Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра мережевих та інтернет технологій

ЗАТВЕРДЖУЮ
завідувач кафедри
мережевих та інтернет технологій
_____ Ю.В. Кравченко
«_____»_____2021 року

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

галузі знань 17 «Електроніка та телекомунікації»
за спеціальністю 172 «Телекомунікації та радіотехніка»

на тему:

ПРОЕКТУВАННЯ ПЕРСИСТЕНТНОЇ БАЗИ ДАНИХ ТИПУ КЛЮЧ-ЗНАЧЕННЯ

Виконав: студент групи МІТ - 41

Палієнко Микита Олександрович _____
(прізвище ім'я по-батькові) (підпис)

Керівник: професор кафедри мережевих та інтернет технологій

д.т.н., Кравченко Ю.В _____
(посада, прізвище ім'я по-батькові) (підпис)

Київ 2021

Міністерство освіти і науки України
Київський національний університет імені Тараса Шевченка

Факультет інформаційних технологій
Кафедра мережевих та інтернет технологій

ЗАТВЕРДЖУЮ
завідувач кафедри
мережевих та інтернет технологій

_____ Ю.В. Кравченко

«_____»_____2021 року

ЗАВДАННЯ
НА ДИПЛОМНУ РОБОТУ

Здобувачу вищої освіти Палієнко Микиті Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи:

«Проектування персистентної бази даних типу ключ-значення»

затверджена на засіданні кафедри МІТ «11» грудня 2020 р. протокол № 6

2. Термін здачі закінченої роботи «30» травня 2021р

3. Вихідні дані до проекту (роботи)

Застосунок для перегляду статистики та порівняння робіт студентів

Програмний продукт

4. Зміст пояснювальної записки (перелік питань, що їх потрібно розробити, обсяг – 35-50 стор.)

1. Дослідження методів та алгоритмів проєктування бази даних. Постановка задачі.
2. Постановка задачі.
3. Оцінка результату побудови персистентної бази даних

5. Перелік графічного матеріалу 8-12 слайдів

Вибір інструментів для реалізації бази даних.

Проектування компонентів.

Реалізація програмного продукту.

Дослідження продуктивності БД.

Висновки

Дата видачі завдання

Керівник роботи д.т.н., професор кафедри МІТ Ю.В Кравченко.
(підпис) (посада, прізвище, ім'я, по батькові)

Завдання прийняв до виконання
(підпис) (прізвище, ім'я, по батькові)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ РОБОТИ

Номер	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Підготовчий	29.01.2021	
2	Розділ 1	01.03.2021	
3	Розділ 2	01.04.2021	
4	Розділ 3	01.05.2021	
5	Доповідь та слайди	27.05.2021	
6	Пояснювальна записка	30.05.2021	

Здобувач вищої освіти _____
(підпис) (прізвище, ім'я, по батькові)

Керівник _____
(підпис) (прізвище, ім'я, по батькові)

РЕФЕРАТ

Пояснювальна записка: 61с., 34 рис., 4 табл., 2 додатки, 15 джерел.

Актуальність роботи: Актуальність теми дослідження обумовлена новизною ракурсу вивчення внутрішньої реалізації структури бази даних типу ключ-значення.

Зв'язок роботи з науковими програмами, планами, темами: Тема кваліфікаційної роботи відповідає науковому напряму кафедри Факультету Інформаційних технологій Київського національного університету імені Тараса Шевченка. В даній роботі було використано знання та навички отримані під час навчання таким дисциплінам як: Мережеві технології, Обчислювальні методи, додаткові розділи проектування програмного забезпечення.

Мета і задачі дослідження: відобразити результати проведених наукових досліджень, провести математичні розрахунки, обґрунтувати, прийняті при виконанні роботи структурні, системні та конструкторські рішення. Пояснити принципи роботи проекту та його окремих компонентів і програмних модулів.

Об’єкт дослідження: Об’єктом дослідження є дослідження зображень та методи їх класифікації, оптимізаційні техніки.

Предмет дослідження: Предметом дослідження є принципи та підходи до імплементації бази даних типу ключ-значення.

Методи дослідження: Під час написання програмного модулю використовувалися алгоритми оптимізації, які засновані на роботі з відповідними структурами даних. Дослідження також включало аналіз першоджерел у вигляді офіційної документації, статей.

Практичне значення одержаних результатів: Одержані результати можуть бути використані у системах, де необхідно зберігати великі обсяги даних із швидким доступом до них.

Ключові слова: ІНФОРМАЦІЙНА СИСТЕМА, АВТОМАТИЗАЦІЯ, ОЦІНЮВАННЯ ЛАБОРАТОРНИХ РОБІТ, ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ, БЕЗПЕРЕРВНА ІНТЕГРАЦІЯ, БЕЗПЕРЕРВНЕ РОЗГОРТАННЯ, БАЗА ДАНИХ, РЕЛЯЦІЙНА БАЗА ДАНИХ, СИСТЕМА АВТОМАТИЗАЦІЇ

ЗМІСТ

Стор.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CI – Continuous Integration (Безперервна інтеграція).

CD – Continuous Deployment (Безперевне розгортання).

ІС – інформаційна система.

ОС – операційна система.

БД – база даних.

СУБД – система управління базами даними.

API – Application Programing Interface (Прикладний програмний інтерфейс).

JDK – Java Development Kit (набір інструментів Java для розробки).

JMH – Java Microbenchmark Harness (інструменти вимірювання продуктивності).

VM – віртуальна машина.

JVM - Java Virtual Machine (віртуальна машина виконання коду Java).

ВСТУП

У світі, де програмне забезпечення працює на комп’ютерах з більш швидкими процесорами та великою пам’яттю, існує очевидна потреба у більш швидких базах даних. Бази даних основної пам'яті (де записи знаходяться переважно в пам'яті) є можливим рішенням. Ці бази даних набагато швидші, ніж їх аналоги на основі дисків, оскільки їх не уповільнює необхідність частих операцій вводу-виводу. Більше того, враховуючи досягнення напівпровідникової технології, бази даних основної пам'яті стали можливим рішенням для зберігання; пам'ять набагато дешевша і набагато більша. У цій роботі обговорено реалізацію послідовного основного сховища пам'яті ключ-значення на основі Redis. Ця реалізація використовує SQLite; Концепція курсора B+-Tree, що дозволяє операціям put і get запускатись у амортизованому постійному часі, коли курсор знаходиться поблизу потрібного місця. Показано, що набір послідовних операцій введення або отримання цього модифікованого B+-Tree виконується за лінійний час і що часткове сортування послідовності операцій за ключем збільшує продуктивність. Також показано, що це покращення продуктивності при частково відсортованих робочих навантаженнях поширюється на сховище формату ключ-значення. Також коротко представено концепцію курсору сховища ключ-значення для подальшого покращення продуктивності сховища формату ключ-значення за відсортованих та частково відсортованих робочих навантажень.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ МЕТОДІВ ТА АЛГОРИТМІВ ПРОЄКТУВАННЯ БАЗИ ДАНИХ.

1.1 Бази даних основної пам'яті

Бази даних основної пам'яті - це клас систем зберігання, де записи розташовані повністю в пам'яті, на відміну від традиційних систем баз даних, де записи розміщуються на зовнішніх запам'ятовуючих пристроях і завантажуються в пам'ять за потреби. Як і можна було очікувати, ці традиційні системи, як правило, працюють повільніше, ніж їх аналоги з основної пам'яті, завдяки введенню / виводу файлів. Раніше ці традиційні системи баз даних були необхідними, оскільки товарне обладнання не мало пам'яті, достатньо великої, щоб вмістити всі дані в базі даних. Однак набагато дешевші ціни на пам'ять означають, що бази даних основної пам'яті є можливим рішенням для зберігання. Про це свідчить поява різних рішень та реалізацій баз даних основної пам'яті, таких як: H-Store, Redis, VoltDB, та MemSQL. Оскільки файл вводу-виводу вже не є горловиною пляшки, бази даних основної пам'яті мають більше можливостей для поліпшення продуктивності. Крім того, їм доводиться робити різні варіанти дизайну щодо індексації, паралельності, кешування та розділення з традиційних баз даних, оскільки введення-виведення файлів більше не є основною горловиною пляшки. Більше того, оскільки ці бази даних не засновані на файлах, вони також повинні вирішувати питання довговічності та реєстрац

1.2 Індексація бази даних та дерева B+

Ключовою особливістю баз даних є рівень індексації або компонент. Рівень індексації забезпечує швидке отримання / доступ до будь-якого запису в базі даних. Це зменшує час доступу до будь-якого запису у базі даних на базі файлів із N сторінками від O (N) до O (logdN) (де d - константа), у випадку B + -Trees - деревного індексу. Також можливі індекси на основі хеш-показників. Деревові індекси підтримують порядок між своїми записами. В результаті вони можуть підтримувати такі запити, як: "Які студенти мають середній бал від 8 до 12?". Бази даних мають принаймні один первинний індекс і можуть мати кілька вторинних індексів для кожного набору записів. Це дозволяє швидко виконувати операції з різними полями записів. Наш сховище ключ-значення можна розглядати як основний індекс набору записів.

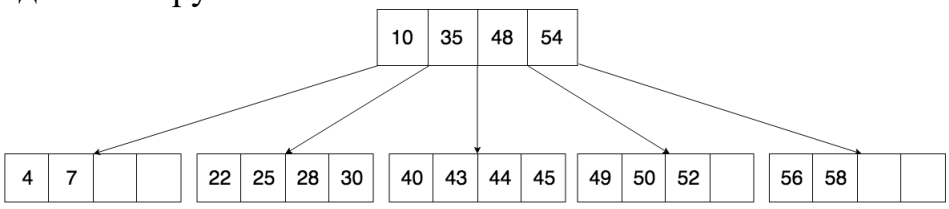


Рисунок 1: Приклад B-дерева з 19 ключами. d = 2.

B + -Дерева є варіантом B-Trees: збалансована структура даних дерева пошуку. B-Trees були вперше представлені як метод індексації записів, що зберігаються у файлах на вторинному сховищі. Малюнок 1 показує приклад B-Tree. Кожен вузол у B-Tree має порядок d, який визначає його ємність, тобто скільки ключів він може вмістити. Кожен вузол, крім кореневого, повинен мати від d до 2d ключів. Корінь може мати від 0 до 2d ключів. Покажчик між двома ключми a і b містить ключі, значення яких більші за a і менше b.

B + -дерева відрізняються від аналогів B-Tree тим, що їх вузли поділяються на послідовність та набір індексів. Малюнок 2 показує просте B + -Tree з одним внутрішнім вузлом (коренем) та п'ятьма листовими вузлами, що містять ключі, збережені у дереві.

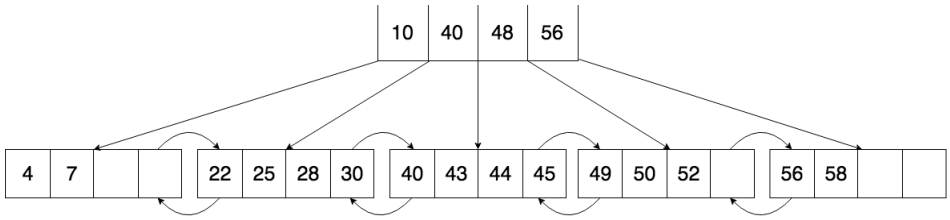


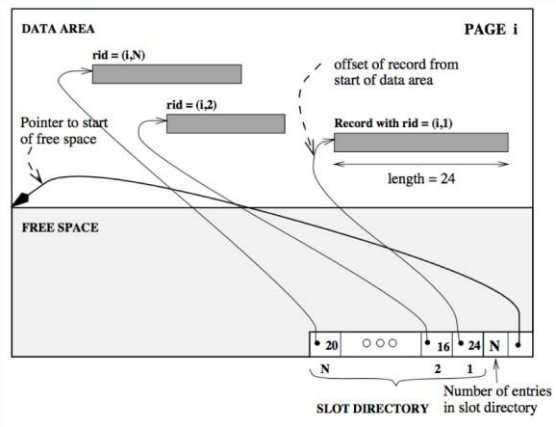
Рисунок 2: Приклад B + - Tree з 15 ключми. d = 2.

Набір послідовностей складається з листя B-дерева. Індексний набір складається з внутрішнього (нелистового) вузлів. Отже, ключі запису знаходяться в листках, тоді як внутрішній вузол містить копії ключів, які направляють пошук до листового вузла, що містить бажаний запис. Ці копії є результатом розділення вузлів. Розбиття вузла відбувається, коли ми намагаємось вставити новий ключ у повний вузол, щоб звільнити місце для нового ключа. Після розбиття вузла копія першого ключа в новоствореному другому вузлі та вказівник на другий вузол пересуваються в батьківський вузол першого вузла. Цей процес триває, якщо батьківський вузол заповнений. Крім того, традиційні B + -дерева мають поперечні зв'язки між листям. Це означає, що набір послідовностей утворює (подвійно) зв’язаний список. Обмеження ключів / записів на листках призводить до простішого видалення та забезпечує швидкий послідовний доступ до ключів. Отримання наступного ключа в послідовності поточних ключів завжди займає постійний час.

1.3 Обробка ключів змінної довжини

B + -Дерева є чудовою структурою даних для вставки, оновлення та пошуку записів. До будь-якого запису в B + -Tree, що містить велику кількість записів,

можна отримати доступ за допомогою декількох вузлів. Дійсно, кількість звернень до вузлів, задіяних в операції, є вигідною моделлю витрат при аналізі продуктивності баз даних на основі файлів і навіть баз даних основної пам'яті. Однак варто врахувати вартість порівняння рядків у B + -Trees за наявності ключів змінної довжини, які могли б поділяти довгі загальні префікси. Це пов’язано з тим, що, крім доступу до вузла, витрачається досить багато часу на пошук відповідного ключа у внутрішньому вузлі (для керівництва пошуком) або у листовому вузлі (для отримання потрібного запису) під час пошуку запису.

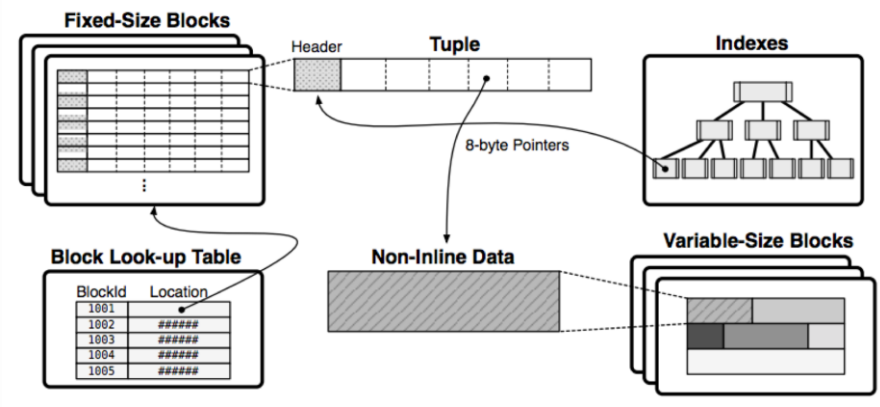


Малюнок 3: Приклад B + -Tree з d = 2.

Більшість реалізацій B + -Tree обробляють ключі змінної довжини, додаючи додатковий рівень опосередкованості. Сторінка / вузол, що містить ключі змінної довжини у файловому B + -Tree, виглядатиме приблизно так, як на малюнку 3. VoltDB (основна база пам’яті) також додає рівень непрямості при зберіганні довгих полів записів. Малюнок 4 показує, як записи зберігаються на рівні зберігання VoltDB. Тут кортежі записів зберігаються у блоках фіксованого розміру. Якщо поле кортежу перевищує 8 байт, воно зберігається в блоці змінного розміру; адреса пам'яті елемента потім зберігається в місці поля в кортежі. В обох представлених випадках існує рівень опосередкованості, пов’язаний з пошуком потрібних ключів у вузлі. Окрім того, що це менш кешоване кеш-пам’яті, є додаткова складність у

підтримці вільної області на сторінці вузла. Більше того, не виключено, що деякий простір може бути витрачений даремно через фрагментацію на сторінці даних.

В + -Дерева також можуть зіткнутися з проблемами з ключами з довгими префіксами. Розглянемо сервер, який періодично обробляє запити, та періодично зберігає у базі даних запитувані ресурси та кількість скільки унікальних клієнтів запитували цей ресурс.

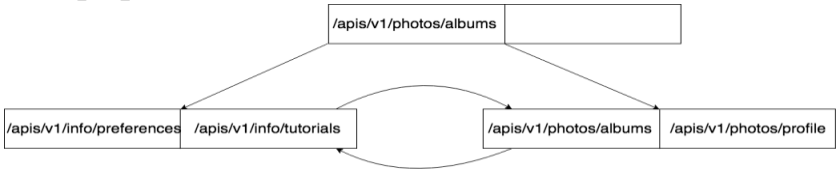


Малюнок 4: Макет сховища VoltDB.

URL-адреса ресурсу	Унікальні клієнти
api / photos	753
api / profile	486
api / info	1000
api / preferences	1000

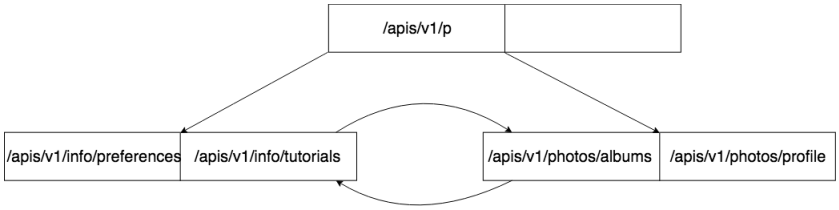
Таблиця 1: Приклад URL-адрес ресурсів HTTP та унікальної кількості клієнтів.

Приклад таблиці бази даних наведено в таблиці 2.3.1. Відповідний первинний індекс B + -Древа на URL-адресах ресурсів буде виглядати як B + -Tree, показаний на малюнку 5 (відображаються лише ключі). Цей простий приклад наочно демонструє, як B + -дерева обробляють довгі ключі, які, можливо, мають довгі загальні префікси. Простір витрачається даремно, економлячи копії довгих ключів і зниження продуктивності. Це пов'язано з тим, що під час пошуку будь-якого ключа в дереві потрібно проводити більше порівнянь символів, щоб визначити лексикографічний зв'язок між двома рядками, які мають спільний довгий загальний префікс.



Малюнок 5: Індекс дерева B + -Таблиця 2.3.1 (відображаються лише ключі).

Для простий приклад у табл 2.3.1, ці питання не викликають занепокоєння. Однак можна було б уявити набагато більшу таблицю, де є мільйони рядків, де більшість ключів мають спільні довгі префікси. Ці проблеми були б нестерпними. Навіть якщо взяти префікс B + -Tree - де замість того, щоб зберігати копії ключів у внутрішніх вузлах, використовується найменший можливий роздільник - простір все одно можна витратити, якщо там загальні префікси довгі. На малюнку 6 можна побачити, що корінь префікса B + -Tree індексу таблиці 2.3.1 містить ключ. Якщо найбільший ключ ліворуч від кореня та найменший ключ праворуч від кореня мають спільний довший загальний префікс, тоді розділювач повинен бути довшим.



Малюнок 6: Префікс B + - індекс дерева у таблиці 2.3.1 (відображаються лише ключі).

1.4 Рішення

Враховуючи вищезазначені проблеми, наша реалізація на структурі даних Redis. Це робиться для того, щоб наша база даних міогла правильно обробляти ключі змінної довжини та ключі, що мають спільні довгі загальні префікси.

Redis - це сервер пам’яті ключ-значення в мережі, що підтримує запити діапазону. Це також назва структури даних, яку сервер використовує для зберігання пар ключ-значення. Структура даних Redis розроблена для ефективної підтримки ключів довільної довжини, включаючи ключі з довгими загальними префіксами. Система Redis підтримує операції getc (k), putc (k, v), remove (k) та getrangec (k, n). Як можна було очікувати, get повертає значення, пов'язане з ключем k. put додає пару ключ-значення (k, v) у сховище ключ-значення або оновлює значення (v), пов'язане з ключем k. getrange повертає до перших n пар ключ-значення, більших або рівних k у базі даних. Нарешті, параметр c - це список номерів стовпців, які дозволяють клієнту отримувати або встановлювати частини значення ключа.

Структура дерева Redis показана на рисунку 8. Дерево має декілька шарів, що містять один або кілька B + -дерева. Кореневий шар містить щонайбільше одне B + -Tree, другий шар містить щонайбільше 264 дерева - по одному для кожного можливого ключа, який може мати перший шар, і так далі. Кожне B + -дерево і шар відповідають за якийсь 8-байтовий фрагмент ключа. B + -дерева Redis складаються з внутрішніх вузлів та вузлів кордону (див. Малюнок 7).

```
struct interior_node:
    uint32_t version;
    uint8_t nkeys;
    uint64_t keyslice[15];
    node* child[16];
    interior_node* parent;

union link_or_value:
    node* next_layer;
    [opaque] value;
```

```
struct border_node:
    uint32_t version;
    uint8_t nremoved;
    uint8_t keylen[15];
    uint64_t permutation;
    uint64_t keyslice[15];
    link_or_value lv[15];
    border_node* next;
    border_node* prev;
    interior_node* parent;
    keysuffix_t keysuffixes;
```

Рисунок 7: Структури даних вузла Redis [6]

Прикордонні вузли схожі на традиційні вузли листя B + -Древа. Вони мають перехресні посилання (для сприяння операції getrange), ключі та значення ключів. Однак прикордонні вузли Redis унікальні тим, що їх ключі ідентифікуються за допомогою 8-байтового фрагмента ключа та довжини ключа. Це означає, що одне Redis B + -Tree може представляти кілька (не більше 10) ключів на кожному фрагменті ключів. Щонайбільше 9 різних ключів для кожного можливого префікса зрізу ключів (ключі довжиною від 0 до довжини 8) та 10-й ключ, якщо є суфікс ключа або посилання на наступний шар. Не всі фрагменти ключів можуть мати 10 різних ключів. Зріз ключів без нульових байтів може мати лише два різних ключі. Ключ із нульовим байтом в кінці може мати три різні ключі. Більше того, для кожного ключа в прикордонному вузлі є значення або посилання на B + -Tree в наступному шарі. Чи вказує ключ на значення чи інше B + -Tree, залежить від значення поля довжини ключа. Якщо довжина ключа ключа більше 8, тоді ключ має суфікс та значення або має посилання на інший шар замість значення.

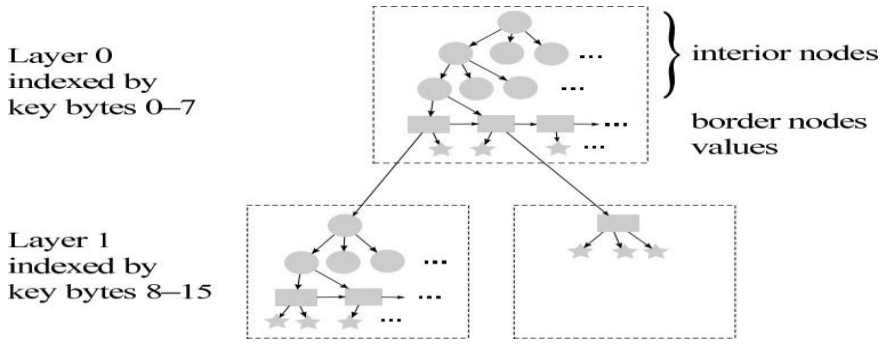


Рисунок 8: Структура дерева Redis

Редіс прикордонні вузли також містять структури даних суфікса ключів у тому випадку, якщо суфікси ключа не спільно використовуються іншими ключами з тим самим префіксом. Тобто, якщо ми маємо ключі "AB" і "ABCDEFGH!" у нашому сховищі ключ-значення немає необхідності створювати другий шар для індексації останнього байта другого ключа. Навпаки, ми зберігаємо "AB" (2 байти) і "ABCDEFGH!" (9 байт) з відповідними фрагментами ключів та довжинами ключів у B + -Tree з одним вузлом (рівень 0). Потім ми встановлюємо суфікс ключів для

останнього з фрагментів ключів як "!". Однак, якщо у нас є інший ключ "ABCDEFGH *", фрагмент ключа "ABCDEFGH" тепер повинен вказувати на новий шар. Крім того, рядок, суфікс якого "!" вставляється в новий шар із початковим значенням та новим рядком, суфікс якого "*" вставляється в новий шар із його значенням. Як правило, якщо ключ довжиною 8 годин, його значення зберігатиметься на рівні x, де x <h. Отже, якщо ключ становить 8 байт або менше, його значення буде посилатися на фрагмент ключів у B + -Tree кореневого шару. Якщо ключ довжиною 40 байт, його значення зберігатиметься в одному з шарів від 0 до 4. Це пов’язано з тим, що суфікси ключів можуть зберігатися в прикордонному вузлі, якщо жодні інші префікси повного ключа не діляться в ключах до шару та частина суфікса. Структура даних keyuffix означає, що непотрібні шари не створюються, коли існує багато відносно довгих ключів, які не мають спільних префіксів. З іншого боку, якщо є довгі ключі, які мають спільні префікси, то за частину загального префіксу відповідають кілька шарів (відносно невеликі шари).4.

РОЗДІЛ 2. ПОСТАНОВКА ЗАДАЧІ.

2.1. Операції

Ця загальна система реалізована як бібліотека мови програмування Go.NewStore повертає нещодавно створений екземпляр сховища ключ-значення. Get отримує значення, пов'язане з даним ключем у сховищі. Put додає пару ключ-значення до сховища ключ-значення або оновлює значення, пов'язане з існуючим ключем. Наша структура створюється з масиву символів із зазначеною довжиною.

2.2. Огляд архітектури

Ця система використовує дві внутрішні бібліотеки: бібліотеки B + -Tree і бібліотеку Node. Бібліотека B + -Tree дозволяє створювати B + -Древа та виконувати операції над цими B + -Древами. Бібліотека Node реалізує варіант Redis’s Node (вузли на межі двох шарів btree). Див. Розділ3.3 для більш детальної інформації. Ми також реалізуємо бібліотеку утиліт для деяких загальних визначень та операцій, що використовуються в різних бібліотеках.

2.3. Використання B + -Tree

Структура даних B + -Tree у нашій системі схожа на традиційну B + -Tree. Однак у своїй реалізації ми не використовуємо перехресні зв’язки між вузлами листя, а навпаки, додаємонову незалежну допоміжну структуру даних, що називається курсором B + -Tree. Курсор B + -Tree може розглядатися як масив покажчиків вузлів та відповідних покажчиків на записи в кожному вузлі (рис.2). Кожен запис у внутрішньому вузлі складається з ключа та вказівника праворуч від нього. Перший запис не має жодного ключа, пов’язаного з ним. Це просто вказівник. У листових вузлах записи складаються з ключів та покажчиків на запис / значення, з якими вони пов'язані. Перший елемент курсору вказує на запис у кореневому вузлі. Другий елемент у курсорі вказує на запис у дочірнім елементі запису, на який вказано в корені, і так до тих пір, поки елемент курсору не вкаже

на запис у листовому вузлі. Отже, курсор вказує на всі записи (і вузли) в унікальному шляху від запису в кореневій до запису у листовому вузлі.

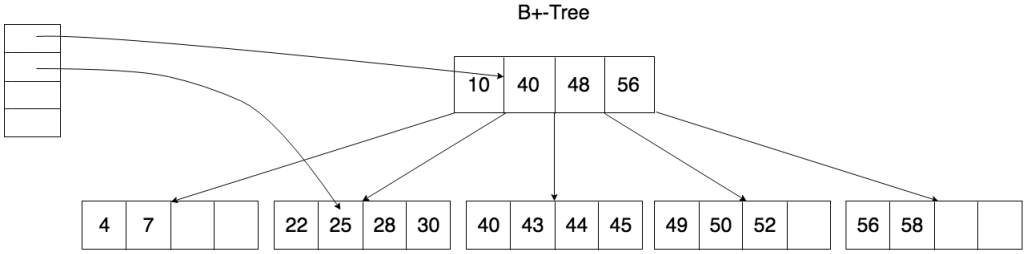


Рисунок 9: Курсор B + -дерева, спрямований на ключ 25.

Важливо знати, що надихнуло використовувати це B+ дерево з концепцією курсору від SQLite. Реалізація B-Tree SQLite підтримує як B + -Trees (у їх документації називається B * -Trees), так і B-Trees. B + -Дерева використовуються як те, що SQLite називає "Таблиця B-Дерева", тоді як B-Дерева використовуються як "Індекс B-Дерев". Принадність цієї структури даних полягає в тому, що вона забезпечує як швидке послідовне отримання, так і розміщення. Більш конкретно, послідовна операція отримання або путів займає амортизований постійний час. Якщо ці твердження відповідають дійсності, вставка набору відсортованих пар ключ-значення займає лінійний час, а часткове сортування набору ключів, задіяних у серії операцій отримання та / або путу, може призвести до значного збільшення продуктивності. Це також означало б, що бази даних, що використовують цю схему B + -Tree, можуть скоротити час, необхідний для операцій, що створюють дерева / таблиці з відсортованих даних, наприклад відновлення відмов, реплікація даних. Більше того, якщо запити надходять партіями, їх можна частково відсортувати, щоб скоротити час обробки.

2.4. Курсори та операції

Інтерфейс модуля B + -Tree має різні методи роботи на B + -Trees, включаючи: однак у цьому розділі ми зосередимося на операціях MOVE і PUT на B + -Trees з FANOUT (максимальна кількість ключів на вузол). FANOUT можна варіювати, і в нашій реалізації йому встановлюється значення за

замовчуванням15, подібно до вузла FANOUT Redis.

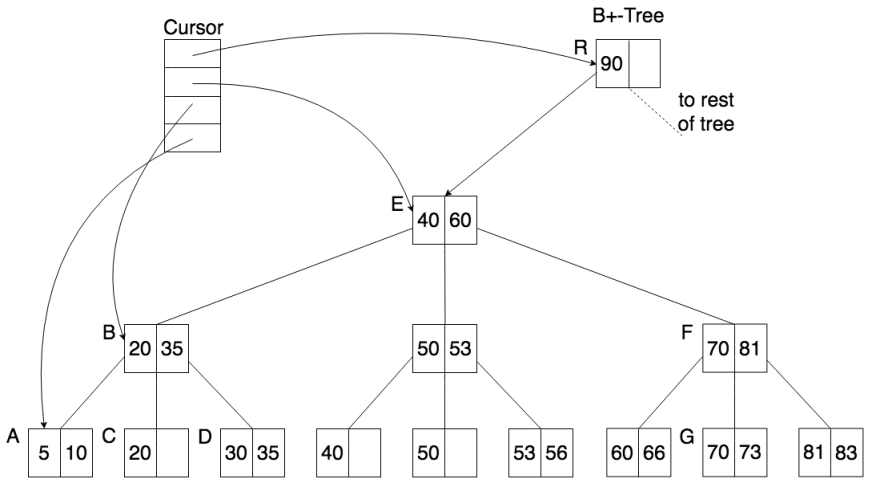
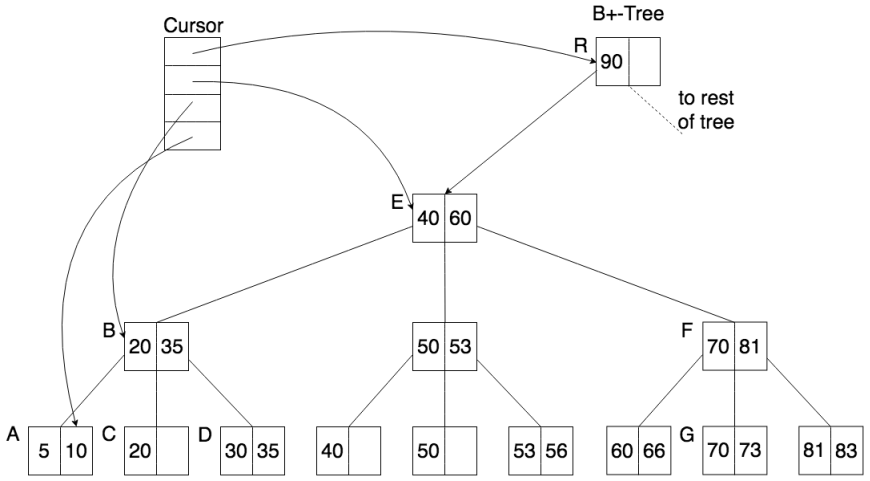


Рисунок 10: Курсор B + -дерева, спрямований на ключ 5.

Операції MoveToKey та MoveToKey прості, якщо потрібний ключ знаходиться в одному вузлі. Ми просто шукаємо правильний ключ у поточному вузлі. Наприклад, якщо курсор знаходиться на ключі 5 на малюнку10, операція MoveToNext або MoveToKey (10) повинна просто перемістити курсор до наступного запису у вузлі, запису 10. Однак, якщо потрібний ключ знаходиться в листовому вузлі, один або кілька вузлів подалі від поточного листового вузла, роботи буде більше беруть участь. У традиційній реалізації B + -Tree операція переходу до наступного просто використовує поперечні зв'язки між листовими вузлами, щоб дістатися до наступного вузла, а отже, і до наступного ключа. Тим не менше, випадковий пошук завжди починається з кореня. У нашій реалізації B + -Tree випадковий пошук (або наступна операція до запису у вузлі брата або сестри) починається з поточного листового вузла і піднімається і спускається по

дереву за необхідності, щоб дістатися до потрібного ключа.



Малюнок 11: Курсор B + -дерева, спрямований на ключ 10.

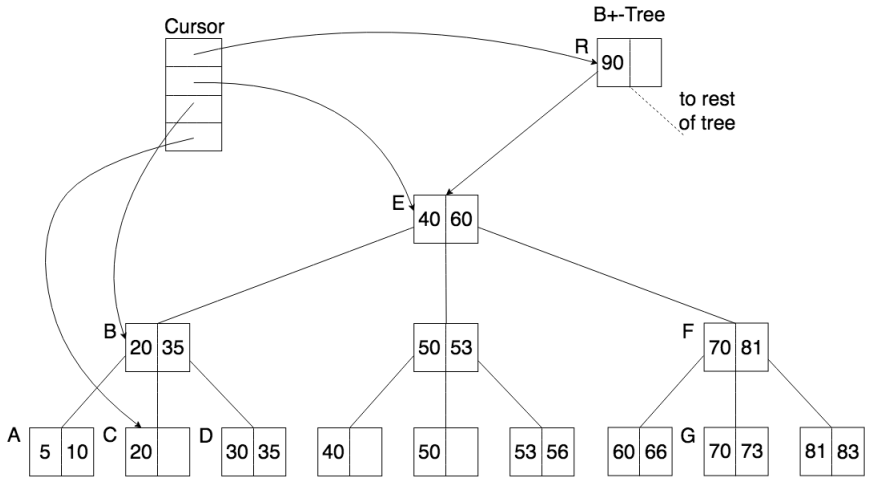


Рисунок 12: Курсор B + -дерева, спрямований на ключ 20.

Більш конкретно, у випадку операції MoveToNext курсор спочатку переміщується до найближчого вузла-предка, який не вказує на останній запис всередині. тобто вузол предка, для якого курсор може перейти до наступного запису всередині. Потім курсор встановлює вказівник на цьому рівні до наступного запису у вузлі, а потім опускається до найменшого дочірнього елемента запису. Наприклад, розглянемо операцію MoveToNext з ключі 10 (рис11), це переміщує останній вказівник курсора із запису 10 на вузлі A до запису

20 на вузлі C (рис 12). До для цього курсор спочатку піднімається до вузла B. Потім він переміщує вказівник на цей рівень до наступного запису у вузлі B. Нарешті, він опускається до поточного запису у найменшому дочірньому листі вузла B. Якби курсор спочатку знаходився в останньому записі у вузлі B, йому довелося б продовжувати зростати, поки він не дійде до кореня або вузла, де його вказівник на цей вузол / рівень раніше не вказував на останній запис у вузлі. Наприклад, якщо курсор знаходився на ключі 35 (вузол D), перехід до наступного запису (ключ 40) передбачає переміщення вгору до вузла E, переміщення покажчика входу в цьому вузлі до наступного запису, а потім опускання вліво листя дитини.

Для при випадковому пошуку курсор повинен переміститися вгору до найближчого вузла, який, безумовно, є родоначальником потрібного ключа пошуку. Це вузол, де ключ пошуку більший або дорівнює першому запису і суворо менший за останній запис. Єдиним винятком з цього правила є корінь. Корінь є предком кожного вузла в дереві. Після того, як був знайдений певний предок ключа, курсор може спуститись вниз по дереву від цього вузла до потрібного листа. Наприклад, нехай курсор знаходиться на ключі 20 (рис12). Що відбувається, коли викликається MoveToKey (50)? Щоб перейти до ключі 50, курсор спочатку піднімається до вузла E. Він не виходить за межі вузла E, оскільки 50 знаходиться між найменшою та найбільшою ключем вузла E. Потім ми опускаємось до вузла H, а потім до вузла I, який містить ключ 50 (рис12). Що робити, якщо тоді здійснено дзвінок на MoveToKey (73)? У цьому випадку курсор піднімається аж до кореня R, а потім опускається до вузла E, потім до вузла F, а потім до вузла G. Курсор вказує на ключ 73 (рис.14). Курсор повинен підніматися аж до вузла R (корінь), оскільки неможливо просто переглядати записи вузла E, щоб точно знати, що вузол E, безумовно, є родоначальником ключа 73. Усі записи вузла E можуть повідомити нам, що діти найбільшого входу більше або дорівнюють 60. Не виключено, що інший дочірній матері батька E може бути

родоначальником ключа 73. Це було б у випадку, якщо перший ключ у вузлі R менше 73. Це приклад найгіршого випадку випадкового пошуку за схемою B + -Tree. У гіршому випадку для дзвінка MoveToKey потрібен час, пропорційний 2logN. logN час, щоб піднятися до кореня, і logN час, щоб спуститися з кореня до відповідного листового вузла. Це на відміну від випадкового пошуку в традиційній схемі B + -Tree, яке завжди займає час, пропорційний logN. Однак важливо зазначити, що вартість підйому не буде значною, оскільки більшість (якщо не всі) вузлів у висхідному шляху, ймовірно, перебувають у кеш-пам'яті (точніше вищі рівні кеш-пам'яті), оскільки вони нещодавно були відвідані . З іншого боку, кожного разу, коли ми виконуємо випадковий пошук сусіднього вузла, нам вигідно не потрібно підніматися до кореня, а більшість вузлів, які ми відвідуємо, перебувають на вищих рівнях кеш-пам'яті.

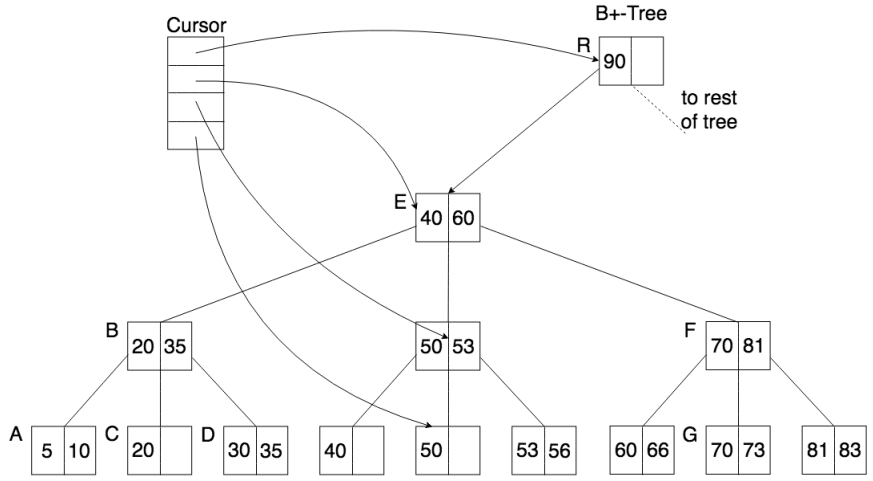
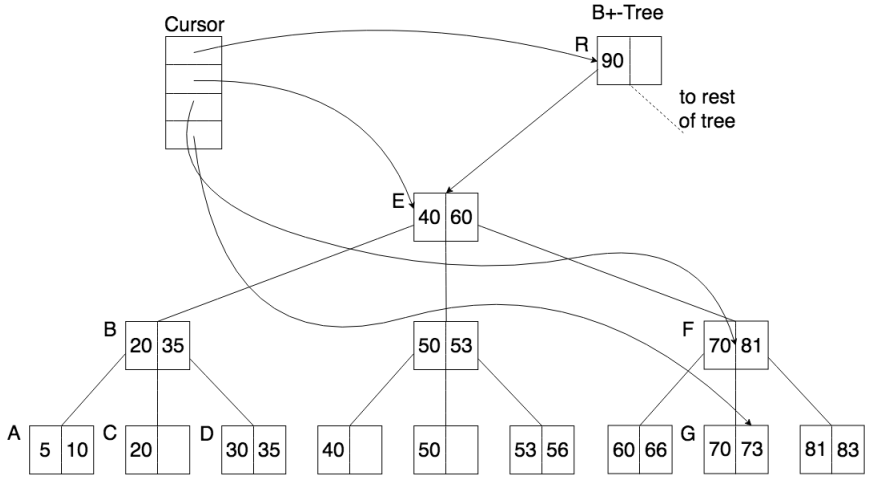


Рисунок 13: Курсор B + -дерева, спрямований на ключ 50.

Послідовні (наступні) операції також швидкі. Велику частину часу вони спричиняють просто перехід до наступного елемента у вузлі, кілька разів вони спричиняють перехід від вузла до його правого вузла-сестри, що часто включає менше доступу до вузла, а іноді, у гіршому випадку, включає 2-годинний доступ до вузла, де h - висота дерева. Вставки можуть отримати користь від використання цієї схеми (тобто використання курсору, який запам'ятовує своє поточне розташування та шлях до нього). З послідовними вставками, більшу частину часу поточний вузол має достатньо місця для наступного ключа. Іноді нам потрібно

розділити вузол (розбиття вузлів коротко описано в2.2).Здебільшого достатньо одного розколу. Іноді розщеплення поширюється вгору до одного або декількох вузлів. Іноді розщеплений вузол поширюєтьсяна кілька рівнів. Як результат, ми очікуємо, що послідовне або випадкове розміщення поблизу поточного розташування курсору буде швидким і просто включає витрати на переїзд до сусіднього місця та введення ключа; обидва ці кроки (переміщення та розміщення) часто швидкі, якщо ключ, яку потрібно помістити в дерево, знаходиться поруч із ключем, на яку вказує курсор.



Малюнок 14: Курсор B + -дерева, спрямований на ключ 73.

На закінчення, окрім зменшення кількості звернень до вузлів, необхідних для операцій переміщення та розміщення, B + -Tree зі схемою курсора отримує велику вигоду від розташування кешу, коли курсор не рухається занадто далеко. Менше шансів отримати доступ до непотрібних вузлів у шляху від батьківського до бажаного ключа, швидше за все, повторно відвідати раніше відвідані вузли. Наприклад, під час обходу дерева з ключів 5 до 35 (уявіть запит діапазону або послідовне оновлення), це B + -Tree зі схемою курсору відвідує лише вузли A, B, C та D. Примітка. Ми відвідуємо вузол B FANOUT (два) рази . Це означає, що якщо послідовність операцій частково відсортовано за ключем, ми можемо зазнатизначний приріст продуктивності, залежно від того, наскільки ми

відсортуємо послідовність операцій. Важливо зазначити, що використання SQLite (яке використовує це B + -Tree зі схемою курсору) не завжди вигідно, якщо курсор знаходиться біля ключі, близької до ключі пошуку. Дійсно, функція sqlite3BtreeNext SQLite поводиться подібно до нашої функції MoveToNext: більшу частину часу наступний ключ буде знаходитись у поточному вузлі, а курсори SQLite піднімаються по дереву лише за потреби. Тим не менше, функція SQLite MoveTo завжди починає пошук потрібного ключа з кореня, якщо курсор уже не знаходиться у правильному положенні. Це на відміну від нашої реалізації MoveToKey, яка ніколи не починає пошук з кореня. Отже, очікується, що частково випадкова послідовність операцій буде виконуватися так само швидко, як випадкова послідовність операцій, і в SQLite не буде підвищеної продуктивності.

2.5. Впровадження сховища

У цьому розділі ми обговорюємо нашу реалізацію сховища формату ключ-значення. Наш сховище ключ-значення реалізовано на мові програмування C як бібліотека (звана KeyValueStorage). На даний момент він підтримує операції, описані в розділі3.1. Він є клієнтом модуля B + -Tree 3.3 та модуль Node. Наш модуль B + -Tree, як є, може обробляти лише 8-байтові ключі фіксованої довжини. Тому його потрібно розширити для обробки ключів змінної довжини. Як згадувалося в розділі2.3, бази даних та системи зберігання, що використовують B + -Дерева, як правило, обробляють ключі змінної довжини, додаючи рівень опосередкованості між масивом слотів вузла та ключем запису / запису в області даних вузла; ми також побачили, як B + - дерева не оптимізовані для роботи з ключами з довгими загальними префіксами. Ми обговорили, як Redis ефективно обробляє ключі змінної довжини та ключі з довгими загальними префіксами. Як результат, наш модуль KeyValueStorage базується на Redis. Це також трикоподібна конкатенація B + -Tree.

.

2.6. Модуль зберігання ключа-значення та модуль Node

Наш модуль зберігання ключ-значення базується на Redis. Однак є кілька відмінностей. Як уже згадувалося раніше, це клієнт модуля B + -Tree3.3.Ми також прийняли рішення про впровадження прикордонних вузлів окремо від дерева B +; листові вузли нашого B + -Древа не є вузлами кордону. Листяні вузли нашого модуля B + -Tree (перелік1) мати зрізи ключів фіксованої довжини та пов'язане з ними значення, але не мають полів довжини ключа або суфіксів ключів, як у Redis (див. рис 7). Крім того, наші граничні вузли містять фрагмент ключа, масив значень і суфікс ключа (перелік 2). Кожен прикордонний вузол пов'язаний зі зрізом ключа і не має масиву довжин ключів; довжина ключа є неявною в позиції значення в масиві значень. Якщо в бордерноді є суфікс ключа, це означає, що 10-й ключ не має посилання на наступний шар, і навпаки.

Ця конструкція, що відокремлює концепцію бордового вузла від концепції B + -Древа на два окремі модулі для використання сховищем формату ключ-значення, забезпечує більшу модульність. Модулі B + -Tree та Node можна легко замінити різними реалізаціями. Важливо також зазначити, що ця реалізація Node використовує менше пам'яті, коли є ключі з декількома загальними префіксами. Наприклад, якщо є 10 ключів зі зрізом «00000000», де кожен символ є символом NULL. У Redis буде 10 ключових фрагментів, 10 значень (посилань) і 10 довжин. У нашій схемі в B + -Tree був би один ключовий зріз, одне посилання на Node, на яке вказує запис у листовому вузлі, 10 значень і 4 накладні поля в Node. У цьому сценарії ми використовуємо приблизно половину місця, яке використовує Redis. Однак, коли кожен фрагмент ключа унікальний, Redis має фрагмент ключа, значення та довжину ключа, пов'язані з кожним ключем, тоді як ми маємо фрагмент ключа, посилання на вузол межі та пробіл для 10 значень, пов'язаних з кожним ключем. Зверніть увагу, що ми можемо оптимізувати використання

простору нашої реалізації Node, видаливши поле зрізу ключів у Node і використовуючи більш ефективну реалізацію Node, коли з фрагментом ключів Node є лише кілька ключів. Це поле є зайвим, оскільки воно вже буде присутнє в записі B + -Tree, що вказує на граничну вершину. Зверніть увагу, що ми можемо оптимізувати використання простору нашої реалізації Node, видаливши поле зрізу ключів у Node і використовуючи більш ефективну реалізацію Node, коли з фрагментом ключів Node є лише кілька ключів. Це поле є зайвим, оскільки воно вже буде присутнє в записі B + -Tree, що вказує на граничну вершину. Зверніть увагу, що ми можемо оптимізувати використання простору нашої реалізації Node, видаливши поле зрізу ключів у Node і використовуючи більш ефективну реалізацію Node, коли з фрагментом ключів Node є лише кілька ключів. Це поле є зайвим, оскільки воно вже буде присутнє в записі B + -Tree, що вказує на граничну вершину.

Ще одна відмінність між нашою реалізацією сховища ключ-значення та Redis полягає в тому, що для того, щоб скористатися перевагами нашого B + -Tree з реалізаціями курсору, ми постійно пов'язуємо кожне B + -Tree з курсором. Це для забезпечення більш високої продуктивності, коли ключ, що бере участь у наступній операції, лексикографічно близький до ключа, який бере участь у попередній операції

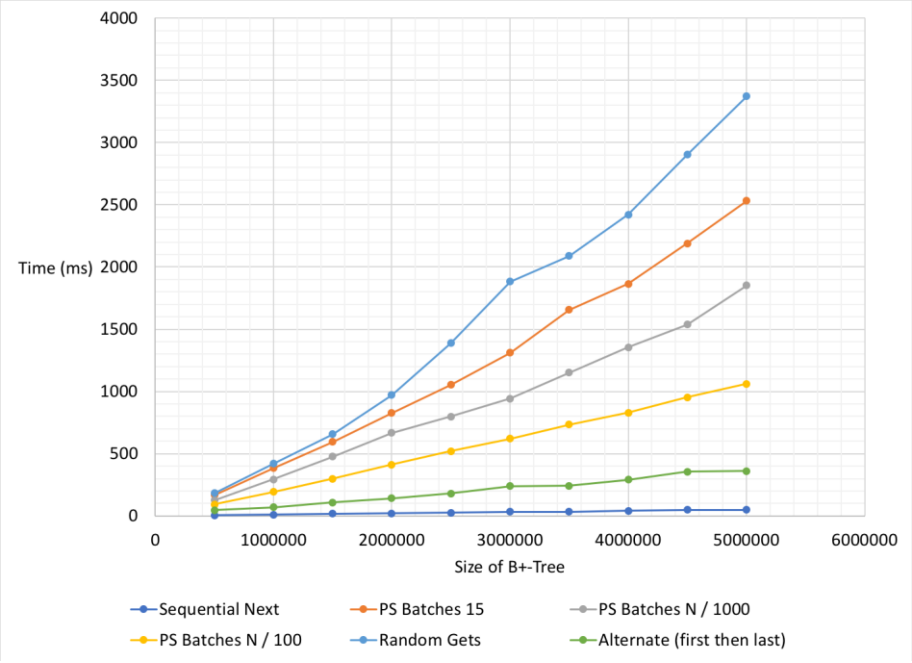
.

.

РОЗДІЛ 3. ОЦІНКА РЕЗУЛЬТАТУ ПОБУДОВИ ПЕРСИСТЕНТНОЇ БАЗИ ДАНИХ

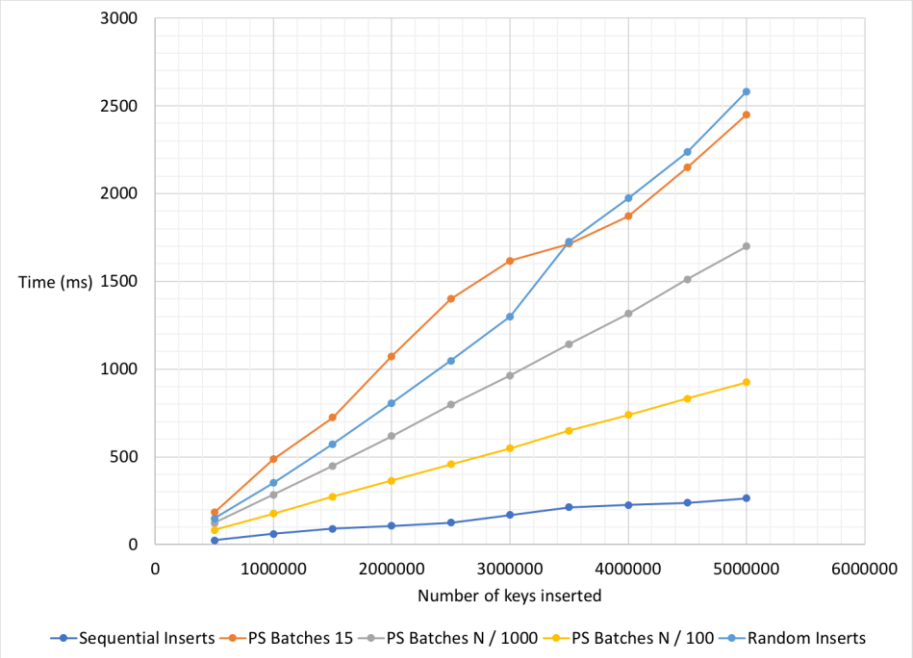
3.1. B + -Tree

Щоб оцінити ефективність операцій отримання B+-Tree, ми визначили, скільки часу потрібно для переміщення та отримання всіх ключів у B + -Tree для різних навантажень. Тобто, ми відвідали або отримали всі ключі в B + -Tree послідовно, у частково відсортованому порядку та випадковим чином для різних розмірів B + -Tree. Існує три типи частково відсортованих навантажень: відсортовані по партіях з 15 пар ключ-значення (PS Пакети 15), навантаження з 1000 відсортованих пакетів, кожна з розміром N / 1000 (PS Пакетів N / 1000), і, нарешті, робочі навантаження з 100 відсортовані партії, кожна з розміром N / 100 (PS партії N / 100). Для кожного B + -дерева з розміром N було створено B + -Tree, вставляючи всі ключі від 0 до N - 1. Результати цього експерименту показані на рисунку 17.



Малюнок 17: Час відвідати або отримати всі ключі в B + -Tree для різних розмірів B + -Древа та різного навантаження.

Подібним чином для оцінки продуктивності операцій з розміщенням B + -Tree ми визначили час, скільки часу потрібно для вставки послідовності ключів у B + -Tree із зазначеними вище робочими навантаженнями (послідовними, частково відсортованими та випадковими). Результати показані на малюнку 18.



Малюнок 18: Час вставити всі ключі від 0 до розміру - 1 у порожнє B + -Tree для різних розмірів та різного навантаження.

Нарешті, ми використовувати гіпотезу подвоєння [9] емпірично виміряти порядок зростання послідовних та випадкових операцій. Гіпотеза подвоєння стверджує, що якщо T (N), то T (2N) ~ 2b. Отже, розрахувавши часові співвідношення часу, який ми зайняли, ми можемо оцінити порядок зростання нашого B + -Древа при різних робочих навантаженнях.

Як ми бачимо з рисунків 17 і 18 послідовні операції дуже швидко вводять це B + -Tree, тоді як випадкові операції є найповільнішими. Ми також бачимо, що наші B + -Дерева мають вищі результати, якщо серія операцій put or get частково сортується у відсортовані партії. Чим більші відсортовані партії пропорційні кількості ключів, які потрібно вставити або отримати, тим більший приріст

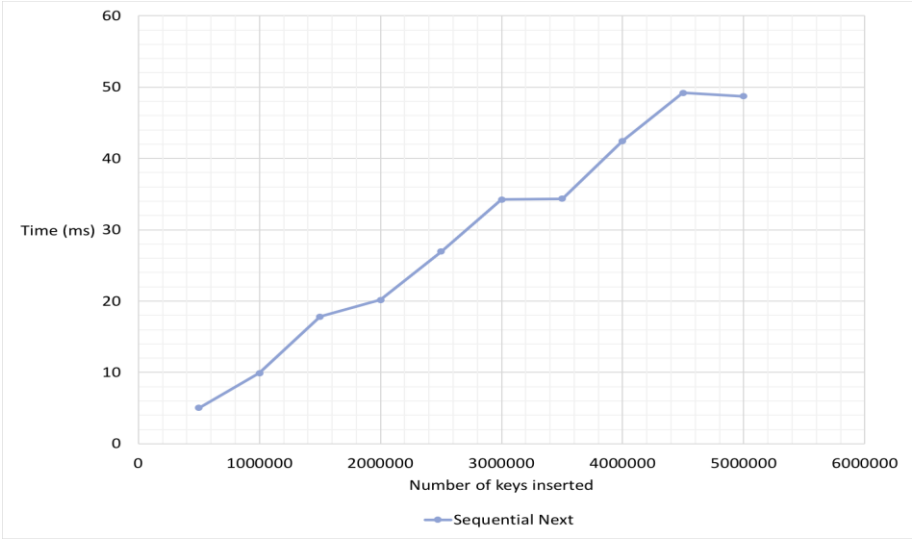
продуктивності. З іншого боку, ми очікували, що почергове отримання першого та останнього ключів у B + -Tree його розмір, скільки разів буде моделювати найгіршу продуктивність, оскільки кожне отримання коштує 2-кратного доступу до вузла. Незважаючи на те, що теоретично це повинно моделювати найгіршу поведінку дерева B + - Tree на практиці не робить через кешування. Якщо постійно отримуються однакові два ключі, кожен раз проходять одні й ті самі вузли-предки, щоб перейти від одного ключа до іншого.

Цей експеримент підкреслює значення зручних для кешування структур даних при впровадженні баз даних основної пам’яті, а також висвітлює, наскільки дорогими можуть бути пошуки основної пам’яті в порівнянні з пошуками кешу.

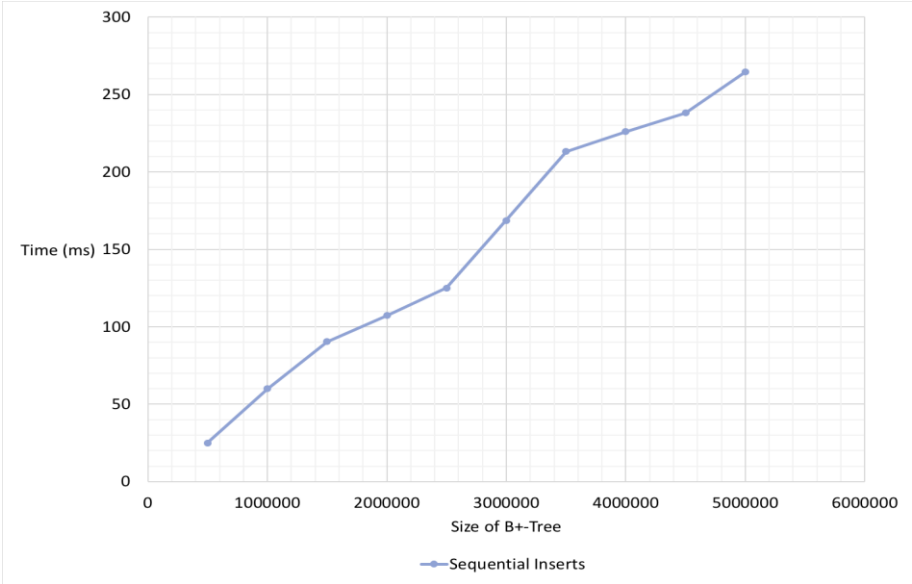
Цифри 19 і 20 дозволяють нам краще оцінити швидкість послідовних операцій. Ця реалізація B + -Tree може послідовно отримувати всі ключі з B + -Древа розміром 5 мільйонів приблизно за 49 мілісекунд; це приблизно 9,8 наносекунд на наступну операцію. Ми можемо послідовно вставити 5 мільйонів ключів у B + -Tree приблизно за 265 мілісекунд; це близько 53 наносекунд на наступну операцію.

Кількість ключів (мільйони)	Перейти до Далі		Випадкове отримання	
	Час (мс)	коефіцієнт журналу	Час (мс)	коефіцієнт журналу
0,5	5.03		182,48	
1	9,89	0,975	420.07	1.203
2	20.16	1,027	970,84	1.209
4	42,45	1,074	2421,73	1.319

Таблиця 2: Коефіцієнт журналу часу роботи



Малюнок 19: Час послідовного відвідування всіх ключів на дереві B + для різних розмірів B + -дерева



Малюнок 20: Час послідовної вставки всіх ключів від 0 до розміру - 1 у порожнє дерево B+ Tree.

Кількість ключів (мільйони)	Послідовний шлях		Випадковий пут	
	Час (мс)	коефіцієнт журналу	Час (мс)	коефіцієнт журналу
0,5	25.01		148,83	
1	59,89	1.260	350,62	1.236
2	107,29	0,841	805,68	1.200
4	225,94	1,074	1973.40	1.292

Таблиця 3: Коефіцієнт журналу тривалості послідовних та випадкових операцій путів для різних розмірів ключів.

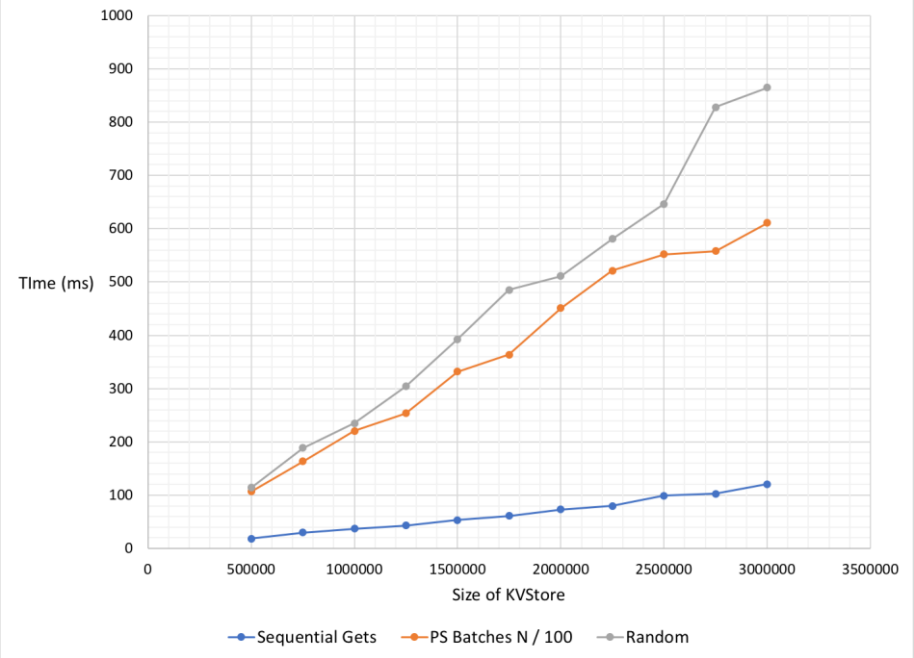
Використовуючи гіпотезу подвоєння, ми бачимо, що часова складність N послідовних операцій put or get фактично лінійна (див. Див 4.1 і 4.1). Коефіцієнт журналу сходиться приблизно до одиниці. Однак для випадкових операцій, навіть якщо коефіцієнт журналу округлюється до 1, дробова частина натякає на додатковий логарифмічний термін. Це узгоджується з нашими сподіваннями, що N випадкових операцій путів або отримання є лінійними.

.

3.2. Експерименти KeyValueStorage

Подібно до експериментів B + -Tree, ми оцінювали, скільки часу потрібно для отримання всіх ключів в об'єкті KeyValueStorage для різних розмірів та робочих навантажень, де кожен об'єкт KeyValueStorage будується із випадковими ключами. Ми бачимо, що, як і очікувалося, послідовні надходження дуже швидкі. Ми також бачимо, що часткове сортування ключів запитів забезпечує збільшення продуктивності порівняно із випадковими ключами запитів. Однак приріст продуктивності від часткового сортування не такий великий, як можна було б очікувати. Причини цього можуть бути різні. Можливо, навіть якщо наступний ключ, який вставляється або отримується, відрізняється від попереднього ключа

лише на останніх декількох байтах, Ця поточна реалізація KeyValueStorage все одно перетинає B + -Tree від першого шару до відповідного шару, де ключі розходяться. Незважаючи на те, що ми можемо пропустити всі шари, де фрагменти ключів наступного ключа збігаються із фрагментами ключів попереднього ключа.

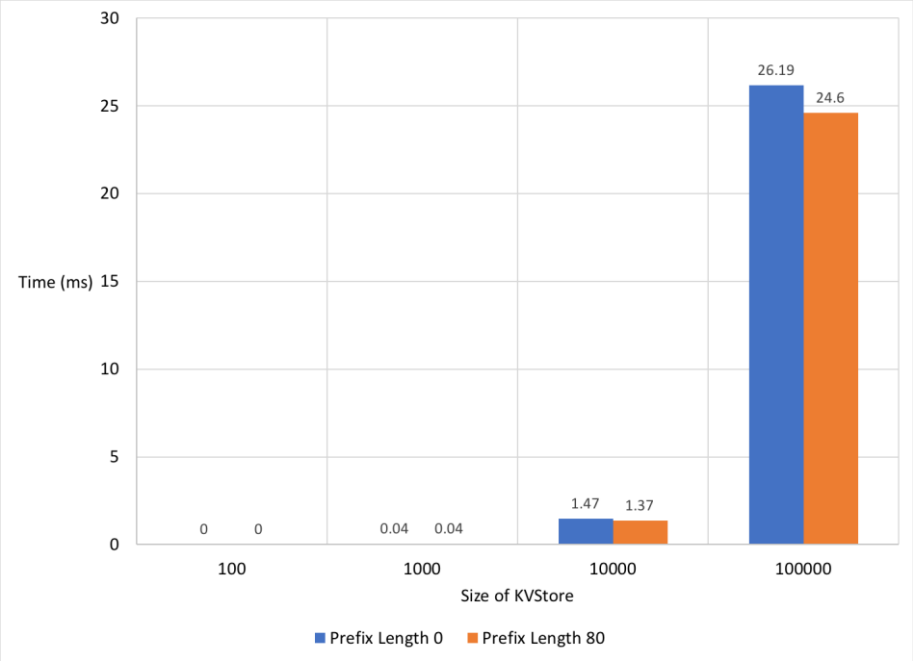


Малюнок 21: Час отримати всі ключі в KeyValueStorage для різних розмірів та різного навантаження.

Ключі мають довжину 30 байт і не повинні мати загальних префіксів. Ключі генеруються випадковим чином.

Від це обговорення в розділі 2.3,очікується, що типова реалізація B + -Tree і, за розширенням, сховище формату ключ-значення, побудоване з цього B + -Tree, зазнають погіршення продуктивності, коли більшість його ключів мають довгі загальні префікси. Однак Ця реалізація KeyValueStorage належним чином обробляє цей сценарій, як показано на малюнку. Ця реалізація KeyValueStorage працює трохи краще для ключів із довгими загальними префіксами. Це пов’язано з тим, що перші кілька шарів KeyValueStorage складаються лише з двох вузлів, кожен (листовий вузол B + -Древа з одним входом та бордовим вузлом). Отже,

щоб дістатися до будь-якого ключа, слід відвідати всі вузли, що містять загальний префікс, і тому ці вузли будуть знаходитись на вищих рівнях кеш-пам'яті.



Малюнок 22: Час на отримання всіх ключів в KeyValueStorage для різних розмірів для різної загальної довжини префікса.

Ключі довжиною 100 байт. Ми порівнюємо поведінку операцій get, коли перші 80 байт однакові, а решта 20 байтів випадкові, у порівнянні з випадковими всіма 100 байтами. Ключі генеруються випадковим чином.

.

ВИСНОВКИ

У цій дипломній роботі ми на високому рівні описали реалізацію нашого послідовного сховища формату ключ-значення основної пам'яті. Ми дали короткий огляд баз даних основної пам’яті, B + -Tree та Redis у розділі 2. Ми обговорили реалізацію сховища ключ-значення, B + -Tree з реалізацією курсору та чому послідовні операції B + -Tree займають амортизований постійний час у розділі 3. Ми оцінили поведінку нашої бібліотеки B + -Tree та бібліотеки KeyValueStorage у розділі4.Тут ми побачили, що B + -Tree з ідеєю курсора дійсно дає амортизовані постійні операції послідовного отримання (традиційні B + -Дерева можуть гарантувати лише операції отримання постійного часу). Більше того, ми показали, що часткове сортування послідовності входів веде до підвищення продуктивності для нашого B + -Древа та KeyValueStorage. Ми також показали, що тріада Редіса за схемою B + -Tree може правильно обробляти довгі загальні префікси. Нарешті, ми коротко представили спосіб розширення курсору B + -Tree до курсору KeyValueStorage для подальшого прискорення послідовних та частково відсортованих запитів у нашій системі.

ПЕРЕЛІК ПОСИЛАНЬ

1.Martin Fowler, Continuous Integration [Електронний ресурс]. – Режим доступу: https://www.martinfowler.com/articles/continuousIntegration.html

2.Rafał Leszko Continuous Delivery with Docker and Jenkins: Create Secure Applications by Building Complete CI/CD Pipelines, 2nd Edition. [Електронний ресурс].– Режим доступу: https://www.perlego.com/book/526974/continuous-delivery-with-docker-and-jenkins-pdf

3.David Farley, Jez Humble, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. [Електронний ресурс]. – Режим доступу:http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/lecturas/10_Continuous_Delivery_Dzone_Refcardz.pdf

4.Sander Rossel, Continuous Integration, Delivery, and Deployment: Reliable and Faster Software Releases with Automating Builds, Tests, and Deployment [Електронний ресурс]. – Режим доступу: http://tisten.ir/wp-content/uploads/2019/02/Sander-Rossel-Continuous-Integration-Delivery-and-Deployment-Packt-2017.pdf

5.Nikhil Pathania, Pro Continuous Delivery: With Jenkins 2.0 [Електронний ресурс]. – Режим доступу: https://www.oreilly.com/library/view/pro-continuous-delivery/9781484229132/

6.Mikael Krief, Building a good CI/CD pipeline [Електронний ресурс]. – Режим доступу:https://subscription.packtpub.com/book/cloud_and_networking/9781838642730/15/ch15lvl1sec128/building-a-good-ci-cd-pipeline

7.Alan Hohn, Hands-On Continuous Integration and Delivery with Jenkins X and Kubernetes [Електронний ресурс]. – Режим доступу: https://subscription.packtpub.com/video/cloud_and_networking/9781838982065

8.Shalabh Aggarwal, Flask Framework Cookbook - Second Edition [Електронний ресурс].– Режим доступу: https://subscription.packtpub.com/book/web_development/9781789951295

9. Jack Chan, Ray Chung, Jack Huang, Python API Development Fundamentals [Електронний ресурс]. – Режим доступу: https://subscription.packtpub.com/book/web_development/9781838983994

10. Gaston C. Hillar, Hands-On RESTful Python Web Services - Second Edition [Електронний ресурс]. – Режим доступу: https://subscription.packtpub.com/book/application-development/9781789532227

11. Mihalis Tsoukalos, Mastering Go - Second Edition [Електронний ресурс]. – Режим доступу: https//subscription.packtpub.com/book/programming/9781838559335

12. Naren Yellavula, Hands-On RESTful Web Services with Go - Second Edition [Електронний ресурс]. – Режим доступу: https://subscription.packtpub.com/book/web_development/9781838643577

13. Luca Ferrari, Enrico Pirozzi, Learn PostgreSQL [Електронний ресурс]. Режим доступу: https://subscription.packtpub.com/book/data/9781838985288

14. Vallarapu Naga, Avinash Kumar, PostgreSQL 13 Cookbook [Електронний ресурс]. Режим доступу: https://subscription.packtpub.com/book/data/9781838648138
15. Kenneth Geisshirt, Emanuele Zattin, Aske Olsson, Rasmus Voss, Git Version Control Cookbook - Second Edition [Електронний ресурс]. – Режим доступу: https://subscription.packtpub.com/book/application-development/9781789137545