Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

АТЕСТАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Метод оптимізації багатопроцесорної обробки запитів

(тема)

Виконав:
студент II курсу, групи СПзм-18-2
Політаєва А.І.
(прізвище, ініціали)

Спеціальність
123 – Комп'ютерна інженерія
(код і повна назва спеціальності)

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

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: проф. Міхаль О.П.
(посада, прізвище, ініціали)

Допускається до захисту

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

2020 р.

Харківський національний університет радіоелектроніки

Факультет навчально-науковий центр заочної форми навчання

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 – Комп'ютерна інженерія
(код і повна назва)

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

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20___ р.

ЗАВДАННЯ

НА АТЕСТАЦІЙНУ РОБОТУ

студентові Політаєвій Альбіні Ігорівні
(прізвище, ім'я, по батькові)

1. Тема роботи Метод оптимізації багатопроцесорної обробки запитів

затверджена наказом по університету від " 30 " березня 2020 р. № 43 Стз

2. Термін подання студентом роботи до екзаменаційної комісії 18 травня 2020 р.

3. Вхідні дані до роботи

мультізапит
оптимізація
обробка
система з паралельною обробкою

4. Перелік питань, що потрібно опрацювати в роботі

Паралельні обчислення
Оцінка продуктивності багатопроцесорних обчислювальних систем
Традиційна організація мультізапитів
Метод оптимізації обробки запитів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) 14 слайдів

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

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

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд предметної області. Аналіз	31.03.20-10.04.20	
	літературних джерел		
2	Вибір та обгрунтування методики	11.04.20-20.04.20	
	дослідження		
3	Вибір інструментальних засобів	21.04.20-24.04.20	
4	Проведення експериментів	25.04.20-07.05.20	
	Оформлення ПЗ	08.05.20-11.05.20	
6	Подання атестаційної роботи	12.05.20-13.05.20	

Дата видачі завдання 30 березня 2020 р.

Студент _____
(підпис)

Керівник роботи _____ проф. Міхаль О.П.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка атестаційної роботи: 68 с., 25 рис., 1 дод., 22 джерела.

ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, КОМП’ЮТЕРНА МЕРЕЖА, ЗАПИТ, ЕКСПЕРИМЕНТ, МУЛЬТІЗАПИТ, БАГАТОПРОЦЕСОРНА СИСТЕМА.

Метою атестаційної роботи є дослідження методів обробки запитів з використанням мультипроцесорів.

Розглянуто ефективність виконання багатозадачного запиту в базі даних багатопроцесорної обчислювальної системи. Розроблено метод забезпечення оптимізацій багатопроцесорної обробки багатозадачних запитів. Модифіковано існуючий алгоритм розподілу елементарних запитів на процесори. Експериментально доведено, що мінімальний час виконання запиту може бути досягнуто при виконанні елементарних запитів у відповідному порядку, визначеному умовою впорядкованості. Розглянуто математичні методи оцінки методів оптимальної обробки багатозадачних запитів. Відомим методом збільшення продуктивності баз даних обчислювальних засобів є одночасне виконання декількох запитів, що утворюють багатозадачний запит.

ABSTRACT

Master’s thesis: 68 pages, 25 figures, 1 appendics, 22 sources.

PARALLEL COMPUTING, COMPUTER NETWORK, REQUEST, EXPERIMENT, MULTI-REQUEST, MULTIPROCESSOR SYSTEM.

The purpose of the certification work is to study the methods of processing requests using multiprocessors.

The efficiency of multitasking query execution in the database of a multiprocessor computer system is considered. A method for providing optimizations for multiprocessor processing of multitasking requests has been developed. The existing algorithm for allocating elementary requests to processors has been modified. It has been experimentally proven that the minimum query execution time can be achieved by executing elementary queries in the appropriate order determined by the ordering condition. Mathematical methods for estimating methods for optimal processing of multitasking queries are considered. A known method of increasing the performance of computing databases is the simultaneous execution of several queries that form a multitasking query.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

GRID – географічно розподілена інфраструктура

MQO – багаторазова оптимізація запитів

ОЗП – оперативний запам’ятовуючий пристрій

ОС – обчислювальна система

ВСТУП

В даний час спостерігається зростання інтересу до використання багатопроцесорних обчислювальних систем для вирішення великомасштабних трудомістких обчислювальних завдань. У зазначених системах обчислювальне середовище формується безліччю обчислювальних вузлів, з’єднаних деякої мережею для обміну даними, здатними одночасно (паралельно) вирішувати різні завдання. Однак, існує обмеження на масштабованість обчислювальної системи. Починаючи з певного моменту додавання нових процесорів в систему може не поліпшити, а навіть погіршити її ефективність. Тому постає питання в розробці методів оптимізації багатопроцесорної обробки мультизапитів.

Коефіцієнт прискорення продуктивності обчислювальної системи, обумовлений збільшенням кількості процесорів, описується законом Амдала, який був сформульований в 1967 Оцінку прискорення по Амдалу часто використовували при аналізі багатьох паралельних систем распараллелівать алгоритмів. Однак, в даний час вчені та інженери все більше говорять про необхідність модифікації закону Амдала, оскільки його формулювання містить серйозні допущення. Наприклад, при виведенні формули прискорення, яке може бути отримано на обчислювальної мультипроцессорной системі, в порівнянні з однопроцессорной, не враховувалися тимчасові витрати на здійснення обміну інформацією. Більш того, провідний інженер NVIDIA Corp., Майкл Вольф, виступив із закликом відправити закон Амдала в почесну відставку, мотивуючи це різноманітністю сучасних мультипроцесорних архітектур і суттєвими відмінностями трудомісткості завдань в залежності від обсягу оброблюваних даних.

Метою роботи є дослідження методів обробки запитів з використанням мультипроцесорів.

Завданнями є:

- аналіз існуючих методів багатопроцесорної обробки запитів;

- розгляд питань підвищення ефективності багатопроцесорної обробки запитів;

- розробка програмних засобів для аналізу продуктивності.

1 ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ

Паралельні обчислення дуже важливі в сучасному світі, так як для виконання завдань, які зараз вирішуються, потужності обчислювальних ЕОМ не вистачає, так само відомо, що одинична ЕОМ має свої межі. Такі обчислювальні машини дуже складно створити, так як кількість даних, які необхідно обробити стає все більше і більше, тому для вирішення проблем в цій галузі потрібні немалі зусилля вчених і конструкторів, так само ця область вимагає великих фінансових вкладень. Величезний крок у цьому напрямку був зроблений США з її програмою «Accelerated Strategic Computing Initiative» (ASCI), дослівно «Прискорена стратегічна комп'ютерна ініціатива». Ця програма має на увазі створення суперкомп'ютерів. Суперкомпютер - спеціалізована обчислювальна машина, значно перевершує за своїми технічними параметрами і швидкості обчислень більшість існуючих в світі комп'ютерів.

1.1 Системи з паралельною обробкою

Виділяють чотири типи архітектури систем паралельної обробки:
а) конвеєрна і векторна обробка.

У конвеєрних обчислювальних системах основний обсяг операцій з обробки даних виконується одним або декількома конвеєрними процесорами Конвеєр оперує векторами даних, які є одновимірними масивами. Конвеєрна обробка ефективна тільки тоді, коли завантаження конвеєра майже повна.

Конфлікти в конвеєрі команд.

Конфлікти в конвеєрі команд називають ризиками. Структурний ризик – спроба декількох команд одночасно звернутися до одного і того ж ресурсу. Рішення ризику: вводять кеш пам'ять при цьому виділяють кеш команд і кеш даних. Вплив на продуктивність – не велика.

Ризик за даними – викликаний зв’язком команд за даними. Зустрічається дуже часто, і виділяють 3 типи:

- читання після запису – коли наступна команда читає операнд до його записи попередньої команди;

- запис після читання – коли наступна за даною команда встигає записати нове значення до того, як його прочитає попередня;

- запис після запису – попередня команда записала своє значення пізніше, ніж наступна.

Найбільш часто зустрічається читання після запису. Для боротьби використовуються як програмні, так і апаратні рішення. Програмні: працюють на стадії компіляції програми, при цьому оптимізуючий компілятор вставляє між командами (де може статися конфлікт) нейтральні до цього конфлікту команди. Структурні (є основними): найбільш очевидним рішенням є зупинка конвеєра до виконання поточної команди. У більш складних рішеннях обчислюється ступінь конвеєра, на якій можливий конфлікт і до даної ситуації конвеєр зупиняється.

Ризик по управлінню (найбільше проблем) – викликаний неоднозначністю вибору наступної команди в разі появи команди переходу, або передачі управління. Вирішується наступними методами: використовувати буфер при вибірці, множинні потоки, затримка переходу, прогноз переходів.

б) SIMD комп’ютери. SIMD комп’ютер має N процесорів, N потоків даних, один потік команд. Процесори отримують команди і на кожному кроці всі процесори виконують одну і ту ж команду над даними з власної локальної пам’яті.

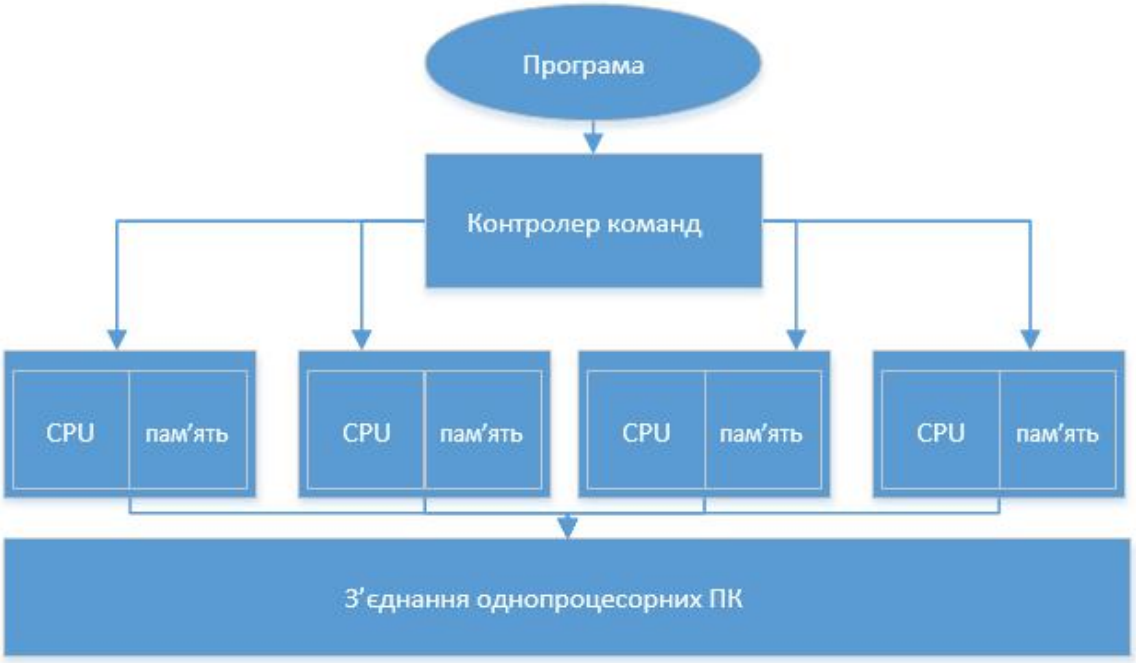

Рисунок 1.1 – Мережа, що з'єднує однопроцесорні ПК

3) MIMD комп'ютери. MIMD комп'ютер має N процесорів, незалежно використовують N потоків команд і обробних N потоків даних.

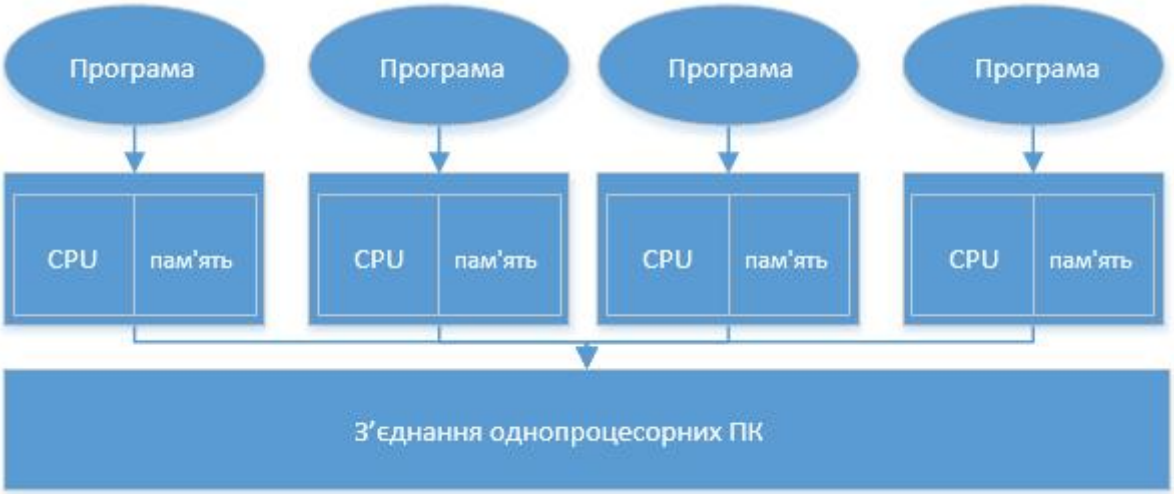

Рисунок 1.2 – MIMD мережа

4) MSIMD комп'ютери. У межах кожного завдання компілятор автоматично векторизуется відповідні цикли. Машини типу MSIMD дають

можливість використовувати два принципу декомпозиції: векторні операції для тих частин програми, які підходять для цього, і гнучкі можливості MIMD-архітектури для інших частин програми.

1.2 Системи з декількома процесорами

1.2.1 Багатопроцесорні системи

Багатопроцесорна система – система, де команди на шині можуть бути сприйняті більше ніж від одного мікропроцесора. Система містить N процесорів працюють із загальною оперативною пам’яттю і управляється однією операційною системою. Поява багатопроцесорних систем обумовлено великою кількістю причин. Головна з них – прагнення до ще більшого підвищення продуктивності обчислювальних засобів. Для задач реального часу потужності одного процесора часто вже не хвататет, і тому використовується більш одного процесора працює паралельно іншим. Таке розпаралелювання обчислень на кількох процесорах сильно зменшує час вирішення завдань. Ще однією причиною появи таких систем є протиріччя між низькою вартістю мікропроцесора і високою вартістю різних: накопичувачів інформації, пристроїв введення (виведення), засобів відображення інформації та т. і. Дану проблему можна вирішити, якщо багато мікропроцесорів будуть спільно використовувати дороге обладнання. При створенні багатопроцесорних систем виникає багато проблем, наприклад: розпаралелювання обчислювального процесу для ефективного завантаження процесорів системи, подолання конфліктів при спробах декількох процесорів використовувати один і той же ресурс системи. Багатопроцесорні системи бувають однорідні і неоднорідні. В однорідних системах система повинна мати ті самі процесори. У неоднорідних же системах складається з різних спеціалізованих процессоров. Мікропроцессор може мати багаторівневу або однорівневу структуру.

У однорівневої многопроцессорной системі процесори утворюють один загальний рівень обробки даних. У другому ж випадку система містить один головний і один або кілька допоміжних процесорів. Дані процесори використовуються для виконання різних рівнів обробки інформації. Дуже часто менш потужний допоміжний процесор виконує введення інформації з різних терміналів і її попередню обробку.

Рисунок 1.3 – Однорівнева багатопроцесорна система

Рисунок 1.4 – Багаторівнева система

Організація зв’язку процесорів з оперативною пам’яттю. За способом організації зв’язку процесорів з оперативною пам’яттю системи відносяться до одного з трьох типів: із загальною (однаково доступний), індивідуальної (роздільної), індивідуальної основний і загальної оперативної допоміжної пам’яттю даних.

1.2.2 Системи із загальною пам’яттю

Багатопроцесорні системи із загальною оперативною пам’яттю ефективні при вирішенні завдань, де дані можуть бути розміщені в різних модулях загальної оперативної пам’яті. Тоді процесори системи будуть працювати з цими модулями з малим шансом перетину. Досягають це за допомогою матричного комутатора. Витрати на такий матричний комутатор ростуть пропорційно добутку числа модулів пам’яті на число процесорів і пристроїв введення-виведення. Кожен з процесорів має доступ до будь-якого модулю оперативної пам’яті. Все ОЗУ працюють паралельно і допускають n одночасних звернень з боку процесорів або пристроїв введення-виведення. Якщо декількох процесорів або пристроїв введення-виведення звернуться до одного і того ж модулю пам’яті виникає конфліктна ситуація і комутатор обслуговує першим пристрій з найвищим пріоритетом.

Багатопроцесорні системи з індивідуальної пам’яттю даних ефективні для вирішення класу задач, де є чітко розділяються підмножини даних, що розміщуються у відповідних модулях пам’яті. Кожен процесор звертається до свого модулю оперативної пам’яті. Для рідкісного обміну даними між процесорами передбачені блоки обміну (БО) і оперативна допоміжна спільна пам’ять ОЗП. Операції обміну ініціюються процесорами.

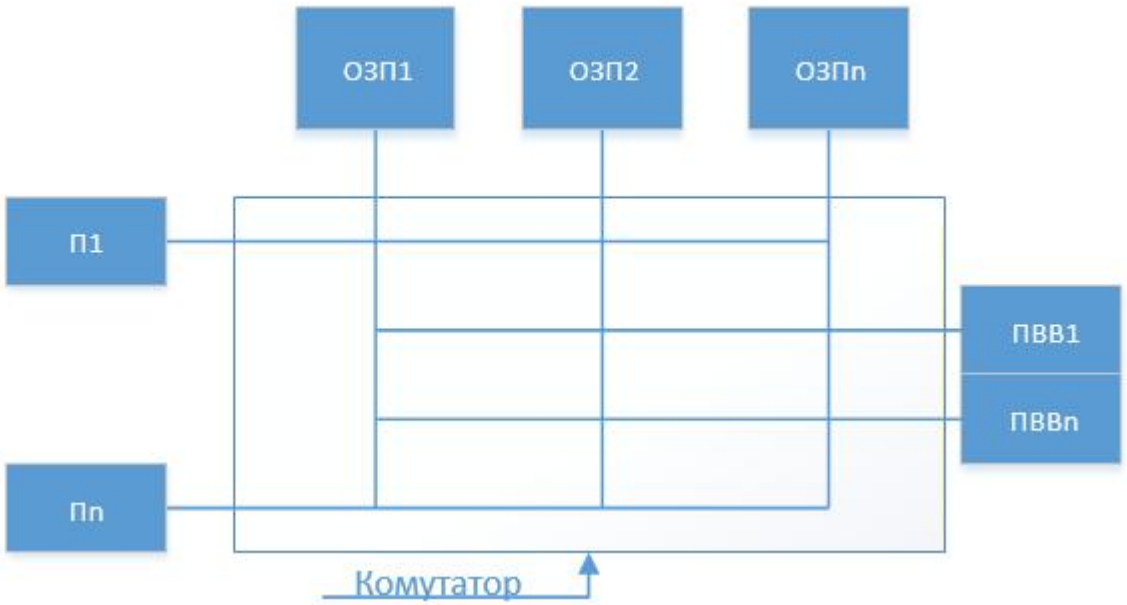

Рисунок 1.5 – Багатопроцесорна система матричного типу

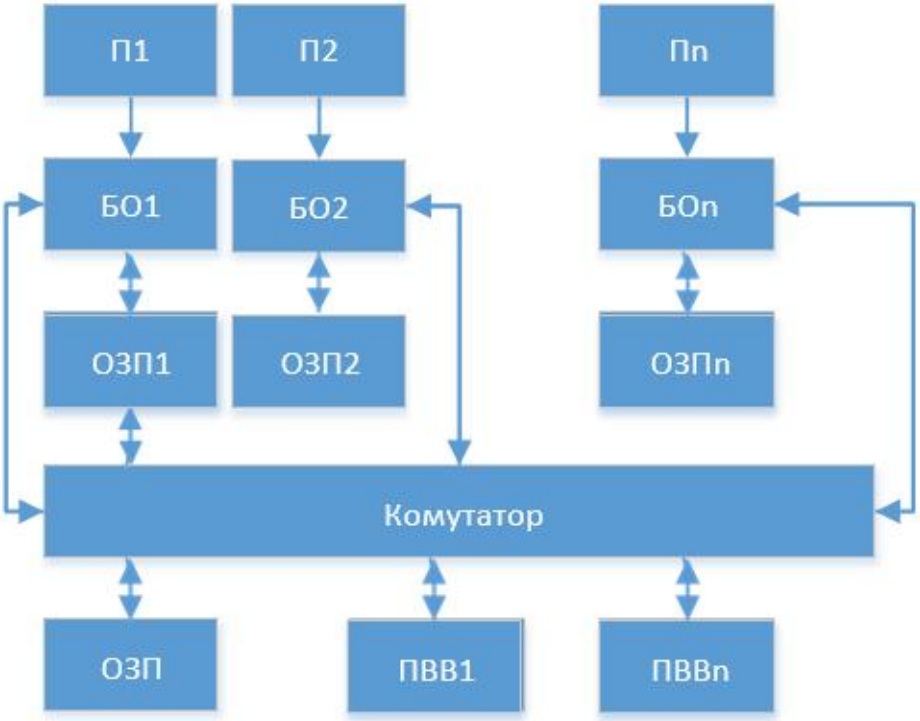

Рисунок 1.6 – Багатопроцесорна система з індивідуальною пам’яттю

Багатопроцесорна систем з індивідуальною пам’яттю і одномодульною спільною пам’яттю даних, особливо ефективна для багатопрограмної роботи з колективним використанням великого архіву базових даних. Пристрій управління пам’яттю регулює можливі конфлікти при одночасному зверненні декількох процесорів до загальної одномодульної пам’яті даних.

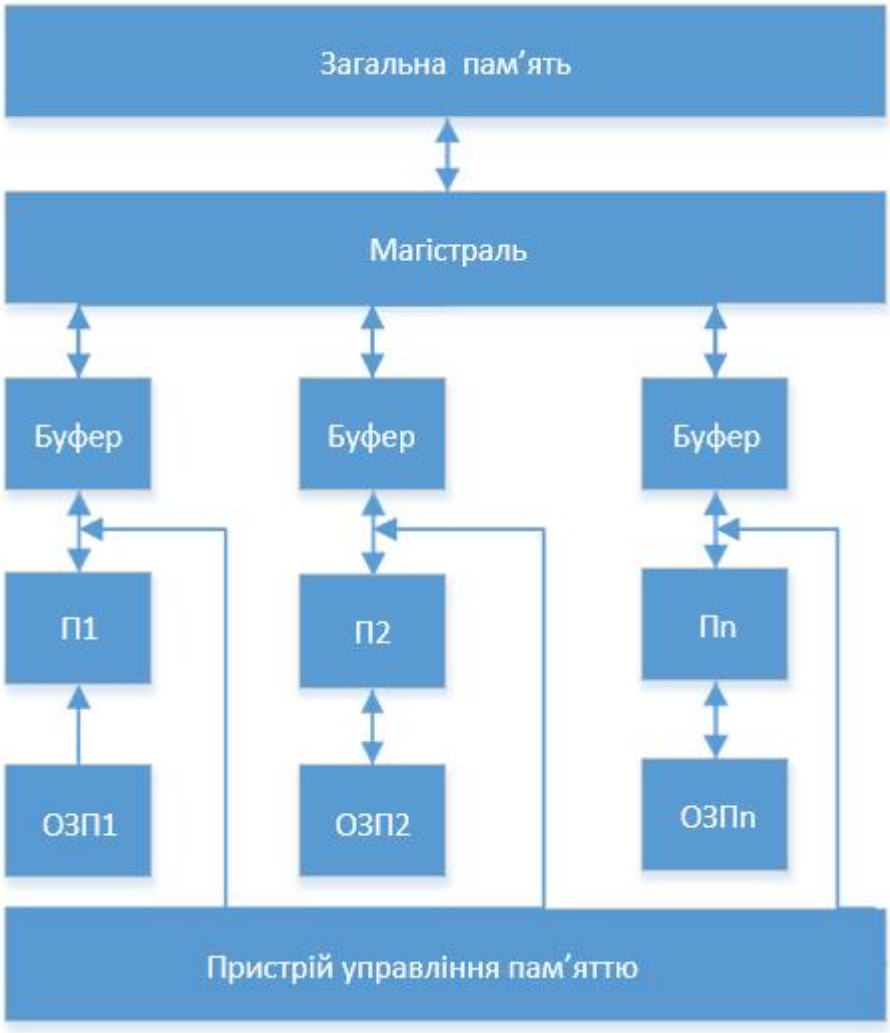

Рисунок 1.7 – Багатопроцесорна система з загальною пам’яттю даних

Структури багатопроцесорних систем. Нульмерні (Загальна шина). Використовувалася при формуванні найпростіших обчислювальних систем. Основний недолік – обмеження щодо нарощування кількості процесорів. Одновимірна. Немає зосередженого ресурсу, і кожен процесор має двох сусідів. Володіє більшою надійністю і допускає одночасну реалізацію між

процесорами. При побудові промислових систем використовувалася структура кольца. Двувимірна. Кожен процесор має чотирьох сусідів. Такі структури використовувалися при створенні промислових систем як об’єднуючі процесори.

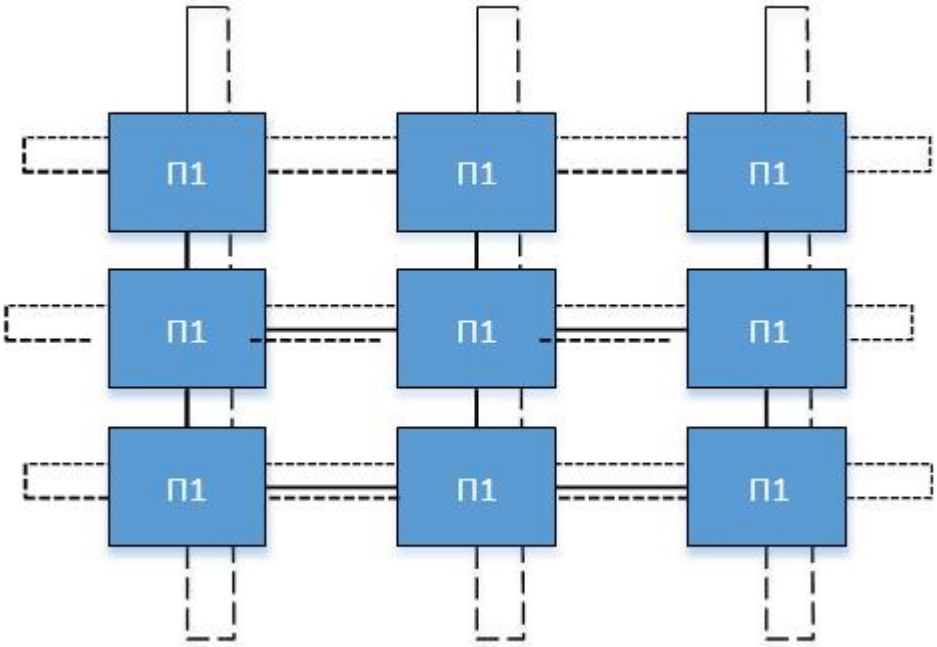

Рисунок 1.8 – Двовимірна структура

1.3 Мультікомп’ютерні системи

Являють собою вузли, кожен вузол має свої модулі: управління дисками, пам’яттю, процесором, відправки повідомлення на інші вузли, прийом від інших вузлів. Основна різниця між мультікомпьютернимі операційними системами і розподіленими системами полягає в тому, що в першому випадку зазвичай мається на увазі, що апаратне забезпечення повністю керовано. Безліч розподілених систем, однак, будується на базі існуючих операційних систем.

1.4 Суперкомп’ютери

Суперкомп’ютер – спеціалізована обчислювальна машина, значно перевершує за своїми технічними параметрами і швидкості обчислень більшість існуючих в світі комп’ютерів.

Для сучасних суперкомп’ютерів характерно:

- масовий паралелізм;
- масштабованість;
- стохастичний характер ресурсів.

2 ОЦІНКА ПРОДУКТИВНОСТІ БАГАТОПРОЦЕСОРНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМ

Фундаментальне розуміння Амдала щодо паралельної обробки було успішно застосовано в багатьох контекстах, включаючи розробку високопродуктивних обчислень (ВПО) систем і включення готових багатопроцесорних систем, відомих як кластери «Біовульф». Тож не дивно, що з появою багатопроцесорних (CMPS), з десятками або сотнями тисяч процесорних ядер, комп'ютерні вчені застосували закон Амдала в освоєнні архітектури CMP.

CMP вперше з'явився в науково-дослідних лабораторіях в середині 1990-х років, і до середини 2000-х років, великі виробники комерційних мікропроцесорів, включаючи IBM, Sun, AMD і Intel все випускали продукти CMP. Цей зсув парадигми в чіп-багатпроцесорній сфері призвів до нових можливостей для високої продуктивності і високої ефективності обчислень, але реалізації оптимальних архітектур є непростим завданням для сучасних комп'ютерних архітектур. Зокрема, як перехід CMPS з декількох процесорних ядер на чіпі в десятки або сотні, важливість проектування збільшується на рівні системи значно. Наступне покоління CMPS включатиме в себе асиметричні або гетерогенні процесори декількох різновидів, складну ієрархію пам'яті, а також асортимент спеціалізованих прискорювачів алгоритму. У цьому випадку частина системи не можуть бути оптимізовані окремо, а необхідний глобальний підхід оптимізації.

2.1 Оцінка прискорення обчислень в GRID системах

Якщо припустити, що при обміні даними між сервером і обчислювальним вузлом другий проводить один раз активність сервера (пінг

пакети), потім відправляє лічені дані і отримує новий пакет. тоді отримаємо, що для всіх пакетів в завданні затримка обчислюється як:

$$T_1 = q * k_1, \quad (2.1)$$

де q – загальна кількість пакетів;

k – час затримки (через перевірки активності сервера).

Передача даних від сервера до обчислювальному вузлу і назад.

Коли стався коннект сервера і обчислювального вузла починається обмін даними з мінімальною швидкістю. Але так як сервер здатний обслуговувати не один обчислювальний вузол одночасно і цих обчислювальних вузлів багато, то вважається, що обмін відбувається з максимальною для сервера швидкістю. Якщо середня швидкість буде менше максимальної швидкості сервера, то можна буде визначити верхню межу можливого прискорення. Взявши цю інформацію до уваги, виходить, що затримка на передачу даних буде обчислюватися як:

$$T2 = q * r/e \quad (2.2)$$

де q – загальна кількість пакетів;

r – обсяг даних обміну складається з результатів обчислень обчислювального вузла і даних нового пакета від сервера;

e – швидкість обміну з боку сервера (біт в секунду).

Обмін між сервером і обчислювальним вузлом можна розділити на кілька етапів. У першому етапі обчислювальний вузол з’єднується з сервером, а потім сервер передає пакет. У наступних етапах всі дії повторюються таким чином, що йде з’єднання, передача результату, передача пакета. Слід зауважити, що між передачею результату і передачею пакета відсутні додаткові тимчасові затримки, і обсяг даних об’єднується. Так само враховується, що передача відбувається послідовно, так як сервера може

знадобитися перевірка або обробка результатів перед відправкою нового пакета.

Черга обчислювальних вузлів до сервера. Якщо кількість вузлів дуже величезне, то вони шикуються в чергу до сервера для обміну даними. Частоту звернення до сервера обчислювальних вузлів обчислюють таким способом:

$$z = (c*S)/t \tag{2.3}$$

де c – кількість обчислювальних вузлів;

s – час обробки пакета сервером;

t – час обробки пакета обчислювальним вузлом.

Час обробки пакета на сервері – час, який витрачає сервер між закінченням прийому результатів обчислень початком передачі нового пакету. Час обробки пакета на обчислювальному вузлі – час, який витрачає обчислювальний вузол між закінченням прийому пакета і початком з'єднання з сервером для відправки результатів обчислень.

Довжина черги до сервера (3) буде обчислюватися як:

$$m(z,n) = \frac{\frac{z^{n+1}}{n*n!*(1-\frac{z}{n})^2}}{\sum_{k=0}^{n}\frac{z^k}{k!}+\frac{z^{n+1}}{n!*\max((n-z),0)}}, \tag{2.4}$$

де z – як часто обчислювальні вузли звертаються до сервера;

n – кількість серверних каналів для обробки запиту обчислювальних вузлів.

Імовірність появи черги до сервера (3) буде обчислюватися як:

$$p(z,n) = \min(1 - \sum_{h=0}^{n}\frac{\frac{z^h}{h!}}{\sum_{k=0}^{n}\frac{z^k}{k!}+\frac{z^{n+1}}{n!*\max((n-z),0)}}, \tag{2.5}$$

Підсумкова формула для оцінки прискорення. В результаті узагальнена формула оцінки прискорення в GRID системах матиме вигляд:

$$S(a,c,k,t,e,r,n) = \frac{1}{(a+(1-a)*\left(\frac{1}{c}+\frac{k}{t}+\frac{r}{e*t}*\left(1+p\left(z\left(c,\frac{r}{e},t\right),n\right)*m\left(z\left(c,\frac{r}{e},t\right),1\right)\right)\right))},$$

Де a – частка послідовних обчислень;

c – кількість обчислювальних вузлів;

k – тимчасова затримка перед початком передачі даних;

t – час обробки одного пакета на одному обчислювальному вузлі;

e – серверна швидкість передачі; r – об’єм пакету; n – кількість каналів обробки обчислювальних вузлів на сервері;

m – довжина черги до сервера (визначається на основі 4);

p – ймовірність появи черги до сервера (встановлюється на основі 5.

Точка максимуму прискорення обчислень в залежності від кількості обчислювачів визначається як:

$$\frac{d}{dc}S(a,c,k,t,e,r,n) = 0\,,$$

2.2 Мінімізація кількості обчислювальних вузлів в GRID системах

Завдання сформульоване таким чином: для заданого часу обчислень T необхідно знайти оптимальну кількість обчислювальних вузлів. Інші параметри фіксовані. Як відомо, оптимізуватися може тільки розпаралелювання частина завдання (для неї послідовна частина дорівнює 0) і прискорення, з огляду на закон Амдала, буде дорівнює кількості працюючих обчислювачів. Врахувавши що, завдання вирішується t секунд на одному обчислювачі, то на c обчислювачах виходить:

$$T_A = \frac{t_1}{c} \tag{2.6}$$

$$S(c,k,t,e,r,n) = \frac{1}{(\frac{1}{c}+\frac{k}{t}+\frac{r}{e*t}*(1+p(z(c,\frac{r}{e}t),n)*m(z(c,\frac{r}{e}t),1)))}, \tag{2.7}$$

2.3 Гетерогенні GRID

Гетерогенні системи – системи з неоднорідними технічними характеристиками і вузлами, для яких характерні різні часові витрати на різних вузлах при однакових пакетах. Для аналізу розрахунків в гетерогенних системах необхідно поняття одиниці обчислювальної потужності вузла. Обчислювальна потужність вузла (N) – кількість оброблених вузлом еталонних пакетів за одну секунду. Еталонний пакет – один з пакетів в завданню, щодо якого будуть оцінюватися параметри інших пакетів і обчислювальних вузлів. Таким чином, завдання може розглядатися у вигляді наборів пакетів Q (N), а GRID система у вигляді набору обчислювальних вузлів C (N). Сукупність Q (N), C (N) описують кожен вузол.

2.4 Модифікований закон Амдала

Закон Амдала у вигляді

$$S_p = \frac{1}{\alpha+\frac{1-\alpha}{p}}$$

Де S_p – величина прискорення виконання програми на p процесорах порівнює виконання на одному процесорі, α – частка послідовної частини програми. (1 – α) – частка распараллелівать частини програми) не дозволяє його реалізувати на довільному числі процесорів. По-перше, до реалізації

програми в системі необхідно затратити час на те, що б дізнатися які частини програми можна распаралліть. По-друге, необхідно визначити частку распараллеліной частини і послідовної, а так же тимчасові витрати на здійснення обміну інформацією для паралельних обчислень. Можна припустити, що при збільшенні процесорів можна досягти точки максимального прискорення S , а потім воно піде на спад. Тому було запропоновано наступне «поправка до закону. Частка програми, що передбачається для розпаралелювання, повинна ділитися на дві частини: частки програми, безпосередньо реалізуються кожним окремим процесором, і частки системних втрат програми, що витрачаються в системі на кожен процесор. При цьому частка системних втрат програми повинна бути пропорційна числу процесорів, що використовуються в системі «. Тому закон Амдала прийме наступний вигляд:

$$S_{pk} = \frac{1}{\alpha+(1-a)(\frac{1}{p}+kp^{n})}, \tag{2.8}$$

де k і n безрозмірні константи.

Дані константи повинні бути знайдені або на основі аналітичного дослідження, або на експериментальних даних. При знаходженні даних констант ми зможемо визначити оптимальне число процесорів по наступній формулі:

$$p_{0k} = \sqrt[n+1]{\frac{1}{kn}}.$$

Так як програми можуть мати ієрархічну структуру, то для них може знадобитися розпаралелювання на окремих фазах. Тому необхідна модифікація закону для таких випадків. Отримати загальне вираз для закону при будь–якому числі фаз не представляється можливим. Тому ми

розглянемо викладену ідею тільки для випадку, коли обчислювальний процес має тільки дві фази. Тоді для другої фази отримаємо:

$$G_q = \frac{\frac{1-\alpha}{p_0}}{\frac{1-\alpha}{p_0}\beta+(1-\frac{1-\alpha}{p_0}\beta)(\frac{1}{q}+dq^m)}, \quad (2.9)$$

$\frac{1-\alpha}{p_0}$ – оптимальна частка програми, яка виконується після распаралелліванія на першій фазі процесу без урахування системних втрат;

β – частка програми на другій фазі;

d^q – поправка, що враховує величину системних втрат на організацію паралельних обчислень процесорами другої фази;

d, m – безрозмірні константи.

3 ТРАДИЦІЙНА ОРГАНІЗАЦІЯ МУЛЬТІЗАПИТІВ

Зі зростаючою важливістю оперативної аналітичної обробки даних техніка більш складних оптимізацій запитів стала вирішальною. Для того щоб бути ефективними, оптимізатори повинні адаптуватися до нових операторів, зміни в методах оцінки вартості і т.д. Ці вимоги привели до нинішнього покоління оптимізаторів запитів, з яких два оптимізатора Starburst [26] і Volcano [14] є найбільш представницькими. У той час як IBM DB2 оптимізатор [10] заснований на Startburst, Microsoft SQL-Сервер оптимізатор, що заснований на Volcano. Основною відмінністю між двома підходами є те, яким чином альтернативні плани створюються. Оптимізатор Startburst генерує плани від низу до верху, тоді як оптимізатор Volcano створює плани зверху. План складається з операторів (наприклад, вибрати, приєднатися, сортування). Вартість оператора є функцією статистичного зведення, яка включає в себе розмір відносини і число різних значень атрибутів, розподіл цих значень атрибутів в термінах гістограм і т.д. Хоча точність цих статистичних даних має вирішальне значення, оцінка вартості плану може бути чутлива до цієї статистики і підтримку цих статистичних даних може вимагати багато часу. Проблемі ефективного підтримки досить точної статистики приділяється велика увага в літературі. Крім того, модель витрат повинна враховувати вплив, наприклад, буферизації відносин в кеші бази даних, моделі доступу до пам’яті, доступність виконання оператора, конвеєрне виконання і т. і.

3.1 Евристичні алгоритми

«Багаторазова оптимізація запитів» (MQO) вивчалася з 1980-х років. МКО намагається зменшити вартість виконання групи запитів, виконуючи спільні завдання тільки один раз, тоді як традиційна оптимізація запитів

розглядає єдиний запит за один раз.

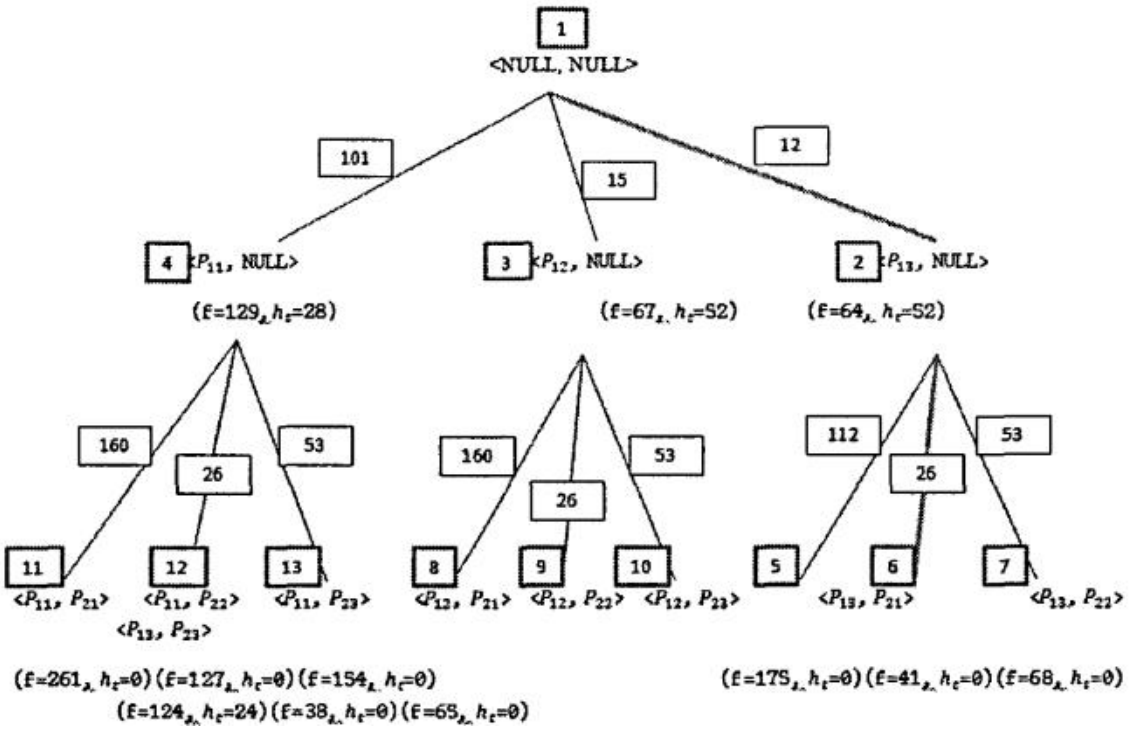

Рисунок 3.1 – Часткова область пошуку з використанням евристичної функції

Проблема MQO була сформульована в [1, 16, 33] як повна NP проблема оптимізації.

Нехай вхідні параметри для MQO задані як:

- Q запити: q_1, q_2, .., q_Q.
- T задачі: t_1, t_2, .., t_T.
- P плани.

Приклад на рисунку 1.2 пояснює проблему MQO.

Розглянемо типову архітектуру системи, в якій застосовуються СКБД з оптимізатором запитів, наведену в патенті.

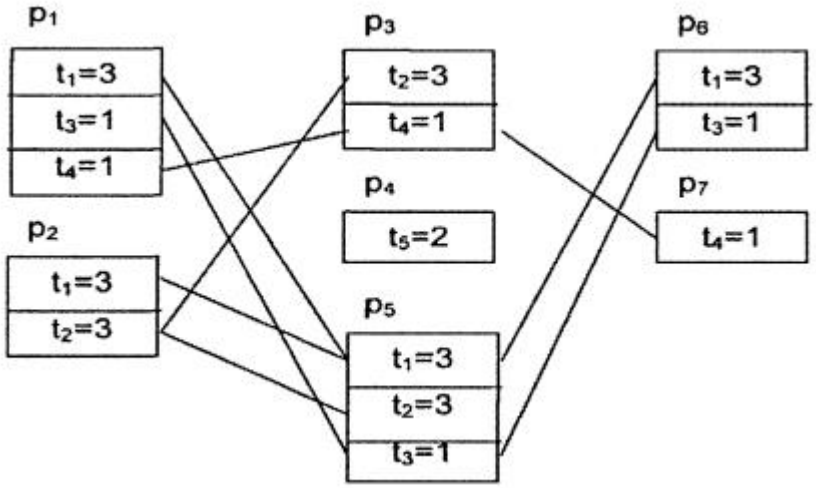

Рисунок 3.2 – Часткова область пошуку з використанням евристичної функції

3.2 Типова архітектура системи

Даний варіант в цілому являє собою комп'ютерну систему, в якій один або більше число комп'ютерів (система обробки інформації 100, один або більше число комп'ютерів клієнта 101, 101a і т.д., керуючий комп'ютер 102, один або більше число серверів БД 105, 105a) з'єднані між собою з боку клієнта мережею 103 і з боку сервера мережею 104. мережа клієнта 103 і мережу сервера 104 можуть бути локальними мережами. Ці мережі можуть також бути мережевим з'єднанням між комп'ютерами і / або мережевим з'єднанням між елементами процесора в паралельному комп'ютері. Більш того, мережа з боку клієнта 103 і мережу з боку сервера 104 можуть бути однією і тією ж мережею.

3.2.1 Система обробки даних

Система обробки даних 100, комп'ютери клієнта 101, 101a, керуючий комп'ютер 102, сервери БД 105, 105a, можуть бути комп'ютерами будь-якого типу, включаючи персональний комп'ютер, робочу станцію, паралельний комп'ютер, комп'ютер мейіфрейм і невеликий переносний ноутбук. На Рис.

1.8: кожен елемент системи обробки даних 100 - комп’ютер клієнта 101, 101a, комп’ютер управління 102, сервери БД 105, 105a показані як незалежні окремі комп’ютери.

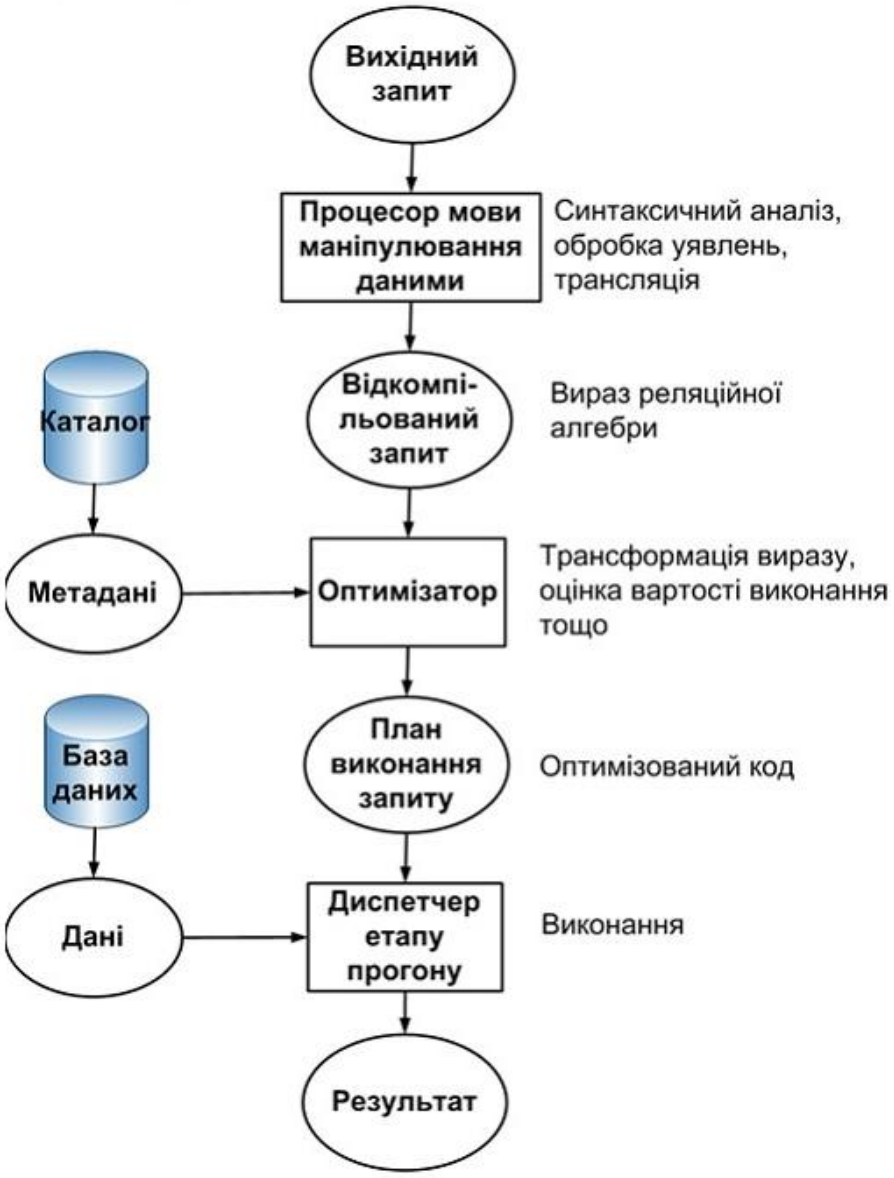

Рисунок 3.3 – Оптимізатор запитів

Однак будь-яка комбінація цих комп’ютерів може також являти собою один комп’ютер. Системи обробки даних 100, комп’ютера клієнта 101, 101a, комп’ютера управління 102, серверів БД 105, 105a, мережі з боку клієнта 103, мережі з боку сервера 104 і їх конфігурація на рисунку 3.3.

Додатки 120, 120a, які є програмами для обробки інформації

користувачем, функціонують на комп’ютерах клієнта 102, 102а. Додаток 102 робить запит (наприклад, запит, сформульований на SQL), щоб зробити посилання або оновити БД, якщо це необхідно. Сервери БД 105, 105а та інші мають одну або кілька БД. СУБД 122, яка являє собою програму, призначену для того, щоб робити посилання або оновлювати БД у відповідь на запит від інших програм, працює на серверах БД 105, 105а. У багатьох випадках СУБД 122 містить одну або більше число БД, як своїх цільових даних для управління в пристроях зовнішньої пам’яті 106, 106а. БД може являти собою реляційну БД, яка складається з однієї або більшого числа таблиць, що мають одну або більше число стовпців, а може також являти собою БД інших типів (БД мережевого типу, ієрархічну БД, об’єктно-орієнтовану БД або сховище об’єктів) або системи файлів . Система обробки даних 100 отримує первинний запит, зроблений одним з комп’ютерів клієнта 101, 101а, створює один або більше число вторинних запитів і передає їх серверам БД 105, 105а, в тому випадку, якщо це необхідно, виконує посилання або оновлення даних, так як це визначено в первинному запиті, а потім повертає отримані в результаті дані на комп’ютер клієнта, з якого був зроблений первинний запит. Керуючий комп’ютер 102 використовує управляє додаток 121. Керуючий додаток 121 - це програма управління системою обробки інформації 100 або всієї системи.

Модуль обробки введення/виводу ПО, аналізатор запитів 111, оптимізатор запитів 112, модуль обробки запитів 113, контролер оптимізації 114, пристрій зовнішньої пам’яті 115 є елементами, складовими модуль обробки інформації 100. Модуль обробки введення/виводу ПО приймає заявку про запит від комп’ютерів клієнта 101, 101а та заявку про управління від комп’ютера, що управляє 102 і відповідає на ці заявки. Аналізатор запитів 111 виробляє лексичний аналіз, синтаксичний аналіз, і семантичний аналіз заявки про запит, отриманої модулем обробки введення / виводу ПО, виробляє стандартні перетворення умов запиту, в разі, якщо це необхідно, а потім створює на основі запиту дерево запиту. Оптимізатор запитів 112

оптимізує запит, використовуючи дерево запиту, сгенерированное аналізатором запитів 111, і розробляє процедуру для серії операцій (план виконання запиту), щоб отримати результати обробки запиту. У разі реляційної БД серія операцій в плані виконання запиту включає процес вибірки, проекцію, з'єднання, угруповання і сортування. План виконання запиту – це структура даних, яка описує, як застосовувати ці операції щодо цільової БД (тобто описуються цільова БД, цільової сервер БД 105 і порядок обробки запиту). Модуль обробки запитів 113 виконує план виконання запиту, розроблений оптимізатором запиту 112. Модуль виконання запитів 113 може зажадати шляхом посилки запиту на сервери БД 105, 105a, щоб сервери БД 105, 105a частково або повністю виконали цю серію операцій. Модуль обробки запитів 113 може також сам частково або повністю виконати цю серію операцій шляхом обробки даних отриманих від серверів БД 105, 105a.

3.2.2 Контролер оптимізації

Контролер оптимізації 114 інтерпретує заявку про управління, отриману модулем обробки введення / виводу ПО, і зберігає визначення класифікації запиту і напрямок операцій з обробки запиту, включені в заявку про управління в пристрої зовнішньої пам'яті 115, в разі необхідності. Більш того, контролер оптимізації 114 отримує в своє розпорядження запит (і додаткову інформацію про запит), який раніше вчинив на аналізатор запитів 111, отримує направлення операцій з обробки запиту, яке відповідає цьому запиту згідно з визначенням класифікації запиту, а потім видає направлення операцій по обробці запитів на оптимізатор запитів 112 і модуль обробки запитів 113.

Для реляційних систем оптимізація являє собою як проблему, так і сприятливу можливість. Проблема полягає в тому, що для досягнення прийнятного рівня продуктивності оптимізація в подібних системах просто

необхідна. Причому однією з сильних сторін і безперечних переваг реляційного підходу є те, що реляційні вирази реалізуються і оптимізуються на досить високому семантичному рівні. На противагу цьому, в нереляційних системах, де запити користувачів виражаються в більш низькому семантичному рівні, будь-яка «оптимізація» повинна виконуватися самим користувачем вручну (тут термін «оптимізація» взятий в лапки, оскільки зазвичай він використовується для позначення автоматичної, а не ручний оптимізації) . У подібних системах користувач (а не система) визначає, які саме низькорівневі операції повинні бути виконані і в якій послідовності. І якщо користувач прийняв неправильне рішення, то система не здатна виправити становище. Відзначимо також, що для роботи в подібних системах користувач повинен володіти деякими навичками в програмуванні, інакше він не зможе досить повно застосовувати засоби цих систем. Перевага автоматичної оптимізації полягає в тому, що користувач може не замислюватися над кращим способом вираження своїх запитів (про те, як слід сформулювати запит, щоб система виконала його з максимальною продуктивністю). Але і це далеко не все. Цілком ймовірно, що оптимізатор сформує запит краще, ніж сам користувач. Для такого твердження є кілька вагомих причин. Нижче наведені лише деякі з них:
1. Хороший оптимізатор має великий обсяг корисної інформації, для користувача зазвичай недоступна. Говорячи конкретніше, оптимізатор має певні статистичними даними (в тому числі про кардинальність стовпчиків таблиць і т.д.), в тому числі наступними:

- кількість різних значень кожного типу;

- поточну кількість рядків в кожній базової змінної таблиць;

- поточна кількість різних значень для кожного атрибута кожної базової змінної таблиці;

- кількість входжень кожного значення в кожному з атрибутів тощо.

Вся ця інформація зберігається в системному каталозі. Завдяки наявності таких даних оптимізатор може більш точно оцінити ефективність

будь-якої можливої стратегії реалізації конкретного запиту. Виходячи з отриманих результатів він зможе вибрати найкращу стратегію реалізації запиту. Якщо з часом статистичні показники бази даних значно зміняться (наприклад, в результаті її фізичної реорганізації), то при реалізації запиту може виявитися доцільніше використовувати іншу стратегію, відмінну від прийнятої раніше. Іншими словами, виникне необхідність повторної оптимізації, або реоптімізації. У реляційних системах процес реоптімізації цілком тривіальний і зводиться до простої повторній обробці вихідного запиту системним оптимізатором. У нереляційних системах виконання реоптімізації, як правило, вимагає коригування програми і, цілком можливо, може виявитися взагалі неможливим. 3. оптимізатор – це програма, яка за визначенням «завжди доводить почате роботу до кінця», на відміну від людини. Оптимізатор здатний розглядати буквально сотні різних стратегій реалізації конкретного запиту, в той час як програміст ледь проаналізує більше трьох – чотирьох можливих стратегій (по крайней мере, досить глибоко).

У оптимізатор реалізовані знання і досвід кращих з кращих програмістів, в результаті чого ці знання і досвід стають доступними всім. Це дозволяє застосовувати обмежений набір ресурсів, наданий широкому колу користувачів, найбільш ефективно і економічно. Наведені вище міркування є переконливим доказом зробленої на початку даного розділу заяви про те, що оптімізуемість (тобто можливість оптимізації запитів) є сильною стороною реляційних систем.

Загальне призначення оптимізатора полягає у виборі найбільш ефективної стратегії обчислення реляційного вираження. Ще одне вступне зауваження. Досить часто цю тему пов’язують з оптимізацією запитів. Однак подібний термін кілька неточний, оскільки вираз, потрібно оптимізувати (запит), у справі може формуватися в контексті який відрізняється від інтерактивної вибірки інформації з бази даних. Зокрема, він може бути частиною операції оновлення, а не операції вибірки, що розуміється під

запитом. Більш того, сам по собі термін оптимізація є дещо перебільшеним, оскільки зазвичай не існує гарантій, що обрана стратегія реалізації дійсно оптимальна в деякому об’єктивному сенсі. На практиці під оптимізованої стратегії реалізації зазвичай розуміється просто покращений варіант вихідного неоптимізованого вираження. (Тим не менш, в деяких рідкісних випадках можна цілком обгрунтовано стверджувати, що обрана стратегія реалізації дійсно буде оптимальною в певному сенсі).

3.3 Оптимізація однопроцесорної обробки запитів

Нехай безліч запитів $З_i$ (і від 1 до N), утворюють мультизапит, при цьому запити $З_i$ сформовані як кон’юнкція елементарних запитів, з яких ряд елементарних запитів повторно входять в запити $З_i$.

Відомим методом збільшення продуктивності баз даних обчисювальних систем є одночасне виконання декількох запитів, що утворюють мультизапит. У нашій роботі розглядається ефективність виконання мультизапита в базах даних одно- та багатопроцесорної обчислювальної системи.

Мінімальний час виконання одного запиту в однопроцесорній обчисювальній системі визначається відповідно до умов впорядкованості(1). При виконанні елементарних запитів в порядку, визначеному умовою (2). Для упорядкованих стовпців таблиці співвідношенням (3) при виконанні елементарних запитів в порядку, визначеному умовою (4).

де N – рядків таблиці бази даних,

K – кількість стовпців таблиці,

Ті – час обробки і-го елементарного запиту $ЕЗ_i$ для одного рядка таблиці, Рі – ймовірність успіху при обробці і -го елементарного запиту $ЕЗ_i$ для одного рядка таблиці (дані одного рядка таблиці відповідають умові, заданої елементарним запитом $ЕЗ_i$).

Мультізапит може виконуватися двома способами: незалежно один від

одного і спільно, коли виділяються підмножини співпадаючих елементарних запитів, які виконуються в першу чергу з метою зменшення сумарного числа елементарних запитів.

Існує завдання визначення часу виконання кон'юнктивний мультізапроса при незалежній і спільного обробітку з тим, щоб визначити метод оптимізації однопроцессорной обробки мультізапросов, при якому досягається мінімальний час його виконання.

$$T_{нез} = T_{З_1} + T_{З_2} = n\left((a + pa^2 + p^2a^3 + \cdots + p^{k-2}a^{k-1}) + (1 + pa^{k-1})\right) =$$

$$= n\left(1 + pa^{k-1} + a\frac{1-(pa)^{k-1}}{1-pa}\right).$$

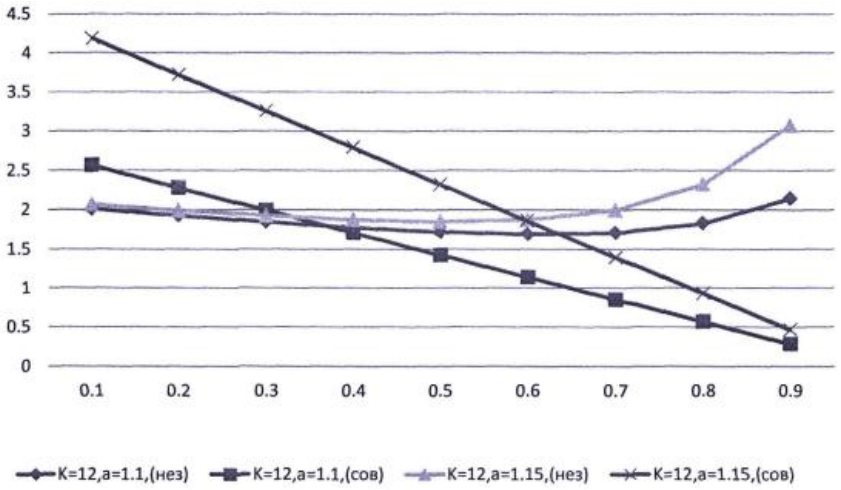

Рисунок 3.3 – Незалежна обробка запитів мультізапиту для неупорядкованої таблиці

Нехай дані в стовпцях таблиці даних впорядковані [12]. При незалежної обробці запитів загальний час їх виконання дорівнює

$$T_{нез} = T_{З_1} + T_{З_2} = n\left((pa + p^2a^2 + p^3a^3 + \cdots + p^{k-1}a^{k-1}) + (p1 + p^2a^{k-1})\right) =$$

$$= n\left(p + p^2a^{k-1} + pa\frac{1-(pa)^{k-1}}{1-pa}\right).$$

Спільна обробка є краще незалежної, якщо вона забезпечує мінімальний час виконання, тобто якщо справедливо нерівність:

$$p + p^2 a^{k-1} + pa\frac{1-(pa)^{k-1}}{1-pa} > pa^{k-1} + p^2 + p\frac{1-(pa)^{k-1}}{1-pa} - p.$$

3.4 Стадії обробки запиту

Перетворення запиту відразу у його внутрішню форму. На цій стадії виконується базове перетворення даного запиту в деякий внутрішній вид, більш зручний для його машинної обробки. В результаті маємо розгляд конструкції суто зовнішнього рівня (як варіант, різноманітні варіанти конкретного синтаксису обраної мови запитів) і готується ґрунт для подальших стадій її оптимізації. Примітка. На цьому етапі також виконується обробка уявлень (тобто процес заміни посилань на представлення виразів, що відповідають представленням). Очевидне питання: «Яке формальне представлення слід використовувати для внутрішнього подання запиту?». Незалежно від того, яка форма формального представлення обрана, вона повинна бути достатньо повною, щоб дозволяти надсилати будь-які запити, які можуть бути визначені зовнішньою мовою запитів. Крім того, обраний метод формального викладу повинен бути максимально нейтральним, в тому сенсі, що він не повинен визначати подальші рішення щодо оптимізації. Найчастіше та чи інша модифікація абстрактного синтаксичного дерева використовується для внутрішнього представлення запитів, яке в даному випадку називається деревом запитів.

Рисунок 3.4 – Дерево реалізаці запиту

Однак для наших цілей найзручніше припустити, що один із формальних методів, з якими ми вже знайомі, використовується для внутрішнього представлення запитів: реляційна алгебра або реляційне числення. У цьому випадку дерево запитів, подібне до показаного на рис. 33.2, може розглядатися просто як альтернативне схематичне зображення деякого виразу, записаного в системі нотацій одного з двох запропонованих вище формальних методів (реляційна алгебра або реляційне числення) . Припустимо, що обраним формальним методом є реляційна алгебра. З цього моменту будемо вважати, що внутрішнє подання дерева запитів, показане на рис. 33.2, є наступним алгебраїчним виразом (приклад 3).

Внутрішнє представлення дерева:

({SP JOIN S) WHERE P# = P# (’P2’)) {SNAME}.

Перетворення запиту в канонічну форму.

На цьому етапі оптимізатор виконує кілька операцій оптимізації, які гарантовано є прийнятними незалежно від фактичних значень даних та існуючих способів доступу до них. Справа в тому, що мови реляції зазвичай дозволяють сформулювати будь-який запит (крім, мабуть, найпростішого) кількома різними способами. Наприклад, навіть простий запит «Ідентифікувати назви постачальників деталей з номером P2» в SQL можна

записати буквально десятками способів, не рахуючи таких тривіальних варіантів, як заміна a = b на b = a або p AND q на q I стор. Продуктивність обчислення запитів не повинна залежати від обраного користувачем формату запиту. Тому наступним кроком при обробці запиту є перетворення його внутрішнього подання в якусь еквівалентну канонічну форму. Мета цього перетворення – усунути зовнішні відмінності в еквівалентних поданнях запиту і, що ще важливіше, знайти представлений запит, в деякому сенсі більш ефективним, ніж оригінальне подання. Зауваження щодо канонічних форм. Поняття канонічної форми є центральним у багатьох розділах математики та суміжних дисциплін. Канонічну форму можна визначити наступним чином. Нехай Q – сукупність об’єктів (запитів) і нехай існує поняття їх еквівалентності (а саме запити q1 і q2 еквівалентні, якщо і лише тоді, коли вони дають однакові результати). Тоді підмножина C множини Q являє собою сукупність канонічних форм для запитів із множини Q у значенні еквівалентності, визначеної вище тоді і лише тоді, коли кожному об’єкту q з множини Q відповідає лише один об’єкт c з множини C. У цьому випадку ми говоримо, що об’єкт c є канонічною формою об’єкта q. Усі властивості об’єкта q, які нас цікавлять, також притаманні об’єкту c. Тому для доказу результатів, які нас цікавлять, достатньо вивчити менш потужний набір об’єктів C, а не більш потужний набір Q.

На цьому етапі оптимізатор виконує кілька операцій оптимізації, які гарантовано є прийнятними незалежно від фактичних значень даних та існуючих способів доступу до них. Справа в тому, що реляційні мови зазвичай дозволяють сформулювати будь-який запит (крім, мабуть, найпростішого) кількома різними способами. Наприклад, навіть при простому перекритті тесту Етапу 1 в еквівалентній, але швидко ефективній формі, оптимізатор використовує певні і відомі правила або законні перетворення. Нижче я намагаюся застосувати такі правила.

Вибір потенційних низькорівневих процедур.

Після перетворення внутрішньої форми запиту у більш відповідну

(канонічну) форму оптимізатор повинен визначити, як виконати набір запитів у цій канонічній формі. На цьому етапі враховуються такі фактори, як наявність індексів та інших фізичних шляхів доступу, статистичний розподіл існуючих значень даних, фізична кластеризація збережених даних тощо. Зауважте, що на етапах 1 і 2 цим аспектам взагалі не приділяли уваги. Основна стратегія – трактувати вираз запиту як ряд операцій низького рівня, які певною мірою залежать одна від одної. Далі наведено приклад такої взаємозалежності. При програмному виконанні проекційних операцій зазвичай вимагається, щоб рядки, які читаються, були у певній послідовності, що виключає дублювання отриманих рядків. У свою чергу, це означає, що операція, що виконується безпосередньо перед проекцією, повинна видавати вихідні кортежі в точно необхідній послідовності. Для кожної операції низького рівня (і, можливо, для декількох комбінацій подібних операцій, які є загальними), оптимізатор має набір процедур реалізації низького рівня. Наприклад, існує набір процедур виконання операції скорочення: за умови рівності певного значення потенційного ключа; на основі індексованого атрибута, за яким здійснюється зменшення; на основі атрибута хеш і т.д. Кожна процедура пов’язана з параметризованою формулою витрат, яка дозволяє оцінити вартість процедури (тобто рівень витрат, необхідних для її виконання). Найчастіше вартість визначається підрахунком кількості необхідних операцій введення-виведення диска, але деякі системи також враховують використання процесора та інші фактори. Ці формули витрат використовуються на етапі 4. Раніше формули витрат для різних процедур впровадження під різні початкові припущення обговорювали та аналізували.

Далі, використовуючи інформацію з каталогу стану бази даних (існуючі індекси, поточну кардинальність таблиць змінних тощо) та інформацію про взаємозалежності, згадані вище, оптимізатор вибирає одну або більше процедур-кандидатів для кожної операції низького рівня у запиті. Цей процес зазвичай називають вибором шляху доступу. Примітка. Слід зазначити, що в терміні вибір шляху доступу використовується для

позначення певних у цій лекції етапів 3 та 4, а не лише 3-го етапу. Дійсно, на практиці дуже важко відрізнити, де закінчується один етап, а інший починається ; просто етап 3 більш-менш плавно переходить до 4-ї стадії.

Генерація різних варіантів планів обчислення запиту і вибір плану з мінімальними витратами.

На останньому етапі процесу оптимізації створюється набір потенційних планів запитів, після чого вибирається найкращий (тобто найменш дорогий) план виконання запитів. Кожен план виконання формується як сукупність певного набору можливих процедур реалізації, і кожна операція низького рівня в запиті відповідає одній процедурі. Зауважте, що для початкового запиту зазвичай може бути багато планів виконання запитів (можливо, навіть багато). На практиці, напевно, не варто генерувати всі можливі плани запитів, оскільки деякі з них можуть бути комбінаторними версіями інших планів виконання того ж запиту, а процес вибору найбільш ефективного плану може бути надто дорогим. Вибираючи план з найнижчою вартістю, рекомендується (можливо, навіть потрібно) дотримуватися деяких евристичних правил, що дозволяють обмежити кількість аналізованих планів запитів до розумних меж. Практика обмеження кількості запитів до розумних обмежень інакше називається скороченням простору пошуку, оскільки це може розглядатися як зменшення діапазону варіантів, проаналізованих оптимізатором (пошуковим простором), до контрольованого розміру. Безперечно, метод визначення вартості будь-якого можливого плану необхідний для вибору плану з найменшими витратами. По суті, вартість плану – це просто сума витрат на окремі процедури, що входять до його складу. Тому завданням оптимізатора є розрахунок формул витрат для кожної окремої процедури. Основна проблема полягає в тому, що формули вартості процедури залежать від розміру таблиці (або таблиць), яку обробляє процедура. Оскільки всі запити, крім найпростіших, вимагають створення деяких проміжних результатів (принаймні концептуально), оптимізатор повинен мати можливість оцінити розмір цих проміжних результатів і

використовувати отримані значення при розрахунку формул витрат. На жаль, розмір проміжних наборів даних сильно залежить від конкретних значень збережених даних, тому точна оцінка витрат може бути досить складним завданням.

Перетворення виразів.

У цьому розділі обговорюються правила трансформації (або закони), які часто використовуються на етапах 2 процесу оптимізації. Звичайно, майте на увазі, що спроби використовувати для написання одного правила конверсії можна будь-коли, тоді як застосовується інше правило конверсії. Наприклад, середні джерела записів – це не дуже поширені запити, сформульовані з двома дітьми, які були досить чітко розроблені. Однак при перекритті вони пишуть, що вони часто ростуть, часто використовують інші правила трансформації. У цьому сенсі подання є дуже важливим випадком. Ймовірно, ми розглянемо запит «Вибрати всі типи життя в представленні v», де представлений V-проект, який буде запропоновано для створення атрибутів S # і CITY). Іншими словами, відкриваючись від вірусного джерела, оптимізатор буде поетапно зберігатися різними правилами перетворення, поки ви не отримаєте виразного вигляду, який, вводячись у евристичні правила оптимізатора, буде запропонований оптимальним для відповідного запису.

Семантичні перетворення.

Приклад 4. З’єднання типу «зовнішній ключ – відповідний потенційний ключ»: (SP JOIN S) {P#}.

Приклад 5. Семантично еквівалентний вираз: SP {P#}.

Це з’єднання відноситься до таких з’єднань, як «зовнішній ключ – відповідний потенційний ключ». У ній зовнішній ключ у змінному відношенні SP узгоджується з потенційним ключем у змінному відношенні S. Отже, кожен кортеж у змінному відношенні SP з’єднаний із певним кортежем у змінному відношенні S. Таким чином, від кожного кортежа в коефіцієнт змінної SP у загальному результаті значення атрибута p #. Іншими словами,

підключення може не здійснюватися взагалі! Розглянутий приклад 4 можна замінити прикладом 5., проте, будьте обережні – таке перетворення застосовується лише до обгрунтованості семантики (значення) конкретної ситуації. У загальному випадку кожен операнд в операції з’єднання вносить кортежі у з’єднання, у яких немає відповідних кортежів в інших операндах, і тому деякі кортежі не сприяють загальному результату. Таким чином, найчастіше трансформація такого типу буде неприйнятною. Однак у цьому конкретному випадку кожен кортеж у змінному відношенні SP повинен мати відповідний кортеж у змінному відношенні S відповідно до встановленого обмеження цілісності (фактично це обмеження цілісності за посиланням), що вказує на те, що кожна подача деталей повинна бути пов’язаний з конкретним постачальником. . Тому, враховуючи це зауваження, можна стверджувати, що в цьому випадку розглянута трансформація є цілком прийнятною. Перетворення, допустиме лише тому, що існує специфічне встановлене обмеження цілісності, називається смисловим перетворенням, а оптимізація, досягнута в результаті таких перетворень, називається семантичною оптимізацією. Семантичну оптимізацію можна визначити як процес перетворення одного запиту в інший, якісно інший запит, який, однак, гарантує результат, ідентичний результату вихідного запиту, через те, що оброблені дані задовольняють певним обмеженням цілісності. Важливо розуміти, що в принципі будь-яке обмеження цілісності може використовуватися для семантичної оптимізації. Іншими словами, ця методика не обмежується використанням лише обмежень цілісності посилання, як у цьому випадку. Наприклад, припустимо, база даних постачальників і деталей обмежує зберігання всіх червоних частин у Лондоні. Розглянемо наступний запит. Визначте всіх постачальників, які постачають лише червоні деталі та знаходяться в тому ж місті, де зберігається принаймні одна їх частина. Це досить складний запит! Але через розглянуте обмеження цілісності його можна звести до цього виду. Визначте всіх постачальників з Лондона, які постачають тільки червоні

деталі. Примітка. Семантична оптимізація належним чином використовується лише в деяких сучасних комерційних продуктах. Але, в принципі, семантична оптимізація здатна забезпечити значне підвищення продуктивності праці (можливо, набагато вище того, що можна досягти за певну годину будь-якими традиційними методами оптимізації).

3.5 Статистичні показники баз даних

На етапах 3 та 4 загального процесу оптимізації (називаються етапи вибору шляху доступу) використовується статистика бази даних, що зберігається в її каталозі. Для демонстраційних цілей деякі ключові статистичні дані щодо двох комерційних продуктів, DB2 та Ingres, коротко обговорюються нижче (з кількома додатковими коментарями). Ось деякі основні статистичні дані, що використовуються в базі даних DB2. Для кожної базової таблиці записуються такі показники:

- кардинальність;
- кількість сторінок, зайнятих таблицею;
- частка площі столу, яку займає стіл. Для кожного стовпця кожної базової таблиці закріплюються такі показники:
- кількість різних значень у стовпці;
- друге за величиною значення у стовпці;
- друге найменше значення у стовпці;
- десять значень у стовпці (лише для індексованих стовпців), які є найпоширенішими, а також кількість зустрічей кожного з цих значень.

Для кожного показника реєструються такі показники:

- індикатор, який вказує, чи є кластерний індекс (тобто індекс, у якому логічний порядок ключових значень збігається з фізичним порядком розміщення цих значень на диску);
- для кластерних індексів;
- частка індексованої таблиці, яка знаходиться в послідовності

кластерів;

- кількість сторінок сторінок в індексі;

- кількість рівнів в індексі.

Примітка. Наведені вище статистичні дані не оновлюються в режимі реального часу (тобто з кожним оновленням бази даних) через великі затримки, які потрібно використовувати для цього підходу. Натомість статистику оновлюють вибірково, використовуючи системну утиліту RUNSTATS, яка запускається на вимогу адміністратора бази даних, наприклад після реорганізації бази даних. Подібна заява може бути застосована до більшості інших комерційних товарів (але не всіх), включаючи систему Ingres, де відповідна утиліта називається OPTIMIZEDB. Oracle використовує ANALYZE. Ось деякі основні статистичні показники бази даних, що накопичуються в базі даних Інгрес. Примітка. В системі Ingres індекс трактується як особливий випадок збереженої таблиці. Тому наступні статистичні дані для базових таблиць та стовпців також розраховуються для індексів. Для кожної базової таблиці записуються такі показники:

- кардинальність;

- кількість первинних сторінок таблиці;

- кількість сторінок переповнення таблиці.Для кожного стовпця кожної базової таблиці закріплені такі показники:

- кількість різних значень у стовпці;

- максимальне, мінімальне та середнє значення для стовпця;

- реальні значення у стовпці та частота їх виникнення.

4 МЕТОД ОПТИМІЗАЦІЇ ОБРОБКИ ЗАПИТІВ

4.1 Аналіз даних

Зменшення сумарного часу виконання запитів мультізапиту може бути досягнуто за рахунок спільної обробки підмножини елементарних запитів, що є перетином запитів мультізапиту. Відомо [25, 36], що порядок обробки елементарних запитів є істотним параметром часу обробки запитів. Тут розглядається задача формування плану виконання мультізапиту в мультипроцесорній базі даних.

Нехай безліч запитів $З_i$ (і від 1 до N), утворюють мультизапит, при цьому запити $З_i$ сформовані як кон'юнкція елементарних запитів, з яких ряд елементарних запитів повторно входять в запити $З_i$.

Відомим методом збільшення продуктивності баз даних обчисювальних систем є одночасне виконання декількох запитів, що утворюють мультизапит. У нашій роботі розглядається ефективність виконання мультизапита в базах даних одно– та багатопроцесорної обчислювальної системи.

Мінімальний час виконання одного запиту в однопроцесорній обчисювальній системі визначається відповідно до умов впорядкованості(1). При виконанні елементарних запитів в порядку, визначеному умовою (2). Для упорядкованих стовпців таблиці співвідношенням (3) при виконанні елементарних запитів в порядку, визначеному умовою (4).

де N – рядків таблиці бази даних,

K – кількість стовпців таблиці,

Ті – час обробки і-го елементарного запиту $ЕЗ_i$ для одного рядка таблиці, Рі – ймовірність успіху при обробці і -го елементарного запиту $ЕЗ_i$ для одного рядка таблиці (дані одного рядка таблиці відповідають умові, заданої елементарним запитом $ЕЗ_i$).

Мультізапит може виконуватися двома способами: незалежно один від одного і спільно, коли виділяються підмножини співпадаючих елементарних запитів, які виконуються в першу чергу з метою зменшення сумарного числа елементарних запитів.

Нехай запити мультізапиту визначено такими параметрами: k – кількість елементарних запитів, що утворюють запити мультізапит $З_1$, $З_2$…$З_v$ елементарні запити утворюють *d* груп по *u* елементарних запитів в кожній групі, кожен запит $З_i$ (і від одного до v) містить дві групи елементарних запитів з номерами: $З_{i...vu+u}$.

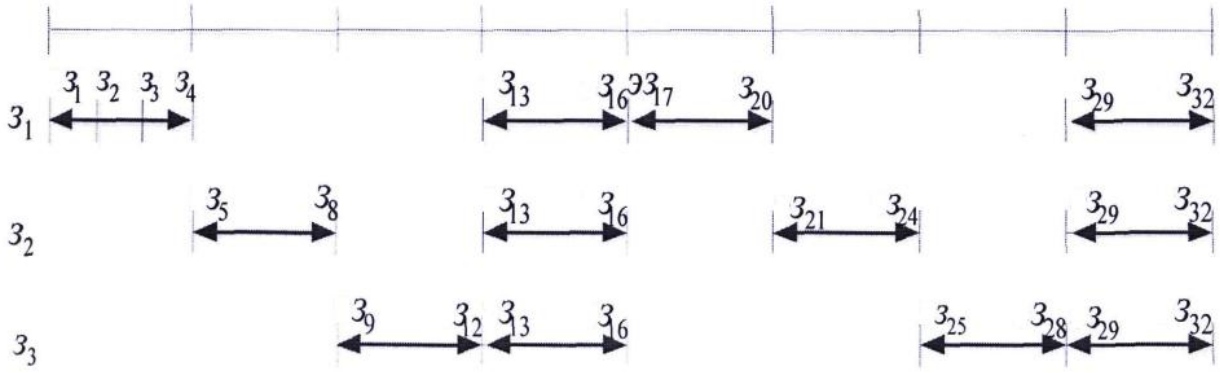

Рисунок 4.1 – Мультізапит при значеннях параметрів k=32, d=2, u=4, v=3

Нехай параметри елементарних запитів $З_i$: необхідний час т; обробки елементарних запитів в рядку таблиці відповідає статичній або лінійній залежностям і ймовірність успіху є постійною величиною р, = р, і = 1, ... Визначимо час виконання мультізапроса при спільному і неспільному виконанні запитів (далі для упорядкованих таблиць).

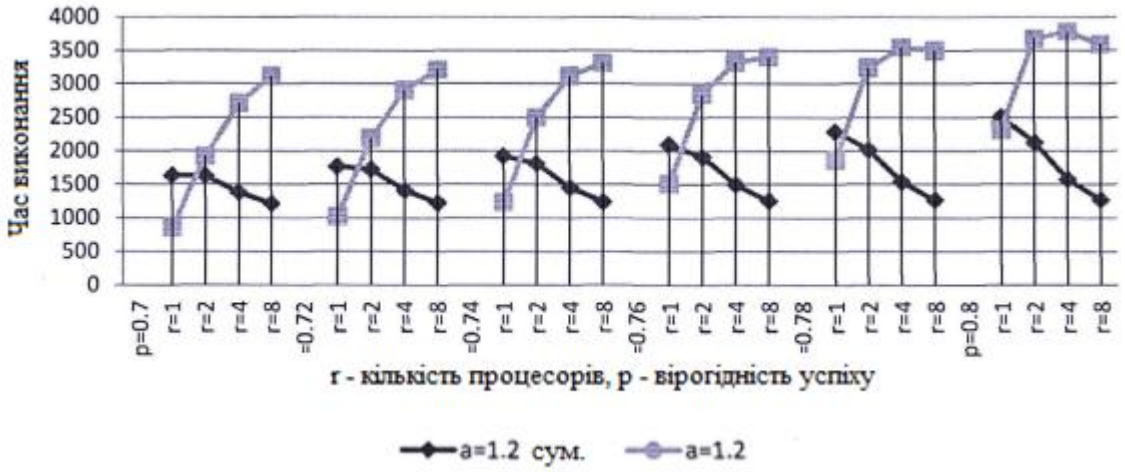

Рисунок 4.2 – Результати роботи

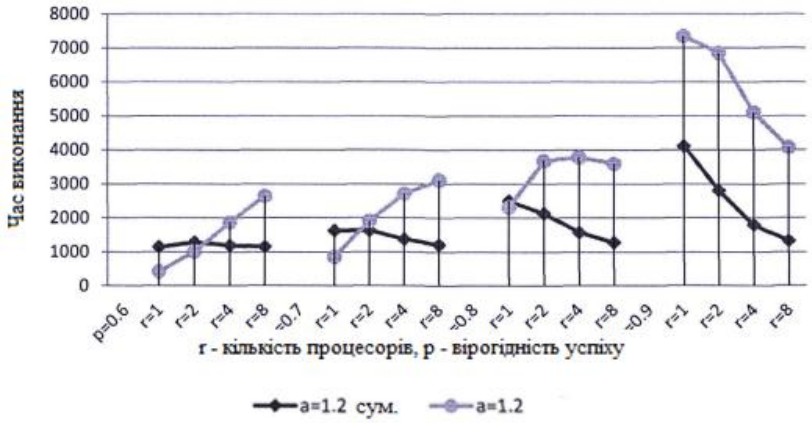

Рисунок 4.3 – Рисунок 4.2 – Результати роботи

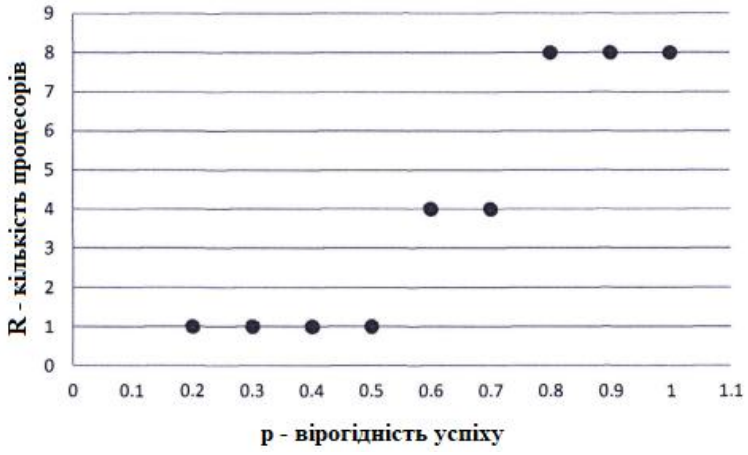

Рисунок 4.4 – Результати роботи

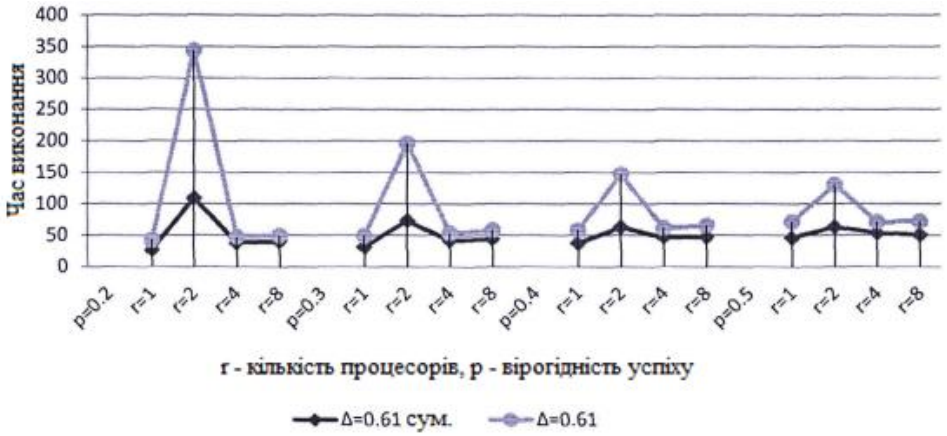

Рисунок 4.5 – Результати роботи

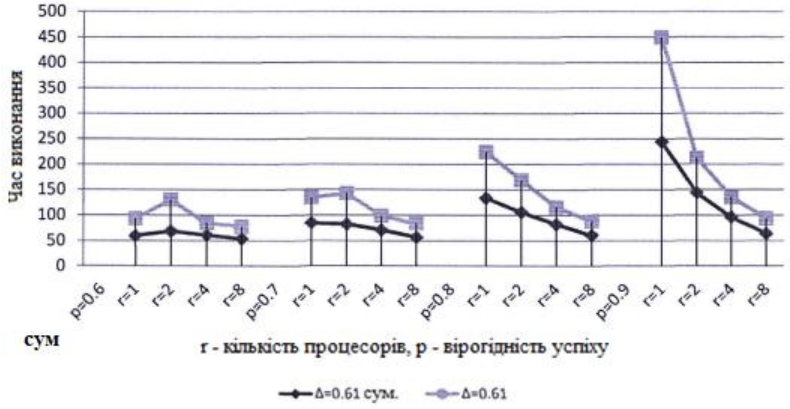

Рисунок 4.6 – Результати роботи

Рисунок 4.7 – Результати роботи

4.2 Розробка програмних засобів

Однією з захоплюючих перспектив, з точки зору паралельного програмування, є все зростаюче розмаїття багатопроцесорних систем. Виробники процесорів з гордістю говорять про можливість запропонувати недорогі і доступні процесори з чотирма або вісьмома ядрами для звичайних персональних комп'ютерів, і процесори з десятками ядер для високопродуктивних серверів. Середні сучасні робочі станції або потужні ноутбуки часто забезпечуються потужними графічними процесорами (GPU), що володіють можливістю обслуговувати сотні потоків виконання. На додаток до двох звичайних прийомів паралельного програмування з'явилася інфраструктура, яка надається як послуга (Infrastructure-as-a-Service, IaaS), з щотижня знижується вартістю використання, в особі якої ви в одну мить можете отримати хмарне оточення з декількома тисячами ядер. Зростання продуктивності, одержуваний за рахунок застосування прийомів паралельного програмування, досить великий, щоб від нього легко було відмовитися. Додатки, в основному займаються операціями введення / виводу, можуть отримати величезні переваги за рахунок перенесення цих операцій в окремі потоки, і виконання їх асинхронно і паралельно, завдяки чому забезпечується більш висока чуйність і масштабованість. Переважно обчислювальні додатки, що реалізують паралельні алгоритми, здатні збільшувати свою продуктивність на порядок, за рахунок використання всіх доступних процесорів, або на два порядки, за рахунок використання ядер графічного процесора. Пізніше буде показано, як можна в 130 разів прискорити просту реалізацію множення матриць лише за рахунок зміни декількох рядків коду, що забезпечують виконання на графічному процесорі.

Існує безліч причин розбиття додатків на кілька потоків виконання. Застосування асинхронних операцій введення / виводу може підвищити чуйність додатки. У більшості додатків з графічним інтерфейсом за все зміни в інтерфейсі відповідає єдиний потік; цей потік ніколи не повинен

блокуватися на тривалі періоди часу, щоб своєчасно реагувати на дії користувача. Розпаралелювання роботи між декількома потоками виконання дозволяє повніше використовувати системні ресурси. Сучасні системи, побудовані на багатоядерних процесорах і багатоядерних графічних процесорах, здатні на порядки збільшувати продуктивність простих обчислювальних алгоритмів за рахунок їх розпаралелювання. Одночасне виконання декількох операцій введення / виводу (наприклад, одночасне отримання цін з декількох сайтів туристичних агентств або зміна файлів в декількох розподілених веб-сховищах) може збільшити загальну пропускну здатність, тому що в подібних ситуаціях більшість часу витрачається на очікування завершення введення / виводу, а час, що звільнився може бути витрачено на виконання додаткових операцій введення / виводу або обробку результатів вже завершилися операцій.

Було поставлене завдання: обробка файлу в заежності від його розміру. Були створені 2 функції: перша створює файл розміром X байт і заповнює його рандомними числами. Друга перевіряє, на скільки обчислювачів ми можемо розділити цей файл. Далі вона ділить цей файл на N обчислювачів. Програма виводить час пошуку в файлі.

При виконанні функції створення файлу передається його розмір в функцію sozdaniefaila.

```
public class Program
{
    static void Main(string[] args)
    {
        IEnumerable<uint> counters = new List<uint>();

        Stopwatch timer = Stopwatch.StartNew();
        counters = PrimesInRange(100, 200000);
        timer.Stop();

        Console.WriteLine("Прошло: {0} мс., найдено {1} процессов",
            timer.ElapsedMilliseconds,
            counters.Count());
}
```

Приклад 4.1 – Фрагмент коду розроблених програмних засобів

```
    public static IEnumerable<uint> PrimesInRange(uint start, uint end)
    {
        List<uint> primes = new List<uint>();

        for (uint number = start; number < end; ++number)
        {
            if (IsPrime(number))
            {
                primes.Add(number);
            }
        }

        return primes;
    }

    private static bool IsPrime(uint number)
    {
        if (number == 2) return true;
        if (number % 2 == 0) return false;

        for (uint divisor = 3; divisor < number; divisor += 2)
        {
            if (number % divisor == 0)
                return false;
        }

        return true;
    }
}
```

Приклад 4.1 – Продовження

Розмір в байтах	Час виконання, мс	Обчислювачів
5368709120	677169000	1
2684354560	337370000	2
1342177280	167186000	*4*
1073741824	134196000	5
671088640	85925000	8
536870912	68376000	10
335544320	41837000	16
268435456	33591000	20
167772160	20914000	32
134217728	16625000	40
83886080	10370000	64
67108864	8356000	80

Рисунок 4.8 – Дані для перевірки роботи програмного забезпечення

Так як зчитування в програмі відбувається побайтово, то переводимо наш розмір в байти, отримуємо 5368709120 байт. Далі перевіряємо, на скільки частин ми його можемо розділити, щоб відправити обчислювальним вузлам. Далі програма обчислює виконання пошуку даних в файлі на кожному активному вузлі. Далі наведені результати для файлу розміром 5 Гб.

Розробка додатку виконувалася в середовищі Visual Studio 2015; Результати роботи програми. По цих даних можна створити графік залежності часу виконання від розміру даних.

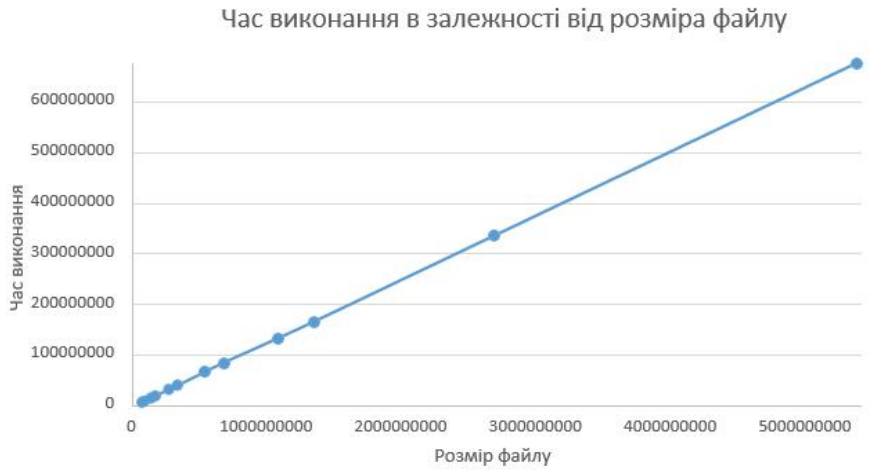

Рисунок 4.8 – Час виконання в залежності від розміру файлу

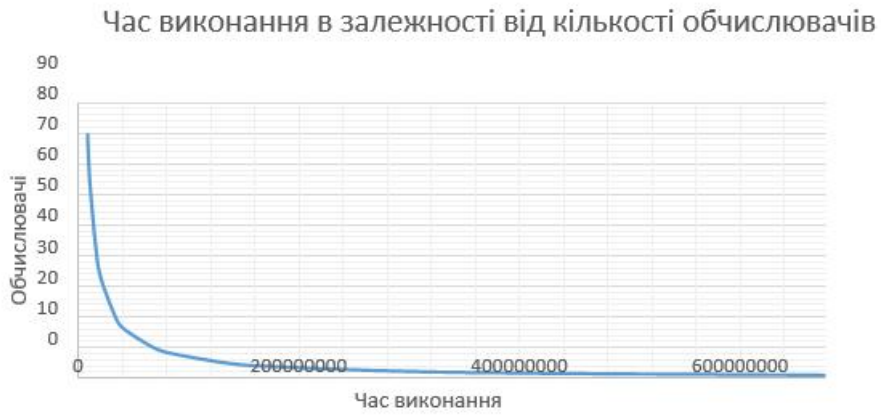

Рисунок 4.9 – Час виконання в залежності від кількості обчислювачів

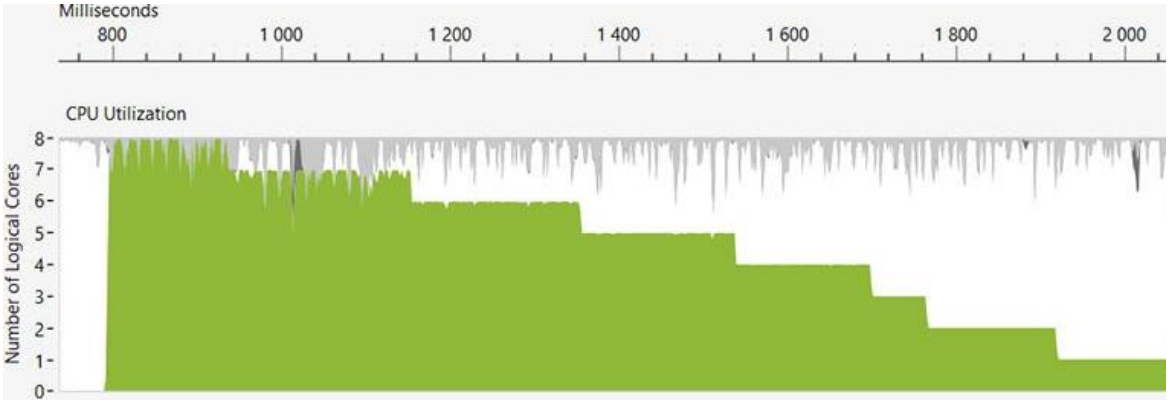

Рисунок 4.10 – Програмний інтерфейс

```
        public static IEnumerable<uint> PrimesInRange(uint start, uint end)
        {
            List<uint> primes = new List<uint>();
            uint range = end - start;

            uint numThreads = (uint)Environment.ProcessorCount;     uint chunk =
range / numThreads;

            Thread[] threads = new Thread[numThreads];

            for (uint i = 0; i < numThreads; ++i)
            {
                uint chunkStart = start + i * chunk;
                uint chunkEnd = chunkStart + chunk;

                threads[i] = new Thread(() =>
                {
                    for (uint number = chunkStart; number < chunkEnd; ++number)
                    {
                if (IsPrime(number))
                {
                    lock (primes)
                    {
                        primes.Add(number);
                    }
                }
                    }
                });
                threads[i].Start();
            }

            foreach (Thread thread in threads)
            {
                thread.Join();
            }

            return primes;
        }
```

Приклад 4.2 – Фрагмент коду розроблених програмних засобів

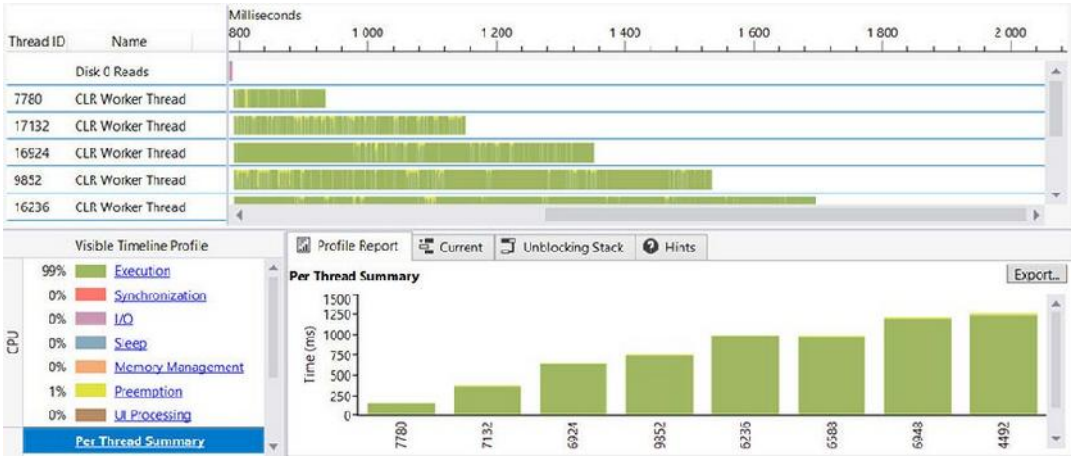

Рисунок 4.11 – Програмний інтерфейс

Як видно на рисунку 4.10 , загальне використання процесора спочатку підвищилося майже до 8 логічних ядер (100%), а потім поступово знижувалося. На рисунку 4.11 видно, що деякі потоки впоралися зі своєю часткою роботи значно раніше за інших. Потік 7780 виконувався менше 200 мсек, а потік 4492 більше 800 мсек. Насправді ця програма може працювати набагато швидше послідовною версії (хоча масштабованість і не буде лінійної), особливо якщо задіяти велику кількість ядер.

ВИСНОВКИ

Запропоновано і обґрунтовано метод оптимізації за часом виконання мультізапиту при зверненні до бази даних на основі упорядкування елементарних запитів. Розроблено метод забезпечення оптимізації багатопроцесорної обробки мультізапитів.

Доведено, що параметр ймовірності успіху при виконанні елементарного запиту є істотним параметром, що впливає як на вибір спільного і неспільного методу обробки мультізапиту, так і на визначення числа процесорів.

Запропоновано оптимальний алгоритм розподілу елементарних запитів на процесори. Знайдене оптимальне співвідношення для мінімального часу виконання мультізапроса для упорядкованих або невпорядкованих даних таблиць при спільній обробці процесорами об’єднаної безлічі елементарних запитів.

У середовищі Visual Studio розроблено програмні засоби, які використовують розроблені методи та алгоритми.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Дяченко В.О., Міхаль О.П., Політаєва А.І. Методи та засоби обробки запитіву комп’ютерних мережах // Сучасні напрями розвиткуінформаційно-комунікаційних технологій та засобів управління. Матеріали десятої міжнародної НТК. –Баку: ВА ЗС АР; Харків: НТУ «ХПІ»; Харків: ДП«ПДПРОНДІАВІАПРОМ»; Жиліна: УмЖ, 2020. – 9-10 квітня 2020. – С. 84.

2. Гольдштейн М.Л. Мультипроцессорная вычислительная система на базе транспьютерной идеологии // Алгоритмы и программные средства параллельных вычислений: Сб. науч. тр. / ИММ УрО РАН.Екатеринбург: УрО РАН, 1995.

3. Григорьев Ю.А. Разработка научных основ проектирования архитектуры распределенных систем обработки данных: Дис. д-ра техн. наук. МГТУ им. Н.Э. Баумана, 17.10.96 г.

4. Кузнецов С.Д. - «Базы данных: языки и модели», Москва, Бином, 2008, 720 с. Кузнецов С.Д. Логическая оптимизация запросов в реляционных СУБД // Программирование.

5. Кузнецов С.Д. Методы оптимизации выполнения запросов в реляционных СУБД // Сб. Итоги науки и техники. Вычислительные науки. - Т.1. -М: ВИНИТИ, 1989

6. Кузнецов С.Д. Основы баз данных. - М.: Изд-во "Интернет-университет информационных технологий - ИНТУИТ.ру", 2005.

7. Н.А. Гребенников, В.М. Постников //Организация баз данных, 2002 г. (Московский государственный технический университет им. Н.Э. Баумана).

8. Сиротюк В.О. Модели и методы синтеза оптимальных логических структур и базы метаданных репозитария распределенных баз данных в АСУ // Автоматика и телемеханика (М.). 1999. №1.

9. Соколинский Л.Б. Организация параллельного выполнения запросов в многопроцессорной машине баз данных с иерархической архитектурой // Программирование. 5 О.Соколинский Л.Б. Параллельные машины баз данных // Природа. Естественно-научный журнал Российской академии наук. -2001.

10. Воеводи В. В., Воеводил Вл. В. Параллельные вычисления. СПб.: БХВ- Петербург, 2004, – 599 с.

11. Дробнов С. Е. Обобщение закона Амдала для большого объема параллельных вычислений в GRID // Научный вестник МИРЭА. № 2 (9) 2010 г. С. 98.

12. Top500 [Электроный ресурс]. – http://www.top500.org/lists/2016/06/ (общащение 21 июня 2016)

13. Смагин В.А. Основы теории надёжности программного обеспечения – Санкт-Питербург,2009. – 335 с.

14. A.S. Cassidy and A.G. Andreou «Analytical Methods for the Design and Optimization of Chip-Multipocrssor Architectures» Proc. 43rd Ann. Conf. Information Sciences and Systems, 2009.

15. R.G Brown, «Maximizing Beowulf Performance» Proc. Fourth Ann. Linux Showacase and Conf., 2000.

16. Хорошевский В.Г. Архитектура вычислительных систем. – М.: МГТУ им. Н.Э. Баумана, 2005. – 512 с.

17. Парамонов И. Ю., Смагин В. А. Закон Амдала для иерархических вычислительных систем // Информационные системы и технологии. 2012.

№ 4. С. 20-23.

18. A.S. Cassidy, K. Yu, H. Zhou, and A.G. Andreou, «A High-Level Analytical».

19. Model for Application Specific CMP Design Exploration," Proc. Conf. Design Automation and Test in Europe (DATE), 2011.

20. M. Hill and M. Marty, "Amdahl's Law in the Multicore Era," Computer, vol.

41, no. 7, pp. 33-38, July 2008.

21. Олифер В., Олифер Н. Компьютерные сети. Принципы, технологии, протоколы: Учебник для вузов. 5-е изд. - СПб.: Питер, 2016. - 992 с.

22. Таненбаум Э. С. Архитектура компьютера:[пер. с англ.]. – Издательский дом» Питер», 2011. – 843 с.