Отчет по лабораторной работе №1.

Предмет: Управление Данными.

Выполнили:

Александр Дыдычкин,

Александр Сергеев

студенты 14БИ-1

Исходный код: <https://github.com/GoAlexander/myDataBase>

**1) Временная статистика**

Тестировалось на:

HP PAVILION dv6-2025er

Диск: Western Digital BlackWD7500BPKX

Объем: 750 Гб

Объем буферной памяти: 16 М

Скорость вращения: 7200 rpm

Size of database is 10000

Addition result (one operation, ariphmetic average): 0ms

Addition result (10000 operations): ≈977ms

Find by name result: ≈4ms

Find by name old result (without seek): ≈31ms

Delete by name result: 1ms

Find by date result (2 elems): 1ms

Delete by date result (2 elems): 3ms

*// Находим и удаляем сразу 10000 элементов:*

Find by date result: ≈870ms

Delete by date result: ≈582ms

**2) Анализ сложности-эффективности алгоритмов в O-символике**

Добавление записи в БД:

Функция: add()

Сложность: O(const)

Комментарий: Плюс затраты на конкатенацию до длины строки 50-3-10-3 = 34, т. е. в наихудшем случае конкатенация строки будет проводиться 34 раза.

Поиск по ключевому полю в БД:

Функция: get()

Сложность: O(log n) (в наихудшем случае)

Поиск по не ключевому полю в БД:

Функция: getBySecondField()

Сложность: O(const) + O(n)

Комментарий: Вообще, зависит от количества найденных элементов. Алгоритм найдет элементы за O(1) (!), но ему еще нужно выгрузить их в память (прочитать элементы), а это уже O(n), где n - это количество найденных элементов.

Удаление записи из БД по ключевому полю:

Функция: delete()

Сложность: O(const)

Удаление записи из БД по не ключевому полю:

Функция: deleteBySecondField()

Сложность: O(const) + O(n), где n количество элементов в значении одной корзины hashmapSecond.

**3) Особенности реализации**

-Быстрый поиск элементов осуществляется благодаря использованию HashMap

-Для поддержания работоспособности БД в оперативной памяти нужно хранить только: 2 экземпляра HashMap и одну переменную. Они являются нашими метаданными.

-HashMap не поддерживает синхронизацию, поэтому поиск будет быстрее, чем теоретическая реализация с HashTable

-Благодаря сериализации метаданных открытие и закрытие БД происходит быстрее и с минимальной нагрузкой на CPU.

(В теоретической альтернативной реализации можно было бы не сохранять вышеуказанные метаданные, а каждый раз при открытии строить новые HashMap`ы)

-Удаление единичного элемента БД из происходит со скоростью удаления “корзины” (пары) из HashMap, т. к. физически элемент не удаляется из БД

-экспорт БД в формат .xls

-seek() протестирован и уверенно работает на Windows и GNU/Linux!

**4) Причина выбора реализации с не фиксированным размером хеш-таблицы.**

Была выбрана реализация с искусственно не фиксированным размером HashMap, так как хотели сделать максимально быстрый поиск (тем более, что задание обуславливает “войну” именно за скорость), даже в ущерб бОльшему потреблению оперативной памяти.

Конечно, преимущества фиксированной хеш-таблицы очевидны:

* меньшего использования оперативной памяти, а точнее константному;
* благодаря пункту выше, потенциально БД может хранить бесконечное количество данных (логично, что “забить” оперативную память легче, чем постоянную память, хотя в нашей реализации, когда в хеш-таблице хранится минимум - это еще тот вопрос).

Но опять же, задание не было нацелено на максимально масштабируемую и “экономичную” БД, задание было про быструю БД.

В заключении этой части хотелось бы выделить плюсы нашей реализации БД (с не фиксированной хеш-таблицей):

* быстрые операции поиска, удаления, редактирования;
* меньшее количество I/O операций;
* коллизии должны прекрасно разрешаться внутри HashMap (настоятельно рекомендую прочитать справочную информацию, находящуюся в конце отчета)

**5) Заключение**

В результате работы был произведен сбор данных, выполнено проектирование БД, а также ее программная реализация. Приобрели новые знания и навыки.

Также, хотелось бы отметить, что остается простор для дальнейшего совершенствования и оптимизации написанного проекта, к примеру:

-Еще раз проанализировать вопрос балансировки между потребляемой оперативной памятью и скоростью поиска;

-метод переиндексации БД (физическое удаление из БД “мусора”);

-более эффективный метод автозаполнения строк сервисным символом (“\_”);

-различные мелкие усовершенствования.

**Справочная информация:**

HashMap (Java 8)

The default load factor: 0.75 (then double in size)

Resolving collisions and O-metrics:

Prior to Java 8, HashMap and all other hash table based Map implementation classes in Java handle collision by chaining, i.e. **they use linked list to store map entries which ended in the same bucket due to a collision. If a key end up in same bucket location where an entry is already stored then this entry is just added at the head of the linked list there**. In the worst case this degrades the performance of the get() method of HashMap to O(n) from O(1). In order to address this issue in the case of frequent HashMap collisions, Java8 has started using a balanced tree instead of linked list for storing collided entries. This also means that in the worst case you will get a performance boost from O(n) to O(log n).

HashMap will initially use the linked list but when the number of entries crosses a certain threshold it will replace the linked list with a balanced binary tree.

Again: Java has chosen **chaining algorithm** for hashMap, so in case of collisions, items are chained together just like in a linkedList.

Comments:

Each bucket uses a form of linked list which has no limitation except memory. So in theory a HashMap can hold an unlimited number of elements. In practice you won't even get to 2^30 because you will have run out of memory long before that.

Links:

<https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>

<http://javarevisited.blogspot.ru/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html>

<https://www.linkedin.com/pulse/10-things-java-developer-should-know-hashmap-chinmay-parekh>