Проведем нагрузочное тестирование нашего потокобезопасного сервера с помощью *wrk* в несколько соединений. Проанализируем результаты профилирования под нагрузкой с помощью *async-profiler*. Профилирование будем проводить трех типов:

- *CPU profiling*;
- *Allocation profiling*;
- *Lock profiling*.

Вначале рассмотрим случай, когда сброс на диск выполняется одним отдельным потоком, а затем проверим улучшит ли увеличение числа потоков результаты профилирования.

1 CPU profiling

Заполним наше хранилище небольшим количеством данных (28M):

```
$ wrk -t4 -c4 -d240s -R10000 -s ./wrk/put.lua --latency
```

Если попрофилировать под данной нагрузкой в режиме *cpu*, то наибольшее количество времени занимает передача данных по сети. Сама вставка данных в каждом потоке занимает не больше $\approx 2\%$ всего времени (рис. 1).

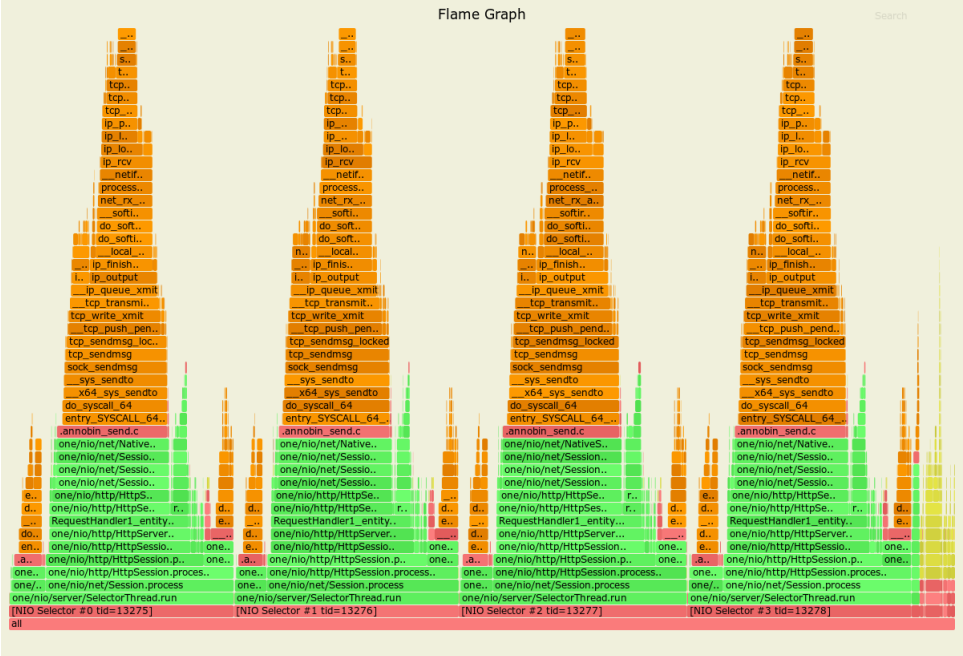

Рис. 1

Проверим, как наш сервер будет обрабатывать 10000 запросов на чтение в секунду в течение минуты:

```
$ wrk -t4 -c4 -d60s -R10000 -s ./wrk/get.lua --latency
```

Как видно на рис. 2 обработка запросов на чтение при этой нагрузке в каждом потоке не превышало 12%.

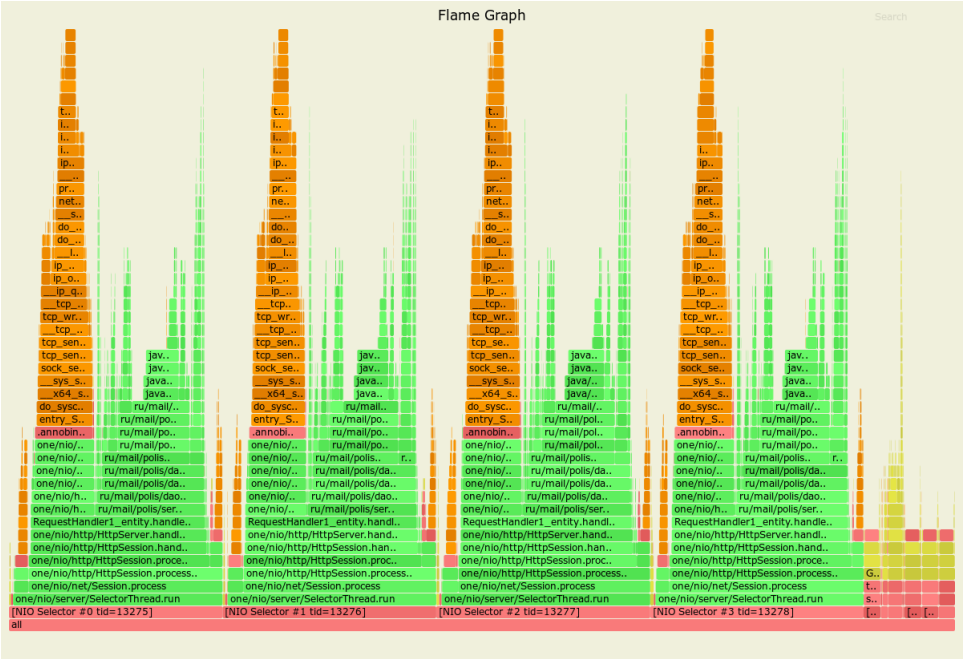

Рис. 2

Рассмотрим теперь результаты профилирования, когда сервер был нагружен запросами на удаление:

```
$ wrk -t4 -c4 -d60s -R10000 -s ./wrk/delete.lua --latency
```

На рис. 3 видно, что удаление данных в каждом потоке занимает небольшое количество времени ($\approx 2\%$).

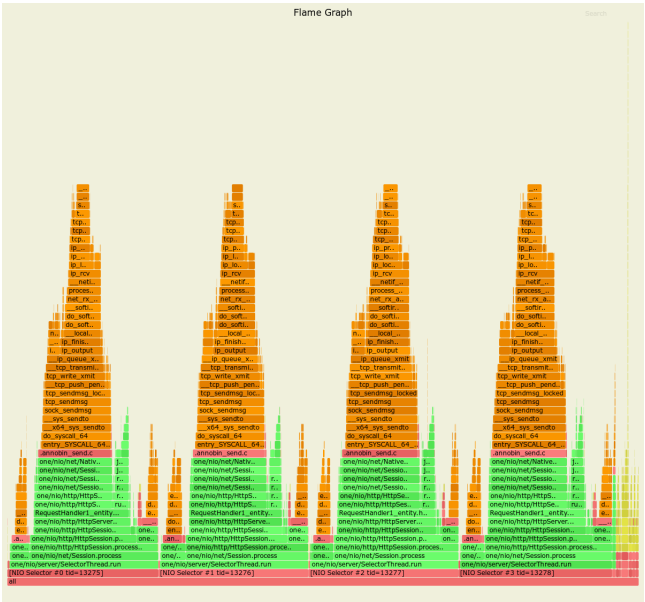

Рис. 3

В этих трех случаях максимальное время задержки не превышало *60ms*.

2 Allocation profiling

Проверим, какие результаты профилирования под этой же нагрузкой будут в режиме *alloc*. На рис. 4 видно, что при запросах на вставку данных в куче большую часть памяти занимают объекты, создаваемые для парсинга запроса и отпраки ответа. При сохранении данных в *Memory table* не более 2% памяти занимали объекты, являющиеся абстрацией строк (объекты класса *Row*) и абстракцией ячеек таблицы (объекты класса *Value*).

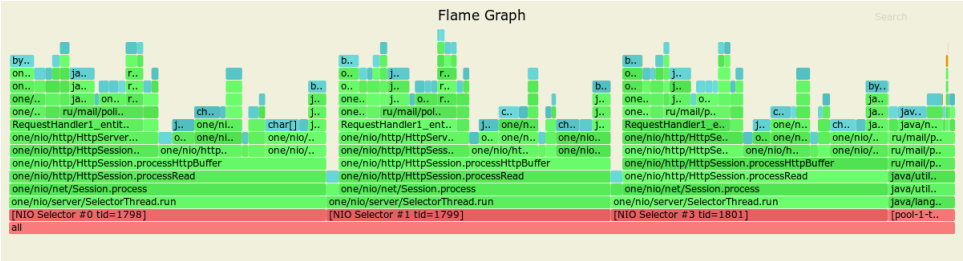

Рис. 4

При запросах на чтение в куче наибольшее количество памяти (суммарно не более 20%) было выделено объектам класса *DirectByteBuffer*, который создавались при вызове метода *SSTable.iterator* (рис .5).

Рис. 5

При запросах на удаление данных, как и при вставке большую часть памяти занимают объекты, необходимые при работе с сетевыми запросами, а при удалении данных не более 3% памяти занимали объекты, являющиеся абстрацией строк (объекты класса *Row*) и абстракцией ячеек таблицы (объекты класса *Value*) (рис. 6).

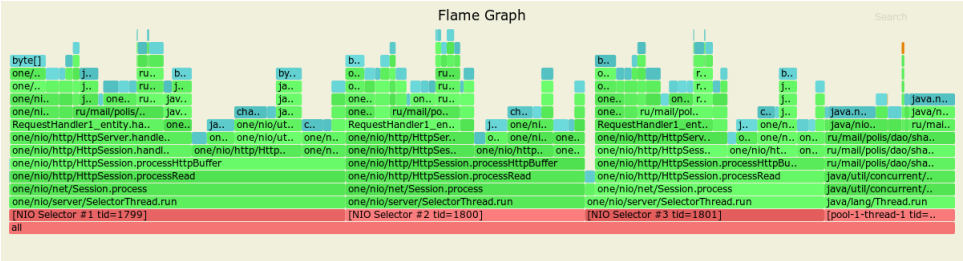

Рис. 6

3 Lock profiling

Используя режим *lock*, рассмотрим: как происходят блокировки на нашем сервере при различных запросах. На рис. 7 можно заметить, что в 2-ух потоках 28% времени занимала вставка данных. Связанно это с тем, что эти потоки $\approx 15\%$ времени ждали, чтобы взять блокировку на запись при вызове метода *MemTablePool.setToFlush*.

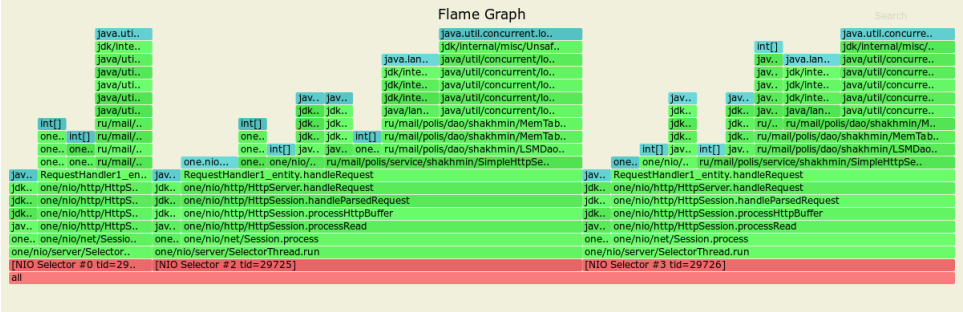

Рис. 7

Для того тобы посмореть сколько времени потоки находятся при блокировки на чтение, нагрузим наш сервер одновременно запросами на запись и на чтение, предварительно заполнив хранилище небольшим объемом данных:

```
$ wrk -t4 -c4 -d60s -R100000 -s ./wrk/put.lua --latency
$ wrk -t4 -c4 -d60s -R10000 -s ./wrk/get.lua --latency
```

На рис. 8 видно, что одновременно запросы на запись и на чтение обрабатывал только один поток, причем ожидание блокировок как на запись, так и на чтение в этом потоке заняло одинаковое количество времени по $\approx 17\%$. Также на этом рисунке можно заметить, что поток, который сбрасывает данные на диск тоже ждал блокировки на запись при вызове метода *MemTablePool.flushed*.

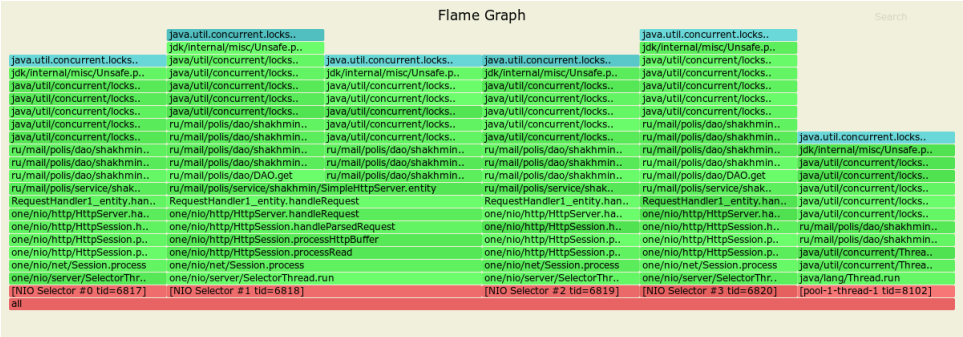

Рис. 8

При запросах на удаление время ожидания у потоков связано только с взятием блокировки на запись при вызове метода *MemTablePool.setToFlush* (рис. 9).

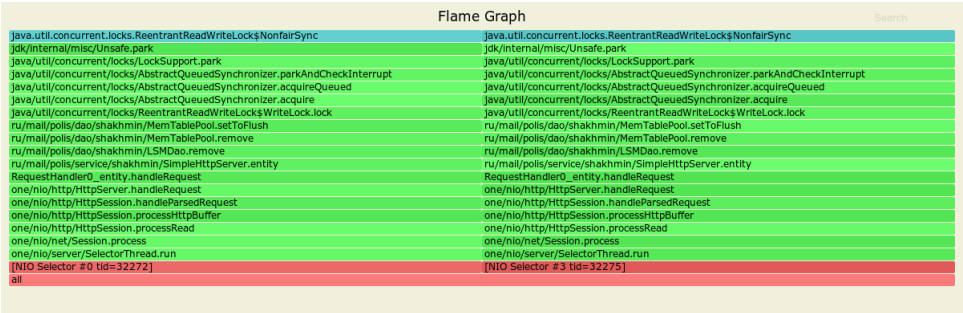

Рис. 9

4 Несколько потоков для сброса на диск

Интересно было узнать: улучшит ли результаты профилирование увеличение числа потоков для сбрасывания данных на диск. Для этого увеличим пул потоков до размера $N_{cpu}+1$. В данном случае это значение оказалось равным 5. Также увеличим количество запросов в секунду до 100000. Профилирование будем проводить в режиме *cpu*.

Рассмотрим рис. 10 и рис. 11. На них представлены результаты профилирования при одной и той же нагрузке, но на рис. 10 один поток обрабатывал сброс на диск, а на рис. 11 пять потоков. Сама вставка данных заняла одинаковое количество времени, но уменьшилось время сброса данных на диск. Если с одним потоко это занимало $\approx 4\%$ всего времени, то при пяти потоках у каждого потока это заняло не больше 1% времени.

При запросах на удаление ситуация оказалась такой же, как и при запросах на вставку данных (рис. 12 и рис. 13).

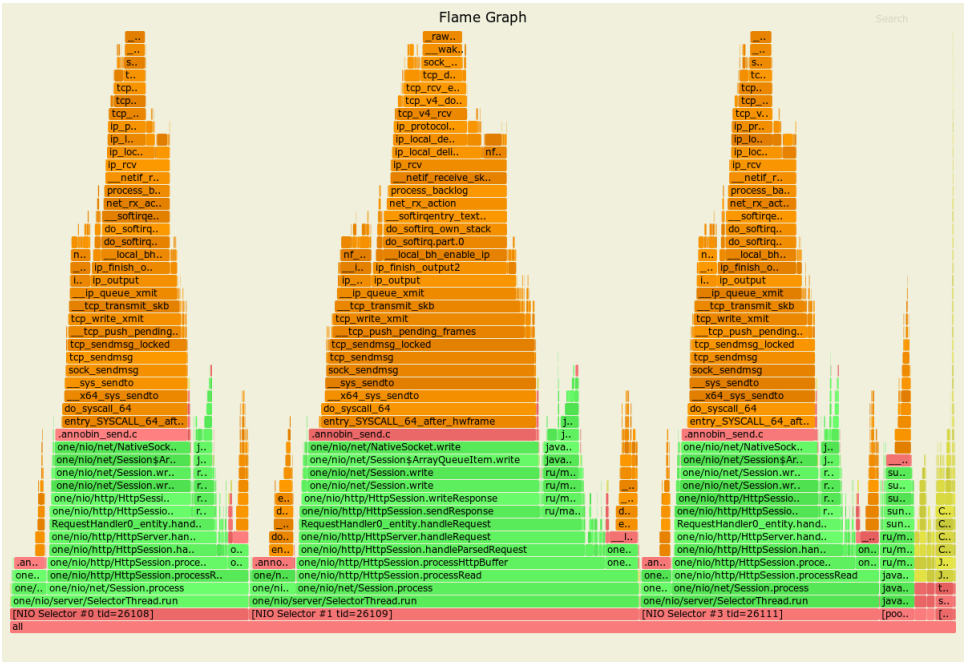
Flame Graph
Search
[NIO Selector #0 tid=26108]
[NIO Selector #1 tid=26109]
[NIO Selector #3 tid=26111]
all

Рис. 10


Flame Graph
Search
[NIO Selector #0 tid=25125]
[NIO Selector #2 tid=25127]
[NIO Selector #3 tid=25128]
all

Рис. 11

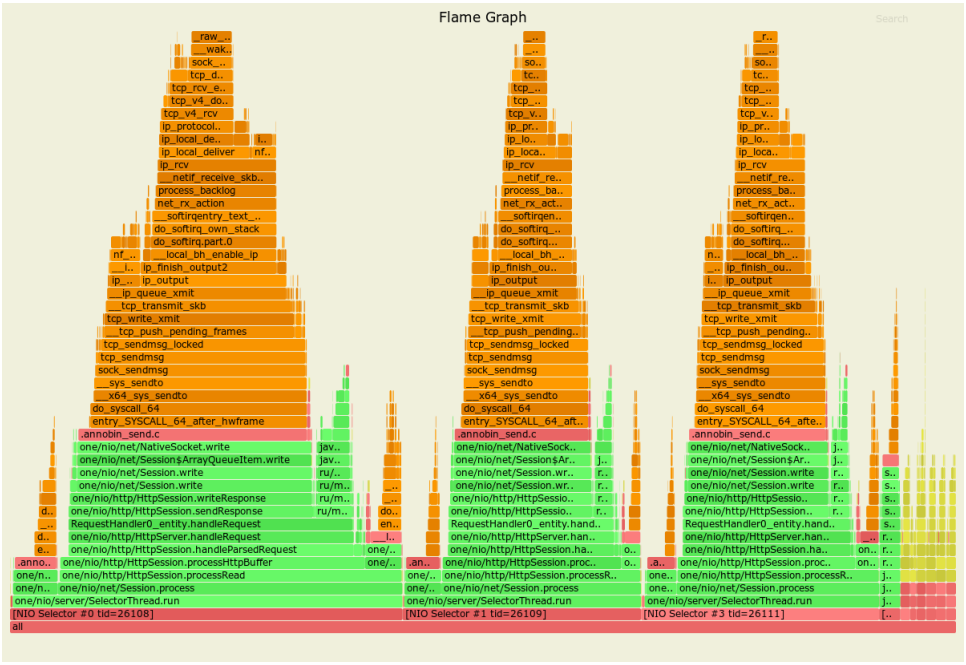
Flame Graph
Search
[NIO Selector #0 tid=26108]
[NIO Selector #1 tid=26109]
[NIO Selector #3 tid=26111]
all

Рис. 12


Flame Graph
Search
[NIO Selector #0 tid=25125]
[NIO Selector #2 tid=25127]
[NIO Selector #3 tid=25128]
all

Рис. 13