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

Інститут прикладного системного аналізу

Кафедра системного проектування

До захисту допущено:

В. о. завідувача кафедри

_____Анатолій Петренко

«___» _____20__ р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Інтелектуальні сервіс-орієнтовані розподілені обчислення»

спеціальності 122 «Комп'ютерні науки»

на тему: «Застосування нейронних мереж для оптимізації ефективності запитів до СУБД»

Виконав:
студент IV курсу, групи ДА-71
Загородній Дмитро Олександрович _____

Керівник:
Старший викладач, Булах Б. В. _____

Консультант з економічного роділу:
к.е.н, доцент Рощина Надія Василівна _____

Консультант з нормоконтролю:
к.т.н., доц. Кирюша Б. А. _____

Рецензент:
к.т.н., доц. Недашківська Н. І. _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент (-ка) _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Інститут прикладного системного аналізу
Кафедра системного проектування

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 «Комп'ютерні науки»

Освітньо-професійна програма «Інтелектуальні сервіс-орієнтовані розподілені обчислення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Анатолій Петренко
«___» _____20__ р.

ЗАВДАННЯ
на дипломну роботу студенту
Загородньому Дмитру Олександровичу

1. Тема роботи «Застосування нейронних мереж для оптимізації ефективності запитів до СУБД», керівник роботи Булах Богдан Вікторович, к.т.н. Доцент кафедри, затверджені наказом по університету від «___» _____ 20__ р. №_____
2. Термін подання студентом роботи
3. Вихідні дані до роботи
4. Зміст роботи
5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний.	к.е.н, доцент Рощина Надія Василівна		

7. Дата видачі завдання 27 березня 2021

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1.	Вивчення літератури за темою роботи.	12.04.2021	
2.	Підготовка першого розділу.	20.04.2021	
3.	Підготовка другого розділу.	29.04.2021	
4.	Розробка програмного продукту.	8.05.2021	
5.	Підготовка третього розділу	20.05.2021	
6.	Підготовка економічної частини	26.05.2021	
7.	Оформлення розділів відповідно до нормоконтролю.	30.05.2021	
8.	Підготовка презентації доповіді.	1.06.2021	
9.	Оформлення дипломної роботи.	5.06.2021	

Студент Дмитро Загородній

Керівник Богдан Булах

РЕФЕРАТ

Дипломна робота: 88 ст., 39 рис, 8 табл., 1 дод., 22 джерел.

БАЗА ДАНИХ. ПЛАН ЗАПИТУ. НЕЙРОННА МЕРЕЖА. НАВЧАННЯ З ПІДКРІПЛЕННЯМ. СИСТЕМА УПРАВЛІННЯ БАЗОЮ ДАНИХ.

Тема: Застосування нейронних мереж для оптимізації ефективності запитів до СУБД.

Об’єкт дослідження: Сучачні СУБД, план запиту та оптимізація плану запиту.

Предмет дослідження: Використання машинного навчання, а саме нейронних мереж для підвищення ефективності та оптимізації створення плану запиту.

Мета роботи: Дослідити сучасні СУБД з акцентом на механізми оптимізації запитів. Дослідити можливості використання нейронних мереж для задачі оптимізації запитів. Визначити чи ефективно використовувати нейронні мережі для цієї задачі. Розробити додаток що зможе продемонструвати ефективність використання нейронних мереж для оптимізації запитів.

Було розроблено додаток що використовує модель навчання з підкріпленням та оптимізує порядок створення з’єднань у запиті. Тестами було показана ефективність використання нейронних мереж для вирішення задачі оптимізації запитів.

ABSTRACT

Thesis: pages, 39 figures, 8 tables, 1 appendices, 22 sources.

DATABASE. QUERY PLAN. NEURAL NETWORK. REINFORCED LEARNING. DATABASE MANAGEMENT SYSTEM.

Topic: Using neural networks to optimize DBMS queries performance.

Object of research: Use machine learning, namely neural networks to increase efficiency and optimize the creation of a query plan.

Subject of research: Use machine learning, namely neural networks to increase efficiency and optimize the creation of a query plan.

Purpose: Investigate modern databases with an emphasis on query optimization mechanisms. Investigate the possibilities of using neural networks for the task of query optimization. Determine whether to use neural networks effectively for this task. Develop an application that can demonstrate the effectiveness of the use of neural networks to optimize queries.

An application has been developed that uses a reinforced learning model and optimizes the order in which requests are created. Tests have shown the effectiveness of using neural networks to solve the problem of query optimization.

Зміст

ВСТУП

Основні сучасні інформаційні технології базуються на концепції зберігання даних, згідно якої усі дані мають бути організовані у бази даних, так щоб вони могли відображати реальний світ та його мінливість, а також бути зручними та задовольняти усім інформаційним потребам користувачів. Для створення і функціонування таких баз даних існують спеціальні програмні комплекси, які називаються системами управління базами даних (СУБД).

Усі підприємства та установи, починаючи від банків, яким необхідно зберігати інформацію про клієнтів, їх рахунки та транзакції, та закінчуючи лікарнями, для яких необхідна база пацієнтів, хвороб та іншого, використовують сучасні бази даних. Усі сучасні інформаційні системи використовують сучасні СУБД для зберігання своїх даних. Наразі дані та їх структура з кожним роком стають все складніші та складніші, кількість користувачів стрімко зростає, а питання оптимізації операцій з даними стає все більш актуальним. Через це і постає проблема оптимізації СУБД. У даній роботі будуть розглянуті способи оптимізації запитів до сучасних СУБД, а саме методи використання нейронних мереж для оптимізації запитів.

Оптимізація запитів — це 1) функція СУБД, яка виконує пошук оптимального плана виконання запиту з усіх можливих для заданого запиту, 2) процес зміни запиту та/або структури БД в цілях зменшення використання розрахункових ресурсів при виконанні запиту. Один і той же результат може бути отриманий СУБД різноманітними способами (планами виконання запиту), які можуть суттєво відрізнятися як за затратами ресурсів, так і за часом виконання. Головним завданням оптимізації запитів є системні скорочення ресурсів, необхідних для виконання запиту, що в кінцевому підсумку надає користувачеві результати за менший проміжок часу, що робить додаток комфортнішим для користувача; надає можливість програмному додатку обслуговувати більше запитів за одиницю часу, оскільки кожен запит

займає менше часу, ніж неоптимізовані запити; зменшує навантаження на обладнання і дозволяє серверу працювати більш ефективно.

Наразі є не мало досліджень про можливості оптимізації баз даних з використанням машинного навчання, починаючи від лінійної регресії, яка використовується з метою більш точного розрахунку використання ресурсів кожним з планів, закінчуючи використанням нейронних мереж з дослідженням їх використання для заміни сучасних методів оптимізації плану (грубої сили, динамічне програмування, генетичний алгоритм). У порівнянні с сучасними методами оптимізації запитів, використання нейронних мереж дозволяє пришвидшити роботу пошуку плану запиту до 10 раз швидше.

Метою дипломної роботи є вияснення ефективності використання нейронних мереж для оптимізації СУБД, а саме оптимізації запитів до СУБД.

РОЗДІЛ 1. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ ПРО СУБД ТА МЕХАНІЗМИ ОПТИМІЗАЦІЇ ЗАПИТУ.

1.1. Основні теоретичні відомості про СУБД.

Система управління базами даних (СУБД) – набір програмних та лінгвістичних засобів для загального чи спеціального призначення, що забезпечують управлінням, створенням та використанням баз даних. В загальному розумінні СУБД – комплекс програм, які дозволяють створювати базу даних та маніпулювати даними, тобто вставляти, оновляти, видаляти та вибирати. СУБД також має забезпечувати безпеку та надійність зберігання даних, а також їх цілісність. СУБД також надає можливості для адміністрування БД [1].

Основою архітектури СУБД є трирівнева система організації ANSI-SPARC, при цій архітектурі існує незалежний рівень, який необхідний для ізоляції програми від моделі представлення даних на низькому рівні. Рівні більш детально:

- зовнішній – цей рівень є представленням бази даних з боку користувача, тобто інтерфейс;
- концептуальний – програмне представлення бази даних, цей рівень описує збережені у БД дані, а також зв’язки між ними. Саме цей рівень підтримує різноманітні зовнішні представлення, а також підтримується внутрішнім рівнем;
- внутрішній – являє собою фізичне представлення бази даних у пам’яті комп’ютера [1].

Є два рівні незалежності:

- логічна незалежність – незалежність зовнішніх моделей та інтерфейсів від змін, що можуть бути внесені у концептуальну модель;
- фізична незалежність – захищеність концептуального рівня від можливих змін у зовнішньому рівні [1].

Основними функціями СУБД є:

- використання зовнішньої пам’яті та управління даними у ній;
- використання оперативної пам’яті та управління даними у ній;
- ведення журналу змін, резервне копіювання та відновлення даних після збоїв;
- підтримка мови запитів БД (мова маніпулювання даними) [1].

Основними характеристиками СУБД є:

- контроль надлишковості даних;
- цілісність даних, та підтримка цілісності;
- несуперечливість даних у БД;
- цілісність даних може бути описана з використанням обмежень;
- прикладні програми є незалежними від даних;
- можливість спільного використання даних;
- досить високий рівень безпеки[1].

Склад СУБД.

Сучасні СУБД містять наступні компоненти:

- ядро, цей компонент відповідає за управління даними у зовнішній та оперативній пам’яті та за ведення журналу змін;
- процесор мови бази даних – забезпечує оптимізацію запитів на створення, витягування та зміну даних, а також створення машинно-незалежного внутрішнього коду для виконання;
- підсистему підтримки часу виконання, котра інтерпретує програми маніпуляції с даними, які створюють інтерфейс користувача с СУБД;
- сервісні програми – зовнішні утиліти, які надають додаткові можливості по обслуговуванню інформаційної системи [1].

Класифікації СУБД.

По моделі даних, приклади:

- реляційні;
- сітьові;
- ієрархічні;
- об’єктно-реляційні;
- об’єктно-орієнтовані.

По степені розподіленості:

- локальні СУБД (усі частини СУБД розміщуються на одному комп’ютері);
- розподілені СУБД (різні частини СУБД можуть розміщуватися не лише на одному, але і на більшій кількості комп’ютерів).

По методу доступу до БД:

- **файл серверні**. В файл-серверних СУБД файли даних розташовуються централізовано на файл-сервері. СУБД розташовується на кожному клієнтському комп’ютері. Доступ СУБД до даних може виконуватись через локальну мережу. Синхронізація читань та оновлень може виконуваись шляхом файлових блокувань. Перевагою цієї архітектури є знижене навантаження на процесор файлового серверу. Недоліки: ускладнення або неможливість централізованого керування, можливе високе навантаження локальної мережі, ускладнення або неможливість забезпечення таких важливих характеристик, як висока безпека, висока доступність та висока надійність. Застосовуються частіше всього в локальних додатках, які користуються функціями управління баз даних, в системах з низькою інтенсивністю обробки даних та низькими піковими навантаженнями на БД. На даний момент файл-серверна технологія є застарілою, а її використання в крупних інформаційних системах – недолік. Приклади: Microsoft Access, FoxPro, Visual FoxProX, dBase [1];

- **клієнт серверні**. Клієнт-серверна СУБД розташовується на сервері разом з БД та виконує доступ до БД безпосередньо в монопольному режимі. Усі клієнтські запити на обробку даних виконуються на клієнт-серверній СУБД централізовано. Недоліком клієнт-серверних СУБД є підвищені потреби до серверу. Перевагами є зручність централізованого управління, потенційно більш низька загрузка локальної мережі, зручність забезпечення таких важливих характеристик, як висока безпека, висока доступність та висока надійність. Приклади таких БД: Oracle Database, Firebird, Interbase, IBM DB2, Informix, MS SQL Server, Sybase Adaptive Server Enterprise, PostgreSQL, MySQL, Caché[1];
- **вбудовані**. Вбудована СУБД, котра може поставлятися як зіставна частина декого програмного продукту, не вимагаючи процедури самостійної установки. Вбудована СУБД необхідна для локального зберігання даних додатку і не розрахована для колективного використання у мережі. Фізично вбудована СУБД зазвичай реалізована у вигляді бібліотеки, яку можна підключити . Доступ до даних зі сторони додатку може відбуватися через SQL або через спеціальні програмні інтерфейси.

  Приклади: SQLite, OpenEdge, BerkeleyDB, Microsoft SQL Server Compact [1].

Стратегії роботи з зовнішньою пам’яттю.

СУБД з безпосереднім записом.

У таких СУБД усі змінені блоки даних одразу записуються у зовнішню пам’ять при отриманні сигналу підтвердження будь-якої транзакції. Така стратегія використовується лише при високій ефективності зовнішньої пам’яті [1].

СУБД з відкладеним записом.

В таких СУБД зміни акумулюються у буферах зовнішньої пам’яті до настання будь-якої з наступних подій:

- контрольна точка;
- брак простору у зовнішній пам’яті, відведеного під журнал. СУБД створює контрольну точку та починає писати журнал спочатку, стираючи попередню інформацію;
- зупинка СУБД чекає, коли весь вміст всіх буферів зовнішньої пам’яті буде перенесено у зовнішню пам’ять, після чого робить відмітки, що зупинка бази даних виконана коректно;
- брак оперативної пам’яті для буферів зовнішьої.

Така стратегія дозволяє уникнути частого обміну з зовнішньою пам’яттю та значно збільшити ефективність роботи СУБД [1].

1.2. План виконання запиту.

План запиту — впорядкований набір кроків, які необхідні для доступу до даних в СУБД SQL. Так як мова SQL є декларативною, то зазвичай існує багато альтернативних способів виконання заданого користувачем запиту з широким діапазоном продуктивності. Післлля того як запит потрапляє в базу даних, оптимізатор запитів виконує оцінку планів запиту і повертає той, який вважається найкращим. Користувачам та адміністраторам баз даних часто доводиться перевіряти та налаштовувати плани, які були створені оптимізатором. Це виконується для підвищення продуктивності, так як оптимізатори запитів не є досконалими [2].

План в цілому розділяється на дві стадії:

- Вибірка результатів;
- Сортування і групування, виконання агрегацій.

Сортування і групування – це не опціональна стадія, котра виконується, якщо не знайдено шляхів доступу для отримання результату в запрошеному порядку[2].

Вибірка результатів виконується наступними способами:

- Вкладені цикли;
- Злиття;
- Поєднання хешуванням.

Вкладені цикли.

Вкладені цикли – це вкладені ітеративні процеси пошуку даних у кожній із з’єднуваних таблиць.

Зовнішній цикл витягує усі необхідні строчки з зовнішньої таблиці. Якщо частина або усі обмеження для зовнішньої таблиці можуть бути використані для пошуку по індексу, то на кожній ітерації циклу в індексі знаходяться розміщення усіх необхідних строк та виконується прямий доступ до таблиці. В іншому випадку таблиця сканується повністю. Обмеження які залишились використовуються для фільтрації вибраних строк. Для кожної строки яка залишилась визивається внутрішній цикл.

Внутрішній цикл по умовам з’єднання та даним зовнішнього циклу шукає строки у внутрішній таблиці. Якщо частина або усі обмеження для внутрішньої таблиці, а також обмеження, отримані от зовнішнього циклу, можуть бути використані для пошуку по індексу, то на кожній ітерації циклу в індексі знаходяться розташування усіх необхідних строк та виконується прямий доступ до таблиці. В іншому випадку таблиця сканується повністю. Обмеження які залишились використовуються для фільтрації вибраних строк.

Цикли можуть вкладатися довільну кількість разів. В цьому випадку внутрішній цикл стає зовнішнім для наступного і т.д.

На кожній ітерації самого глибокого циклу обрані з таблиці строки конкатенуються, для отримання однієї строки кінцевого результату.

Якщо для деякого цикла виконується пошук по індексу, і всіх колонок в індексі достатньо для отримання кінцевого результату, то прямий доступ до таблиці в цьому циклі не виконується [2].

Переваги:

- Найзагальніший, а тому і незамінний алгоритм з'єднання. При допомозі поєднання вкладеними циклами можливо реалізувати будь-яку умову з'єднання. (Інші алгоритми мають обмеження по створеним з їх допомогою умовами з'єднання. Наприклад, коли умова виражається нерівністю);
- Самий швидкий алгоритм, якщо необхідно отримати лише першу строку результату;
- При використанні пошуку по індексу алгоритм найкраще за усіх маштабується. Тобто при збільшенні об'єму даних в поєднуваних таблицях час виконання запиту збільшується практично лінійно при тих самих апаратних ресурсах.

Недоліки:

- Самий повільний алгоритм. Усі інші алгоритми мають переваги у швидкості (але лише в строго визначених обставинах). Втім, деякі автори вказують, що програш у швидкості на реальних системах у порівнянні з іншими алгоритмами несуттєвий.

Злиття.

Якщо об'єднувані таблиці мають індекси по полям які порівнюються, то поєднання може бути виконано за допомогою злиття. Обидва індекса скануються та в них знаходяться однакові значення. Якщо колонок в індексах

достатньо для отримання кінцевого результату, то читання таблиць не виконується. В іншому випадку виконується прямий доступ до таблиць які зливаємо для отримання колонок, які не входять в індекси, але необхідних для отримання результату.

Якщо злиття недостатньо для отримання кінцевого результату, то для кожної строки, отриманої злиттям, може виконуватися до двох серій вкладених циклів, відповідно, для кожної зі зливаємих таблиць [2].

Переваги:

- Поєднання злиттям ефективніше, ніж алгоритм поєднання вкладеними циклами, при умові, що списки є відсортовані. В принципі, накладні розходи на сортування можуть бути розподілені між декількома операціями злиття;
- Поєднання злиттям на відміну від поєднання з використанням хешування може використовуватися при великих розмірах поєднуваних таблиць.

Недоліки:

- Головним недоліком алгоритму є необхідність в попередньому сортуванні списків. Накладні розходи можуть бути занадто високими;
- При реалізації у СУБД, поєднання злиттям потребує більше пам’яті та менш гнучко, ніж алгоритм поєднання вкладеними циклами. В зв’язку з цим на практиці рекомендують уникати цього виду поєднання. В багатьох СУБД поєднання злиттям не за замовчанням не виконується оптимізатором запитів та повинно бути включено явно;
- Поєднання злиттям, як і алгоритм поєднання хешуванням, потребує в умовах не менше однієї умови рівності.

Поєднання хешуванням.

Алгоритм отримує на вхід дві таблиці та умови поєднання. Результатом його роботи є таблиця з результатами поєднання.

Менша з двох вхідних таблиць поміщається у спеціальну структуру даних у пам’яті: хеш-таблицю, котра забезпечує дуже високу швидкість пошуку. Тоді для кожної строки з більшої таблиці виконується пошук значень, які відповідають умові поєднання. Результати поміщаються у вихідну таблицю [2].

Переваги:

- Поєднання хешуванням суттєво швидше методу поєднання вкладеними циклами. При невеликому розмірі меншої таблиці це самий ефективний метод поєднання.

Недоліки:

- В умовах поєднання має бути мінімум одна умова рівності;
- Велика необхідність в пам’яті для побудови хеш-таблиці, що сильно обмежує маштабованость алгоритму при збільшенні розмірів меншої таблиці;

Хеш-таблиця має бути побудована повністю, до того як перший результат буде записаний у таблицю результат, що робить цей вид поєднання неприйнятним при необхідності отримати першу строку результату як можна скоріше.

1.3. Оптимізація запитів.

Оптимізація запитів – функція СУБД, яка виконує пошук оптимального плану виконання запиту зо всіх можливих для запиту заданого користувачем, також оптимізацією запитів можна назвати процес зміни запиту чи структури бази даних з ціллю зменшення використання обчислювальних ресурсів при виконанні запиту. При виконанні запиту СУБД може отримати один і той самий результат різними способами, котрі можуть значно відрізнятися по часу

виконання, а також по ресурсам які були використані. Задачею оптимізатора якраз і є знаходження оптимального плану [3].

В реляційних СУБД оптимальний план виконання запиту – це послідовність використання операторів реляційної алгебри до відношень, котра для конкретного стану БД може бути виконана з мінімальним використанням обчислювальних ресурсів [3].

На даний момент відомі такі стратегії пошуку оптимального плану запиту:

- грубої сили шляхом перебору та оцінки усіх перестановок таблиць, що поєднуються, тобто це повний перебір усіх варіантів та вибір найкращого, незважаючи на те що буде обраний найкращий, не є оптимальною стратегією так як витрачені ресурси не оправдовують результату;
- алгоритми динамічного програмування;
- генетичний алгоритм для більш складних запитів.

Також деякі СУБД дозволяють адміністратору БД втручатися в пошук оптимального плана, це може бути як мінімальне втручання, так і повне та чітке вказування плану запиту [3].

Плани виконання можуть порівнюватися з урахуванням багатьох різноманітних факторів, які відрізняються у різних СУБД, в тому числі:

- наявність індексів;
- можливість виконання операції злиття;
- потенційна кількість строк, які будуть витягуватись з кожної таблиці, це значення отримується зі статистики;
- спосіб читання записів у таблицях, блоків таблиць та індексів.

Частіше за все поєднання виконується вкладеними циклами, але у деяких випадках цей алгоритм може виявитись менш ефективним, ніж інші,

спеціалізовані, алгоритми. Наприклад, якщо у таблиць є індекси по полям, які ми поєднуємо, або одна або обидві таблиці достатньо малі, щоб ми мали можливість відсортувати їх у пам’яті, то використовується алгоритм злиття [3].

Суть оптимізації – це пошук мінімуму функції вартості перестановки таблиць. Оптимізатор повинен завжди вміти оцінювати вартість для будь–якої довільної перестановки, незалежно від обраної стратегії, в цей самий час самі перестановки для подальшого аналізу подаються іншим алгоритмом. Множина перестановок може відрізнятися від усього простору перестановок. Виходячи з цього псевдокод алгоритму роботи оптимізатора можна записати наступним чином:

```
current_tables_order := find_tables_order;
best_tables_order := current_tables_order;
min_cost := max_cost;

do
     cost := get_cost(current_tables_order);
     if cost < min_cost then
          best_tables_order := current_tables_order;
          min_cost := cost;
     endif
     current_tables_order := find_next_tables_order();
while (available_next_tables_order);
```

Поговоримо більш детально про стратегії оптимізації.

Стратегія грубої сили.

При використанні стратегії грубої сили оптимізатор виконує аналіз усієї множини перестановок та усіх таблиць, які вибираються та порівнює сумарні оцінки вартості виконання поєднань для кожної перестановки. На практиці для оптимізації було запропоновано включати в простір для аналізу лише

лівосторонні з’єднання. Для кожної таблиці з перестановок також оцінюється можливість використання індексів. Перестановка з мінімальною вартістю запиту і є кінцевий план виконання запиту [3].

Стратегія на основі генетичного алгоритму.

При збільшенні числа таблиць в запиті загальна кількість можливих перестановок росте як n!, виходячи з цього сильно зростає і час для оцінки кожної з цих перестановок. Через це виникає проблема оптимізації запитів на основі великого числа таблиць. У пошуках вирішення цієї проблеми було запропоновано використання генетичного алгоритму для оптимізації запитів, який би давав не саме оптимальне рішення, але за менш короткий відрізок часу. Використовуючи генетичний алгоритм ми аналізуємо лише частину від усього простору перестановок. Таблиць, які є у запиті, ми кодуємо у хромосоми, далі виконуються операції мутації та схрещування. На кожній ітерації виконується відновлення хромосом та відбір хромосом, що надають мінімальну оцінку вартості поєднання. В результаті відбору залишаються лише ті хромосоми, що надали найменшу, у порівнянні з попередніми, оцінку вартості. Після повторення декількох ітерацій вибирається найбільш ефективне рішення [3].

Оцінка числа строк, які витягуємо.

Оцінка числа строк, які витягуються з таблиці виконується для прийняття рішення про повне сканування таблиці замість доступу по індексу. Рішення приймається на думці, що кожне читання листової сторінки індексу с диску тягне за собою одне або більше позиціювання та одне або більше читань сторінок таблиці. Оскільки індекс містить ще і нелистовані сторінки, то витягування більш 0,1 – 1% строк з таблиці, як правило, ефективніше виконувати повним скануванням таблиці.

Більш точна оцінка може бути отримана на основі наступних показників:

1. Число строк, які витягуємо.

2. Середня довжина ключа в індексі.
3. Середнє число строк у сторінці індексу.
4. Довжина сторінки індексу.
5. Висота B*-дерева в індексу.
6. Середня довжина строки в таблиці.
7. Середнє число строк в сторінці таблиці.
8. Довжина сторінки таблиці.

СУБД намагається організувати збереження блоків даних однієї таблиці послідовно з ціллю виключити накладні витрати на позиціювання при повному скануванню. Ефективність повного сканування також збільшується за рахунок упереджувального читання. При упереджувальному читанню СУБД одночасно видає зовнішній пам’яті команди читання декількох блоків. Сканування починається по завершенню читання будь-якого з блоків. Одночасно продовжується читання блоків що залишились. Ефективність досягається за рахунок паралелізму читання та сканування [3].

Статистика.

Для оцінки потенційного числа строк, що витягуємо з таблиці, реляційні СУБД використовують статистику. Статистика має вигляд гістограм для кожної колонки таблиці, де по горизонталі шкала значень, а по висоті стовпця відмічається оцінка числа строк в відсотках від загального числа строк [3].

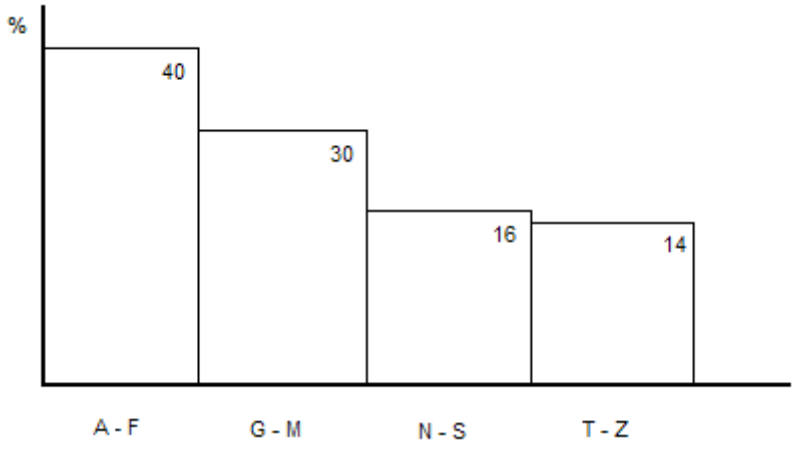


Рисунок 1.1. Гістограма для колонки таблиці [3].

Таким чином, якщо з таблиці витягуються строки зі значенням колонки C з обмеженням [V1, V2], то можна оцінити число рядків, які потрапляють у цей інтервал. Алгоритм оцінки кількості числа витягуємих строк наступний:

1. Визначити, в які інтервали гістограми попадає обмеження [V1, V2].
2. Знайти оцінки числа строк Ri для кожного інтервалу i у відсотках.
3. Якщо [V1, V2] потрапляє в деякий інтервал [S1, S2] частково чи повністю лежить в інтервалі, то:
    - Знайти перетин [V1, V2] та [S1, S2].
    - Відкоректувати число значень в частковому інтервалі (це або Ri * (V1 – S2 + 1) / (S1 – S2 + 1), або Ri * (S1 – V2 + 1) / (S1 – D2 + 1), або Ri * (V1 – V2 + 1) / (S1 – S2 + 1));
4. Інакше оцінка для інтервалу дорівнює Ri.
5. Просумувати оцінки у відсотках для усіх інтервалів.
6. Перевести оцінку у відсотках у число строк.

Як правило СУБД не знає і не може знати точну кількість рядків у таблиці (навіть для виконання найпростіших запитів виконується сканування первинного індексу), оскільки в базі можуть зберігатися одночасно декілька образів однієї і тієї самої таблиці з різним числом рядків. Для оцінки кількості рядків використовуються наступні дані:

1. Число сторінок у таблиці.
2. Довжина сторінки.
3. Середня довжина рядка у таблиці.

Статистика також може зберігатися зростаючим ітогом. У такому випадку кожен інтервал містить сумарну оцінку усіх попередніх інтервалів та власну оцінку. Для отримання оцінки числа рядків для обмеження [V1, V2] достатньо з оцінки інтервалу, в котрий потрапляє V2, відняти оцінку інтервалу, в котрий потрапляє V1 [3].

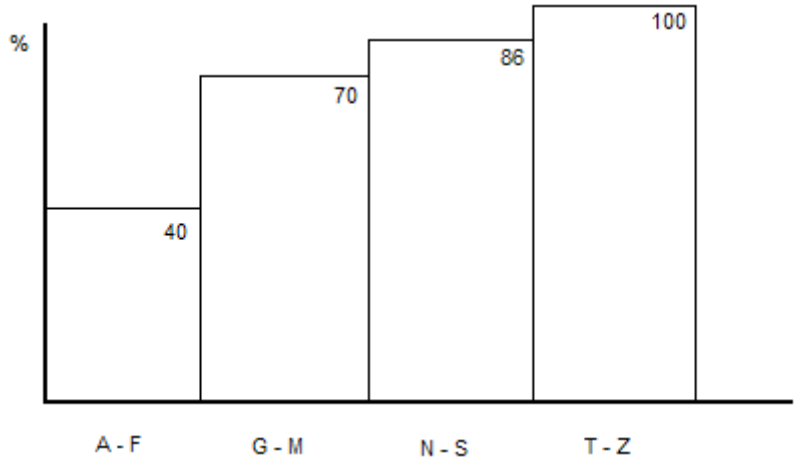


Рисунок 1.2. Статистика зростаючим ітогом [3].

Збір статистики для побудови гістограм виконується або спеціальними командами СУБД, або фоновими процесами СУБД. При цьому, через те, що база може зберігати великий об’єм даних, робиться вибірка меншого об’єму із всієї генеральної сукупності рядків. Оцінка репрезентативності (достовірності) вибірки може виконуватись, наприклад, по критерію узгодження Колмогорова.

Якщо дані у таблиці значно змінюються у короткий проміжок часу, то статистика перестане бути актуальною та оптимізатор приймає невірні рішення о повному скануванню таблиць. Режим роботи бази даних має бути спланований таким чином, щоб підтримувати актуальну статистику, або не використовувати оптимізацію на основі статистики [3].

1.4. Висновки до розділу.

У розділі були розглянуті основні властивості та аспекти сучасних СУБД. Був розглянутий план запиту, та процес його створення силами СУБД. Також були розглянуті методи оптимізації запитів силами СУБД. Були розглянуті способи з’єднання таблиць, їх переваги та недоліки. Виходячи з проаналізованої інформації можна постановити проблему оптимізації запитів, що буде зроблено у наступному розділі.

РОЗДІЛ 2. ПРОБЛЕМИ СУЧАСНИХ АЛГОРИТМІВ ОПТИМІЗАЦІЇ ТА ОПТИМІЗАЦІЯ ЗАПИТІВ З ВИКОРИСТАННЯМ НАВЧАННЯ З ПІДКРІПЛЕННЯМ.

2.1. Постановка проблеми оптимізації запитів.

Як було вказано раніше, SQL – декларативна мова. Виходячи з цього маємо, що користувач вказує СУБД лише операції мають бути проведені з даними. За вибір способу виконання цих операцій відповідає вже сама СУБД. Для прикладу простий запит:

```
SELECT title FROM titles WHERE year > 1980;
```

може бути виконаний двома способами: читання усіх записів з таблиці titles та перевірка кожної з них на виконання умови (рис.2.1), або використовувати індекс по полу year (рис.2.2). У другому випадку ми не проглядаємо лишні записи, але витрачаємо більше часу на обробку одного запису через операції з індексами [6].

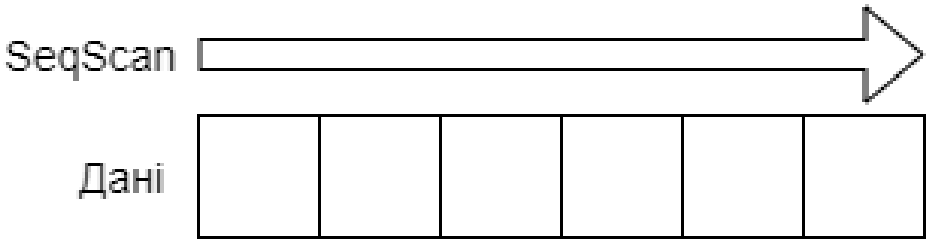


Рисунок 2.1. Читання усіх записів.

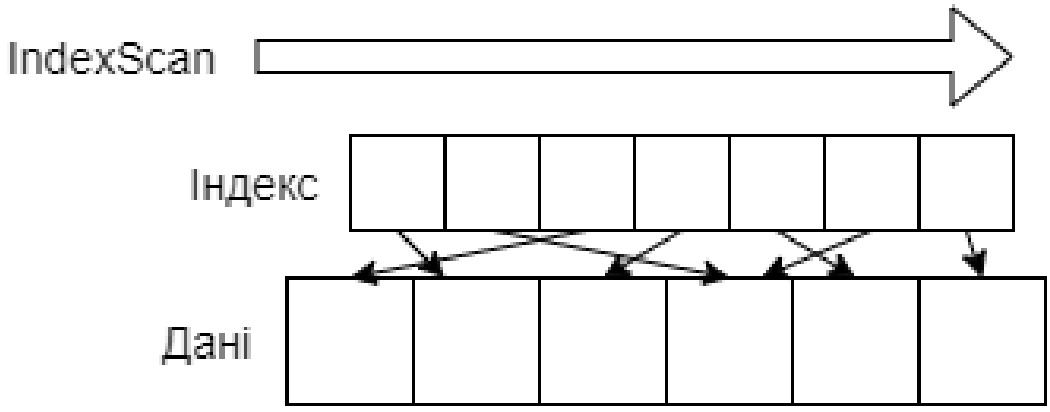


Рисунок 2.2. Сканування індексів.

Розглянемо складніший запит:

SELECT titles.descritpion FROM titles, genres WHERE title.genre_id = genres.id;

Це з’єднання можливо виконати вже трьома різними способами:

- вкладений цикл проглядає усі можливі пари записів з двох таблиць та перевіряє для кожної пари виконання умови (рис.2.3);
- злиття відсортує обидві таблиці по полям id та genre_id відповідно, а далі використає метод двох вказівників для пошуку усіх пар записів, які задовольняють умові. Цей метод аналогічний методу сортування злиттям (рис.2.4);
- хешування будує хеш-таблицю по полю найменшої таблиці. Хеш-таблиця дозволяє для кожного запису з однієї таблиці знайти запис з відповідністю (рис.2.5).

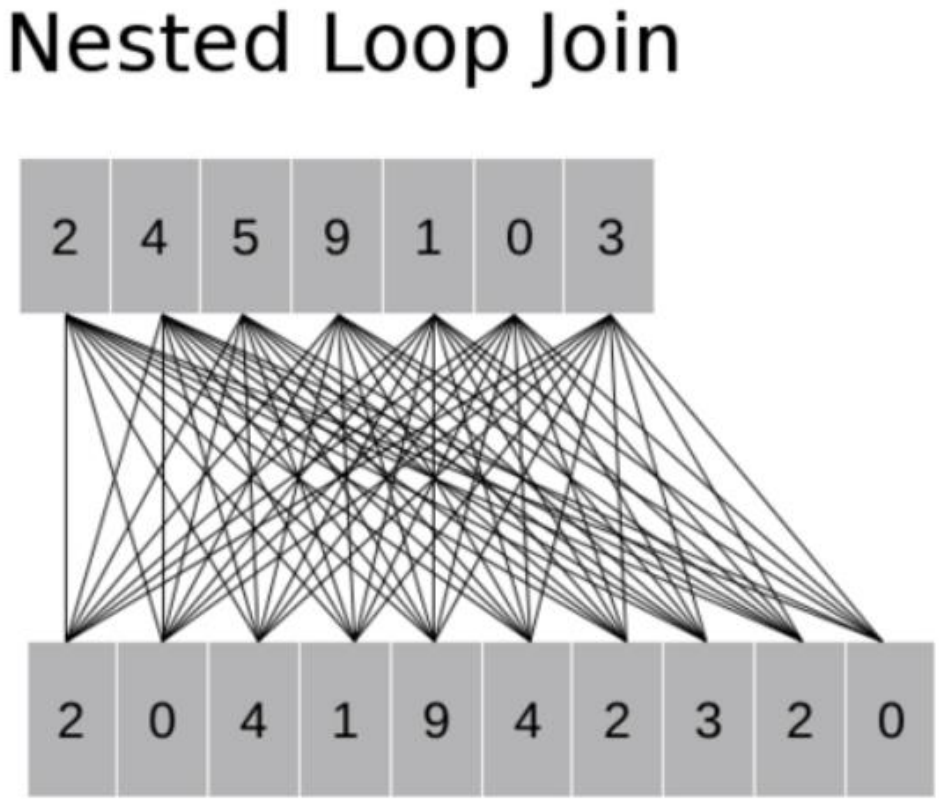


Рисунок 2.3. Поєднання вкладеними циклами [6].

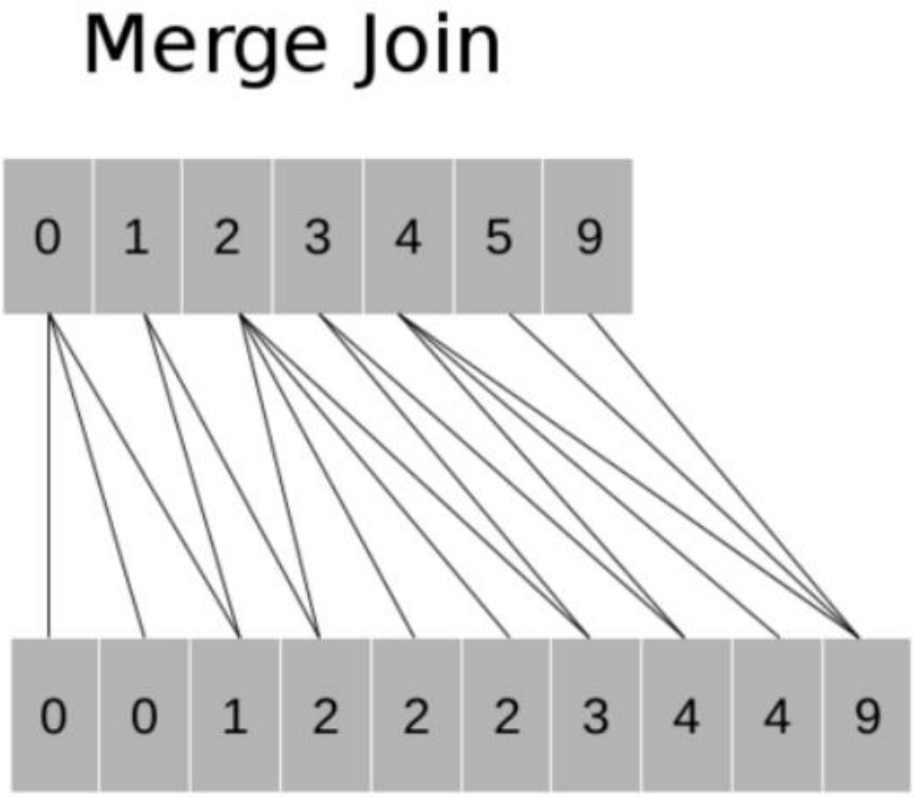


Рисунок 2.4. Поєднання злиттям [6].

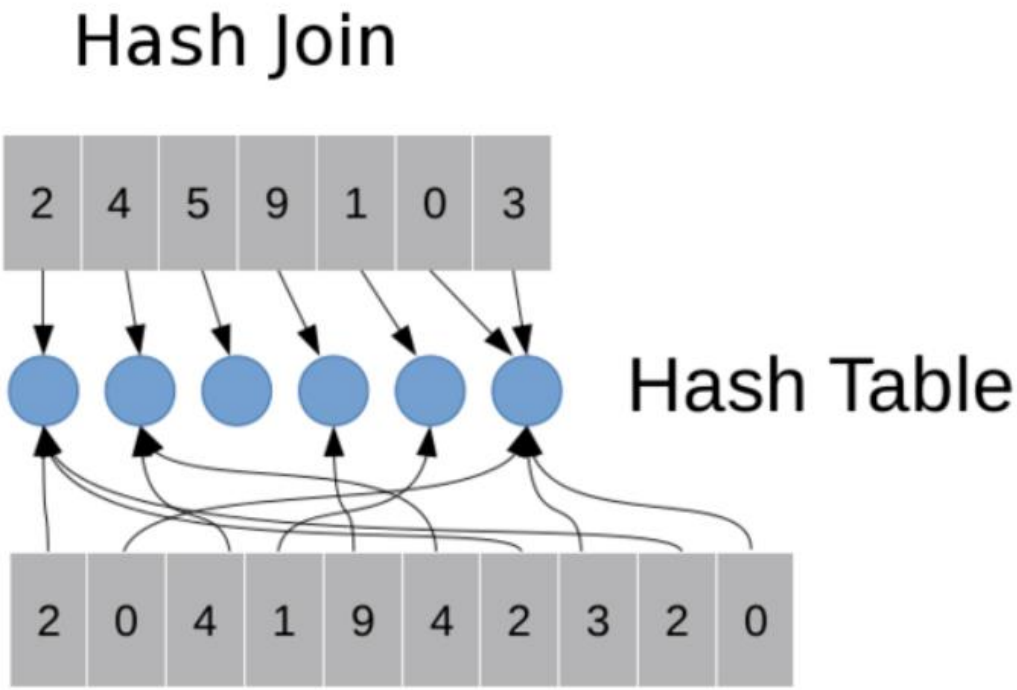


Рисунок 2.5. Поєднання з використанням хеш-таблиці [6].

Також якщо в запиті необхідно виконати більше однієї операції з’єднання таблиць, то також можна виконувати їх у різному порядку. Отриманий результат буде деревом виконання запиту (рис.2.6), що і являє собою план виконання запиту.

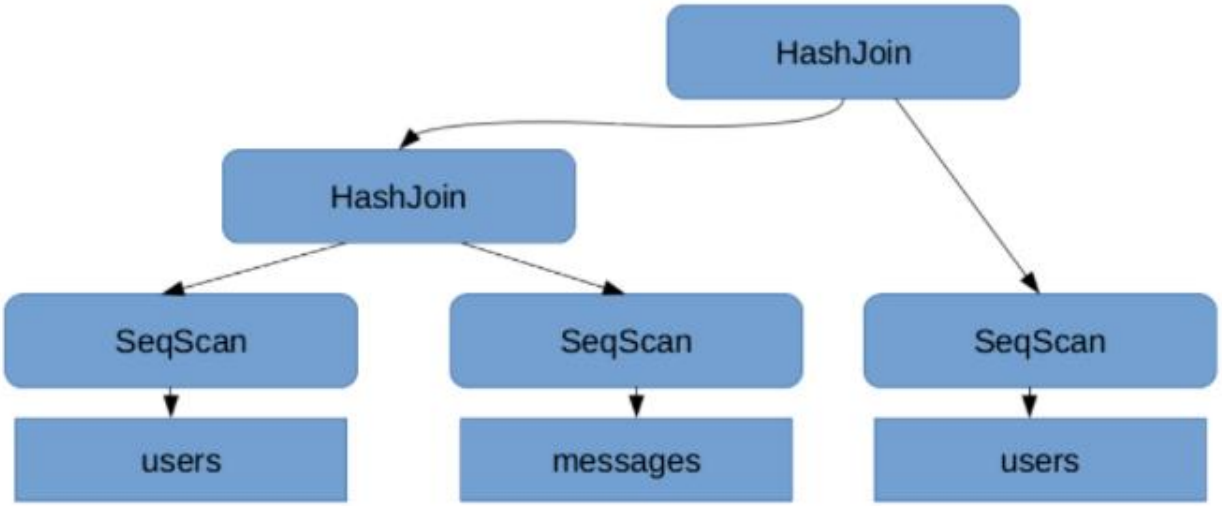


Рисунок 2.6. Дерево виконання запиту [6].

Сучасні СУБД надають інструменти для перегляду обраного плану запиту. Наприклад в PostgreSQL щоб подивитись план для конкретного запиту можна використати команду EXPLAN:

EXPLAN SELECT * FROM …

Для того щоб виконати запит та подивитись обраний для нього план можна використати команду EXPLAIN ANALYSE:

EXPLAIN ANALYSE SELECT * FROM …

Час виконання різних планів одного й того ж самого запиту може суттєво відрізнятись на багато порядків. Тому правильний вибір плану запиту сильно впливає на роботу СУБД. На прикладі СУБД PostgreSQL розберемося як вибирається план запиту зараз.

Процес пошуку оптимального плана можна розділи на дві частини:

- по-перше, необхідно вміти оцінювати вартість будь-якого плану – кількість ресурсів, що необхідні для його виконання. У випадку, коли на

серверi виконуються інші задачі та процеси, оцінений час є прямо пропорціональний кількості витрачених на нього ресурсів. Тому можна вважати, що вартість плану – це його час виконання в деяких умовних одиницях;

- по-друге, необхідно вибрати план з мінімальною оцінкою вартості. Легко показати, що кількість планів зростає експоненційно зі збільшенням складності запиту, тому не можна просто перебрати усі плани, оцінити вартість кожного та обрати самий дешевший. Для пошуку оптимального плану використовуються більш складні алгоритми дискретної оптимізації: динамічне програмування по підмножинам для простих запитів та генетичний алгоритм для складних [6].

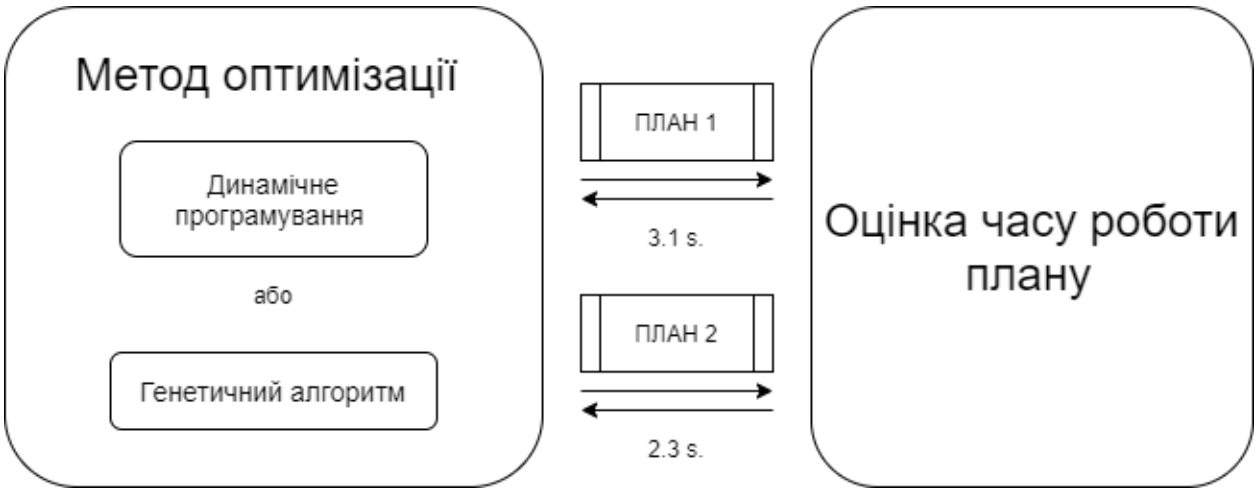


Рисунок 2.7. Схема процесу пошуку оптимального плану.

Основною проблемою даного підходу є те що СУБД ніяк не враховує попередній досвід, через що, наприклад, деякі запити, які часто виконуються можуть мати неоптимальний план запиту, і СУБД все одно буде використовувати його. Проблема є з обома етапами побудови запиту, як кожен раз буде проводитись помилкова оцінка часу плану роботи, так і обраний метод оптимізації може виявитись неоптимальним. Саме цю проблему можливо вирішити завдяки використанню машинного навчання, в нашому

випадку ми будемо розглядати оптимізацію побудови плану з використанням навчання з підкріпленням.

2.2. Навчання з підкріпленням.

Навчання з підкріпленням – область машинного навчання, яка вивчає те як інтелектуальні агенти повинні приймати рішення у середовищі з ціллю максимізації отриманої нагороди [7].

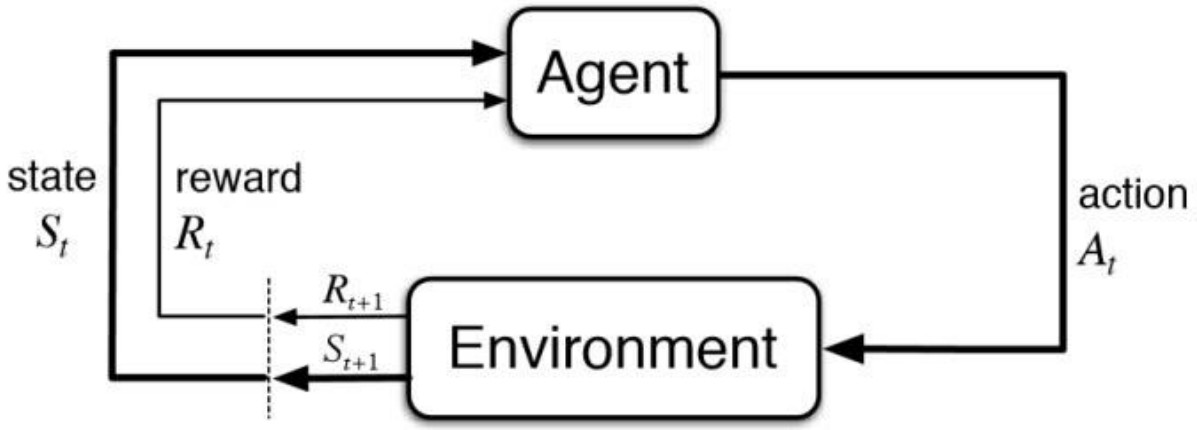


Рисунок 2.8. Схема навчання з підкріпленням [8]**.**

Навчання з підкріпленням – одна з трьох основних парадигм машинного навчання, наряду з навчанням з вчителем та навчанням без вчителя. Навчання з підкріпленням відрізняється від навчання з учителем тим, що не потребує датасету з помічених пар вводу / виводу та не потребує явного виправлення неоптимальних дій. Замість цього основна увага приділяється пошуку балансу між дослідженням (невідомої території) та експлуатації (наявних знань) [7].

Середовище зазвичай описуємо у формі марковського процесу прийняття рішень (MDP), оскільки багато алгоритмів навчання з підкріпленням для цього контексту використовують методи динамічного програмування. Основною відміною між класичними методами динамічного програмування та алгоритмами навчання з підкріпленням у тому, що останні не пропонують знання точної математичної моделі MDP та націлені на великі MDP, де точні методи стають неможливими [7].

Завдяки своїй спільності, навчання з підкріпленням вивчається у багатьох дисциплінах таких, як теорія ігор, теорія управління, теорія інформації, дослідження операцій, моделювання на основі оптимізації, багатоагентні системи, статистика і т. д. В літературі по дослідженню операцій та контролю навчання з підкріпленням називають приблизним динамічним програмуванням або нейродинамічним програмуванням. Проблеми, в яких можливе використання навчання з підкріпленням, також вивчались у теорії оптимального управління, котрий в основному пов'язаний з існуванням і характеристикою оптимальних рішень та алгоритмів для їх точного обчислення, та в меншій мірі з навчанням чи приближенням, особливо у відсутності математичної моделі навколишнього середовища. В економіці та теорії ігор навчання з підкріпленням може використовуватись для пояснення того, як може виникнути рівновага при обмеженій раціональності [7].

Базове навчання з підкріпленням моделюється як марковський процес прийняття рішень:

- набір станів середовища та агента S;
- набір дій агенту A;
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ вірогідність переходу (в моменту часу t) від стану s до стану s' під дією a;
- $R_a(s, s')$ нагорода після переходу від стану s до стану s' під дією a;

Ціль навчання з підкріпленням полягає у тому, щоб агент засвоїв оптимальну чи майже оптимальну політику, котра максимізує функцію нагороди, чи інший представлений програмістом сигнал підкріплення котрий набирається із негайних нагород. Це схоже на процеси, які проходять в психології тварин. Наприклад, біологічний мозок запрограмований так, щоб інтерпретувати такі сигнали, як біль та голод, як від'ємне підкріплення, а задоволення та ситість – як позитивне підкріплення. В деяких обставинах тварини можуть навчитися вести себе, оптимізуючи ці нагороди. Це говорить про те, що тварини здатні до навчання з підкріпленням [7].

Базовий агент навчання з підкріпленням взаємодіє зі своїм середовищем дискретними часовими кроками. У кожний момент часу t агент отримує стан середовища s_t та нагороду r_t. Далі він обирає дію a_t з набору доступних дій, котра далі відправляється у середовище. Середовище переходить у новий стан s_{t+1} та вираховується нагорода r_{t+1} пов'язана з переходом (s_t, a_t, s_{t+1}). Ціль агенту навчання з підкріпленням – вивчити політику:

$$\pi : A * S \rightarrow [0,1], \pi(a, s) = \Pr(a_t = a \mid s_t)$$

що максимізує нагороду що очікується [7].

Формулювання проблеми у вигляді MDP передбачає, що агент безпосередньо спостерігає за поточним станом середовища, в цьому випадку кажуть, що проблема повністю спостерігається. Якщо агент має доступ лише до підмножини станів, або якщо спостережувані стани пошкоджені шумом, агент, як кажуть, має часткову спостережливість, і формально проблема повинна бути сформульована як частково спостережуваний марковський процес прийняття рішень. В обох випадках набір доступних для агента дій може бути обмежений. Наприклад, стан залишку на рахунку можна обмежити як позитивний; якщо поточне значення стану дорівнює 3, а перехід стану намагається зменшити значення на 4, перехід не буде дозволений [7].

Коли ефективність агента порівнюється з ефективністю агента, який діє оптимально, різниця в продуктивності породжує поняття жалю. Для того, щоб діяти майже оптимально, агент повинен міркувати про довгострокові наслідки своїх дій (тобто максимізувати майбутні доходи), хоча безпосередня винагорода, пов'язана з цим, може бути негативною.

Таким чином, навчання для підкріплення особливо добре підходить для проблем, які включають довгострокову та короткострокову компромісну винагороду. Воно було успішно застосовано до різних проблем, включаючи управління роботом, планування ліфтів, телекомунікації, нарди, шашки та Go (AlphaGo) [7].

Два елементи роблять навчання з підкріпленням потужним: використання образів для оптимізації продуктивності та використання апроксимації функцій для роботи з великими середовищами. Завдяки цим двом ключовим компонентам навчання з підкріпленням можна використовувати в великих середовищах в наступних ситуаціях:

- модель навколишнього середовища відома, але аналітичне рішення недоступне;
- дана лише імітаційна модель середовища (оптимізація на основі симуляції);
- єдиний спосіб зібрати інформацію про середовище – це взаємодіяти з нею.

Перші дві проблеми можуть бути розглянуті як проблеми планування (оскільки існує певна форма моделі), в той час як останню можна розглядати як реальну проблему навчання. Однак навчання з підкріпленням перетворює обидві проблеми планування у проблеми машинного навчання [7].

Алгоритми, які використовуються найчастіше.

Q-Learning та SARSA (State-Action-Reward-State-Action) – два популярні алгоритми навчання з підкріпленням без моделей. Вони розрізняються своїми стратегіями розвідки, в той час як їх стратегії експлуатації схожі. В той час як Q-навчання є методом поза політики, де агент взнає значення, основане на поточній дії a, яка з’являється з іншої політики. SARSA – це метод у політиці, де він вивчає цінність на основі своїх поточних дій, що випливають з його поточної політики. Ці два методи прості в реалізації, але їм не вистачає універсальності, так як вони не дозволяють оцінювати значення для невідомих станів. Це можна вирішити використовуючи більш досконалі алгоритми, такі як Deep Q-Networks, які використовують нейронні мережі для оцінки Q-значень. Але DQN можуть обробляти лише дискретні маловимірні простори

дій. DDPG (Deep Deterministic Policy Gradient) – алгоритм типу актор-критик, що може вирішити цю проблему [8].

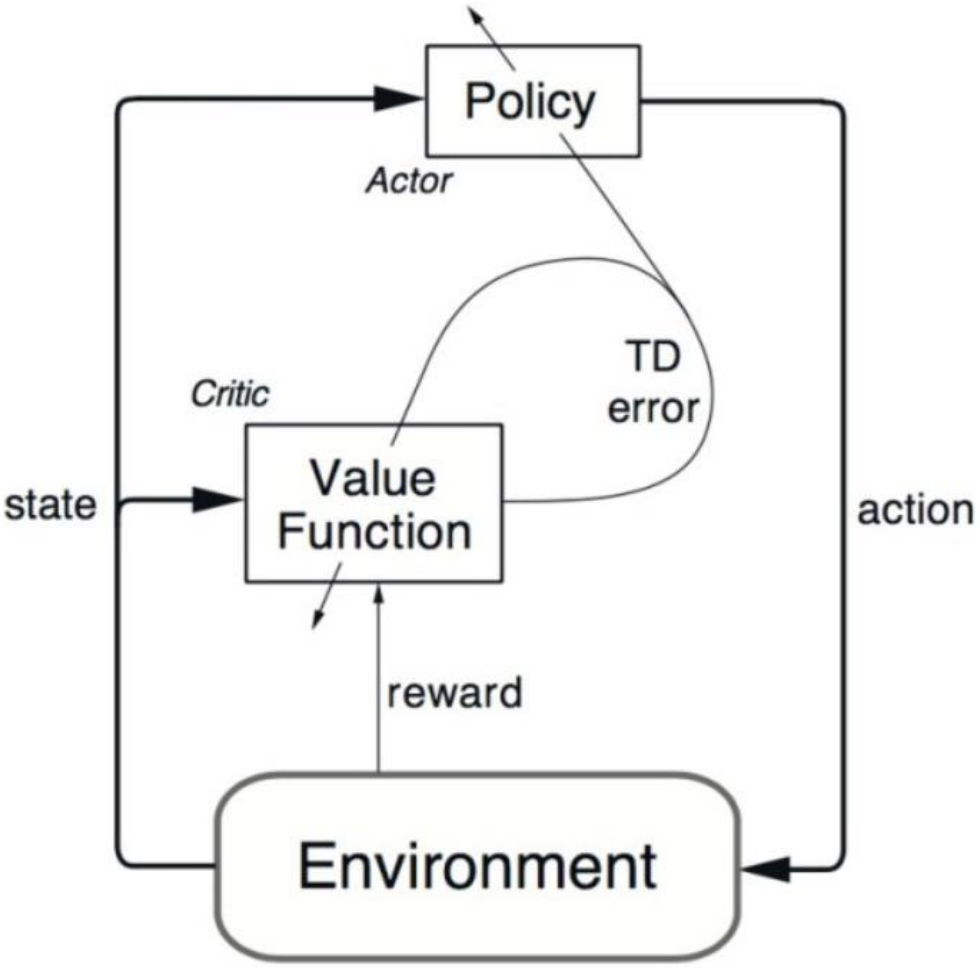


Рисунок 2.9. Архітектура актор-критик для навчання з підкріпленням [8].

2.3. Оптимізація запитів з використанням навчання з підкріпленням.

Постановка задачі: замість вирішення проблеми впорядкування з’єднань з використанням методів динамічного програмування або генетичних алгоритмів, ми формулюємо проблему як процес прийняття рішень Маркова (MDP) та вирішуємо її за допомогою використання навчання з підкріпленням [13].

Тобто ми маємо визначити певне середовище, що може надавати агенту свій стан, далі в залежності від дії обраної агентом, змінювати стан, та очікувати нової дії від агента. Виконуємо цей процес ітеративно доки не

досягаємо деякого кінцевого стану, коли не залишиться доступних дій – отримання плану впорядкування з’єднань, після цього агент завершує серію та отримує нагороду в залежності від прийнятих рішень. Ціль агента – максимізувати нагороду, спираючись на досвід [12].

Порядок з’єднань ми можемо уявити у вигляді бінарного дерева з’єднань (рис.2.10).

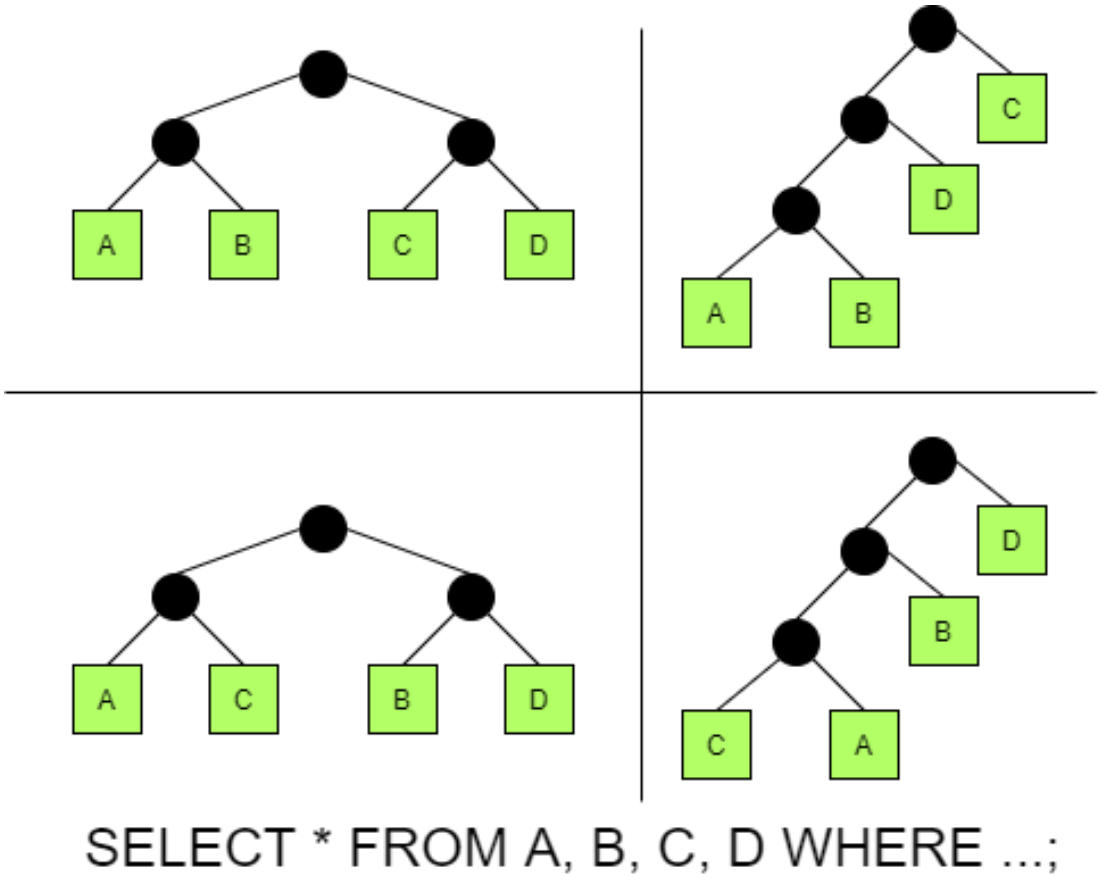


Рисунок 2.10. Дерева бінарних з’єднань**.**

Щоб використовувати навчання з підкріпленням для вирішення задачі про порядок з’єднань, ми збираємося використати наступні елементи:

- кожен стан буде представляти собою бінарне дерево з’єднань. Далі ми перетворюємо це дерево у вектор стану та додаємо додаткову інформацію, таку як предикат з’єднання та предикат вибору;
- кожна дія являє собою об’єднання двох піддерев у одне дерево;
- епізод закінчується, коли ми приєднали усі відносини;

- винагорода буде розраховуватись на основі оцінки вартості моделі з результату порядку з’єднань [12].

Більш детально про вектор стану.

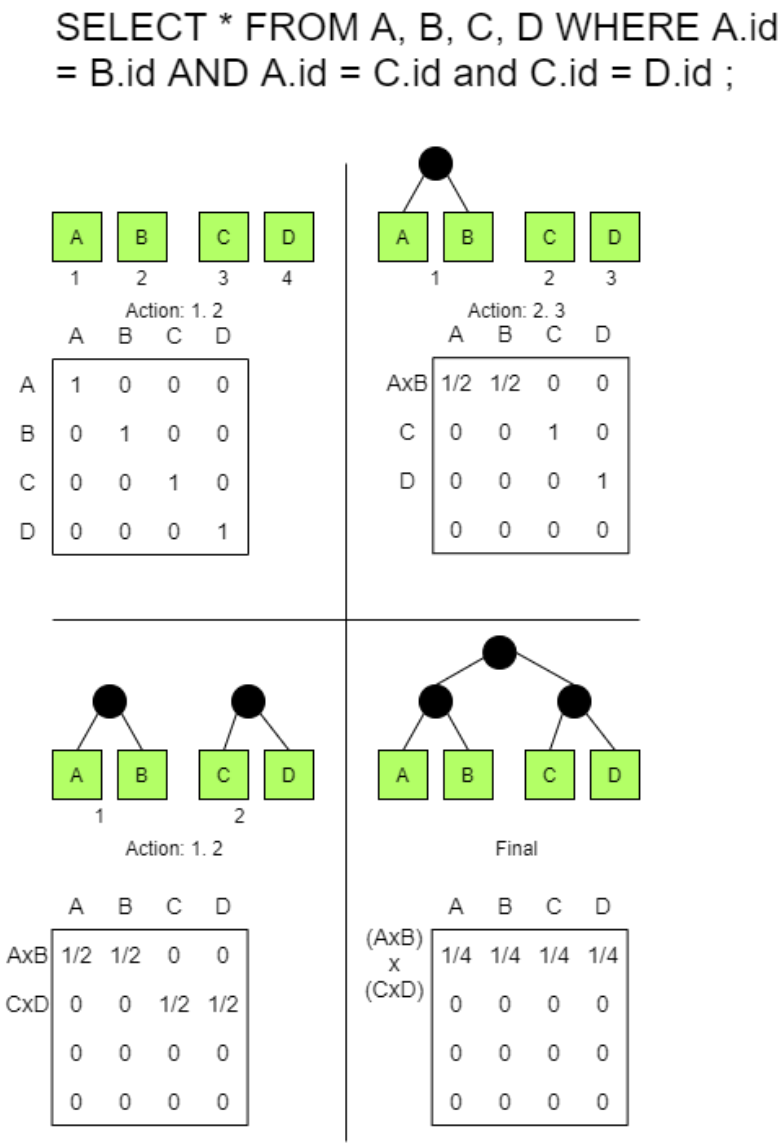


Рисунок 2.11. Побудова бінарного дерева з’єднань.

Рисунок 2.12. Вектор атрибутів та матриця з’єднань.

Ми уявляємо бінарне дерево з’єднання та додаткову інформацію о з’єднаннях у вигляді векторів. Ми кодуємо кожне піддерево, як вектор v з розміром n, де n – кількість відносин у базі даних. Значення vi дорівнює нулю, якщо відношення не знаходиться у дереві бінарного з’єднання, та 1/h(i,x), де h(i,x) – висота відношення і в піддереві x (відстань від кореню). Наприклад, на рис.1 ми представляємо піддерево A як [1 0 0 0], тому що в цьому випадку відношення A має висоту – один, а інші відношення в піддереві відсутні. Далі ми уявляємо що приєднали A до B, що призводить до нового відношення AxB. Тепер A та B мають висоту – два, отже, ми заповнюємо відповідні значення в рядку AxB значенням ½. C та D залишаються рівними 0, так як вони не входять в це піддерево.

Крім того, ми вектор атрибутів та матрицю з’єднань. Матрицею з’єднань буде бінарна матриця m розміром nxn, де значення m(i, j) буде дорівнювати одиниці, якщо є з’єднання між відношеннями та нуль, у іншому випадку. На рис.2.11 значення m(1,2) = m(2,1) дорівнює одиниці, так як в нас є з’єднання між відношеннями A та B (A.id = B.id). Значення для m(1,3) = m(3,1) також дорівнює одиниці, тому що в нас є з'єднання між A та C (A.id = C.id). Вектор атрибутів – це k – мірний бінарний верктор, де k – загальна кількість атрибутів усіх відношень. Значення елементів в цьому векторі буде дорівнювати одиниці, якщо атрибут використовується у запиті, та рівний нулю у іншому випадку. На рис.2.12 A.id дорівнює одиниці, тому що в запиті SQL в нас використовується цей атрибут [12].

Ми навчаємо агента з підкріпленням знаходячи оптимальну політику, яка надає найбільшу винагороду. Ми виконуємо це використовуючи нейронну мережу, котра приймає вектор стану в якості вхідних даних. Отримуємо вектор можливих дій, що містить n елементів, кожний з яких є можливою дією. Архітектуру описаного можна побачити на рисунку 2.13.

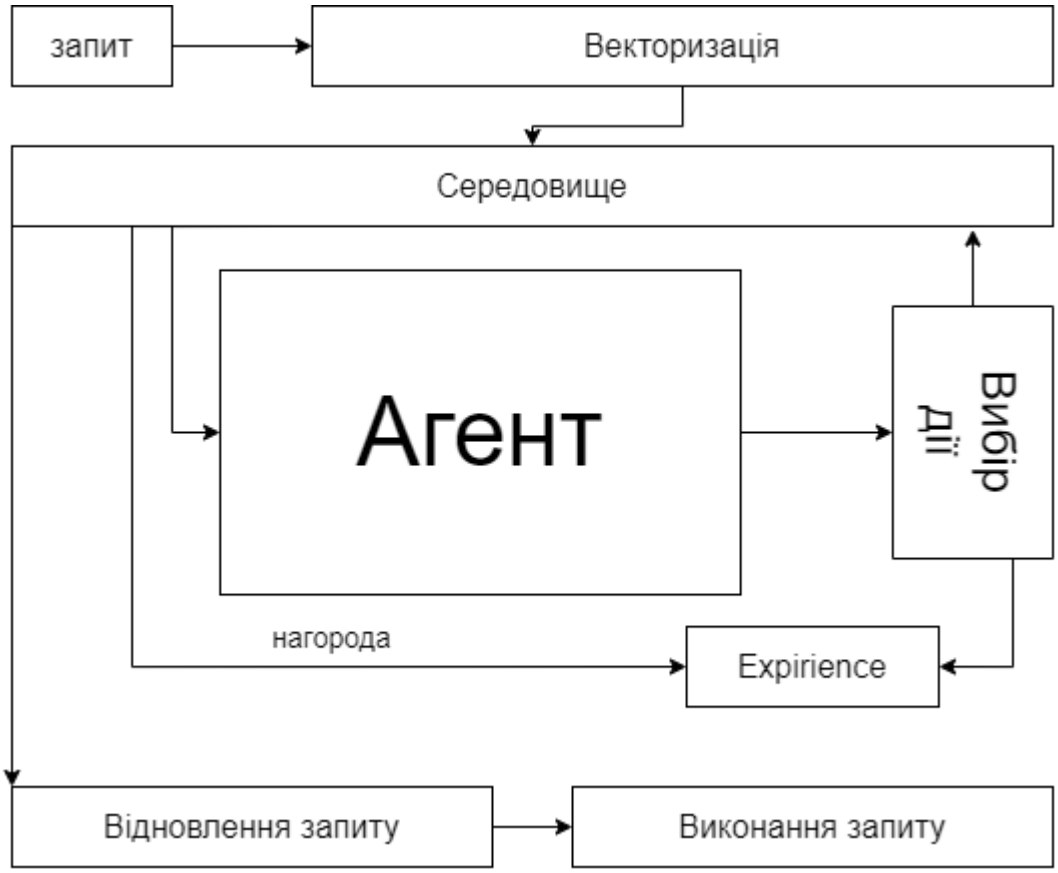


Рисунок 2.13. Схема використання навчання з підкріпленням для задачі створення оптимального порядку з’єднань.

2.4. Висновок до розділу.

У розділі була виконана постановка проблеми виконання створення плану запиту силами СУБД. Був зроблений висновок, що основною проблемою цього процесу є відсутність попереднього досвіду, через що один і той самий запит може постійно виконуватись з неоптимальним планом. Було розглянуто навчання з підкріпленням та постановка задачі створення плану запиту у рамках цього алгоритму. Були визначені усі елементи навчання з підкріпленням для задачі оптимізації запитів, такі як – середовище, вектори стану та дії. Була розроблена схема алгоритму, що буде реалізований програмно у наступному розділі.

РОЗДІЛ 3. СТВОРЕННЯ ПРОГРАМНОГО ПРОДУКТУ, ЩО ДОЗВОЛИТЬ ПРОТЕСТУВАТИ ЕФЕКТИВНІСТЬ ВИКОРИСТАННЯ НЕЙРОННИХ МЕРЕЖ ДЛЯ ОПТИМІЗАЦІЇ ЗАПИТІВ ДО СУБД.

Програма буде розроблена на основі відомостей у главі 2.3. Для початку визначимо технології що будуть використані для розробки. У якості мови програмування була обрана мова Python, для роботи з навчанням з підкріпленням буде використовуватись фреймворк Tensorforce, розроблений на основі Tensorflow, в якості тестуємої СУБД обрана PostgteSQL. Далі більш детально про обрані технології.

3.1. Мова програмування Python.

Рисунок 3.1. Логотип Python [14].

Python – високорівнева мова програмування загального призначення с динамічною строгою типізацією та автоматичним керуванням пам’яттю, мова орієнтована на підвищення продуктивності розробника, скорочення часу на розробку, читабельність коду, а також на кроссплатформеність. Мова Python є об’єктно орієнтованою. У Python усе це об’єкти. Незвичною особливістю мови є виділення блоків коду відступами, що значно покращує читаємість коду. Синтаксис мови достатньо простий та мінімалістичний, через що значно рідше з’являється необхідність звернення до документации, що значно пришвидшує розробку. Мова є інтерпритованою, часто використовується також для написання простих скриптів. Основним недоліком мови є більш низька швидкість роботи, а також використання більшого об’єму пам’яті на відміну

від аналогічного коду написаного на компільованих мовах програмування, таких як C, C++ [14].

Python є мультипарадигмальною мовою програмування, підтримує об’єктно-орієнтоване, структурне, функціональне, імперативне, процедурне програмування та метапрограмування. Задачі загального програмування вирішуються за рахунок динамічної типізації. Аспектно-орієнтоване програмування частково може бути вирішене завдяки декораторам, більш повноцінна підтримка забезпечується додатковими фремворками. Логічне та контрактне програмування може бути реалізовано з використанням бібліотек чи розширень [14].

Основними архітектурними рисами мови програмування є:

- автоматичне керування пам’яттю;
- механізми обробки виключень;
- динамічна типізація;
- необхідність відділення блоків через відступи;
- підтримка багатопотокових обчислень з глобальним блокуванням інтерпритатору;
- високорівневі структури даних;
- можливість розбиття програм на модулі, які в свою чергу, можна об’єднувати у пакети.

Основною реалізацією Python є інтерпритатор CPython, який підтримує більшість актуальних платформ та є стандартом мови програмування. Цей інтерпритатор розповсюджується за вільною ліцензією Python Software Foundation License, яка дозволяє використовувати його у будь яких програмах та додатках без обмежень. CPython компілює вихідний код у високорівневий байт-код, котрий виконується на стековій віртуальній машині. До основних реалізацій мови програмування також відносяться Jython, IronPython та PyPy [14].

Стандартна бібліотека Python включає в себе великий набір корисних функцій та модулей, починаючи від роботи зі строками та текстами, а закінчуючи можливостями асинхронного програмування, мультипроцессінгу та можливостями для написання мережевих додатків. Додаткові можливості, такі як написання веб-додатків, створення комп'ютерних ігор, математичне моделювання, використання моделей машинного навчання та нейромереж можуть бути реалізовані за допомогою великої кількості сторонніх бібліотек, а також інтерпритацією бібліотек, які були написані на C, C++, при цьому сам інтерпритатор Python може бути інтегрований у проекти написані на цих мовах програмування [14].

На сьогодні Python став однією з самих популярних мов програмування, використовується у багатьох сферах, серед яких:

- веб розробка;
- машинне навчання та нейромережі;
- розробка ігор;
- DevOps;
- розробка прикладного програмного забезпечення;
- написання різних скриптів та парсерів.

Python відмінно підходить для навчання, так як завдяки своїй високій читабельності та відсутності компіляції, мова дозволяє сконцентруватися на вивченні різноманітних парадигм, концепцій та алгоритмів. Також завдяки інтерпритованості мови значно спрощується відладка та виправлення помилок у коді. Данна мова програмування є популярною також серед крупних компаній, таких як Google, Instagram чи Facebook [14].

3.2. Tensorforce.

Tensorforce – бібліотека на основі TensorFlow для прикладного навчання з підкріпленням.

TensorFlow – відкрита програмна бібліотека для машинного навчання розроблена компанією Google для вирішення задач побудови та тренування нейронних мережей з ціллю автоматичного знаходження та класифікації образів, досягаючи якості людського сприйняття. Застосовується як для досліджень, так і для розробки власних продуктів Google. Основний API для роботи з бібліотекою реалізований для мови програмування Python, але також існують реалізації і для інших мов програмування, таких як: C, C++, R, Go, Swift, Java, Sharp [16].

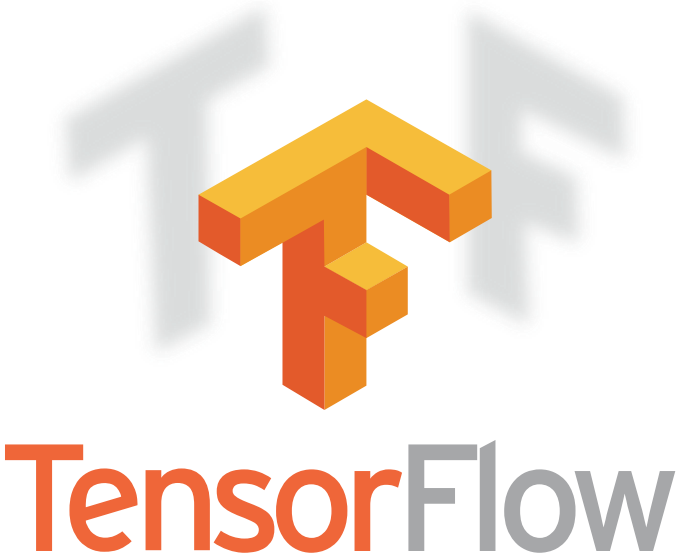


Рисунок 3.2. Логотип TensorFlow [15].

Tensorforce – фреймворк для глибокого навчання з підкріпленням з відкритим вихідним кодом, у якому особлива увага надається модульному гнучкому дизану бібліотеки та просте використання для додатків та досліджень на практиці [19].

Фреймворк слідує набору високорівневих дизайнерських рішень, котрі відрізняють його від інших подібних бібліотек:

- Модульний дизайн на основі компонентів – реалізації функцій, насамперед, намагаються бути як умога більш універсальними та мати можливості налаштування.

- Розділення алгоритму RL та додатку – алгоритми не залежать від типу та структури входів (станів / спостережень) та виходів (дій / рішень), а також від взаємодії із середовищем додатку.
- Повноцінні моделі TensorFlow – уся логіка навчання з підкріпленням, включаючи потік управління, реалізована в TensorFlow, щоб забезпечити можливість переносу графів обчислень незалежно від мови програмування додатків та спростити розгортання моделей [19].

Фреймворк дозволяє окремо визначити середовище та агента. Для визначення середовища рекомендується використання функції Environment.create(…), наприклад:

```
environment = Environment.create(
    environment='c_env', level='CartPole',
    max_episode_timesteps=100
)
```

В якості альтернативи можлива ініціалізація середовща з використанням файлу конфігурації:

```
{
    "environment": "c_env",
    "level": "CartPole"
}
```

Файли кофігурації середовища можна загрузити вказавши шлях до них:

```
environment = Environment.create(
    environment='env.json',
    max_episode_timesteps=100
)
```

Також є можливість реалізації середовища за допомогою наслідування класу Environment з Tensorforce:

```
class MyEnv(Environment):
    …
```

У цьому випадку буде необхідно визначити наступні методи:

- states() – визначаємо специфікації вектору стану середовища;
- actions() – визначаємо тип та кількість можливих дій агенту;
- reset() – метод що викликається після завершення кожного епізоду;
- execute() – метод що викликається після вибору необхідної для виконання дії, отримує номер дії.

Агент створюється за допомогою метода Agent.create(), наприклад:

```
agent = Agent.create(
    agent='ppo', environment=env, update=4,
    optimizer=dict(optimizer='adam', learning_rate=1e-5)
    reward_estimation=dict(horizon=15)
)
```

В якості альтернативи агент може бути заданий з використанням файла конфігурації так само як і середовище.

Для навчання та оцінки агенту рекомендується використання службових програм виконання, наприклад – Runner, котра пропонує ряд параметрів та конфігурацій. Базовий експеримент, який складається з навчання та послідуючої оцінки, можна записати у декілька стрічок коду:

```
runner = Runner(
    agent=my_agent,
    environment=my_env,
    max_episode_timesteps=100
)

runner.run(num_episodes=250)

runner.run(num_episodes=150, evaluation=True)

runner.close()
```

Службові класи виконання забезпечують коректну обробку взаємодії агенту із середовищем і тому мають виконуватись усюди де це можливо. В

якості альтернативи, якщо потребується більш детальний контроль над взаємодією агента і середовища, цикл навчання може бути визначений самостійно [19].

3.3. СУБД PostgreSQL.

Рисунок 3.3. Логотип PostgreSQL [17].

PostgreSQL – потужна об’єктно-реляційна база даних з відкритим кодом, активна розробка якої нараховує вже більше 30 років, за які СУБД побудувала собі репутацію за надійність, функціональність та продуктивність. Існують реаліації для багатьох UNIX-подібних систем та для Microsoft Windows [17].

СУБД підтримує більшу частину стандарту SQL та пропонує велику кількість сучасних функцій:

- зовнішні ключі;
- складні запити;
- тригери;
- транзакційну цілісність;

- оновляємі представлення;
- мультиверсіонний контроль паралелізму.

Крім того, СУБД може бути розширена користувачем різними способами, наприклад шляхом додання нових:

- функцій;
- операторів;
- типів даних;
- агрегатних функцій;
- процедурних мов;
- індексних методів.

А завдяки ліберальній ліцензії PostgreSQL може використовуватися, модифікуватися та розповсюджуватись ким завгодно безкоштовно для будь-яких цілей, будь то частні, комерційні або академічні [17].

Деякі SQL команди що також підтримуються PostgreSQL:

- SELECT – витягти дані з БД;
- UPDATE – оновити дані у БД;
- DELETE – видалити дані з БД;
- INSERT INTO – вставити дані у БД;
- CREATE DATABASE – створити нову базу даних;
- ALTER DATABASE – модифікувати базу даних;
- CREATE TABLE – створити нову таблицю;
- ALTER TABLE – модифікувати таблицю;
- DROP TABLE – видалити таблицю;
- CREATE INDEX – створити індекс (search key);
- DROP INDEX – видалити індекс.

3.4. Реалізація програми.

Для початку нагадаємо архітектуру реалізуємої програми:

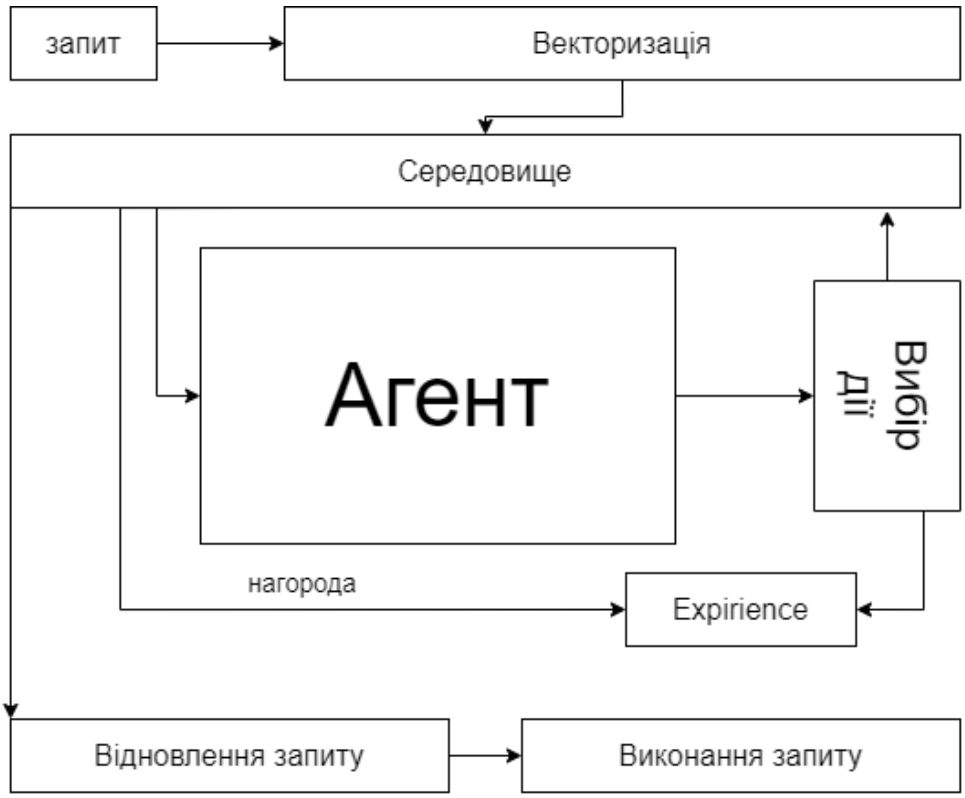


Рисунок 3.4. Схема програми.

Отже необхідно реалізувати вказану вище схему. Поглянемо на створений проект та його файли:

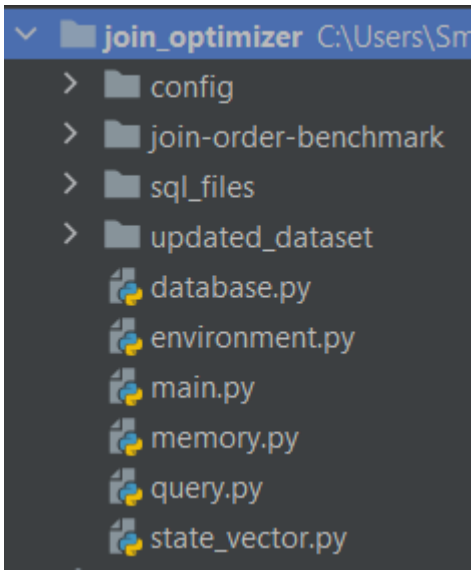


Рисунок 3.5. Директорія проекту.

Файл main.py є вхідною точкою у програмі, у ньому ініціалізуються усі необхідні класи та ресурси, такі як середовище та агент.

Середовище реалізоване як клас, який наслідує від стандартного класу середовища фреймворку Tensorflow, і міститься у файли environment.py. Для роботи з базою, а також для менеджементу датасету з запитами середовище використовує клас Database з файлу database.py. Цей клас також використовується для оцінки вартості запиту, після завершення створення порядку з’єднань запиту. Розбір вхідного запиту на необхідні частини, а також його відновлення відбувається у файлі query.py класом Query. Створення векторів стану відбувається у файлі state_vector.py. Клас Memory з файлу memory.py необхідний для зберігання статистики нагород та вартості створених запитів.

Більш детально про виконання програми.

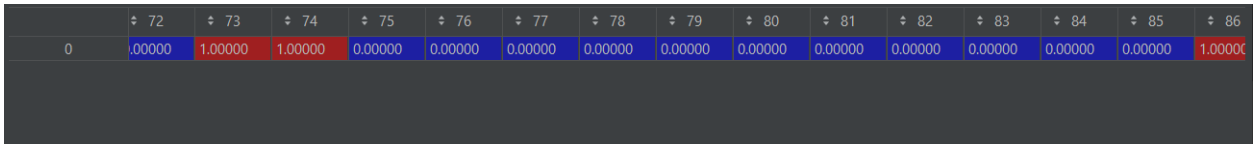
Ми створюємо середовище передаючи список запитів на яких хочемо навчати агента, кількість епізодів, а також режим роботи середовища. Режим роботи може бути двох типів:

- loop – при цьому режимі виконання запитів буде відбуватись циклічно, тобто якщо список запитів наступний [1a.sql, 2a.sql, 3a.sql], то виконуватись вони будуть у наступному порядку [1a.sql, 2a.sql, 3a.sql, 1a.sql, 2a.sql, 3a.sql, …] і так далі доки не закінчяться епізоди;
- one_q – при цьому режимі ми ділимо кількість епізодів на кількість запитів, та виконуємо кожен отриману кількість разів і переходимо до наступного.

При ініціалізації середовища створюється об’єкт класу що відповідає за роботу з базою даних. Після створення середовища ініціалізується агент з вказаними параметрами, та відбувається запуск тестування через runner.run(), де ми вказуємо кількість епізодів.

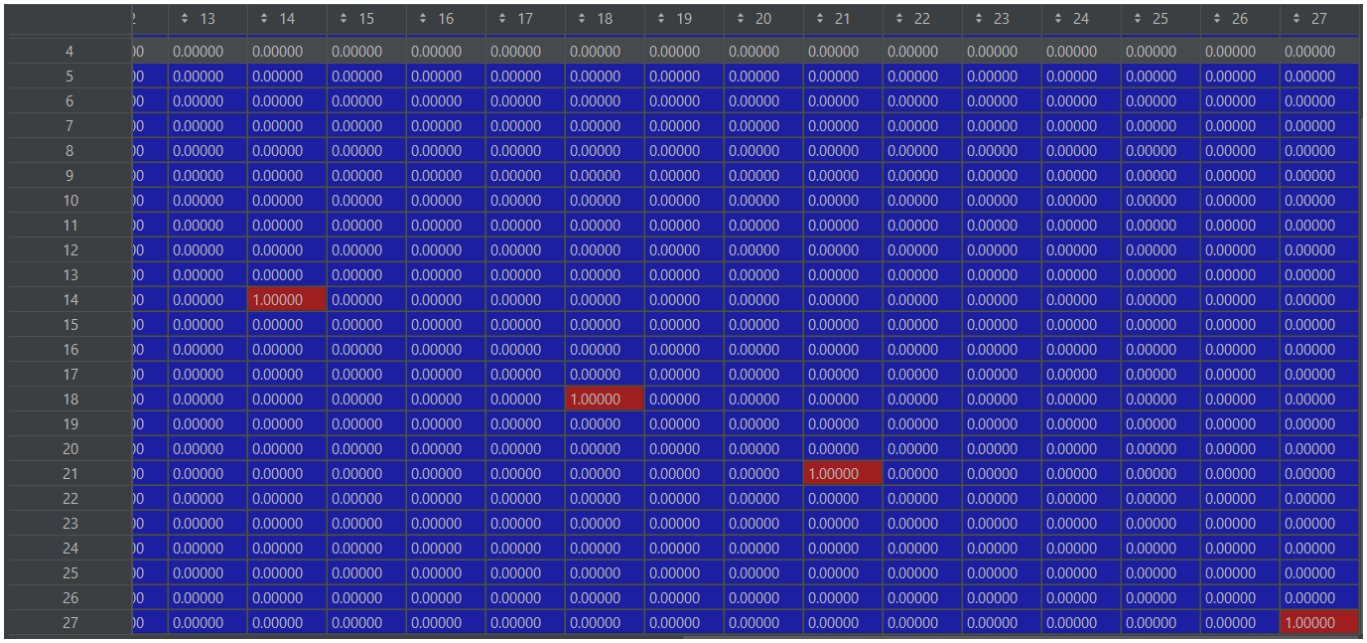
Далі для кожного окремого епізоду створюється об’єкт для роботи з запитом, який створює усі необхідні ресурси для створення вектору станів, та подальшого відновлення запиту з новим порядком з’єднань. Після цього створюються вектора станів:

- вектор атрибутів (рис. 3.6);
- матриця з’єднань(рис. 3.7);
- матриця дерева з’єднань(рис. 3.8).



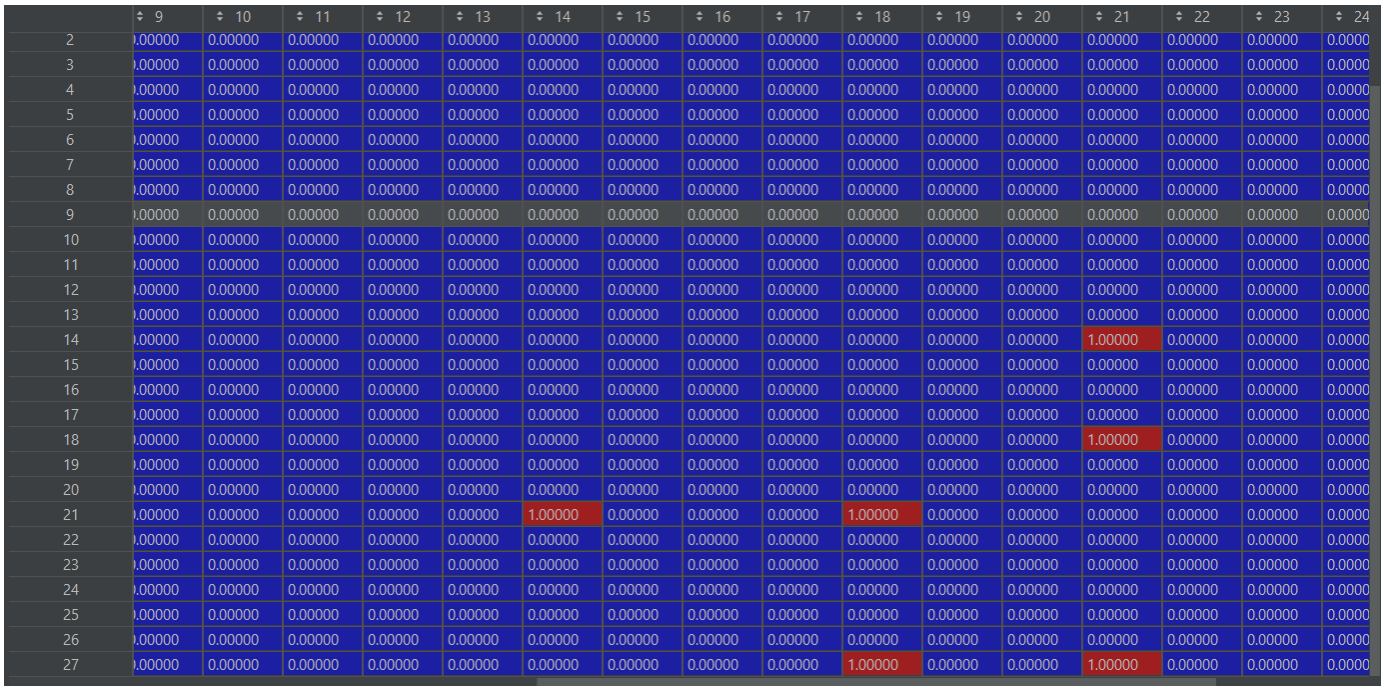
	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86
0	.00000	1.00000	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000

Рисунок 3.6. Вектор атрибутів.



	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
4	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
5	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
6	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
7	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
8	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
9	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
10	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
11	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
12	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
13	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
14	0.00000	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
15	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
16	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
17	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
18	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
19	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
20	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
21	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
22	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
23	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
24	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
25	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
26	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
27	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000

Рисунок 3.7. Матриця з’єднань.



	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
2	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
3	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
4	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
5	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
6	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
7	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
8	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
9	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
10	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
11	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
12	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
13	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
14	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.0000
15	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
16	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
17	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
18	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.0000
19	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
20	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
21	.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
22	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
23	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
24	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
25	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
26	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.0000
27	.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	1.00000	0.00000	0.00000	1.00000	0.00000	0.00000	0.0000

Рисунок 3.8. Матриця дерева з’єднань.

Після побудови векторів станів, вони передаються агенту, на що він відповідає обраною дією. Дія і є з’єднанням що необхідно приєднати, оновлюємо дерево з’єднань, та передаємо агенту нові оновлені стани, і так ітеративно доки не будуть приєднані усі таблиці. Після цього за допомогою створеного порядку з’єднань створюється запит еквівалентний вхідному, але з визначеним порядком, далі проводимо оцінку запиту. З отриманої вартості вираховуємо, враховуючи попередній досвід збережений у пам’яті, нагороду для агента. Після передачі нагороди переходимо до наступного епізоду, навчання відбувається у такому вигляді доти доки не закінчяться епізоди. Приклад роботи програми можна побачити на рис. 3.9.

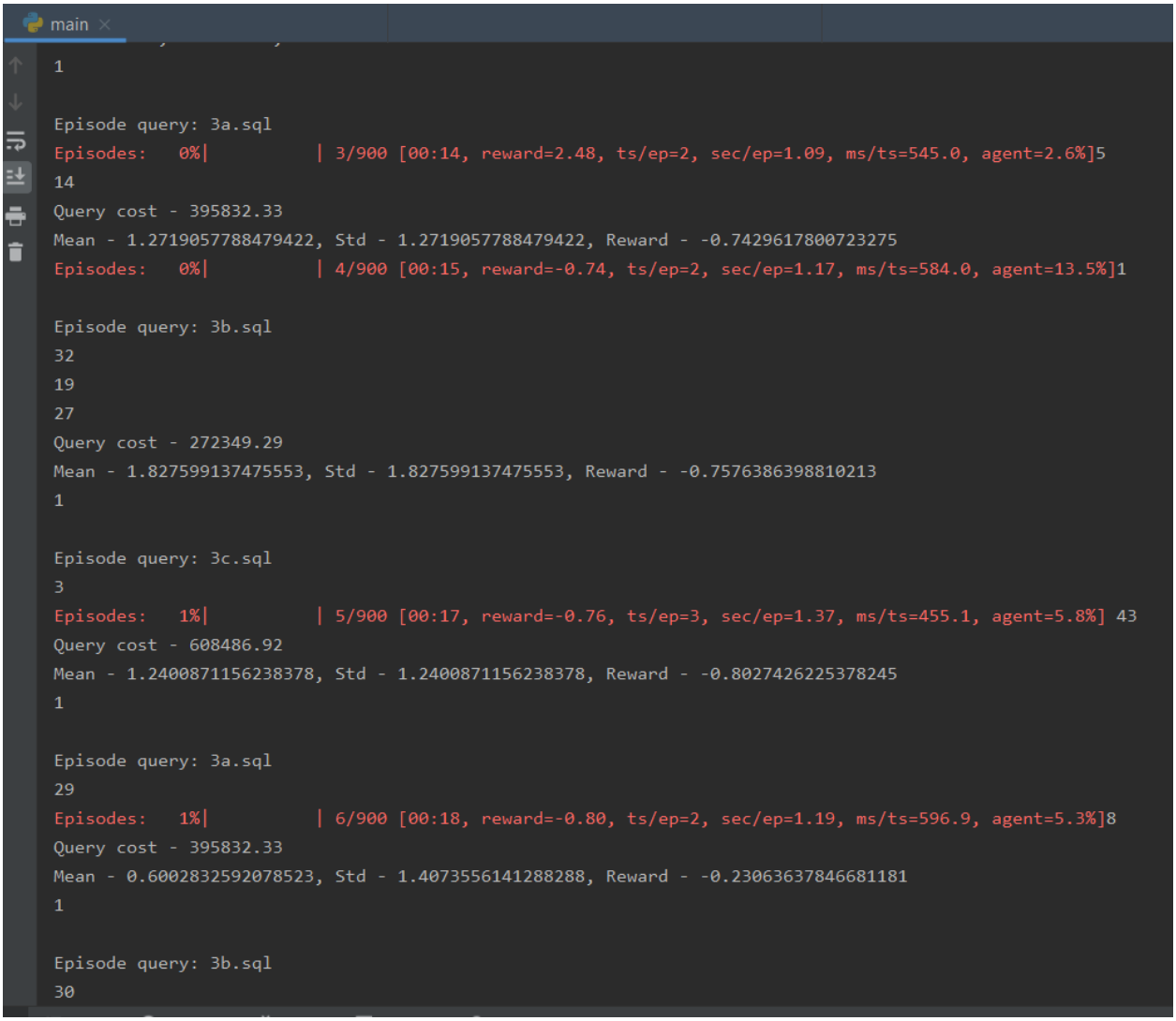


Рисунок 3.9. Приклад роботи програми.

По закінченню роботи програми виводяться графіки результатів навчання.

Основними недоліками створеної програми є:

- програма оптимізує лише створення порядку з’єднань для запиту. Нерозв’язаними проблемами залишаються помилкова оцінка кількості кортежів для запиту з великою кількістю з’єднань, а також вибір типу з’єднань між таблицями (вкладені цикли, метод злиття, побудова хеш-таблиці);
- навчена модель здатна працювати лише з тією базою даних, на ослові запитів до якої було проведено тестування, навіть при зміні таблиць у існуючій базі даних доведеться знову навчати агента. Тому виникає

проблема з тим що для кожного окремого випадку необхідна своя власна модель.

3.5. Проведення тестів.

Для тестування програми була використана база даних IMDB, а саме її репліка створена спеціально для тестування. Створена база даних (рис.3.10) містить 21 таблицю, які в свою чергу сумарно налічують декілька десятків мілiонів записів.

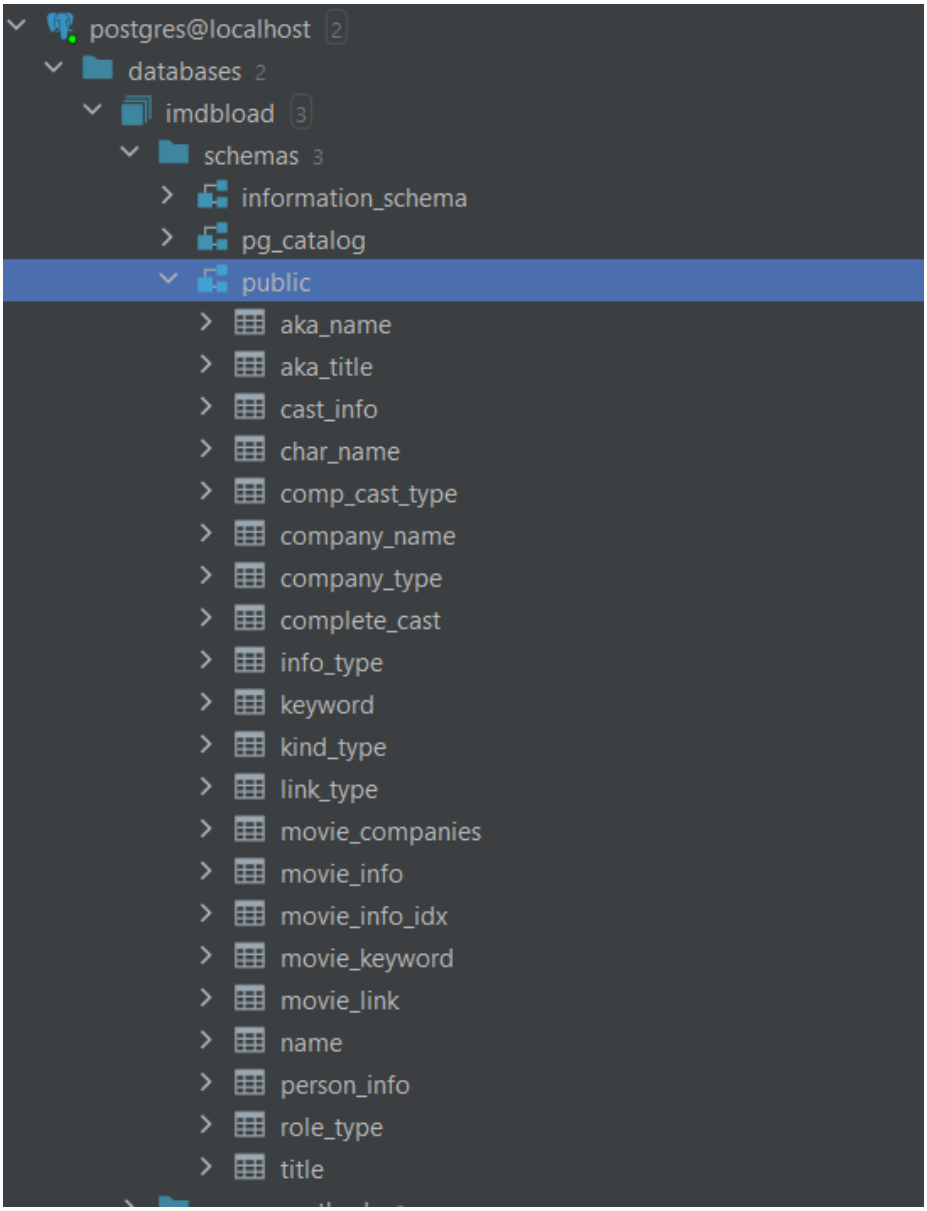


Рисунок 3.10. База даних для тестування.

У якості датасету для навчання та тествування використовується join-order-benchmark зі статті “How Good Are Query Optimizers, Really?”. Він складає собою набір зі 127 запитів з різноманітним числом з’єднань. Датасет можна побачити на рис. 3.11:

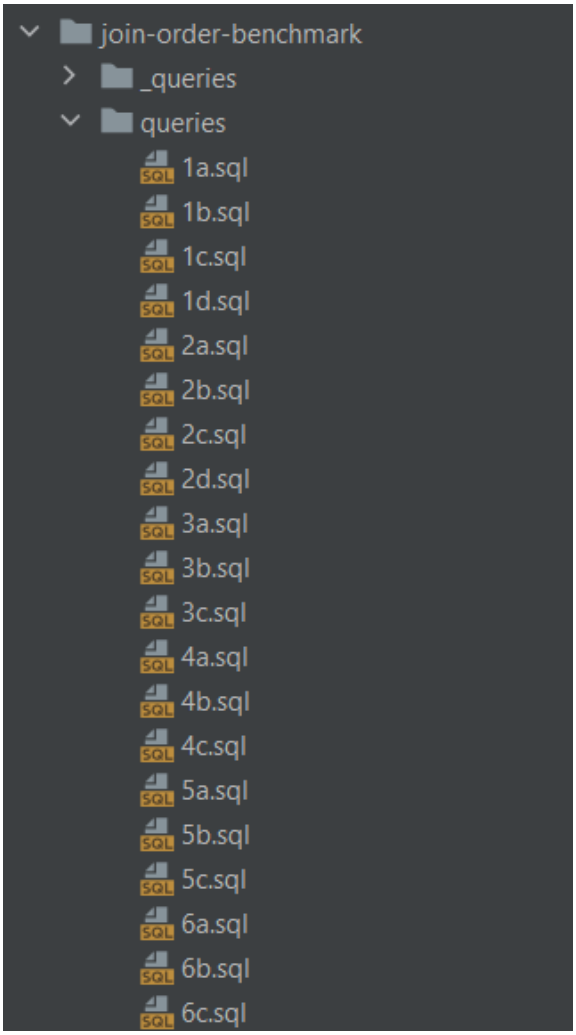
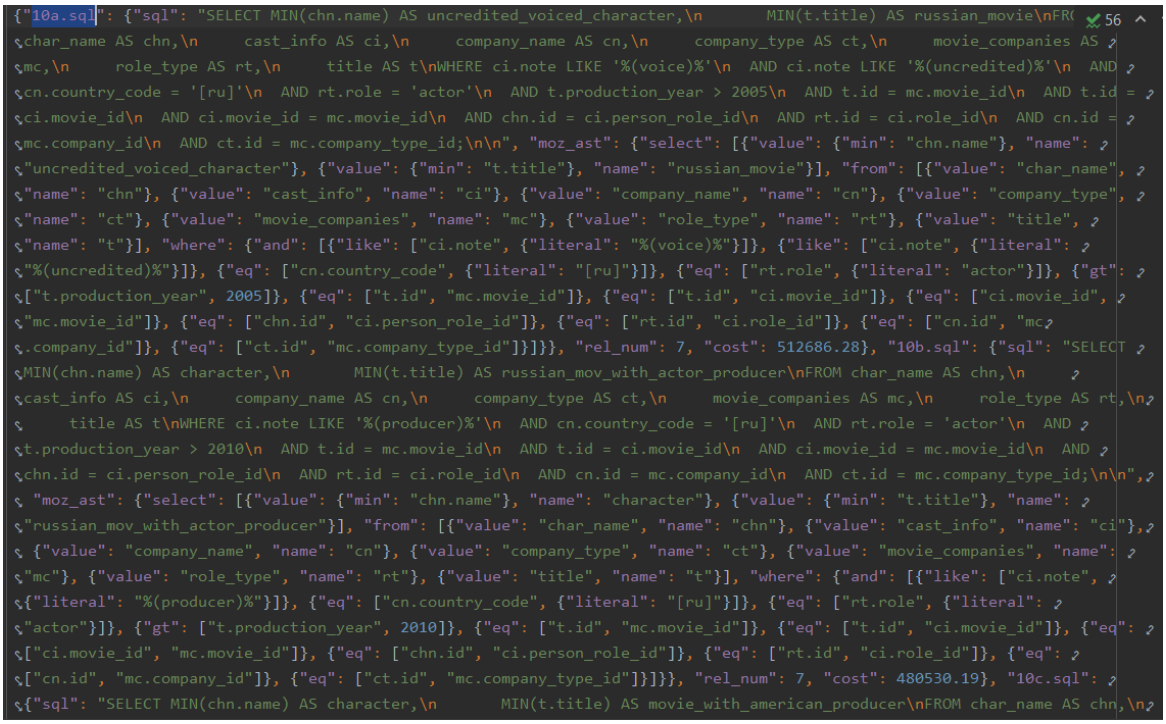


Рисунок 3.11. Датасет для тестування.

Далі, для зручності, за допомогою створеної утиліти upd_dataset.py, був створений JSON файл що містить усі запити, запити розібрані на частини у вигляді словнику, а також вартість запиту у СУБД. Розклад запиту відбувається за допомогою бібліотеки moz_sql_parser, а саме функції parse() звідти. Вона приймає на вхід “сирий” SQL запит та повертає на виході розкладений запит у вигляді словнику, де окремо виділено що витягується, з яких таблиць, та умови витягування. Ці дані далі необхідні для створення векторів стану, та подальшого відновлення запиту після завершення епізоду.



```
{"10a.sql": {"sql": "SELECT MIN(chn.name) AS uncredited_voiced_character,\n       MIN(t.title) AS russian_movie\nFR
char_name AS chn,\n     cast_info AS ci,\n     company_name AS cn,\n     company_type AS ct,\n     movie_companies AS
mc,\n     role_type AS rt,\n     title AS t\nWHERE ci.note LIKE '%(voice)%'\n  AND ci.note LIKE '%(uncredited)%'\n  AND
cn.country_code = '[ru]'\n  AND rt.role = 'actor'\n  AND t.production_year > 2005\n  AND t.id = mc.movie_id\n  AND t.id =
ci.movie_id\n  AND ci.movie_id = mc.movie_id\n  AND chn.id = ci.person_role_id\n  AND rt.id = ci.role_id\n  AND cn.id =
mc.company_id\n  AND ct.id = mc.company_type_id;\n\n", "moz_ast": {"select": [{"value": {"min": "chn.name"}, "name":
"uncredited_voiced_character"}, {"value": {"min": "t.title"}, "name": "russian_movie"}], "from": [{"value": "char_name",
"name": "chn"}, {"value": "cast_info", "name": "ci"}, {"value": "company_name", "name": "cn"}, {"value": "company_type",
"name": "ct"}, {"value": "movie_companies", "name": "mc"}, {"value": "role_type", "name": "rt"}, {"value": "title",
"name": "t"}], "where": {"and": [{"like": ["ci.note", {"literal": "%(voice)%"}]}, {"like": ["ci.note", {"literal":
"%(uncredited)%"}]}, {"eq": ["cn.country_code", {"literal": "[ru]"}]}, {"eq": ["rt.role", {"literal": "actor"}]}, {"gt":
["t.production_year", 2005]}, {"eq": ["t.id", "mc.movie_id"]}, {"eq": ["t.id", "ci.movie_id"]}, {"eq": ["ci.movie_id",
"mc.movie_id"]}, {"eq": ["chn.id", "ci.person_role_id"]}, {"eq": ["rt.id", "ci.role_id"]}, {"eq": ["cn.id", "mc
.company_id"]}, {"eq": ["ct.id", "mc.company_type_id"]}]}}, "rel_num": 7, "cost": 512686.28}, "10b.sql": {"sql": "SELECT
MIN(chn.name) AS character,\n       MIN(t.title) AS russian_mov_with_actor_producer\nFROM char_name AS chn,\n
cast_info AS ci,\n     company_name AS cn,\n     company_type AS ct,\n     movie_companies AS mc,\n     role_type AS rt,\n
     title AS t\nWHERE ci.note LIKE '%(producer)%'\n  AND cn.country_code = '[ru]'\n  AND rt.role = 'actor'\n  AND
t.production_year > 2010\n  AND t.id = mc.movie_id\n  AND t.id = ci.movie_id\n  AND ci.movie_id = mc.movie_id\n  AND
chn.id = ci.person_role_id\n  AND rt.id = ci.role_id\n  AND cn.id = mc.company_id\n  AND ct.id = mc.company_type_id;\n\n",
 "moz_ast": {"select": [{"value": {"min": "chn.name"}, "name": "character"}, {"value": {"min": "t.title"}, "name":
"russian_mov_with_actor_producer"}], "from": [{"value": "char_name", "name": "chn"}, {"value": "cast_info", "name": "ci"},
 {"value": "company_name", "name": "cn"}, {"value": "company_type", "name": "ct"}, {"value": "movie_companies", "name":
"mc"}, {"value": "role_type", "name": "rt"}, {"value": "title", "name": "t"}], "where": {"and": [{"like": ["ci.note",
{"literal": "%(producer)%"}]}, {"eq": ["cn.country_code", {"literal": "[ru]"}]}, {"eq": ["rt.role", {"literal":
"actor"}]}, {"gt": ["t.production_year", 2010]}, {"eq": ["t.id", "mc.movie_id"]}, {"eq": ["t.id", "ci.movie_id"]}, {"eq":
["ci.movie_id", "mc.movie_id"]}, {"eq": ["chn.id", "ci.person_role_id"]}, {"eq": ["rt.id", "ci.role_id"]}, {"eq":
["cn.id", "mc.company_id"]}, {"eq": ["ct.id", "mc.company_type_id"]}]}}, "rel_num": 7, "cost": 480530.19}, "10c.sql":
{"sql": "SELECT MIN(chn.name) AS character,\n       MIN(t.title) AS movie_with_american_producer\nFROM char_name AS chn,
```


Рисунок 3.12. Створений датасет у вигляді JSON документу.

Поглянемо на результати навчання. Для прикладу візьмемо наступний набір запитів: 3a.sql, 3b.sql, 3c.sql. Поставимо загальну кількість епізодів на 900. Результати будуть подані у вигляді графіків для кожного епізоду окремо, де по осі x епізоди, а по осі y вартість побудованого запиту. Поглянемо на результати навчання:

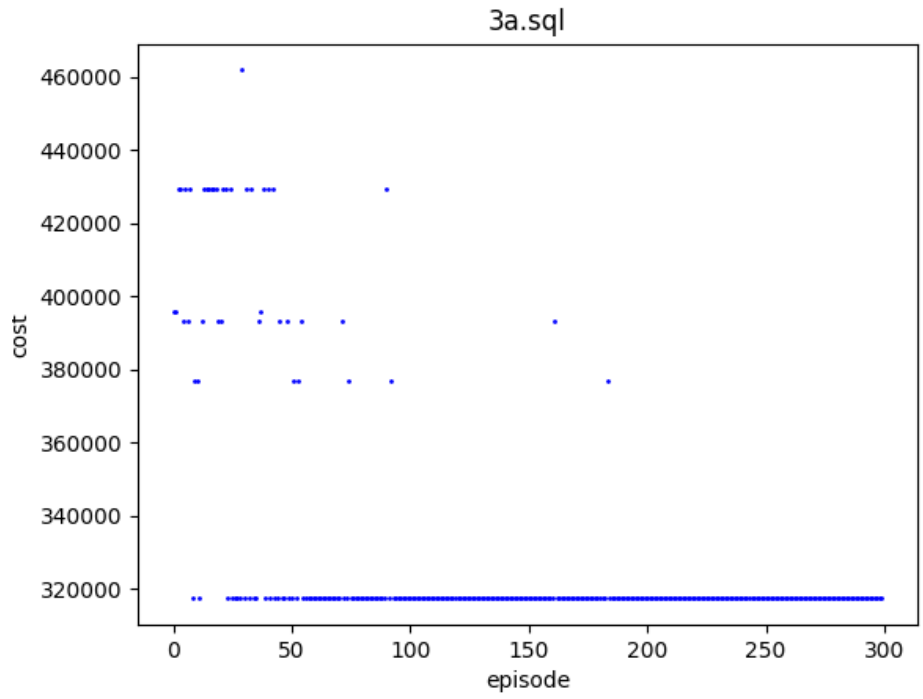


Рисунок 3.13. Результати навчання для запиту 3a.sql.

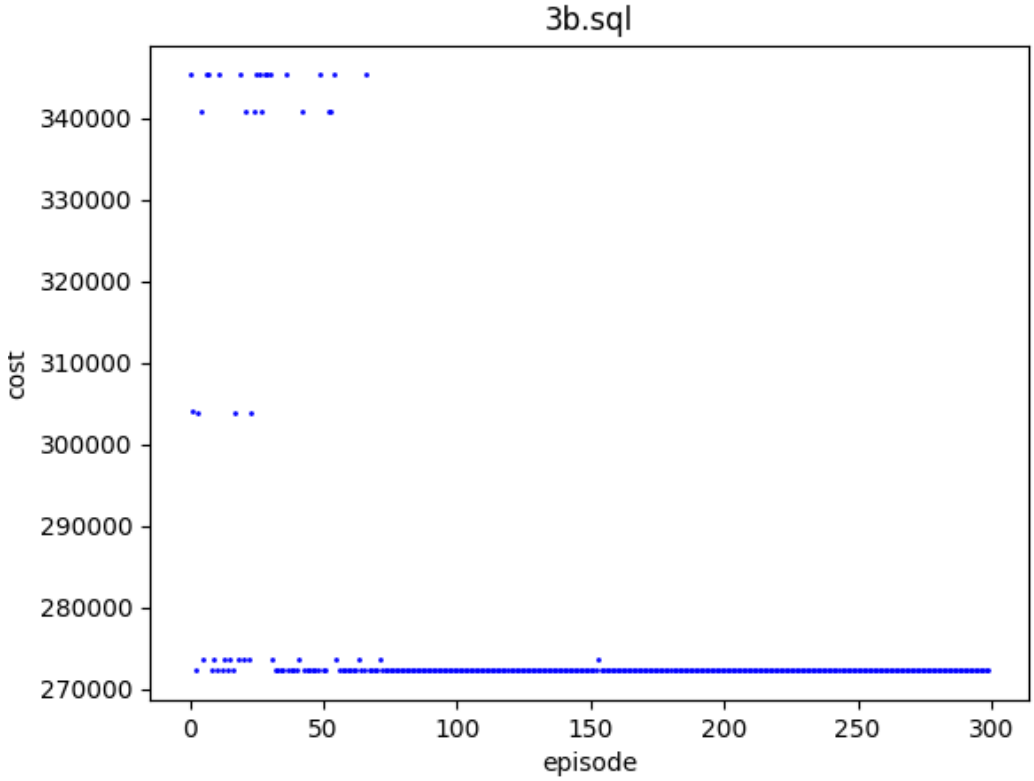


Рисунок 3.14. Результати навчання для запиту 3b.sql.

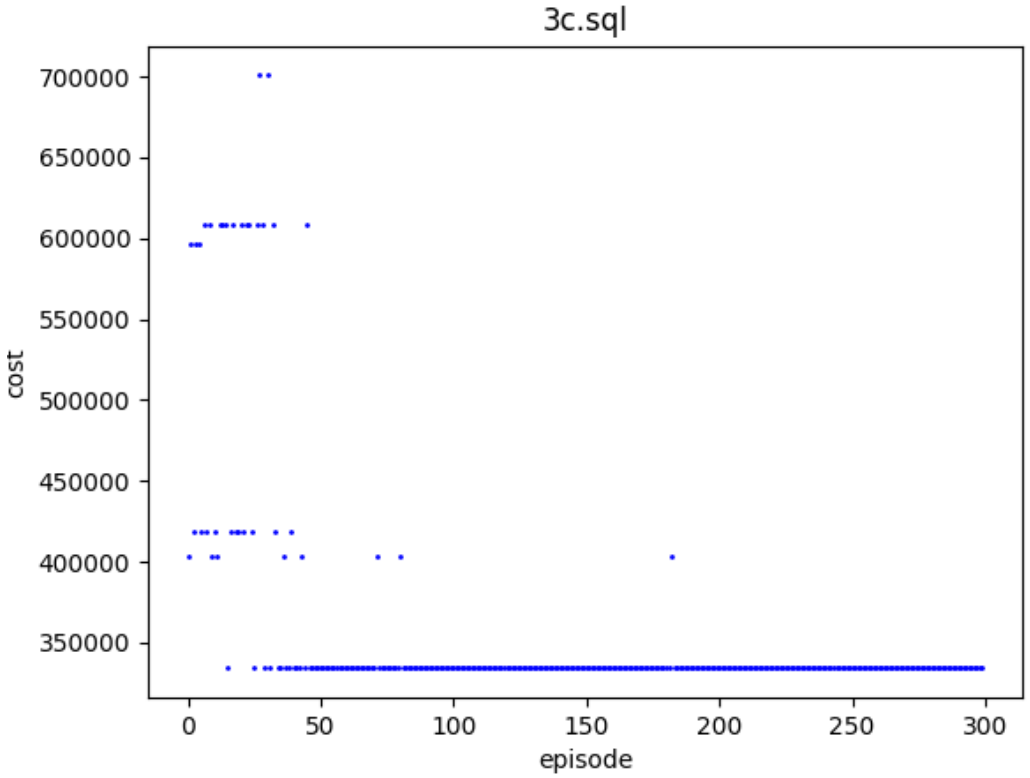


Рисунок 3.15. Результати навчання для запиту 3b.sql.

Складемо таблицю у якій порівняємо отримані результати з вартістю запиту, який складає СУБД PostgreSQL.

Запит	Вартість за результами навчання	Результат СУБД PostgeSQL
3a.sql	316616	317251
3b.sql	273152	272263
3c.sql	332972	333025

Таблиця 3.1. Порівняння результатів навчання з роботою СУБД.

Як ми можемо бачити, для даного набору запитів, агент після навчання працює на тому ж рівні ефективності, що і СУБД, іноді навіть ефективніше, але ми не побачили значного приросту. Це може бути пов’язано з невеликою кількістю з’єднань у даній групі запитів, тому виконаємо навчання на наборі запитів з великою кількістю з’єднань, а саме: 19a.sql, 19b.sql, 19c.sql. Поглянемо на результати:

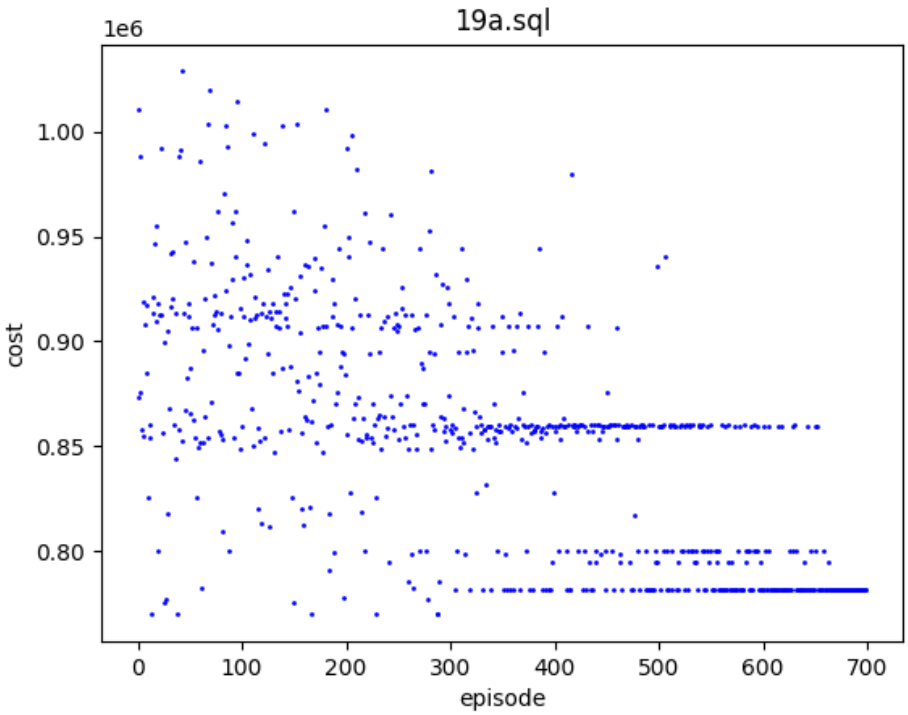


Рисунок 3.16. Результати навчання для запиту 19a.sql.

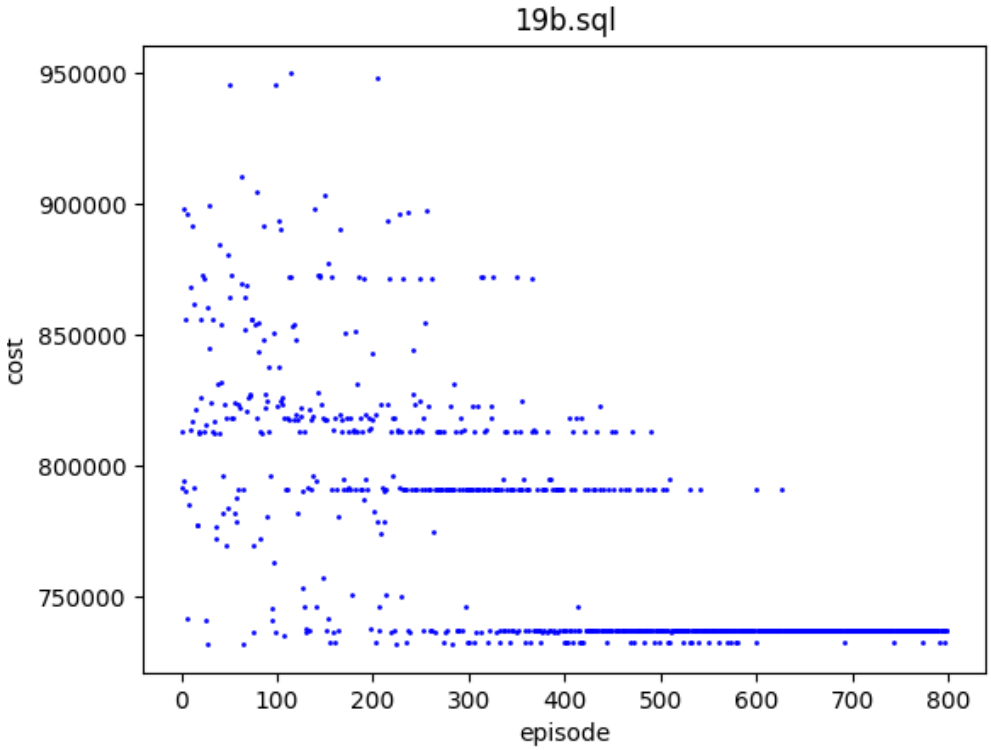


Рисунок 3.17. Результати навчання для запиту 19b.sql.

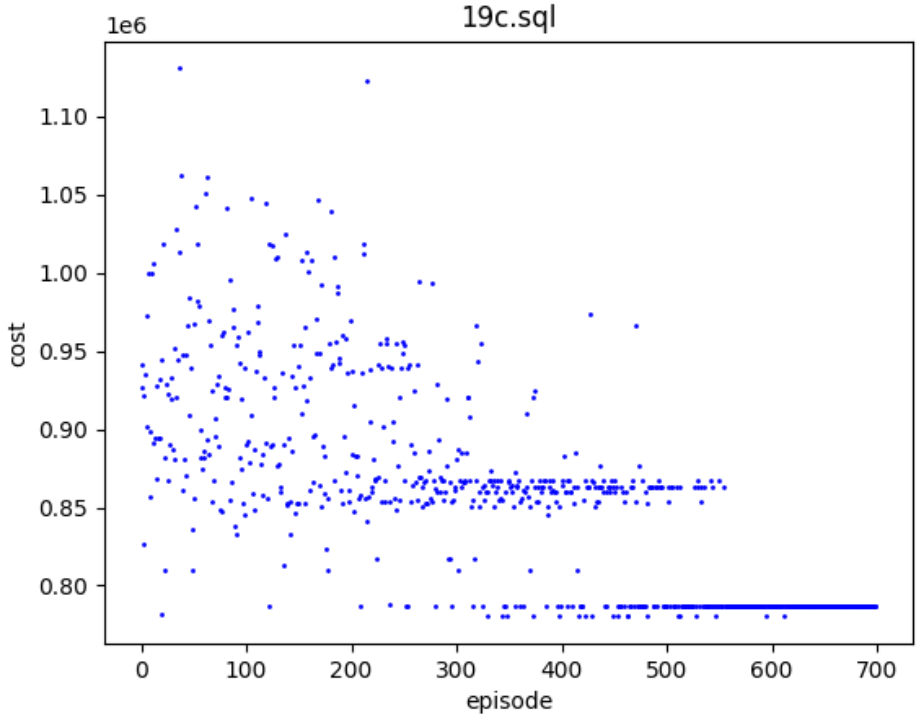


Рисунок 3.18. Результати навчання для запиту 19c.sql.

Складемо таблицю у якій порівняємо отримані результати з вартістю запиту, який складає СУБД PostgreSQL:

Запит	Вартість за результами навчання	Результат СУБД PostgeSQL
19a.sql	751255	767747
19b.sql	730011	731337
19c.sql	765125	766012

Таблиця 3.2. Порівняння результатів навчання з роботою СУБД.

Як бачимо був отриманий хоч і не великий, ми досі залишаємося на рівні СУБД, але все ж приріст ефективності. Збільшити його можна було б якби агент вибирав не лише порядок з’єднання, але і метод приєднання.

Провівши тестування для різних груп запитів приходимо до середніх результатів навчання агента:

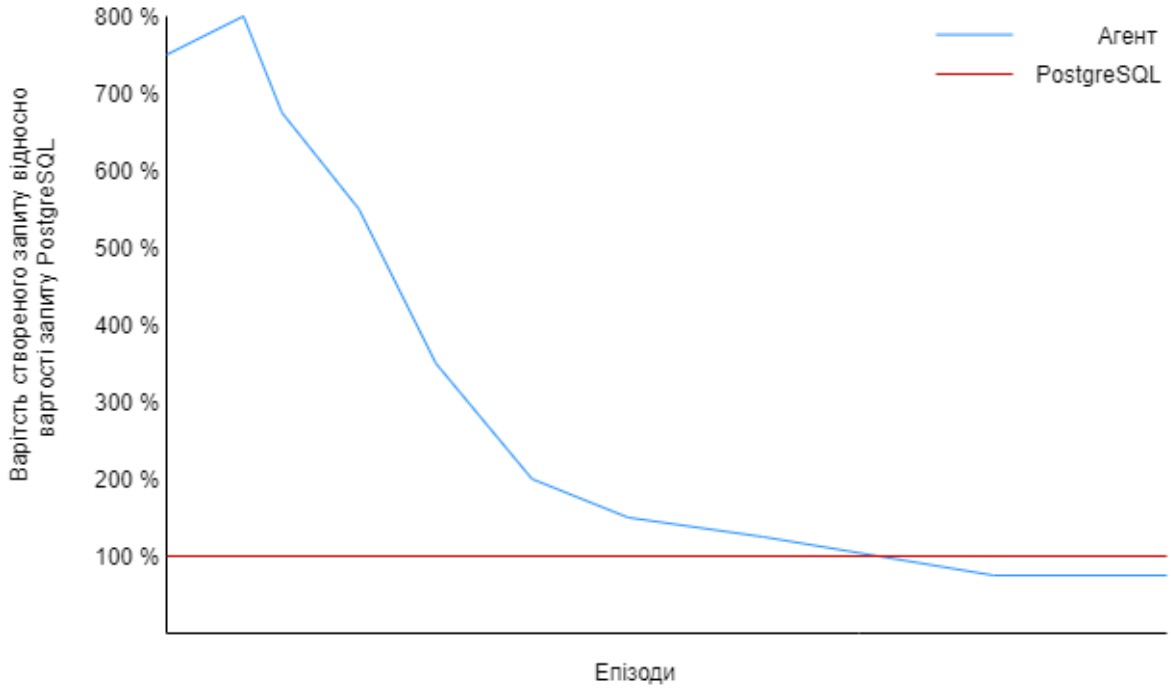


Рисунок 3.19. Результати тестування.

Отже робимо висновок, що використання навчання з підкріпленням є ефективним для вирішення проблеми пошуку порядку з’єднань для запитів з великою кількістю з’єднань та великих БД.

3.6. Висновки до розділу.

У розділі були розглянуті інструменти розробки використані під час реалізації програмного продукту. Була розроблена програма, що використовуючи модель навчання з підкріпленням оптимізує запити до СУБД створюючи оптимальний порядок з’єднань. Були проведені тести розробленого додатку та виявлено ефективність використання даного методу для вирішення поставленої проблеми. Були зроблені висновки про те що для запитів з великим числом з’єднань використання навчання з підкріпленням є ефективним.

РОЗДІЛ 4. ФУНКЦІОНАЛЬНО ВАРТІСНИЙ АНАЛІЗ ПРОДУКТУ.

У розділі буде проведена оцінка основних характеристик розробленого програмного продукту, який розроблений для тестування можливості використання нейронних мереж для оптимізації запитів до СУБД.

4.1. Постановка задачі проектування.

Розробити програмний продукт, що буде займатися оптимізацією запитів до СУБД використовуючи навчання з підкріпленням. Продукт призначений для проведення тестування ефективності застосування навчання з підкріпленням та нейромереж для вирішення проблеми оптимізації запитів, продукт також призначений для великих визначених баз даних у якості оптимізатора порядку з'єднань для запитів з великою кількістю з'єднань.

4.2. Обгрунтування функцій програмного продукту.

Головною функцією F_0 є – розробка програмного продукту що буде вирішувати проблему створення порядку з'єднань. Виділяємо наступні функції беручи за основу головну:

F_1 – вибір мови програмування для розробки програмного продукту;

F_2 – вибір фреймворку, що буде використовуватись під час розробки;

F_3 – вибір СУБД для проведення тестування;

F_4 – вибір середовища розробки.

Кожна з вище вказаних функцій має декілька варіантів реалізації:

Функція F_1:

- Python;
- C++.

Функція F_2:

- Tensorforce;
- створення власної реалізації навчання з підкріпленням використовуючи TensorFlow.

Функція F_3:

- PostgreSQL;
- MySQL.

Функція F_4:

- PyCharm;
- Visual Studio Code.

Морфологічна карта системи:

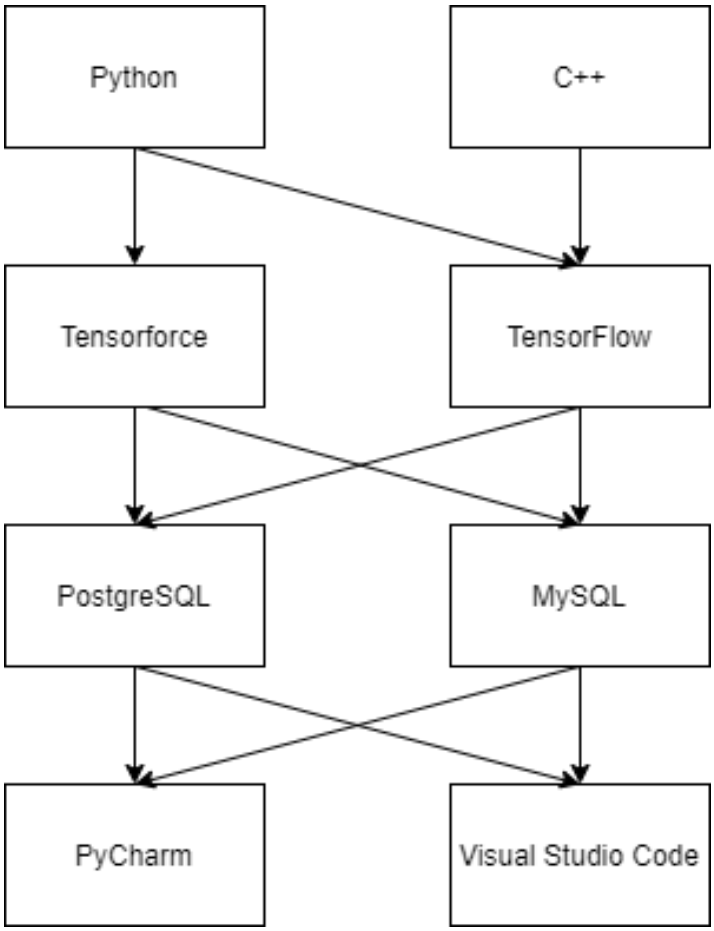


Рисунок 4.1. Морфологічна карта.

Використовуючи морфологічну карту побудуємо позитивно-негативну матрицю (табл 4.1):

Функції	Варіанти реалізації	Переваги	Недоліки
F_1	*А*	Швидкість розробки, читаємий код, доступність великої кількості бібліотек, кросплатформенсть.	Відносно низька швидкість роботи, проблема посилюється на великих даних. Проблеми з паралелізмом, через GIL.
	Б	Швидке виконання коду.	Швидкість розробки значно нижча за швидкість розробки на Python.
F_2	*А*	Швидкість та простота розробки, готовий функціонал.	Обмежені можливості.
	Б	Великі можливості.	Багато часу на розробку.
F_3	*А*	Можливість оптимальної роботи з великою кількістю даних.	Складніша за аналоги.
	Б	Простіша у вивченні за PostgreSQL.	Виникають проблеми при роботі з великим об’ємом даних.
F_4	*А*	Багато інструментів для розробки.	Громіздка IDE, доступна лише для Python.
	Б	Простота налаштування та використання.	Відсутність багатьох корисних інструментів.

Таблиця 4.1. Позитивно-негативна матриця.

На основі створеної матриці, робимо висновок, що деякі варіанти можемо відкинути одразу, так як вони не повністю відповідають усім умовам задачі.

F1. Для нас більш важлива швидкість розробки та читаємість коду, тому віддаємо перевагу варіанту А.

F2. Для швидшої розробки додатку доцільніше використовувати фреймворк з готовими інструментами для розробки, тому обираємо варіант А.

F4. Так як обрана мова програмування – Python, доцільніше використовувати потужну IDE зо всіма необхідними інстументами. Тому обираємо варіант А.

Тому маємо лише два варіанти:

F1а – F2а – F3а – F4а

F1а – F2а – F3б – F4а

Далі проведемо оцінку обраних варіантів.

4.3. Обґрунтування системи параметрів ПП.

Для характеристики програмного продукту далі використовуємо наступні параметри:

$X1$ – час розробки програмного продукту;

$X2$ – необхідний для виконання та збереження даних об’єм пам’яті;

$X3$ – час роботи програмного продукту, обробки даних;

$X4$ – точність роботи навченого агента.

Будемо розглядати три варіанти значень параметрів: гірше, середнє, краще. Обираємо параметри відповідно до вимог замовника та умов експлуатації розробленого програмного продукту

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірше	середнє	краще
Швидкість розробки програмного продукту.	X1	год	20	15	10
Об’єм пам’яті .	X2	Мб	1024	512	256
Швидкодія мови програмування, швидкість обробки даних.	X3	мс	10	6	4
Точність роботи навченого агента.	X4	%	50	75	95

Таблиця 4.2. Система параметрів додатку.

За даними таблиці побудуємо графіки:

Рисунок 4.2. Час на розробку програмного продукту.

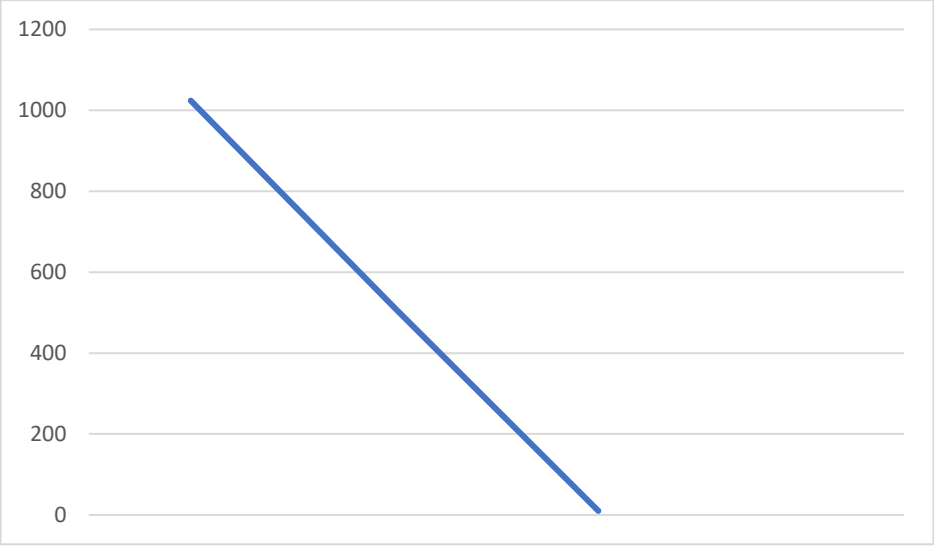


Рисунок 4.3. Об’єм пам’яті.

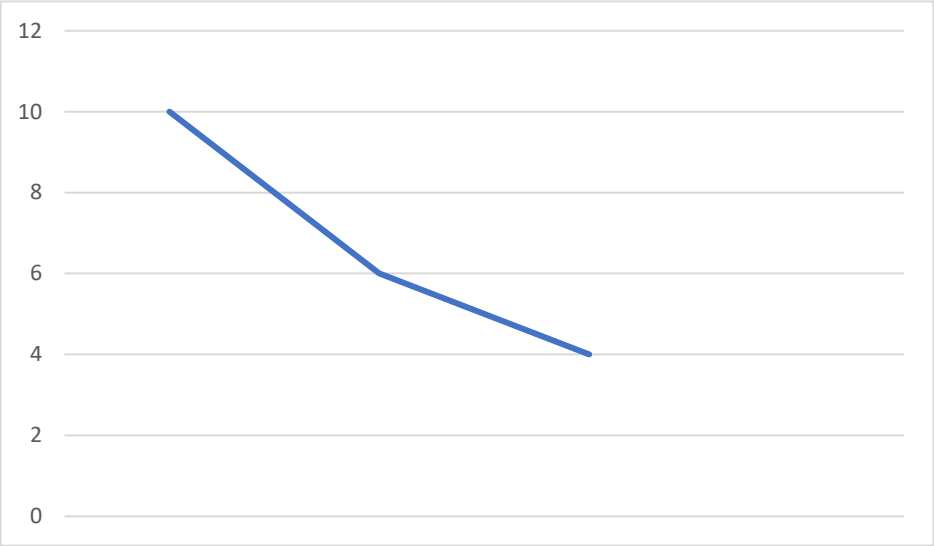


Рисунок 4.4. Швидкодія мови програмування.

Рисунок 4.5. Точність розробки навченого агента.

Далі методом попарного порівняння визначемо значимість кожного параметру. Нехай оцінку буде проводити експертна комісія із 7 людей. Нехай 1 ранг буде означат и те, що даний параметр не є сильно важливим для вирішення задачі написання програмного продукту, а 4 ранг – параметр є значущим.

параметр	Ранг параметра відповідно оцінці експерта							Сума Рангів	відхилення	Квадрат відхилення
	1	2	3	4	5	6	7			
X1	2	1	1	2	1	2	1	10	-7,5	56,25
X2	1	2	2	1	2	2	2	12	-5,5	30,25
X3	3	4	3	3	3	3	3	22	4,5	20,25
X4	4	3	4	4	4	3	4	26	8,5	72,25
Разом	10	10	10	10	10	10	10	70	0	179

Таблиця 4.3. Результати ранжування параметрі.

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 179}{7^2(4^3 - 4)} = 0{,}73 > W_k = 0{,}67. \qquad (4.1)$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Далі користуючись результатами виконаного ранжування проведемо попарне порівняння усіх параметрів та результати занесемо у наступну таблицю 4.4.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	>	<	<	>	<	=	<	<	1,5
X1 і X3	<	<	<	<	<	<	<	<	1,5
X1 і X4	<	<	<	<	<	<	<	<	1,5
X2 і X3	<	<	<	<	<	<	<	<	1,5
X2 і X4	<	<	<	<	<	<	<	<	1,5
X3 і X4	<	>	<	<	<	=	<	<	1,5

Таблиця 4.4. Попарне порівняння параметрів.

Далі для кожного параметра розрахуємо коефіцієнт вагомості за наступними формулами:

$$K_{\text{ві}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \tag{4.2}$$

$$b_i = \sum_{i=1}^{N} a_{ij} \tag{4.3}$$

Далі починаючи з другого кроку порахуємо відносні оцінки використовуючи наступні формули

$$K_{\text{ві}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \tag{4.4}$$

$$b_i' = \sum_{i=1}^{N} a_{ij} b_j \tag{4.5}$$

Параметриx_i	Параметриx_j				Перша ітер.		Друга ітер.	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$
X1	1,0	1,5	1,5	1,5	5,5	0,34	21,25	0,36
X2	0,5	1,0	1,5	1,5	4,5	0,28	16,25	0,28
X3	0,5	0,5	1,0	1,5	3,5	0,22	12,25	0,2
X4	0,5	0,5	0,5	1,0	2,5	0,16	9,25	0,16
Всього:					16	1	59	1

Таблиця 4.5. Розрахунок коефіцієнтів вагомості.

4.4. Аналіз рівня якості варіантів реалізації функцій.

Врахувавши усі дані з порівнянь варіантів будемо розглядати наступні варіанти використання ПП:

1. F1а – F2а – F3а – F4а
2. F1а – F2а – F3б – F4а

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	А	X1	10	4	0,26	1,4
F2	А	X4	90	8	0,32	1,6
F3	А	X3	6	5	0,27	1,2
	Б	X3	10	2	0,26	0,68
F3	А	X2	1024	6	0,21	1,7

Таблиця 4.6. Розрахунок показників рівня якості заданих варіантів.

Тепер можна визначити рівень якості кожного з отриманих варіантів

К1 = 1,4 + 1,6 + 1,2 + 1,7 = 5,9;

К2 = 1,4 + 1,6 + 0,68 + 1,7 = 5,38.

Як можемо бачити з вище представлених розрахунків, краще є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.5. Економічний аналіз варіантів розробки ПП.

Для визначення вартості розробки програмного продукту буде проведений розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка алгоритму для подальшого створення програми.
2. Спираючись на розроблений алгоритм розробити програму.
3. Використовуючи датасет провести навчання агента.

Завдання 1 відповідає групі Б відповідно до ступеня новизни, та до 1 групи за складністю. Завдання 2 відноситься до групи В для 1 варіанту та до групи Б для другого, за складністю до групи 1. 3 завдання відповідає групі В та складності 2.

Для завдання 1 використовуємо нормативно-довідкову інформацію, завдання 2 та 3 – бази данних.

Для завдання 1 маємо: $Т_р = 64$ людино-днів, $К_п = 1{,}021$, $К_{ск} = 1$, $К_{ст} = 0{,}8$

Отримаємо: $Т_1 = 64 * 1{,}021 * 1 * 0{,}8 = 52{,}3$ людино-дні

Провторимо для завдань 2 і 3:

Для завдання 2 маємо: $Т_р = 43$ людино-днів, $К_п = 0{,}68$, $К_{ск} = 1$, $К_{ст} = 0{,}8$

Отримаємо: $Т_2 = 43 * 0{,}68 * 1 * 0{,}8 = 23{,}4$ людино-днів для 1 варіанту, та

$Т_2 = 68 * 1{,}01 * 1 * 0{,}8 = 55$ для 2 варіанту

Для завдання 3 маємо: $Т_р = 12$ людино-днів, $К_п = 0{,}6$, $К_{ск} = 1$, $К_{ст} = 0{,}8$

Отримаємо: $Т_3 = 12 * 0{,}6 * 1 * 0{,}8 = 5{,}76$ людино-днів

Складемно трудомісткість завдань, для кожного з варіантів

$T_I = (52{,}3 + 23{,}4 + 5{,}76) * 8 = 651{,}7$ людино-годин

$T_{II} = (52{,}3 + 55 + 5{,}76) * 8 = 904{,}5$ людино-годин

Варіант II має більшу трудомісткість.

В розробці бере участь один програміст з окладом 15000 грн. і адмін бази даних з окладом 11000 грн. Визначимо зарплату за годину:

$$C_ч = \frac{15000 + 11000}{2 \cdot 21 \cdot 8} = 77{,}4 \text{ грн.}$$

Розрахуємо заробітну плату розробників за варіантами:

I. $C_{зп} = 651{,}7 * 77{,}4 * 1{,}2 = 60529{,}9$ грн.

II. $C_{зп} = 904{,}5 * 77{,}4 * 1{,}2 = 84009{,}96$ грн

Відрахування на єдиний соціальний внесок(22%):

I. $C_{від} = 60529{,}9 * 0{,}22 = \ 13316{,}58$ грн

II. $C_{від} = 84009{,}96 * 0{,}22 = 18482{,}2$ грн

Визначимо витрати на оплату однієї машино-години (См). Оскільки одна машина обслуговує одного програміста з окладом 15000 грн. і коефіцієнтом зайнятості 0,2, отримаємо:

$C_г = 12 * 15000 * 0{,}2 = 36000$ грн

Порахуємо додаткову заробітну плату:

$C_{зп} = 36000 * (1 + 0{,}2) = 43200$ грн

Відрахування на соціальний внесок

$C_{від} = 43200 * 0{,}22 = 9504$ грн

Амортизаційні відрахування отримуємо при амортизації 25% і вартості машини – 16000 грн.

$C_а = 1{,}15 * 0{,}25 * 16000 = 4600$ грн

Витрати на ремонт:

$$С_р = 1{,}15 * 0{,}05 * 16000 = 920 \text{ грн}$$

Ефективний годинниковий фонд часу ПК за рік:

$$Т_{еф} = (365 - 142 - 16) * 8 * 0{,}9 = 1490{,}4 \text{ год}$$

Витрати на оплату електроенергії:

$$С_{ел} = 1490{,}4 * 0{,}3 * 0{,}68 * 3{,}51 = 1009{,}19 \text{ грн}$$

Накладні витрати:

$$С_н = 16000 * 0{,}67 = 10720 \text{ грн}$$

Річні експлутаційні витрати:

$$С_{експ} = 43200 + 9504 + 4600 + 920 + 1009{,}19 + 10720 = 69953{,}19 \text{ грн}$$

Таким чином собівартість однієї ЕОМ складає:

$$С_{м-г} = 69953{,}19/1490{,}4 = 46{,}94 \text{ грн}$$

Оскільки всі роботи по розробці програмного продукту ведуться она ПК, розрахуємо витрати в залежності від обраного варіанту:

I. $С_м = 46{,}94 * 651{,}7 = 30590{,}80$ грн

II. $С_м = 46{,}94 * 904{,}5 = 42457{,}23$ грн

Накладні витрати(67%):

I. $С_н = 60529{,}9 * 0{,}67 = 40555{,}03$ грн

II. $С_н = 84009{,}96 * 0{,}67 = 56286{,}67$ грн

Вартість розробки програмного продукту відповідно до варіанту:

I. $С_{пп} = 60529{,}9 + 13316{,}58 + 30590{,}80 + 40555{,}03 = 144992{,}31$ грн

II. $С_{пп} = 84009{,}96 + 18482{,}2 + 42457{,}23 + 56286{,}67 = 201236{,}06$ грн

Вибір кращого варіанту

$$К_{тер1} = \frac{5,9}{144992,31} = 4,07 * 10^{-5}$$

$$К_{тер2} = \frac{5,38}{201236,06} = 2,67 * 10^{-5}$$

Найефективнішим виявився варіант 1 з коефіцієнтом техніко-економічного рівня рівня

$$К_{тер1} = 4,07 * 10^{-5}$$

4.5. Висновки до розділу.

Отже після того як був виконаний функціонально-вартісний аналіз було виявлено, що перший варіант є оптимальним. Показник першого варіанту $4,07 * 10^{-5}$ виявився найбільшим. Отже обраний варіант має наступні параметри:

- реалізація програмного продукту виконується на мові програмування Python;
- в якості фреймворку для розробки використовуємо Tensorforce;
- в якості СУБД для розгортання БД та проведення тестування обираємо PostgreSQL;
- розробка програмного продукту буде проведена у IDE Pycharm.

ВИСНОВКИ ПО РОБОТІ.

Отже, під час виконання дипломної роботи були розглянуті та проаналізовані сучасні СУБД з акцентом на оптимізацію запитів. Було розглянуте поняття плану запиту та методів для його оптимізації створення. Після проведення аналізу створення плану запиту була виявлена основна його проблема – оптимізатори запитів сучасних СУБД не використовують попередній досвід оптимізації, а самі алгоритми оптимізації, такі як, динамічне програмування та генетичний метод, не завжди дають оптимальний план запиту, через що можуть виникати проблеми з ефективністю СУБД.

Для вирішення даної проблеми було прийняте рішення використовувати нейронні мережі. Після більш детального дослідження проблеми було прийняте рішення використовувати модель навчання з підкріпленням, що ефективно застосовується у багатьох сучасних сферах. Таке рішення було прийнято, так як задачу оптимізації запитів, а саме її аспект – створення порядку з’єднань, можна представити у вигляді задачі прийняття рішень Маркова.

Для того щоб ми могли використовувати навчання з підкріпленням необхідно визначити компоненти цієї моделі, а саме: середовище, агента, вектора стану та нагорода. У якості середовища ми беремо взаємодію нашої програми з БД, у якості вектору стану ми використовуємо атрибути та зв’язки запиту, що розглядаємо, а також матрицю що відображає процес створення порядку з’єднань, у якості винагороди беремо оцінку СУБД вартості створеного запиту, у умовних одиницях, відносно оцінки плану запиту створеного СУБД.

Після визначення усіх необхідних компонентів був розроблений програмний продукт, що оптимізує запити шляхом створення порядку з’єднань за допомогою агента навчання з підкріпленням. Далі було проведено

навчання моделі та тестування, для цих цілей використовувалась база даних imdb та, перетворений для наших цілей, датасет join-order-benchmark.

Отже, отримані тести дають зрозуміти що використання нейромереж на запитах з великою кількістю з’єднань є ефективним. Було виявлено що оптимізатор запитів, що розроблений на основі навчання з підкріпленням, після навчання, ефективно вирішує проблему оптимізації порядку з’єднань у плані запиту. Для ще більшого підвищення ефективності оптимізації даним методом можна створити оптимізатор, що не лише створює порядок з’єднань у запиті, але і обирає метод з’єднань. Але є і недоліки розробленого рішення, а саме те що для кожної бази даних необхідно використовувати індивідуального агента, якого необхідно навчати окремо, а також при змінах у схемі бази даних агента необхідно перевчити. Отже робимо висновок що використання нейронних мереж для оптимізації запитів є ефективним рішенням, що дозволить пришвидшити час створення запиту та його виконання, але його доцільно використовувати для баз даних з чітко визначеною схемою. Використовуючи навчання з підкріпленням для вирішення цієї проблеми можна обійти у швидкодії сучасні динамічні алгоритми що для цього створені.

СПИСОК ЛІТЕРАТУРИ

1. Система управління базами даних. URL: https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B0%D0%BC%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85
2. План виконання запиту. URL: https://ru.wikipedia.org/wiki/%D0%9F%D0%BB%D0%B0%D0%BD_%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81%D0%B0
3. Оптимізація запитів. URL: https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F_%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D1%96%D0%B2
4. Методи оптимізації виконання запитів у реляційних СУБД. URL: http://citforum.ru/database/articles/art_26.shtml
5. Query Optimizer in PostgreSQL. URL: https://bitnine.net/blog-postgresql/query-optimizer-in-postgresql/
6. Применение машинного обучения для увеличения производительности PostgreSQL. URL: https://habr.com/ru/company/postgrespro/blog/273199/
7. Reinforcement learning. URL: https://en.wikipedia.org/wiki/Reinforcement_learning
8. RL – Proximal Policy Optimization (PPO) Explained. URL: https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12
9. Vanilla Deep Q Networks. URL: https://towardsdatascience.com/dqn-part-1-vanilla-deep-q-networks-6eb4a00febfb

10. Deep Q-Learning Tutorial: minDQN. URL: https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc
11. 5 Things You Need to Know about Reinforcement Learning. URL: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html
12. Using Reinforcement Learning to Produce Better Join Ordering Strategy. URL: https://towardsdatascience.com/using-reinforcement-learning-to-produce-better-join-ordering-strategy-2fd2761ebf3a
13. SQL Query Optimization Meets Deep Reinforcement Learning. URL: https://rise.cs.berkeley.edu/blog/sql-query-optimization-meets-deep-reinforcement-learning/
14. Python wiki. URL: https://ru.wikipedia.org/wiki/Python
15. TensorFlow. URL: https://www.tensorflow.org/
16. TensorFlow wiki. URL: https://ru.wikipedia.org/wiki/TensorFlow
17. PostgreSQL документація. URL: https://www.postgresql.org/docs/
18. PostgreSQL wiki. URL: https://ru.wikipedia.org/wiki/PostgreSQL
19. Tensorforce. URL: https://tensorforce.readthedocs.io/en/latest/index.html
20. Join-order-benchmark. URL: https://github.com/gregrahn/join-order-benchmark
21. How Good Are Query Optimizers, Really? URL: http://www.vldb.org/pvldb/vol9/p204-leis.pdf
22. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. URL: https://www.cl.cam.ac.uk/~ey204/teaching/ACS/R244_2020_2021/papers/li_VLDB_2019.pdf

ДОДАТОК А. ЛІСТИНК ПРОГРАМИ.

main.py

```
from tensorforce.agents import Agent
from tensorforce.execution import Runner

from environment import JoinOptimizerEnv
from memory import ModelMemory
from config.network_spec import network


def main():
    num_episodes = 800

    memory = ModelMemory()
    environment = JoinOptimizerEnv(num_episodes, ['19b.sql'], 'loop', memory)

    agent = Agent.create(
        agent='ppo',
        environment=environment,
        max_episode_timesteps=100,
        batch_size=64,
        network=network,
        memory=10000,
        update_frequency=4,
    )

    runner = Runner(agent=agent, environment=environment)

    runner.run(num_episodes=num_episodes)
    runner.close()

    memory.draw_memory_results()


if __name__ == '__main__':
    main()
```

database.py

```
import json
from pathlib import Path
from typing import Dict, Tuple, List

import psycopg2

from config import dbconfig


class Database:
    DATASET_PATH = 'updated_dataset/json_dataset.json'

    def __init__(self):
        self.con = self.connect()

        self.queries_dict = self.get_queries_dict()
```

```
        self.tables, self.tables_attrs = self.get_tables_info()
        self.relations, self.relations_tables, self.relations_attrs =
self.get_relations_info()
        self.attrs = self.get_all_attrs()

    @staticmethod
    def connect() -> psycopg2.connect:
        c_str = f'host={dbconfig.HOST} ' \
                f'port=5432 ' \
                f'dbname={dbconfig.DATABASE} ' \
                f'user={dbconfig.USER} ' \
                f'password={dbconfig.PASSWORD}'

        try:
            con = psycopg2.connect(c_str)
            return con
        except psycopg2.Error as e:
            print(f'Connection error - {e}')

    def get_queries_dict(self) -> Dict:
        return json.loads(Path(self.DATASET_PATH).read_text())

    def get_tables_info(self) -> Tuple[List[str], Dict]:
        query = Path('sql_files/tables_info_q.sql').read_text()

        cursor = self.con.cursor()
        cursor.execute(query)
        tables_info = cursor.fetchall()
        cursor.close()

        tables_attrs = {}
        for tb in tables_info:
            if tb[0] not in tables_attrs.keys():
                tables_attrs[tb[0]] = [tb[1]]
            else:
                tables_attrs[tb[0]].append(tb[1])

        tables = list(tables_attrs.keys())

        return tables, tables_attrs

    def get_relations_info(self) -> Tuple[List[str], Dict, Dict]:
        relations = set()
        relations_tables = {}
        relations_attrs = {}

        for query in self.queries_dict.values():
            for elem in query['moz_ast']['from']:
                rel = elem['name']
                tb = elem['value']

                relations.add(rel)
                relations_tables[rel] = tb
                relations_attrs[rel] = self.tables_attrs[tb]

        return sorted(list(relations)), relations_tables, relations_attrs

    def get_all_attrs(self) -> List[str]:
        attrs = []
        for key, val in self.relations_attrs.items():
            attrs += [f'{key}.{v}' for v in val]
```

```
        return sorted(attrs)

    def get_query_cost(self, query: str) -> float:
        join_collapse_limit = 'SET join_collapse_limit = 1'
        query = f'{join_collapse_limit};EXPLAIN (FORMAT JSON) {query};'

        cursor = self.con.cursor()
        cursor.execute(query)
        row = cursor.fetchone()
        cursor.close()

        return row[0][0]['Plan']['Total Cost']
```

environment.py

```
from typing import Dict, List, Tuple

import numpy as np
from tensorforce import Environment

from database import Database
from state_vector import StateVector
from query import Query
from memory import ModelMemory


class JoinOptimizerEnv(Environment):
    def __init__(self, total_episodes: int, queries_to_run: List['str'], mode: str,
memory: ModelMemory):
        super().__init__()

        self.db = Database()
        self.rel_num = len(self.db.relations)
        self.attrs_num = len(self.db.attrs)

        self.total_episodes = total_episodes
        self.mode = mode
        self.queries_to_run = queries_to_run
        self.c_query = None
        self.c_query_ind = 0
        self.c_state_vector = None
        self.c_state = None
        self.memory = memory

        self.c_episode = 0
        self.c_actions = []
        self.episodes_per_query = int(self.total_episodes / len(queries_to_run))

    def states(self) -> Dict:
        return {
            'attrs_vector': dict(
                type='float',
                shape=(self.attrs_num, ),
            ),
            'join_tree_matrix': dict(
                type='float',
                shape=(self.rel_num*self.rel_num, ),
            ),
```

```
            'join_matrix': dict(
                type='float',
                shape=(self.rel_num * self.rel_num, )
            )
        }

    def actions(self) -> Dict:
        return dict(type='int', num_values=45)

    def reset(self, num_parallel=None) -> Dict:
        query_name = self.get_query_name()
        self.c_query = Query(self.db, query_name)

        if not self.memory.in_memory(query_name):
            self.memory.add_query_to_memory(query_name, self.c_query.query_cost)

        self.c_actions = []

        self.c_state_vector = StateVector(self.c_query, self.db.relations,
self.db.attrs)
        self.c_state = self.c_state_vector.create_states()

        print(f'\nEpisode query: {query_name}')

        return self.c_state

    def get_query_name(self) -> str:
        if self.mode == 'one_q':
            query_name = self.queries_to_run[self.c_query_ind]
            self.episodes_per_query -= 1
            if not self.episodes_per_query:
                self.c_query_ind += 1
                self.episodes_per_query = int(self.total_episodes /
len(self.queries_to_run))
        elif self.mode == 'loop':
            query_name = self.queries_to_run[self.c_query_ind]
            self.c_query_ind = (self.c_query_ind + 1) % len(self.queries_to_run)
        else:
            print('Invalid mode.')
            raise ValueError

        return query_name

    def execute(self, actions: int) -> Tuple[Dict, int, int]:
        print(actions)
        self.c_state['join_tree_matrix'] =
self.c_state['join_tree_matrix'].reshape(self.rel_num, self.rel_num)
        self.c_state['join_matrix'] =
self.c_state['join_matrix'].reshape(self.rel_num, self.rel_num)

        valid_actions = self.get_c_actions()
        terminal = self.is_terminal(len(valid_actions))

        c_action = valid_actions[actions % len(valid_actions)]
        self.c_actions.append(c_action)
        self.update_join_matrix(c_action)

        if terminal:
            if len(valid_actions) == 2:
                last_action = self.get_c_actions()
                self.c_actions.append(last_action[0])
```

```
            join_ordering = self.get_join_ordering()
            new_query = self.c_query.reconstruct_query(join_ordering)
            reward = self.get_reward(new_query)
        else:
            reward = 0

        self.c_state['join_tree_matrix'] = self.c_state['join_tree_matrix'].flatten()
        self.c_state['join_matrix'] = self.c_state['join_matrix'].flatten()

        return self.c_state, terminal, reward

    @staticmethod
    def is_terminal(actions_len: int) -> bool:
        return actions_len <= 2

    def get_c_actions(self) -> List[Tuple[int, int]]:
        actions = []

        for i in range(0, self.rel_num):
            for j in range(i + 1, self.rel_num):
                for ind1, el1 in enumerate(self.c_state['join_matrix'][i]):
                    for ind2, el2 in enumerate(self.c_state['join_matrix'][j]):
                        if all([el1, el2,
self.c_state['join_tree_matrix'][ind1][ind2]]):
                            if (i, j) not in actions:
                                actions.append((i, j))

        return list(actions)

    def update_join_matrix(self, action: Tuple[int, int]) -> None:
        vec1 = self.c_state['join_matrix'][action[0]]
        vec2 = self.c_state['join_matrix'][action[1]]

        for i in range(0, self.rel_num):
            if vec1[i]:
                vec1[i] = vec1[i] / 2
            if vec2[i]:
                vec1[i] = vec2[i] / 2

        self.c_state['join_matrix'] = np.delete(self.c_state['join_matrix'],
action[1], 0)

        add_vec = np.zeros(shape=self.rel_num, dtype=float)
        self.c_state['join_matrix'] = np.vstack((self.c_state['join_matrix'],
add_vec))

    def get_join_ordering(self) -> List:
        c_relations = self.db.relations.copy()

        join_ordering = []
        for action in self.c_actions:
            c_relations[action[0]] = [c_relations[action[0]], c_relations[action[1]]]
            c_relations.pop(action[1])

            join_ordering = c_relations[action[0]]

        return join_ordering

    def get_reward(self, query: str) -> float:
        query_cost = self.db.get_query_cost(query)
```

```
        reward = 1 / query_cost * 100000

        mean, std = self.memory.get_mean_and_std(self.c_query.query_name)
        reward = ((reward - mean) / (std + 0.1))

        print(f'Query cost - {query_cost}')
        print(f'Mean - {mean}, Std - {std}, Reward - {reward}')

        self.memory.append_to(self.c_query.query_name, 'costs', query_cost)
        self.memory.append_to(self.c_query.query_name, 'rewards', reward)

        return reward

    def max_episode_timesteps(self):
        return 100
```

memory.py

```
from typing import Tuple

import numpy as np
import matplotlib.pyplot as plt


class ModelMemory:
    def __init__(self):
        self.memory = {}

    def in_memory(self, query_name: str) -> bool:
        return query_name in self.memory.keys()

    def add_query_to_memory(self, query_name: str, query_cost) -> None:
        self.memory[query_name] = {
            'rewards': [0],
            'costs': [],
            'p_cost': query_cost
        }

    def append_to(self, query_name: str, memory_attr: str, data: float) -> None:
        self.memory[query_name][memory_attr].append(data)

    def get_mean_and_std(self, query_name: str) -> Tuple[float, float]:
        arr = np.array(self.memory[query_name]['rewards'])
        return arr.mean(), arr.std()

    def draw_memory_results(self) -> None:
        for ind, (query_name, elem) in enumerate(self.memory.items()):
            plt.figure(ind + 1)

            costs = np.array(elem['costs'])
            plt.xlabel('episode')
            plt.ylabel('cost')
            plt.title(query_name)
            plt.plot(costs, 'b.', MarkerSize=2)

        plt.show(block=True)
```

state_vector.py

```python
import numpy as np
from typing import Dict, Tuple, List

from query import Query
from database import Database


class StateVector:
    def __init__(self, query: Query, relations: List[str], attrs: List[str]):
        self.query = query

        self.attrs_vector = self.get_attrs_vector(attrs)
        self.join_tree_matrix, self.join_matrix = self.get_join_tree_mxs(relations)
        print(1)

    def get_attrs_vector(self, attrs: List[str]) -> np.array:
        attrs_vector = np.zeros(len(attrs), dtype=float)

        for attr in self.query.query_attrs:
            try:
                attrs_vector[attrs.index(attr)] = 1
            except ValueError:
                print(f'Unknown attr - {attr}')

        return attrs_vector

    def get_join_tree_mxs(self, relations: List[str]) -> Tuple[np.array, np.array]:
        relations_num = len(relations)

        join_tree_matrix = np.zeros((relations_num, relations_num), dtype=float)
        join_matrix = np.zeros((relations_num, relations_num), dtype=float)

        for join_pair in self.query.query_attrs_joins:
            try:
                j1_ind = relations.index(join_pair[0])
                j2_ind = relations.index(join_pair[1])

                join_tree_matrix[j1_ind][j2_ind] = 1
                join_tree_matrix[j2_ind][j1_ind] = 1
                join_matrix[j1_ind][j1_ind] = 1
                join_matrix[j2_ind][j2_ind] = 1
            except ValueError:
                print(f'Unknown pair - {join_pair}')

        return join_tree_matrix, join_matrix

    def create_states(self) -> Dict:
        states = {
            'attrs_vector': self.attrs_vector,
            'join_tree_matrix': np.array(self.join_tree_matrix).flatten(),
            'join_matrix': np.array(self.join_matrix).flatten(),
        }

        return states


if __name__ == '__main__':
    db = Database()
```

```
    _query = Query(db, '3a.sql')
    sv = StateVector(_query, db.relations, db.attrs)
```

query.py

```
from typing import Tuple, List, Dict

from database import Database


class Query:
    def __init__(self, db: Database, query_name: str):
        self.query_name = query_name
        self.db = db
        self.query = db.queries_dict[query_name]
        self.query_ast = self.query['moz_ast']
        self.query_cost = self.query['cost']

        self.query_relations = self.get_query_relations()
        self.query_attrs = self.get_query_attrs()
        self.query_attrs_joins, self.query_aj_with_rel = self.get_query_attrs_joins()
        self.attrs_aliases = self.get_aliases_to_tables()
        self.rel_to_al = {}

        self.join_counter = 0

        self.select_operators = {'min': 'MIN', 'max': 'MAX'}
        self.where_operators = {
            'in': 'IN',
            'like': 'LIKE',
            'not_like': 'NOT LIKE',
            'between': 'BETWEEN',
            'eq': '=',
            'neq': '!=',
            'gt': '>',
            'lt': '<',
            'gte': '>=',
            'lte': '<=',
        }

    def get_query_relations(self) -> List[str]:
        return [rel['name'] for rel in self.query_ast['from']]

    def get_aliases_to_tables(self):
        return {rel['name']: rel['value'] for rel in self.query_ast['from']}

    def get_query_attrs(self) -> List[str]:
        query_attrs = []
        for cond in self.query_ast['where']['and']:
            if list(cond.keys())[0] == 'exists':
                query_attrs.append(list(cond.values())[0])
            elif list(cond.keys())[0] in ['or', 'and']:
                for cond1 in cond[list(cond.keys())[0]]:
                    if list(cond1.keys())[0] == 'exists':
                        query_attrs.append(list(cond1.values())[0])
                    else:
                        for el in list(cond1.values())[0]:
                            if isinstance(el, str):
                                query_attrs.append(el)
```

```
            else:
                for el in list(cond.values())[0]:
                    if isinstance(el, str):
                        query_attrs.append(el)

        return list(set(query_attrs))

    def get_query_attrs_joins(self) -> Tuple[List[Tuple[str, str]], Dict]:
        query_attrs_joins = []
        query_aj_with_rel = {}

        for cond in self.query_ast['where']['and']:
            if list(cond.keys())[0] == 'eq' and isinstance(cond['eq'][0], str) and
isinstance(cond['eq'][1], str):
                ars = (cond['eq'][0].split('.')[0], cond['eq'][1].split('.')[0])
                rls = (cond['eq'][0].split('.')[1], cond['eq'][1].split('.')[1])
                query_attrs_joins.append(ars)
                query_aj_with_rel[ars] = rls
                query_aj_with_rel[ars[-1::-1]] = rls[-1::-1]

        return query_attrs_joins, query_aj_with_rel

    def reconstruct_query(self, ordering: List) -> str:
        relations_aliases = {rel: [rel] for rel in self.query_relations}

        joins_q, join_alias = self.rec_construct_joins(ordering, relations_aliases)
        select_part = self.get_select_query_part(join_alias)
        where_part = self.get_where_query_part(join_alias)

        query = f'{select_part} FROM {joins_q} {where_part}'
        return query

    def rec_construct_joins(self, tree: List or str, relations_aliases: Dict) ->
Tuple[str, str]:
        if isinstance(tree, str):
            return tree, tree

        left, left_al = self.rec_construct_joins(tree[0], relations_aliases)
        right, right_al = self.rec_construct_joins(tree[1], relations_aliases)

        new_al = f'join{self.join_counter}'
        self.join_counter += 1

        self.rel_to_al[left_al] = new_al
        self.rel_to_al[right_al] = new_al

        relations_aliases[new_al] = [left_al, right_al]

        attr1, attr2 = self.query_aj_with_rel[(left_al, right_al)]

        if left == left_al:
            left = f'{self.attrs_aliases[left]} AS {left}'
        if right == right_al:
            right = f'{self.attrs_aliases[right]} AS {right}'

        sel_a_str = self.get_selection_attrs(relations_aliases, left_al, right_al)
        self.upd_relation_aliases((left_al, right_al), new_al)

        sub_q = f'(SELECT {sel_a_str} FROM {left} JOIN {right} ON {left_al}.{attr1} =
{right_al}.{attr2}) {new_al}'
```

```
        return sub_q, new_al

    def upd_relation_aliases(self, old_p: Tuple, new_al: str):
        self.query_aj_with_rel.pop(old_p)
        self.query_aj_with_rel.pop(old_p[-1::-1])

        d_keys = list(self.query_aj_with_rel.keys())
        for k1, k2 in d_keys:
            r1, attr1 = k1, self.query_aj_with_rel[(k1, k2)][0]
            r2, attr2 = k2, self.query_aj_with_rel[(k1, k2)][1]

            if any([k1 == old_p[0], k1 == old_p[1]]):
                r1 = new_al
                attr1 = f'{k1}_{attr1}'
            if any([k2 == old_p[0], k2 == old_p[1]]):
                r2 = new_al
                attr2 = f'{k2}_{attr2}'
            if k1 != r1 or k2 != r2:
                self.query_aj_with_rel.pop((k1, k2))
                self.query_aj_with_rel[(r1, r2)] = (attr1, attr2)

    def get_selection_attrs(self, rel_als: Dict, left_al: str, right_al: str) -> str:
        sel_attrs = []

        self.rec_get_selection_attrs(sel_attrs, rel_als, '', left_al, left_al)
        self.rec_get_selection_attrs(sel_attrs, rel_als, '', right_al, right_al)

        selection_attrs_str = ', '.join(sel_attrs)

        return selection_attrs_str

    def rec_get_selection_attrs(self, sel_attrs: List[str], rel_als: Dict, path: str,
alias: str, base_alias: str):
        rel = rel_als[alias]
        if len(rel) > 1:
            for r in rel:
                path1 = path + r + '_'
                self.rec_get_selection_attrs(sel_attrs, rel_als, path1, r,
base_alias)
        else:
            attrs = self.db.relations_attrs[rel[0]]
            for attr in attrs:
                t = path + attr
                sel_attrs.append(f'{base_alias}.{t} AS {base_alias}_{t}')

    def create_attr_alias(self, attr: str, alias: str) -> str:
        rel, attr_ = attr.split('.')

        while rel in self.rel_to_al:
            attr_ = f'{rel}_{attr_}'
            rel = self.rel_to_al[rel]

        return f'{alias}.{attr_}'

    def get_select_query_part(self, join_alias: str) -> str:
        select_list = []
        select_ast = self.query_ast['select']

        if not isinstance(select_ast, list):
            select_ast = [select_ast]
```

```
        for el in select_ast:
            val = el['value']
            if not isinstance(val, str):
                key = list(val.keys())[0]
                alias = self.create_attr_alias(val[key], join_alias)

                val = f'{self.select_operators[key]}({alias})'
            else:
                val = self.create_attr_alias(val, join_alias)

            if 'name' in el.keys():
                name = f' AS {el["name"]}'
            else:
                name = ''

            select_list.append(f'{val}{name}')

        return 'SELECT ' + ', '.join(select_list)

    def get_where_query_part(self, join_alias: str) -> str:
        where_ast = self.query_ast['where']

        where_part = self.get_and_or_part(where_ast, join_alias)
        if where_part:
            return f'\nWHERE \n{where_part}'
        else:
            return ''

    def get_and_or_part(self, where_ast: Dict, join_alias: str) -> str:
        and_or_list = [[], []]
        for ind, op in enumerate(['and', 'or']):
            if op in where_ast:
                op_conds = where_ast[op]
                for cond in op_conds:
                    if not ('eq' in cond and isinstance(cond['eq'][0], str) and
isinstance(cond['eq'][1], str)):
                        and_or_list[ind].append(self.construct_condition(cond,
join_alias))

        and_p = ' AND '.join(and_or_list[0])
        or_p = ' OR '.join(and_or_list[1])

        return f'{and_p} \n{or_p}'

    def construct_condition(self, cond: Dict, join_alias: str) -> str:
        key = list(cond.keys())[0]
        if key in ['and', 'or']:
            return f'( {self.get_and_or_part(cond, join_alias)} )'
        elif key == 'between':
            rl_list = []
            for i in range(1, 3):
                if isinstance(cond[key][i], dict):
                    rl_list.append(f'\'{cond[key][i]["literal"]}\'')
                else:
                    rl_list.append(f'{cond[key][i]}')

            fin_value = ' AND '.join(rl_list)
        elif isinstance(cond[key][1], dict):
            let = cond[key][1]['literal']
            if isinstance(let, list):
                fin_value = ' ( \'' + '\', \''.join(let) + '\' ) '
```

```
            elif key == 'in':
                fin_value = f'( \'{let}\' )'
            else:
                fin_value = f'\'{let}\''
        elif isinstance(cond[key][1], int):
            fin_value = str(cond[key][1])
        elif key == 'exists':
            attr_alias = self.create_attr_alias(cond[key], join_alias)
            return f'{attr_alias} IS NOT NULL '
        else:
            fin_value = self.create_attr_alias(cond[key][1], join_alias)

        attr_alias = self.create_attr_alias(cond[key][0], join_alias)

        return f'{attr_alias} {self.where_operators[key]} {fin_value}'
```