Computer Science Center

Отчет о курсовом проекте

Курс: Базы данных.

Выполнил: Суворов Егор Фёдорович

Санкт-Петербург, осень 2013

1 Введение

1.1 Постановка задачи

В процессе курса было предложено реализовать собственное хранилище данных. Функциональность требовалось наращивать постепенно, в соответствии с выдаваемыми Feature Request.

1.2 Используемые инструменты

В качестве языка реализации требовалось использовать Scala или Java. Из-за незнания Scala основным инструментом для работы стал язык Java. В качестве среды разработки использовалась IntelliJ IDEA Community Edition как один из лучших инструментов для разработки на Java.

Перед началом работы я не был знаком с юнит-тестированием и системами автоматической сборки для Java. Опыта работы над чем-то, отличным от олимпиадных задач, также не было. Из-за чётких требований к их наличию проблема выбора не стояла — я решил использовать JUint и Maven последних версий.

В процессе потребовалось включить в проект еще несколько сторонних библиотек: Apache Commons (класс `StopWatch` для точного и удобного измерения времени в процессе тестирования), SnakeYAML (для разбора файлов конфигурации на языке YAML) и Logback (для логирования). Подробнее о необходимости каждой библиотеки будет рассказано позднее.

Для организации работы была выбрана система контроля версий Git и сервис Github, так как я уже имел успешный работы с этими инструментами. Никаких серьёзных поводов для перехода на Mercurial или Bitbucket у меня не было.

1.3 Результаты

Было реализовано распределённое хранилище пар «ключ — значение» с поддержкой стандартных запросов CRUD с доступом через консоль. Для получения/изменения данных требуется установление одного сетевого соединения и один запрос к серверу, адрес которого прописан в конфигурационном файле каждого клиента. Поддерживаются произвольные ключи и данные. Каждый отдельный сервер хранит свои данные в одном файле. Такие файлы зависят только от версии программы и могут быть легко перемещены на другие машины.

Основная функциональность проверяется юнит-тестами. Также были проведены нагрузочные тесты и тесты с одновременнами запросами в несколько потоков.

1.4 Структура отчёта

В следующих разделах я опишу все этапы работы в хронологическом порядке: какие были поставлены задачи, какие методы были придуманы и реализованы, удалось ли достичь цели.

В последнем разделе будет указано на совершённые ошибки и будут еще раз кратко описаны все возможности получившейся системы.

2 Начало работы. FR0

2.1 Предыстория

Обязательным нулевым заданием в курсе была простейшая реализация «телефонной книги», которая должна была способна выполнять четыре операции CRUD, иметь консольный/сетевой интерфейс. Также было необходимо реализовать сохранение данных на диск, чтобы была возможность перезапуска основного сервера.

После окончания deadline для данного задания оказалось, что его выполнили всего четыре человека. В связи с этим преподаватель принял решение сделать следующее задание (FR1) обязательным и не приносящим баллов для всех невыполинвших FR0.

2.2 Постановка требований

В начале работы я решил поставить себе несколько требований, которым должно удовлетворять моё приложение на всех или почти всех этапах работы:

1. Корректность. Даже в случае внезапного падения приложения, старые данные не должны теряться, а файлы с данными должно быть можно просто перенести на другой компьютер и сразу же загрузить в другую копию.

2. Don't Repeat Yourself.

3. Каждый класс отвечает за чётко выделенный кусок функциональности, причём за как можно меньшую по смыслу. Это требование призвано поддерживать тестируемость кода и сдерживать рост классов. Однако в некоторых случах разделение класса на несколько может порождать еще десять килобайт связующего кода — таких случаев следует избегать.

4. Гарантированная скорость. Приложение должно быть как можно быстрее, использовать самые быстрые способы взаимодействия с файлами. Слово «гарантированная» обозначает, что недопустимо использование алгоритмов с хорошим амортизационным временем работы и плохим неамортизационным (скажем, Splay-дерево).

5. Нетривиальные места и инварианты классов должны проверяться при помощи `assert`'ов или выбрасывания `RuntimeException` в «невозможных» ситуациях. Это сильно упрощает отладку и проверку корректности, так как любое нарушение инварианта становится заметно намного раньше проверки ответа.

2.3 Первое хранилище и начала архитектуры

Из-за отсутствия опыта работы с Maven, JUnit и разработки больших проектов первую версию хранилища было решено сделать максимально простой. Это позволило мне сосредоточиться на изучении юнит-тестирования и Maven вместо отладки сложного движка.

С первых минут стало ясно, что проект будет большой и требуется разбивать код на как можно более независимые классы. Первым делом было принято идеологическое разделение всех классов на два пакета:

1. `net.yeputons.cscenter.dbfall2013.engines` — сюда помещаются различные классы, имеющие отношение к хранению и получению данных. Единственный способ взаимодействия с ними — прямой вызов различных методов Java.

2. `net.yeputons.cscenter.dbfall2013.clients` — в этом пакете содержатся «front-end» классы, назначение которых - предоставление удобного интерфейса доступа к различным движкам в различных конфигурациях. Например, по сети, через консоль и тому подобное.

2.3.1 Создание интерфейса и базового движка

Первым делом я решил описать единый интерфейс (`DbEngine`) для всех движков, которых планировалось реализовать несколько (от простых к сложным). Так как первым заданием было key-value хранилище, то за основу был взят стандартный интерфейс `Map`. Первой попыткой было использование интерфейса `Map<byte[], byte[]>`, однако это решение было признано неудачным из-за отсутствия встроенного компаратора для `byte[]` (требуется использование методов класса `Arrays`), что приводило к неработоспособности простейшего движка на основе `TreeMap<byte[], byte[]>`, что категорически меня не устраивало — первая версия должна была занимать несколько строчек.

К сожалению, сразу же всплыли проблемы такого решения: в интерфейсе `Map` имеется большое количество методов, не попадающих в CRUD (например, удаление всех элементов, получение количества ключей в хранилище, получение полного списка ключей). Некоторые методы легко выразились через другие (например, метод `empty()`, который мог вызвать метод `size()`). Так как наличие подобных методов облегчает процесс тестирования, было решено написать базовый абстрактный класс `SimpleEngine`, который бы выражал максимальное количество операций через тривиальные — получение размера, CRUD, очистка хранилища и получение множества лежащих в хранилище ключей. Такой подход позволил не реализовывать в простых движках редко требуемые операции и получить возможность использовать их ценой быстродействия. Реализации класса `SimpleEngine` заняла менее двух килобайт.

Далее требовалась какая-нибудь реализация хранилища, чтобы можно было начать разрабатывать клиент. В качестве такой реализации прекрасно подошла «обёртка» над `HashTree`, названная `InMemoryEngine` и занимающая порядка килобайта. Этот движок уже мог выполнять все операции, но не сохранял данные между запусками.

2.3.2 Первый клиент

Второй необходимой частью кода было создание клиента, который мог бы работать с произвольным движком и предоставлять единый интерфейс взаимодействия. Так как для взаимодействия с консольным приложением не требуется ничего, кроме этого приложения, было решено не делать сетевой клиент.

Первая версия консольного клиента умела создавать движок (создаваемый класс жёстко «зашит» в код клиента) и выполнять несколько простых консольных команд: `quit`, `help`, `clear`, `size`, `keys`, `put`, `get`, `del`. Предполагалось, что каждая команда занимает одну строку, а ключи и данные представляют собой строки без спецсимволов и пробелов. Для удобства была добавлена поддержка escape-последовательностей (перевод строки, пробел и обратный слэш), что, впрочем, не пригодилось. Хочется обратить внимание, что CRUD команды `create` и `update` были для удобства и простоты объединены в одну — `put`.

Таким образом, разбор очередной команды сводился к чтению строки, вызову метода `split` и сравниванию первого token со всеми возможными командами. Первая версия занимала около четырёх килобайт.

2.4 Изучение новых технологий

2.4.1 Первые тесты

На данном этапе был написан только один тест — `SimpleEngineTest`, который выполнял десяток разных команд и проверял размер хранилища и возвращаемые им значения. Целью данного теста была базовая проверка движка «на адекватность», поэтому никаких специальных хитрых случаев (кроме `null`-значений) не было придумано.

2.4.2 Maven

Так как вся работа производилась в IDEA, потребность в системе сборки возникла лишь ближе к отправке задания на проверку. В интернете были найдены достаточно простые инструкции по настройке и первый `pom.xml` содержал лишь зависимость от JUnit и команду для сбора jar-файла с указанным главным классом (`ConsoleClient`).

Таким образом, после выполнения команды `mvn package` автоматически компилировались исходные коды, запускался единственный тест и собирался jar-файл, который можно было запустить командой «`java -jar *.jar`».

2.5 Второй движок и клиент

2.5.1 Сохранение данных на диск

Единственным оставшимся пунктом для успешной сдачи FR0 оставалось сохранение данных на диск. Для этого был создан новый класс `LogFileEngine`, наследующийся от `InMemoryEngine`. Это хранилище просто записывало в файл последовательность совершённых операций `put` и `del` в хронологическом порядке. Таким образом, для загрузки требовалось просто прочитать из файла и проэмулировать все изменения, начиная с пустого хранилища. Это, разумеется, не самый быстрый, но достаточно простой способ.

Каждая запись `put` хранится в следующем формате: сначала идёт little-endian четырёхбайтное число — длина ключа в байтах, затем сам ключ. После этого в аналогичном формате записано новое значение. Формат записи `del` точно такой же, но вместе длины значения записано число `-1`.

В тот момент мне показалось, что простой записи таких элементов подряд может быть недостаточно для поддержания корректности. Поэтому я добавил в начало файл еще одно четырёхбайтное число — количество корректных записей (изначально равное нулю). После этого для записи очередной команды в лог требовалось выполнить две вещи: дописать в конец файла данные и изменить первые четыре байта файла. Для этого прекрасно подошёл стандартный класс `RandomAccessFile`

2.5.2 Изменение юнит-теста

После возникновения второго движка возникла необходимость в его тестировании. Заниматься копированием кода не хотелось и я изучил параметризованные тесты JUnit. Теперь в статическом методе класса `SimpleEngineTest` создавались несколько движков и каждый прогонялся на полном наборе простых тестов.

2.6 Измерение скорости

На этом этапе мне уже стало интересно, с какой скоростью может производиться запись в хранилище на основе `LogFileEngine`. Для этого был написан класс `BenchmarkClient`, выполняющий вставки случайных ключей и данных длиной в десять байт в хранилище. Тестирование проводилось на ноутбуке с SSD и процессором Intel Core i5 с частотой 1.7 ГГц. Получившаяся производительность — порядка 7000 добавлений в секунду.

3 FR1

3.1 Постановка задачи и возможные пути решения

В предыдущих реализациях все данные в процессе работы лежат в памяти и работа с файлами требуется только для сохранения на случай прерывания работы. Это позволяет использовать встроенные в Java структуры данных и использовать весьма простые форматы файлов, так как считать нужно будет один раз и целиком. С другой стороны, такой подход неприменим при больших объёмах данных и ограниченной ОЗУ.

Так как приложение должно работать гарантированно быстро, требуется хранить в файле некую структуру данных, обеспечивающую быстрый доступ к ключам и значениям. Одной из первых моих мыслей было заменить ключи на их хэши, чтобы сделать все ключи одинаковой длины и равномерно распределить. Это можно делать, так как хранилище ничего не гарантирует про относительный порядок ключей (и даже не гарантирует, что он сохраняется). Я выбрал SHA-1 как довольно популярный и старый алгоритм, для которого пока еще не найдено коллизий (по этой причине отпал md5).

Это было первое упрощение: все ключи теперь являются 160-битными числами и равномерно распределены. Разумеется, исходные ключи всё равно хранятся и проверяются на случай коллизий (в случае таковой приложение движок выбрасывает `RuntimeException` с комментарием). После этого я составил небольшой список структур данных, которые я знаю:

1. Красно-чёрное дерево. Гарантирует время работы, но требует $\log_2 N$ считываний с жёсткого диска. Здесь и далее N — число элементов в хранилище.
2. Splay-дерево. Отпадает, так как одна операция легко может занять линейное время.
3. Декартово дерево. Просто пишется, гарантирует время работы в среднем случае, но имеет высоту порядка $2\log_2 N$, что требует большого количества чтений с жёсткого диска.
4. B-дерево и его разновидности. Требуют порядка $\log_k N$ считываний с жёсткого диска, где k можно варьировать.
5. Бор (trie). Похож на B-дерево наличием большого количества детей у вершины, но намного проще с точки зрения реализации, так как не требуется перебалансировка дерева.

Я посчитал, что двоичные деревья будут намного медленее B-дерева и бора, поэтому оставалось выбрать из двух пунктов. Бор показался проще с точки зрения кода и так начал появляться класс `HashTrieEngine`

3.2 Первая реализация `HashTrieEngine`

3.2.1 Напоминание про бор

Бор — это корневое дерево, где на каждом ребре написан символ (в моей реализации — число от 0 до 255). Каждой вершине соответствует «строка», которую можно прочитать, спускаясь от корня вниз к вершине. В листах можно хранить данные, соответствующие ключам.

3.2.2 Реализация

Так как подгрузка данных из файла в любом случае потребовалось бы, я решил на ней и остановиться, не добавляя вначале никакого кэша в памяти. Это позволило сократить размер кода и ограничиться константным размером памяти независимо от количества элементов в множестве.

Для доступа к файлу с данными использовался класс `RandomAccessFile`. Формат файла был следующим (все числа по умолчанию считаются знаковыми little-endian):

Начало	Конец	Назначение
0	3	Количество «живых» элементов в хранилище
4	1027	Описание корневой вершины бора
1028	...	Внутренние вершины

Все не-листовые вершины (включая корневую) описываются следующим образом:

Начало	Конец	Назначение
0	3	Смещение потомка по ребру «0» (в байтах, от начала файла, или ноль, если такого потомка нет)
4	7	Смещение потомка по ребру «1»
⋮	⋮	⋮
1000	1023	Смещение потомка по ребру «255»

Любой лист дерева располагается на расстоянии в 20 рёбер от корня. Формат листа отличается, так как он должен хранить данные:

Начало	Конец	Назначение
0	3	длина *значения* элемента (X)
4	$3 + X$	значение элемента
$4 + X$	$7 + X$	длина *ключа* элемента (Y)
$8 + X$	$7 + X + Y$	ключ элемента

Хранение ключа является избыточным, если мы не верим в коллизии и не требуется получение всех ключей множества. К сожалению, это не наш случай. Тем не менее, для ускорения чтения именно значение элемента хранится раньше — тогда больше вероятность попасть в кэш диска (например, если предыдущая прочитанная вершина стояла перед текущей в файле). Стоит отметить, что для последующих оптимизаций хранение ключа стало существенным требования.

При удалении ключа из хранилища лист помечается как удалённый. Формат удалённого листа очень прост: описание ключа отсутствует, а в поле «длина значения» записано «-1». Мне показалось это наиболее простым с точки зрения реализации решением.

При изменении значения по ключу создаётся новая листовая вершина, даже если новые данные короче старых. Это было исправлено в следующих реализациях.

3.3 Прочие обновления

Также в процессе работы над FR1 были сделаны следующие незначительные улучшение, не относящиеся к движку:

1. Добавлена поддержка запуска приложения из системы сборки командой «`mvn exec:java`»
2. Добавлен интерфейс `FileStorableDbEngine` с методами `flush` и `close`
3. Добавлена опция `-batch` в клиент для отключения вывода на экран приглашения и работы в batch-режиме
4. Исправлен баг с падением клиента при получении EOF

3.4 Новый тест

С появлением интерфейса `FileStorableDbEngine` стало ясно, что требуется специальные тесты для проверки корректности сохранения и восстановления данных. Таким тестом стал `FileStorableDbEngineTest`, состоящий из стресс-теста, выполняющего случайные операции (создание, изменение, удаление, пересоздание движка) с `LogFileEngine`/`HashTrieEngine`. Для контроля корректности использовался стандартный `HashMap`.

К сожалению, запуск показал, что текущая реализация `HashTrieEngine` работает непростительно долго: 20 обращений в случайные места файла — не шутка.

3.5 Ускорение

Первой и наболее очевидной была асимптотическая оптимизация: незачем хранить весь путь от корня до листа, достаточно остановиться в первой вершине, у которой нет прочих ответвлений.

Несмотря на простое описание, это оказалось нетривиальной задачей. В процессе для упрощения работы были созданы классы `TrieNode`, `LeafNode`, `InnerNode` с методами чтения и записи в/из файла. Все классы, относящиеся к бору, были вынесены в отдельный пакет.

Следующая оптимизация использовала тот факт, что запись в конец файла выполняется дольше, нежели изменение уже готовых кусков. Теперь движок выделял файл «с запасом», при необходимости удваивая размер файла. Разумеется, все эти обновления потребовали изменить формат хранения данных. Теперь в него добавился номер версии, но, как и прежде, никаких сложных оптимизаций не производилось и формат остался довольно прямым и понятным:

Начало	Конец	Назначение
0	3	Версия (строка из четырёх байт «`YDB1`», где `1` — символ с кодом 1)
4	7	Количество используемых байт в файле
8	1032	Описание корневой вершины
1033	...	Описания остальных вершин

Так как любая вершина теперь могла служить как внутренней, так и корневой, независимо от глубины, в начале каждого описания появился дополнительный байт, описывающий тип вершины — листовая (т.е. содержащая ключ и значение) или внутренняя (т.е. имеющая более одного ребёнка). В остальном формат остался прежним. Также было улучшено обновление данных — если новые данные не длинее старых, их можно записать поверх старых.

Следующей оптимизиацией стало введение отдельного метода для подсчёта размера множества при инициализации вместо вызова функции `keySet()`. Последней оптмизацией стало использование `MappedByteBuffer` (memory-mapped file) напрямую, без прослойки в виде `RandomAccessFile`.

Все эти оптимизации позволили достичь быстродействия в 1000 добавлений случайных десятибайтных значений в секунду.

3.6 Поддержка больших файлов

Следующим большим этапом стала поддержка больших файлов (больше 2GB). Оказалось, что `MappedByteBuffer` не умеет с такими работать. К сожалению, готовой библиотеки для быстрой работы с огромными файлами я не нашёл и по советам со StackOverflow был написан класс `HugeMappedFile`. Этот класс создавал массив из `MappedByteBuffer`, каждый из которых отвечал за кусок файла размером 2^{30} байт. При обращении к какому-либо байту/последовательности байт вызывалась функция соответствующего куска memory-mapped file.

Следующей задачей стала замена четырёхбайтных сдвигов внутри файла на восьмибайтные. К сожалению, это не принималось во внимание раньше, поэтому замена заняла некоторое время. Версия была изменена на `YDB2`

После поддержки больших файлов стало возможно увеличить число элементов в нагрузочном тесте до $3 \cdot 10^6$, на моём компьютере этот тест отрабатывал за ≈ 5 минут вместе с проверкой добавленных данных. Сбрасывание дискового кэша не производилось вообще, это позволило достичь цифры в 20 000 добавлений в секунду.

3.7 Итоги FR1

FR1 было написано и успешно сдано на два балла, один из которых пошёл в зачёт. Второй не был учтён, так как у меня не было выполнено обязательное FR0 и FR1 его замещало. Я считаю, что мне в полной мере удалось реализовать свои задумки по поводу FR1.

4 FR2

4.1 Постановка задачи и размышления

В этом задании требовалось реализовать минимальную поддержку распределённого хранения данных: имеется некоторая статическая конфигурация серверов, известная каждому клиенту. Требуется равномерно распределить нагрузку по серверам.

Так как у меня уже использовались SHA-1 хэши от ключей, было решено использовать их же для балансировки нагрузки. Таким образом автоматически получались равномерность распределения и предпосылки для введения vnodes — каждый сервер отвечал за некоторый отрезок SHA-1 хэшей. Разумеется, тогда некоторые из детей корневой вершины на сервере могли оказаться недостижимыми, но было решено с этим не бороться в угоду простоте.

Разумеется, конфигурацию серверов нужно где-то хранить. Мне не хотелось жёстко прописывать её в код, поэтому требовались настроечные файлы. XML мне показался слишком громоздким и неудобным, и я решил попробовать язык YAML. Вот пример конфигурационного файла на YAML:

```
sharding:
  - startHash: "00"
    address: localhost:23917
  - startHash: "40"
    address: localhost:23918
  - startHash: "80"
    address: localhost:23919
  - startHash: "C0"
    address: localhost:23920
```

4.2 Базовые классы для Sharding

Был добавлен новый пакет `net.yeputons.cscenter.dbfall2013.scaling`, содержащий следующие классы:

1. `Router` — расширение класса `SimpleEngine`, устанавливает соединения с серверами, посылает им команды и принимает ответы. На каждый запрос инициируется новое подключение.
2. `RouterCommunicationException`
3. `ShardDescription` — вспомогательный класс из трёх полей: `address`, `startHash` и `endHash`
4. `ShardingConfiguration` — описание конфигурации серверов. Этот класс умеет по хэшу ключа возвращать соответствующий `ShardDescription`
5. `ShardingNode` — сервер, обслуживающий клиентов на основе `HashTrieEngine`. Именно этот класс является «входной точкой» для процесса ноды. В начале работы создаётся один класс `HashTrieEngine`, а для каждого нового соединения (клиента) создаётся отдельный поток, обрабатывающий его запросы.

Протокол сетевого взаимодействия довольно прост:

1. Сразу после установки соединения клиент может отправлять команды, никакой авторизации нет. Сервер может только посылать ответы
2. Каждая команда — это три байта, обычно сокращение от названия команды (`put`, `del`, `get`, `hi!`).
3. В ответ на команду сервер присылает двухбайтный статус (`no` или `ok`). В случае ответа `no` далее будет передан массив — строка с описанием произошедшей ошибки в кодировке ASCII
4. Массивы передаются следующим образом: вначале идёт четырёхбайтное число — количество байт в массиве, далее следуют сами байты
5. Таким образом даже при обрыве соединения клиент может понять, что данные были переданы не до конца.

4.3 Изменения для stand-alone запуска нод

Так как приложение уже могло быть запущено в минимум двух режимах, был добавлен специальных класс `Bootstrapper`, который исходя из параметров командной строки выбирал, какой класс запустить дальше — `ConsoleClient`, `ShardingNode` или же все ноды из заданной конфигурации одновременно. Последняя опция была добавлена для упрощения тестирования на одной машине.

4.4 Возникшие сложности и оптимизация

Первой вознишей сложностью было требование интерфейса `Map` для метода `put`: он должен был возвращать предыдущее значение элемента. Так как выполнение этого требования обозначало бы добавление ненужного в большинстве случаев траффика, было решено возвращать `null` всегда, таким образом нарушив контракт `Map`.

Следующей сложностью был сбор текущего списка элементов со всех нод. Я решил не заморачиваться с соответствием версий, атомарностью и итераторами, поэтому сейчас в методах `keySet` и `entrySet` создаётся `Set` и все ключи/элементы со всех нод просто собираются в память. Разумеется, на больших объёмах данных это не работает, но довольно полезно для отладки.

Ключевым моментом оптимизации стало кэширование соединений с нодами: никакие соединения не обрываются после окончания обработки команды. Вместо этого новое соединение складывается в множество активных соединений и, если потребуется установить связь с этой нодой еще раз, будет переиспользовано старое соединение. Так как установка TCP-соединения — длительная операция, данная оптимизация увеличила скорость работы на синтетическом тесте в разы.

4.5 Тестирование

Был написан новый класс для тестирования — `ShardingTest`. Также был введён новый абстрактный класс `AbstractStressTest`, содержащий методы, облегчающие стресс-тестирование (такие как «выполнить случайную операцию»). По умолчанию для теста запускается четыре ноды, обслуживающие одинаковые по размеру сегменты. После этого выполняется два вида стресс-тестов:

1. Однопоточное. Быстро выполняется 350 случайных модификаций с полным сравнением множества элементов с эталоном после каждой из операций при помощи метода `entrySet`. Ключи и значения — случайные последовательности из девяти байт. Предназначено для базовой проверки на правильность ответов на запросы.

2. Многопоточное нагрузочное для проверки на безотказную работу с множеством клиентов. Из-за активного использования `assert`'ов внутри кода данный вида теста также выполняет проверку корректности. Создаётся десять клиентов (больше, чем нод), каждый из которых выполняет 4000 случайных запросов.

Также во всех видах стресс-тестов был улучшен алгоритм генерации случайных операций:

1. С вероятностью 50% добавляется новое случайное значение со случайным ключом

2. С вероятностью 25% удаляется случайный существующий ключ

3. С вероятностью 25% удаляется случайный ключ

Все ключи и значения, как и раньше — случайные последовательности из девяти байт. Такая структура запросов позволило выявить и исправить проблемы, связанные с удалением элементов из хранилища и некорректной работой класса `ShardingNode`.

4.6 Возникшие проблемы и их решения

После запуска тестов обнаружились проблемы с многопоточностью: при наличии одного клиента всё работало, а при большем количестве всё падало по `assert`. Было решено не пытаться добавить многопоточность с поддержкой изменений в `HashTrieEngine`, а запретить одновременный доступ к движку из нескольких потоков одной ноды.

Однако даже такая мера не избавила от постоянных странных ошибок и нарушений инвариантов. После длительного исследования я обнаружил, что используемый мной класса `MessageDigest` не только не является потокобезопасным, но и объявлен как статическое поле. Таким образом, несколько нод, запущенных в одной Java-машине, конфликтовали за этот класс, несмотря на то, что обращения шли к разным движкам. Проблема была решена очевидным методом — объект этого класса сделан нестатическим.

4.7 Добавление логирования

В процессе работы на многопоточными тестами стало ясно, что отлаживать несколько запущенных одновременно приложений путём вывода сообщений в стандартный поток довольно сложно. Преподавателем было предложено добавить логирование и мой выбор пал на систему «logback-classic». Это относительно простой и в то же время расширяемый логгер для Java. В лог выводятся сообщения о новых клиентах, запуске и закрытии нод, об ошибках при обработке сообщений (например, о неудачном соединении с нодой). Также в лог стали выводиться все информационные сообщения с тестов, в том числе и сообщения со статистикой и средней скоростью работы.

4.8 Результаты реализации FR2

FR2 было успешно сдано на один зачётный балл. В процессе работы было исправлено большое количество непотокобезопасных мест и улучшено качество тестов (помимо уже сказанного, были добавлены тесты на очистку хранилища и на корректность работы функций `entrySet`/`keySet`).

5 FR5

5.1 Постановка задачи и подходы

После длительной работы системы некоторая часть данных (удалённые или непереиспользованные) просто занимает место в файле данных, причем в случайных местах. Например, при добавлении большого количества ключей и их последовательном удалении размер файла остаётся большим. Задача — добавить функциональность «дефрагментации» так, чтобы размер файлов не диске несильно превышал суммарный размер живых данных. Разумеется, при этом нода должна продолжить принимать и корректно обрабатывать запросы.

Я рассматривал два принципиальных подхода к решению задачи:

1. «Живая дефрагментация». В этом случае происходит сдвиг всего неиспользуемого пространства в конец файла. Для этого требуется обойти весь бор и отметить используемые части файла, после чего произвести атомарный сдвиг каждой вершины «влево до упора». Это требует создания дополнительного массива большого размера для хранения пометок об используемости/неиспользуемости куска файла.

2. «Копировать и заменить». В этом случае создаётся новых файл, в который поэлементно «с нуля» записываются все живые элементы текущей коллекции. После окончания операции, старый файл заменяется новым. Этот метод требует больше места на диске, однако он показался мне проще идеологически, поэтому и был выбран.

5.2 Улучшение `HashTrieEngine.keySet`

Для реализации любого из подходов требуется поэлементный обход бора. Текущая реализация `keySet` не позволяла этого сделать — она складывала все элементы в память, что было допустимо для отладки, но не на больших данных. Требовался другой подход, позволяющий получить все элементы по одному с небольшими затратами по памяти.

Таким подходом стал новый класс `HashTrieIterator`, который хранит текущую вершину и путь до неё от корня. Этот класс реализует интерфейс `Iterator`, а класс `HashTrieEngine` — интерфейс `Iterable`, таким образом можно с использованием константного объёма памяти обойти все элементы бора. Предполагается, что в процессе обхода бор не меняется, иначе потребовались бы дополнительные ухищрения для сохранения корректности итератора (например, лист мог бы быть заменён на внутреннюю вершину при добавлении очередного элемента), чего не хотелось делать.

Разумеется, методы `keySet` и `entrySet` также были переведены на итераторы и проверены при помощи уже существующих стресс-тестов — однопоточных и многопоточного с несколькими нодами. В процессе тестирования было выявлено несколько ошибок и неточно сформулированных инвариантов, что приводило к падениям программы в половине случаев. Все найденные баги были исправлены и все тесты были пройдены.

5.3 Реализация дефрагментации

Так как дефрагментация может быть длительным процессом, а обработку запросов останавливать нельзя, было придумано обходное решение: на время дефрагментации создаётся дополнительное хранилище в памяти на основе `HashMap` и в него складываются все запросы, пришедшие за время дефрагментации. Таким образом не ухудшается время доступа и нет необходимости в сложных обновлениях итератора для корректной работы с изменяющимся множеством.

В начале работы дефрагментатора создаётся временный `HashTrieEngine` на временном файле и `HashMap` для хранения запросов. После этого при помощи итератора все текущие элементы сохраняются во временное хранилище, к которому затем применяются все пришедшие модификации. На этот период основное хранилище становится совершенно недоступным, однако при условии небольшого количество пришедших запросов операция применения отложенных запросов длится совсем недолго. После приведения временного хранилища в соответствие с реальной картиной мира, старый файл заменяется новым и происходит разблокировка.

Потребовалось модифицировать все методы `HashTrieEngine` для корректной работы в многопоточной среде с поддержкой «отложенных операций».

5.4 Тестирование

Был создан новый класс `HashTrieCompactingTest`. В нём в одном потоке выполняется 5000 случайных модификаций хранилища и после каждой модификации с вероятностью 1% в отдельном потоке запускается дефрагментация. После каждого запроса происходит полное сравнение данных в текущем хранилище и в эталоне. В процессе тестирования было поймано несколько ошибок, связанных с отсутствием необходимых блокировок, так что я считаю данный тест успешно выполнившим свою задачу.

5.5 Менеджер нод

Для удобства запуска дефрагментации и управления нодами был написан класс `ClusterManager` — консольная утилита для наблюдения за кластером. Она позволяет вывести список нод с обслуживаемыми ими хэшами, проверить соединение с каждой из нод, а также запустить, выключить, пропинговать или запустить дефрагментацию на произвольной ноде.

5.6 Результаты выполения FR5

Мной был реализован метод дефрагментации, позволяющий нодам не переставать отвечать на запросы надолго. Также было написаны тесты для эмуляции многопоточных нагрузок, которые помогли отследить нетривиальные ошибки блокировок.

FR5 было сдано на два зачётных балла, что вместе с предыдущими заданиями давало необходимый минимум для получения зачёта.

6 Заключение

6.1 Возможности системы на данный момент

1. Базовый движок:
   (a) В качества базового движка используется бор по SHA-1 хэшам ключей. Ребро соответствует восьми битам хэша.
   (b) Применяется сжатие хвостов путей. Таким образом из каждой вершины, кроме корня, может быть от двух до 256 переходов. Для корня ограничения снизу нет.
   (c) Поддерживаются файлы размера более 4 ГБ (при наличии такой возможности у файловой системы).
   (d) Полностью и без нарушений контракта поддерживается интерфейс `Map<ByteBuffer, ByteBuffer>`.
   (e) `null` в качестве ключа или значения не поддерживается.
   (f) Для быстрого доступа используются memory-mapped files (в том числе для файлов больше 4 ГБ).

   (g) Место под хранилище резевируется заранее, что позволяет обновлять memory mapping логарифмическое от размера файла число раз.
   (h) Получение размера осуществляется без обращения к жёсткому диску
   (i) Все операции используют константное количество ОЗУ
   (j) Поддерживается итерирование по элементам (с константным числом памяти).
   (k) Удаление элементов по итератору и модификация коллекции в процессе итерирования недопустима.
   (l) Инициализация движка происходит за линейное от размера файла время (требуется только для подсчёта количества элементов, которое, к сожалению, не хранится).
   (m) Поддержка дефрагментации «в фоне», с сохранением отклика и кэшированием поступающих запросов в памяти.
   (n) Дефрагментация требует однократного запаса свободного пространства на диске для сохранения новой базы данных.

2. Консольный клиент:
   (a) Подгружает информацию о доступных серверах из файла конфигурации
   (b) Batch-режим и интерактивный режим (с приглашением ввода)
   (c) Встроенная помощь по командам
   (d) Поддержка CRUD-команд и вывода всех элементов из хранилища (в том числе для распределённых систем)

3. Клиент кластера:
   (a) Чтение статической конфигурации из файла
   (b) Установка соединения с нодами по мере необходимости
   (c) Переиспользование уже установленных соединений с нодами
   (d) Корректное закрытие соединений при вызове метода `close`
   (e) Поддержка почти всех методов интерфейса `Map` (метод `put` всегда возвращает `null`)
   (f) Операции `keySet` и `entrySet` требуют размера памяти, пропорционального суммарному объёму элементов.

4. Менеджер кластера:
   (a) Загрузка конфигурации из файла
   (b) Вывод списка нод с их текущим состоянием (онлайн/оффлайн) и диапазонами хэшей, за который они отвечают.
   (c) Выключение нод
   (d) Запуск дефрагментации на ноде

5. Bootstrapper (является главным классом в jar-файле):
   (a) Запуск клиента
   (b) Запуск менеджера
   (c) Запуск одной ноды
   (d) Запуск всех нод на основе конфигурации из файла

6.2 Написанные юнит-тесты

Здесь под «случайными операциями» подразумеваются случайные операции, генерируемые по алгоритму, описанному в 4.4.

1. `SimpleEngineTest` — запускается для каждого из трёх базовых движков. Проверяет корректное выбрасывание исключения при использовании `null` в качестве ключа/значения, методы `clear`, `equals`. Выполняет некоторое количество фиксированных операций добавления/изменения/получения значений, после каждой операции происходит сравнения количества элементов множества с требуемым.

2. `ShardingTest` (однопоточный) — запускает четыре ноды с равномерным распределением ключей и выполняет 350 случайных операций изменения. После каждой операции происходит полное сравнение всего хранилища с эталоном путём вызова методов `keySet` и `entrySet`

3. `ShardingTest` (многопоточный) — запускает четыре ноды с равномерным распределением ключей и десять клиентских потоков. В каждом из потоков выполняется 4000 случайных операций. Проверок не выполняется, корректность проверяется встроенными в движки `assert`'ами.

4. `HashTrieCompactingTest` — выполняется 5000 случайных операций, после каждой происходит полное сравнение всего множества с эталоном. После каждого шага с вероятностью 1% в отдельном потоке запускается дефрагментация.

5. `FileStorableDbEngineTest` (тестирует два типа движков, работающие с файлами) — выполняется 500 случайных операций, после каждой с вероятностью 30% движок уничтожается и создаётся новый на том же файле данных. После каждой операции происходит полное сравнение данных с эталоном.

6. `BigStressTest` — в `HashTrieEngine` кладутся $3 \cdot 10^6$ элементов. После этого в случайном порядке проверяется, что каждый из добавленных элементов действительно лежит в хранилище.

6.3 Допущенные ошибки

1. В самом начале был выбран чрезвычайно обширный интерфейс `Map`, что привело к необходимости нарушать его контракт для быстродействия. Возможно, было бы лучше взять своё сужение данного интерфейса и расширять его по мере необходимости.

2. С самого начала не был продуман механизм работы с большими файлами, что вызвало необходимость перехода с четырёхбайтных сдвигов на восьмибайтные в середине работы и написание собственного класса для работы с большими файлами

3. Из-за незнания механизма memory-mapped files в Java не удалось адекватным образом реализовать отключения mapping файла в память — сейчас все на совести garbage collector.

4. Возможно, было бы лучше воспользоваться встроенными в язык средствами сериализации

5. Не использовались механизмы логирования с самого начала — это удобнее, чем вывод в стандартный поток

6. Не всегда детально продумывался алгоритм перед написанием, что приводило к долгой отладке на больших данных, потому что на маленьких объёмах не воспроизводились крайние случаи.