimization in homogeneous distributed databases. Article: DOI:10.13140/RG.2.2.27334, 2019. URL:https://www.researchgate.net/publication/332632303_Dynamic_Programming_Solution_for_Query_Optimization_in_Homogeneous_Distributed_Databases

13. Білоус Р., Крилов Є., Анікін В. Методи оптимізації запитів розподілених БД. Адаптивні системи автоматичного управління, 2021. 3–11 с. URL: https://doi.org/10.20535/1560-8956.39.2021.247364
14. Peter Gulutzan, Trudy Pelzer. SQL performance tuning. E-Book: ISBN: 0-201-79169-2, 2002. 528 p.
15. Цирульник М. Як і для чого вивчати мову програмування Rust. Стаття: DOU 42770, 2023. URL: https://dou.ua/forums/topic/42770/.
16. SpaceLab. Чому мова програмування Rust така популярна? URL: https://spacelab.ua/articles/chomu-mova-programuvannya-rust-taka-populyarna/
17. Стусь О.В. Математична логіка та теорія алгоритмів. навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2017. 150 с.
18. Кузьменко І.М. Теорія графів. Київ : КПІ ім. Ігоря Сікорського, 2020. 71 с.
19. Official Egg library documentation. URL: https://egraphs-good.github.io.
20. M. L. Rupley. Introduction to query processing and optimization. USA: techreport: Indiana University, 2008, 15 p.
21. Джордан Морроу. Як витягти з даних максимум. Навички аналітики для неспеціалістів. Електронна книга: видавництво: Альпіна Діджітал, 2022. 250 с.
22. Habimana, J. Query optimization techniques – tips for writing efficient and faster SQL queries. International Journal of Scientific & Technology Research, 4(10), 2015. 22–26 p.
23. M. Sharma. Query optimization using SQL transformations. International Journal of IT, Engineering and Applied Sciences Research, vol. 1, no. 1, 2012. pp. 100–104.
24. Робін Дж. Вілсон. Ведення в теорію графів. Електронна книга: видавництво Діалектика, 2012. 240 с.
25. James Rumbaugh, Ivar Jacobson, Grady Booch. The unified modeling language reference manual. Addison Wesley Longman Inc. ISBN 0-201-3099, 1999. 584 p.
26. Тернопільський національний економічний універистет. Моделювання бізнес-процесів та архітектури програмного забезпечення. Тернопіль: науч. посібник, 2015. 105 с.
27. Alan Beaulieu. Learning SQL. E-Book, 2007. 337 p.