МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет прикладной математики, информатики и механики

Кафедра программного обеспечения и администрирования

информационных систем

Расширение экспериментального программного обеспечения для моделирования структур хранения данных

Дипломная работа

Специальность 010503 Математическое обеспечение

и администрирование информационных систем

Специализация Информационные системы

Допущено к защите в ГЭК \_\_.\_\_.2015

Зав. кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ д. ф.-м.н., проф. М.А. Артемов

Обучающийся \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Н.С. Скорняков

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ к. т. н., доц. К.Е. Селезнев

Воронеж 2015

**Аннотация**

Работа посвящена доработке и расширению существующего приложения, позволяющего сравнивать между собой различные контейнеры данных на основании подсчёта количества требуемых операций для выполнения той или иной нагрузки.

**Содержание**

[Введение 4](#_Toc422253398)

[1. Постановка задачи 6](#_Toc422253399)

[2. Анализ задачи 7](#_Toc422253400)

[2.1. Механизм анализа контейнеров 7](#_Toc422253401)

[2.2. Рефакторинг 11](#_Toc422253402)

[2.3. Хеш-таблицы 12](#_Toc422253403)

[2.4. Кэширование 13](#_Toc422253404)

[2.5. Адаптивные контейнеры данных 14](#_Toc422253405)

[3. Средства реализации 17](#_Toc422253406)

[4. Требования к аппаратному и программному обеспечению 18](#_Toc422253407)

[5. Интерфейс пользователя 19](#_Toc422253408)

[5.1. Команды консольного интерфейса 19](#_Toc422253409)

[5.2. Входные файлы 19](#_Toc422253410)

[5.3. Вывод результатов 21](#_Toc422253411)

[6. Реализация 23](#_Toc422253412)

[6.1. Рефакторинг 23](#_Toc422253413)

[6.2. Хеш-таблицы 30](#_Toc422253414)

[6.3. Кэширование 31](#_Toc422253415)

[6.4. Адаптивные контейнеры данных 32](#_Toc422253416)

[7. План тестирования 34](#_Toc422253417)

[7.1. Тестирование хеш-таблицы 34](#_Toc422253418)

[7.2. Тестирование контейнеров с кэшами 35](#_Toc422253419)

[7.3. Тестирование адаптивного контейнера 37](#_Toc422253420)

[Заключение 39](#_Toc422253421)

[Список литературы 40](#_Toc422253422)

[Приложение 1. Пример входного XML-файла 41](#_Toc422253423)

[Приложение 2. Листинг классов 42](#_Toc422253424)

Введение

С развитием информационных технологий и информатизацией общества происходит стремительный рост количества данных в цифровом виде (так называемая цифровая вселенная). Более того большинство создаваемой информации создаётся изначально именно в цифровом виде. По данным компании EDC количество информации в цифровом виде в 2013 году составляло 4,4 зеттабайт информации, а к 2020 году это значение, как предполагается, достигнет цифры 44 зеттабайт данных. Для сравнения менее 10 лет назад, в 2007 году размер цифровой вселенной составлял лишь 0,281 зеттабайт.

Огромное количество цифровых данных очень сильно влияет на скорость её обработки, что заставляет искать всё новые способы эффективного хранения информации и быстрого доступа к ней.

На сегодняшний день существует большое количество способов хранения данных отличающихся по тем или иным показателям. Кроме того существуют способы комбинирования различных хранилищ данных, что порождает огромное число всевозможных вариантов структурирования цифровых данных.

Анализируя работу хранилищ данных в тех или иных ситуациях можно сделать вывод о пригодности их для решения тех или иных классов задач, что в условиях постоянно увеличивающегося количества информации играет важную роль.

Существует множество факторов влияющих на производительность доступа, как внутренних (определяемых политикой хранения), например, уникальность хранимых данных, так и внешних, к которым можно отнести характер нагрузки на хранилище, то есть преобладание тех или иных операций производимых над хранилищами.

К примеру, в базах данных существует такой объект как индекс. Ускорение работы с использованием индексов достигается в первую очередь за счёт того, что индекс имеет структуру, оптимизированную под поиск. В большинстве случаев при построении индекса используются такие структуры данных, как сбалансированные деревья (B-деревья) или хеш-таблицых[4, 5]. Но сбалансированные деревья в данном случае имеют один большой недостаток – достаточная трудоемкость перестроения. Также существуют системы, у которых сильно ограничены ресурсы, в связи с этим в них должны использоваться наиболее подходящие структуры хранения данных. Разумеется, вывод об оптимальном контейнере хранения данных можно сделать на основе анализа работы системы. Но в таком случае встает вопрос, что будет, если возникнет ситуация, при которой данный контейнер окажется не оптимальным?

Целью данной работы является расширение существующего экспериментального программного обеспечения для увеличения возможностей по анализу различных ситуаций. Адаптивный выбор структуры хранения данных позволит системе выбирать оптимальный контейнер на основании статистики запросов к контейнеру данных.

1. Постановка задачи

На основе существующего решения по анализу эффективности контейнеров выполнить следующие действия:

1. Провести рефакторинг приложения.
2. Реализовать дополнительные типы контейнеров:

* хэш-таблицы;
* кеширующих контейнеры;
* адаптивные контейнеры с линейным классификатором.

2. Анализ задачи

2.1. Механизм анализа контейнеров

2.1.1. Модель нагрузки

Нагрузка представляет собой различные последовательности операций вставки, выборки и удалениях[6, 8]. Существует 6 видов нагрузки (рис. 2.1):

Рис. 2.1. Схема видов нагрузки

* вставка (insert) – набор операций вставки данных в хранилище;
* выборка (select) – набор операций выборки данных;
* удаление(remove) – набор операций удаления данных;
* существующие данные (existing data) – некоторая преднагрузка на хранилище;
* последовательность (sequence) – последовательное выполнение всех внутренних нагрузок;
* блок (block) – случайный порядок выполнения внутренних нагрузок.

Каждая нагрузка имеет ряд параметров:

* сount – количество выполняемых операций;
* min и max – диапазон значений элементов. Чем меньше диапазон, тем больше вероятность наличия двух и более одинаковых команд;
* alias – псевдоним нагрузки. Данный параметр необходим для переиспользования значений данной нагрузки;
* from – указатель на псевдоним другой нагрузки;
* label – маркер нагрузки. Позволяет помечать нагрузку, что может быть полезно для подсчёта промежуточного количества операций.

Все базовые нагрузки обязательно должны содержаться в какой-либо структурной нагрузке. Вложенность структурных нагрузок неограниченна. Нагрузку такой структуры проще всего анализировать с помощью рекурсивных алгоритмов.

2.1.2. Примеры моделирования нагрузки

Поскольку нагрузка имеет древовидную структуру, то моделировать её удобно с помощью XML, где именем тега будет являться тип нагрузки, атрибуты – параметрами тега, а вложенные теги – вложенные нагрузки.

**Пример 1.** Обычная работа с заданным соотношением типов операций. Данная модель в виде xml-структуры может быть представлена следующим образом:

<block>

<insert from=”0” to=”10000” count=”300”/>

<delete from=”0” to=”10000” count=”200”/>

<select from=”0” to=”10000” count=”100”/>

</block>

**Пример 2.** Обычная работа с первоначальным накоплением и заданным соотношением типов операций:

<sequence>

<insert from=”0” to=”10000” count=”1000”/>

<block>

<insert from=”0” to=”10000” count=”300”/>

<delete from=”0” to=”10000” count=”200”/>

<select from=”0” to=”10000” count=”100”/>

</block>

</sequence>

**Пример 3.** Чтение с заданным соотношением удачного и неудачного поиска:

<sequence>

<insert alias=”A” from=”0” to=”10000” count=”1000”/>

<block>

<select from=”0” to=”10000” count=”100” from =”A”/>

<select from=”0” to=”10000” count=”200” not\_from =”A”/>

</block>

</sequence>

2.1.3. Методы индексации данных

В качестве исследуемых методов индексации, выбраны такие распространенные структуры, как массивы, списки, двоичные деревья, хеш-таблицы [1, 2, 3].

Каждый метод индексации данных должен предоставлять базовый интерфейс для работы с контейнером. Такой интерфейс должен содержать три базовых операции:

1. Поиск (Select). Операция чтения данных из контейнера.
2. Вставка (Insert). Операция вставки данных.
3. Удаление (Remove). Операция удаления данных.

Каждая операция получает на вход данные, которые необходимо вставить в контейнер: в данном случае числовой идентификатор.

2.1.4. Способ оценки эффективности работы контейнеров

Для того чтобы оценить какой-либо из методов хранения данных, необходимо определить скорость выполнения операций нагрузки над контейнером.

В общем случае можно выделить следующие простейшие операции, из которых будет состоять всякое действие, производимое над контейнером:

* assign – вставка элемента в хранилище, а также его перемещение;
* compare – сравнение двух элементов;
* hash – вычисление хеш-функции или классифицирующей функции.

И аналогичные операции, применяемые для кэшей или адаптивных контейнеров:

* deferred\_assing – отложенная/асинхронная вставка элемента в контейнер
* deferred\_compare – отложенное/асинхронное сравнение элементов;
* deferred\_hash – отложенное/асинхронное вычисление хеш-функции или классифицирующей функции.

На основании подсчёта этих операций можно сделать вывод об оптимальной структуре хранения данных: наиболее оптимальной структурой будет являться, та структура, у которой сумма этих операций (с учётом веса каждой операции для каждого конкретного контейнера) будет минимальной.

Для получения более точных результатов необходимо провести некоторое количество вычислительных экспериментов с разной нагрузкой на разные контейнеры данных и собрать статистику по данным экспериментам. Такая статистика должна представлять собой совокупность записей следующего вида:

* нагрузка;
* контейнеры;
* производительность контейнеров.

2.1.5. Общие замечания о механизме тестирования

В данной дипломной работе все рассматриваемые хранилища имеют следующие параметры:

* тип хранимых данных – целочисленное со знаком;
* способ хранения данных – все данные хранятся по значению;
* изменяемость данных – предполагается, что все значения в контейнере не являются изменяемыми;
* уникальность данных – каждое значение в контейнере является уникальным и не повторяется;
* все операции для всех контейнеров имеют равный вес: 1 для «обычных» операций и 0 для отложенных

Кроме того будем полагать что контейнер на момент начала тестирования пуст. Эти факторы в той или иной мере будут влиять на итоговые результаты работы.

2.2. Рефакторинг

Существующая система анализа эффективности контейнеров позволяет сравнивать контейнеры на основании нагрузки, состоящей из вставок (*Insert*), выборок элемента (*Select*) и удаления (*Remove*) на основании количества сравнений (*Compare*) и вставок элементов (*Assign*), которые требуются для выполнения заданной нагрузки. Проект содержит также несколько уже реализованных контейнером:

* простой массив (*SimpleArray*);
* простой список (*SimpleList*);
* отсортированный массив (*SortedList*);
* отсортированный список (*SortedArray*);
* двоичное дерево (*BinaryTree*);

С другой стороны проект имеет ряд недостатков:

* жёсткая привязка обработки xml-файла к выводу – то есть исходная реализация предполагала единственный способ анализа данных, через xml-файл и на данной базе невозможно задавать альтернативный источник нагрузки;
* неудачный механизм создания и обработки нагрузки – представлял собой рекурсивную структуру без явной типизации видов нагрузки, а кроме того жёстко привязывался к классу, отвечающему за вывод результатов нагрузки;
* ошибки и недочёты при реализации хранилищ – например, для деревьев использовался рекурсивный алгоритм поиска, который мог приводить к переполнению стека;
* отсутствие механизма «вложенных контейнеров» – отсутствие возврата результата выполнения операций get, set и remove не позволяло реализовать взаимодействие между контейнерами.

Целью рефакторинга является избавление от этих недостатков, а также обеспечение большей гибкости и модульности проекта.

2.3. Хеш-таблицы

Хэш-таблица – хранилище данных, реализующая интерфейс ассоциативного массива. Хеш-таблицы позволяют хранить данные в виде пары: ключ-значение, и выполнять операции вставки, поиска и удаления элемента по ключу. В лучшем случае любая из этих операций может быть выполнена за операций, а в худшем, за .

Для генерации ключа используется хеш-функция. Хеш-функция преобразует данные произвольной длины в последовательность битов фиксированной длины.

Для хеш-функций справедливы следующие утверждения. Пусть ‑ некоторая хеш-функция, а и – элементы, тогда:

* если , то ;
* если , то может быть не равно .

Второй случай, когда для разных элементов результат выполнения хеш-функции одинаков, называется коллизией.

Хорошая хеш-функция должна удовлетворять двум свойствам:

* быстро вычисляться;
* минимизировать количество коллизий.

Для хеш-таблиц существуют два основных способа разрешать коллизии:

* открытая адресация;
* метод цепочек.

Метод открытой адресации проверяет ячейки хеш-таблицы в некотором порядке до тех пор, пока не найдётся свободная ячейка (последовательность проб).

Метод цепочек предполагает, что каждая ячейка хеш-таблицы представляет собой некоторое хранилище данных, в которое и производится вставка элемента. Соответственно для данного способа разрешения коллизий необходимо кроме параметров самой таблицы указывать еще и параметры и контейнера, разрешающего коллизии.

Как уже упоминалось, основным достоинством хеш-таблицы является время доступа к элементам ‑ при некоторых разумных допущениях.

К недостаткам хеш-таблицы можно отнести следующее:

* при увеличении количества хранимых элементов, требуется перехеширование (хотя в случае метода цепочек, перехеширование не является обязательным);
* отсутствие упорядоченности элементов таблице;
* сложность поиска минимального и максимального элемента в таблице.

2.4. Кэширование

Кэш – промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Доступ к данным в кэше осуществляется быстрее, чем выборка данных из более медленной памяти или удаленного источника, однако объём кэша существенно ограничен по сравнению с хранилищем исходных данных. Из-за ограничения на объём «быстрой» памяти и как следствие размеров кэша, кэш является лишь вспомогательной структурой данных и требует обязательного задания контейнера, для которого будет он использоваться.

Фактически кэш является такой же структурой данных, как и хранилище, но имеет ограниченный объём. Соответственно при каждом обращении к хранилищу данных сначала проверяется кэш, а лишь затем происходит обращение к хранилищу данных.

Поскольку для кэши имеют сильно ограниченный объём, то возникает необходимость вытеснять из кэша те элементы, которые менее всего могут понадобиться в будущем. В рамках данной дипломной работы рассматриваются следующие алгоритмы вытеснения:

* LRU (*Least Recently Used*) – алгоритм при котором из кэша вытесняются данный, которые не использовались дольше всего;
* MRU (*Most Recently Used*) – алгоритм при котором из кэша вытесняется последний используемый буфер;
* LFU (*Least Frequently Used)* – алгоритм, при котором из кэша вытесняется элемент использованный реже всех.

Использование кэшей даёт однозначный выигрыш в производительности при чтении данных. С другой стороны, при записи данных выигрыш в производительности можно получить только путём снижения надёжности, с использованием механизма отложенной записи, когда данные переносятся в контейнер из кэша не сразу, а по таймеру или по запросу. В таком случае возможна ситуация, когда запрос к данным хранящимся на данный момент только в кэше придёт в контейнер.

2.5. Адаптивные контейнеры данных

Адаптивный контейнер – составной контейнер данных, совмещающий в себе два или более контейнера (один активный контейнер, а остальные пассивные). При выполнении операций над такими контейнерами данных, в первую очередь производится операция над активным контейнером, а затем, в случае необходимости, аналогичные операции асинхронно выполняются для остальных контейнеров. Далее будут рассматриваться адаптивные контейнеры, содержащие в себе два внутренних контейнера – один активный и один пассивный.

Также такой контейнер содержит журнал нагрузки, в котором хранятся информация о последних операциях. Эти данные необходимы для выбора активного контейнера данных, с использованием *классифицирующей функции*. Классифицирующую функцию можно определить следующим образом: пусть есть два контейнера и , а также некоторая нагрузка , в рамках которой в контейнер производятся вставок, выборок элементов и удалений элементов. Тогда классифицирующей функцией будет называться такая функция , что , если нагрузка требует меньше операций в контейнере , чем в , и в обратном случае. Стоит, однако, заметить, что найти такую функцию , которая при любой нагрузке из произвольных элементов и различной заполненности контейнеров, позволяла бы точно определить какой из контейнеров лучше для заданной нагрузки, едва ли возможно. Тем не менее, для сравнения двух контейнеров, при некоторых допущениях, может быть достаточно линейной функции вида:

.

В целом механизм работы адаптивного контейнеров , содержащего два контейнера и с функцией адаптивности , принимающей значения большие 0, когда эффективнее . можно описать следующим алгоритмом:

1. Инициализировать контейнер, выбрав из двух контейнеров один активный, и положив , где , если активным выбран контейнер и , если активным выбран .
2. Применять операции к активному контейнеру, при необходимости асинхронно дублируя операции в пассивный.
3. Каждую операцию записывать в специальный журнал.
4. Как только в журнал помещено очередные записей, вычислить значение классифицирующей функции .
5. Если < 0, то дождаться завершения всех асинхронных операций, проводимых над пассивным контейнером, и поменять его местами с активным.
6. Вернуться к п. 2.

Вместо проверок с определённым шагом, можно использовать проверку по таймеру или уровню загруженности контейнера операциями.

Кроме того в таком контейнере может быть реализован механизм переключения активного контейнеру по какому либо дополнительному триггеру, например если одним из контейнеров является хеш-таблица и необходимо выполнить её перехеширование.

Таким образом адаптивный контейнер данных в общем случае может работать быстрее чем каждое из составляющих его контейнеров по отдельности и компенсировать недостатки того или иного контейнера. Тем не менее, у таких контейнеров есть ряд недостатков:

* изменение характера нагрузки с шагом близким , нивелирует прирост в скорости работы контейнера;
* при большом количестве параллельных запросов к хранилищу придётся дожидаться завершения большого числа асинхронных запросов к пассивному контейнеру перед сменой, либо жертвовать надёжностью ради скорости выполнения операций;
* дополнительные расходы памяти, поскольку необходимо поддерживать одновременно несколько контейнеров.

3. Средства реализации

Для реализации использовались следующие инструментальные средства и технологии:

* язык программирования Java 8 [10];
* IDE IntelliJ IDEA 14.1.1 Community Edition;
* библиотека для работы с xml-файлами jdom2;
* библиотека для логгирования log4j.

Данный выбор обусловлен, в том числе, использующимися в существующем проекте технологиями.

4. Требования к аппаратному и программному обеспечению

Для функционирования программного обеспечения необходимо выполнение следующих требований к аппаратному и программному обеспечению:

* процессор с частотой не менее 2.6 ГГц;
* оперативная память размером не менее 1024 Мб;
* не менее 20 Мб свободного дискового пространства;
* виртуальная машина Java версии 1.8.

5. Интерфейс пользователя

5.1. Команды консольного интерфейса

Приложение имеет консольный интерфейс. Все операции над ним задаются путём использования следующих аргументов при запуске:

* -if <имя файла> – задаёт имя xml-файла, из которого читается нагрузка. Данный параметр является обязательным;
* -of <имя файла> – задаёт имя выходного csv-файла. Если запустить приложение без этого ключа, то результаты будут выведены на консоль;
* -с – при использовании с предыдущим ключом дублирует выходные данные на консоль;
* -e – вывод с раздельным показом числа сравнений, вставок и вычислений.

5.2. Входные файлы

В качестве входных данных приложения используются XML-файлы. Корневым элементом для файла должен являться тег <burden>, внутри которого должна содержаться информация о тестировании, разделённая на три блока.

5.2.1 Нагрузка

Информация о нагрузке задаётся блоком c тегом <data>. Данный блок является обязательным и должен содержать один из тегов:

* <sequence> – для последовательного выполнения вложенной нагрузки;
* <block> – для выполнения вложенной нагрузки в случайном порядке.

Каждый из этих тегов обязательно должен содержать один или более вложенных тегов, в качестве которых могут выступать либо <block> с <sequence>, либо следующие простые виды нагрузки:

* <insert> – вставка;
* <select> – выборка;
* <remove> – удаление.

Параметры нагрузки описываются в виде атрибутов тега. Допустимы следующие атрибуты:

* min – минимальное значение для генерируемой операции;
* max – максимальное значение для генерируемой операции;
* label – метка данной нагрузки;
* from – альтернативный источник данных для нагрузки. Если задан данный параметр, то значения min и max будут проигнорированы.

Для параметров min и max допустимо задание не только числом, но и строкой, обозначающей название параметра, из которого будет браться значение. В этом случае имя параметра должно быть заключено в символы «%».

5.2.2. Контейнеры

<storages> – блок, описывающий контейнеры, на которые будет проводиться нагрузка. Данный блок также должен обязательно присутствовать в XML-файле. Контейнер данных описывается тегом <storage>, который должен иметь обязательный атрибут class, задающий класс контейнера данных. Следующие теги расположенные внутри тега storage задают различные параметры контейнеров и кэшей:

* <table\_size> c атрибутом value задаёт размер хеш-таблицы;
* <load\_factor> с атрибутом value задаёт коэффициент заполнения хеш-таблицы, по превышению которого будет необходимо выполнить перехеширование;
* <collision\_resolution> c атрибутом class задаёт класс контейнера для разрешения коллизий;
* <cache> с атрибутом class задаёт кэш контейнера;
* <algorithm> с атрибутом value задаёт алгоритм вытеснения элементов из кэша;
* <cache\_size> с атрибутом value задаёт размер кэша;
* <inner\_storages> с атрибутами active и passive задаёт классы активного и пассивного контейнеров для адаптивных контейнеров;
* <classifier> с атрибутами a, b, c, d – задаёт параметры линейного классификатора. Атрибут step, задаёт с каким шагом должно выполняться вычисление данной функции;
* <journal> с атрибутом size задаёт размер журнала.

5.2.3. Параметры

Все параметры нагрузки задаются блоком с тегом <properties>. Данный блок не является обязательным и может отсутствовать. Внутри данного блока располагаются блоки с тегом <property>. Атрибут name данного тега задаёт имя параметра, а вложенные блоки с тегом value значения этих параметров.

Пример XML-файла показан в приложении 1.

5.3. Вывод результатов

На рис 5.1 показан вывод результатов на консоль. Здесь число без скобок обозначает суммарное число «обычных» операций сравнений, присваиваний и вычислений, а значение в скобках число «отложенных» операций. Значение «Performance» есть отношение суммарного числа операций (без учета отложенных операций) первого тестируемого контейнера к текущему выводимому. То есть за 100% считается производительность первого контейнера данных.

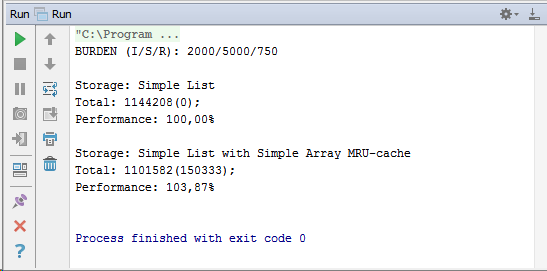


Рис. 5.1. Пример консольного вывода

Пример выходных результатов в CSV-файле представлен на рисунке 5.2.

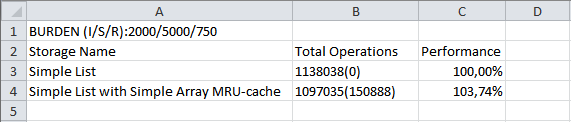


Рис. 5.2. Вывод в CVS-файл

6. Реализация

6.1. Рефакторинг

6.1.1. Счётчики операций

Для подсчёта количества операций в контейнере используется класс SimpleCounterSet. Его диаграмма представлена на рисунке 6.1:

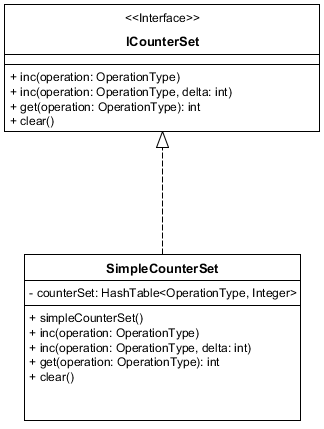


Рис. 6.1. Счётчики операций.

Счётчики операций хранятся в хеш-таблице, в качестве ключей которой выступают элементы перечисления OperationType. Всего таких элементов 6:

* ASSIGN – присваивание/вставка элемента на позицию в контейнер;
* COMPARE – сравнение двух элементов между собой;
* HASH – вычисление хеш-функции;
* DEFERRED\_ASSIGN – отложенная вставка элемента (например, в кэш);
* DEFERRED\_COMPARE – отложенное сравнение;
* DEFERRED\_HASH – отложенное вычисление хеш-функции.

Единственное отличие данного блока от исходного проекта в добавленных элементах перечисления. Изначально оценка проводилась только по ASSIGN и COMPARE.

6.1.2. Контейнеры данных

Все контейнеры данных, которые должны быть протестированы, реализуют интерфейс IDataStorage extends IDataSet. График классов до рефакторинга (на примере массива SimpleArray) представлен на рисунке 6.2:

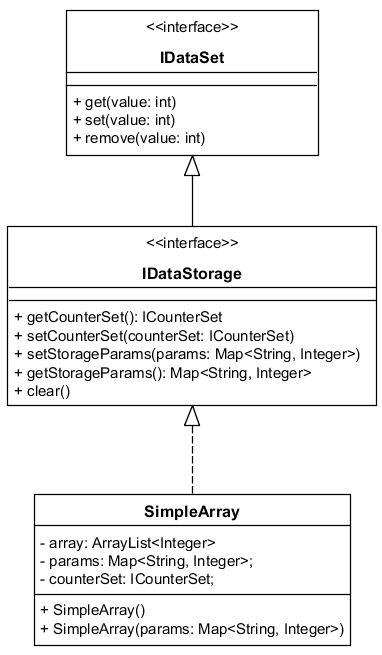


Рис. 6.2. Реализация контейнеров в оригинальном проекте

В процессе переработки были сделаны следующие изменения:

1. Интерфейс IDataSet переименован в IDataContainer. Для методов данного интерфейса добавлено возвращаемое значение. Данное изменение позволило реализовать взаимодействие контейнеров между собой в адаптивных контейнерах и кэшах.
2. В интерфейс IDataContainer добавлен метод void uncheckedSet(int value). Данный метод будет добавлять элемент без проверки существования такого элемента в контейнере, в то время как метод boolean set(int value) должен проверять уникальность элементов перед вставкой.
3. В интерфейс IDataContainer добавлен метод Iterator<Integer> getIterator() – позволяющий осуществлять перебор элементов в контейнере.
4. В интерфейс IDataStorage был добавлен метод String getStorageName(), возвращающий название текущего контейнера.
5. Был добавлен промежуточный абстрактный класс AbstractStorage, в который были перенесен счётчик, а также его геттер и сеттер.
6. Частично переработаны сами контейнеры: поправлены некоторые ошибки, добавлена проверка на уникальность элементов. Для отсортированного массива реализован двоичный поиск.

6.1.3. Нагрузка на контейнер

В исходном проекте описание нагрузки хранилось в классе Burden, а за выполнение нагрузки отвечал класс DataStreamImpl, реализующий интерфейс IDataSet. Кроме того DataStreamImpl содержал метод flush, выводящий результирующую информацию о результатах эмуляции нагрузки.

После рефакторинга два данных класса были заменены на большее количество узкоспециализированных классов.

За простейшую операцию отвечают теперь классы реализующие интерфейс Icommand. Всякая команда, реализующая данный интерфейс, должна реализовывать следующие два метода:

* void execute(IdataStorage storage) – метод, выполняющий над хранилищем, передаваемым в качестве параметра, некоторое действие;
* void printToStream (IdataStream stream) – печать команды в поток.

На базе данного интерфейса реализуются классы, представляющие собой составные элементы нагрузки (рис. 6.3):

* InsertCommand – вставка элемента в контейнер;
* SelectCommand – выборка элемента из контейнера;
* RemoveCommand – удаление элемента из контейнера.

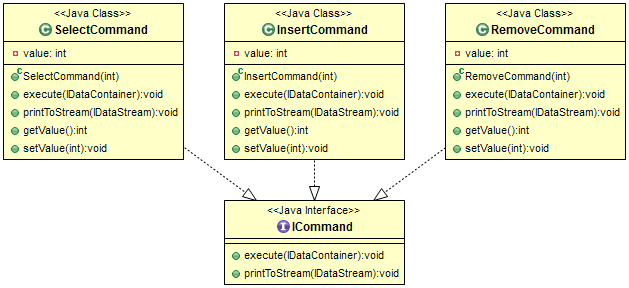


Рис. 6.3. Уровень команд

Следующий уровень – контейнеры команд. Контейнеры команд – классы, реализующие в интерфейс IcommandSource. Данный интерфейс содержит следующие методы:

* Icommand next() – возвращают следующую команду из контейнера;
* void restart() – сбрасывает счётчик команд;
* void printToStream(IdataStream stream) – печать команд содержащихся в потоке в контейнер;
* void setParameters(Map<String, String> setParameters) – устанавливает параметры контейнера команд.

Контейнеры команд можно разделить на два типа:

* простые – контейнеры, содержащие команды только одного типа (Select, Insert, Remove);
* составные – которые, соответственно, могут содержать команды разных типов (Block, Sequence).

Для каждого из этих типов контейнеров общая реализация перенесена в соответствующий абстрактный класс. Кроме того, данные классы имеют конструктор, позволяющий принимать элемент xml-таблицы вида:

<insert alias=”Ins1” count=”3” min=”%min%” max=”%max%”/>,

а также параметры, и по ней генерировать нагрузку.

Значения для команд генерируются с помощью класса StandartRandomValueSet, реализующего абстрактный класс BaseValueSet. Данный класс содержит следующие методы:

* int generateRandom() – возвращает случайное число;
* int generateRandomAndStore() – генерирует случайное число и сохраняет его для последующего переиспользования;
* boolean hasStoredValues() – служит для проверки наличия сохранённых значений;
* Integer reuseRandom() – возвращает случайное число из тех, что ранее были сохранены или null, если таких значений нет;
* void clearStoredValues() – очищает список сохранённых чисел.

Наконец за применение нагрузки к контейнерам отвечает класс DataSetPlayer. Данный класс инициализируется некоторой нагрузкой (объектом IcommandSource) и имеет следующие методы:

* void play(List<IdataStorage> storages) – применяет нагрузку к контейнерам;
* void printToStream(IdataStream stream) – выводит нагрузку в поток;
* void setParameters(Map<String, Integer> parameters) – устанавливает параметры нагрузки.

6.1.4. Обработка XML-файлов

За обработку XML-файлов отвечает класс XMLLoader, используемого взамен класса ProcessLoadXML. ProcessLoadXML отвечал за полный цикл работы приложения, начиная от парсинга XML-файла и заканчивая выводом результатов, путём взаимодействия с классом DataStreamImpl. За всю эту работу отвечал метод void processXML(String filename). Новый класс выполняет лишь проверку на существование файла и определяет какой класс будет заниматься конструированием того или элемента из данного узла XML-файла:

* конструированием нагрузки занимаются классы, наследники ICommandSource;
* для создания контейнеров данных используется «фабрика» контейнеров StorageGenerator;
* для генерации параметров используется класс Parameter, который парсит узел XML-файла в карту вида Map<String, List<Integer>>, и позволяет создать объект класса Iterator<Map<String, Integer>> для перебора параметров.

Перенос обработки различных частей XML-файла в сами классы, представляемые данным элементом, позволило упростить обработку узлов.

После обработки файла все данные сохраняются во внутренние переменные, которые можно достать, используя соответствующие геттеры.

6.1.5. Вывод результатов

За вывод результатов в файл отвечают классы, реализующие интерфейс IResultPrinter. Данный интерфейс имеет два метода:

* void printResult(List<IDataStorages> storages, boolean showDetails) – выводит информацию о результатах нагрузки применённых к контейнерам storages. Флаг showDetails указывает необходимо ли выводить информацию по операциям раздельно.
* void printResult(List<IDataStorages> storages, ICommandSource burden, boolean showDetails) – работает аналогично предыдущему методу, но выводит еще и информацию о нагрузке.

Данный интерфейс реализуют два класса: ConsoleResultPrinter и CSVResultPrinter, отвечающие за вывод результатов, соответственно, на консоль и в CSV-файл.

UML-диаграмма классов, полученная после проведения рефакторинга, представлена на рисунке 6.4:

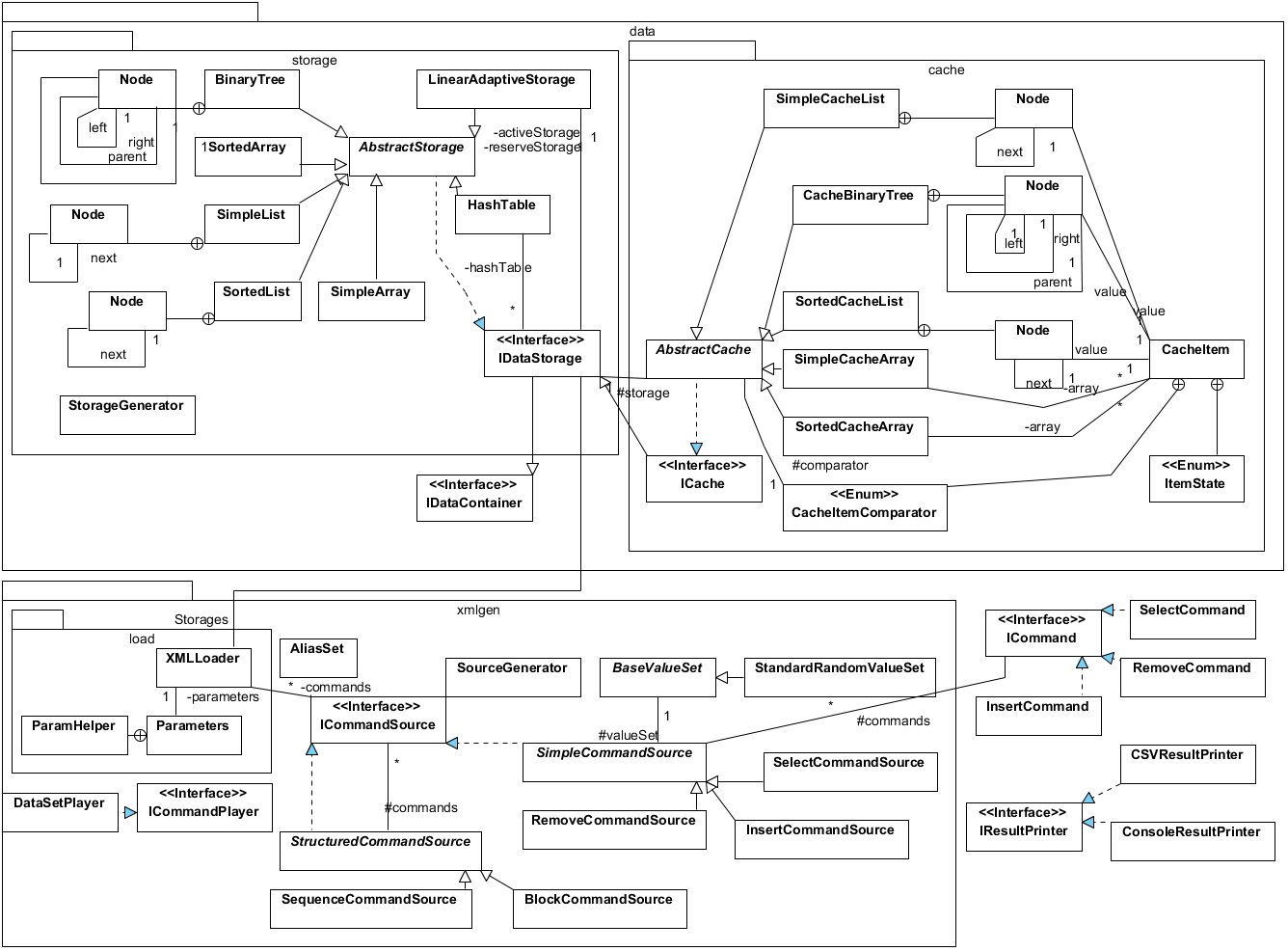


Рис. 6.4. UML-диаграмма классов

6.2. Хеш-таблицы

Хеш-таблицы в рамках данной реализации представляют собой массив контейнеров – наследников класса IDataStorage. По умолчанию в качестве внутреннего контейнера используется класс SimpleArray для разрешения коллизий методом цепочек. Хеш-таблица имеет следующие параметры:

* double loadFactor – параметр, коэффициент загрузки контейнера, по превышении которого будет производиться перехеширование. По умолчанию данное значение равно 0, то есть перехеширование не будет производиться при любом количестве хранимых элементов, что допустимо при разрешении коллизий методом цепочек;
* int initialCapacity – размер контейнера при инициализации. Данный параметр задаёт стартовый размер хеш-таблицы. Значение по умолчанию – 100;
* String storageClass – строка-имя класса используемого при коллизиях.

Также данный класс имеет следующие методы (кроме тех, что реализуют интерфейс IDataStorage):

* private boolean needRehash() – проверяет необходимо ли перехеширование хеш-таблицы;
* private void rehash() – выполняет перехеширование.

6.3. Кэширование

Кэш представляет собой контейнер данных ограниченного размера, реализующий интерфейс ICache, расширяющий IDataStorage. В качестве хранимых элементов кэш использует структуру данных CacheItem, содержащую следующие поля:

* int value – значение, хранимое в элементе кэша;
* int usageCount – количество использований элемента кэша (используется в LFU алгоритме);
* long usageTime – временная метка последнего использования кэша;
* ItemState itemState – состояние элемента, которое может принимать два значения: NORMAL – значение по умолчанию и DELETED для пометки удалённых из контейнера элементов.

Для сравнения элементов используется метод int compare(CacheItem i1, CacheItem i2), перечисления CacheItemComparator.

Кэш реализуется аналогично обычным контейнерам, но имеет ряд дополнительные полей и методов:

* int cacheSize – размер кэша;
* IDataStorage storage – внутренний контейнер данных, то есть всякое обращение к контейнеру происходит через обращение к кэшу;
* CacheItemComparator comparator – механизм вытеснения элементов из кэша;
* Thread waitingThread – поток, выполняющий изменения кэша;
* void clearCache() – очищает кэш. Метод void clear() очищает и кэш и контейнер.

Еще одним важным отличием кэша является тот факт, что все операции над ним считаются «отложенными», поскольку кэш располагается, как правило, в более быстрой памяти и время, затрачиваемое на одну операцию, гораздо меньше, чем в основном контейнере.

Кэши на основе SimpleArray и SimpleList хранят элементы, отсортированные в соответствии с алгоритмом вытеснения, что позволяет добавлять и удалять элементы из кэша за O(1) операций. Остальные виды кэша реализуют дополнительный метод, позволяющий искать следующий элемент для удаления.

6.4. Адаптивные контейнеры данных

Адаптивный контейнер данных представляет собой контейнер, в котором содержатся два контейнера, а также классифицирующая функция. Оно реализуется классом LinearAdaptiveStorage extends AbstractStorage и содержит следующие поля и методы:

* IDataStorage activeStorage – текущий активный контейнер;
* IDataStorage reserveStorage – текущий пассивный контейнер;
* ICounterSet activeOperations – счётчик команд для активного контейнера;
* ICounterSet deferredOperations – счётчик команд отложенных операций (выполняемых над текущим пассивным контейнером). Наличие двух счётчиков необходимо для распараллеливания операций и упрощения синхронизации;
* double[] isrCoeffs – коэффициенты для вычисления классифицирующей функции;
* boolean isNegativeResult – флаг отмечающий знак предыдущего вычисления классифицирующей функции;
* int calculationsStep – определяет через сколько операций должно происходить вычисление адаптирующей функции;
* final Map<Сharacter, Integer> operationHistory – журнал операций трех типов: вставки – I, выборки – S, удаления – R;
* int storedOperationsCount – количество операций, сохранённых в журнале;
* final List<Thread> deferredOperationsList – список асинхронных операций над пассивным контейнером;
* private boolean needToSwapStorage() – проверка на необходимость смены активного хранилища;
* private void updateStorage() – меняет местами активный контейнер с пассивным в случае необходимости.

Порядок обновления активного контейнера в реализации следующий:

1. Проверяем количество операций в журнале (storedOperationsCount).
2. Если storedOperationsCount < calculationsStep, то выходим.
3. Дожидаемся завершения всех асинхронных операций и очищаем их список.
4. Вычисляем значение классифицирующей функции.
5. Если значение функции не изменило знак или результат вычисления достаточно близок к 0, то выходим.
6. Меняем местами активный контейнер с пассивным.

7. План тестирования

7.1. Тестирование хеш-таблицы

В лучшем случае хеш-таблицы позволяют выполнять операции вставки, выборки и удаления за O(1) операций. Для тестирования корректности работы хеш-таблицы выполним ставку малого числа случайных элементов в хеш-таблицу большого исходного размера. А затем большого числа случайных элементов в хеш-таблицу малого размера.

Параметры первого теста:

* нагрузка (вставки/выборки/удаления) – 100/0/0;
* используемые контейнеры – HashTable и SimpleList;
* размер хеш-таблицы – 1000 элементов;
* контейнер для разрешения коллизий SimpleList;
* перехеширование отключено.

Результаты тестирования представлены на рисунке 7.1:

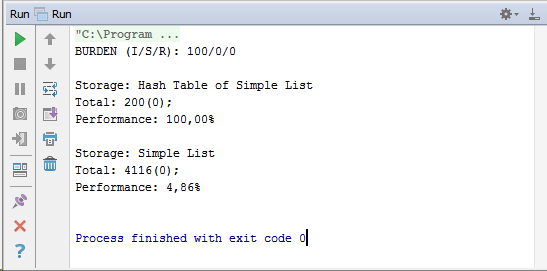


Рис. 7.1. Первый тест хеш-таблицы

Параметры первого теста:

* нагрузка – 5000/0/0;
* используемые контейнеры – HashTable и SimpleList;
* размер хеш-таблицы – 5 элементов;
* контейнер для разрешения коллизий SimpleList;
* перехеширование отключено.

Результаты тестирования представлены на рисунке 7.2:

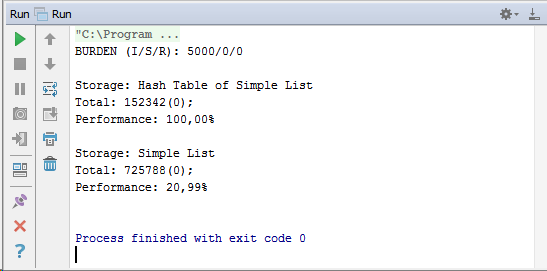


Рис. 7.2. Второй тест хеш-таблицы

7.2. Тестирование контейнеров с кэшами

Контейнеры с кэшем дают прирост в скорости чтения данных. Кэширование данных будет тестироваться на примере кэша на основе SimpleList’а, поскольку в первую очередь интересна производительность контейнеров, а не кэшей. Кроме того SimpleList на малых размерах кэшей показывает хорошую производительность за счёт того что вытеснение элемента происходит за O(1) операций: вытесняемый элемент находится либо в начале, либо в конце списка и вставка также производится либо в начало, либо в конец элемента.

Первый тест:

* нагрузка – 5000/0/0;
* используемые контейнеры: simpleList и SimpleList c SimpleList LRU-кэшем.

Результаты первого теста представлены на рисунке 7.3. Как можно заметить использование кэша очевидно не влияет на прирост в скорости записи, хотя на небольшом диапазоне данных позволит избежать проверок на наличие элемента в основном контейнере.

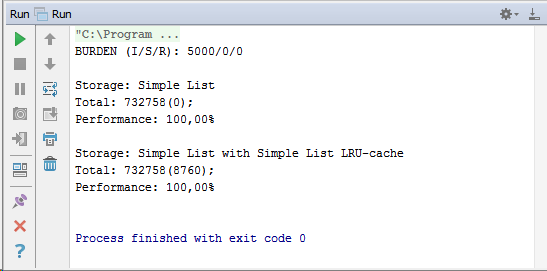
* 

Рис. 7.3. Первый тест кэша

Для второго теста кэшей покажем, что использование кэшей для чтения действительно даёт уменьшение количества операций над основным контейнером. Параметры второго теста:

* нагрузка 1000/20000/0;
* используемые контейнеры: SimpleList и SimpleList с тремя видами кэшей;
* размер кэшей – 25 элементов.

Результаты тестирования представлены на рисунке 7.4:

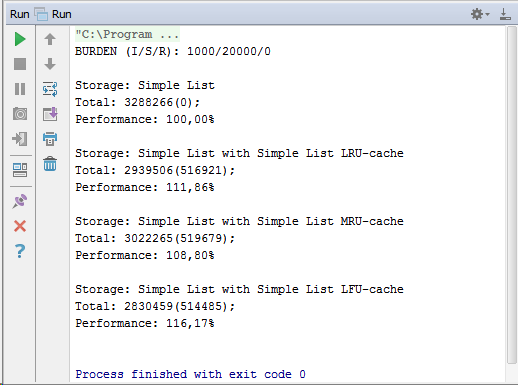


Рис. 7.4. Второй тест кэша

7.3. Тестирование адаптивного контейнера

Для тестирования адаптивных контейнеров данных, в качестве внутренних контейнеров данных возьмём SortedList и SortedArray. Операции поиска в SortedList проводятся за операций, то есть при данной реализации за n операций сравнения будет точно известно, есть ли такой ли элемент в контейнере или нет. Для SortedArray поиск осуществляется за операций при использовании бинарного поиска. В свою очередь добавление и удаление из контейнера производится за меньшее время при использовании SortedList, поскольку не требуется перемещать элементы внутри контейнера. Для упрощения возьмём примитивную классифицирующую функцию , то есть будем использовать SortedArray когда количество выборок элементов больше числа удалений и SortedList в ином случае.

Параметры первого теста:

* нагрузка – 1000/1000/1000;
* используемые контейнеры: SortedArray, SortedList, LinearAdaptiveStorage;
* изначальный активный контейнер SortedArray;
* шаг проверки классифицирующей функции 10.

Результаты тестирования представлены на рисунке 7.5:

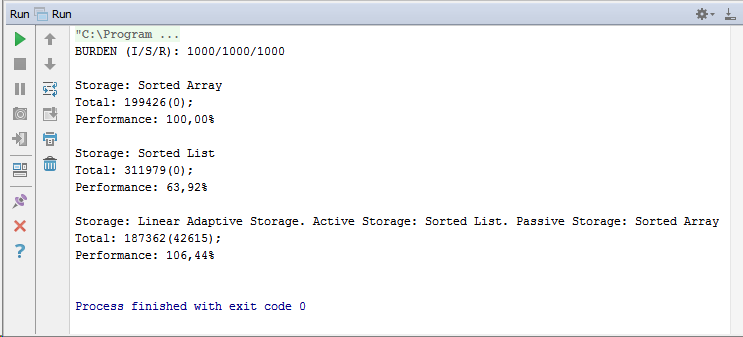


Рис. 7.5. Первый тест адаптивного контейнера

Для второго теста выставим шаг проверки в 1000, то есть наименее удачный с точки зрения нагрузки. Результаты теста на рисунке 7.6:

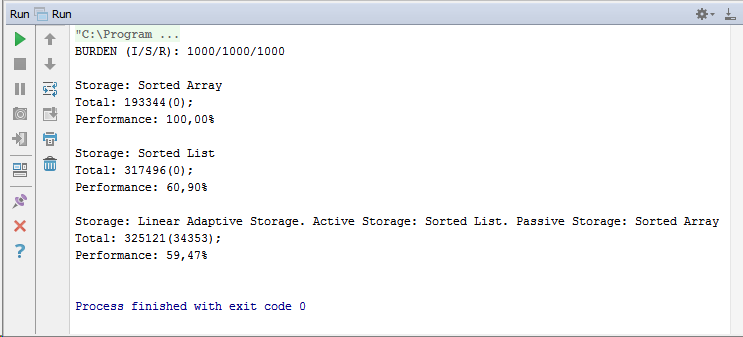


Рис. 7.6. Второй тест адаптивного контейнера

Заключение

На основе существующего решения по анализу эффективности контейнеров были выполнены следующие задачи:

1. Проведён рефакторинг;
2. Реализованы дополнительные типы контейнеров:

* хэш-таблицы;
* кеширующих контейнеры;
* адаптивные контейнеры с линейным классификатором.

Список литературы

1. Алгоритмы: построение и анализ / Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К; под ред. И. В. Красикова – 2-е изд. – Москва : Вильямс, 2005. – 1296 с.
2. Блох Д. Java. Эффективное программирование / Джошуа Блох – Москва : «Лори», 2002. – 220 с.
3. Вирт Н. Алгоритмы и структуры данных. / Н. Вирт – Москва : ДМК Пресс, 2010. – 272 с.
4. Гарсиа-Молина Г. Системы баз данных. Полный курс. / Гектор Гарсиа-Молина, Джеффри Д. Ульман, Дженнифер Уидом – Москва: Вильямс, 2004. – 1088 с.
5. Дейт Крис Дж. Введение в системы баз данных / Крис Дж. Дейт; пер. с анг. К. Птицин – Москва: Вильямс, 2006. – 1328 с.
6. Зобов В.В. Инструмент для моделирования нагрузки на контейнеры данных/ В.В. Зобов, К.Е. Селезнев // Материалы четырнадцатой научн.- метод. конференции «Информатика: проблемы, методология, технологии», 10-11 февраля 2011 г. – Воронеж, 2014. – Т. 3. – с. 154-161.
7. Кнут Д. Искусство программирования, том 1. Основные алгоритмы. / Д. Кнут – 3-е изд. – Москва : Вильямс, 2006. – 720 с.
8. Мартин Дж. Организация баз данных в вычислительных системах / Дж. Мартин – Москва : Мир, 1980. – 664 с.
9. Смит Дж. Мак-Колм Элементарные шаблоны проектирования / Джеймс Мак-Колм Смит – Москва : Вильямс, 2012. – 304 с.
10. Шилдт Г. Java. Полное руководство / Герберт Шилдт – 8-е изд. – Москва : Вильямс, 2012. – 1102 с.
11. Эккель Б. Философия Java / Брюс Эккель; пер. с анг. Е. Матвеев – Санкт-Петербург : Питер, 2009. – 640с.

Приложение 1. Пример входного XML-файла

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<burden>

<data>

<sequence count=*"1"*>

<insert alias=*"Ins1"* count=*"1000"* min=*"%min%"* max=*"%max%"* label=*"test1"*/>

<select count=*"500"* from=*"Ins1"* />

</sequence>

</data>

<storages>

<storage class=*"com.vsu.amm.data.storage.SimpleList"*>

<cache class=*"com.vsu.amm.data.cache.SimpleCacheList"*>

<size value=*"20"* />

<comparator value=*"LRU"* />

</cache>

</storage>

<storage class=*"com.vsu.amm.data.storage.LinearAdaptiveStorage"*>

<active\_storage class=*"com.vsu.amm.data.storage.SimpleList"* />

<passive\_storage class=*"com.vsu.amm.data.storage.SimpleArray"* />

<classificator a=*"10"* b=*"20"* c=*"-46"* d=*"-3"* step=*"50"* />

</storage>

</storages>

<properties>

<property name=*"max"*>

<value>300</value>

<value>400</value>

</property>

<property name=*"min"*>

<value>0</value>

<value>200</value>

</property>

</properties>

</burden>

Приложение 2. Листинг классов

SelectCommandSource.java

package com.vsu.amm.command.xmlgen;  
  
import com.vsu.amm.Constants;  
import com.vsu.amm.Utils;  
import com.vsu.amm.command.SelectCommand;  
  
import org.jdom2.Element;  
  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.Map;  
  
public class SelectCommandSource extends SimpleCommandSource {  
  
 public SelectCommandSource(Element elem, AliasSet aliasSet, Map<String, Integer> params) {  
 if (elem == null)  
 return;  
  
 String from = elem.getAttributeValue("from");  
 if (aliasSet != null)  
 valueSet = aliasSet.getAlias(from);  
  
 boolean reuseValues = valueSet != null;  
  
 Integer tmp;  
  
 if (valueSet == null) {  
  
 String minAttr = elem.getAttributeValue("min");  
 String maxAttr = elem.getAttributeValue("max");  
  
 valueSet = new StandardRandomValueSet(minAttr, maxAttr, params);  
 }  
  
 String alias = elem.getAttributeValue("alias");  
  
 if (!Utils.*isNullOrEmpty*(alias) && aliasSet != null)  
 aliasSet.putAlias(alias, valueSet);  
  
 tmp = getAttributeValue(elem, "count", params);  
 int count = tmp == null ? 1 : tmp;  
  
 commands = new ArrayList<>(count);  
  
 if (reuseValues && valueSet.hasStoredValues())  
 for (int i = 0; i < count; i++)  
 commands.add(new SelectCommand(valueSet.reuseRandom().intValue()));  
 else if (!Utils.*isNullOrEmpty*(alias))  
 for (int i = 0; i < count; i++)  
 commands.add(new SelectCommand(valueSet.generateRandomAndStore()));  
 else  
 for (int i = 0; i < count; i++) {  
 commands.add(new SelectCommand(valueSet.generateRandom()));  
 }  
  
 label = elem.getAttributeValue("label");  
 }  
 public SelectCommandSource(Integer selectCount, AliasSet aliasSet) {  
 if (aliasSet == null)  
 valueSet = new StandardRandomValueSet(0, Constants.*DEFAULT\_MAX\_VALUE*);  
 else  
 valueSet = aliasSet.getAlias(Constants.*DEFAULT\_ALIAS\_NAME*);  
 if (valueSet.hasStoredValues()) {  
  
 for (int i = 0; i < selectCount; i++)  
 commands.add(new SelectCommand(valueSet.reuseRandom()));  
 } else  
 for (int i = 0; i < selectCount; i++)  
 commands.add(new SelectCommand(valueSet.generateRandom()));  
 }  
   
 @Override  
 public Map<String, Integer> getBurdenDescription() {  
 Map<String, Integer> info = new HashMap<>();  
 info.put("Insert", 0);  
 info.put("Select", 0);  
 info.put("Remove", 0);  
 if (commands == null)  
 return info;  
 info.put("Select", commands.size());  
 return info;  
 }  
}

HashTable.java

package com.vsu.amm.data.storage;  
  
import java.util.HashMap;  
import java.util.Iterator;  
import java.util.Map;  
  
import com.vsu.amm.stat.ICounterSet;  
import com.vsu.amm.stat.ICounterSet.OperationType;  
import com.vsu.amm.stat.SimpleCounterSet;  
  
public class HashTable extends AbstractStorage {  
 private static final String *HASH\_TABLE\_STORAGE\_PARAM\_PREFIX* = "storage.";  
 private static final String *HASH\_TABLE\_STORAGE\_CLASS\_PARAM\_NAME* = "class";  
 private static final String *HASH\_TABLE\_SIZE\_PARAM\_NAME* = "hash\_table\_size";  
 private static final String *LOAD\_FACTOR\_PARAM\_NAME* = "load\_factor";  
 private static final int *DEFAULT\_HASH\_TABLE\_SIZE* = 5;  
 private static final double *DEFAULT\_LOAD\_FACTOR* = 0;  
 private static final String *DEFAULT\_STORAGE\_CLASS* = "com.vsu.amm.data.storage.SimpleList";  
 private IDataStorage[] hashTable;  
 private int hashTableSize;  
 private int itemsCount = 0;  
 private double loadFactor;  
 private Map<String, Integer> storageParams;  
 private String storageClass;  
  
  
 private void rehash() {  
 Iterator<Integer> iter = getIterator();  
 int newSize = hashTableSize \* 2 + 1;  
 IDataStorage[] newTable = new IDataStorage[newSize];  
 if (iter != null) {  
 while (iter.hasNext()) {  
 Integer item = iter.next();  
 int hv = Math.*abs*(item.hashCode() % newSize);  
 counterSet.inc(OperationType.*CALCULATION*);  
 if (newTable[hv] == null)  
 newTable[hv] = StorageGenerator.*getDataStorage*(storageClass, storageParams);  
 newTable[hv].uncheckedSet(item);  
 }  
 }  
 hashTable = newTable;  
 hashTableSize = newSize;  
 }  
  
 private boolean needRehash() {  
 if (loadFactor <= 0)  
 return false;  
 return itemsCount \* 1.0 / hashTableSize > loadFactor;  
 }  
  
 public HashTable() {  
 this.hashTableSize = *DEFAULT\_HASH\_TABLE\_SIZE*;  
 this.storageClass = *DEFAULT\_STORAGE\_CLASS*;  
 this.hashTable = new IDataStorage[hashTableSize];  
 }  
   
 @Override  
 public IDataStorage cloneDefault() {  
 HashTable s = new HashTable();  
 s.setCounterSet(new SimpleCounterSet());  
 return s;  
 }  
  
 @Override  
 public void uncheckedSet(int value) {  
 itemsCount++;  
 if (needRehash())  
 rehash();  
  
 counterSet.inc(OperationType.*CALCULATION*);  
 int key = new Integer(value).hashCode() % hashTableSize;  
 counterSet.inc(OperationType.*CALCULATION*);  
 *//элемента не существует в хранилище* if (hashTable[key] == null) {  
 hashTable[key] = StorageGenerator.*getDataStorage*(storageClass, storageParams);  
 hashTable[key].setCounterSet(counterSet);  
 }  
 hashTable[key].uncheckedSet(value);  
  
 }  
  
 @Override  
 public String getStorageName() {  
 return "Hash Table of " + StorageGenerator.*getDataStorage*(storageClass, null).getStorageName();  
 }  
  
 public HashTable(Map<String, Integer> params, String storageClass, Map<String, Integer> storageClassParams) {  
  
 if (params.containsKey(*HASH\_TABLE\_SIZE\_PARAM\_NAME*)) {  
 hashTableSize = params.get(*HASH\_TABLE\_SIZE\_PARAM\_NAME*);  
 } else {  
 hashTableSize = *DEFAULT\_HASH\_TABLE\_SIZE*;  
 }  
  
 if (params.containsKey(*LOAD\_FACTOR\_PARAM\_NAME*)) {  
 loadFactor = params.get(*LOAD\_FACTOR\_PARAM\_NAME*) / 100.0;  
 } else  
 loadFactor = *DEFAULT\_LOAD\_FACTOR*;  
  
 this.storageClass = *DEFAULT\_STORAGE\_CLASS*;  
 this.storageParams = storageClassParams;  
 if (storageClass != null) {  
 try {  
 StorageGenerator.*getDataStorage*(storageClass, storageClassParams);  
 this.storageClass = storageClass;  
 } catch (Exception ignored) {  
 }  
 }  
  
 this.hashTable = new IDataStorage[hashTableSize];  
 }  
  
 @Override  
 public void clear() {  
 super.clear();  
 itemsCount = 0;  
 for (int i = 0; i < hashTableSize; i++)  
 if (hashTable[i] != null)  
 hashTable[i].clear();  
 }  
  
 @Override  
 public boolean get(int value) {  
 int key = new Integer(value).hashCode() % hashTableSize;  
 counterSet.inc(OperationType.*CALCULATION*);  
 if (hashTable[key] == null)  
 return false;  
 return hashTable[key].get(value);  
 }  
  
 @Override  
 public boolean set(int value) {  
 int key = new Integer(value).hashCode() % hashTableSize;  
 counterSet.inc(OperationType.*CALCULATION*);  
 *//элемента не существует в хранилище* if (hashTable[key] == null) {  
 hashTable[key] = StorageGenerator.*getDataStorage*(storageClass, null);  
 hashTable[key].setCounterSet(counterSet);  
 }  
 boolean inserted = hashTable[key].set(value);  
 if (inserted)  
 itemsCount++;  
  
 if (needRehash())  
 rehash();  
  
 return inserted;  
 }  
  
 @Override  
 public boolean remove(int value) {  
 int key = new Integer(value).hashCode() % hashTableSize;  
 counterSet.inc(OperationType.*CALCULATION*);  
 if (hashTable[key] == null)  
 return false;  
 if (hashTable[key].remove(value)) {  
 itemsCount--;  
 return true;  
 }  
 return false;  
 }  
  
 @Override  
 public Iterator<Integer> getIterator() {  
 Iterator<Integer> fstIter = null;  
 int i = 0;  
 while (i < hashTableSize) {  
 if (hashTable[i] != null) {  
 fstIter = hashTable[i].getIterator();  
 break;  
 }  
 i++;  
 }  
 final int fi = i;  
 final Iterator<Integer> fiter = fstIter;  
  
 return new Iterator<Integer>() {  
  
 int index = fi + 1;  
 Iterator<Integer> iter = fiter;  
  
 @Override  
 public boolean hasNext() {  
 if (iter == null)  
 return false;  
  
 if (iter.hasNext())  
 return true;  
 iter = null;  
 while (index < hashTableSize) {  
 if (hashTable[index] != null) {  
 iter = hashTable[index].getIterator();  
 break;  
 }  
 index++;  
 }  
 return iter != null;  
 }  
  
 @Override  
 public Integer next() {  
 if (!hasNext())  
 return null;  
 return iter.next();  
 }  
 };  
 }  
}

SimpleCacheList.java

package com.vsu.amm.data.cache;

import com.vsu.amm.data.cache.CacheBinaryTree.Node;

import com.vsu.amm.data.cache.CacheItem.CacheItemComparator;

import com.vsu.amm.data.cache.CacheItem.ItemState;

import com.vsu.amm.data.storage.IDataStorage;

import com.vsu.amm.stat.ICounterSet.OperationType;

import com.vsu.amm.stat.SimpleCounterSet;

import java.util.Iterator;

import java.util.Map;

/\*\*

\* Кэш на основе связного списка.

\* Элементы в кэше отсортированы в соответствии с политикой замены элемента.

\* Таким образом как только размер кэша начнёт превышать максимальный размер, достаточно будет удалить первые элементы

\* Created by Nikita Skornyakov on 25.05.15

\*/

public class SimpleCacheList extends AbstractCache {

private Node head;

private Node tail;

@Override

public IDataStorage cloneDefault() {

SimpleCacheList s=new SimpleCacheList();

s.setCounterSet(new SimpleCounterSet());

return s;

}

@Override

public String getStorageName() {

String name = "";

if (storage != null)

name = storage.getStorageName() + " with ";

return name + "Simple List " + comparator.toString() + "-cache";

}

@Override

public void uncheckedSet(int value) {

if (storage != null)

storage.uncheckedSet(value);

}

@Override

protected void shrinkCache() {

while (itemsCount > cacheSize && head != null) {

head = head.next;

itemsCount--;

}

}

public SimpleCacheList() {

this.tail = null;

this.head = null;

}

@Override

public void setStorageParams(Map<String, Integer> params) {

Integer size = params.get("size");

if (size != null)

cacheSize = size;

Integer ir = params.get("rate");

if (ir != null)

insertionRate = ir;

if (storage != null)

storage.setStorageParams(params);

}

@Override

public void clear() {

super.clear();

head = null;

}

@Override

public boolean get(int value) {

if (cacheSize <= 0)

return storage != null && storage.get(value);

waitForThread();

Node prev = null;

Node tmp = head;

while (tmp != null) {

counterSet.inc(OperationType.DEFERRED\_COMPARE);

if (tmp.getValue() == value) {

update(prev);

return tmp.value.getState() == ItemState.NORMAL;

}

prev = tmp;

tmp = tmp.next;

}

//элемент не найден

boolean containsInStorage = false;

if (storage != null)

containsInStorage = storage.get(value);

//если элемента в хранилище нет, то выходим.

if (!containsInStorage)

return false;

insertInCache(value);

return true;

}

@Override

public boolean set(int value) {

if (cacheSize <= 0)

return storage != null && storage.set(value);

waitForThread();

waitingThread = new Thread() {

@Override

public void run() {

//кэш пуст

if (head == null)

return;

//в кэше есть элементы, поэтому сначала ищем среди них

if (head.getValue() == value) {

if (head.value.getState() == ItemState.DELETED)

head.value.setState(ItemState.NORMAL);

update(null);

return;

}

Node tmp = head;

while (tmp.next != null) {

counterSet.inc(OperationType.DEFERRED\_COMPARE);

//если элемент найден в кэше

if (tmp.next.getValue() == value) {

if (tmp.next.value.getState() == ItemState.DELETED)

tmp.next.value.setState(ItemState.NORMAL);

counterSet.inc(OperationType.DEFERRED\_ASSIGN);

update(tmp);

return;

}

tmp = tmp.next;

}

}

};

waitingThread.start();

insertInCache(value);

return storage != null && storage.set(value);

}

@Override

public boolean remove(int value) {

waitForThread();

if (head == null)

return storage != null && storage.remove(value);

Node tmp = head;

Node prev = null;

while (tmp != null) {

if (tmp.getValue() == value) {

if (tmp.value.getState() == ItemState.DELETED) {

update(prev);

return true;

}

else {

tmp.value.setState(ItemState.DELETED);

update(prev);

return storage != null && storage.remove(value);

}

}

prev = tmp;

tmp = tmp.next;

}

return storage != null && storage.remove(value);

}

@Override

public Iterator<Integer> getIterator() {

if (storage == null)

return new Iterator<Integer>() {

@Override

public boolean hasNext() {

return false;

}

@Override

public Integer next() {

return null;

}

};

return storage.getIterator();

}

@Override

public void clearCache() {

super.clearCache();

head = null;

tail = null;

}

//вставка элемента которого не существовало в кэше

private void insertInCache(int value) {

if (!shouldValueBeInserted())

return;

waitForThread();

counterSet.inc(OperationType.DEFERRED\_ASSIGN);

//в отдельном потоке вставляем элемент

waitingThread = new Thread() {

@Override

public void run() {

while (cacheSize < itemsCount) {

if (head == tail) {

head = new Node(value);

tail = head;

return;

}

head = head.next;

}

if (head == null) {

counterSet.inc(OperationType.DEFERRED\_ASSIGN);

head = new Node(value);

tail = head;

itemsCount = 1;

return;

}

//вставляем элемент в соотвествии с с политикой записи

//для MRU и LFU в начало

//для LRU в конец

if (comparator == CacheItemComparator.LRU)

tail.next = new Node(value);

else {

Node n = new Node(value);

n.next = head;

head = n;

}

if (cacheSize > itemsCount)

itemsCount++;

}

};

waitingThread.start();

}

/\*\*

\*перемещение узла в соответствии с политикой записи кэша

\*@param prev элемент перед перемещаемым

\*/

private void update (Node prev) {

waitForThread();

waitingThread = new Thread() {

@Override

public void run() {

Node p = prev;

//если элемент один, то не трогаем его

if (head == tail) {

head.value.update();

return;

}

//перемещаем голову

if (p == null) {

head.value.update();

if (comparator == CacheItemComparator.MRU)

return;

if (comparator == CacheItemComparator.LRU) {

Node tmp = head;

tail.next = tmp;

head = head.next;

tmp.next = null;

return;

}

counterSet.inc(OperationType.DEFERRED\_COMPARE);

if (comparator.compare(head.value, head.next.value) < 0)

return;

//вытаскиваем узел из списка

Node tmp = head;

head = head.next;

Node n = head;

//тащим его на свою позицию

while (n.next != null && comparator.compare(tmp.value, n.next.value) > 0) {

counterSet.inc(OperationType.DEFERRED\_COMPARE);

n = n.next;

}

tmp.next = n.next;

n.next = tmp;

return;

}

Node tmp = p.next;

p.next = tmp.next;

//элемент внутри списка

//перемещаем в начало

if (comparator == CacheItemComparator.MRU) {

tmp.next = head;

head = tmp;

return;

}

//перемещаем в конец

if (comparator == CacheItemComparator.LRU) {

tmp.next = null;

tail.next = tmp;

tail = tmp;

return;

}

//перемещаем внутри списка

while (p.next != null && comparator.compare(tmp.value, p.next.value) > 0) {

counterSet.inc(OperationType.DEFERRED\_COMPARE);

p = p.next;

}

tmp.next = p.next;

p.next = tmp;

}

};

waitingThread.start();

}

class Node {

CacheItem value;

Node next;

public Node(int value) {

this.value = new CacheItem(value);

this.next = null;

}

public int getValue() {

return value.getValue();

}

public void setValue(int value) {

this.value.setValue(value);

}

public Node getNext() {

return next;

}

public void setNext(Node next) {

this.next = next;

}

}

}

LinearAdaptiveStorage.java

package com.vsu.amm.data.storage;  
  
import com.vsu.amm.stat.ICounterSet;  
import com.vsu.amm.stat.ICounterSet.OperationType;  
import com.vsu.amm.stat.SimpleCounterSet;  
  
import java.util.\*;  
  
*/\*\*  
 \* Created by Nikita Skornyakov on 30.05.2015.  
 \*/*public class LinearAdaptiveStorage extends AbstractStorage {  
 private IDataStorage activeStorage;  
 private ICounterSet activeOperations = new SimpleCounterSet();  
 private IDataStorage reserveStorage;  
 private ICounterSet deferredOperations = new SimpleCounterSet();  
 private double[] isrCoeffs; *//коэффициенты для расчёта функции эффективности хранилища* boolean isNegativeResult = false;  
 private int calculationsStep; *//как часто должны проводиться вычисления на эффективность хранилища* private final Map<Character, List<Long>> operationHistory = new HashMap<Character, List<Long>>(); *//история операций;* int storedOperationsCount;  
 int journalSize;  
 private final List<Thread> deferredOperationsList = new ArrayList<Thread>(); *//список отложенных операций* public LinearAdaptiveStorage() {  
 operationHistory.put('I', new LinkedList<>());  
 operationHistory.put('S', new LinkedList<>());  
 operationHistory.put('R', new LinkedList<>());  
 }  
  
 private void checkFunc() {  
 if (storedOperationsCount < calculationsStep)  
 return;  
  
 boolean swap = needToSwapStorage();  
 *//очищаем хранилице* if (calculationsStep == journalSize) {  
 operationHistory.get('I').clear();  
 operationHistory.get('S').clear();  
 operationHistory.get('R').clear();  
 } else  
 for (int i = 0; i < calculationsStep; i++) {  
 Character minCh = null;  
 long min = -1;  
 if (operationHistory.get('I').size() > 0) {  
 min = operationHistory.get('I').get(0);  
 minCh = 'I';  
 }  
 if (operationHistory.get('S').size() > 0) {  
 if (min == -1) {  
 min = operationHistory.get('S').get(0);  
 minCh = 'S';  
 } else  
 if (operationHistory.get('S').get(0) < min) {  
 min = operationHistory.get('S').get(0);  
 minCh = 'S';  
 }  
 }  
 if (operationHistory.get('R').size() > 0) {  
 if (min == -1) {  
 min = operationHistory.get('R').get(0);  
 minCh = 'R';  
 } else  
 if (operationHistory.get('R').get(0) < min) {  
 min = operationHistory.get('R').get(0);  
 minCh = 'R';  
 }  
 }  
 if (minCh == null)  
 break;  
 operationHistory.get(minCh).remove(0);  
 }  
 storedOperationsCount = 0;  
  
 if (!swap)  
 return;  
  
 *//дожидаемся окончания выполнения всех отложенных операций* for (Thread t : deferredOperationsList)  
 if (t == null)  
 continue;  
 else  
 synchronized (t) {  
 try {  
 t.join();  
 } catch (Exception ignored) {}  
 }  
 *//очищаем список* deferredOperationsList.clear();  
  
 *//меняем хранилища местами* IDataStorage ref = activeStorage;  
 activeStorage = reserveStorage;  
 reserveStorage = ref;  
 activeStorage.setCounterSet(activeOperations);  
 reserveStorage.setCounterSet(deferredOperations);  
  
 }  
  
 */\*\*  
 \* проверка на необходимость смены активного хранилища  
 \* @return {@code true} если хранилища необходимо поменять местами  
 \*/* private boolean needToSwapStorage() {  
 *//вычисляем линейную функцию* double res = isrCoeffs[0] \* operationHistory.get('I').size() +  
 isrCoeffs[1] \* operationHistory.get('S').size() +  
 isrCoeffs[2] \* operationHistory.get('R').size() +  
 isrCoeffs[3];  
 *//если результат вычислентия достаточно близок к нулю, то смена хранилищ не даст никакой выгоды* if (Math.*abs*(res) < 1e-2)  
 return false;  
  
 if ((res < 0) ^ isNegativeResult) {  
 isNegativeResult = !isNegativeResult;  
 return true;  
 }  
  
 return false;  
  
 }  
  
  
 public IDataStorage getActiveStorage() {  
 return activeStorage;  
 }  
  
 public void setActiveStorage(IDataStorage activeStorage) {  
 this.activeStorage = activeStorage;  
 activeStorage.setCounterSet(activeOperations);  
 }  
  
 public IDataStorage getReserveStorage() {  
 return reserveStorage;  
 }  
  
 public void setReserveStorage(IDataStorage reserveStorage) {  
 this.reserveStorage = reserveStorage;  
 reserveStorage.setCounterSet(deferredOperations);  
 }  
  
 public double[] getIsrCoeffs() {  
 return isrCoeffs;  
 }  
  
 public void setIsrCoeffs(double[] isrCoeffs) {  
 this.isrCoeffs = isrCoeffs;  
 }  
  
 public int getCalculationsStep() {  
 return calculationsStep;  
 }  
  
 public void setCalculationsStep(int calculationsStep) {  
 this.calculationsStep = calculationsStep;  
 }  
  
 @Override  
 public String getStorageName() {  
 return "Linear Adaptive Storage. Active Storage: " + activeStorage.getStorageName() + ". Passive Storage: " +  
 reserveStorage.getStorageName();  
 }  
  
  
 @Override  
 public boolean get(int value) {  
 *//select проводим только по активному хранилищу* operationHistory.get('S').add(new Date().getTime());  
 storedOperationsCount++;  
 boolean res = activeStorage.get(value);  
 if (storedOperationsCount == calculationsStep)  
 checkFunc();  
 return res;  
 }  
  
 @Override  
 public boolean set(int value) {  
 operationHistory.get('I').add(new Date().getTime());  
 storedOperationsCount++;  
 boolean res = activeStorage.set(value);  
 Thread t = null;  
 if (res)  
 t = new Thread() {  
 @Override  
 public void run() {  
 synchronized (reserveStorage) {  
 reserveStorage.uncheckedSet(value);  
 }  
 }  
 };  
 if (t != null) {  
 deferredOperationsList.add(t);  
 t.start();  
 }  
  
 if (storedOperationsCount == calculationsStep)  
 checkFunc();  
 return res;  
 }  
  
 @Override  
 public boolean remove(int value) {  
 operationHistory.get('R').add(new Date().getTime());  
 storedOperationsCount++;  
 boolean res = activeStorage.remove(value);  
 Thread t = null;  
 if (res)  
 t = new Thread() {  
 @Override  
 public void run() {  
 synchronized (reserveStorage) {  
 reserveStorage.remove(value);  
 }  
 }  
 };  
 if (t != null) {  
 deferredOperationsList.add(t);  
 t.start();  
 }  
  
 if (storedOperationsCount == calculationsStep)  
 checkFunc();  
  
 return res;  
 }  
  
 @Override  
 public Iterator<Integer> getIterator() {  
 return activeStorage.getIterator();  
 }  
  
 @Override  
 public void uncheckedSet(int value) {  
 activeStorage.uncheckedSet(value);  
 }  
  
 @Override  
 public ICounterSet getCounterSet() {  
 ICounterSet cs = new SimpleCounterSet();  
 cs.inc(OperationType.*ASSIGN*, activeOperations.get(OperationType.*ASSIGN*));  
 cs.inc(OperationType.*COMPARE*, activeOperations.get(OperationType.*COMPARE*));  
 cs.inc(OperationType.*CALCULATION*, activeOperations.get(OperationType.*CALCULATION*));  
 cs.inc(OperationType.*DEFERRED\_ASSIGN*, deferredOperations.get(OperationType.*ASSIGN*));  
 cs.inc(OperationType.*DEFERRED\_COMPARE*, deferredOperations.get(OperationType.*COMPARE*));  
 cs.inc(OperationType.*DEFERRED\_CALCULATION*, deferredOperations.get(OperationType.*CALCULATION*));  
 return cs;  
 }  
}

ConsoleResultPrinter.java

package com.vsu.amm.output;  
  
import com.vsu.amm.command.xmlgen.ICommandSource;  
import com.vsu.amm.data.storage.IDataStorage;  
import com.vsu.amm.stat.ICounterSet;  
import com.vsu.amm.stat.ICounterSet.OperationType;  
  
import java.util.List;  
import java.util.Map;  
  
*/\*\*  
 \* Вывод результатов в консоль  
 \*/*public class ConsoleResultPrinter implements IResultPrinter{  
  
 @Override  
 public void printResult(List<IDataStorage> storages, boolean showDetailed) {  
 if (storages == null || storages.isEmpty())  
 return;  
  
 StringBuilder sb = new StringBuilder();  
  
 int performanceCnt = 0;  
 for (IDataStorage storage : storages) {  
 sb.setLength(0);  
  
 System.*out*.println("Storage: " + storage.getStorageName());  
 ICounterSet cs = storage.getCounterSet();  
 if (cs == null) {  
 System.*out*.println("Operations count: N/A");  
 continue;  
 }  
 int asg = cs.get(OperationType.*ASSIGN*);  
 int cmp = cs.get(OperationType.*COMPARE*);  
 int clc = cs.get(OperationType.*CALCULATION*);  
 int opCnt = asg + cmp + clc;  
 int dasg = cs.get(OperationType.*DEFERRED\_ASSIGN*);  
 int dcmp = cs.get(OperationType.*DEFERRED\_COMPARE*);  
 int dclc = cs.get(OperationType.*DEFERRED\_CALCULATION*);  
 int defOpCnt = dasg + dcmp + dclc;  
 if (showDetailed) {  
 sb.append("Operations Count (ASSIGN/COMPARE/CALCULATION): ");  
 sb.append(asg).append("(").append(dasg).append(")/");  
 sb.append(cmp).append("(").append(dcmp).append(")/");  
 sb.append(clc).append("(").append(dclc).append(")/");  
 System.*out*.println(sb);  
 }  
 sb.setLength(0);  
 sb.append("Total: ");  
 sb.append(opCnt).append("(").append(defOpCnt).append(")");  
 System.*out*.println(sb);  
 if (performanceCnt == 0) {  
 performanceCnt = opCnt;  
 }  
 System.*out*.println(String.*format*("Performance: %.2f%%", performanceCnt \* 100.0 / opCnt));  
 System.*out*.println();  
 }  
 }  
  
 @Override  
 public void printResult(List<IDataStorage> storages, ICommandSource burden, boolean showDetailed) {  
 StringBuilder sb = new StringBuilder();  
 Map<String, Integer> descr = burden.getBurdenDescription();  
 sb.append("BURDEN (I/S/R): ");  
 sb.append(descr.get("Insert")).append("/").append(descr.get("Select")).append("/").append(descr.get("Remove"));  
 System.*out*.println(sb);  
 System.*out*.println();  
  
 int performanceCnt = 0;  
 for (IDataStorage storage : storages) {  
 sb.setLength(0);  
  
 System.*out*.println("Storage: " + storage.getStorageName());  
 ICounterSet cs = storage.getCounterSet();  
 if (cs == null) {  
 System.*out*.println("Operations count: N/A");  
 continue;  
 }  
 int asg = cs.get(OperationType.*ASSIGN*);  
 int cmp = cs.get(OperationType.*COMPARE*);  
 int clc = cs.get(OperationType.*CALCULATION*);  
 int opCnt = asg + cmp + clc;  
 int dasg = cs.get(OperationType.*DEFERRED\_ASSIGN*);  
 int dcmp = cs.get(OperationType.*DEFERRED\_COMPARE*);  
 int dclc = cs.get(OperationType.*DEFERRED\_CALCULATION*);  
 int defOpCnt = dasg + dcmp + dclc;  
 if (showDetailed) {  
 sb.append("Operations Count (ASSIGN/COMPARE/CALCULATION): ");  
 sb.append(asg).append("(").append(dasg).append(")/");  
 sb.append(cmp).append("(").append(dcmp).append(")/");  
 sb.append(clc).append("(").append(dclc).append(")/");  
 System.*out*.println(sb);  
 }  
 sb.setLength(0);  
 sb.append("Total: ");  
 sb.append(opCnt).append("(").append(defOpCnt).append(")");  
 System.*out*.println(sb);  
 if (performanceCnt == 0) {  
 performanceCnt = opCnt;  
 }  
 System.*out*.println(String.*format*("Performance: %.2f%%", performanceCnt \* 100.0 / opCnt));  
 System.*out*.println();  
 }  
 }  
}