Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

Інструменти та алгоритми індексації блокчейн платформ.
Текстова частина до курсової роботи
за спеціальністю «Комп'ютерні науки» - 122

Керівник курсової роботи
ас.
Гороховський К.С.
____________________ (Підпис)
"___" _____________ 2022 року
Виконав студент
КН-4 Грачов О. В.
"___" _____________ 2023 року

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

ас.

__________ Гороховський К.С.

„_____” _____________ 2023 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Грачову Олександру Віталійовичу

факультету інформатики 4 курсу бакалаврської програми

ТЕМА: **Інструменти та алгоритми індексації блокчейн платформ.**

Зміст ТЧ до курсової роботи:
Індивідуальне завдання
Вступ
Розділ 1. Дослідження та аналіз предметної області
Розділ 2. Розробка і опис застосунку
Розділ 3. Дослідження застосунку
Висновки
Список літератури
Додатки (за необхідністю)

Дата видачі „___” _________ 2023 р.

Керівник_______________

(підпис)

Завдання отримав_______________

(підпис)

Календарний план виконання роботи

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	01.10.2023	
2.	Огляд літератури за темою роботи	01.11.2023	
3.	Написання програмного застосунку	01.12.2023	
4.	Проведення дослідження	01.02.2024	
5.	Написання текстової частини	04.04.2024	
6.	Фінальна перевірка	01.05.2024	
7.	Захист курсової роботи	15.05.2024	

Студент ________________________________

Керівник ____________________________ “___ ___”_ _________2023

Зміст

Календарний план виконання роботи 3
Анотація 6
Вступ 7
Розділ 1. Загальний огляд і вступ 10
1.1 Загальні відомості 10
1.2 Збереження даних в блокчейн мережі 11
1.3 Вилучення даних з блокчейну 12
1.5 LevelDB як СУБД для вузла 16
1.6 Merkle Patricia Trie 18
1.7.1 Графові бази даних 19
1.7.2 Neo4j 19
1.7.3 Індексація в Neo4j 21
Розділ 2: Технічні деталі та вимоги 24
2.1 Головна мета застосунку 24
2.2 Опис користувачів системи 24
2.3 Технічні вимоги до функціональності системиДля користувачів: 24
2.4 Специфікації вимог до даних 24
Розділ 3. Розробка застосунку 26
3.1 Ідея 26
3.2 Обґрунтування вибору засобів розробки 26
3.2.1 Neo4j 26
3.2.2 Docker 27
3.2.3 Golang 27
3.2.4 Gin 27
3.2.5 IDE 28
3.3 Модель СУБД 28
3.4 Код прослуховувача 30
3.5 API 33
3.6 Дослідження швидкодії 37
3.6.1 Метод оцінки алгоритму 37
3.6.2 Таблиці порівняння роботи програми на різних об’ємах даних 41
3.6.3 Графіки логарифмічного масштабу порівняння роботи програми на різних об’ємах даних 43
3.6.4 Аналіз 45
3.7 Розгортка проекту 46

Висновки..47
Джерела: 48
Додатки..50
Додаток 1. Опис API 50
Додаток 2. Характеристики ПК..52

Анотація

Дипломна робота присвячена аналізу інструментів та алгоритмів індексації використовуваних у блокчейн-технологіях. Вивчення цих методик має на меті визначення їх ефективності та оптимальності для підвищення продуктивності систем збереження даних. Досліджено різні підходи до збереження індексованих даних, що дозволяють оптимізувати виконання запитів та спрощення процесів обробки даних.

Розроблено аналітичний огляд різних баз даних, таких як LevelDB, та їх модифікацій у контексті блокчейн, що дозволило виявити потенційні напрямки для подальшого вдосконалення технологій зберігання інформації. Реалізовано порівняльний аналіз ефективності використання індексів у таких системах.

Досліджено специфіку використання індексаційних алгоритмів у блокчейн, зокрема вивчення їх впливу на швидкість обробки та відповідність сучасним вимогам безпеки та масштабованості. Текстова частина роботи також описує необхідні теоретичні відомості про застосовані інструменти та методології.

Ключові слова: блокчейн, індексація, LevelDB, оптимізація, алгоритми, бази даних, технології зберігання.

Вступ

Актуальність теми

Актуальність теми дипломної роботи, яка зосереджена на аналізі інструментів та алгоритмів індексації в контексті блокчейн-технологій, полягає в зростаючій важливості цифрових технологій в сучасному світі. З огляду на постійне збільшення обсягів даних і вимог до їх обробки, ефективне управління даними стає критично важливим для оптимізації технологічних процесів у багатьох галузях, включаючи фінанси, медицину та логістику.

Зокрема, використання блокчейн-технологій зростає, оскільки вони забезпечують високий рівень безпеки та прозорості. Технології індексації відіграють ключову роль у забезпеченні швидкого доступу до даних, зниженні часу обробки запитів та підвищенні загальної продуктивності систем. Впровадження ефективних алгоритмів індексації може значно підвищити ефективність блокчейн-додатків, сприяючи їх ширшому впровадженню та використанню в різноманітних секторах економіки.

Також, з огляду на загальну тенденцію до цифровізації та автоматизації бізнес-процесів, вдосконалення індексаційних алгоритмів може допомогти організаціям краще впоратися з викликами, пов'язаними з великими даними та їх аналітикою. Таким чином, дослідження в цій області є своєчасним і важливим для розвитку інформаційних технологій і може сприяти реалізації нових можливостей для інновацій та підвищення конкурентоспроможності.

Об'єкт дослідження

Індексація даних у блокчейн-технологіях.

Предмет дослідження

Алгоритми та інструменти індексації, що використовуються для оптимізації запитів та зберігання даних у блокчейн-системах.

Мета дослідження
Аналізувати і оцінювати різні інструменти та алгоритми індексації з метою виявлення найефективніших методів для покращення продуктивності та оптимізації обробки даних в блокчейн-мережах.

Постановка задачі

1. **Аналіз існуючих інструментів індексації:** Вивчення поточних методів індексації використовуваних у блокчейн-технологіях для визначення їхніх сильних та слабких сторін.
2. **Оцінка алгоритмів в розглянутих інструментах:** Оцінка алгоритмів і підходів, що використовуються в сучасних інструментах індексації блокчейну
3. **Ідентифікація оптимальних локацій для дослідження та розробки програмного застосунку:** Ідентифікація та аналіз потенційних викликів, які існують в сучасних методах індексації використовуваних у блокчейн-технологіях.
4. **Створення власного застосунку:** Розробка власного програмного застосунку на основі графової бази даних Neo4j та мови програмування Golang.

Структура роботи:

Перший розділ: Загальний огляд і вступ

У цьому розділі надається загальна інформація про блокчейн та індексацію, розкриваються основні виклики та проблеми, пов'язані з індексацією в блокчейні. Викладається аналіз існуючих рішень та обґрунтування вибору підходу до вирішення виявлених проблем.

Розділ також включає огляд наявних на ринку рішень та їх аналіз, що допомагає виявити недоліки та потенціал для вдосконалення.

Другий розділ: Технічні деталі та вимоги

Детальний опис технічного завдання проєкту, включаючи опис користувачів, функціональні вимоги та вимоги до даних. У цьому розділі також розглядаються основні компоненти та архітектура системи, включаючи інтеграцію з блокчейн мережами та методи індексації.

Третій розділ: Розробка та імплементація

Вибір інструментів: Обгрунтування вибору Neo4j та Golang як основних інструментів для розробки застосунку. Включає порівняння з іншими інструментами та обґрунтування, чому обрані технології найкраще відповідають вимогам проєкту.

Детальний опис реалізації ключових компонентів системи, включаючи базу даних, серверну логіку, інтеграцію з блокчейнами, а також реалізацію індексаційних алгоритмів.

Наведення результатів роботи програми, демонстрація ефективності впроваджених рішень та їх впливу на продуктивність системи.

Розділ 1. Загальний огляд і вступ

1.1 Загальні відомості

Індексація блокчейну - це технологія, яка полегшує процес пошуку інформації, що зберігається в блокчейні. Замість того, щоб переглядати дані блок за блоком, індексація блокчейну дозволяє аналізувати інформацію і зберігати її в централізованій базі даних різними методами. Потім інформацію можна аналізувати, організовувати, індексувати (в контексті звичайних баз даних) чи опрацьовувати як і будь які інші дані в звичайній СУБД.

Блокчейн, як розподілений реєстр, зберігає дані. Часто виникає ситуація, що дані про транзакції, які зберігаються в ланцюжку, потребують пошуку. Наприклад, один зі сценаріїв може виникнути при використанні блокчейну, який зберігає дані про електронні медичні картки. Цілком можливо, що з часом нам знадобиться доступ до історії лікування певного пацієнта. У цьому випадку відповідні записи про пацієнта можуть бути розподілені між багатьма блоками. Пошук можна здійснювати нестандартно, переглядаючи пов'язаний список, починаючи з останнього блоку, у зворотному напрямку і збираючи відповідні записи. Звичайно, можливо, що під час пошуку буде відвідано багато нерелевантних блоків. Крім того, наявні API запити напряму до блокчейну нод залежать від платформи, є обмеженими і підтримують лише базові запити.

Ідея полягає в тому, щоб забезпечити швидкий механізм онлайн-запитів, який синхронізує запити та інтеграцію з більшістю блокчейнів, таких як Ethereum, Bitcoin і Hyperledger, а також мати можливість виконувати більш складні запити за допомогою певного синтаксису.

Як раз саме індексація блокчейну полегшує кінцевим користувачам і розробникам отримання необхідної інформації про історичні події в блокчейні. Завдяки індексації в блокчейні ми отримуємо безпеку даних в

мережі в поєднанні зі зручністю та ефективністю пошуку в централізованій базі даних поза мережею.

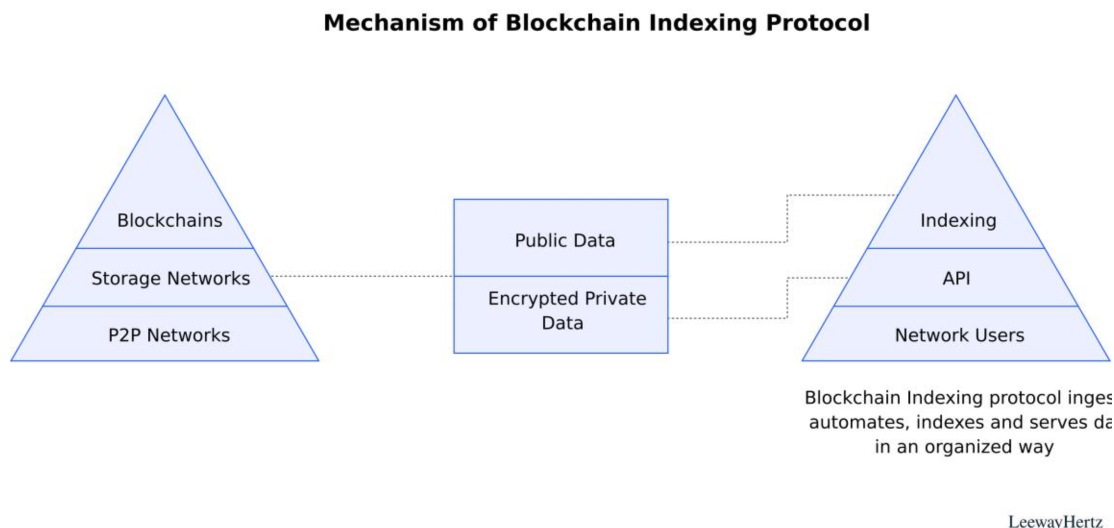


Рис. 1.1 Діграма механізму індексування блокчейн платформ [5]

1.2 Збереження даних в блокчейн мережі

По-перше варто вказати про те, де саме зберігається інформація в блокчейн мережах, і як до неї можна доступитися. Тут варто згадати про термін ноди (або вузла) в контексті блокчейну.

Вузли - це комп'ютери або сервери, що підключені до мережі і мають повну або часткову копію блокчейну. Кожен блок з транзакціями зберігається на кожному вузлі, що дозволяє забезпечити децентралізований характер системи та запобігти втраті даних у випадку відмови одного чи декількох вузлів.

Щоб отримати доступ до даних у блокчейні, користувач може взаємодіяти з будь-яким вузлом мережі за допомогою спеціального програмного забезпечення, такого як Ethereum node для мережі Ethereum або Bitcoin node для мережі Bitcoin. Ці програми дозволяють взаємодіяти з блокчейн мережею, запитуючи інформацію про блоки, транзакції, баланси та інше.

Наприклад, ви можете встановити Ethereum node, такий як Geth (go ethereum), на своєму комп'ютері, і він буде синхронізуватися з рештою мережі, отримуючи та зберігаючи дані про всі блоки та транзакції. Після цього ви можете використовувати цей вузол, щоб отримувати інформацію про стан мережі, виконувати транзакції та багато іншого.

Отже, кожен вузол у блокчейні має копію даних, і доступ до цих даних можливий через взаємодію з вузлом за допомогою спеціального програмного забезпечення. В свою чергу, кожен вузол може використовувати різне локальне програмне забезпечення для збереження даних у себе в пам’яті.

1.3 Вилучення даних з блокчейну

Для ілюстрації використання технологій блокчейну, оберемо Ethereum, яка є однією з найвизначніших та найширше використовуваних платформ серед розробників у цій галузі.

Для початку, нам треба мати ендпоїнт, за яким ми зможемо звертатись до провайдера для взаємодії з блокчейн мережею. Для цього ми можемо:

1. Або використати існуючі платформи, що надають послуги з доступом до мережі eth (наприклад, infura, alchemy).
2. Або встановити клієнтське вузлове програмне забезпечення на власний пристрій. Це дозволить стати активним вузлом в мережі. (Серед відомих клієнтів Ethereum можна виділити такі рішення як Geth, Nethermind і т.д. Кожен з яких має свої унікальні особливості та переваги. Важливим кроком після інсталяції клієнта є синхронізація з блокчейном, яка включає завантаження даних обсягом приблизно 700 ГБ станом на момент написання цієї роботи)

Після цього ми можемо створювати запити до мережі через існуючий ендпоїнт. Одним із зручних інструментів для взаємодії з блокчейном є

бібліотека web3.js, яка розроблена спеціально для JavaScript розробників. Web3.js дозволяє формулювати та відправляти запити до вузла, використовуючи локальну копію блокчейну для пошуку та отримання інформації.

Процедура отримання даних з блокчейну виглядає наступним чином:

- Підключення до вузла Ethereum через бібліотеку web3.
- Ідентифікація місця зберігання потрібних даних за допомогою унікального хеша, що діє як адреса, куди буде направлено запит.
- Відправлення структурованого запиту через web3 до вузла Ethereum, який аналізує запит та звертається до своєї локальної копії блокчейну.
- Отримання результатів: вузол виконує пошук по своїй базі даних блокчейну і повертає користувачу необхідну інформацію, що відповідає запиту.

```
// Імпорт бібліотеки
const Web3 = require('web3');

// Підключення до ноди
const web3 = new Web3(new
Web3.providers.HttpProvider(`https://${network}.infura.io/v3/${proce
ss.env.API_KEY}`));

// Запит дійсного номеру блоку
const block = web3.eth.getBlockNumber();

// Запит даних блоку
const data = await web3.eth.getBlock(0);

// Запит балансу
```

```
const balance = web3.eth.getBalance(“hash“);

// Запит кількості транзакцій
const count = web3.eth.getTransactionCount(“hash”);
```

Приклад використання web3.js

1.4 Статистика

Як можна побачити на діаграмі і таблиці в кінці розділу, найширш вживаною базою даних в блокчейн проектах залишається NoSQL LevelDB. LevelDB добре оптимізована для ефективного запису даних та під час виконання послідовних операцій запису або видалення.

Крім технічних аспектів, LevelDB підтримується Google, що забезпечує їй широку популярність та використання серед розробників на мовах програмування, таких як C++, Go, Java, та JavaScript. Більш детальний розгляд даної СУБД можна побачити в розділі 1.5

Другою по популярністю є RocksDB - розгалуження бази даних LevelDB, було розроблено Facebook для задоволення підвищених вимог до продуктивності сховищ ключ-значення, особливо у контексті використання флеш-пам'яті. Це забезпечує швидший доступ до даних завдяки оптимізації під флеш-сховища та ефективно адаптується до різних типів робочих навантажень.

Хоча основна структура збереження даних в RocksDB схожа на LevelDB, тобто використовуються впорядковані пари ключ-значення, RocksDB пропонує додаткові оптимізації, такі як колонкове зберігання в постійній пам'яті. Цей підхід дозволяє забезпечити більшу гнучкість і ефективність при зберіганні та обробці великих обсягів даних.

Також на графіку можна помітити інші СУБД, проте їх частка відносно незначна і тому в контексті цієї дипломної роботи вони не були розглянуті.

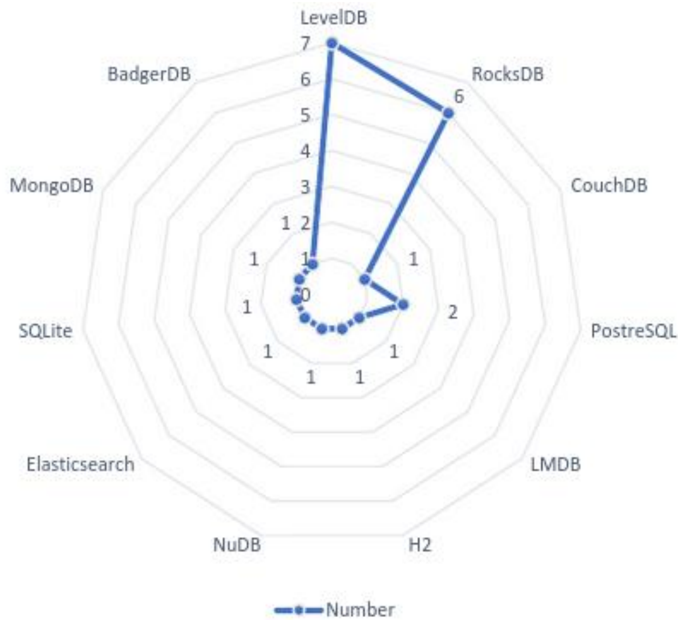


Рис. 1.1 Графік використання СУБД блокчейн платформами [3]

Project	Version	Database solution
Ethereum Geth	v1.9.10	LevelDB
Ethereum Parity	v2.6.8-beta	RocksDB
Ethereum Nethermind	v1.4.7	RocksDB
Hyperledger Besu	v1.4.0-beta2	RocksDB
Hyperledger Fabric	v2.0.0-beta	LevelDB / CouchDB
Hyperledger Burrow	v0.29.5	LevelDB
Hyperledger Indy	1.12.1	RocksDB
Hyperledger Iroha	v1.1.1	PostgreSQL
Hyperledger Sawtooth	v1.2.3	LMDB
Corda	release-os-4.4-RC01	H2
Quorum	v2.4.0	LevelDB
Multichain	v2.0.5	LevelDB
Bitcoin	v0.019.0.1	LevelDB
Ripple	v1.4.0	NuDB / RocksDB
BitShares	v3.3.2	Elasticsearch
Stellar	v12.2.0	SQl-based (SQLite / PostgreSQL)
BigchainDB	v2.0.0	MongoDB
Haskoin	v0.9.15	RocksDB
VeChain	v1.2.0	LevelDB
Insolar	v1.3.0	BadgerDB

Рис. 1.2 Таблиця використання СУБД блокчейн платформами [3]

1.5 LevelDB як СУБД для вузла

Варто більш детально розглянути одну з найбільш вживаних СУБД для вузлів блокчейну - LevelDB. Основною перевагою використання є його здатність працювати з великими обсягами даних завдяки одній особливості цієї бази даних - лог-структуроване злиття дерев (LSM). В основі цієї технології лежить принцип що частина даних зберігається в оперативній пам'яті до певного об'єму, після чого перекидуються на диск. Завдяки цьому, LevelDB стає ідеальним вибором для блокчейн систем, де історія транзакцій постійно збільшується.

Також, із особливостей збереження даних, варто виділити використання RLP - кодування даних блоків для збереження як значення в СУБД. Причина очевидна - в LevelDB у значення ключа може бути тільки одне значення, а блок має декілька полів для збереження, тож потрібний метод для кодування всієї інформації блоку в одну суцільну стрічку, тут RLP і стає у нагоді.

Крім того, частою практикою є винесення певних даних в окрему таблицю. Наприклад, для швидшого отримання списку гаманців в мережі можна винести адреси гаманців в окрему таблицю

Також цікаво дослідити інфраструктурну організацію платформ. Наприклад, згідно з джерелом (https://bitcoincoredocs.com/files.html)[12], система Bitcoin використовує декілька ключових директорій для забезпечення своєї функціональності та зберігання критично важливих даних. Ці директорії включають:

- blocks/blk*.dat: У директорії blocks/ зберігаються самі блоки у бінарному форматі. Кожен файл blk*.dat представляє собою набір блоків, що зберігаються для ефективного доступу та пошуку.

- blocks/index/*: Ці файли буквально забезпечує індексування в базі даних, а також швидкий доступ до метаданих, таких як: розмір блоків, час їх створення та інші характеристики, що допомагають у взаємодії з блокчейном. Без нього пошук блоків був би надзвичайно повільним процесом.

- chainstate/*: Ці файли містять критичні дані про всі поточні UTXO (Unspent Transaction Outputs) та включають важливі метадані про транзакції, з яких ці виходи походять. Завдяки точному визначенню наявних непотрачених виходів, ця схема зберігання відіграє роль у процесі верифікації нових блоків та транзакцій, забезпечуючи цілісність та безперервність блокчейн-мережі.

- blocks/rev.dat:* Файли rev*.dat вказують на "revision" або перегляд видалених блоків. Вони містять видалені блоки та їхні дані. Ці файли також пов'язані з LevelDB та допомагають у відновленні видалених блоків за потреби.

А також інші, з якими можна детальніше ознайомитись за джерелом: (https://bitcoincoredocs.com/files.html) [12]

Отже, забезпечуючи швидку та ефективну роботу з даними, LevelDB є чудовим вибором для вузлів мережі та допомагає підтримувати стабільну і надійну роботу блокчейн системи.

1.6 Merkle Patricia Trie

Окрім всього описаного вище, не можна не згадати Merkle Patricia Trie. Це структура даних, для забезпечення ефективної верифікації консистенції даних у розподіленій мережі. Подібна технологія дозволяє двом вузлам швидко перевірити, чи їхні копії блокчейну є ідентичними, що є критично важливим під час синхронізації.

Для чого це потрібно і чому просто не перевіряти дані блок за блоком? По одній причині - ефективність. Дерево Меркла дозволяє мінімізувати кількість обмінюваних даних між вузлами, оптимізуючи трафік та зберігаючи цілісність інформації в мережі.

Це бінарне дерево, листям якого є хеші об'єктів, що зберігаються. Для побудови наступного рівня попарно беруться хеші двох сусідніх листків, конкатенуються і обчислюється хеш від результату конкатенації. Таким чином, корінь дерева є хешем всього дерева. Якщо прибрати або додати елемент, корінь зміниться.

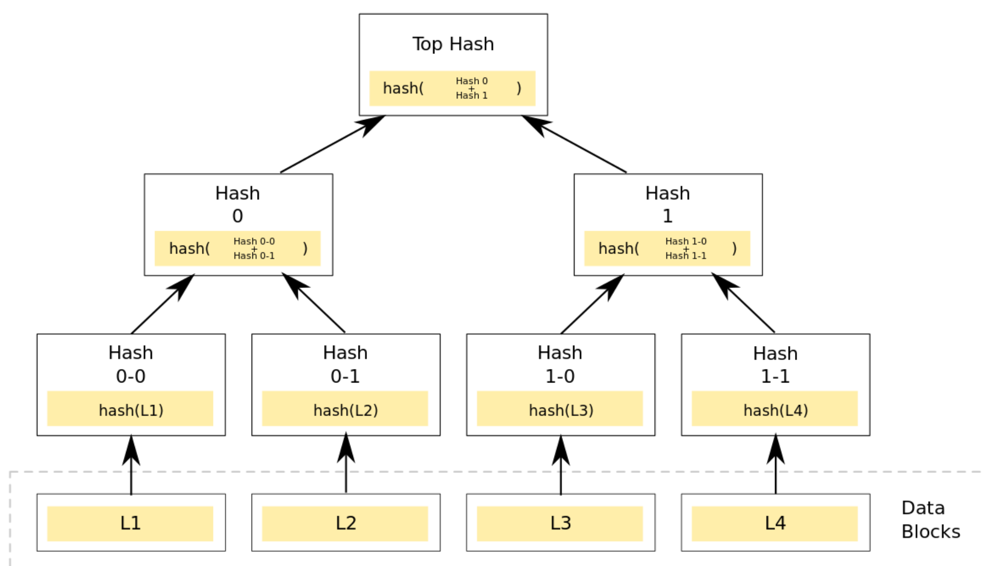


Рис. 1.3 Графічне відображення алгоритму Mercle tree [13]

1.7.1 Графові бази даних

Як альтернатива звичайним ключ-значення базам даних, існують інші підходи до збереження даних. Один із таких підходів - графові бази даних. Важко не погодитись, що природа блокчейну (ланцюга блоків) трохи співпадає з графом.

Окрім того, блокчейн в обличчі графових баз даних може набувати різних форм, а не лише ланцюга блоків. Це може бути одним із вирішень проблеми конфліктів злиття даних з різних вузлів. Саме тому тему графових БД було вирішено дослідити ближче.

1.7.2 Neo4j

Як приклад графових БД будемо розглядати Neo4j. Це NoSQL, високомасштабована графова база даних, що відповідає вимогам ACID-сумісного транзакційного бекенду для розробки програм та реалізує високоефективну модель графа на рівні зберігання, використовуючи вказівники для навігації та обходу графічної структури. З погляду продуктивності, Neo4j демонструє стабільність та ефективність у реальному часі при обробці багатошарових запитів до обширних та взаємопов'язаних наборів даних.

Також слід відзначити універсальність моделі графа, яка гнучко адаптується для надання оптимальних рішень, враховуючи потреби користувача. Мова запитів Cypher, яка є декларативною та аналогічною SQL, оптимізована для операцій з графами, та зараз використовується в інших базах даних, таких як SAP HANA Graph та Redis graph.

Модель графа властивостей Neo4j організовує дані у вигляді вузлів, відносин і властивостей. Вузли є сутностями в графі, які можуть мати будь-яку кількість атрибутів (пари ключ-значення), що називаються

властивостями. Їх також можна класифікувати за мітками, кожна з яких представляє конкретну роль для вузлів, що позначені нею.

Два семантично пов'язаних вузли можна з'єднати напрямленим ребром. Ребра характеризуються своїм типом і, як і вузли, також можуть мати властивості. Додатково, через ефективний спосіб зберігання будь-яну кількість або тип ребер можна спільно використовувати двома вузлами без втрати продуктивності.

Neo4j також пропонує ростучу, відкриту бібліотеку графових алгоритмів, які оптимізовані для швидких результатів. З мінімальним чи взагалі без кодування ці алгоритми розкривають приховані патерни та структури в збережених з'єднаних даних щодо пошуку шляхів, центральності та виявлення зв’язків.

Крім того, в наявних інструментах присутній Neo4j Browser - графічний інтерфейс користувача (GUI), який можна запустити через веб-браузер, дозволяє проводити запити, візуалізацію та взаємодію з даними. Усі ці можливості роблять Neo4j ідеальним інструментом для представлення, візуалізації та аналізу важливих та високорівнево зв'язаних даних блокчейну.

1.7.3 Індексація в Neo4j

Індекс в контексті СУБД - це копія деяких даних у базі даних, яка створюється з метою полегшення та прискорення пошуку пов'язаних даних. Це може призводити до додаткового використання простору для зберігання та сповільнення записів. Визначення, що індексувати, і що не індексувати, є важливим завданням, оскільки воно впливає на продуктивність та обсяг даних.

При розгляді індексації в графових базах даних, як Neo4j, важливо враховувати різні аспекти цього процесу. Після створення індексу, система управління базою даних (DBMS), відповідає за його керування та підтримку.

Neo4j автоматично використовуватиме створений та піднятий в онлайн індекс для оптимізації запитів та полегшення пошуку даних.

Існує кілька видів індексів:

- Діапазонний індекс
- Індекс пошуку
- Текстовий індекс
- Точковий індекс
- Індекс повного тексту
- Векторний індекс

Cypher дозволяє створювати range індекси на одному чи кількох властивостях для всіх вузлів або відносин з заданою міткою або типом відносин:

- Індекс, створений на одному властивості для будь-якої мітки або типу відносин, називається single-property index.

- Індекс, створений на більш, ніж одній властивості для будь-якої мітки або типу відносин, називається composite index.

Крім того, текстові та точкові індекси є видом індексів одного властивості, з обмеженням, що вони визнають лише властивості зі стрічковими та точковими значеннями відповідно. Вузли чи відносини з індексованою міткою або типом відносин, де індексована властивість має інший тип значення, не включаються в індекс.

Також офіційна документація надає такі правила використання індексів:

- Найкращою практикою є надання індексу імені при його створенні. Якщо індекс не має явно наданого імені, він отримує автогенероване ім'я
- Ім'я індексу повинно бути унікальним серед індексів і обмежень
- Створення індексу за замовчуванням не є ідемпотентним, і виникне помилка, якщо ви спробуєте створити один і той самий індекс двічі. Використання ключового слова IF NOT EXISTS робить команду ідемпотентною, і жодної помилки не буде, якщо ви спробуєте створити один і той самий індекс двічі.

Короткий опис використання індексів neo4j

- **Текстові індекси**: Застосовуються для обробки запитів, які містять оператори CONTAINS, STARTS WITH, та ENDS WITH. Ці індекси оптимізовані для роботи з текстовими рядками, дозволяючи швидко знаходити відповідності в тексті.
- **Точкові індекси**: Ідеально підходять для задач, де потрібно визначити відстань або знаходити об'єкти в межах певної

обмежувальної рамки. Вони забезпечують високу швидкість обробки геопросторових даних.

- **Діапазонні індекси**: Використовуються у більшості інших випадків, де необхідно виконувати широкий спектр запитів, включаючи ті, що здійснюють пошук по діапазону значень.
- **Індекси пошуку**: Застосовуються за замовчуванням для розв'язання предикатів щодо міток вузлів і типів відносин.
- **Індекс повного тексту**: Дозволяє виконувати більш комплексні запити на відповідність тексту в індексованих рядкових властивостях. Відрізняється від діапазонних і текстових індексів можливістю токенізації індексованих значень.
- **Векторні індекси в Neo4j**: Використовують Apache Lucene та ієрархічний навігаційний граф Small World (HNSW) для виконання запитів на приблизно найближчих сусідів (k-ANN), що забезпечує пошук на основі схожості між властивостями вузлів.
  [14]

Розділ 2: Технічні деталі та вимоги

2.1 Головна мета застосунку

Головна мета розробленого застосунку полягає в поліпшенні доступності та ефективності виконання запитів до даних, що зберігаються в блокчейн-платформах, через використання передових методів індексації. Застосунок має на меті забезпечити аналітикам зручні інструменти для швидкого доступу до блокчейн-даних без необхідності прямого звернення до кожного блоку в ланцюжку. Це дозволить користувачам ефективно аналізувати та обробляти інформацію, зменшуючи час відгуку та збільшуючи продуктивність системи.

2.2 Опис користувачів системи

Система передбачає наявність тільки однієї ролі користувачів: аналітиків, які виконують комплексні запити до індексованих даних. Вони зосереджені на оптимізації запитів для аналізу великих обсягів даних.

2.3 Технічні вимоги до функціональності системи

Для користувачів:

- Виконання складних запитів до індексованих даних.
- Аналіз історичних даних транзакцій.
- Візуалізація даних для легшого розуміння зв'язків і тенденцій.

2.4 Специфікації вимог до даних

Дані про транзакції, що зберігаються в системі:

- Хеш транзакції.
- Дата та час створення транзакції.
- Кількість валюти, що брала участь в транзакції
- Інформація про ічасників транзакції (адреси відправника та отримувача)

Розділ 3. Розробка застосунку

3.1 Ідея

Під час вивчення матеріалу та аналізу проблеми традиційного підходу до аналізу та візуалізації блокчейн мереж була помічена необхідність реалізувати і дослідити збереження даних блокчейн транзакцій у вигляді представлення N гілок. Під кожну пару гаманців відправник-отримувач в блокчейн мережі буде існувати окрема гілка. Розуміючи, що графові бази даних можуть ефективно відобразити взаємозв'язки між об'єктами, було вирішено використати саме цю технологію.

За початкову ідею було вирішено створити програму, що прослуховуватиме івенти EVM провайдера мережі Infura. На кожний нову сукупність блоків програма вилучатиме транзакції з блоків і зберігатиме методом пар гаманців у графовій базі даних neo4j, а також надаватиме свій UI для пошуку взаємодій між гаманцями, що дозволить нам ефективно відслідковувати та аналізувати транзакції між конкретними користувачами. Це створює нові можливості для вивчення взаємодій в блокчейні та розкриває потенціал для подальших досліджень у цьому напрямку.

3.2 Обґрунтування вибору засобів розробки

3.2.1 Neo4j

Вибір графової бази даних Neo4j обумовлений її широкою популярністю, ефективністю у вирішенні завдань, пов'язаних з обробкою складних запитів і аналізом зв'язків між великою кількістю об'єктів. Neo4j дозволяє моделювати відносини між даними як зв'язки у графі,

забезпечуючи швидкий доступ та обробку інформації, що є критично важливим для аналізу блокчейн-транзакцій. Зокрема, було впроваджено ряд оптимізацій та різних необхідних індексів, які дозволяють ефективно працювати з базою даних Neo4j, забезпечуючи швидкі та точні запити до графової структури транзакцій блокчейну.

3.2.2 Docker

Використання Docker забезпечує легке розгортання та консистенцію середовища на всіх етапах розробки та впровадження. Контейнеризація з Docker дозволяє створювати ізольовані середовища для кожного компонента системи, спрощуючи тим самим управління залежностями і уніфікацію процесів розгортання.

3.2.3 Golang

Ця мова програмування була обрана через її продуктивність, простоту, вбудовану підтримку багатопотоковості, має потужну стандартну бібліотеку та широку спільноту, що робить її ідеальним вибором для розробки серверної частини високонавантажених систем. Також Golang використовується в найбільш вживаним дистрибютивом ethereum – geth, тож деякі частини мого коду використовують код бібліотеки по ефіріуму. Посилання на репозиторій: https://github.com/ethereum/go-ethereum.

3.2.4 Gin

Веб-фреймворк Gin використовується завдяки його високій продуктивності та ефективності. Gin дозволяє швидко розробляти RESTful

API, що є необхідністю для забезпечення взаємодії клієнтської частини з сервером. Легкість використання та мінімалістичний дизайн Gin сприяють швидкій розробці та зменшенню кількості потенційних помилок.

3.2.5 IDE

У процесі розробки використовувалась інтегрована середу розробки (IDE) у вигляді Goland, а також пізніше Visual Studio Code для вирішення проблем з відладкою коду, що значно спростило написання та підтримку застосунку. Також, для забезпечення якості мого програмного забезпечення, я використовував різні інструменти дебаггінгу та тестування.

Ці технології разом формують міцну основу для розробки сучасного програмного забезпечення, здатного ефективно обробляти специфічні вимоги блокчейн-застосунків та забезпечувати високу доступність та швидкість обслуговування запитів.

3.3 Модель СУБД

Спочатку була створена структура, де від 1 центрального «генезіс» вузла йшли ребра HAS_CHILD з полями from: <address> та to: <address> до іншого вузла. Подальші вузли для цієї гілки будуть зберігатись зі зв’язком HAS_CHILD від батьківської транзакції, що представляють собою ланцюг транзакцій від одного гаманця до іншого. Кожна нода нашого графу є, фактично, транзакцією мережі з відповідними полями. План описаний вище приблизно виглядає подібним чином:

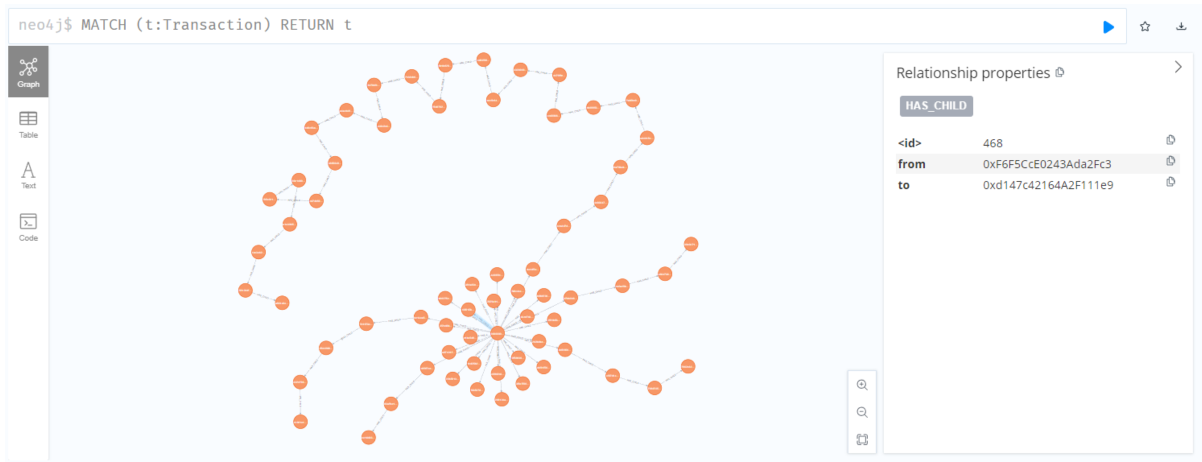


Рис. 3.1 Графічне відображення наповнення бази даних при початковій ідеї

Після реалізації і дослідження цього підходу було винайдено покращену ідею з відсутнім центральним вузлом, де визначення пар гаманців в гілці на себе бере перший вузол гілки, що має ярлик BaseTransaction та відповідні поля: “from” та “to”

В кінцевому представлення вузли (транзакції) виглядають таким чином:

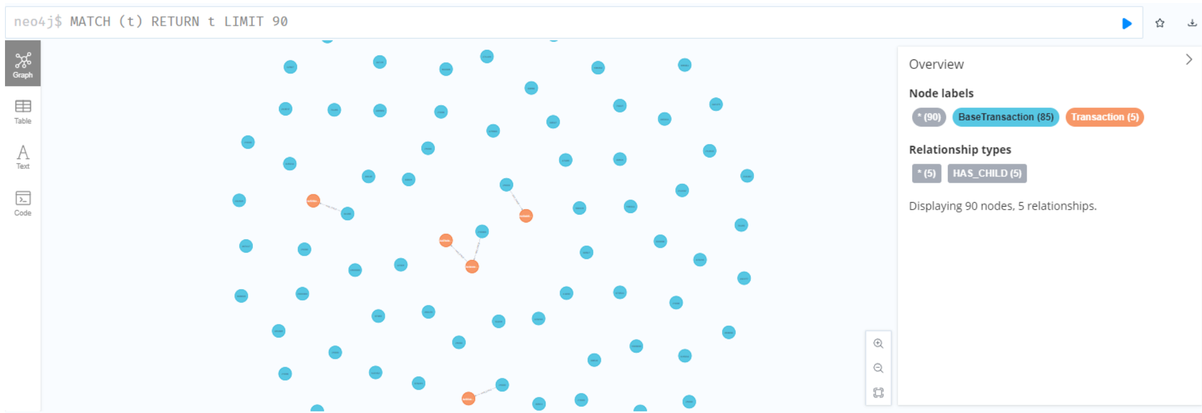


Рис. 3.2 Графічне відображення наповнення бази даних при фінальній ідеї

Для отримання візуального представлення графу, використано графічний інтерфейс, яку «із коробки» надає neo4j.

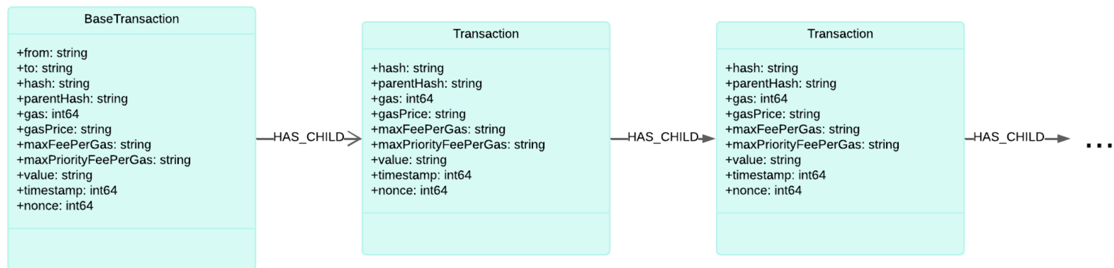


Рис. 3.3 Модель збереження даних в Neo4j

Загалом, підхід, що описаний вище, базується на впевненості, що використання графової бази даних є ключовим для досягнення оптимальної пошуку та аналізу транзакцій блокчейну, так як звичайні бази даних ключ-значення, не можуть надати достатньо зручного пошуку по зв'язкам між гаманцями через свою сфокусованість на ефективність

3.4 Код прослуховувача

Для початку, задача потребувала написати прослуховувач блокчейн мережі, щоб отримувати звідти нові блоки. У коді нижче застосовано бібліотеку ethclient для підключення(підписки) до мережі по вебсокету. Коли ж у мережі з'являється новий блок, виконується прохід по масиву транзакцій і виклик processTransaction функції для кожної транзакції в блоці.

```
func Listen() {
  url, _ := os.LookupEnv("BLOCKCHAIN_URL")

  client, err := ethclient.Dial(url)

  headers := make(chan *types.Header)
  sub, _ := client.SubscribeNewHead(context.Background(), headers)
  defer sub.Unsubscribe()
```

```
  for header := range headers {
    block, err := client.BlockByHash(context.Background(), header.Hash())

    for _, tx := range block.Transactions() {
      processTransaction(tx)
    }
  }
}
```

Код прослуховувача мережі

Далі обробник транзакції вишукує всі необхідні поля і створює новий об’єкт з аргументами транзакції. Після чого блок передається до генератора транзакції і зберігається в базі даних

```
func processTransaction(tx *types.Transaction) {
  if tx.To() == nil {
    return
  }

  from := common.GetFromAddress(tx)
  gas, nonce := tx.Gas(), tx.Nonce()

  transactionArgs := &transaction.Transaction{
    From:                 from,
    To:                   common.BytesToAddress([]byte(tx.To().Hex())),
    Gas:                  &gas,
    GasPrice:             tx.GasPrice(),
    MaxFeePerGas:         tx.GasFeeCap(),
    MaxPriorityFeePerGas: tx.GasTipCap(),
    Value:                tx.Value(),
    Nonce:                &nonce,
  }

  transaction := ts.GenerateTransaction(transactionArgs)
 transaction.Hash = common.StringToMyHash(tx.Hash().Hex())
 ts.SaveTransaction(transaction)
```

```
}
```

Код обробника транзакції (викликається для кожної транзакції блоку)

В методі GenerateTransaction ми просто додаємо батьківський хеш транзакції. При виклику функції збереження, створюємо транзакцію з тегом BaseTransaction, якщо транзакція перша, що має таку пару гаманців у відправниках і отримувачах, або тег Transaction, якщо вона не перша:

```
func (ts *TransactionService) SaveTransaction(transactionData *common.Transaction) {
  if transactionData.ParentHash != nil {
    ts.Repository.CTransaction(transactionData)
  } else {
    ts.Repository.CBaseTransaction(transactionData)
  }
}
```

Код сервісу, обрання створення типу вузла

Приклад створення нового вузла з відповідним зв'язком:

```
func (r *Repository) CTransaction(transactionData *common.Transaction) {
  dbTransaction, err := r.Session.BeginTransaction(r.Ctx,
                                                               func(*neo4j.TransactionConfig) {})
  defer dbTransaction.Close(r.Ctx)

  tx_queries.CTransactionN(dbTransaction, transactionData)
  tx_queries.CHasChildRelQuery(dbTransaction, transactionData.ParentHash, transactionData)

  if err := dbTransaction.Commit(r.Ctx); err != nil {
    panic(err)
  }
}
```

Код репозиторію по збереженню вузла

```go
func CTransactionN(dbTx neo4j.ExplicitTransaction, transactionData *common.Transaction) {
  params := map[string]interface{}{
    "hash": transactionData.Hash.ToString(),
    "parentHash": transactionData.ParentHash.ToString(),
    "gas": int64(*transactionData.Gas),
    "gasPrice": transactionData.GasPrice.String(),
    "maxFeePerGas": transactionData.MaxFeePerGas.String(),
    "maxPriorityFeePerGas": transactionData.MaxPriorityFeePerGas.String(),
    "value": transactionData.Value.String(),
    "timestamp": int64(*transactionData.Timestamp),
    "nonce": int64(*transactionData.Nonce),
  }
  template := "MERGE (t:Transaction { " +
    "hash: $hash, " +
    "parentHash: $parentHash, " +
    "gas: $gas, " +
    "gasPrice: $gasPrice, " +
    "maxFeePerGas: $maxFeePerGas, " +
    "maxPriorityFeePerGas: $maxPriorityFeePerGas, " +
    "value: $value, " +
    "timestamp: $timestamp, " +
    "nonce: $nonce}) " +
    "RETURN t"

  query := core.NewQueryBuilder(dbTx).
    WithParams(params).
    WithTemplate(template).
    WithLogPath("../logs/save_transaction.txt").
    Build()

  query.Run()
}
```

Код створення запиту до СУБД Neo4j

3.5 API

Наступною задачею було реалізувати спосіб доступу до бази даних. Один, що існує «із коробки» є адміністративна сторінка neo4j, що доступна після розгортання проекту за адресою localhost:7474. В цьому GUI можна створювати запити до бази даних мовою Cypher напряму.

Тим не менш, для зручнішого користування застосунком я вважав за потрібне реалізування невеликого сервісу, що надає API до найкорисніших даних застосунку, а саме:

- Отримання гілки транзакцій
- Отримання всіх адрес гаманців, з яких приходили або на які відправлялись транзакції з гаманця x
- Отримання транзакції за хешем
- Отримання гілки транзакцій з фільтрацією по часу

(Опис АПІ можна знайти в Додатку 1)

При розробці було використано відому бібліотеку для побудови back-end застосунків мовою golang - gin, що набагато полегшило і пришвидшило процес створення API. Головна функція для старту сервера приведена нижче. В ній вказуються основні шляхи до контролерів і порт.

Уся побудова АПІ відбувається за принципом MVC, де бекенд чітко поділений на контролери, сервіси та модель (репозиторій):

```
func Run() {
  router := gin.Default()

  router.Use(middlewares.CORSMiddleware())
  blockchainController := blockchain.NewController(blockchain.NewService())

  router.GET("/", blockchainController.GetByHash)
  router.GET("/branch", blockchainController.GBranch)
  router.GET("/interrelated", blockchainController.GInterrelatedAddresses)
  router.GET("/addresses", blockchainController.GAddresses)

  router.Run("0.0.0.0:8080")
}
```

Код налаштування шляхів REST API

Кожен контролер виглядає подібним чином. В ньому працює парсинг і валідація вхідних параметрів, потім викликається основний сервіс, що займається обробкою параметрів, пошуком та перетворенням даних, а потім повертає їх нагору

Нижче приведений код контроллера по отриманню всіх гаманців, що мають пов'язані транзакції з гаманцем x (x – надходить як параметр запиту).

```
func (cont *Controller) GInterrelatedAddresses(c *gin.Context) {
  var input payloads.GInterrelatedAddresses
  if parseOk, validOk := utilities.ParseInput(&input, c), utilities.ValidateInput(input, c); !(parseOk &&
validOk) {
          return
  }

  start := time.Now()
  addresses := cont.Service.GInterrelatedAddresses(input.Address)
  elapsed := time.Since(start)

  c.JSON(http.StatusOK, addresses)
}
```

Код контролеру

```
func (r *Repository) GInterrelatedAddresses(address *common.Address)
tx_queries.InterrelatedAddresses {
  result, err := r.Session.ExecuteRead(r.Ctx, func(tx neo4j.ManagedTransaction) (interface{}, error) {
          return tx_queries.GInterrelatedAddresses(tx, address), nil
  })
  record := result.([]*neo4j.Record)

  if len(record) == 0 {
          return tx_queries.InterrelatedAddresses{
            FromAddresses: []string{},
            ToAddresses: []string{},
          }
  }
```

```
  toAddressesRecord, _ := record[0].Get("toAddresses")
  fromAddressesRecord, _ := record[0].Get("fromAddresses")

  interrelatedAddresses := tx_queries.InterrelatedAddresses {
    FromAddresses: mapAddresses(fromAddressesRecord.([]interface{})),
    ToAddresses: mapAddresses(toAddressesRecord.([]interface{})),
  }

  return interrelatedAddresses
}
```

Код бізнес логіки, що викликає низькорівневу функцію для створення запиту

Також нижче продемонстрована функція GInterralatedAddresses функція з пакету запитів. В ній вказуються параметри, текст запиту і проводиться вилучення даних з СУБД.

```
func GInterrelatedAddresses(dbTx neo4j.ManagedTransaction, address *common.Address)
[]*neo4j.Record {
  params := map[string]interface{}{
          "address": address.ToString(),
  }

  template := "OPTIONAL MATCH (b1:BaseTransaction {from: $address}) " +
          "OPTIONAL MATCH (b2:BaseTransaction {to: $address}) " +
          "WITH COLLECT(distinct b1.to) as toAddresses, COLLECT(distinct b2.from) as fromAddresses " +
          "RETURN toAddresses, fromAddresses"

  query := core.NewQueryBuilder(dbTx).
          WithParams(params).
          WithTemplate(template).
          WithLogPath("../logs/get_interrelated_addresses.txt").
          Build()

  return query.Run()
}
```

Код низькорівневої логіки для виконання запиту до БД

3.6 Дослідження швидкодії

3.6.1 Метод оцінки алгоритму

Для розуміння ефективності та доцільності використання neo4j треба задати певні параметри оцінювання. Найбільш звичним і правильним методом оцінки алгоритмів пошуку є часова складність O(n), проте в графовій базі даних важко оперувати подібним поняттям, бо складні запити до neo4j можуть оптимізовуватись по-різному в залежності від версії до версії. Саме так neo4j описує роботу кожного запиту:

Запит Cypher починається як декларативний запит, представлений як рядок, який описує графовий шаблон для збігу в базі даних. Після розбору рядок запиту проходить через оптимізатор запитів (також відомий як планувальник), який генерує імперативний(логічний) план, для визначення найбільш ефективного способу виконання запиту, враховуючи поточний стан бази даних. У кінцевій фазі цей логічний план перетворюється в виконавчий фізичний план, який фактично виконує запит до бази даних.

Натомість neo4j оперує поняттям db hits та estimated rows
Пояснення db hits з офіційного джерела neo4j: Кожен оператор надсилатиме запит до сховищного двигуна для виконання роботи, такої як отримання або оновлення даних. Одна операція бази даних (db hits) є абстрактною одиницею цієї роботи сховищного двигуна.

Ось всі дії, які спричиняють один або кілька db hits:

- Дії створення
  - Створення вузла
  - Створення відношення
  - Створення нової мітки вузла
  - Створення нового типу відношення
  - Створення нового ідентифікатора для ключів властивостей з однаковою назвою

- Дії видалення
    - Видалення вузла
    - Видалення відношення
    - Дії оновлення
    - Встановлення однієї або декількох міток на вузлі
    - Видалення однієї або декількох міток з вузла

- Дії, специфічні для вузлів
    - Отримання вузла за його ID
    - Отримання ступеня вузла
    - Визначення, чи є вузол щільним
    - Визначення, чи встановлена мітка на вузлі
    - Отримання міток вузла
    - Отримання властивості вузла
    - Отримання існуючої мітки вузла
    - Отримання назви мітки за її ID або її ID за назвою

- Дії, специфічні для відношень
    - Отримання відношення за його ID
    - Отримання властивості відношення
    - Отримання існуючого типу відношення
    - Отримання назви типу відношення за його ID або його ID за назвою

- Загальні дії
    - Отримання назви ключа властивості за його ID або його ID за назвою ключа
    - Пошук вузла або відношення через пошук за індексом або скануванням індексу

  - Пошук шляху в змінній довжині розширення
  - Пошук найкоротшого шляху
  - Запит на значення до сховища лічильника

- Дії схеми
  - Додавання індексу
  - Скасування індексу
  - Отримання посилання на індекс
  - Створення обмеження
  - Скасування обмеження
  - Виклик процедури
  - Виклик користувацької функції.

Крім того, в дипломній роботі також розглянуто швидкість роботи пошуку даних в середньому часі виконання запиту на звичайному комп'ютері, характеристики якого вказані після таблиць і графіків

Для дослідження оптимізації та ефективності роботи бази даних було зроблено сотні запитів до АПІ з різними параметрами. Також створено порівняльні таблиці зі складністю виконання (db hits, rows estimated) та середньою швидкістю опрацьовування запиту на різних об'ємах даних. Характеристики персонального комп'ютеру, на якому проводились дослідження можна побачити в Додатку 2.

Для чистоти експерименту трекери часу було поставлено з моменту виклику певного запиту до neo4j і до моменту відповіді бази даних. Це зроблено з метою максимально оминути будь-який інший додатковий функціонал програми для чистоти розуміння швидкодії підходу до, описаного вище, збереження і вилучення даних з Neo4j. Інший код може зробити хибне уявлення щодо ефективності виконання запитів, бо навколишній код, очевидно, не є максимально оптимізованим і ефективним.

Також нижче наведено індекси, які було застосовано для пришвидшення роботи пошуку даних:

```
neo4j$ SHOW indexes yield name, entityType, labelsOrTypes, properties, indexProvider;
```

	name	entityType	labelsOrTypes	properties	indexProvider
1	"b_from_index"	"NODE"	["BaseTransaction"]	["from"]	"range-1.0"
2	"b_hash_index"	"NODE"	["BaseTransaction"]	["hash"]	"range-1.0"
3	"b_to_index"	"NODE"	["BaseTransaction"]	["to"]	"range-1.0"
4	"index_343aff4e"	"NODE"	null	null	"token-lookup-1.0"
5	"index_f7700477"	"RELATIONSHIP"	null	null	"token-lookup-1.0"
6	"t_hash_index"	"NODE"	["Transaction"]	["hash"]	"range-1.0"

Started streaming 6 records after 18 ms and completed after 19 ms.

Name	Entity type	Labels or types	Properties	Index provider
b_from_index	Node	BaseTransaction	from	range
b_hash_index	Node	BaseTransaction	hash	range
b_to_index	Node	BaseTransaction	to	range
t_hash_index	Node	Transaction	hash	range

3.6.2 Таблиці порівняння роботи програми на різних об'ємах даних

Абстрактна складність виконання запитів в *db hits*					
Кількість записів / Тип складного запиту	50	2000	15000 +	50000	200000
Вітка транзакцій	58	58	58	86	30
Транзакція за хешем	15	15	15	15	15
Всі пов'язані гаманці з гаманцем X	4	4	4	4	1
Список адрес в базі даних	164	6047	42272	145589	485345
Додавання нової транзакції	33	35	24	33	33

Як можна помітити на таблиці вище, кількість дій двигуна neo4j майже не змінюється від збільшення кількості записів в базі даних. Єдиним виключенням є отримання списку всіх адрес. Це можна пояснити тим, що БД робить прямий перебір по всім вузлам BaseTransaction. Це, звісно, можна вирішити методом індексації цих вузлів або методом винесення списку адрес в окрему структуру, що набагато пришвидшить його вилучення. Проте, в теорії, цей запит виконується досить рідко і тому питання оптимізації цього запиту залежить від методу використання застосунку.

Абстрактна складність виконання запитів в *rows estimated*

Кількість записів	50	2000	15000 +	50000	200000
Вітка транзакцій	5	10	24	32	33
Транзакція за хешем	5	10	10	10	10
Всі пов’язані гаманці з гаманцем X	7	110	1212	2470	2997
Список адрес в базі даних	102	3972	28166	97062	323166
Додавання нової транзакції	3	3	3	3	3

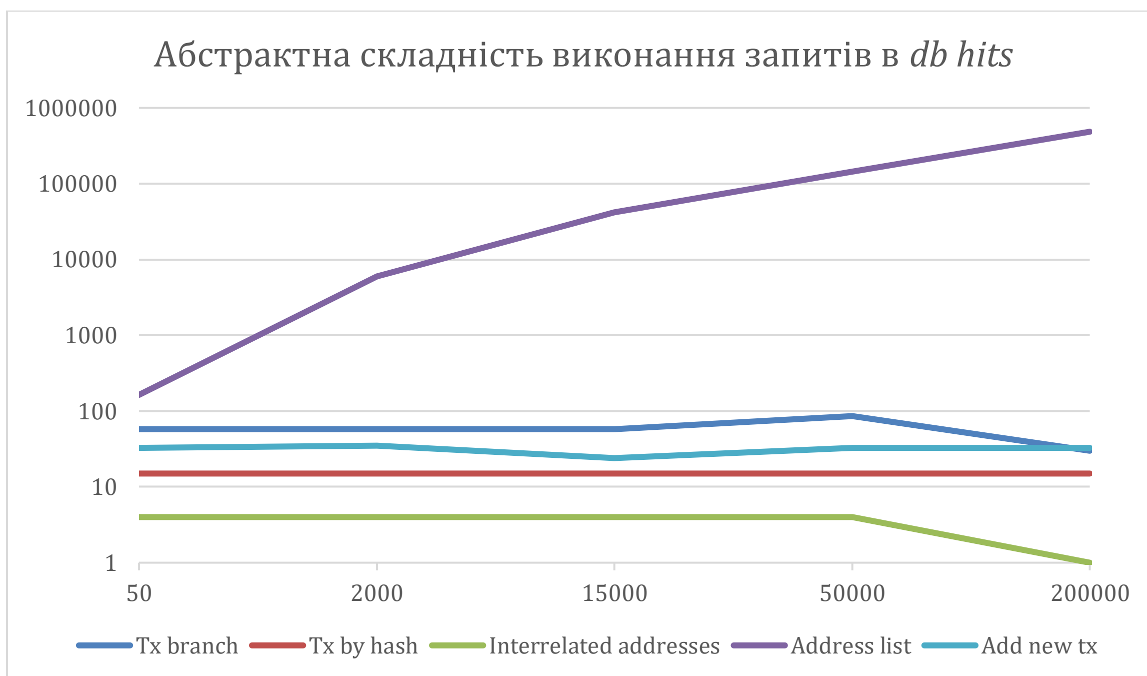
На таблиці вище можна побачити схожу картину, усі запити відпрацьовують приблизно з одноковою ефективністю, окрім отримання взаємопов’язаних гаманців та, згаданий раніше, список адрес.

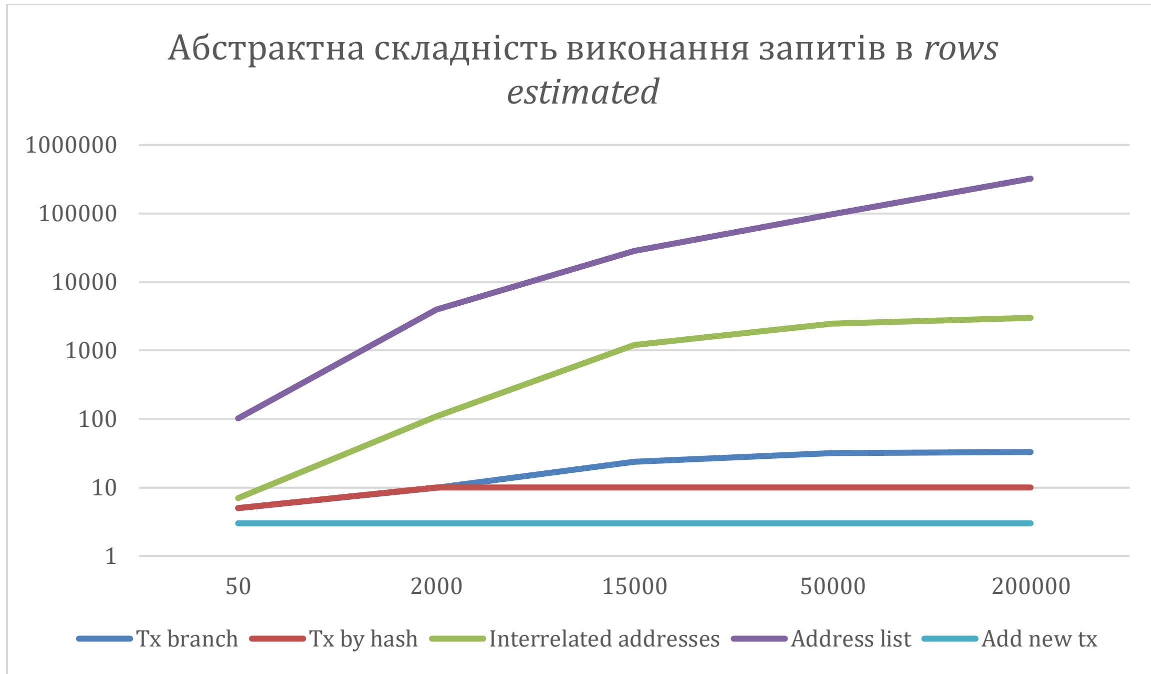
Тривалість виконання запитів					
Кількість записів	50+	2000+	15000 +	50000	200000+
Вітка транзакцій	1.19 ms	1.35 ms	0.98 ms	1.04 ms	1.58 ms
Транзакція за хешем	1.75 ms	1.12 ms	0.94 ms	0.91 ms	1.04 ms
Всі пов’язані гаманці з гаманцем X	1.03 ms	1.91 ms	1.51 ms	1.53 ms	4.47 ms

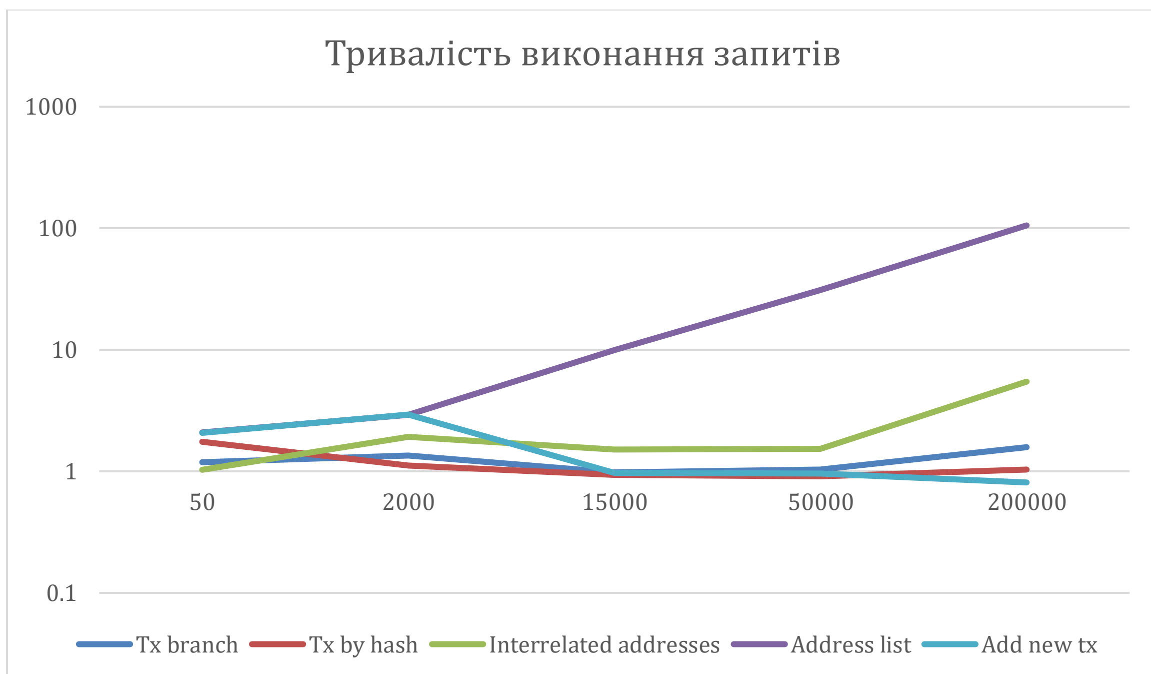
Список адрес в базі даних	2.09 ms	2.92 ms	9.90 ms	31 ms	105.56 ms
Додавання нової транзакції	2.08 ms	2.93 ms	0.97 ms	0.96 ms	0.81 ms

Дані з кількістю витраченого часу на запит корелюють з таблицями, що були розглянуті вище.

3.6.3 Графіки логарифмічного масштабу порівняння роботи програми на різних об'ємах даних




Абстрактна складність виконання запитів в *rows estimated*
1000000
100000
10000
1000
100
10
1
50
2000
15000
50000
200000
Tx branch
Tx by hash
Interrelated addresses
Address list
Add new tx


Тривалість виконання запитів
1000
100
10
1
0.1
50
2000
15000
50000
200000
Tx branch
Tx by hash
Interrelated addresses
Address list
Add new tx

3.6.4 Аналіз

У контексті дослідження ефективності збереження транзакцій у базі даних Neo4j виникає кілька ключових висновків. Перш за все, аналіз даних підтверджує стійкість підходу до збереження транзакцій незалежно від обсягу даних у базі. Це вказує на високу масштабованість системи, що є важливою перевагою для розробників інформаційних систем.

Проте, виявлено, що деякі операції, зокрема отримання списку всіх адрес або взаємопов'язаних гаманців, можуть потребувати оптимізації для забезпечення оптимальної продуктивності при збільшенні обсягу даних. Це вимагає подальших досліджень щодо оптимізаційних підходів, таких як індексація відповідних вузлів або розробка спеціалізованих структур даних. Наприклад - винесення усіх адрес гаманців в окрему таблицю.

Необхідно також відзначити, що загальна тривалість виконання запитів залишається невеликою, що свідчить про загальну ефективність підходу. Проте, варто враховувати, що час виконання запитів для отримання списку адрес у базі даних збільшується значно швидше за інші запити при зростанні обсягу даних.

Отже, узагальнюючи, підхід до збереження транзакцій у базі даних Neo4j, вказаний вище, проявляє себе як ефективний, але вимагає уваги до оптимізації окремих запитів для забезпечення оптимальної продуктивності при збільшенні обсягу даних. Це створює певні перспективи для подальших досліджень у сфері оптимізації роботи з базою даних Neo4j.

Крім того, під час проведення експериментів виявлено, що перша спроба кожного типу запитів відпрацьовує значно довше, ніж будь-які подальші запити з тими ж параметрами. Цей феномен може свідчити про наявність у системі Neo4j механізму кешування або оптимізації роботи для подібних запитів.

Це явище є цікавим об'єктом для подальших досліджень, оскільки це може суттєво підвищити продуктивність системи. Для ретельнішого розуміння принципів роботи системи Neo4j рекомендується провести додаткові експерименти та аналіз механізмів оптимізації, що відбуваються внутрішньо.

3.7 Розгортка проекту

Кроки для запуску застосунку:

1) Клонувати репозиторій за посиланням: https://github.com/SASanchezz/wallet-branch-searcher
2) В корневій директорії в .env файлі задати BLOCKCHAIN_URL, а саме це має бути url infura мережі, яку треба почати прослуховувати з вказаним АПІ ключем.
3) Далі в корневій директорії проекту прописати команду “docker-compose up -d --build”

Після чого програма почне записувати всі транзакції нових блоків в базу даних, надасть АПІ для користування і сторінку за посиланням: http://localhost:3000/

А також за http://localhost:7474/ буде доступний графічний інтерфейс що надає neo4j.

Висновки

Отже, ця дипломна робота провела глибокий аналіз інструментів та алгоритмів індексації, використовуваних у блокчейн-технологіях, з акцентом на їх ефективності та оптимальності. Розглядаючи різні підходи до збереження індексованих даних, було виявлено ключові аспекти, що сприяють поліпшенню обробки запитів і ефективної системи зберігання даних у блокчейні.

Основні результати дослідження підкреслюють важливість оптимізації індексаційних алгоритмів для підвищення продуктивності транзакцій у блокчейн-мережах. Зокрема, було визначено, що індексація, оптимізована під конкретні типи запитів, може значно знизити час відгуку та збільшити загальну продуктивність системи.

Крім того, аналіз різних баз даних, як-от LevelDB та її модифікації, показав, що правильний вибір технологічної платформи має критичне значення для вирішення специфічних задач платформ.

У вигляді практичної частини вдалося створити інноваційний підхід до індексації блокчейну, який може бути використаний для різних додатків та досліджень у сфері блокчейн технологій. Цей проект відкриває нові можливості для аналізу та взаємодії з транзакціями в блокчейні, розширюючи горизонти дослідження у цьому захоплюючому напрямку.

У висновку, це дослідження сприяло кращому розумінню комплексності та важливості індексації в контексті блокчейн технологій. Результати дипломної роботи можуть слугувати основою для подальших інновацій у розробці та оптимізації інструментів для блокчейн-платформ, що відкриває нові перспективи для покращення технологічної ефективності в цій швидко розвиваючій галузі.

Джерела:

1. Role of GraphDB in FinTech, Blockchain Ledgers [електронний ресурс] - Режим доступу до ресурсу: https://www.researchgate.net/publication/365650697_Role_of_GraphDB_in_FinTech_Blockchain_Ledgers
2. Indexing the universe of blockchains with Covalent [електронний ресурс] - Режим доступу до ресурсу: https://medium.com/1kxnetwork/indexing-the-universe-of-blockchains-with-covalent-7a9686783dc1
3. A Brief Review of Database Solutions Used within Blockchain Platforms [електронний ресурс] - Режим доступу до ресурсу:https://www.researchgate.net/publication/342941591_A_Brief_Review_of_Database_Solutions_Used_within_Blockchain_Platforms
4. Rarible Protocol [електронний ресурс] - Режим доступу до ресурсу: https://github.com/rarible
5. An Overview of Blockchain Indexing Protocol [електронний ресурс] - Режим доступу до ресурсу: https://www.leewayhertz.com/blockchain-indexing-protocol/
6. We tracked 800 million transactions in the Ethereum Blockchain. Here is how we did it. [електронний ресурс] - Режим доступу до ресурсу: https://www.tarlogic.com/blog/download-Ethereum-blockchain/
7. A Graph Model Based Blockchain Implementation for Increasing Performance and Security in Decentralized Ledger Systems [електронний ресурс] - Режим доступу до ресурсу: https://www.researchgate.net/publication/342616825_A_Graph_Model_Based_Blockchain_Implementation_for_Increasing_Performance_and_Security_in_Decentralized_Ledger_Systems

8. How To Query The Ethereum Blockchain [електронний ресурс] - Режим доступу до ресурсу: https://rockset.com/blog/how-to-query-the-Ethereum-blockchain/
9. Search-performance indexes [електронний ресурс] - Режим доступу до ресурсу:https://neo4j.com/docs/cypher-manual/current/indexes-for-search-performance/
10. How Bitcoin Uses Google's LevelDB Storage Engine [електронний ресурс] - Режим доступу до ресурсу: https://www.linkedin.com/pulse/how-bitcoin-uses-googles-leveldb-kvs-manage-its-bamiyan-gobets https://bitcoincoredocs.com/files.html
11. go-ethereum [електронний ресурс] - Режим доступу до ресурсу: https://github.com/ethereum/go-ethereum
12. Bitcoin Core file system [електронний ресурс] - Режим доступу до ресурсу: https://bitcoincoredocs.com/files.html
13. Merkle tree [електронний ресурс] - Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Merkle_tree
14. Neo4j docs [електронний ресурс] - Режим доступу до ресурсу: https://neo4j.com/docs/cypher-manual/5/indexes/

Додатки

Додаток 1. Опис API

1) Отрмиання даних про транзакцію за хешем
http://localhost:8080/?hash=<transaction_hash>
Параметри запиту:

Назва	Тип	Опис
hash	string	Хеш транзакції

Приклад відповіді


```
{
  "hash": "08712dd9b4ae717aae5f8041c5af8db39b49513f69da36ce014790f53b60bdbc",
  "gas": 7197809264300506357,
  "gasPrice": 6657594124675139955,
  "maxFeePerGas": 2921011592928235497,
  "maxPriorityFeePerGas": 1080766594002680276,
  "value": 6088969736370536943,
  "timestamp": 6036522182726,
  "nonce": 6368776032124926283
}
```


2) Отримання (макс 100) основних данних про транзакції (hash, value, timestamp) між двома гаманцями в межах певного часу. Якщо часові рамки не задані, то повертаються останні 100 транзакцій цієї гілки
http://localhost:8080/branch?from=<from_address>&to=<to_address>&before=<before_timestamp>&after=<after_timestamp>&limit=<limit>

Параметри запиту:

Назва	Тип	Опис
from	string	Адреса гаманця, з якого надходили транзакції
to	string	Адреса гаманця, до якого надходили транзакції
before	number	Дата в unix форматі, до якої мають бути знайдені транзакції

after	number	Дата в unix форматі, після якої мають бути знайдені
limit	number	Кількість транзакцій, що треба повернути. Макс значення - 100

Приклад відповіді


```
[
  {
    "hash": "0000000000000000000000000000000000000000000000000000000000000000",
    "gas": 1,
    "gasPrice": 0,
    "maxFeePerGas": 0,
    "maxPriorityFeePerGas": 0,
    "value": 0,
     "timestamp": 0,
    "nonce": 7
  },
  {
    "hash": "8cb5f9163141527e5d49dea09928618c5d6c451b23f2fe7fca2602b20b551364",
    "gas": 17885022668937202638,
    "gasPrice": 6385443110578909873,
    "maxFeePerGas": 2660472888221132622,
    "maxPriorityFeePerGas": 7910474684118863958,
    "value": 5491277803372554260,
    "timestamp": 6018165282726,
    "nonce": 5600529479493418461
  },
  {
    "hash": "1910cb1f6d7d062672def3c5ffb7c2b3083d5289c39e84695d418ec98de0ebc1",
    "gas": 12192354376332124899,
    "gasPrice": 7740972398397844885,
    "maxFeePerGas": 4389763620809849906,
    "maxPriorityFeePerGas": 4546885345690798618,
    "value": 2873975295286711103,
    "timestamp": 6235765282726,
    "nonce": 13275428991553310906
  }
]
```


3) Отримання усіх адрес гаманців, з яких даний гаманець отримував або на які гаманці надсилав транзакції

http://localhost:8080/interrelated?address

Параметри запиту:

Назва	Тип	Опис

address	string	Адреса гаманця

Приклад відповіді

{
"from": ["0x26cE7c1976C5eec83e”],
"to": ["0x00000000003b3cc22a"]
}

4) Отримання усіх адрес в базі даних

http://localhost:8080/addresses

Приклад відповіді

[
"0x65A8F07Bd9A8598E1b",
"0x26cE7c1976C5eec83e",
"0x1bC5618bC0be183690",
"0x6C8e73F928D0c9cdE0"
]

Додаток 2. Характеристики ПК

Операційна система	Windows 10, 64-bit
Процесор	AMD® Ryzen 7 5800x 8-core processor × 16
Відеокарта	GeForce RTX 3060 Ti
Пам’ять	32 Гб