УДК 004.042

Р. С. Клейменов, Т. А. Ліхоузова

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ КЕШУВАННЯ SQL ЗАПИТІВ

Анотація: Розглядається проблема ефективного кешування SQL запитів. В статті проведено порівняльний аналіз стандартних структур даних мови програмування Java, розглянуто їх основні особливості,переваги та недоліки. Наведено результати серії експериментів по тестуванню структур даних як сховища ключів (SQL стрічок), яку було проведено на заздалегідь імплементованому кеші запитів. Запропоновано структури даних, яких немає в стандартній бібліотеці Java, виконано їх реалізацію. Експериментально доведено, що використання таких структур даних дає кращі результати при кешуванні SQL запитів.

Ключові слова: кешування, Java, структури даних.

Опис проблеми

В додатках, написаних на Java,дуже часто з метою зменшення навантаження на бази даних використовують кешування. Одним з видів кешування даних на стороні сервера є кешування запитів до баз даних[1]. Для ефективного і раціонального використання функціональності з кешування запитів необхідно розуміти, як і з якими структурами даних може працювати даний вид кешування. В Інтернеті можна знайти достатньо інформації про те, що таке кеш, як і коли його застосовувати, приклади його використання. На порядок менше інформації віддають сучасні пошукові системи питанню - що таке кешування запитів і навіщо воно потрібне. І майже відсутня інформація про структури даних, які можуть бути використані в інструментах кешування запитів до баз даних, не говорячи вже конкретно про мову програмування Java. Саме тому постала задача проаналізувати існуючі структури даних мови програмування Java й ті, які потенційно могли б задовольнити потреби функціональності з кешування запитів.

Огляд існуючих рішень

Що таке структура даних? Структури даних – це способи організації даних в комп’ютерах. Часто разом зі структурою даних пов’язується і специфічний перелік операцій, що можуть бути виконаними над даними, організованими в таку структуру [2].

Правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів їх обробки. Добре побудовані структури даних дозволяють оптимізувати використання машинного часу та пам’яті

© Р. С. Клейменов, Т. А. Ліхоузова

комп’ютера для виконання найкритичніших операцій. Відома формула «Програма = Алгоритми + Структури даних» дуже точно виражає необхідність відповідального ставлення до такого підбору.

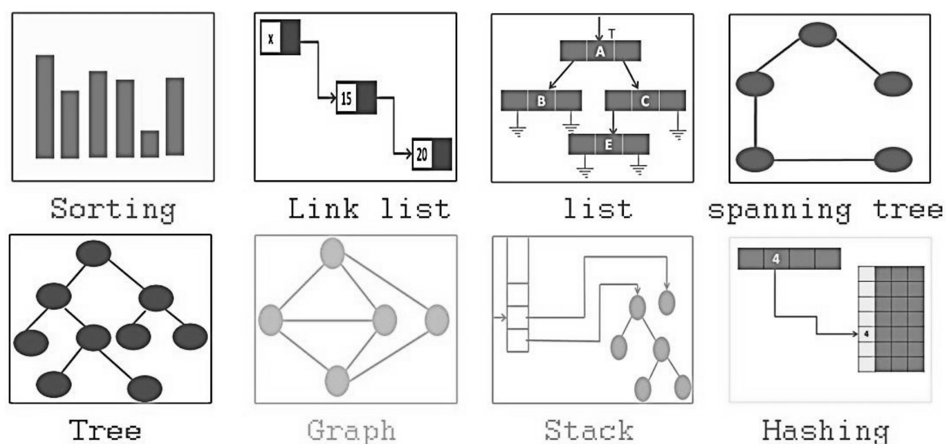

Рис. 1. Приклад структур даних

Одна з найбільш вживаних структур – *ArrayList* –є аналогом звичайного масиву даних (котрий присутній майже в будь-якій мові програмування), але має одну відмінну рису – можливість автоматично змінювати свій розмір під час роботи програми [3].

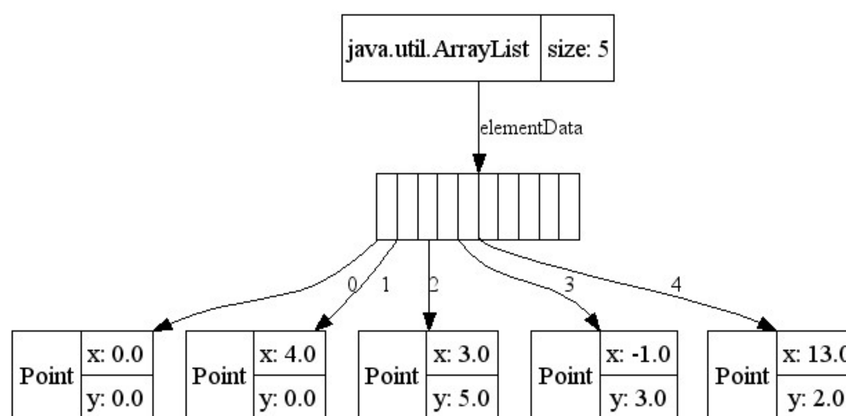

Рис. 2. ArrayList

При додаванні нового елементу в дану структуру перевіряється, чи достатньо місця в масиві для вставки нового елементу, якщо місця не достатньо, то створюється новий масив більшого розміру, а дані що були попередньо збережені, переносяться в новий, після чого новий елемент додається в кінець масиву.

Під час видалення елементу послідовно переглядаються всі елементи до тих пір поки не буде знайдено перше співпадіння ключа пошуку з елементом масиву. Видаляється перший знайдений елемент. Також можливе видалення елементу по індексу. Пошук елементу в даній структурі можливий за індексом або через послідовний перебір елементів.

Проведемо експеримент з даною (і наступними) структурами:

Візьмемо 3 набори даних - 100, 1000 й 10000 елементів. Кожний елемент вибірки - деякий запит до бази даних (SQL стрічка). Оскільки для інструментів кешування головними робочими характеристиками є швидкість отримання елементу з пам’яті та об’єм кешу в цілому, відповідно виміряємо їх. Для кожної вибірки проводиться 5000 вимірів - ключ, котрий ми хочемо знайти в даній структурі, додається в випадковому порядку, тобто при кожному вимірі він може бути доданий як перший, п’ятий, або останній і т.д. по рахунку елемент.

Тестування структур даних, як сховища ключів (SQL стрічок), було проведено на заздалегідь імплементованому кеші запитів, що відповідає наступним вимогам:

- має зручний інтерфейс;
- легкий та зрозумілий в налаштуванні;
- без надмірної функціональності;
- має просту інтеграція з SQL інтерфейсом Java;
- автоматична конвертація результату SQL запиту як Java об’єктів;
- по мінімуму використань сторонніх бібліотек.

Таблиця 1.

Результат експерименту з ArrayList

Кількість елементів (одиниць)	100	1000	10000
Середній час пошуку елементу (наносекунд)	1765	8500	112776
Об’єм структури (байт)	13280	132976	1336256

Наступною важливою структурою є *LinkedList* – структура даних що представляє собою двозв’язний список елементів, де кожен елемент даної структури містить покажчик на наступний елемент списку та на попередній [3].

При додаванні елементу створюється нова проміжна структура у ланцюжку, на дану структуру встановлюються покажчики з попереднього елемента і з наступного. При видаленні елементу перевизначаться покажчики на попередній і наступний елемент. Для пошуку елементів в даній структурі використовується ітератор, тобто пошук елементів здійснюється шляхом послідовного перебору

ISSN 1560-8956

елементів. Також можливий пошук елементів по індексу, але даний пошук все одно використовує ітератор.

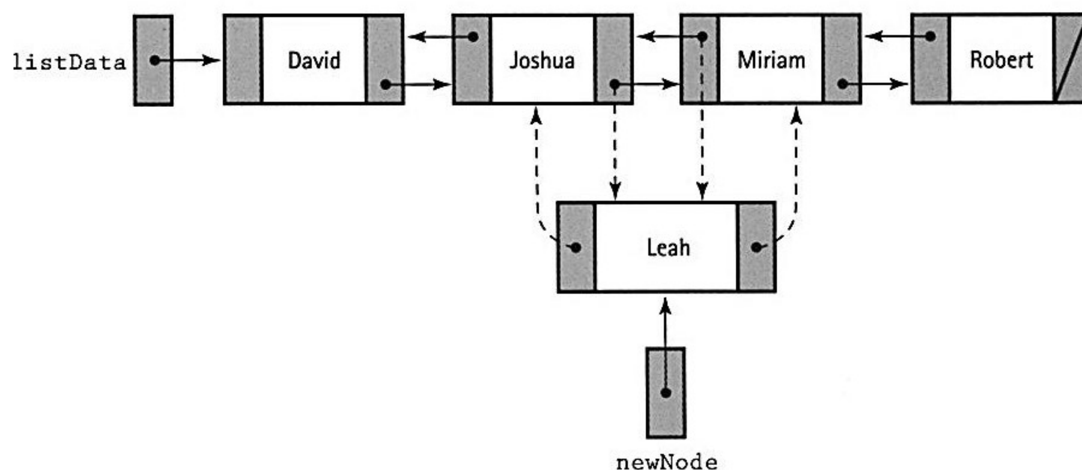

Рис. 3. Структура даних LinkedList

Таблиця 2.

Результат експерименту з LinkedList

Кількість елементів (одиниць)	100	1000	10000
Середній час пошуку елементу (наносекунд)	2055	9318	133631
Об’єм структури (байт)	15232	152914	1522534

Структура даних *HashMap* побудована на основі хеш-таблиць та зберігає данні у вигляді ключ/значення [3]. Дана структура може зберігати дані різних типів, вирішення колізій побудовано на основі методу ланцюжків. Враховуючи те, що для нас важливим показником є швидкість знаходження елементу у даній структурі, опишемо його: якщо значення ключа є пустим, то відбувається пошук елементу у ланцюжку, що знаходиться у комірці з індексом 0. Якщо значення ключа пошуку не пусте, вираховується хеш-значення даного ключа, за цим значенням вираховується значення комірки в якій зберігається ланцюжок значень. Шляхом послідовного порівняння відповідного ключа з ключами у ланцюжку знаходиться елемент, що відповідає даному ключу. Також дана сткутура надає можливість використовувати ітератори, що дозволяють отримати всі ключі та всі значення, або всі пари ключ/значення.

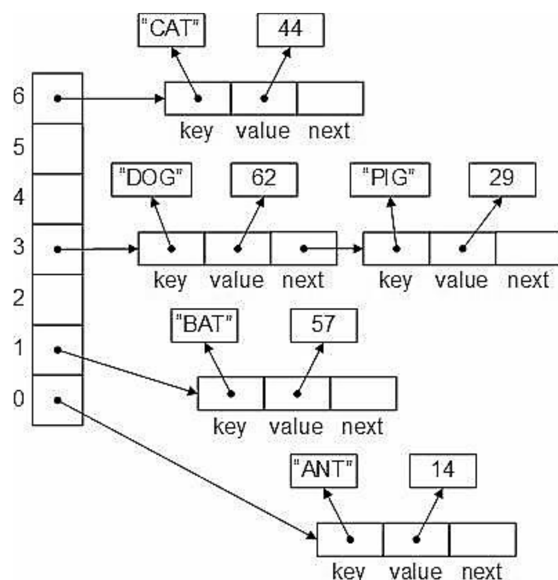

Рис. 4. Структура даних HashMap

Таблиця 3.

Результат експерименту з HashMap

Кількість елементів (одиниць)	100	1000	10000
Середній час пошуку елементу (наносекунд)	281	312	345
Об'єм структури (байт)	14688	144256	1425600

Структура даних *TreeMap* зберігає пари ключ-значення у вигляді червоно-чорного дерева, в результаті чого дані елементи є відсортованими за значеннями ключа (у випадку строкової інформації – відсортовані дані у лексикографічному порядку) [4].

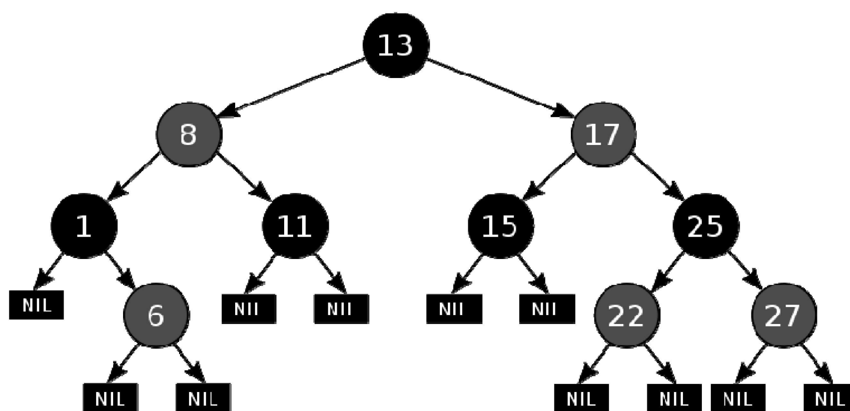

Рис .5. Структура даних RB-Tree, на якій працює TreeMap

Як працює червоно-чорне дерево можна розібратися за посиланням [3]. На-

ISSN 1560-8956

ведемо лише основні його характеристики: додавання елементу в дану структуру призводить до відповідних операцій балансування на червоно-чорному дереві, в зв’язку з цим повинно бути задано правило, за яким елемент, що додається, буде порівнюватись з іншими елементами, що вже є в дереві (для строкових даних це лексикографічні правила). Видалення елементу також призводить до відповідних операцій ре-балансування червоно-чорного дерева. Пошук елементу в даній структурі аналогічний пошуку в звичайному бінарному дереві пошуку.

Таблиця 4.

Результат експерименту з TreeMap

Кількість елементів (одиниць)	100	1000	10000
Середній час пошуку елементу (наносекунд)	607	962	2132
Об’єм структури (байт)	14448	145123	1445555

Структури *HashSet*[5] та *TreeSet*[6] не розглядаються через те, що вони імплементовані на основі HashMap та TreeMap і не підходять для цільового використання в кешуванні запитів.

Враховуючи префіксну структуру SQL запитів цікаво розглянути структури Trie та Radix tree.

Trie– префіксне дерево – структура даних, дерево, в якому шлях від кореня до листа визначає рядок. Рядки з однаковими префіксами мають спільний шлях від кореня довжиною цього префікса [7].

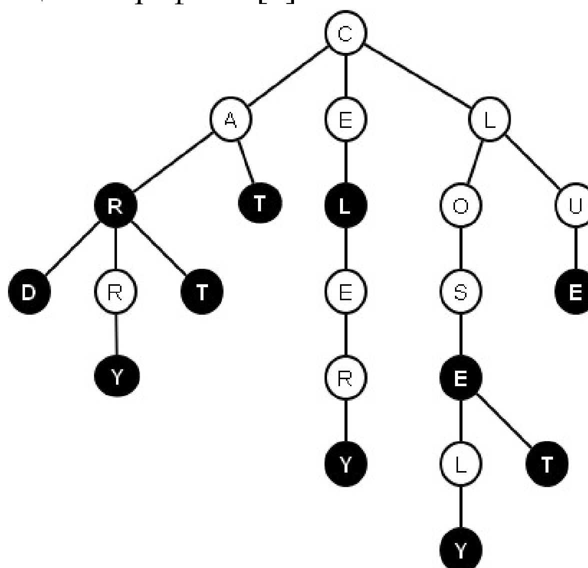

Рис .6. Структура даних Trie

Префіксне дерево дозволяє зберігати асоціативний масив, ключами якого є

рядки. На відміну від бінарних дерев, в листі дерева не зберігається ключ. Значення ключа можна отримати послідовним переглядом всіх батьківських вузлів, кожний з яких зберігає один або кілька символів алфавіту. Корінь дерева пов’язаний з порожнім рядком. Таким чином, нащадки вузла мають загальний префікс, звідки і пішла назва даного абстрактного типу даних. Значення, пов’язані з ключем, зазвичай не пов’язані з кожним вузлом, а тільки з листами і, можливо, деякими внутрішніми вузлами.

Стандартна реалізація мови програмування Java не має реалізації даної структури, тому вона була імплементована власноруч.

Таблиця 5.

Результат експерименту з Trie

Кількість елементів (одиниць)	100	1000	10000
Середній час пошуку елементу (наносекунд)	1689	7600	19002
Об’єм структури (байт)	31648	260256	5685600

Radix tree– компактне префіксне дерево – це структура даних, що представляє собою оптимізовану по пам’яті реалізацію префіксного дерева [8].

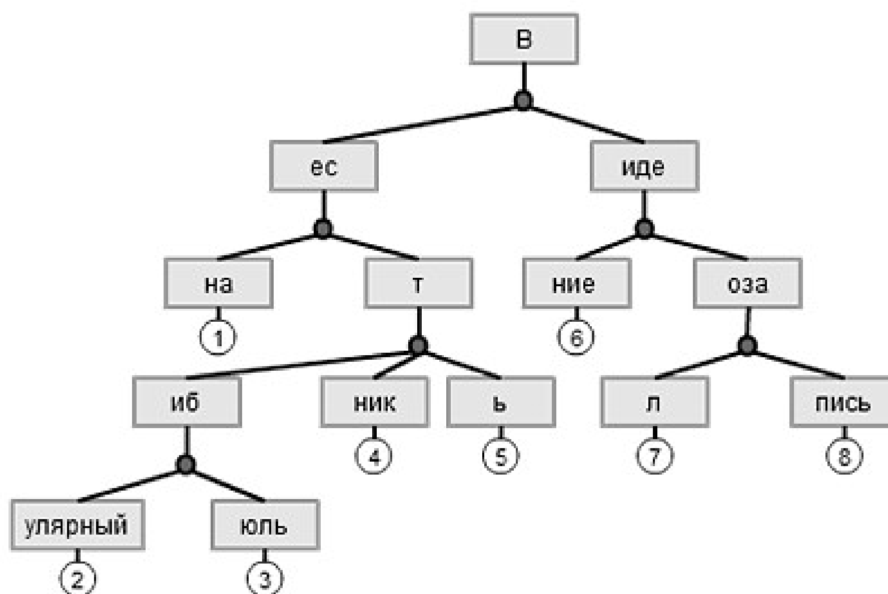

Рис. 7. Структура даних Radix tree

У базисному дереві вузол А, який є єдиним нащадком вузла В, зливається з вузлом В. Складність операцій пошуку, додавання і видалення елемента з базис-

 ISSN 1560-8956

ного дерева оцінюється як O(k), де k - довжина оброблюваного елементу. Час роботи не залежить від кількості елементів в дереві. На відміну від звичайних префіксних дерев, вузол базисного дерева може бути позначений як одним елементом (символом, бітом і т.д.), так і послідовністю елементів. Це робить базисне дерево більш ефективним для наборів рядків (особливо якщо самі рядки досить довгі), і також для наборів, що мають невелику кількість довгих префіксів.

Стандартна реалізація мови програмування Java також не має реалізації даної структури, тому вона була імплементована власноруч.

Таблиця 5.

Результат експерименту з Radix tree

Кількість елементів (одиниць)	100	1000	10000
Середній час пошуку елементу (наносекунд)	315	390	450
Об’єм структури (байт)	11685	109256	1105600

Висновки

Проаналізувавши результати експериментів можна зробити висновки: як сховище SQL стрічок найкраще використовувати структури даних HashMap, оскільки вона має найбільшу швидкість пошуку елемента й майже найменший розмір сховища та Radix tree, оскільки дана структура має найменший розмір сховища (приблизно на 30% менший за розмір сховища HashMap) й майже найменшу швидкість пошуку елементу (приблизно на 10% повільніша за HashMap).

В результаті роботи було створено простий у використані кеш запитів що відповідає вищезазначеним вимогам та може мати дві імплементації сховища:

- сховище створене на основі структури даних HashMap;
- сховище створене на основі структури даних Radix tree.

Перше сховище використовується у випадках коли для кінцевого продукту критичний час отримання необхідної інформації, друге сховище - критичний (обмежений) розмір оперативної пам’яті.

Список використаних джерел

1. *Клейменов Р.С.* Проблеми кешування даних при використанні мови програмування Java // Р.С. Клейменов, Т.А. Ліхоузова / Міжвідомчий науково-технічний збірник «Адаптивні системи автоматичного управління» № 2(31), 2017

2. *Вирт Н.Е.* Алгоритмы и структуры данных : научно-популярная книга / М.: Мир, 1989.— 360 с.

3. *Downey A.* Think Data Structures : science book // Green Tea Press, 2017 — 211 p.

4. *GoodrichM.* Data Structures and Algorithms in Java : science book / Michael T.

Goodrich, Roberto Tamassia / 4th edition — Department of Computer Science Brown University, 2015 — 924 p. / p. 521-581

5. https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html
6. https://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html
7. *Седжвик Р*. Алгоритмы на Java : научно-популярная книга // Р.Седжвик, К.Уэйн / 4-е изд., перераб. и доп. — М.: Вильямс, 2013.— 843 с.
8. *Ахо А*. Структуры данных и алгоритмы : научно-популярная книга // Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман / 1-е изд., перераб. и доп. — М.: Вильямс, 2016.— 620 с.

 ISSN 1560-8956