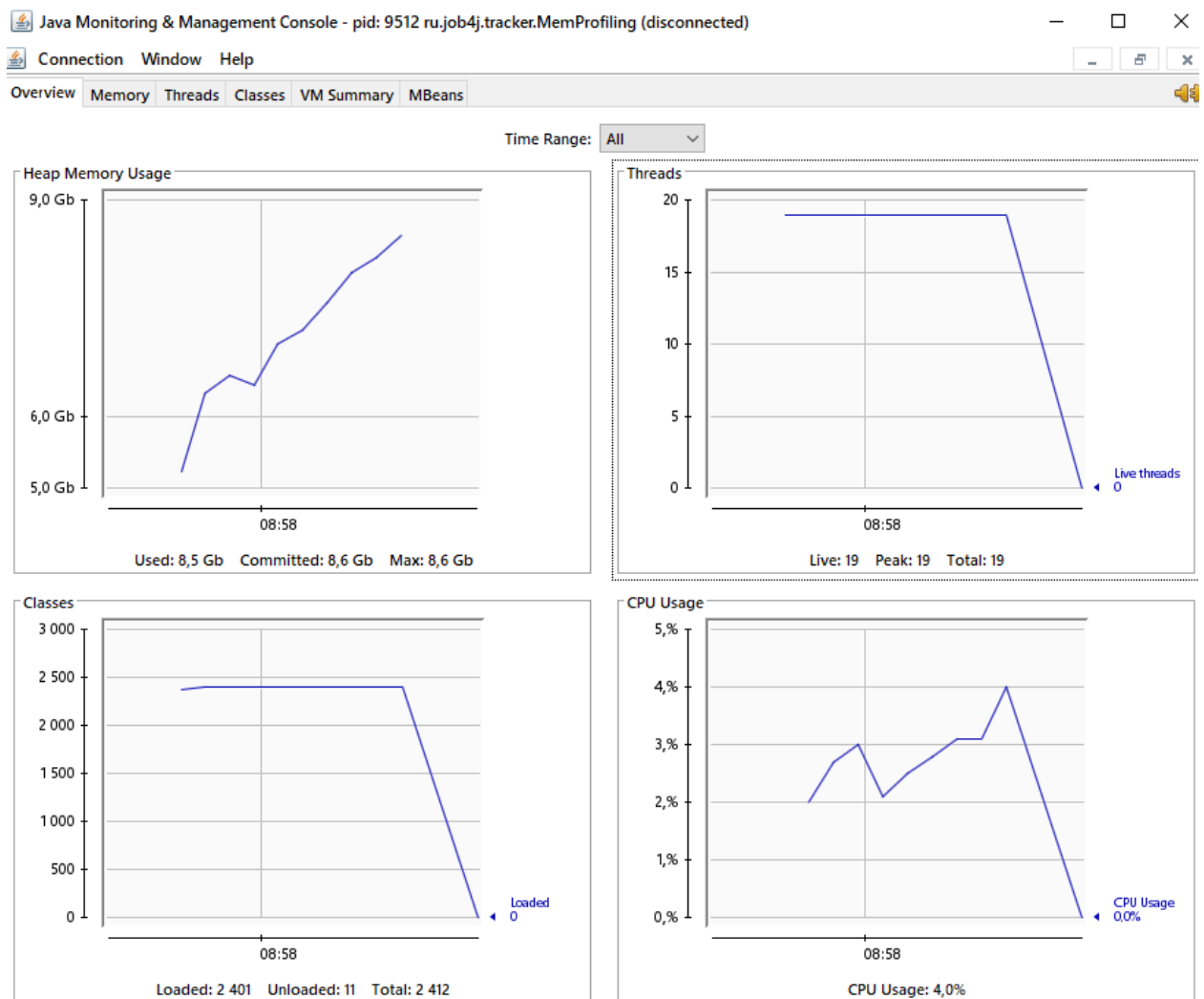


1. Хорошенько загружаем «мусором» нашу программу. В данном случае с помощью StringBuffer (рекурсия имени Item, т.е. каждое имя содержит все имена предыдущих объектов + i). Таким образом мы точно используем весь heap space, максимальное значение которого равно 8,6Гб.

```
public static void main(String[] args) {  
    MemTracker memTracker = new MemTracker();  
    StringBuffer stringBuffer = new StringBuffer();  
    for (int i = 0; i < 100000; i++) {  
        stringBuffer.append("TestItem" + i);  
        Item testItem = new Item(i, stringBuffer.toString());  
        memTracker.add(testItem);  
        System.out.println(testItem);  
    }  
}
```



Так как все объекты у нас уникальные и у каждого есть рабочая ссылка, соответственно все они должны жить долго, то сборщик мусора хоть и запускается для чистки каких-то служебных скрытых объектов (эту очистка выдает не идеальная растущая прямая памяти хипа), но в целом не влияет на геометрическую прогрессию заполнения хипа (спасибо стрингбафферу за растущий размер имени объекта Item). По итогу 8,6 Гб нам хватило на создание всего 36409 объектов (последний ID созданного Item).

```

m36391TestItem36392TestItem36393TestItem36394TestItem36395TestItem36396
created = 01-августа-понедельник-2022 08:58:24}
Item {id=36409,
name
='TestItem0TestItem1TestItem2TestItem3TestItem4TestItem5TestItem6TestItem
TestItem23TestItem24TestItem25TestItem26TestItem27TestItem28TestItem29Te
stItem45TestItem46TestItem47TestItem48TestItem49TestItem50TestItem51Test
Item67TestItem68TestItem69TestItem70TestItem71TestItem72TestItem73TestIt

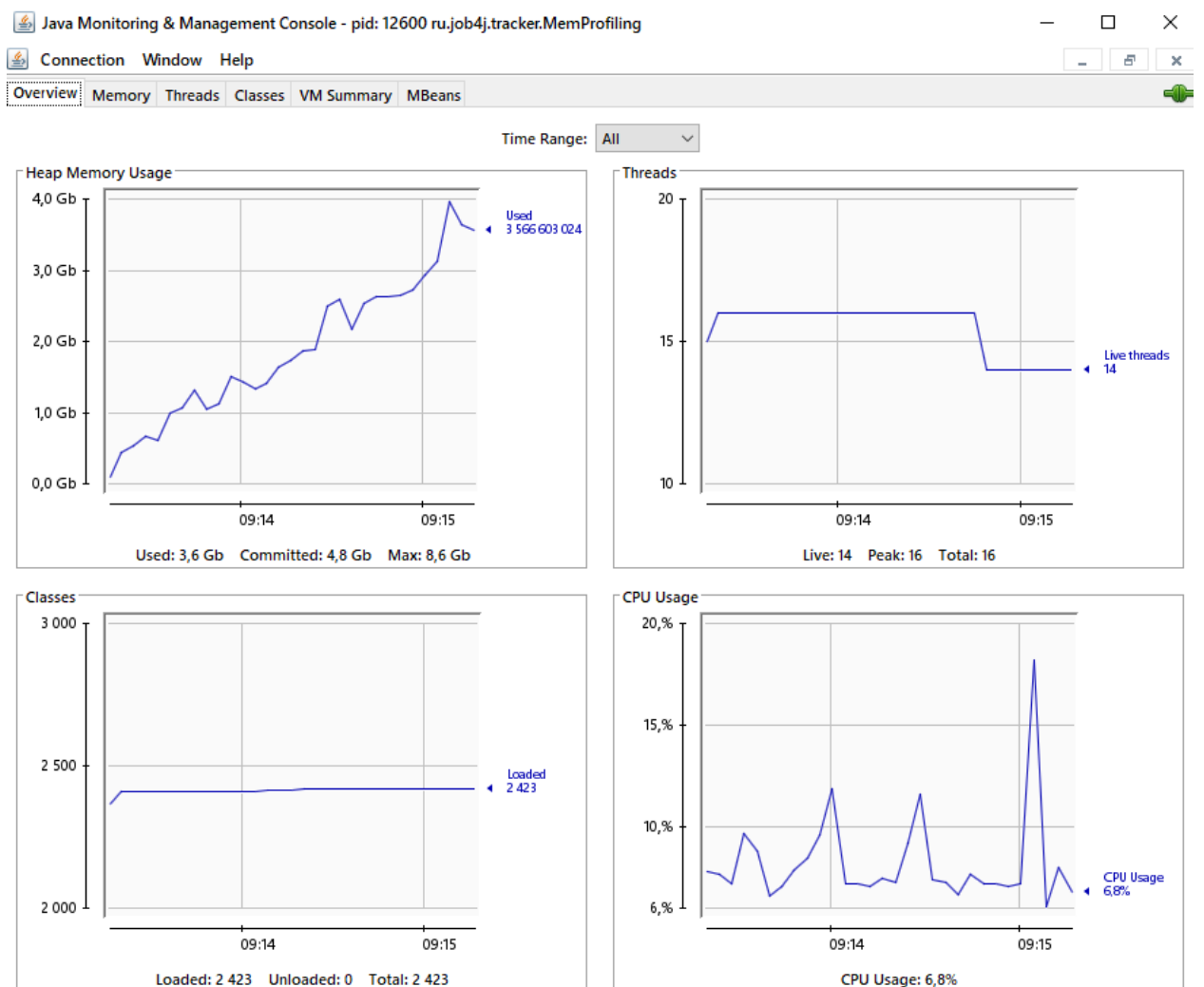
```

2. Прекратим мучать систему несуществующими начальными параметрами и просто создадим 30 млн. заявок

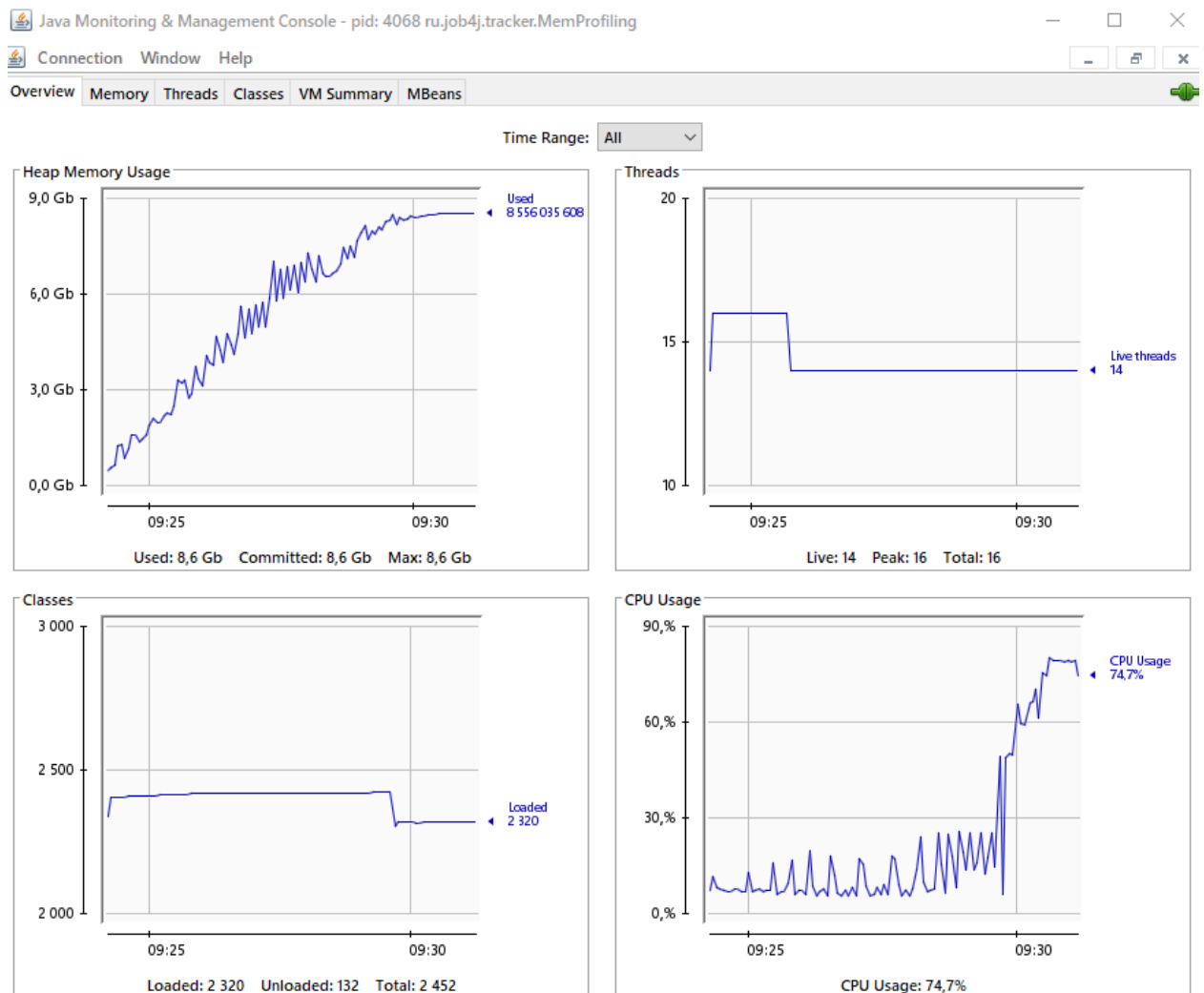
```

public static void main(String[] args) {
    MemTracker memTracker = new MemTracker();
    for (int i = 0; i < 300000000; i++) {
        Item testItem = new Item();
        memTracker.add(testItem);
        System.out.println(testItem);
    }
}

```



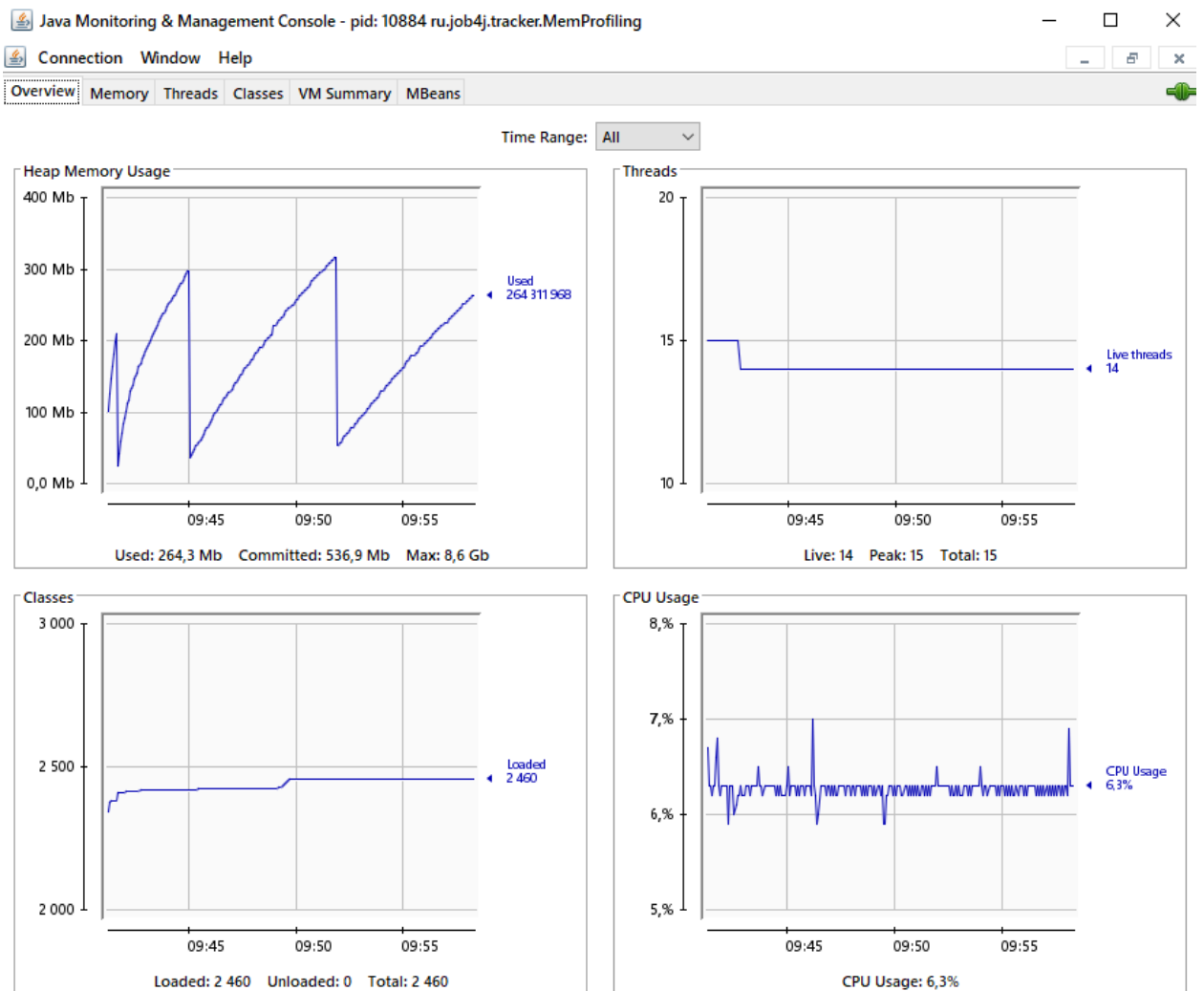
Хип по-прежнему геометрически растет (но в данном случае работа GC уже более очевидна как по освобождению хипа, так и по загрузке ЦП) и в пике достигает почти половины от максимально возможного объема кучи. Делаем вывод, что 100млн. заявок уронит нашу систему. Попробуем!!!



Уронить не получилось.... По графику видно, что как только мы подползли к верхней границе максимума хипа, процессор взял на себя основную нагрузку, а хип стал статичным... Т.е. подобравшись к верхней границе хипа наша производительность упала с сотен тысяч заявок в секунду до десятка заявок в секунду, если не меньше. Сижу думаю....

- Добавим еще функцию удаления заявки и посмотрим как программа поведет себя при заполнении + удалении тех же 100млн. заявок

```
public static void main(String[] args) {
    MemTracker memTracker = new MemTracker();
    for (int i = 0; i < 100000000; i++) {
        Item testItem = new Item();
        memTracker.add(testItem);
        if (i % 2 == 0) {
            memTracker.delete(id: i - 1);
        }
        System.out.println(testItem);
    }
}
```



ВЫВОД: Скорость очень сильно упала (вычисления все-же требуют времени) и ситуация относительно выравнялась. На графиках видно, что при определенном уровне количества заявок сортировщик мусора успевает справляться со своими задачами и при поддержании общего количества заявок на определенном уровне система вполне стабильно работает. Скорость же напрямую зависит от количества запрашиваемых операций над заявкой (добавление\удаление\поиск\изменение).

Используемые инструменты: jconsole.