МИНОБРНАУКИ РОССИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

(ФГБОУ ВПО «ВГУ»)

Факультет прикладной математики, информатики и механики

Кафедра Математического и Прикладного Анализа

**ИССЛЕДОВАНИЕ ВОЗМОЖНОСТЕЙ АДАПТИВНОГО ВЫБОРА ОПТИМАЛЬНЫХ СТРУКТУР ХРАНЕНИЯ ДАННЫХ**

Магистерская диссертация

по направлению 010300 – Фундаментальная информатика и информационные технологии

магистерская программа «Параллельное программирование и параллельные вычисления»

Допущен к защите в ГАК \_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| Зав. Кафедрой  \_\_.\_\_.*201*\_ *г*. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | д. ф.-м. н., проф. Шашкин А.И. |
| Руководитель  \_\_.\_\_.*201*\_ *г*. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | к. т. н., доц. Селезнев К.Е. |
| Студент  \_\_.\_\_.*201*\_ *г*. | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* | Зобов В.В. |

Воронеж – 2014

# Содержание

[Содержание 2](#_Toc391324931)

[Введение 4](#_Toc391324932)

[1. Постановка задачи 6](#_Toc391324933)

[2. Анализ задачи 7](#_Toc391324934)

[2.1. Структура нагрузки 7](#_Toc391324935)

[2.2. Методы индексации данных 9](#_Toc391324936)

[2.2.1. Массивы 10](#_Toc391324937)

[2.2.2 Списки 11](#_Toc391324938)

[2.2.3. Деревья 12](#_Toc391324939)

[2.2.3. Хеширование 13](#_Toc391324940)

[2.2.4. Кеширование 14](#_Toc391324941)

[2.3. Способ оценки метода индексации данных 16](#_Toc391324942)

[2.4. Выбор оптимального способа хранения 17](#_Toc391324943)

[2.5. Определение структуры нагрузки по журналу 18](#_Toc391324944)

[2.5.1. Алгоритм «Априори» 18](#_Toc391324945)

[2.5.2. Алгоритм определения структуры нагрузки 19](#_Toc391324946)

[3. Реализация 21](#_Toc391324947)

[3.1. Общая архитектура 21](#_Toc391324948)

[3.1.1. Модуль выбора оптимального хранилища данных 21](#_Toc391324949)

[3.1.2. Модуль определения структуры нагрузки по журналу 22](#_Toc391324950)

[3.1.3. Модуль пользовательского создания XML-файла 22](#_Toc391324951)

[3.2. Структура входных файлов 22](#_Toc391324952)

[3.3. Структура выходных файлов 23](#_Toc391324953)

[3.4. Внутренняя архитектура 23](#_Toc391324954)

[3.4.1 Значимые иерархии классов 23](#_Toc391324955)

[3.4.2 Внутренние архитектуры модулей 26](#_Toc391324956)

[4. Интерфейс пользователя 31](#_Toc391324957)

[5. Средства реализации 34](#_Toc391324958)

[6. Требования к аппаратному и программному обеспечению 35](#_Toc391324959)

[7. Эксперименты 36](#_Toc391324960)

[Заключение 38](#_Toc391324961)

[Список литературы 40](#_Toc391324962)

[Приложения 41](#_Toc391324963)

[Приложение 1. Пример входного XML-файла 41](#_Toc391324964)

[Приложение 2. Листинг. Интерфейсы. Классы структур данных 42](#_Toc391324965)

[Приложение 3. Листинг. Классы – модели данных 71](#_Toc391324966)

[Приложение 4. Листинг. Классы – обработчики 76](#_Toc391324967)

# Введение

Происходит информатизация, т.е. развитие информационных технологий, обеспечивающих достижение и поддержание уровня информированности членов общества. Увеличивается количество знаний, получаемых человечеством. Несколько лет назад объем создаваемых во всем мире данных превысил 1 зеттабайт – это примерно миллиард целиком заполненных жестких дисков емкостью 1 Тбайт, и уже превышает все доступное на сегодняшний день пространство хранения. Следовательно, возникает необходимость эффективной организации их хранения и управления доступом к ним.

Существует множество различных вариантов хранения информации, каждый их которых обладает собственными характеристиками. Кроме того, существуют методы для комбинирования контейнеров (модулей хранения информации). Таким образом, возможно построение огромного числа различных контейнеров, отвечающих за хранение информации и выполняющих операции поиска, вставки и удаления данных.

В ходе анализа работы тех или иных хранилищ данных при различной нагрузке и параметрах самих контейнеров, можно сделать выводы об их эффективности. Такие выводы являются важной составляющей при выборе из всего многообразия контейнеров оптимального для каждого конкретного случая, в зависимости от нагрузки и других факторов, возникающих при работе с контейнерами.

К примеру, в базах данных существует такой объект как индекс. Индекс – объект, создаваемый с целью повышения производительности поиска данных. 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-таблицы [9]. 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? Поэтому данное исследование является актуальным. Адаптивный выбор структуры хранения данных позволит системе выбирать оптимальный контейнер на статистики запросов к контейнеру данных.

# 1. Постановка задачи

Провести исследование возможностей адаптивного выбора оптимальных структур хранения данных. В связи с этим был поставлен ряд задач:

1. Разработать способ задания нагрузки на хранилища данных.
2. Разработать программное обеспечение, в котором должны быть реализованы:
   1. Способ моделирования заданной нагрузки на хранилище данных.
   2. Различные структуры хранения данных.
   3. Алгоритм выбора оптимального способа хранения данных.
   4. Алгоритм определения типа нагрузки по журналу.
   5. Возможность создания нагрузки пользователем.

# 2. Анализ задачи

## 2.1. Структура нагрузки

Нагрузка представляет собой различные последовательности операций вставки, выборки и удаления [7]. Существует 6 видов нагрузки (рис. 2.1.):

Рис. 2.1. Схема видов нагрузки.

* Sequence. Данная нагрузка означает, что все внутренние нагрузки идут в строгой последовательности.
* Block. Данная нагрузка означает, что все внутренние нагрузки должны быть перемешаны в случайной последовательности.
* Insert. Нагрузка представляет собой набор операций вставки данных в хранилище.
* Select. Нагрузка представляет собой набор операций выборки данных.
* Remove. Нагрузка представляет собой набор операций удаления данных.

Для каждой нагрузки могут быть заданы дополнительные параметры:

* Количество операций (count). Обязательный параметр для каждой нагрузки. По умолчанию равен 1.
* Метка (label). Используется для выделения значений нагрузки в журнале, а также при подсчете операций, которые произвело хранилище, при выполнении нагрузки помеченной меткой. Используется только в случае, когда родительской нагрузкой является sequence.
* Диапазон значений. Данный параметр является составным и описывается двумя параметрами:
  + Минимальное значение (min). Минимальное возможное значение сгенерированного ключа.
  + Максимальное значение (max). Максимальное значение такого ключа.
* Псевдоним (alias). Показывает, что значения, сгенерированные данной нагрузкой, должны быть сохранены, т.к. могут быть использованы при генерировании другой нагрузки.
* Ссылка на другую нагрузку (from). Показывает, что при генерации нагрузки должны быть использованы значения, сгенерированные в другой нагрузке.

Также значения параметров могут задаваться отдельно, в виде дополнительных массивов значений. Это позволяет генерировать различную нагрузку одинаковой структуры.

Все базовые нагрузка обязательно должны содержаться в какой-либо структурной нагрузке. Вложенность структурных нагрузок неограниченна. Нагрузку такой структуры проще всего анализировать с помощью рекурсивных алгоритмов [8].

Также к описанию нагрузки можно отдельно отнести нагрузку Data, которая описывает уже существующие данные в хранилище. Данная нагрузка также может иметь параметры:

* Количество значений (count).
* Диапазон значений.
  + Минимальное значение (min).
  + Максимальное значение (max).
* Конкретное значение нагрузки.

## 2.2. Методы индексации данных

В качестве исследуемых методов индексации, выбраны такие распространенные структуры, как массивы, списки, деревья (сбалансированные бинарные деревья и B-деревья), хеш-таблицы [3].

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

* Get. Операция записи данных в хранилище.
* Set. Операция поиска данных.
* Remove. Операция удаления данных.

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

Внутри каждой операции, кроме операции вставки существующих данных, происходит подсчет количества простейших операций с хранилищем. Поэтому каждый метод индексации данных должен также предоставлять методы для хранения и получение информации о количестве таких операций.

### 2.2.1. Массивы

Массив – упорядоченный набор данных, для хранения данных одного типа, идентифицируемых с помощью одного или нескольких индексов. Элементы массива располагаются в последовательных ячейках памяти, обозначаются именем массива и индексом. Каждое из значений, составляющих массив, называется его компонентой (или элементом массива). В простейшем случае массив имеет постоянную длину и хранит единицы данных одного и того же типа.

Количество используемых индексов массива может быть различным. Массивы с одним индексом называют одномерными, с двумя – двумерными и т. д. Одномерный массив нестрого соответствует вектору в математике, двумерный – матрице. Чаще всего применяются массивы с одним или двумя индексами, реже – с тремя, ещё большее количество индексов встречается крайне редко.

Достоинства:

* лёгкость вычисления адреса элемента по его индексу (поскольку элементы массива располагаются один за другим);
* одинаковое время доступа ко всем элементам;
* малый размер элементов: они состоят только из информационного поля.

Недостатки:

* для статического массива — отсутствие динамики, невозможность удаления или добавления элемента без сдвига других;
* для динамического массива – более низкое (по сравнению с обычным статическим) быстродействие и дополнительные накладные расходы на поддержку динамических свойств и/или гетерогенности;
* угроза выхода за границы массива и повреждения данных.

### 2.2.2 Списки

Список – динамическая структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

Достоинства:

* лёгкость добавления и удаления элементов;
* размер ограничен только объёмом памяти компьютера и разрядностью указателей;
* динамическое добавление и удаление элементов.

Недостатки:

* сложность определения адреса элемента по его индексу (номеру) в списке;
* на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны);
* работа со списком медленнее, чем с массивами, так как к любому элементу списка можно обратиться, только пройдя все предшествующие ему элементы;
* элементы списка могут быть расположены в памяти разреженно, что окажет негативный эффект на кэширование процессора;
* над связными списками гораздо труднее (хотя и в принципе возможно) производить параллельные векторные операции, такие как вычисление суммы.

### 2.2.3. Деревья

Двоичное дерево – иерархическая структура данных. Узел двоичного дерева помимо данных хранит ссылки на правого и левого потомков, для которых он является предком. У двоичного дерева явно выделен один узел, называемый корнем дерева. У него в отличие от остальных узлов нет предка. У остальных узлов ровно один предок. У любого узла может быть от 0 до 2 потомков. Узлы без потомков называются листьями. Узлы с одинаковым расстоянием от корня образуют слой (или уровень). Двоичное дерево по своей природе рекурсивно. Потомок любого узла является корнем некоторого дерева. Можно говорить о левом и правом поддеревьях. Важной характеристикой двоичного дерева является высота – максимальная длина пути от корня до некоторого узла. Также у бинарного дерева есть такая характеристика как баланс. Баланс узла определяется как высота его левого поддерева минус высота его правого поддерева. Сбалансированным бинарным деревом является такое бинарное дерево, у которого абсолютное значение баланса каждого узла меньше или равно 1.

B-дерево – это структура хранения данных, являющаяся разновидностью дерева поиска. Особенностями В-деревьев является: сбалансированность, ветвистость, отсортированность и логарифмическое время работы всех стандартных операций (поиск, вставка, удаление). В отличие от бинарных деревьев В-деревья допускают большое число потомков для любого из узлов. Это свойство называется ветвистостью. Благодаря ветвистости, В-деревья очень удобны для хранения крупных последовательных блоков данных, поэтому такая структура часто находит применение в базах данных и файловых системах.

Достоинства В-деревьев:

* Во всех случаях полезное использование пространства вторичной памяти составляет свыше 50 %. С ростом степени полезного использования памяти не происходит снижения качества обслуживания.
* Произвольный доступ к записи реализуется посредством малого количества подопераций (обращения к физическим блокам).
* В среднем достаточно эффективно реализуются операции включения и удаления записей; при этом сохраняется естественный порядок ключей с целью последовательной обработки, а также соответствующий баланс дерева для обеспечения быстрой произвольной выборки.
* Неизменная упорядоченность по ключу обеспечивает возможность эффективной пакетной обработки.

Основной недостаток В-деревьев состоит в отсутствии для них эффективных средств выборки данных (т.е. метода обхода дерева), упорядоченных по отличному от выбранного ключу.

### 2.2.3. Хеширование

Хеширование – класс методов поиска, идея которого состоит в вычислении хеш-кода, однозначно определяемого элементом с помощью хеш-функции, и использовании его, как основы для поиска. Хеш-таблица – структура данных, реализующая интерфейс ассоциативного массива. Представляет собой эффективную структуру данных, позволяющую хранить пары (ключ, значение), в которых ключ является хеш-кодом.

Хеш-функция преобразования ключей должна обеспечивать как можно более равномерное распределение ключей по всему диапазону значений индекса. Других ограничений на распределение нет, но на самом деле желательно, чтобы оно оказалось случайным. Также очевидно, что эта функция должна допускать эффективное вычисление, т.е. состоят из очень небольшого числа основных арифметических операций [2]. Пример равномерной хеш-функции [1]:

На практике при выборе хеш-функции пользуются различными эвристиками, основанными на специфике задач.

В общем случае, однозначного соответствия между исходными данными и хеш-кодом нет в силу того, что количество значений хеш-функций меньше, чем вариантов исходных данных, поэтому существуют элементы, имеющие одинаковые хеш-коды – так называемые коллизии, но если два элемента имеют разный хеш-код, то они гарантированно различаются. Вероятность возникновения коллизий играет немаловажную роль в оценке качества хеш-функций.

Существует два способа разрешения коллизий при использовании хеширования:

* открытая адресация;
* метод цепочек.

Достоинства:

* Важное свойство хеш-таблиц состоит в том, что, при некоторых разумных допущениях, все три операции (поиск, вставка, удаление элементов) в среднем выполняются за время O(1), время для наихудшего случая - O(n).

Недостатки:

* Итерация не в порядке возрастания ключей.
* Необходимость «перехеширования» при увеличении числа хранимых объектов.
* Нельзя реализовать быстро работающие дополнительные операции MIN, MAX и алгоритм обхода всех хранимых пар в порядке возрастания или убывания ключей.
* Не поддерживает упорядоченности, и не сохраняет порядок следования элементов.
* Возможность коллизий

### 2.2.4. Кеширование

Кэш – промежуточный буфер с быстрым доступом, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Доступ к данным в кэше осуществляется быстрее, чем выборка исходных данных из более медленной памяти или удаленного источника, однако её объем существенно ограничен по сравнению с хранилищем исходных данных.

Кэш состоит из набора записей. Каждая запись ассоциирована с элементом данных или блоком данных (небольшой части данных), которая является копией элемента данных в основной памяти. Каждая запись имеет идентификатор, определяющий соответствие между элементами данных в кэше и их копиями в основной памяти.

Когда клиент кэша обращается к данным, прежде всего, исследуется кэш. Если в кэше найдена запись с идентификатором, совпадающим с идентификатором затребованного элемента данных, то используются элементы данных в кэше. Такой случай называется попаданием кэша. Если в кэше не найдена запись, содержащая затребованный элемент данных, то он читается из основной памяти в кэш, и становится доступным для последующих обращений. Такой случай называется промахом кэша. Процент обращений к кэшу, когда в нём найден результат, называется уровнем попаданий или коэффициентом попаданий в кэш.

При модификации элементов данных в кэше выполняется их обновление в основной памяти. Задержка во времени между модификацией данных в кэше и обновлением основной памяти управляется так называемой политикой записи.

Политики записи:

* В кэше с немедленной записью каждое изменение вызывает синхронное обновление данных в основной памяти.
* В кэше с отложенной записью (или обратной записью) обновление происходит в случае вытеснения элемента данных, периодически или по запросу клиента.

В терминах данного исследования кеширование не является самостоятельной структурой данных, а идет в дополнение к уже имеющимся структурам (массивам, спискам и т.д.), расширяя их возможности.

## 2.3. Способ оценки метода индексации данных

Для того чтобы оценить какой-либо из методов индексации данных, необходимо понять, что происходит при операциях (запись, выборка и удаление данных) с хранилищем.

Внутри хранилища каждую операцию можно разложить на композицию некоторого количества двух простейших операций: сравнения и присваивания (рис. 2.2).

Метод индексации данных

Операции

вставки, поиска, удаления

n операций сравнения

Базовая операция

m операций присваивания

Общее количество операций сравнения (COMPARE)

Общее количество операций присваивания (ASSIGN)

Общее количество простейших операций (COMPARE + ASSIGN)

Рис. 2.2. Схема разложения базовых операций.

Для каждого из описанных выше методов индексации подсчет таких операций производится по определенному алгоритму, отличающемуся для каждой структуры данных.

Результатом применения всех запросов (всех операций вставки, поиска и удаления) к хранилищу данных будут являться следующие величины:

* общее количество операций сравнения;
* общее количество операций присваивания;
* количество операций сравнения по каждой нагрузке или совокупности нагрузок, отмеченной определенным lable'ом;
* количество операций присваивания по каждой нагрузке или совокупности нагрузок, отмеченной определенным lable'ом.

## 2.4. Выбор оптимального способа хранения

В качестве критерия сравнения способов хранения данных стоит выбрать оценку суммарного количества операций сравнения и присваивания. Оптимальной структурой хранения данных в таком случае будет являться та структура, у которой сумма таких операций будет минимальной.

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

* Значения параметров нагрузки.
* Оптимальный контейнер хранения данных.
* Параметры контейнера.
* Количество операций сравнения и присваивания.

Накопленная статистика в дальнейшем позволит, на основе знаний о структуре нагрузки и её параметрах, производить выбор оптимального хранилища, не проводя никаких экспериментов.

## 2.5. Определение структуры нагрузки по журналу

Определение структуры нагрузки по журналу является важной задачей в рамках данного исследования, потому что в большинстве случаев нет данных о точной структуре нагрузки, которая применялась к хранилищу данных, но почти всегда есть журнал операций над хранилищем.

Восстановив структуру нагрузки по журналу можно, на основе некоторых, накопленных ранее, статистических данных можно будет сделать вывод об оптимальной структуре хранения, не производя лишних операций с хранилищем. Так как это может требовать больших ресурсов.

Задачу определения структуры нагрузки по имеющемуся журналу операций с хранилищем данных, можно разделить на два этапа:

1. Применение алгоритма «Априори».
2. Восстановление структуры нагрузки из данных, полученных на первом этапе.

### 2.5.1. Алгоритм «Априори»

Алгоритм «Априори»:

1. Выделить из файла-журнала четыре множества:
   1. Множество элементов, которые уже существовали в хранилище.
   2. Множество добавленных элементов.
   3. Множество выбранных элементов.
   4. Множество удаленных элементов.
2. Пересечь попарно все полученные множества.
3. Повторять пункт 2 для вновь полученных множеств (рис. 2.1) пока не выполнится одно из условий:
   1. Количество элементов каждого нового множества не будет меньше 10% от суммы элементов начальных множеств.
   2. Результатом пересечения останется одно множество, являющееся пересечением всех начальных множеств.

Рис. 2.1. Множества, полученные в результате работы алгоритма.

### 2.5.2. Алгоритм определения структуры нагрузки

Алгоритм определения структуры нагрузки:

1. Для каждого множества, получившиеся на последней итерации алгоритма «Априори», определить пересечением каких начальных множеств оно получилось.
2. Для каждого такого множества создать нагрузку в соответствии с начальными множествами. Количеством выполнений для каждой создаваемой нагрузки будет количество элементов во множестве.
3. После обработки всех множеств нижнего уровня, перейти к множествам находящимся на уровень выше.
4. Повторить алгоритм для этих множеств, добавляя нагрузку, полученную ранее, в виде блоков.

На выходе работы алгоритма получим развернутый вариант изначальной нагрузки. При сворачивании результата работы алгоритма получим изначальную структуру нагрузки, а также примерные значения её параметров.

# 3. Реализация

## 3.1. Общая архитектура

Реализацию программного комплекса можно разделить на три функциональных модуля:

* модуль выбора оптимального хранилища данных;
* модуль определения структуры нагрузки по журналу;
* модуль пользовательского создания XML-файла.

### 3.1.1. Модуль выбора оптимального хранилища данных

Данный модуль отвечает за основной функционал программного комплекса. Модуль имеет две схемы работы в зависимости от входных данных. Схемы работы модуля представлены на рис. 3.1. и 3.2.

Рис. 3.1. Первая схема работы модуля выбора оптимального хранилища.

При данной схеме работы комплекс получает на вход XML-файл, содержащий описание нагрузки, её параметров и описание хранилищ. На выходе возвращает журнал операций над хранилищем и CSV-файл, в котором для всевозможных значений параметров указан оптимальный способ хранения.

Рис. 3.2. Вторая схема работы модуля выбора оптимального хранилища.

По данной схеме работы комплекс получает на вход журнал операций над хранилищем и список хранилищ. На выходе возвращает CSV-файл, в котором указан оптимальный способ хранения для нагрузки, описанной в журнале.

### 3.1.2. Модуль определения структуры нагрузки по журналу

Модуль позволяет восстановить структуру нагрузки по имеющемуся журналу. Схема работы модуля представлена на рис. 3.3.

Рис. 3.3. Схема работы модуля определения нагрузки по журналу.

### 3.1.3. Модуль пользовательского создания XML-файла

Модуль позволяет создать xml-файл входных данных, используя пользовательский ввод. Схема работы модуля представлена на рис. 3.4.

Рис. 3.4. Схема работы модуля создания XML-файла.

## 3.2. Структура входных файлов

Для передачи входных данных в программу используется XML-файл особой структуры.

Основными блоками файла являются:

* Блок e\_data. Данный блок описывает исходные данные, с которыми должны инициализироваться хранилища данных. Может быть пустым.
* Блок data, внутри которого задается нагрузка на хранилище. Данный блок обязательным не должен быть пустым, причем при задании нагрузки необходимо, чтобы существовала одна общая, внешняя нагрузка block или sequence.
* Блок storages – блок описания хранилищ с их параметрами.
* Блок param\_values – блок, содержащий все возможные значения параметров для нагрузки.

Пример xml-файла представлен в приложении 1.

## 3.3. Структура выходных файлов

Результатом работы программного комплекса являются:

* Файл-журнал – текстовый файл, описывающий, какие операции проводились с хранилищем данных.
* CSV-файл, в котором для всевозможных значений параметра указан, оптимальный, из заданных во входном файле, способ хранения.

В файл-журнал заносятся все операции в виде пар вида *имя\_операции*/*значение*. Пример пар: INS *data*, GET *data*, REMOVE *data*. Также если у нагрузки задана метка, то в начале и в конце данной нагрузки эта метка будет выставлена, при условии, что базовая нагрузка не находится нагрузке типа Block. Если для нагрузки заданы какие-либо параметры, то для каждого набора таких параметров в журнал будет записана отдельная секция.

В выходной CSV-файл записываются значения параметров нагрузки, оптимальный способ хранения, параметры для способа хранения и количество операций сравнения и присваивания, которые были выполнены хранилищем данных при моделировании нагрузки.

## 3.4. Внутренняя архитектура

## 3.4.1 Значимые иерархии классов

Иерархия классов хранилищ данных (рис. 3.5), получена в ходе наследования от общего интерфейса *IDataSet*, который описывает базовые функции для операций работы со структурой данных [11]. Подклассами данного интерфейса, являются *IDataStorage* и *IDataStream*.

Рис.3.5. Иерархия классов работы с данными.

Интерфейс *IDataStorage* добавляет возможность для работы с классами, отвечающими за подсчет количества простейших операций при работе с хранилищем данных. Любая из структур хранения данных в программном обеспечении является реализацией этого интерфейса.

Интерфейс *IDataStream* предоставляет более широкие возможности для применения нагрузки и служит для описания методов, которые будут использованы при выводе результатов. Класс *LogFileWriter* отвечает за вывод формирование журнала операций с хранилищем данных. Класс *DataStreamImpl* создан с использованием шаблона проектирования «Декоратор» (использует класс *LogFileWriter*) и отвечает за вывод всех полученных результатов (CSV-файл и файл журнала) [4].

Иерархия классов для хранения и изменения количества операций с базой данных (рис. 3.6), получена в ходе наследования от общего интерфейса *ICounterSet*, который описывает базовые функции учета количества простых операций выполненных хранилищем данных. Реализаций данного интерфейса является класс *SimpleCounterSet*, который ведет учет по операциям вставки и сравнения.

Рис. 3.6 Классы, описывающие Counters.

Иерархия классов команд задания нагрузки представлена на рисунке 3.7. Абстрактный класс *AbstractDataCommand* содержит общие для всех команд поля, которые обязательно должны быть у всех команд-потомков. Каждый из классов потомков добавляет свои поля, которые позволяют описать, задаваемую этой командой нагрузку.

Рис. 3.7 Классы, описывающие команды создания нагрузки.

## 3.4.2 Внутренние архитектуры модулей

Внутренняя схема работы модуля определения оптимального хранилища данных на основе данных, полученных из xml-файла, представлена на рисунке 3.8.

ICounterSet

Сгенерированная нагрузка

IDataStorage (м.б. много)

DataStreamImpl

Файл-журнал

CSV-файл

XML-файл

LogFileWriter

ProcessLoadXML

3.8. Внутренняя схема работы модуля выбора оптимального хранилища данных на основе xml-файла.

При анализе входных данных (класс *ProcessLoadXML*) из них выделяются следующие данные:

* Описание существующих данных в хранилище.
* Нагрузка.
* Структуры данных для анализа.
* Параметры.

Далее происходит рекурсивное генерирование нагрузки с использование параметров, если таковые имеются. Также создаются все структуры данных и в них записываются данные, которые были сгенерированы, как существующие. Такие данные не вызывают увеличение сounters. Затем нагрузка применяется к хранилищам и выдаётся результат в виде CSV-файла.

При схеме работы данного модуля, исходными данными при которой является файл журнала (рис. 3.9.), пропускается этап генерирования нагрузки и после разбора журнала нагрузка из него применяется к хранилищам данных.

ICounterSet

IDataStorage (м.б. много)

DataStreamImpl

CSV-файл

Файл-журнал

3.9. Внутренняя схема работы модуля выбора оптимального хранилища данных на основе файла журнала.

Внутренняя схема работы модуля для определения структуры нагрузки по журналу представлена на рисунке 3.10.

Алгоритм «Априори» (BurdenRestore)

Восстановление нагрузки (BurdenRestore)

XML-файл

Файл-журнал

3.10. Внутренняя схема работы модуля для определения структуры нагрузки по журналу.

Данный модуль получает на входе файл-журнал операций с хранилищем данных. Анализирует входные данные. Затем последовательно применяет алгоритм «Априори» и алгоритм восстановления структуры нагрузки. Как только структура восстановлена, происходит её запись в выходной xml-файл.

Внутренняя схема работы модуля для пользовательского создания входного xml-файла представлена на рисунке 3.11.

Анализ введенных данных

Генерирование части исходящих

XML-файл

Ввод одной из доступных команд

Продолжить?

да

нет

3.11. Внутренняя схема работы модуля пользовательского создания входного xml-файла.

Данный модуль получает на вход пользовательский ввод, который состоит из специальных команд. Анализирует эти команды. Затем генерирует по ним данные и компонует выходной xml-файл.

# 4. Интерфейс пользователя

Программный комплекс предоставляет консольный интерфейс для пользователя. Все действия необходимо производить через набор специальных команд.

Существует два вида команд:

* Общие. Этот тип команд предназначен для управления программой.
* Дополнительные. С помощью этих команд производится ввод данных в программу.

Общие команды:

* play\_xml. Данная команда запускает модуль нахождения оптимальной структуры хранения данных на основе xml-файла. Параметры:
  + –if или –infile. Имя входного xml-файла. Обязательный параметр.
  + –of или –outfile. Имя выходного CSV-файла.
  + –lf или –logfile. Имя выходного файла журнала.
* play\_log. Данная команда запускает модуль нахождения оптимальной структуры хранения данных на основе файла-журнала. Параметры:
  + –if или –infile. Имя входного файла-журнала. Обязательный параметр.
  + –of или –outfile. Имя выходного CSV-файла.
* restore\_burden. Данная команда запускает модуль нахождения структуры нагрузки по файлу-журналу.
  + –if или –infile. Имя входного файла-журнала. Обязательный параметр.
  + –of или –outfile. Имя выходного xml-файла.
* generate\_xml. Данная команда запускает модуль пользовательского создания xml-файла. Параметры:
  + –of или –outfile. Имя выходного xml-файла.

Дополнительные команды:

* insert. Команда задания нагрузки типа Insert. Параметры:
  + ­–count. Значение по умолчанию равен 1. Обязательный параметр.
  + –label. Метка для нагрузки.
  + –alias. Имя для ссылки на значения нагрузки.
  + –min. Минимальное значение нагрузки.
  + –max. Максимальное значение нагрузки.
* select. Команда задания нагрузки типа Select. Параметры:
  + –count. Значение по умолчанию равен 1. Обязательный параметр.
  + –label. Метка для нагрузки.
  + –from. Ссылка на значения другой нагрузки.
* remove. Команда задания нагрузки типа Relect. Параметры:
  + –count. Значение по умолчанию равен 1. Обязательный параметр.
  + –label. Метка для нагрузки.
  + –from. Ссылка на значения другой нагрузки.
* storage. Команда задания структуры хранения данных.
  + –class. Java класс хранилища данных. Обязательный параметр.
  + Список параметров хранилища. Все имена и значения параметров записываются через пробел. Количество неограничено.
* parameter. Команда задания параметров для нагрузки.
  + –name или –n. Имя параметра. Обязательный.
  + –value или –v. Значения параметра. Все значения и значения параметров записываются через пробел. Количество неограничено.

# Средства реализации

Для реализации поставленной задачи был выбран объектно-ориентированный язык программирования Java и использованы следующие инструментальные средства и технологии:

* IntelliJIDEA 12.1.3;
* Java Developer Kit 1.7 [10];
* Библиотека для работы с xml-файлами jdom;
* Библиотека логирования log4j.

# Требования к аппаратному и программному обеспечению

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

* процессор не ниже Pentium IV 2.6 GHz;
* оперативная память размером не менее 1024 Мб;
* не менее 5 Гб свободного дискового пространства;
* операционная система Windows XP/Vista/Seven/Server;
* виртуальная машина Java версии 1.7.

# 7. Эксперименты

**Эксперимент 1.** Интенсивная вставка.

Описание: количество операций вставки значительно превышает количество операций выборки и удаления.

Структура нагрузки:

<sequence count="1" label="root">

<insert count="10000" label="Ins" alias="Ins"/>

<select count="100" from="Ins"/>

</sequence>

Исследуемые хранилища:

* простой массив;
* простой список;
* хеш-таблица величиной в сто элементов с простом списков в качестве контейнера для разрешения коллизий;
* бинарное дерево.
* B-дерево.

Результат: оптимальным хранилищем, в данном случае, является хеш-таблица.

**Эксперимент 2.** Интенсивный поиск.

Описание: количество операций поиска значительно превышает количество операций вставки и удаления.

Структура нагрузки:

<sequence count="1" label="root">

<insert count="100" label="Ins" alias="Ins"/>

<select count="10000" from="Ins"/>

</sequence>

Исследуемые хранилища:

* простой массив;
* простой список;
* хеш-таблица величиной в сто элементов с простом списков в качестве контейнера для разрешения коллизий;
* бинарное дерево.
* B-дерево.

Результат: оптимальным хранилищем, в данном случае, является бинарное дерево.

**Эксперимент 3.** Вставка и поиск.

Описание: количество операций поиска и вставки одного порядка.

Структура нагрузки:

<sequence count="1" label="root">

<insert count="20000" label="Ins" alias="Ins"/>

<select count="15000" from="Ins"/>

</sequence>

Исследуемые хранилища:

* простой массив;
* простой список;
* хеш-таблица величиной в сто элементов с простом списков в качестве контейнера для разрешения коллизий;
* бинарное дерево.
* B-дерево.

Результат: оптимальным хранилищем, в данном случае, является хеш-таблица.

# Заключение

В ходе исследования было проанализировано поведение различных структур хранения данных на возможность адаптивного выбора оптимальных из них при определенной нагрузке.

В ходе исследования был разработан специальный способ задания нагрузки на хранилища данных. Такой способ помогает описывать нагрузку практически любой структуры. Также данный способ позволяет определить для нагрузки различные параметры и рассматривать её как не как одну нагрузку, а как массив нагрузок со всевозможными комбинациями значений параметров.

Основным результатом работы является программное обеспечение, в котором реализован следующий функционал:

1. Алгоритм разбора и моделирования заданной нагрузки на хранилище данных.
2. Различные структуры хранения данных:
   1. Массивы.
   2. Списки.
   3. Деревья (бинарные деревья, B-деревья).
   4. Хеширование (хеш-таблицы).
   5. Кеширование.
3. Алгоритм выбора оптимального способа хранения данных, действующий на основе оценки количества простейших операций с хранилищем, на которые можно разложить любую более сложную нагрузочную операцию.
4. Алгоритм определения типа нагрузки по журналу. Алгоритм «Априори».
5. Возможность создания нагрузки пользователем в ручном режиме.

Результаты данного исследования могут быть применены при построении и/или перестроении индексов внутри баз данных, для любых структурированных и неструктурированных данных большого объема (Big Data) и других данных, требующих индексации, с целью добавить возможность адаптивного выбора оптимальной структуры таких индексов в зависимости от различных факторов.

Результаты работы представлены в публикации:

*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, ВГУ – 2014,– Т. 3–, с. 154-161.*

# Список литературы

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

# Приложения

## Приложение 1. Пример входного XML-файла

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

<burden>

<e\_data />

<data>

<block count="1" label="root">

<insert count="10" alias="B" min="0" max="100"/>

<select count="5" from="B" />

<remove count="3" from="B"/>

</block>

</data>

<storages>

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

</storage>

**</storages>**

<param\_values />

</burden>

## Приложение 2. Листинг. Интерфейсы. Классы структур данных

**IDataSet, IDataStorage, SimpleArray, SimpleList, SortedArray, SortedList, CacheArray, SortedCacheArray, HashTable, BinaryTreem BTree.**

public interface IDataSet {

public void get(int value);

public void set(int value);

public void remove(int value);

}

public interface IDataStorage extends IDataSet {

public void setCounterSet(ICounterSet counterSet);

public ICounterSet getCounterSet();

public void setStorageParams(Map<String, String> params);

public Map<String, String> getStorageParams();

public void clear();

}

public class SimpleArray implements IDataStorage {

ArrayList<Integer> array;

Map<String, String> params;

ICounterSet counterSet;

public SimpleArray(){

array = new ArrayList<>();

}

public SimpleArray(Map<String, String> params){

array = new ArrayList<>();

this.params = params;

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

this.params = params;

}

@Override

public Map<String, String> getStorageParams() {

return params;

}

@Override

public void clear() {

array.clear();

}

@Override

public void get(int value) {

Iterator<Integer> iterator = array.iterator();

while((iterator.hasNext()) && (iterator.next() != value)){

counterSet.inc(ICounterSet.COMPARE);

}

counterSet.inc(ICounterSet.COMPARE);

}

@Override

public void set(int value) {

counterSet.inc(ICounterSet.ASSIGN);

array.add(value);

}

@Override

public void remove(int value) {

int index = 0;

int first = 0;

boolean end = false;

while(!end){

for(index = first; index < array.size(); index++){

if (array.get(index) == value){

counterSet.inc(ICounterSet.COMPARE);

counterSet.inc(ICounterSet.ASSIGN, array.size() - index);

break;

} else {

counterSet.inc(ICounterSet.COMPARE);

}

}

if (index != array.size()){

first = index;

array.remove(index);

} else {

end = true;

}

}

}

}

public class SimpleList implements IDataStorage {

private ICounterSet counterSet;

private Node root;

class Node {

int value;

Node next;

public Node(int value){

this.value = value;

this.next = null;

}

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

public Node getNext() {

return next;

}

public void setNext(Node next) {

this.next = next;

}

}

public SimpleList(){

this.root = null;

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

public Map<String, String> getStorageParams() {

return null;

}

@Override

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

}

@Override

public void clear() {

Node tmp = root.next;

Node next;

while(tmp != null){

next = tmp.next;

tmp = next;

}

root = null;

}

@Override

public void get(int value) {

Node tmp = root;

boolean found = false;

while ((tmp != null) && !found ){

counterSet.inc(ICounterSet.COMPARE);

if (tmp.value == value){

found = true;

}

tmp = tmp.next;

}

}

@Override

public void set(int value) {

if (root != null){

counterSet.inc(ICounterSet.ASSIGN);

Node node = new Node(value);

node.next = root;

root = node;

} else {

counterSet.inc(ICounterSet.ASSIGN);

root = new Node(value);

}

}

@Override

public void remove(int value) {

Node curr = root;

Node prev = null;

while (curr != null){

counterSet.inc(ICounterSet.COMPARE);

if (curr.value == value){

if (curr == root){

root = root.next;

curr = root;

prev = null;

} else {

prev.next = curr.next;

curr = prev.next;

}

} else {

prev = curr;

curr = curr.next;

}

}

}

}

public class SortedArray implements IDataStorage {

ArrayList<Integer> array;

Map<String, String> params;

ICounterSet counterSet;

public SortedArray(){

array = new ArrayList<>();

}

public SortedArray(Map<String, String> params){

array = new ArrayList<>();

this.params = params;

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

this.params = params;

}

@Override

public Map<String, String> getStorageParams() {

return params;

}

@Override

public void clear() {

array.clear();

}

@Override

public void get(int value) {

Iterator<Integer> iterator = array.iterator();

while((iterator.hasNext()) && (iterator.next() != value)){

counterSet.inc(ICounterSet.COMPARE);

}

counterSet.inc(ICounterSet.COMPARE);

}

@Override

public void set(int value) {

int index = 0;

if (array.size() == 0){

array.add(value);

counterSet.inc(ICounterSet.ASSIGN);

return;

} else {

while((index < array.size()) && (array.get(index) < value)){

counterSet.inc(ICounterSet.COMPARE);

index++;

}

if (index != array.size()){

counterSet.inc(ICounterSet.COMPARE);

}

if (index == array.size()){

array.add(value);

counterSet.inc(ICounterSet.ASSIGN);

} else {

array.add(index, value);

counterSet.inc(ICounterSet.ASSIGN, array.size() - index + 1);

}

}

}

@Override

public void remove(int value) {

int index = 0;

int first = 0;

boolean end = false;

while(!end){

for(index = first; index < array.size(); index++){

if (array.get(index) == value){

counterSet.inc(ICounterSet.COMPARE);

counterSet.inc(ICounterSet.ASSIGN, array.size() - index);

break;

} else {

counterSet.inc(ICounterSet.COMPARE);

}

}

if (index != array.size()){

first = index;

array.remove(index);

} else {

end = true;

}

}

}

}

public class SortedList implements IDataStorage {

private ICounterSet counterSet;

private Node root;

class Node {

int value;

Node next;

public Node(int value){

this.value = value;

this.next = null;

}

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

public Node getNext() {

return next;

}

public void setNext(Node next) {

this.next = next;

}

}

public SortedList(){

this.root = null;

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

}

@Override

public Map<String, String> getStorageParams() {

return null;

}

@Override

public void clear() {

Node tmp = root.next;

Node next;

while(tmp != null){

next = tmp.next;

tmp = next;

}

root = null;

}

@Override

public void get(int value) {

Node tmp = root;

boolean found = false;

while ((tmp != null) && !found ){

counterSet.inc(ICounterSet.COMPARE);

if (tmp.value == value){

found = true;

}

tmp = tmp.next;

}

}

@Override

public void set(int value) {

if (root != null){

Node curr = root;

Node prev = null;

while ((curr != null) && (curr.value < value)){

counterSet.inc(ICounterSet.COMPARE);

prev = curr;

curr = curr.next;

}

counterSet.inc(ICounterSet.COMPARE);

counterSet.inc(ICounterSet.ASSIGN);

Node node = new Node(value);

if (prev == null){

node.next = root;

root = node;

} else {

node.next = curr;

prev.next = node;

}

} else {

counterSet.inc(ICounterSet.ASSIGN);

root = new Node(value);

}

}

@Override

public void remove(int value) {

Node curr = root;

Node prev = null;

while (curr != null){

counterSet.inc(ICounterSet.COMPARE);

if (curr.value == value){

if (curr == root){

root = root.next;

curr = root;

prev = null;

} else {

prev.next = curr.next;

curr = prev.next;

}

} else {

prev = curr;

curr = curr.next;

}

}

}

}

public class CacheArray implements IDataStorage {

private static final int DEFAULT\_CACHE\_SIZE = 3;

ArrayList<Integer> array;

ICounterSet counterSet;

int cacheSize;

List<Integer> cache;

Map<String, String> params;

public CacheArray() {

array = new ArrayList<>();

cacheSize = DEFAULT\_CACHE\_SIZE;

cache = new LinkedList<>();

}

public CacheArray(Map<String, String> params) {

array = new ArrayList<>();

this.params = params;

if (params.containsKey("cache\_size")){

cacheSize = Integer.parseInt(params.get("cache\_size"));

} else {

cacheSize = DEFAULT\_CACHE\_SIZE;

}

cache = new LinkedList<>();

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

array = new ArrayList<>();

this.params = params;

if (params.containsKey("cache\_size")){

cacheSize = Integer.parseInt(params.get("cache\_size"));

} else {

cacheSize = DEFAULT\_CACHE\_SIZE;

}

cache = new LinkedList<>();

}

@Override

public Map<String, String> getStorageParams() {

return params;

}

@Override

public void clear() {

array.clear();

cache.clear();

}

@Override

public void get(int value) {

if (!cache.contains(value)){

Iterator<Integer> iterator = array.iterator();

while((iterator.hasNext()) && (iterator.next() != value)){

counterSet.inc(ICounterSet.COMPARE);

}

counterSet.inc(ICounterSet.COMPARE);

if (cache.size() == cacheSize){

cache.remove(0);

}

cache.add(value);

}

}

@Override

public void set(int value) {

counterSet.inc(ICounterSet.ASSIGN);

array.add(value);

}

@Override

public void remove(int value) {

if (cache.contains(value)){

Object removableObject = value;

cache.remove(removableObject);

}

boolean end = false;

int index;

int first = 0;

while(!end){

for(index = first; index < array.size(); index++){

if (array.get(index) == value){

counterSet.inc(ICounterSet.COMPARE);

counterSet.inc(ICounterSet.ASSIGN, array.size() - index);

break;

} else {

counterSet.inc(ICounterSet.COMPARE);

}

}

if (index != array.size()){

first = index;

array.remove(index);

} else {

end = true;

}

}

}

}

public class CacheSortedArray implements IDataStorage {

private static final int DEFAULT\_CACHE\_SIZE = 3;

ArrayList<Integer> array;

ICounterSet counterSet;

int cacheSize;

List<Integer> cache;

Map<String, String> params;

public CacheSortedArray(){

array = new ArrayList<>();

cacheSize = DEFAULT\_CACHE\_SIZE;

cache = new LinkedList<>();

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

array = new ArrayList<>();

this.params = params;

if (params.containsKey("cache\_size")){

cacheSize = Integer.parseInt(params.get("cache\_size"));

} else {

cacheSize = DEFAULT\_CACHE\_SIZE;

}

cache = new LinkedList<>();

}

@Override

public Map<String, String> getStorageParams() {

return params;

}

@Override

public void clear() {

array.clear();

cache.clear();

}

@Override

public void get(int value) {

if (!cache.contains(value)){

Iterator<Integer> iterator = array.iterator();

while((iterator.hasNext()) && (iterator.next() != value)){

counterSet.inc(ICounterSet.COMPARE);

}

counterSet.inc(ICounterSet.COMPARE);

if (cache.size() == cacheSize){

cache.remove(0);

}

cache.add(value);

}

}

@Override

public void set(int value) {

int index = 0;

if (array.size() == 0){

array.add(value);

counterSet.inc(ICounterSet.ASSIGN);

return;

} else {

while((index < array.size()) && (array.get(index) < value)){

counterSet.inc(ICounterSet.COMPARE);

index++;

}

if (index != array.size()){

counterSet.inc(ICounterSet.COMPARE);

}

if (index == array.size()){

array.add(value);

counterSet.inc(ICounterSet.ASSIGN);

} else {

array.add(index, value);

counterSet.inc(ICounterSet.ASSIGN, array.size() - index + 1);

}

}

}

@Override

public void remove(int value) {

if (cache.contains(value)){

Object removableObject = value;

cache.remove(removableObject);

}

boolean end = false;

int index;

int first = 0;

while(!end){

for(index = first; index < array.size(); index++){

if (array.get(index) == value){

counterSet.inc(ICounterSet.COMPARE);

counterSet.inc(ICounterSet.ASSIGN, array.size() - index);

break;

} else {

counterSet.inc(ICounterSet.COMPARE);

}

}

if (index != array.size()){

first = index;

array.remove(index);

} else {

end = true;

}

}

}

}

public class HashTable implements IDataStorage{

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 int DEFAULT\_HASH\_TABLE\_SIZE = 100;

private static final String DEFAULT\_STORAGE\_CLASS = "com.vsu.amm.data.storage.SimpleList";

Map<Integer, IDataStorage> hashTable;

int hashTableSize;

Map<String, String> params;

ICounterSet counterSet;

String storageClass;

Map<String, String> storageParams;

public HashTable(){

this.hashTableSize = DEFAULT\_HASH\_TABLE\_SIZE;

this.storageClass = DEFAULT\_STORAGE\_CLASS;

this.storageParams = new HashMap<>();

this.hashTable = new HashMap<>(hashTableSize);

}

public HashTable(Map<String, String> params){

this.params = params;

if(params.containsKey(HASH\_TABLE\_SIZE\_PARAM\_NAME)){

hashTableSize = Integer.parseInt(params.get(HASH\_TABLE\_SIZE\_PARAM\_NAME));

} else {

hashTableSize = DEFAULT\_HASH\_TABLE\_SIZE;

}

this.storageClass = DEFAULT\_STORAGE\_CLASS;

this.storageParams = new HashMap<>();

for (String param : params.keySet()){

if (param.startsWith(HASH\_TABLE\_STORAGE\_PARAM\_PREFIX)){

if (param.endsWith(HASH\_TABLE\_STORAGE\_CLASS\_PARAM\_NAME)){

storageClass = params.get(param);

} else {

storageParams.put(param.substring(param.indexOf(".")), params.get(param));

}

}

}

this.hashTable = new HashMap<>(hashTableSize);

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

this.params = params;

}

@Override

public Map<String, String> getStorageParams() {

return params;

}

@Override

public void clear() {

hashTable.clear();

counterSet = new SimpleCounterSet();

}

@Override

public void get(int value) {

int key = value / hashTableSize;

if (hashTable.containsKey(key)){

hashTable.get(key).get(value);

}

}

@Override

public void set(int value) {

int key = value / hashTableSize;

if (hashTable.containsKey(key)){

hashTable.get(key).set(value);

} else {

Storage storage = new Storage(storageClass, storageParams);

IDataStorage temp = storage.getStorage();

temp.setCounterSet(counterSet);

temp.set(value);

hashTable.put(key, temp);

}

}

@Override

public void remove(int value) {

int key = value / hashTableSize;

if (hashTable.containsKey(key)){

hashTable.get(key).remove(value);

}

}

}

public class BinaryTree implements IDataStorage {

private Node root;

ICounterSet counterSet;

Map<String, String> storageParams;

class Node {

int value;

Node right;

Node left;

Node parent;

int height;

public Node(int value){

this.value = value;

right = null;

left = null;

parent = null;

}

public int getValue() {

return value;

}

public void setValue(int value) {

this.value = value;

}

public Node getLeft() {

return left;

}

public void setLeft(Node left) {

this.left = left;

}

public Node getRight() {

return right;

}

public void setRight(Node right) {

this.right = right;

}

public int getHeight() {

return height;

}

public void setHeight(int height) {

this.height = height;

}

public Node getParent() {

return parent;

}

public void setParent(Node parent) {

this.parent = parent;

}

}

public BinaryTree(){

this.root = null;

}

public BinaryTree(Map<String, String> params){

this.root = null;

this.storageParams = params;

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

this.storageParams = params;

}

@Override

public Map<String, String> getStorageParams() {

return storageParams;

}

@Override

public void clear() {

root = null;

counterSet = new SimpleCounterSet();

}

private Node find(Node node, int value){

if (node == null){

return null;

}

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

counterSet.inc(ICounterSet.COMPARE);

return node;

}

return find(value < node.getValue() ? node.getLeft() : node.getRight(), value);

}

@Override

public void get(int value) {

find(root, value);

}

private Node insert(Node node, int value, Node parent){

if (node == null){

node = new Node(value);

node.setParent(parent);

counterSet.inc(ICounterSet.ASSIGN);

} else {

counterSet.inc(ICounterSet.COMPARE);

if (value < node.getValue()){

node.setLeft(insert(node.getLeft(), value, node));

} else {

node.setRight(insert(node.getRight(), value, node));

}

}

return node;

}

@Override

public void set(int value) {

root = insert(root, value, null);

}

public Node findMin(Node node) {

Node min = node;

if (min == null) return null;

while (min.getLeft() != null) {

min = min.getLeft();

}

return min;

}

/\*\*

\*

\* 1. Удаление элемента без детей – просто освобождаем память.

\* 2. Удаление элемента с одним ребенком – смена указателя родителя указывать директно к ребенку

\* удаляемого элемента и освобождение памяти.

\* 3. Удаление элемента с только одним ребенком и это КОРЕНЬ – перемещение ребенка на место корня

\* и освобождение памяти.

\* 4 Удаление элемента с двумя детьми – это самая сложная операция. Самый подходящий способ

\* исполнения это разменять стоимости удаляемого элемента и максимальную стоимость левого

\* поддерева или минимальную правого поддерева (потому что это сохранит характеристики дерева)

\* и тогда удаляем элемент без или с одним ребенком.

\*

\*

\*

\* @param node

\* @param value

\* @return

\*/

private Node delete(Node node, int value){

Node element;// = find(node, value);

while ((element = find(node, value)) != null) {

if (element == null) return node;

boolean hasParent = element.getParent() != null;

boolean isLeft = hasParent && element.getValue() < element.getParent().getValue();

counterSet.inc(ICounterSet.COMPARE);

if (element.getLeft() == null && element.getRight() == null) {

if (hasParent) {

if (isLeft) {

element.getParent().setLeft(null);

} else {

element.getParent().setRight(null);

}

counterSet.inc(ICounterSet.ASSIGN);

}

} else if (element.getLeft() != null && element.getRight() == null) {

if (hasParent) {

if (isLeft) {

element.getParent().setLeft(element.getLeft());

} else {

element.getParent().setRight(element.getLeft());

}

counterSet.inc(ICounterSet.ASSIGN);

} else {

node = element.getLeft();

}

} else if (element.getLeft() == null && element.getRight() != null) {

if (hasParent) {

if (isLeft) {

element.getParent().setLeft(element.getRight());

} else {

element.getParent().setRight(element.getRight());

}

} else {

node = element.getRight();

}

counterSet.inc(ICounterSet.ASSIGN);

} else {

Node rightMin = findMin(element.getRight());

element.setValue(rightMin.getValue());

counterSet.inc(ICounterSet.ASSIGN);

return delete(rightMin, rightMin.getValue());

}

}

return node;

}

@Override

public void remove(int value) {

root = delete(root, value);

}

}

public class BTree implements IDataStorage {

private static final String CHILDREN\_PAMAN\_NAME = "children\_count";

private static final int MAX\_CHILDREN\_COUNT = 4; // max children per B-tree node = MAX\_CHILDREN\_COUNT-1

ICounterSet counterSet;

Map<String, String> storageParams;

private Node root;

private int treeHeight;

private int elementsCount;

private int childrenCount;

private static final class Node {

private int m; // number of children

private Entry[] children;// = new Entry[MAX\_CHILDREN\_COUNT]; // the array of children

private Node(int k, int maxChildren) {

children = new Entry[maxChildren];

m = k;

} // create a node with k children

}

// internal nodes: only use key and next

// external nodes: only use key and value

private static class Entry {

private int value;

private Node next; // helper field to iterate over array entries

public Entry(int value, Node next) {

this.value = value;

this.next = next;

}

}

// constructor

public BTree() {

root = new Node(0, childrenCount);

}

public BTree(Map<String, String> params) {

root = new Node(0, childrenCount);

this.storageParams = params;

if (storageParams != null && storageParams.containsKey(CHILDREN\_PAMAN\_NAME)){

childrenCount = Integer.parseInt(storageParams.get(CHILDREN\_PAMAN\_NAME));

if (childrenCount > MAX\_CHILDREN\_COUNT){

childrenCount = MAX\_CHILDREN\_COUNT;

}

} else {

childrenCount = MAX\_CHILDREN\_COUNT;

}

}

@Override

public void setCounterSet(ICounterSet counterSet) {

this.counterSet = counterSet;

}

@Override

public ICounterSet getCounterSet() {

return counterSet;

}

@Override

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

this.storageParams = params;

if (storageParams != null && storageParams.containsKey(CHILDREN\_PAMAN\_NAME)){

childrenCount = Integer.parseInt(storageParams.get(CHILDREN\_PAMAN\_NAME));

if (childrenCount > MAX\_CHILDREN\_COUNT){

childrenCount = MAX\_CHILDREN\_COUNT;

}

}

}

@Override

public Map<String, String> getStorageParams() {

return storageParams;

}

@Override

public void clear() {

root = new Node(0, childrenCount);

counterSet = new SimpleCounterSet();

}

// return number of key-value pairs in the B-tree

public int size() {

return elementsCount;

}

// return height of B-tree

public int height() {

return treeHeight;

}

public void get(int value) {

search(root, value, treeHeight);

}

private void search(Node x, int value, int ht) {

Entry[] children = x.children;

// external node

if (ht == 0) {

for (int j = 0; j < x.m; j++) {

counterSet.inc(ICounterSet.COMPARE);

if (value == children[j].value){

return;

}

}

}

// internal node

else {

for (int j = 0; j < x.m; j++) {

if (j+1 == x.m || value < children[j+1].value){

if (j+1 != x.m){

counterSet.inc(ICounterSet.COMPARE);

}

search(children[j].next, value, ht-1);

}

}

}

}

public void set(int value) {

Node u = insert(root, value, treeHeight);

elementsCount++;

if (u == null) {

return;

}

// need to split root

Node t = new Node(2, childrenCount);

t.children[0] = new Entry(root.children[0].value, root);

t.children[1] = new Entry(u.children[0].value, u);

root = t;

treeHeight++;

}

@Override

public void remove(int value) {

}

private Node insert(Node h, int value, int ht) {

int j;

Entry t = new Entry(value, null);

// external node

if (ht == 0) {

for (j = 0; j < h.m; j++) {

counterSet.inc(ICounterSet.COMPARE);

if (value < h.children[j].value)

break;

}

}

// internal node

else {

for (j = 0; j < h.m; j++) {

if ((j+1 == h.m) || value < h.children[j+1].value) {

if (j+1 != h.m) {

counterSet.inc(ICounterSet.COMPARE);

}

Node u = insert(h.children[j++].next, value, ht-1);

if (u == null){

return null;

}

counterSet.inc(ICounterSet.ASSIGN);

t.value = u.children[0].value;

t.next = u;

break;

}

}

}

for (int i = h.m; i > j; i--) {

h.children[i] = h.children[i-1];

}

h.children[j] = t;

h.m++;

if (h.m < childrenCount) {

return null;

} else {

return split(h);

}

}

// split node in half

private Node split(Node h) {

Node t = new Node(childrenCount /2, childrenCount);

h.m = childrenCount /2;

for (int j = 0; j < childrenCount /2; j++)

t.children[j] = h.children[childrenCount /2+j];

return t;

}}

## Приложение 3. Листинг. Классы – модели данных

**Storage, Burden.**

public class Storage {

String storageClass;

Map<String, String> params;

public Storage(String storageClass, Map<String, String> params) {

this.storageClass = storageClass;

this.params = params;

}

public IDataStorage getStorage(){

IDataStorage storage = null;

try {

Class c = Class.forName(storageClass);

Object obj = c.newInstance();

if (obj instanceof IDataStorage){

storage = (IDataStorage)obj;

storage.setStorageParams(params);

}

} catch (ClassNotFoundException e) {

e.printStackTrace();

//log.error(new MasterWorkException(e.getMessage(), e));

} catch (InstantiationException e) {

e.printStackTrace();

//log.error(new MasterWorkException(e.getMessage(), e));

} catch (IllegalAccessException e) {

e.printStackTrace();

//log.error(new MasterWorkException(e.getMessage(), e));

}

return storage;

}

}

public class Burden {

public static enum TagNames{

SEQUENCE("sequence"),

BLOCK("block"),

INSERT("insert"),

SELECT("select"),

REMOVE("remove")

;

private TagNames(final String name){

this.name = name;

}

private final String name;

@Override

public String toString() {

return name;

}

}

private int id;

private int parentId;

private String name;

private String count;

private String label;

private String alias;

private String from;

private String min;

private String max;

private String tempCount;

private List<Integer> childrenId;

private List<Integer> genKeyValueToAlias;

public Burden(int id, int parentId, String name, String count, String label, String alias, String from, String min, String max) {

this.id = id;

this.parentId = parentId;

this.name = name;

this.count = count;

this.label = label;

this.alias = alias;

this.from = from;

this.min = min;

this.max = max;

this.childrenId = null;

this.genKeyValueToAlias = null;

this.tempCount = count;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public int getParentId() {

return parentId;

}

public void setParentId(int parentId) {

this.parentId = parentId;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getCount() {

return count;

}

public void setCount(String count) {

this.count = count;

}

public String getLabel() {

return label;

}

public void setLabel(String label) {

this.label = label;

}

public String getAlias() {

return alias;

}

public void setAlias(String alias) {

this.alias = alias;

}

public String getFrom() {

return from;

}

public void setFrom(String from) {

this.from = from;

}

public String getMin() {

return min;

}

public void setMin(String min) {

this.min = min;

}

public String getMax() {

return max;

}

public void setMax(String max) {

this.max = max;

}

public List<Integer> getChildrenId() {

return childrenId;

}

public void setChildrenId(List<Integer> childrenId) {

this.childrenId = childrenId;

}

public List<Integer> getGenKeyValueToAlias() {

return genKeyValueToAlias;

}

public void setGenKeyValueToAlias(List<Integer> genKeyValueToAlias) {

this.genKeyValueToAlias = genKeyValueToAlias;

}

public String getTempCount() {

return tempCount;

}

public void setTempCount(String tempCount) {

this.tempCount = tempCount;

}

}

## Приложение 4. Листинг. Классы – обработчики

**ProcessLoadXML, DataStreamImpl, LogFileWriter, LogFilePlayer.**

public class ProcessLoadXML {

private static final Logger log = Logger.getLogger(ProcessLoadXML.class);

public static final String SEQUENCE\_NAME = "sequence";

public static final String BLOCK\_NAME = "block";

public static final String INSERT\_NAME = "insert";

public static final String SELECT\_NAME = "select";

public static final String REMOVE\_NAME = "remove";

private static HashMap<Integer, Burden> tags;

private static HashMap<String, Integer> aliases;

private static HashMap<String, List<String>> paramsValues;

private static List<Pair<String, String>> generate(Burden burden) throws Exception {

Pair<String, String> resultValue;

List<Pair<String, String>> result = new LinkedList<Pair<String, String>>();

List<Integer> children;

int count;

String name = burden.getName();

switch (Burden.TagNames.valueOf(name.toUpperCase())){

case SEQUENCE:

count = Integer.parseInt(burden.getCount());

if ((burden.getLabel() != null) && ((burden.getId() == burden.getParentId()) || (!BLOCK\_NAME.equals(tags.get(burden.getParentId()).getName())))){

resultValue = new Pair<String, String>(Constants.LABEL\_COMMAND\_NAME, burden.getLabel());

result.add(resultValue);

}

for(int index = 1; index <= count; index++){

children = burden.getChildrenId();

if (children != null){

for(int i : children){

result.addAll(generate(tags.get(i)));

}

}

}

break;

case BLOCK:

HashMap<Integer, List<Pair<String, String>>> values = new HashMap<Integer, List<Pair<String, String>>>();

count = Integer.parseInt(burden.getCount());

if ((burden.getLabel() != null) && ((burden.getId() == burden.getParentId()) || (!BLOCK\_NAME.equals(tags.get(burden.getParentId()).getName())))){

resultValue = new Pair<String, String>(Constants.LABEL\_COMMAND\_NAME, burden.getLabel());

result.add(resultValue);

}

for(int index = 1; index <= count; index++){

children = burden.getChildrenId();

if (children != null){

for(int i : children){

values.put(i, generate(tags.get(i)));

}

Random random = new Random();

int childrenCount = children.size();

int i, k, c, id, size, begin, end ,elCount;

Burden temp;

Pair<String, String> el;

while (values.size() != 0){

i = random.nextInt(childrenCount);

id = children.get(i);

temp = tags.get(id);

if (SEQUENCE\_NAME.equals(temp.getName()) || BLOCK\_NAME.equals(temp.getName())){

c = Integer.parseInt(temp.getTempCount());

if (values.get(id) != null){

size = values.get(id).size();

/\*if (values.get(id).get(0).equals(temp.getLabel())){

size -= 1;

begin = 1;

} else {

begin = 0;

}\*/

begin = 0;

elCount = size / c;

k = random.nextInt(c);

begin = k \* elCount + begin;

end = begin + elCount - 1;

result.addAll(values.get(id).subList(begin, end + 1));

for(int j = end; j >= begin; j--){

values.get(id).remove(j);

}

temp.setTempCount(String.valueOf(c - 1));

if (values.get(id).size() == 0){

values.remove(id);

temp.setTempCount(temp.getCount());

}

}

} else {

if (values.get(id) != null){

size = values.get(id).size();

k = random.nextInt(size);

el = values.get(id).get(k);

values.get(id).remove(k);

result.add(el);

if (values.get(id).size() == 0){

values.remove(id);

}

}

}

}

}

}

break;

case INSERT:

result.addAll(generateInsert(burden));

break;

case SELECT:

result.addAll(generateSelect(burden));

break;

case REMOVE:

result.addAll(generateRemove(burden));

break;

default:

break;

}

return result;

}

private static List<Pair<String, String>> generateInsert(Burden insert) throws Exception {

Pair<String, String> resultValue;

List<Pair<String, String>> result = new LinkedList<Pair<String, String>>();

Random randomGenerator = new Random();

String alias = insert.getAlias();

List<Integer> keys = new LinkedList<Integer>();

int key;

int count = Integer.parseInt(insert.getCount());

boolean range = false;

int min = 0;

int max = 0;

if ((insert.getMin() != null) && (insert.getMax() != null)){

range = true;

min = Integer.parseInt(insert.getMin());

max = Integer.parseInt(insert.getMax());

}

if ((insert.getLabel() != null) && (!BLOCK\_NAME.equals(tags.get(insert.getParentId()).getName()))){

resultValue = new Pair<String, String>(Constants.LABEL\_COMMAND\_NAME, insert.getLabel());

result.add(resultValue);

}

for(int index = 1; index <= count; index++){

if (range){

key = randomGenerator.nextInt((max - min) + 1) + min;

} else {

key = randomGenerator.nextInt(Integer.MAX\_VALUE);

}

keys.add(key);

resultValue = new Pair<String, String>(Constants.INSERT\_COMMAND\_NAME, String.valueOf(key));

result.add(resultValue);

}

if (alias != null){

insert.setGenKeyValueToAlias(keys);

}

return result;

}

private static List<Pair<String, String>> generateSelect(Burden select) throws Exception {

Pair<String, String> resultValue;

List<Pair<String, String>> result = new LinkedList<Pair<String, String>>();

Random randomGenerator = new Random();

int key;

int count = Integer.parseInt(select.getCount());

List<Integer> keys = null;

String from = select.getFrom();

if (from != null){

keys = tags.get(aliases.get(from)).getGenKeyValueToAlias();

if (keys == null){

throw new Exception("Not found the key values");

}

}

if ((select.getLabel() != null) && (!BLOCK\_NAME.equals(tags.get(select.getParentId()).getName()))){

resultValue = new Pair<String, String>(Constants.LABEL\_COMMAND\_NAME, select.getLabel());

result.add(resultValue);

}

for(int index = 1; index <= count; index++){

if (from != null){

key = keys.get(randomGenerator.nextInt(keys.size()));

} else {

key = randomGenerator.nextInt(Integer.MAX\_VALUE);

}

resultValue = new Pair<String, String>(Constants.SELECT\_COMMAND\_NAME, String.valueOf(key));

result.add(resultValue);

}

return result;

}

private static List<Pair<String, String>> generateRemove(Burden remove) throws Exception{

Pair<String, String> resultValue;

List<Pair<String, String>> result = new LinkedList<Pair<String, String>>();

Random randomGenerator = new Random();

int key;

int count = Integer.parseInt(remove.getCount());

List<Integer> keys = null;

String from = remove.getFrom();

if (from != null){

keys = tags.get(aliases.get(from)).getGenKeyValueToAlias();

if (keys == null){

throw new Exception("Not found the key values");

}

}

if ((remove.getLabel() != null) && (!BLOCK\_NAME.equals(tags.get(remove.getParentId()).getName()))){

resultValue = new Pair<String, String>(Constants.LABEL\_COMMAND\_NAME, remove.getLabel());

result.add(resultValue);

}

for(int index = 1; index <= count; index++){

if (from != null){

key = keys.get(randomGenerator.nextInt(keys.size()));

} else {

key = randomGenerator.nextInt(Integer.MAX\_VALUE);

}

resultValue = new Pair<String, String>(Constants.REMOVE\_COMMAND\_NAME, String.valueOf(key));

result.add(resultValue);

}

return result;

}

private void generateParams(List<String> params, List<String> result, String resultString){

if (params.size() == 0){

result.add(resultString);

} else {

String name = params.get(0);

for (String value : paramsValues.get(name)){

if (params.size() != 1){

generateParams(params.subList(1, params.size()), result, resultString + name + "=" + value + ";");

} else {

generateParams(new ArrayList<String>(), result, resultString + name + "=" + value + ";");

}

}

}

}

public void processXMLNew(String filename, IDataStream stream){

//IDataStorage dataStorage;

List<IDataStorage> dataStorages = new LinkedList<>();

ICounterSet counterSet;

int index;

Burden burden;

Document document;

Element element, rootNode, storagesNode, dataNode, paramsNode;

String storageClass = null;

Map<String, String> storageParams;

String storageParamName;

String storageParamValue;

Map<String, String> paramMap;

String param\_name = null;

String count = null;

String min = null;

String max = null;

String label = null;

String alias = null;

String from = null;

List<Element> children;

List<Integer> childrenId;

paramsValues = new HashMap<>();

Queue<Element> elementsQueue = new LinkedList();

Queue<Integer> idsQueue = new LinkedList();

List<String> param\_names;

List<String> param\_values;

List<String> params\_values;

SAXBuilder builder = new SAXBuilder();

File xmlFile = new File(filename);

try {

document = builder.build(xmlFile);

rootNode = document.getRootElement();

//Get storages

storagesNode = rootNode.getChild(Constants.STORAGES\_TAG\_NAME);

for(Element storageNode : (List<Element>)storagesNode.getChildren()){

storageClass = storageNode.getAttributeValue("class");

storageParams = new HashMap<>();

for(Element storageParam : (List<Element>)storageNode.getChildren()){

storageParamName = storageParam.getAttributeValue("name");

storageParamValue = storageParam.getAttributeValue("value");

storageParams.put(storageParamName, storageParamValue);

}

Storage storage = new Storage(storageClass, storageParams);

dataStorages.add(storage.getStorage());

}

/\*storageNode = rootNode.getChild(Constants.STORAGE\_TAG\_NAME);

storageClass = storageNode.getAttributeValue("class");

storageParams = new HashMap<>();

for(Element storageParam : (List<Element>)storageNode.getChildren()){

storageParamName = storageParam.getAttributeValue("name");

storageParamValue = storageParam.getAttributeValue("value");

storageParams.put(storageParamName, storageParamValue);

}

Storage storage = new Storage(storageClass, storageParams);

dataStorage = storage.getStorage(); \*/

counterSet = new SimpleCounterSet();

for (IDataStorage dataStorage : dataStorages){

dataStorage.setCounterSet(new SimpleCounterSet());

}

DataStreamImpl dataStream = new DataStreamImpl(Constants.CSV\_OUTPUT\_FILE\_NAME, stream, dataStorages);

//Get parameters values

paramsNode = rootNode.getChild(Constants.PARAMS\_TAG\_NAME);

for(Element param : (List<Element>)paramsNode.getChildren()){

param\_name = param.getAttributeValue("name");

param\_values = new ArrayList<>();

for(Element value : (List<Element>)param.getChildren()){

param\_values.add(value.getText());

}

paramsValues.put(param\_name, param\_values);

}

param\_names = new ArrayList<>(paramsValues.keySet());

params\_values = new ArrayList<>();

generateParams(param\_names, params\_values, "");

//Generate burden

int currentId;

int parentId;

int tempId;

for(String params : params\_values){

currentId = 0;

parentId = 0;

tempId = parentId;

tags = new HashMap<Integer, Burden>();

aliases = new HashMap<String, Integer>();

paramMap = new HashMap<>();

for(String s : params.substring(0, params.length() - 1).split(";", -1)){

index = s.indexOf("=");

if (index != -1){

paramMap.put(s.substring(0, index), s.substring(index + 1));

}

}

dataNode = rootNode.getChild(Constants.DATA\_TAG\_NAME);

elementsQueue.addAll(dataNode.getChildren());

idsQueue.add(parentId);

childrenId = new LinkedList<Integer>();

while(!elementsQueue.isEmpty()){

element = elementsQueue.poll();

parentId = idsQueue.poll();

count = element.getAttribute("count") != null ? element.getAttributeValue("count") : null;

count = paramFromMap(paramMap, count);

min = element.getAttribute("min") != null ? element.getAttributeValue("min") : null;

min = paramFromMap(paramMap, min);

max = element.getAttribute("max") != null ? element.getAttributeValue("max") : null;

max = paramFromMap(paramMap, max);

label = element.getAttribute("label") != null ? element.getAttributeValue("label") : null;

label = paramFromMap(paramMap,label);

if (element.getAttribute("alias") != null){

alias = element.getAttributeValue("alias");

alias = paramFromMap(paramMap, alias);

if (aliases.containsKey(alias)){

throw new Exception("This alias already exists");

} else {

aliases.put(alias, currentId);

}

} else {

alias = null;

}

from = element.getAttribute("from") != null ? element.getAttributeValue("from") : null;

from = paramFromMap(paramMap, from);

burden = new Burden(currentId, parentId, element.getName(), count, label, alias, from, min, max);

if ((currentId != 0)){

if (parentId == tempId){

childrenId.add(currentId);

} else {

tags.get(tempId).setChildrenId(childrenId);

childrenId = new LinkedList<Integer>();

childrenId.add(currentId);

tempId = parentId;

}

}

tags.put(currentId, burden);

children = (List<Element>)element.getChildren();

for (Element e : children){

elementsQueue.add(e);

idsQueue.add(currentId);

}

currentId++;

}

if (elementsQueue.isEmpty()){

tags.get(tempId).setChildrenId(childrenId);

}

Burden rootBurden = tags.get(0);

List<Pair<String, String>> result = generate(rootBurden);

result.add(0, new Pair<String, String>(Constants.LABEL\_COMMAND\_NAME, "Params:" + params));

for(Pair<String, String> pair : result){

switch (pair.getKey()){

case Constants.INSERT\_COMMAND\_NAME:

dataStream.set(Integer.parseInt(pair.getValue()));

break;

case Constants.SELECT\_COMMAND\_NAME:

dataStream.get(Integer.parseInt(pair.getValue()));

break;

case Constants.REMOVE\_COMMAND\_NAME:

dataStream.remove(Integer.parseInt(pair.getValue()));

break;

case Constants.LABEL\_COMMAND\_NAME:

dataStream.label(pair.getValue());

break;

default:

break;

}

}

dataStream.flush(params);

}

dataStream.close();

} catch (JDOMException e) {

e.printStackTrace();

//log.error(new MasterWorkException(e.getMessage(), e));

} catch (IOException e) {

e.printStackTrace();

//log.error(new MasterWorkException(e.getMessage(), e));

} catch (Exception e) {

e.printStackTrace();

//log.error(new MasterWorkException(e.getMessage(), e));

}

}

private String paramFromMap(Map<String, String> paramMap, String param){

if (param == null){

return null;

}

String result = param;

String paramName;

if(param.startsWith("%") && param.endsWith("%")){

paramName = param.substring(1, param.length() - 1);

result = paramMap.get(paramName);

}

return result;

}

}

public class DataStreamImpl implements IDataStream {

private static final Logger log = Logger.getLogger(DataStreamImpl.class);

private boolean first = true;

private IDataStream dataStream;

private List<IDataStorage> dataStorages;

private PrintWriter out = null;

public DataStreamImpl(String filename, IDataStream dataStream, List<IDataStorage> dataStorages) {

this.dataStream = dataStream;

this.dataStorages = dataStorages;

//this.counterSet = counterSet;

//dataStorage.setCounterSet(counterSet);

try {

out = new PrintWriter(filename);

} catch (FileNotFoundException e) {

log.error(new MasterWorkException(e.getMessage(), e));

}

}

@Override

public void label(String label) {

if (dataStream != null){

dataStream.label(label);

}

}

@Override

public void get(int value) {

if (dataStream != null){

dataStream.get(value);

}

for (IDataStorage dataStorage : dataStorages)

if (dataStorage != null){

dataStorage.get(value);

}

}

@Override

public void set(int value) {

if (dataStream != null){

dataStream.set(value);

}

for (IDataStorage dataStorage : dataStorages)

if (dataStorage != null){

dataStorage.set(value);

}

}

@Override

public void remove(int value) {

if (dataStream != null){

dataStream.remove(value);

}

for (IDataStorage dataStorage : dataStorages)

if (dataStorage != null){

dataStorage.remove(value);

}

}

public void flush(String params){

Map<String, String> paramsMap = new HashMap<>();

Map<String, String> storageParams;

String[] paramsMas = params.split(";");

String[] temp;

for(String param : paramsMas){

temp = param.split("=");

paramsMap.put(temp[0], temp[1]);

}

StringBuilder builder = new StringBuilder();

if (first){

for(String key : paramsMap.keySet()){

builder.append("Param:").append(key).append(",");

}

builder.append("Storage").append(",");

builder.append("Storage Params").append(",");

builder.append(ICounterSet.COMPARE)

.append(",")

.append(ICounterSet.ASSIGN)

.append("\r\n");

first = false;

}

IDataStorage bestDataStorage = null;

int countersBest = 0;

int countersCurr = 0;

for (IDataStorage dataStorage :dataStorages){

if (bestDataStorage == null){

bestDataStorage = dataStorage;

} else {

countersBest = bestDataStorage.getCounterSet().get(ICounterSet.COMPARE)

+ bestDataStorage.getCounterSet().get(ICounterSet.ASSIGN);

countersCurr = dataStorage.getCounterSet().get(ICounterSet.COMPARE)

+ dataStorage.getCounterSet().get(ICounterSet.ASSIGN);

if (countersCurr < countersBest){

bestDataStorage = dataStorage;

}

}

}

for(String key : paramsMap.keySet()){

builder.append(paramsMap.get(key)).append(",");

}

builder.append(bestDataStorage.getClass().getCanonicalName()).append(",");

storageParams = bestDataStorage.getStorageParams();

if (storageParams != null){

for (String sParam : storageParams.keySet()){

builder.append(sParam).append(";").append(storageParams.get(sParam));

}

}

builder.append(",");

builder.append(bestDataStorage.getCounterSet().get(ICounterSet.COMPARE)).append(",")

.append(bestDataStorage.getCounterSet().get(ICounterSet.ASSIGN))

.append("\r\n");

out.println(builder.toString());

for (IDataStorage dataStorage : dataStorages){

dataStorage.clear();

dataStorage.setCounterSet(new SimpleCounterSet());

}

}

public void close(){

if (out != null){

out.flush();

out.close();

}

}

}

public class LogFileWriter implements IDataStream {

private static final Logger log = Logger.getLogger(LogFileWriter.class);

private PrintWriter out = null;

public LogFileWriter(String filename)

{

try {

out = new PrintWriter(filename);

} catch (FileNotFoundException e) {

log.error(new MasterWorkException(e.getMessage(),e));

}

}

public void close(){

if (out != null){

out.flush();

out.close();

}

}

@Override

public void get(int value) {

//To change body of implemented methods use File | Settings | File Templates.

if (out != null){

out.println("<" + Constants.SELECT\_COMMAND\_NAME + "> <"+ value +">");

}

}

@Override

public void set(int value) {

//To change body of implemented methods use File | Settings | File Templates.

if (out != null){

out.println("<" + Constants.INSERT\_COMMAND\_NAME + "> <"+ value +">");

}

}

@Override

public void remove(int value) {

//To change body of implemented methods use File | Settings | File Templates.

if (out != null){

out.println("<" + Constants.REMOVE\_COMMAND\_NAME + "> <"+ value +">");

}

}

@Override

public void label(String label) {

//To change body of implemented methods use File | Settings | File Templates.

if (out != null){

out.println(label);

}

}

}

public class LogFilePlayer {

private static final Logger log = Logger.getLogger(LogFilePlayer.class);

public void playFile(String filename, IDataSet storage)

{

String line, command, value;

int command\_si, command\_ei;

int value\_si, value\_ei;

try {

BufferedReader br = new BufferedReader(new FileReader(filename));

while( (line = br.readLine()) != null){

if (line.startsWith("<")){

command\_si = line.indexOf("<") + 1;

command\_ei = line.indexOf(">");

command = line.substring(command\_si, command\_ei);

value\_si = line.lastIndexOf("<") + 1;

value\_ei = line.lastIndexOf(">");

value = line.substring(value\_si, value\_ei);

switch (command){

case Constants.INSERT\_COMMAND\_NAME:

storage.set(Integer.parseInt(value));

break;

case Constants.SELECT\_COMMAND\_NAME:

storage.get(Integer.parseInt(value));

break;

case Constants.REMOVE\_COMMAND\_NAME:

storage.remove(Integer.parseInt(value));

break;

default:

break;

}

}

}

br.close();

} catch (FileNotFoundException e) {

log.error(new MasterWorkException(e.getMessage(), e));

} catch (IOException e) {

log.error(new MasterWorkException(e.getMessage(), e)); //To change body of catch statement use File | Settings | File Templates.

}

}

}