УДК 683.519

ДОБРОВОЛЬСЬКИЙ О.Ю.
ПОДРУБАЙЛО О.О.

МЕТОДИ ОПТИМІЗАЦІЇ АЛГОРИТМІВ ІНДЕКСУВАННЯ ТАБЛИЦЬ В СХОВИЩАХ ДАНИХ В ОПЕРАТИВНІЙ ПАМ'ЯТІ

В статі було розглянуто методи оптимізації структури алгоритмів індексування в сховищах даних в оперативній пам'яті. Наведено три способи, що мають за мету за рахунок зменшення розмірів структури збільшити кількість інформації, що буде вміщена в кеш процесора, а відповідно зменшити кількість кеш промахів. Було виконано аналіз запропонованих методів.

In this article considered ways of optimization indexing algorithms structure for in memory data grid. Suggested three ways that have the goal of reduction the size of the structure to increase the amount of information that will be placed in processor cache, and accordingly reduce the amount of cache misses. Performed analyses of proposed methods.

Ключові слова: алгоритми, індекси в базах даних, оптимізації, IMDG, сховища даних в оперативній пам'яті

1. Вступ
2.

Сьогодні, технічним рішенням потрібно відповідати стрімкому збільшенню об'ємів пам'яті. Великі об'єми інформації потребують зовсім інших підходів до реалізації засобів їх обробки. Одним зі способів збільшити швидкість маніпуляцій інформацією є сховища даних в оперативній пам'яті. Звичайна задача пошуку користувача і перевірки паролю на правильність для багатомільйонної таблички втрачає свою тривіальність. Соціальну мережу Facebook, в першому кварталі 2017 року, відвідало понад два мільярди унікальних користувачів [1], що тільки підтверджує важливість швидкого маніпулювання інформацією.

Індексування таблиць завжди було, і наразі все ще є, одним з досить простих і відносно економних способів збільшити швидкість пошуку за певним ключем[2]. Проте, з появою баз і сховищ даних в оперативній пам'яті пріоритети для алгоритмів індексів суттєво змінюються. Це пов'язано перш за все з зміною специфіки самих баз.

2. Аналіз літературних даних та постановка задачі

Індекс – це структура бази даних, що була створена задля покращення швидкості пошуку даних. Дана структура має певний впорядкований вигляд і може бути створена на основі одного або декількох полів. Відповідно читання по цим полям буде набагато швидшим через відсутність потреби перебирати всі значення для пошуку результату. Сьогодні, зазвичай, використовуються такі структури алгоритмів як: Hash, B-tree, B*-tree.

Розглянемо детальніше кожну структуру:

Hash – алгоритм, в якому створюється масив певного розміру, де зберігаються пари чи списки пар даних по типу ключ – значення. Під час додавання елементу в цей масив, використовується обчислення хеш-функції від ключа, котра надалі буде використовуватись як ключ в початковому масиві, по якому ми будемо робити пошук елементів. Через те, що кількість хешів обмежена, то досить часто можна отримати колізію – ситуацію коли хеш-функція двох різних елементів поверне те саме значення. Аби позбавитись колізій в масиві пар, під час спроби доступу до інших елементів того самого ключа, відбувається додавання елемента в кінець списку. Таким чином, пошук елемента буде відбуватись від ключа, який отримано за допомогою хеш функції і до комірки де зберігається елемент.

B-tree, або збалансоване дерево – це структура, де дані організовуються за принципом дерева, а збалансованість відповідає за те, що довжина від рута дерева до його листя буде відрізнятись не більше ніж на одиницю. Дані в такому дереві вже відсортовані і знаходиться потрібний проміжок в котрий необхідно поставити наш ключ. Також є таке значення t, котре задає кількість ключів в вузлі від t-1 до 2t-1. Рут дерева – вершина з нульовою степеню входження. Лист дерева – вершина с

єдиним ребром, кінцева вершина, з якої більше немає куди йти.

Аби додати новий ключ до дерева, необхідно знайти потрібний проміжок і додати туди наш ключ. Якщо ж раптом кількість ключів стала більшою за максимально допустиму, тобто 2t-1, то ми розділяємо цей вузол на два підвузла.

B*-tree - збалансоване дерево з більшою кількістю нащадків в вузлі. Також порівняно з звичайним B-tree листки зберігають посилання на сусіда, що значно спрощує швидкість проходження по елементам. Проте, ця оптимізація швидкості збільшує витрати пам'яті. Аби здійснити додавання ключа, потрібно знайти потрібний вузол, і якщо він не заповнений, додати ключ. Якщо вузол заповнений, то потрібно розбити вузол, створити новий вузол і перемістити в нього половину ключів, а в верхній рівень записати новий вузол. Так потрібно продовжувати доти, доки дерево не збалансується. Якщо потрібно розбити рут, то створюється новий вузол, який від тепер буде новим рутом. Таким чином, в цьому алгоритмі додавання відбувається від листя до рута, а не навпаки, як в звичайному збалансованому дереві.

Алгоритми індексування в IMDG[3] мають інші пріоритети, порівняно з алгоритмами в класичних сховищах даних, а саме:

- В сховищах даних немає потреби мінімізувати кількість звертань до основної пам'яті, через її відсутність[5]. Відповідно, підхід до індексування даних потрібно змінити. В класичних базах даних, через досить високий час читання з жорсткого диску, на задній план відходять обчислення та інші операції, через це основним пріоритетом було мінімізувати їх. У випадку сховищ потреба в цьому набагато менша.
- Розміри кеша процесора, порівняно з оперативною пам'яттю, є набагато меншими, а швидкість читання з неї — набагато більшою а ніж оперативна пам'ять. Тому, через досить велику обмеженість в розмірах, одною з першочергових проблем є оптимізації інформації, що буде збережена в кеші процесора[4]. Варто також зазначити, що в більшості випадків розмістити всю структуру в кеші процесора не вийде, тому потрібно мінімізувати кількість кеш промахів за рахунок збереження інформації, що запитується найчастіше.
- Важливим є також те, що сховища розподілені і індекси також мають бути такими. Тому потрібно будувати індекси таким чином, аби зробивши запит в сховище, не потрібно було отримувати всі дані з індексу, а навпаки відправляти лише шукане значення і отримувати лише результат без зайвих втрат часу на очікування. Таким чином, дуже важливим є узгодженість цих індексів між собою.
- Нормалізація даних і структури так, щоб розміри були найменшими[6].
- Питання паралелізму, який повинен бути передбаченим, аби максимізувати кількість операцій, що будуть виконуватись одночасно. Це перш за все дасть змогу зберегти час, а по друге, через те, що час запиту в кеш процесора є дуже малим, суттєво зменшить простої процесора. Також варто пам'ятати і про те, що індекс має підтримувати можливість паралельного обчислення декількох запитів.

Таким чином, існуючі алгоритми індексування в сховищах даних в оперативній пам'яті потребують переосмислення і доопрацювання.

Через наведені вище причини, потенційним місцем для оптимізації є кеш процесора що має будову що зображена на рис 1. Кеш процесора складається з декількох рівнів, що розташовані відповідно ближче до кеша, і далі від оперативної пам'яті. Основною превагою кеша процесора, на відміну від оперативної пам'яті є швидкість. Порівняння швидкостей зображено на табл. 1. Головним недоліком є дуже малий об'єм пам'яті, що залежить від рівня кеша процесора. Для найбільшого рівня розміри можуть досягати всього лише декількох десятків мегабайт пам'яті, що ділиться між усіма процесорами. Коли ж перший рівень, що знаходиться найближче до процесора може вмістити в собі не більше декількох десятків, чи як максимум сотень кілобайт, що в наш час навряд чи можна назвати великою кількістю простору. Так само як відрізняються розміри кеша від розмірів оперативної пам'яті, відрізняється і швидкість. Відповідно, замість одного запиту до оперативної пам'яті, можна зробити 150 запитів у перший рівень кеша, або 15 запитів до другого рівня кеша процесора. Таким чином, якщо запит, що потребує 10 порівнянь з значенням поля в оперативній пам'яті, буде читати інформацію не з оперативної пам'яті, а з кеша першого рівня, дасть змогу виконати запит у 1500 разів швидше. Така різниця вражає. Саме тому мінімізація кеш промахів є потенційним місцем для оптимізації існуючих алгоритмів.

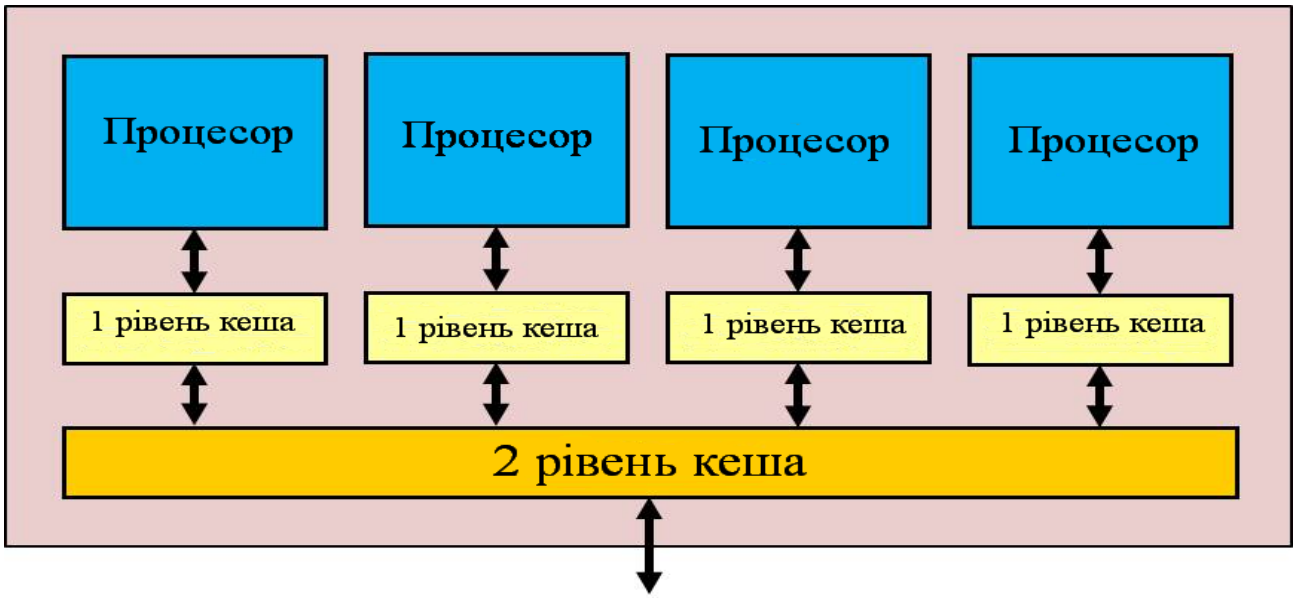

Рис. 1. Будова кеша процесора

Табл.1. Порівняння швидкостей доступу до різних рівнів пам'яті

Рівень	Час доступу
Перший рівень процесора	0,7 нс
Другий рівень процесора	11 нс
Оперативна пам'ять	170 нс

3. Мета дослідження

Метою даної роботи є дослідження можливості оптимізації існуючих алгоритмів індексування розподілених сховищ даних в оперативній пам'яті.

Для досягнення даної мети вирішувалась наступна задача

- Визначення потенційних місць для оптимізації існуючих індексів в сховищах даних в оперативній пам'яті.

4. Методи оптимізації існуючих алгоритмів

В даному розділі будуть розглянуті методи, що передбачають перш за все оптимізацію структури, за рахунок зменшення кеш промахів.

4.1 Структура даних

Методи, що будуть розглянуті в даному розділі, передбачають, що структура даних складається з вузлів. Будемо вважати, що вузол – певна кількість ключових полів або вказівників на інший вузол. Вважаємо, що ключові поля створюють інформаційну частину вузла, а вказівники потрібні для підтримки структури даних. Відношення між вузлами можуть підтримуватись за допомогою вказівників. Цілю всіх розглянутих методів є зменшення кількості кеш промахів, що з'являються під час виконання алгоритмів пошуку.

Методи оптимізації вузлів досягають поліпшення, за рахунок використання вузлом кеш блоків. Зрозуміло, що чим менше розмір вузла, тим зручніше це з точки зору використання кеш блоку.

Розглянемо можливі методи оптимізації використання кеш блоку вузлом.

4.2. Метод видалення ключових блоків

Одним з методів оптимізації є видалення ключових блоків. Ключові поля, що не використовуються у функції пошуку і лише займають місце в кеш блоці і можуть бути видалені з вузла шляхом, наприклад, винесення їх в окрему структуру даних.

Також, можна оптимізувати кеш промахи за допомогою перегрупування полів. Не завжди можливо винесення не використовуємих при пошуці полів в окрему структуру. В цьому випадку, потрібно перегрупувати поля таким чином, аби поля, що використовуються в близькі моменти були б поряд в вузлі. Ця зміна

впливає не лише на ключові поля, але й на вказівники. Для кращого розуміння розглянемо приклад. Приклад. Є табличка Products(year int, name char(20), price int). Таким чином, один запис займає 36 байт. Нехай у нас є функція для якої потрібно знати лише ціну продукту. Відповідно ім'я та рік на пошук ніяким чином впливати не будуть. Відповідно ми можемо винести їх в окрему структуру. Таким чином, при розмірі одного кеш блока в 256 байт, при початковій конфігурації в кеші можливо буде вмістити 7 записів. Тоді ж як після оптимізації кеш блок буде вміщати 32 записи, що майже у 5 разів більше. Відповідно кількість кеш промахів зменшиться приблизно у 5 разів.

4.3. Метод компресії ключових полів

Розглянемо також такий вид оптимізації як компресія ключових полів. Компресія дозволяє зменшити розмір ключових полів і відповідно кількість кеш блоків, що займає вузол або розмістити більше вузлів в одному блоці. Часто, ключові поля є невід'ємними цілими числами, причому розташовані в вузлі в порядку зростання. В такому випадку, можна використати один з методів компресії зростаючої послідовності невід'ємних цілих чисел.

Даний метод компресії заснований на тому, що замість зберігання самого числа, зберігається різниця між ним і попереднім числом, що є також ймовірніше за все додатнім не великим числом. Після цього, число кодується одним з методів кодування. Вагомим недоліком цього методу є те, що для того, щоб дістатись до n-того елементу, потрібно кожного разу розпочинати з самого початку. Проілюструємо даний метод на прикладі. Приклад. Є табличка з студентами Students(name char(20), points int) де points загальна кількість балів набрана студентом за семестр. Функція, що буде використовувати табличку, буде шукати усіх студентів, чия кількість балів буде вищою за n. Таким чином маючи вже посортований по спаданню список ми, знаючи перший елемент, записуємо різниці і кодуємо іх. Функція у будь-якому разі буде змушена починати з першого студента, так як в нього буде найвищий бал, а відповідно він повинен бути включений у будь-який список. Компресія наступних полів дасть змогу зменшити об'єми використаної одним полем пам'яті а відповідно і зменшити кількість кеш промахів.

Іншим методом компресії, котрий використовується в ситуаціях, коли ключові поля мають великий розмір і зміну довжину, є віддалення від кожного ключа префікса фіксованої довжини. Причиною використання цього є те, що дуже часто замість ключа можна використати префікс, наприклад, для порівняння стрічок, можна порівнювати префікси, і лише, якщо вони співпадають звертатись до самих стрічок. Приклад. Розглянемо табличку students(name char(20)). Функція буде знаходити чи існує студент з заданим ім'ям в списку. Нехай табличка заповнена такими даними ('Ivanov', 'Petrov', 'Sidorov'), тоді після оптимізації ми отримаємо такий набір значень ('Iv', 'Pe', 'Si', 'Ivanov', 'Petrov', 'Sidorov'). Таким чином для пошуку потрібно буде превіряти лише перших дві літери. Що пришвидшить операцію порівняння і змешнить розміри одного поля, що буде збережено в кеш. Якщо ж префікси будуть однакові, то тоді потрібно буде звертатись до самого значення.

4.4. Видалення вказівників

Зазвичай поля, що є вказівниками не використовуються в функціях, тому зменшивши кількість вказівників можна зменшити кількість кеш блоків, що займає вузол. Зауважимо, що вказівники потрібні лише для підтримки зв'язків в структурі, тому їх можна замінити на інший спосіб підтримки цих зв'язків, зокрема їх обчислення. Як правило, обчислювання зв'язків засноване на тому що при присутності в структурі певних регулярностей, адресу вузла нащадка можна вирахувати з батьківського вузла і його порядкового номера.

Приклад. Уявимо, що у нас є k-розмірне дерево. Тобто вузол матиме вигляд(key0..keyk, ptr0..ptrk). Нехай всі вузли нащадки знаходяться в пам'яті послідовно тоді їх можна визначити за наступною формулою ptri = ptr0 + i * sizeof(Node), таким чином, можливо прибрати з структури (k – 1) * sizeof(ptr) = 4(k-1) байт. Що може суттєво збільшити кількість інформації в кеші. Зрозуміло, що такі зміни потребують додаткових зусиль для модифікації цього дерева, проте якщо читання з нього відбувається набагато частіше, а ніж модифікація, то оптимізація може бути корисною.

5. Аналіз запропонованих методів оптимізації та обговорення їх доцільності

Кожен з наведених вище методів є повністю самостійним і залежить лише від доцільності їх використання. Жоден з методів не може стати універсальним способом збільшити продуктивність існуючого алгоритму індексування. Проте, дані методи можуть дати змогу за певних умов досить непоганих результатів.

Метод видалення ключових полів є наразі одним з найуніверсальніших методів, через те, що він може бути застосованим в більшості випадків. Винесення з індексної структури інформації, що ніяким чином не впливає на пошук, а лише зберігається там, так як є частиною вузла, майже завжди буде доцільним і правильним рішенням. Таким чином, залежно від кількості інформації, що буде винесена за межі структури вузла, пропорційно збільшиться розмір інформації, що безпосередньо потрібна для пошуку, і буде збережена в кеші процесора. Але через те, що новоутворена структура, де будуть зберігатись решта полів, також повинна десь зберігатись і дублювати зв'язки початкової структури іноді отриманий приріст у швидкості не буде здатний відшкодувати втрати пам'яті на підтримку двох структур одночасно.

Метод компресії ключових полів є найбільш ситуативним, бо поля не завжди можуть бути посортованими або бути додатними числами. Саме ця умова вводить найбільші обмеження на застосування даного способу. Також список, що задовольняє попередню умову, може бути не нормованим на стільки, що запис різниць між полями не дасть відчутної різниці. Використання різниць змушує завжди перебирати починаючи з самого початку, що також не завжди може бути доцільним. Так само варто враховувати і той факт, що декодування полів, потребує часу, хоча і меншого ані ж читання з оперативної пам'яті, проте, в разом з рештою обмежень може зменшити користь від такого методу до мінімуму. Для специфічних ситуації, коли інформація повинна бути відсортована і перший елемент завжди потрібно перевіряти, цей спосіб може бути досить доречним.

Метод видалення вказівників, не дає змогу отримати велику перевагу у зменшені розмірів вузла, так само як і вказівники можуть бути згенеровані таким чином, що їх обчислення буде неможливим, чи буде займати занадто багато часу. Проте, в звичайних структурах, цей метод може зменшити розміри вузла, без вагомих втрат, порівняно з іншими методами і є відносно безпечним способом отримати перевагу.

Загалом, жоден з методів не можна назвати універсальним, але вони можуть бути застосованими в більшості випадків і є відносно безпечними методами оптимізації, які не мають потенційних проблем, окрім доцільності їх використання.

6. Висновок

Проведено дослідження потенційних методів поліпшення існуючих алгоритмів індексування таблиць в сховищах даних в оперативній пам'яті. Було запропоновано три методи оптимізації структури, що дають змогу збільшити кількість інформації, що буде збережена в кеші процесора. Кожен з розглянутих методів є самодостатнім, але потребує певних умов, для доцільного використання.

Таким чином, дослідження показало, що існуючі алгоритми можливо допрацювати таким чином, аби швидкість їх використання була більшою. Проте, кожен з наведених методів потребує подальшого доопрацювання, аби використання нового методу, якщо і не давало приросту за будь-якого випадку, то хоча б не погіршувало швидкість вже існуючої реалізації.

Список посилань

1. Інтернет ресурс. Number of monthly active facebook users worldwide. https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/
2. G. Powell, Beginning Database Design. 2005.
3. Інтернет ресурс. In-memory-data-grid. https://habrahabr.ru/post/126973/
4. Д. А. Шапоренков, Эффективные методы индексирования данных и выполнения запросов в системах управления базами данных в оперативной памяти. 2006.
5. Інтернет ресурс. In-memory-data-grid. https://hazelcast.com/use-cases/imdg/
6. М. Р. Когаловский, Энциклопедия технологий баз данных, 2002.