СОДЕРЖАНИЕ

ВВЕДЕНИЕ ... 6
1. Обзор предметной области ... 7
1.1. Существующие тестирующие окружения ... 7
1.1.1. Synchrobench ... 7
1.1.2. Setbench ... 7
1.1.3. YCSB ... 7
1.2. Недостатки существующих решений ... 8
1.3. Требования к тестирующему окружению ... 8
Выводы по главе 1 ... 9
2. Проектирование тестирующего окружения ... 10
2.1. Сущность Distribution ... 12
2.2. Сущность DataMap ... 12
2.3. Сущность KeyGenerator ... 13
2.4. Сущность ThreadLoop ... 13
Выводы по главе 2 ... 13
3. Пример добавления новой нагрузки ... 15
3.1. Сущность Distribution ... 15
3.2. Сущность DataMap ... 17
3.3. Сущность KeyGenerator ... 18
3.4. Сущность ThreadLoop ... 20
Выводы по главе 3 ... 22
4. Реализованные компоненты ... 23
4.1. Сущности Distributions ... 23
4.1.1. Uniform ... 23
4.1.2. Zipfian ... 23
4.1.3. Skewed Uniform ... 23
4.2. Сущности DataMaps ... 23
4.2.1. Id ... 23
4.2.2. Array ... 23
4.2.3. Hash ... 24
4.3. Сущности KeyGenerators ... 24
4.3.1. Default ... 24
4.3.2. Skewed Sets ... 24

4.3.3. Temporary Skewed Sets . . . 25
4.3.4. Creakers and Wave . . . 26
4.3.5. Leafs Handshake . . . 28
4.4. Сущности ThreadLoops . . . 29
4.4.1. Default . . . 29
4.4.2. Temporary Operations . . . 29
Выводы по главе 4 . . . 30
5. Тестирование бинарных деревьев поиска . . . 31
5.1. Реализации . . . 31
5.1.1. Исправление Concurrency-Friendly BST . . . 31
5.2. Результаты экспериментов . . . 32
5.2.1. Uniform и Zipfian . . . 32
5.2.2. Infinite Leafs Handshake . . . 34
5.2.3. Non-shuffle Wave . . . 37
Выводы по главе 5 . . . 39
ЗАКЛЮЧЕНИЕ . . . 41
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ . . . 42

ВВЕДЕНИЕ

В настоящее время существует огромное чисто конкурентных структур данных [1, 2, 5, 8, 9], которые поддерживают множество пар (*ключ, значение*). Такие структуры данных называются ассоциативными массивами. Над ними можно совершать три типа операций:

— операция вставки (`insert`): добавляет ключ с предоставленным значением в набор и возвращает предыдущее соответствующее значение, при его наличии;
— операция удаления (`remove`): удаляет ключ из набора и возвращает предыдущее соответствующее значение;
— операция чтения (`get`): возвращает значение, соответствующее запрошенному ключу.

Операции вставки и удаления так же можно назвать операциями записи.

Так как существует очень много реализаций ассоциативного массива, то как понять, какая из них работает лучше остальных? Стандартным подходом в исследовательских работах, является запуск структуры данных на наборе нагрузок, где нагрузка — это набор операций, выполняемых процессами, на основе какой-то логики. Если структура данных превосходит остальные на всех нагрузках, то есть выполняет все операции быстрее остальных, её можно считать самой эффективной.

Работа имеет следующее содержание: в Главе 1 описаны существующие решения и их проблемы; в Главе 2 описано спроектированное тестирующее окружение; глава 3 описан пример добавления новой нагрузки; в Главе 4 представлены основные реализованные компоненты, которые уже можно использовать для составления нагрузки; в Главе 5 проведены эксперименты на реализованных нагрузках с тремя популярными конкурентными бинарными деревьями поиска.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Для оценки эффективности структур данных используют тестирующие окружения. В этой главе будут разобраны существующие тестирующие окружения для оценки эффективности конкурентных индексов и их недостатки.

1.1. Существующие тестирующие окружения

Самыми популярными тестирующими окружениями для оценки эффективности конкурентных структур данных:
— `Synchrobench` [6];
— `Setbench` [4];
— `YCSB` [3].

Далее разберём их подробнее.

1.1.1. Synchrobench

Тестирующее окружение `Synchrobench` является самым простым из представленных: оно обеспечивает только равномерную нагрузку. В равномерной нагрузке потоки в бесконечном цикле выбирают тип операции, в соответствии вероятностей операций записи и чтения, и в качестве аргумента принимается ключ, сгенерированный случайным равномерным образом из всего заданного диапазона. `Synchrobench` реализован на `Java` и `C++` и предоставляет множество реализаций структур данных.

1.1.2. Setbench

`Setbench`, в отличие от `Synchrobench`, помимо равномерной нагрузки, обеспечивает `Zipfian` нагрузку, где аргументы операций генерируются на основе `Zipfian` распределения [7].

1.1.3. YCSB

`YCSB` переставляет пять предопределенных нагрузок [3]:
— `Update heavy` (A) — 50% операций записи и 50% операций чтения с использованием `Zipfian` распределения ключей;
— `Read heavy` (B) — 5% операций записи и 95% операций чтения с использованием `Zipfian` распределения ключей;
— `Read only` (C) — 100% операций чтения с использованием `Zipfian` распределения ключей;

— Read latest (D) — 5% операций записи и 95% операций чтения, ключи генерируются при помощи Latest с использованием Zipfian распределения, то есть ключи, которые недавно добавили, имеют большую вероятность;
— Short ranges (E) — 95% операций сканирования, то есть обход всех элементов из заданного диапазона, и 5% операций чтения с использованием Zipfian или равномерного распределения ключей.

1.2. Недостатки существующих решений

У представленных выше тестирующих окружений есть свои недостатки. В первую очередь это скудный набор нагрузок. При этом добавление новых нетривиальных нагрузок является довольно непростой задачей. В текущих окружениях, для этого пользователю придётся внимательно изучить код, чтобы при добавлении новой нагрузки избежать ошибок в работе программы. Либо каждый раз создавать копию тестирующего окружения с новой нагрузкой, что так же является неоптимальным решением. Также YCSB заранее генерирует список запросов перед запуском тестирования. Из-за чего есть ограничение в продолжительности работы. Помимо этого, изменение параметров в нагрузках YCSB является сложным — между ними много сложных зависимостей, что ещё больше ограничивает пользователя в экспериментах.

1.3. Требования к тестирующему окружению

Таким образом, нас не устраивают существующие тестирующие окружения и есть необходимость в новом со следующими требованиям:
— Легкость добавления нагрузок;
— Нагрузки выполнялись в течение произвольного времени;
— Нагрузки желательно должны быть скошенными.

Пройдёмся по каждому пункту подробнее.

Благодаря лёгкому добавлению новых нагрузок можно быстрее проводить различные исследования производительности структур: проанализировав работу структуры данных, предположить, на каких нагрузках она будет лучше работать; добавив эту нагрузку в тестирующее окружение и протестировав на ней структуру данных, подтвердить или опровергнуть выдвинутое предположение. Это будет продемонстрировано в Главе 5. Так же пользователь, при выборе структуры данных для своего продукта, зная, с каким нагруз-

ками она столкнётся, может добавить их в тестирующую систему и выбрать себе самую подходящую структуру данных.

Выполнение нагрузки в течение произвольного времени позволяет запускать тестирование с большой продолжительностью, проверяя тем самым постоянность скорости работы. Например, самонастраивающаяся структура данных может со временем адаптироваться к нагрузке и справляться с ней лучше, либо же наоборот из-за её специфичности справляться с ней всё хуже и хуже, из-за чего будет падать производительность.

Важность последнего пункта связана с тем, что в реально жизни редко встречается равномерная нагрузка, в связи с чем существует самонастраивающиеся структуры данных, в которых более частые запросы обрабатываются быстрее [8, 9].

Выводы по главе 1

Таким образом, было выяснено, что существующие тестирующие окружения не являются оптимальными для должного тестирования и изучения работы конкурентных структур данных. Потому есть необходимость в новом тестирующем окружении, которое будет поддерживать лёгкое добавление новых нагрузок, а также реализовать несколько новых бесконечных и желательно скошенных нагрузок.

ГЛАВА 2. ПРОЕКТИРОВАНИЕ ТЕСТИРУЮЩЕГО ОКРУЖЕНИЯ

В этой главе я расскажу о проектировании моего тестирующего окружения. Для удобства добавления новых нагрузок я поделил нагрузку на четыре типа сущностей:

а) `Distribution` эмулирует распределение случайной величины;
б) `DataMap` преобразует индекс в ключ;
в) `KeyGenerator` генерирует ключ, используя сущности `Distribution` и `DataMap`;
г) `ThreadLoop` выражает логику взаимодействия с предоставленной структурой данных и определяет следующий тип операции.

Комбинируя эти сущности тем или иным образом, можно получать различные нагрузки.

Благодаря такому разделению на четыре типа сущностей добавление новых нагрузок становится более простым. В первую очередь за счёт того, что, при добавлении новой нагрузки, можно переиспользовать уже реализованные сущности.

В существующих тестирующих окружениях преобладает структура кода, представленная в рисунке 1, где один класс `ThreadLoop`, используя лишь один генератор случайных чисел, отвечает за весь ход тестирования. Из-за чего, при добавлении новой нагрузки, пользователю приходится воссоздавать всю логику взаимодействия тестирующего окружения со структурой данных. Даже если новая нагрузка является малым расширением старой. В реализованном мною тестирующем окружении, структура кода которой представлен в рисунке 2, каждая сущность отвечает лишь за свою часть, за счёт чего, пользователю нет необходимости, при добавлении новой сущности, думать о работе остальных.

Все нагрузки имеют следующие параметры:

— рабочий диапазон ключей;
— размер структуры данных перед тестированием;
— количество потоков, на которых будет происходить тестирование;
— количество потоков, используемые на этапе заполнения;
— продолжительность тестирования.

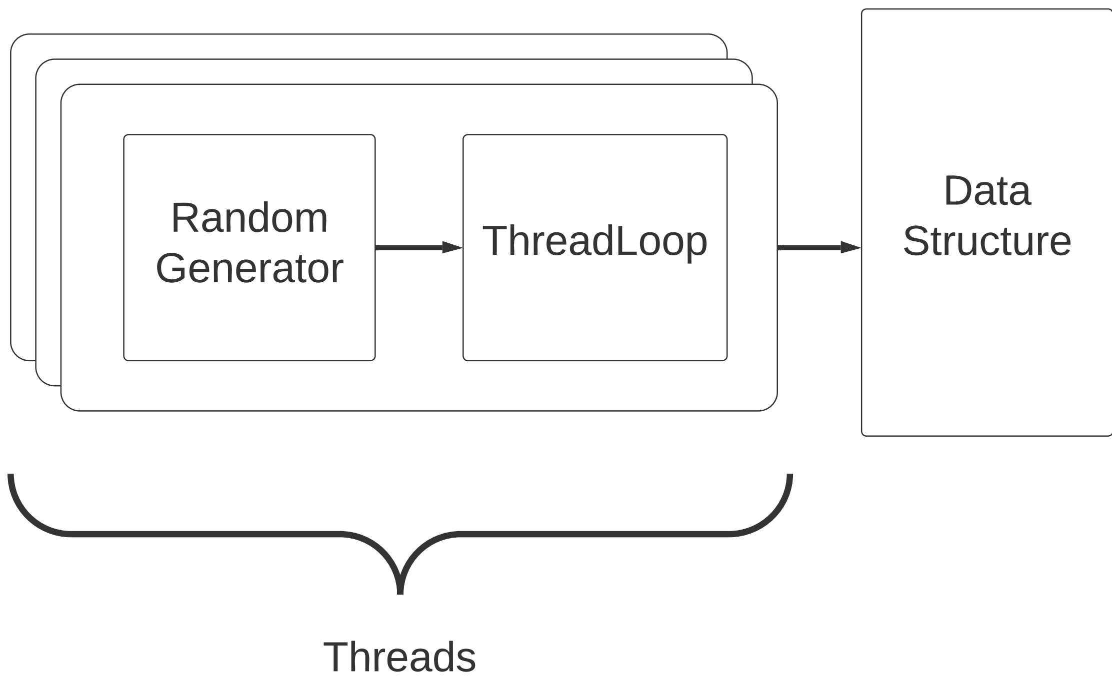

Рисунок 1 – Структура кода в `Synchrobench`

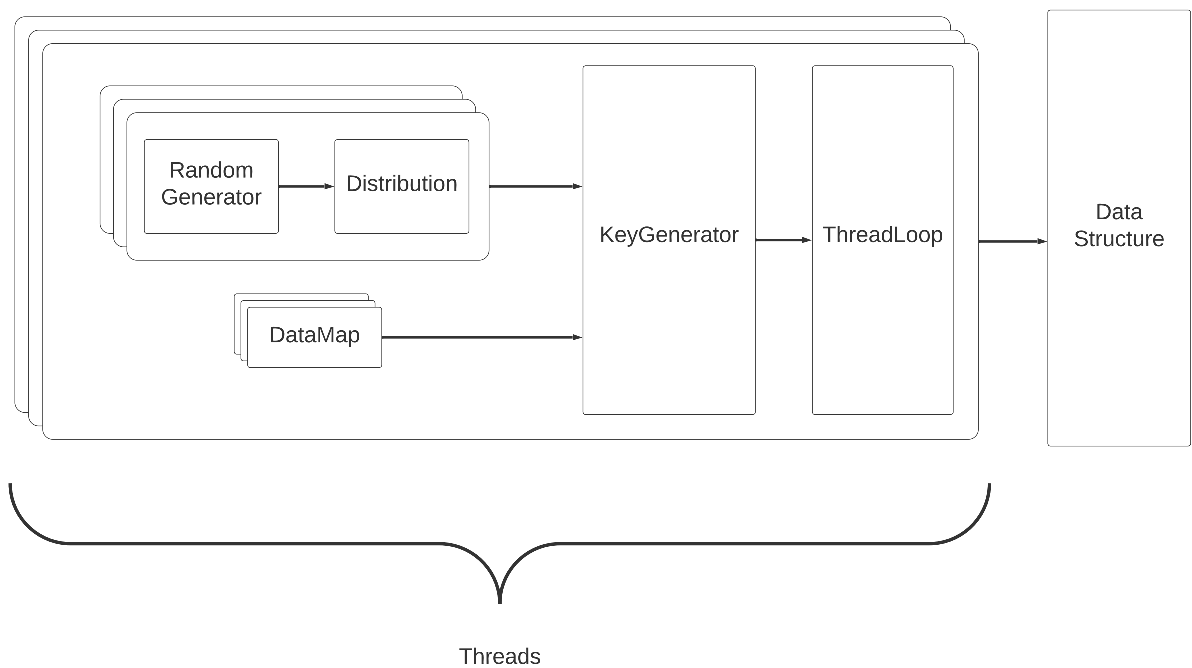

Рисунок 2 – Структура кода в реализованном тестирующем окружении

2.1. Сущность Distribution

Самой низкоуровневой сущностью является `Distribution`. Эта сущность генерирует некую случайную величину, обычно ограниченную некоторым промежутком, которая в последствии будет использоваться в генерации ключа в сущности `KeyGenerator`. `Distribution` реализует интерфейс, представленный на листинге 1. Метод `next()` отвечает за генерацию случайно величины в некотором заранее указанном промежутке по каким-то правилам. Так же существует расширенная версия `MutableDistribution`, представленная на листинге 2. Главное различие в том, что во втором случаем можно изменять диапазон в процессе тестирования.

Листинг 1 – Интерфейс `Distribution`

```
interface Distribution:
    int next()
```

Листинг 2 – Интерфейс `MutableDistribution`

```
interface MutableDistribution extends Distribution:
    void setRange(int range)
    int next(int range)
```

Для упрощенного использования существующих, а также добавления новых распределений, присутствует общий класс `DistributionBuilder`, который разбирает входящие данные, хранит соответствующие параметры, и по ним генерирует соответствующие распределения.

Так, например, `Zipfian Distribution` 4.1.2 эмулирует распределение Ципфа при помощи одного генератора случайных чисел.

2.2. Сущность DataMap

Сущность `DataMap` преобразует индекс в ключ, для этого она реализует интерфейс, представленный на листинге 3. Метод `get(index)` предоставляет по индексу соответствующий ключ.

Так, например, `Array DataMap` 4.2.2 создаёт массив с ключами из всего диапазона ключей, перетасовывает случайным образом и, при выполнении метода `get(int index)`, выдаёт соответствующий индексу значение из массива.

Листинг 3 – Интерфейс `DataMap`

```
interface DataMap:
    K get(int index)
```

2.3. Сущность KeyGenerator

Сущность `KeyGenerator` генерирует ключи для каждого типа операций: `get`, `insert` и `remove`, реализуя интерфейс представленный на листинге 4. Последний метод используется для предварительного заполнения структуры данных перед тестированием. Он необходим, для генераторов, которые требуют особого предварительного заполнения.

Поскольку генератор ключей работает с распределениями, которые возвращают случайную величину. Впоследствии этой работы получается некий индекс, который и преобразуется с помощью `DataMap` в ключ.

Листинг 4 – Интерфейс `KeyGenerator`

```
interface KeyGenerator<K>:
    K nextGet()
    K nextInsert()
    K nextRemove()
    K nextPrefill()
```

Для упрощённой инициализации, для каждой сущности `KeyGenerator` реализован интерфейс `KeyGeneratorBuilder`, который генерирует для каждой сущности `ThreadLoop` свой `KeyGenerator`.

2.4. Сущность ThreadLoop

Сущность ThreadLoop отвечает за сбор статистики и взаимодействие со структурой данных, реализуя интерфейс представленный на листинге 5. Метод `prefill()` отвечает за предварительное заполнение структуры данных. Метод `run()` на протяжении всего тестирования выбирает следующий тип операции, берет из `KeyGenerator` соответствующий ключ и выполняет операцию.

Выводы по главе 2

Было спроектировано тестирующее окружение, где нагрузка поделена на четыре типа сущностей: `Distribution`, `DataMap`, `KeyGenerator`,

Листинг 5 – Интерфейс `ThreadLoop`

```
interface ThreadLoop:
    void run()
    void prefill()
```

`ThreadLoop`. Такой способ позволяет при добавлении новой нагрузки переиспользовать уже реализованные сущности. Также для добавления новой сущности пользователю нет необходимости изучать реализацию других сущностей и всего тестирующего окружения.

ГЛАВА 3. ПРИМЕР ДОБАВЛЕНИЯ НОВОЙ НАГРУЗКИ

Для примера добавления новой нагрузки мы реализуем скошенную нагрузку, описанную в статье [9]. Для начала опишем её.

Эта нагрузка определяется пятью параметрами $n - w - x - y - s$:

— n — размер рабочего набора ключей;
— w% — количество операций записи (`insert` и `remove`);
— x% операций чтения равновероятно совершаются над некоторым случайно выбранным подмножеством ключей размера y%, в то время как остальные операции чтения совершаются над оставшимися ключами;
— операции `insert` и `remove` равновероятно совершаются над некоторым случайно выбранным подмножеством ключей размера s%.

Рассмотрим эти параметры в терминах сущностей, описанных в Главе 2. Первый параметр, n, представляет собой рабочий диапазон ключей, который является общим параметром. Второй параметр, w, является вероятностью выбора операции записи, за что отвечает сущность `ThreadLoop` 2.4. Параметры x и y затрагивают сразу несколько сущностей: `Distribution` 2.1, `DataMap` 2.2 и `KeyGenerator` 2.3. `Distribution` с вероятностью x% будет генерировать случайную величину из промежутка $[0; y \cdot n)$, используя равномерное распределение, и с вероятностью $100 - x$% из промежутка $[y \cdot n; n)$, также используя равномерное рпспределение. Назовём это распределение `Skewed Uniform Distribution`. В качестве `DataMap` будет взят обычный перетасованный массив. Используя эти сущности, `KeyGenerator` будет генерировать ключ для операции чтения. Последний параметр, s, будет реализован схожим образом. Теперь подробнее остановимся на реализации всех этих сущностей.

3.1. Сущность Distribution

Для реализации этой нагрузки необходимы два распределения: равномерное (`Uniform Distribution`) для операций записи, и скошенное (`Skewed Uniform Distribution`) для операций чтения. Так как первое распределение довольно простое, то пропустим его и остановимся на втором.

Первым шагом будет добавление названия нашего нового распределения в перечисление распределений `DistributionType`. Пример представлен на листинге 6.

Листинг 6 – `DistributionType`

```
enum DistributionType:
    {...}, SKEWED_UNIFORM
```

Далее, нам нужно определить параметры нового распределения: это будут `HOT_SIZE` и `HOT_PROB`, отвечающие за размер подмножества относительно всего промежутка значений и вероятность его вызова соответственно. В нашей нагрузке этими параметрами являются y и x. Пример представлен на листинге 7.

Листинг 7 – `SkewedUniformParameters`

```
class SkewedUniformParameters:
    double HOT_SIZE = 0 // y%
    double HOT_PROB = 0 // x%
```

Также необходимо добавить синтаксический анализатор для параметров нового распределения в `DistributionBuilder` в метод `parseDistribution` 8.

Листинг 8 – Анализ аргументов в методе `parseDistribution`

```
boolean parseDistribution(ParseArgument args):
    switch (args.getCurrent()):
        case "-dist-skewed-uniform":
            param = new SkewedUniformParameters(
                args.getNext(), args.getNext())
            this.setDistributionType(SKEWED_UNIFORM)
                .setParameters(param)
            return true
        case {...}
    return false
```

После всего этого можно приступать к реализации самого распределения. Поскольку это распределение внутри подмножеств $[0, y \cdot n)$ и $[y \cdot n, n)$ выбирает случайную величину равномерно, то можно использовать уже реализованную сущность `Uniform Distribution`, определяя лишь из какого распределения будет доставаться следующая случайная величина с вероятностью $x\%$ и $100 - x\%$ соответственно. Реализация представлена на листинге 9.

Листинг 9 – Реализация `SkewedUniformDistribution`

```
class SkewedUniformDistribution implements Distribution:
    int hotLength
    double hotProb
    UniformDistribution hotDist
    UniformDistribution coldDist
    Random random

    int next():
        if (random.nextDouble() < hotProb):
            return hotDist.next()
        else:
            return hotLength + coldDist.next()
```

Напомним, что распределение возвращает лишь случайную величину, которую в дальнейшем `KeyGenerator` при помощи `DataMap` преобразует в ключ.

Для завершении реализации нового распределения осталось добавить конструктор в `DistributionBuilder` в метод `getDistribution()` по некоторому промежутку ключей. Реализация представлена на листинге 10.

Листинг 10 – Реализация конструктора `SkewedUniformDistribution`

```
Distribution getDistribution(int range):
    switch (this.distributionType) {
        case SKEWED_UNIFORM:
            int hotLength = parameters.HOT_SIZE * range;
            return new SkewedUniformDistribution(
                hotLength,
                parameters.HOT_PROB,
                new DistributionBuilder()
                    .getDistribution(hotLength),
                new DistributionBuilder()
                    .getDistribution(range - hotLength))
        case {...}
```

3.2. Сущность DataMap

Для преобразования случайной величины, генерируемой распределениями, в ключ, потребуется сущность `DataMap`. В данной нагрузке ключи распределены по множествам размера $x\%$ и $100 - x\%$ случайным образом. Поэтому при реализации интерфейса `DataMap` 3, определим массив всех ключей и перетасуем их случайным образом. Реализация представлена на листинге 11.

Листинг 11 – Реализация `Array DataMap`

```
class ArrayDataMap implements DataMap:
    int[] data

    ArrayDataMap(int range):
        for (i = 0; i < range; i++):
            data[i] = i + 1
        random.shuffle(data)

    int get(int index):
        return data[index]
```

3.3. Сущность KeyGenerator

После реализации распределения можно приступать к реализации генератора ключей. Для начала необходимо так же добавить название генератора в перечисление 12.

Листинг 12 – `KeyGeneratorType`

```
enum KeyGeneratorType:
    {...}, EXAMPLE_KEYGEN
```

Новый генератор ключей зависит от трёх параметров: x, y и s. Все они нужны для создания распределений для операций чтения и записи. Для получения параметров из командной строки, нужно расширить класс `Parameters` и его основные методы: `parseArg()`, `build()`. Первый метод, `parseArg()`, представляет из себя синтаксический анализатор текущего аргумента. Второй метод, `build()`, является сборщиком параметров после анализа, необходимый для параметров, которые генерируются или собираются на основе других параметров, так как в процессе синтаксического анализа они могут быть ещё неизвестны. Помимо обычных параметров, упомянутых выше, в этом классе будут храниться конструкторы распределений, благодаря которым сборка генератора ключей будет проще. Реализация класса параметров предоставлена на листинге 13.

Перейдем к реализации самого генератора. Он использует для генерации ключа два распределения. Так как они выдают лишь случайную величину, то также для их перевода в ключ потребуется два `DataMap`, так как нам не известно, как соотносятся между собой множества для операций чтения и записи. Реализация представлена на листинге 14.

Листинг 13 – Реализация `ExampleParameters`

```
class ExampleParameters extends Parameters:
    DistributionBuilder getDistBuilder =
        new DistributionBuilder(SKEWED_UNIFORM)
    DistributionBuilder updateDistBuilder =
        new DistributionBuilder(UNIFORM)
    double x, y, s

    void parseArg(ParseArgument args):
        switch (args.getCurrent()):
            case "-x" -> x = args.getNext()
            case "-y" -> y = args.getNext()
            case "-s" -> s = args.getNext()
            default -> super.parseArg(args)

    void build():
        super.build()
        getDistBuilder.setParameters(
            new SkewedUniformParameters(x, y))
```

Листинг 14 – Реализация `ExampleKeyGenerator`

```
class ExampleKeyGenerator implements KeyGenerator:
    DataMap getData
    DataMap updateData
    Distribution getDist
    Distribution updateDist

    int nextGet():
        return getData.get(getDist.next())

    int nextInsert():
        return updateData.get(updateDist.next())

    int nextRemove():
        return updateData.get(updateDist.next())
```

Следующим этап будет реализация интерфейса `KeyGeneratorBuilder`. Метод `generateKeyGenerators()` создаёт массив генераторов на все сущностей `ThreadLoop`. Реализация представлена на листинге 15.

В конце необходимо добавить новый генератор в метод `parseKeyGenerator()`, который по введённым данным определяет какой генератор ключей необходимо использовать во время тестирования. Пример добавления представлен на листинге 16.

Листинг 15 – Реализация `ExampleKeyGeneratorBuilder`

```
class ExampleKeyGeneratorBuilder extends KeyGeneratorBuilder:
    KeyGenerator[] generateKeyGenerators():
        KeyGenerator[] keygens =
            new KeyGenerator[parameters.numThreads]
        DataMap getData =
            new ArrayDataMap(parameters.range)
        DataMap updateData =
            new ArrayDataMap(parameters.range)

        for (i = 0; i < parameters.numThreads; i++):
            keygens[i] = new ExampleKeyGenerator(
                getData, updateData,
                parameters.getDistBuilder
                    .getDistribution(parameters.range),
                parameters.upadeteDistBuilder
                    .getDistribution(parameters.s *
                                     parameters.range)
            )

    return keygens
```

Листинг 16 – Добавление нового генератора ключей `Example` в метод `parseKeyGenerator()`

```
KeyGeneratorBuilder parseKeyGenerator(ParseArgument args)
    Parameters parameters
    KeyGeneratorBuilder keyGeneratorBuilder
    switch (args.getCurrent()):
        case "-example":
            parameters = new ExampleParameters()
            parameters.keygenType = KeyGeneratorType.
               EXAMPLE_KEYGEN

            keyGeneratorBuilder = new ExampleKeyGeneratorBuilder(
               parameters)
        case {...}
    args.next()
    return keyGeneratorBuilder
```

3.4. Сущность ThreadLoop

Добавление новой реализации `ThreadLoop` состоит из 4 этапов:

а) Реализация интерфейса `ThreapLoopParameters` при необходимости;

б) Реализация интерфейса `ThreadLoop`, то есть реализация методов `run()` и `prefill()`;

в) Добавить объявление нового `ThreadLoop` в метод `getThreadLoop()` в классе `ThreadLoopBuilder`;

г) Добавить новый `ThreadLoop` в метод `parseThreadLoop()` в классе `Parameters`.

Метод `run()` представляет из себя цикл со стоп-флагом, в ходе которого мы некоторым образом определяем следующий тип операции, при помощи генератора ключей генерируем ключ соответствующий операции, и исполняем её, подсчитывая при этом статистику об успешности операции. Метод `prefill()` принимает значение `prefillSize` с числом ключей, которые осталось добавить в структуру данных. При этом эта переменная является `AtomicInteger`, так как заполнение может происходить параллельно.

В нашем примере $w\%$ операций являются операциями записи (`insert` и `remove` равновероятны), а оставшиеся $100 - w\%$ — операциями чтения. Для этого в методе `run()` мы будем подбрасывать нечестную монетку, и в зависимости от результата выполнять ту или иную операцию. Пример реализации представлен на листинге 17.

Листинг 17 – Реализация `DefaultThreadLoop`

```
class ThreadMapLoop implements ThreadLoop:
    void run():
        while (!stop):
            double coin = rand.nextDouble()
            if (coin < w/2):
                K key = keygen.nextInsert()
                dataStructure.putIfAbsent(key, key)
            else if (coin < w)
                K key = keygen.nextRemove()
                dataStructure.remove(key)
            else
                K key = keygen.nextGet()
                dataStructure.get(key)

    void prefill(AtomicInteger prefillSize):
        while (prefillSize.get() > 0):
            int curSize = prefillSize.decrementAndGet()
            K v = keygen.nextPrefill()
            if (curSize < 0
                    || dataStructure.putIfAbsent(v, v) != null):
                prefillSize.incrementAndGet()
```

Выводы по главе 3

Таким образом было продемонстрировано: как можно добавить новую нагрузку в тестирующее окружение, а также показана простота этого действия. В то время, как в существующих тестирующих окружениях, пользователю пришлось бы продумывать всю логику взаимодействия структуры данных с тестирующим окружением с нуля.

ГЛАВА 4. РЕАЛИЗОВАННЫЕ КОМПОНЕНТЫ

В этой главе мы продемонстрируем основные реализации сущностей нагрузки, которые уже доступны на двух языках программирования: `Java` и `C++`.

4.1. Сущности Distributions

В этой секции представлены реализованные сущности `Distributions` 2.1.

4.1.1. Uniform

В `Uniform Distribution` случайная величина генерируется равновероятно на заранее заданном промежутке. Из-за своей простоты, реализует так же интерфейс `MutableDistribution`.

4.1.2. Zipfian

`Zipfian Distribution` основан на законе Ципфа [7] с дополнительным параметром α. Случайная величина X выбирается с вероятностью $\frac{1}{X^{\alpha}\zeta(\alpha, range)}$, где $\zeta(\alpha, N) = \sum_{k=1}^{N} k^{-\alpha}$.

4.1.3. Skewed Uniform

`Skewed Uniform Distribution` описан в секции 3.1.

4.2. Сущности DataMaps

В этой секции представлены реализованные сущности `DataMaps` 2.2.

4.2.1. Id

`Id DataMap` никак не меняет порядок ключей, то есть, принимая значение $index$, возвращает $index + 1$. Увеличение индекса на один необходим из-за того, что индексы и случайные величины в генераторе ключей принимают значения $[0; range)$, в то время как ключи $[1, range]$.

4.2.2. Array

`Array DataMap` создаёт массив, заполненный значениями из всего диапазона ключей, перетасовывает из случайным образом и, при выполнении метода `get(int index)` выдаёт соответствующий индексу ключ из массива. Преимуществом такой реализации является случайность, из-за чего новых запуск тестирования не будет идентичным предыдущему. Недостатком является необходимость дополнительной $O(n)$ памяти.

4.2.3. Hash

`Hash DataMap` для преобразования из индекса в ключ использует хеш-функцию. На фоне `Array DataMap` большим преимуществом является отсутствие необходимости в дополнительной памяти, но преобразование не является случайным и возможны коллизии.

4.3. Сущности KeyGenerators

В этой секции представлены реализованные сущности `KeyGenerators` 2.3.

4.3.1. Default

`Default KeyGenerator` является стандартным генератором ключей, который использует одно распределение на все типы операций. Используя это распределению, он генерирует случайную величину и, при помощи `DataMap`, преобразует в ключ.

4.3.2. Skewed Sets

`Skewed Sets KeyGenerator` является улучшением генератора, описанного в секции 3.3. В версии из примера выбор подмножества (далее $HotRead$), на котором операции чтения происходили чаще, и множества (далее $HotWrite$), на котором происходили операции записи, никак не зависят друг от друга, из-за чего у нас могут быть три ситуации:

а) $HotRead \cap HotWrite = \varnothing$;
б) $HotRead \cap HotWrite = Hot : Hot \neq HotRead \wedge Hot \neq HotWrite$;
в) $HotRead \subseteq HotWrite \vee HotRead \supseteq HotWrite$.

В зависимости от ситуации могут получиться разные итоговые распределения ключей. Для наглядности, рассмотрим эти ситуации на самоподстраивающихся структурах данных с логическим удалением, например `SplayList` [9].

В первом случае множество ключей разделится на три подмножества: $HotRead$, $HotWrite$ и оставшееся. Таким образом, если мы зададим частые операции записи, то первые два подмножества будут конкурировать между собой во время настройки структуры данных.

Во втором случае, где множества $HotRead$ и $HotWrite$ пересекаются, конкуренции будет меньше. Структура данных настроится так, что множество

Hot будет в самом быстром доступе, далее за оставшееся место будут конкурировать множества $HotRead \setminus Hot$ и $HotWrite \setminus Hot$, и в самом медленном доступе будет оставшееся множество ключей. Благодаря этому, структуре данных будет проще справляться с нагрузкой, и она будет быстрее отвечать на запросы.

В третьем случае, где одно множество является подмножеством другого вовсе не будет конкуренции между множествами за нахождение в быстром доступе. Из-за чего структура данных будет справляться с нагрузкой в разы лучше, по сравнению с предыдущими случаями.

Из-за того, что выбор ситуации никак не контролируется, а ситуация напрямую сказывается на скорость работы структуры данных, то по результатам теста можно сделать неверные выводы. Для решения этой проблемы в `Skewed Sets KeyGenerator` добавлена возможность регулировать степень пересечения множеств $HotRead$ и $HotWrite$. Для этого генератор ключей использует два распределения `Skewed Uniform Distribution`: для операции чтения и записи по отдельности, используя единую сущность `DataMap`. И зависит он от следующих параметров:

— $rp\%$ операций чтения выполняются на случайно выбранном подмножестве ключей размера $range \cdot rs$ (то есть $HotRead$). Оставшиеся операции чтения выполняются на оставшемся множестве ключей;
— $wp\%$ операций записи выполняются на случайно выбранном подмножестве ключей размера $range \cdot ws$ (то есть $HotWrite$). Оставшиеся операции записи выполняются на оставшемся множестве ключей;
— $inter\%$ ключей находятся, как и в множестве $HotRead$, так и в множестве $HotWrite$.

4.3.3. Temporary Skewed Sets

`Temporary Skewed Sets KeyGenerator` основан на наблюдении, что в зависимости от времени, могут запрашиваться различные данные. Например, **утром** люди хотят узнать прогноз погоды на день, **днём** узнать какие-то сведения по работе, а **вечером** посмотреть новости прошедшего дня. Из примера видно, что в зависимости от временного промежутка, одно подмножество данных запрашивается чаще, на фоне остальных, и для имитации этого поведения, в `Temporary Skewed Sets KeyGenerator` существует два типа состояний:

— k-ое *возбуждённое* состояние — ключи генерируется при помощи k-ого `Skewed Uniform` распределения, где для чаще генерируемых случайных величин в сущности `DataMap` выделено k-ое подмножество ключей;
— состояние *спокойствия* — ключи генерируются при помощи `Uniform` распределения. Это состояние используется для плавного перехода между *возбужденными* состояниями.

Генератор принимает следующие параметры:
— $state\text{-}count$ — число *возбужденных* состояний;
— ht — продолжительность *возбуждённого* состояния по умолчанию. Используется, если не было явно указано продолжительность k-ого *возбуждённого* состояния;
— rt — продолжительность состояния *спокойствия* по умолчанию;
— ht_i — продолжительность i-ого *возбуждённого* состояния;
— rt_i — продолжительность состояния *спокойствия* после i-ого *возбуждённого* состояния;
— во время i-ого *возбуждённого* состояния p_i% операций выполняются над множеством ключей S_i размера $range \cdot s_i$, и $100 - p_i$% операций выполняются над оставшимся множеством ключей. $S_i \cap S_j = \varnothing : i \neq j$.

Продолжительность состояний указывается в числе проведенных операциях.

Важное замечание, для того чтобы генератор был бесконечным, *возбуждённые* состояния выбираются по циклу, то есть после последнего *возбуждённого* состояния и соответствующего *спокойного* состояния, нагрузка возьмёт снова первое *возбуждённое* состояние.

4.3.4. Creakers and Wave

`Creakers and Wave KeyGenerator` основан на наблюдении, что недавно добавленные данные могут чаще запрашиваться, но со временем эти данные устаревают и запрашиваются реже. Например, медиапродукт на видеоплатформе. При выходе нового видео, на фоне остальных, оно будет чаще просматриваемым из-за его новизны и актуальности. Но со временем интерес к этому видео остывает, и его начинают просматривать всё реже, в то время как выходят новые видео, которые просматриваются с большим ажиотажем.

Для имитации такого поведения, у генератора есть сущность *волна* (`Wave`). *Волна* это некоторое подмножество ключей в виде списка, у которого

есть начало (`head`) и конец (`tail`). При генерации нового ключа генератор придерживается следующих правил:

- для операции удаления (`remove`) *волна* возвращает текущий конец списка и перемещает конец на ключ, стоящий перед ним;
- для операции вставки (`insert`) возвращается новый ключ, которого нет в *волне*, и он объявляется новым началом списка;
- для операции чтения (`get`) возвращается ключ, который находится внутри *волны*, при помощи некоторого заданного распределения.

По умолчанию, *волна* использует `Zipfian` распределение, где чем ближе ключ находится к началу, тем с большей вероятностью он будет выбран для операции чтения. Так как диапазон ключей ограничен, то для того, чтобы генератор был бесконечным, *волна* перемещается по циклу по всему набору ключей, тем самым переиспользуя старые ключи. Поэтому, если в структуре данных используется логическое удаление, пользователю стоит подбирать параметры таким образом, чтобы, когда *волна* начинала новый цикл, старые ключи, которые будут переиспользоваться, уже были удалены физически.

Помимо сущности *волны*, в генераторе присутствует сущность *старички* (`Creakers`). Старички состоят из некоторого подмножества ключей, которые будут запрашиваться на протяжении всего тестирования, но редко. Благодаря этой сущности можно проверить на сколько хорошо само-подстраивающихся структуры данных подстраиваются с данными, которые очень часто запрашиваются на данный момент (новые ключи в *волне*), на фоне данных, которые запрашиваются на постоянной основе, но довольно редко (ключи из *старичков*). Так, например, в структурах данных, которые для само-подстраивания используют монотонный счётчик количества запросов, такие как `SplayList` [9], со временем ключи из сущности *старички* будут в быстром доступе, в то время как ключи из сущности волны будут находится в медленном доступе. Это связано с тем, что со временем счётчик *старичков* будет только возрастать и достигать больших значений, из-за чего структуры данных будут считать их самыми востребованными, в то время как ключи из *волны* будут удаляться из структуры раньше, чем их счётчики достигнут больших значений, хотя новые ключи из *волны* запрашиваются в разы чаще, чем *старички*.

`Creakers and Wave KeyGenerator` имеет следующие параметры:

— cs% из всего набора ключей будут состоят в сущности *старички*;
— cp% операций будут вычисляться над сущностью *старички*, и $100 - cp$% будут вычисляться над сущностью *волна*;
— перед тестированием, сущность *волна* будет заполнена ws% из всего набора ключей, исключая ключи, содержащиеся в сущности *старички*;
— c-age — число операций чтения, которые будут совершены над сущностью *старички*, перед тестированием;
— c-$distribution$ — распределение ключей в сущности *старички* (по умолчанию используется `Uniform Distribution`);
— w-$mutable$-$distribution$ — распределение ключей в сущности *волна*, при этом распределение должно реализовывать интерфейс `MutableDistribution` (по умолчанию используется `Zipfian Distribution` с $\alpha = 1$).

Этот генератор ключей идейно схож с нагрузкой `Read latest` из `YCSB`, но в отличии от него, `Creakers and Wave` является бесконечным. Нагрузку из `YCSB` тоже можно сделать бесконечной, но у неё нет возможности генерировать новый ключ с временной сложностью $O(1)$ во время выполнения, что будет сильно влиять на нагрузку. Также в реализованный генератор ключей была добавлена сущность *старички*, которой нет в нагрузке `Read latest`. Потому нагрузка из `YCSB` является менее удобной.

4.3.5. Leafs Handshake

`Leafs Handshake Key Generator` генерирует следующий ключ (далее $key_{new\ insert}$) для операции вставки на основе ключа (далее $key_{last\ removed}$), который был последним сгенерированным ключом для операции удаления. При этом, чем ближе $key_{new\ insert}$ находится к $key_{last\ removed}$, тем больше вероятность того, что именно этот ключ будет использован в операции. Генератор принимает следующие параметры:
— get-$distribution$ — распределение, используемое для генерации ключа для операции чтения (по умолчанию используется `Uniform Distribution`);
— $remove$-$distribution$ — распределение, используемое для генерации ключа для операции удаления (по умолчанию используется `Uniform Distribution`);

— $insert\text{-}mutable\text{-}distribution$ — распределение, реализующее интерфейс `MutableDistribution`, используемое для генерации ключа для операции вставки (по умолчанию используется `Zipfian Distribution` с $\alpha = 1$).

Значение $key_{last\ removed}$ известно всем потокам. При генерации ключа для вставки, случайным образом определяется промежуток, из которого будет выбран $key_{new\ insert}$: $[1; key_{last\ removed} - 1]$ или $[key_{last\ removed} + 1; range]$. Далее, при помощи $insert\text{-}mutable\text{-}distribution$ определяется ключ из выбранного промежутка.

4.4. Сущности ThreadLoops

В этой секции представлены реализованные сущности `ThreadLoops` 2.4.

4.4.1. Default

`Default ThreadLoop` выбирает следующую операцию с некоторой зафиксированной постоянной вероятностью. Подробная реализация описана в секции 3.4. Он принимает следующие параметры:

— ui% операций будут операциями вставки (`insert`);
— ue% операций будут операциями удаления (`remove`);
— Оставшиеся $100 - ui\% - ue\%$ операций будут операциями чтения (`get`).

4.4.2. Temporary Operations

`Temporary Operations ThreadLoop` идейно схож с `Temporary Skewed Sets KeyGenerator`, описанный в секции 4.3.3, за тем исключением, что здесь в зависимости от временного промежутка меняются вероятности вызова той или иной операции. Он принимает следующие параметры:

— $temp\text{-}oper\text{-}count$ — количество временных интервалов;
— ot_i — продолжительность i-ого временного интервала (продолжительность указывается в числе проведенных операциях);
— ui_i% операций будут операциями вставки (`insert`) в течение i-ого временного интервала;
— ue_i% операций будут операциями удаления (`remove`) в течение i-ого временного интервала;
— Оставшиеся $100 - ui\% - ue\%$ операций будут операциями чтения (`get`).

Временные интервалы повторяются циклически, тем самым делая `ThreadLoop` бесконечным.

Выводы по главе 4

Таким образом было реализованы компоненты, при комбинации которых можно воссоздавать различные бесконечные нагрузки, большинство из которых являются скошенными.

ГЛАВА 5. ТЕСТИРОВАНИЕ БИНАРНЫХ ДЕРЕВЬЕВ ПОИСКА

Для тестирования были взяты три широко известные реализации бинарного дерева поиска, написанные на языке `Java`. Основная цель тестов — показать, что при использовании различных нагрузок из нашего тестирующего окружения можно добиться различных относительных производительностей, то есть нет явного победителя. Так мы покажем, что наше тестирующее окружение уже имеет достаточно хорошую репрезентативность нагрузок.

5.1. Реализации

Возьмем три самые эффективные реализации бинарного дерева поиска, написанные на `Java`:

— `BCCO BST` [2].
— `Contention-Friendly (CF) BST` [5].
— `Concurrency-Optimal (CO) BST` [1].

Каждая из реализаций является частично-внешним деревом поиска, но они по-разному работают с физическим удалением и балансировкой дерева. В `BCCO-BST` рабочий поток всегда при возможности удаляет узлы физически и при необходимости балансирует поддеревья. В `CF-BST` рабочий поток делает только логические удаления и не занимается балансировкой. Однако существует специальный поток-демон, который как раз и занимается физическим удалением и балансировкой на фоне. В `CO-BST` рабочий поток, как и в `BCCO-BST`, пытается удалить узлы физически, но не занимается балансировкой вовсе.

Анализируя эксперименты из предыдущих статей с использованием нагрузок из `Synchrobench`, например, [1], можно сделать вывод, что `CO-BST` является самым эффективным, далее идёт `CF-BST`, и самым наименее производительным является `BCCO-BST`. Очевидно, что такое поведение не должно распространяться на все возможные нагрузки. Проведенное тестирование показывает, что это действительно так: в зависимости от нагрузки, можно увидеть разную относительную производительность. Перед этим нужно внести изменение в реализацию `CF-BST`, которая иногда значительно повышает производительность.

5.1.1. Исправление Concurrency-Friendly BST

В ходе экспериментов, было замечено, что реализация `CF-BST` в `Synchrobench` отличается от версии, представленной в статье [5]. Различие

заключалось в работе демон-потока, а точнее в порядке действий. Демон-поток в бесконечном цикле, начиная с корня, рекурсивно проходится по вершинам, пытается логически-удаленные вершины удалить физически и сбалансировать поддерево. В версии, представленной в статье, демон-поток сначала спускается к детям вершины, а после пытается удалить её физически и сбалансировать поддерево. Тем самым демон-поток пытается удалить вершины снизу вверх. В реализации в `Synchrobench` демон-поток сначала пытается удалить узел, и лишь потом спускается к детям и занимается балансировкой.

Таким образом, если в дереве N элементов и все они логически удалены, то в нашем алгоритме демону требуется один рекурсивной проход. В то время как демон-поток из версии `Synchrobench` требует $\log N$ проходов, ведущих к $\sum_{i=1}^{\log N}(2^i - 1) = 2N - \log N - 2$ операций. Можно сделать вывод, что алгоритм демон-потока в `Synchrobench` почти в два раза медленнее, что может сказаться на общей пропускной способности. В экспериментах исправленная версия будет называться `Fixed Concurrency-Friendly BST`.

5.2. Результаты экспериментов

Нагрузки были запущены на двухпроцессорном Intel Xeon Gold 6240R с 24 ядрами каждый, что даёт в сумме 48 ядер. Результаты будут предоставлены начиная с 16 рабочих потоков до 48 возможных с шагом 4. На каждом графике представлены четыре линии: синяя (`CO-BST`), оранжевая (`Fixed CF-BST`), зеленая (`CF-BST`) и красная (`BCCO-BST`). Ось абсцисс представляет число используемых потоков, а ось ординат представляет пропускную способность, то есть число операций, выполняемых за одну секунду. Каждый точка на графиках является результатов эксперимента, проведенного 10 раз в течение 10 секунд.

5.2.1. Uniform и Zipfian

Нагрузки `Uniform` и `Zipfian` являются наиболее распространёнными нагрузками. Для её сборки использовались следующие реализации сущностей:

- `Uniform Distribution` 4.1.1 и `Zipfian Distribution` 4.1.2, для нагрузок `Uniform` и `Zipfian` соответственно;
- `Default KeyGenerator` 4.3.1;

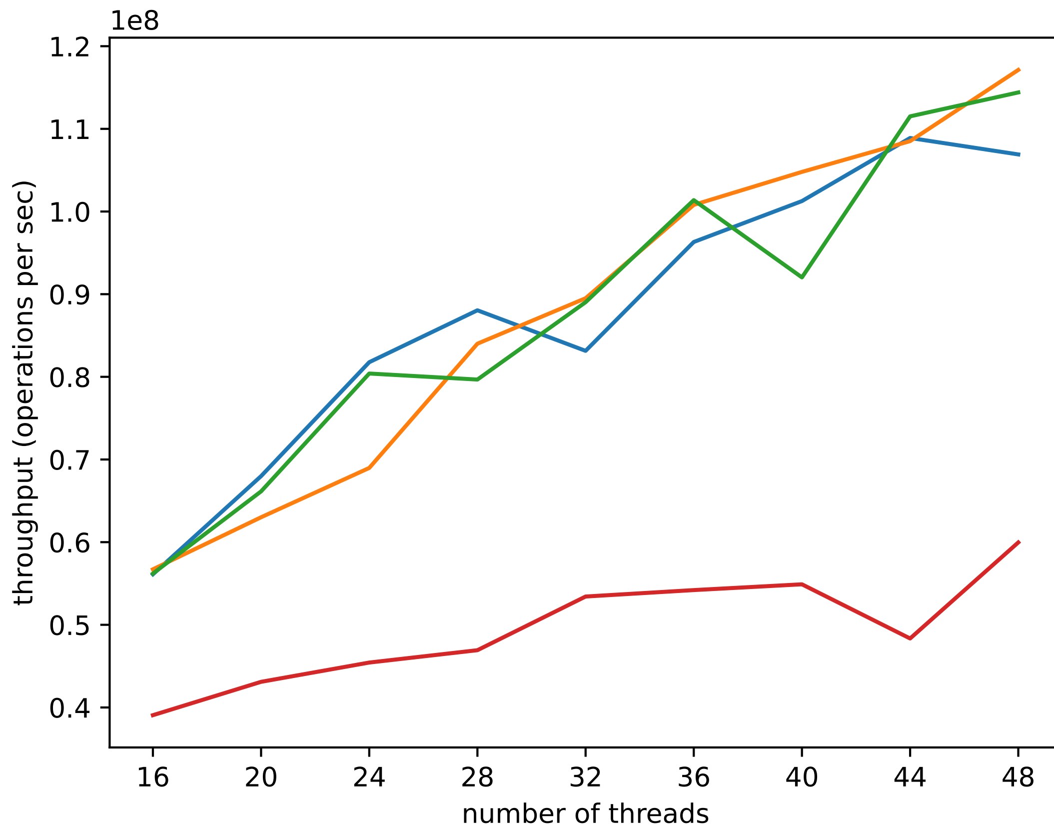

Рисунок 3 – Нагрузка Uniform с $range = 10^4$, $ui = ue = 10\%$

- — Для нагрузки Uniform использовался Id DataMap 4.2.1, а для Zipfian Array DataMap 4.2.2, чтобы распределение самих ключей было случайным;
- — Default ThreadLoop 4.4.1.

Запуск нагрузки Uniform производилось с параметрами: 10^4 — рабочий диапазон ключей (range); $ui = ue = 10\%$. Запуск нагрузки Zipfian производилось с параметрами: 10^5 — рабочий диапазон ключей; $ui = ue = 2.5\%$. Результаты представлены на рисунках 3 и 4.

В случае с нагрузкой Uniform результат обоснован тем, что нагрузка выбирает ключи равномерно, потому ключи будут равномерно распределены по всему дереву. Таким образом дерево является практически сбалансированным изначально, и дополнительная балансировка в процессе нагрузки не является необходимой. Даже наоборот, в случае с BCCO-BST из-за мелких колебаний высот поддеревьев, будет вызываться балансировка, которая берет блокировку на поддерево, что отражается на общей производительности.

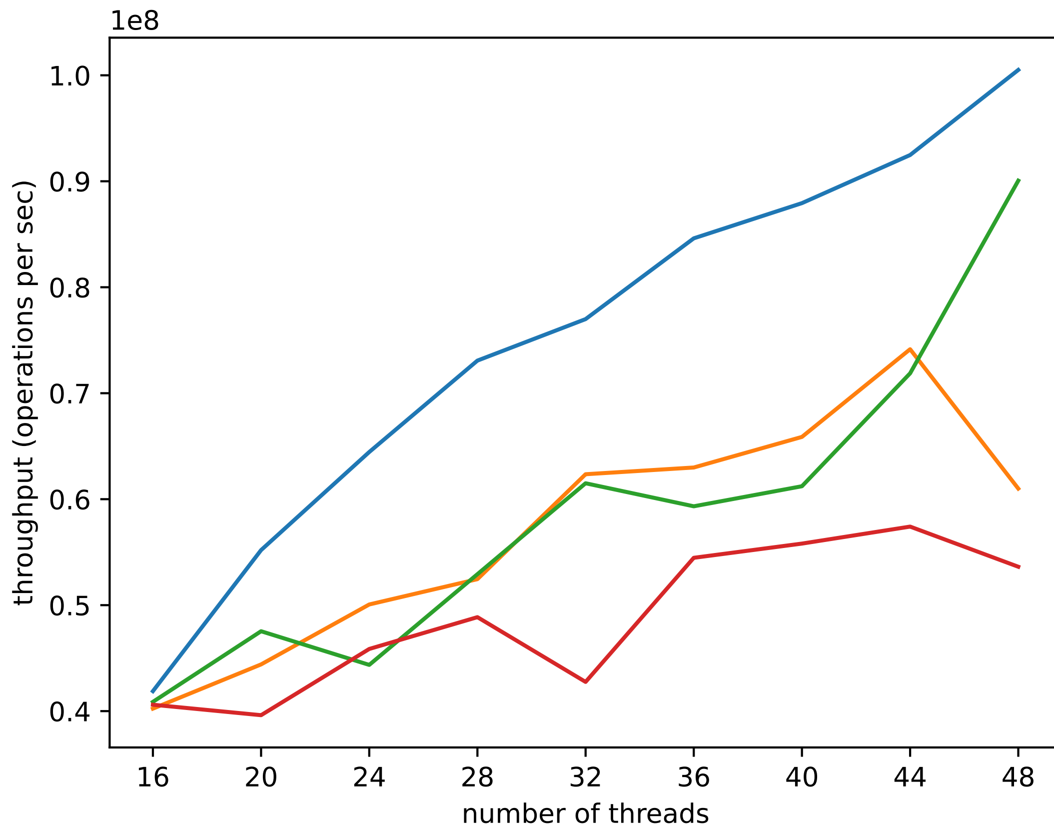

Рисунок 4 – Нагрузка Zipfian с $\alpha = 1$, $range = 10^5$, $ui = ue = 2.5\%$

5.2.2. Infinite Leafs Handshake

Поскольку BCCO-BST всегда пытается удалять вершины физически и балансировать поддерево, можно ожидать, что он будет всегда слишком сильно отставать с точки зрения пропускной способности. Однако это не значит, что он будет работать плохо на любых нагрузках. Во время изучения работы бинарных деревьев поиска была разработана нагрузка Infinite Leafs Handshake. Для её сборки использовались следующие реализации сущностей:

- Zipfian Distribution 4.1.2 для операций вставки и Uniform Distribution 4.1.1 для операций чтения и удаления;
- Leafs Handshake KeyGenerator 4.3.5;
- Id DataMap 4.2.1;
- Temporary Operations ThreadLoop 4.4.2.

У ThreadLoop есть три временных интервала:

- стадия заполнения — операций вставок больше, чем операций удаления;
- стадия чтения — производятся только операции чтения;
- стадия очистки — операций удаления больше, чем операций вставок.

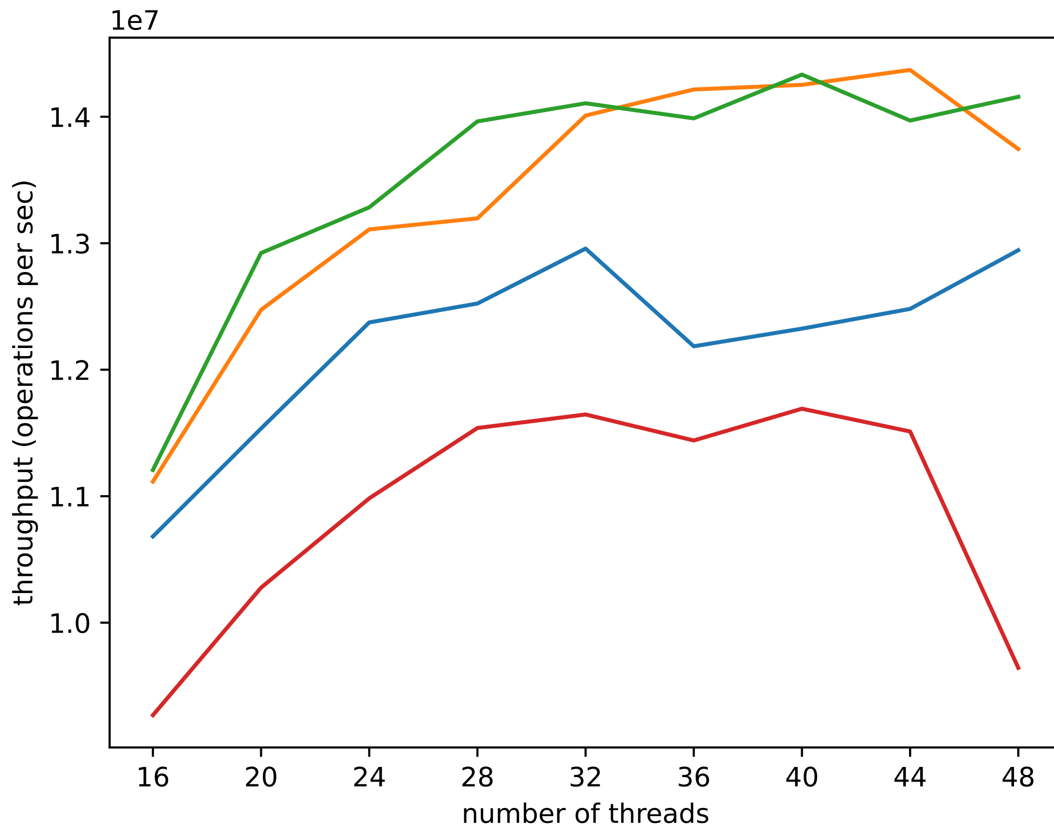

Рисунок 5 – Нагрузка `Infinite Leafs Handshake` с $range = 10^5$.

Во время стадии заполнения на один удаленный узел вставляются два новых соседа. Во время стадии чтения потоки просто выполняют операции чтения, тем самым фиксируя результат предыдущей стадии. Во время стадии очистки потоки случайным образом удаляют узлы из дерева, чтобы восстановить его первоначальный размер. Таким образом, запуская эти стадии поочередно в бесконечном цикле, нагрузка является бесконечной.

Первый эксперимент, результат которого представлен на рисунке 5), запускался с параметрами: $range = 10^5$; $temp\text{-}oper\text{-}count = 3$; $ot_0 = ot_2 = 10000$; $ot_1 = 5000$; $ui_0 = ue_2 = 60\%$; $ui_2 = ue_0 = 40\%$; $ui_1 = ue_1 = 0\%$; для генерации ключей для операции чтения и удаления используется `Uniform Distribution`; для генерации ключей для операции вставки используется `Zipfian Distribution` с $\alpha = 2$.

Как можно заметить, в этом эксперименте `CF-BST` лучше справляется с нагрузкой, в отличии `CO-BST`. Это связано с тем, что нагрузка не является равномерной, из-за чего баланс дерева нарушается и это начинает сказываться на производительности. Но нарушение баланса не на столько значительна, чтобы `BCCO-BST`, с его подходом честной балансировки, вырывался вперед.

Во втором эксперименте нагрузка состояла лишь из двух стадий: заполнения и очистки. Результат представлен на рисунке 6). Запуск производился с параметрами: $range = 10^7$; $temp\text{-}oper\text{-}count = 2$; $ot_0 = ot_1 = 20000$; $ui_0 = ue_1 = 90\%$; $ui_1 = ue_0 = 10\%$; для генерации ключей для операции чтения и удаления используется `Uniform Distribution`; для генерации ключей для операции вставки используется `Zipfian Distribution` с $\alpha = 0.99$.

В отличие от предыдущего эксперимента, `BCCO-BST` справляется с нагрузкой лучше всех, так как скошенность соотношения операций вставки и записи довольно велика, из-за чего дерево быстро разрастается. А так как из-за специфики нагрузки, разрастание происходит в каком-то направлении (неравномерно), потому наличие балансировки становится довольно значительным, чтобы сильно влиять на производительность. Однако, можно заметить, что `CF-BST` начинает сильно отставать, хотя, в отличии от `CO-BST`, у него присутствует какая-никакая балансировка через демон-потока. Связано это с тем, что нагрузка заточена на то, чтобы на стадии заполнения в окрестности удаленной вершины, вставились несколько новых. Из-за чего может произойти следующая ситуация:

- поток логически удаляет вершину k, которая является листом или имеет всего одного ребенка;
- демон-поток занят физическим удалением других вершин, или балансировкой, или же просто рекурсивно обходит дерево где-то вдалеке от вершины k;
- несколько потоков добавляет новые вершины, которые являются соседями вершины k, и эти вершины становятся детьми вершины k.

После этой ситуации, демон-поток, дойдя до вершины k, не может удалить её физически, так как вершину можно удалить физически при условии отсутствия одного или более потомков. Из-за скошенности соотношения операций вставки и записи, появляется большая вероятность того, что вершина k потеряет возможность удалиться физически, а из-за большого размера дерева, демон-поток сталкивается с больше нагрузкой, из-за чего не будет поспевать за физически удалять вершины. Второй эксперимент является совокупностью этих двух проблем, из-за чего дерево в `CF-BST` быстро разрастается, что сильно сказывается на производительности.

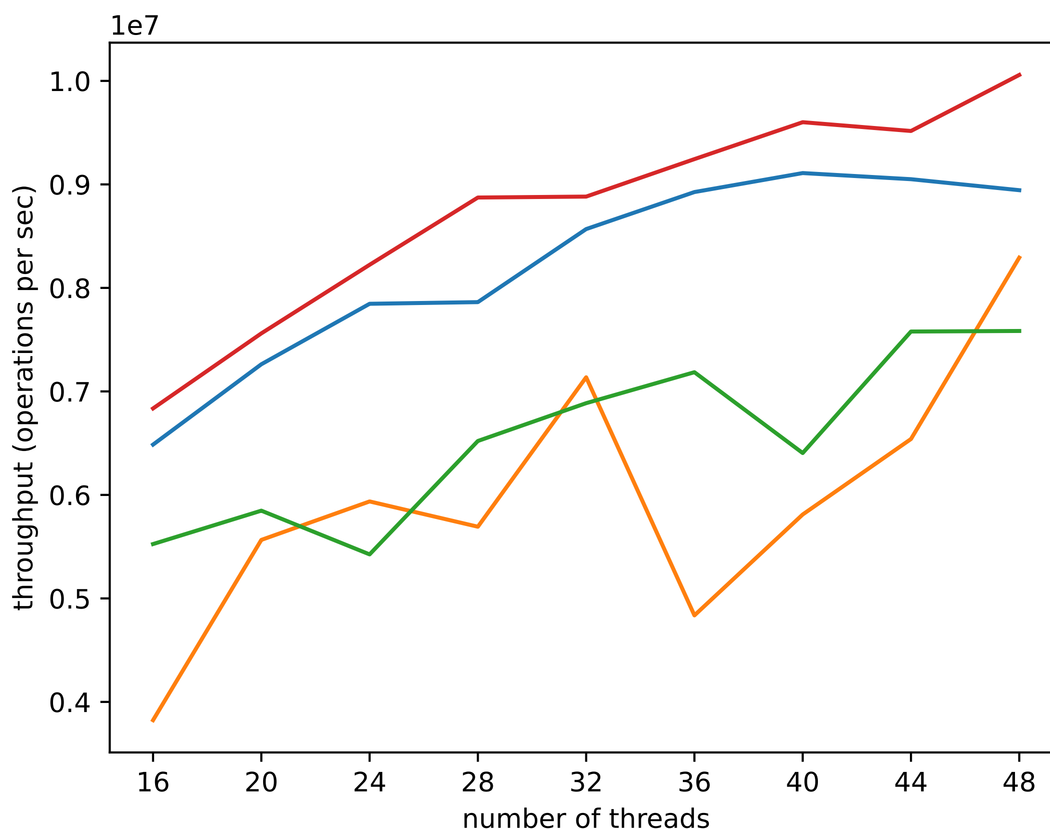

Рисунок 6 – Нагрузка `Infinite Leafs Handshake` с $range = 10^7$.

Третий эксперимент, результат которого представлен на рисунке 7), запускался с параметрами: $range = 10^8$; $temp\text{-}oper\text{-}count = 3$; $ot_0 = ot_1 = ot_2 = 100000$; $ui_0 = ue_2 = 80\%$; $ui_2 = ue_0 = 20\%$; $ui_1 = ue_1 = 0\%$; для генерации ключей для операции чтения и удаления используется `Uniform Distribution`; для генерации ключей для операции вставки используется `Zipfian Distribution` с $\alpha = 0.99$.

Такой результат связан с теми же причинами, которые были описаны выше. Скошенность соотношений операции и размер дерева плохо сказывается на производительности `CF-BST`, но скошенность не достаточна, чтобы `BCCO-BST` опережал `CO-BST`.

5.2.3. Non-shuffle Wave

Для сборки нагрузки `Non-shuffle Wave` использовались следующие реализации сущностей:

- `Zipfian Distribution` 4.1.2;
- `Creakers And Wave Key Generator` 4.3.4 без сущности *старички*;
- `Id DataMap` 4.2.1;

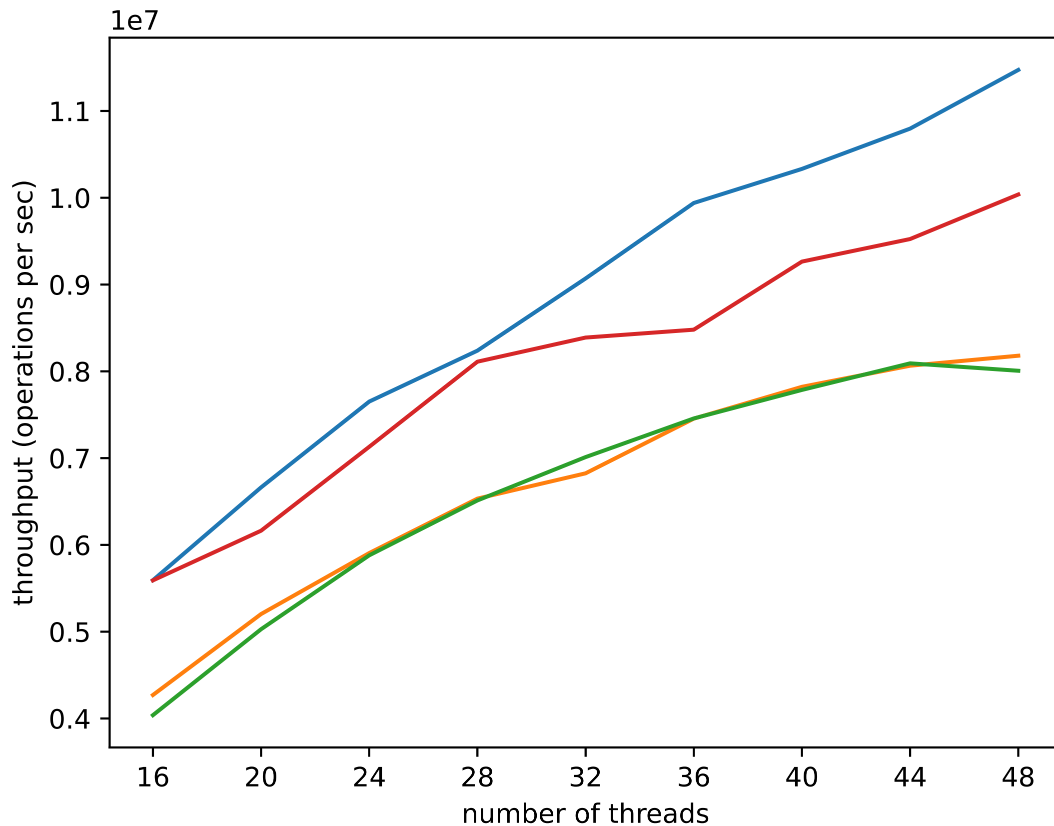

Рисунок 7 – Нагрузка `Infinite Leafs Handshake` с $range = 10^8$.

— `Default ThreadLoop` 4.4.1.

Таким образам, каждый новый вставленный ключ будет с краю дерева, нарушая баланс. Это приводит к проблемам с паропроизводительностью из-за плохой сбалансированности дерева.

Первый эксперимент, результат которого представлен на рисунке 8), запускался с параметрами: $range = 10^6$; $ui = ue = 2.5\%$; $ws = 20\%$; $w\text{-}distribution$ — `Zipfian Distribution` с $\alpha = 1$; $cp = 0\%$.

В этом эксперименте `BCCO-BST` на порядок превосходит другие бинарные деревья поиска, и связано это напрямую с тем, что нагрузка сильно нарушает баланс. `CO-BST`, из-за отсутствия балансировки, вовсе превращается в бамбук, где самый нижний элемент является самым новым. Из-за специфики генератора `Creakers And Wave`, новые элементы *волны* запрашиваются чаще остальных, что ещё сильнее усугубляет ситуацию, и `CO-BST` справляется с нагрузкой хуже всех.

Первый эксперимент, результат которого представлен на рисунке 8), запускался с параметрами: $range = 5 \cdot 10^3$; $ui = ue = 20\%$; $ws = 10\%$; $w\text{-}distribution$ — `Zipfian Distribution` с $\alpha = 1$; $cp = 0\%$.

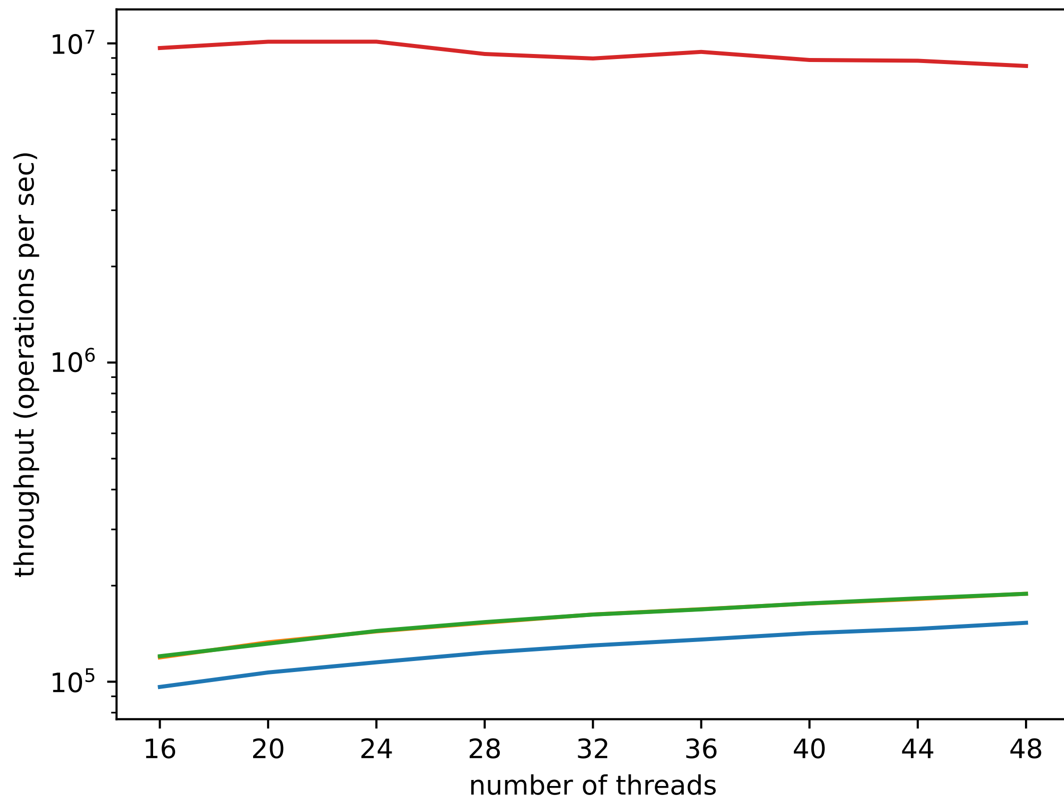

Рисунок 8 – Нагрузка `Non-shuffle Wave` с $range = 10^6$

В отличие от предыдущего эксперимента, размер дерева довольно мал, потому демон-поток в `CF-BST` успевает сбалансировать дерево, из-за чего он начинает опережать `BCCO-BST` по эффективности.

Выводы по главе 5

В результате были получены все возможные перестановки относительных производительностей трёх структур данных: `BCCO-CF-CO` на рисунке 9, `BCCO-CO-CF` на рисунке 6, `CF-BCCO-CO` на рисунке 8, `CF-CO-BCCO` на рисунке 5, `CO-BCCO-CF` на рисунке 7 и `CO-CF-BCCO` на рисунках 3 и 4.

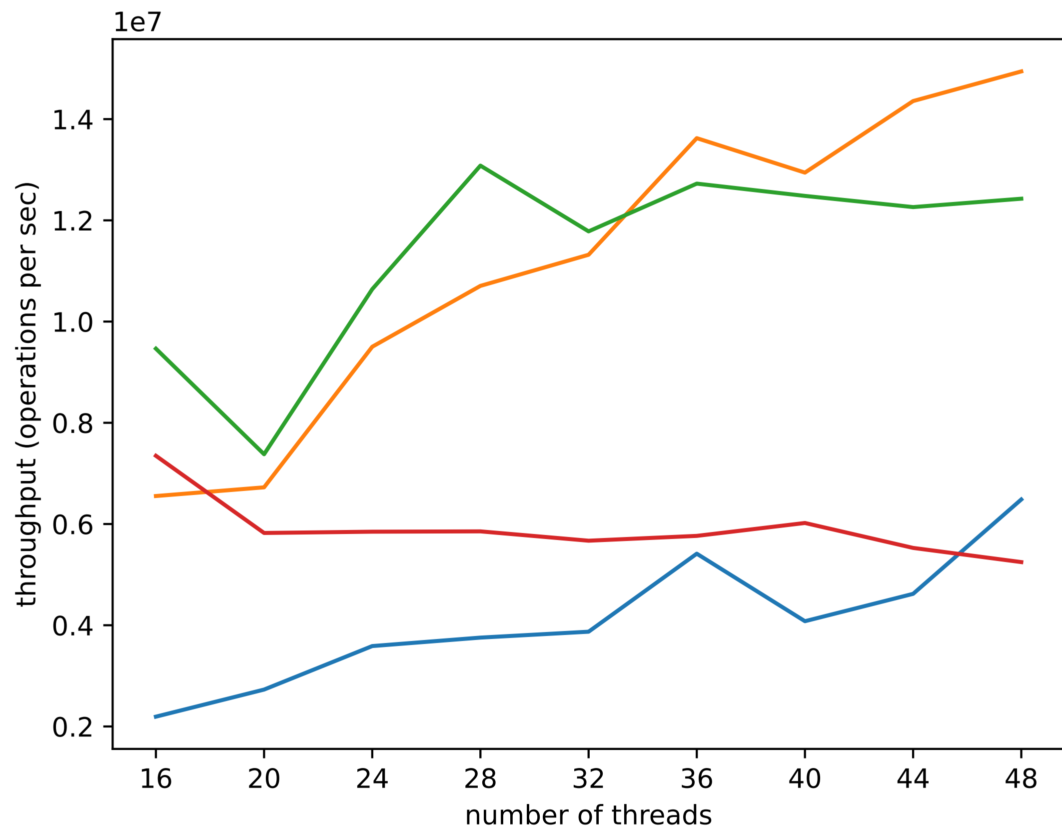

Рисунок 9 – Нагрузка `Non-shuffle Wave` с $range = 5 \cdot 10^3$

ЗАКЛЮЧЕНИЕ

В ходе работы было спроектировано и реализовано новое тестирующее окружение скошенных нагрузок для оценки эффективности конкурентных индексов, которое поддерживает легкое добавление новых нагрузок. Это обеспечивается за счёт разделения нагрузки на 4 типа сущностей, которые можно переиспользовать при добавлении новой нагрузки. Реализованные нагрузки удовлетворяют нашим требованиям: они являются бесконечными и скошенными. А также показали, что наше тестирующее окружение достаточно обширно, что позволяет показать сильные и слабые стороны тех или иных структур данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 A concurrency-optimal binary search tree / V. Aksenov [и др.] // Euro-Par 2017: Parallel Processing: 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28–September 1, 2017, Proceedings 23. — Springer. 2017. — С. 580–593.

2 A practical concurrent binary search tree / N. G. Bronson [и др.] // ACM Sigplan Notices. — 2010. — Т. 45, № 5. — С. 257–268.

3 Benchmarking cloud serving systems with YCSB / B. F. Cooper [и др.] // Proceedings of the 1st ACM symposium on Cloud computing. — 2010. — С. 143–154.

4 *Brown T.*, *Prokopec A.*, *Alistarh D.* Non-blocking interpolation search trees with doubly-logarithmic running time // Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2020. — С. 276–291.

5 *Crain T.*, *Gramoli V.*, *Raynal M.* A contention-friendly binary search tree // Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19. — Springer. 2013. — С. 229–240.

6 *Gramoli V.* More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms // Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2015. — С. 1–10.

7 *Powers D. M.* Applications and explanations of Zipf's law // New methods in language processing and computational natural language learning. — 1998.

8 The CB tree: a practical concurrent self-adjusting search tree / Y. Afek [и др.] // Distributed computing. — 2014. — Т. 27, № 6. — С. 393–417.

9 The splay-list: A distribution-adaptive concurrent skip-list / V. Aksenov [и др.] // Distributed Computing. — 2023. — С. 1–24.