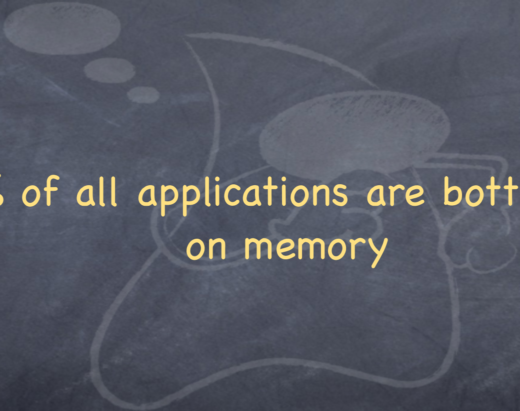
THE TROUBLE WITH

MEMORY

OUR MARKETING SLIDE

- Kirk Pepperdine
  - Authors of jPDM, a performance diagnostic model
- Co-founded jClarity
  - Building the smart generation of performance diagnostic tooling
    - Bring predictability into the diagnostic process
- Co-founded JCrete
  - The hotest unconference on the planet
- Java Champion(s)

What is your performance trouble spot



> 70% of all applications are bottlenecked on memory

and no,
Garbage Collection
is not a fault!!!!

DO YOU USE

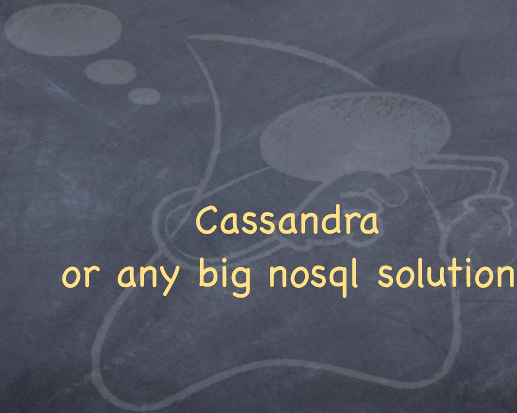
DO YOU USE

Spring Boot

DO YOU USE

Cassandra

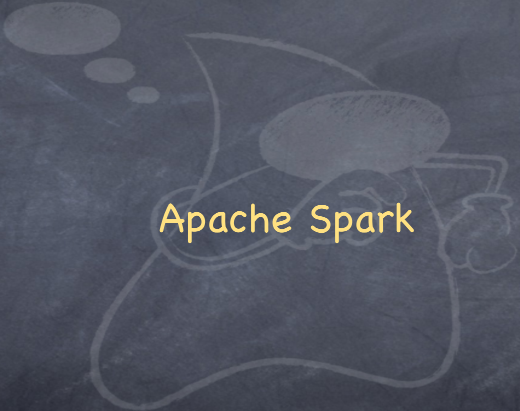
DO YOU USE



Cassandra

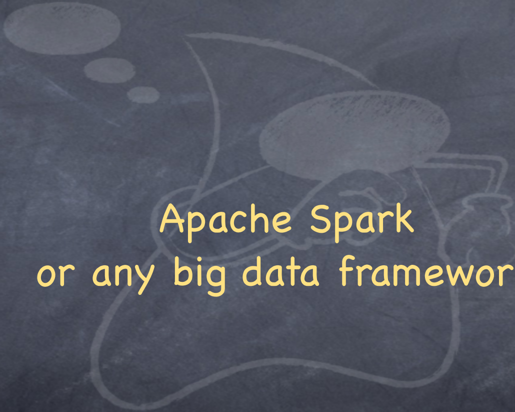
or any big nosql solution

DO YOU USE



Apache Spark

DO YOU USE

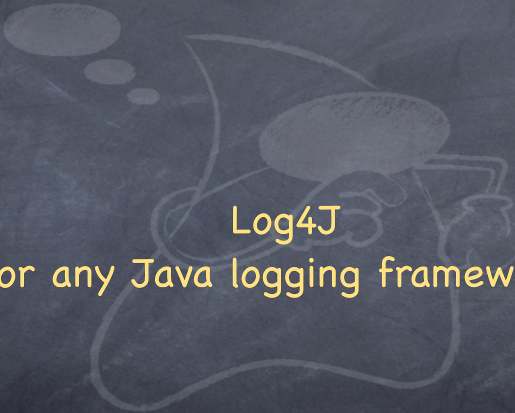


Apache Spark
or any big data framework

DO YOU USE

Log4J

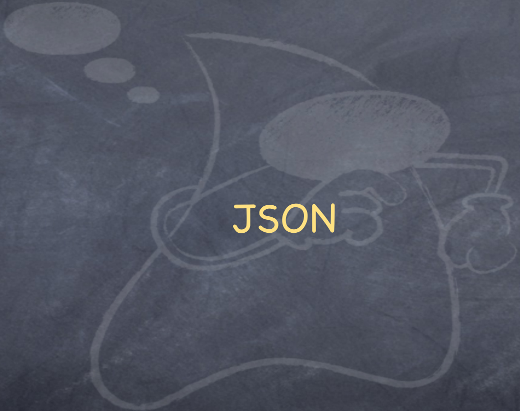
DO YOU USE



Log4J

or any Java logging framework

DO YOU USE



JSON

DO YOU USE

JSON

With almost any Marshalling protocol

DO YOU USE

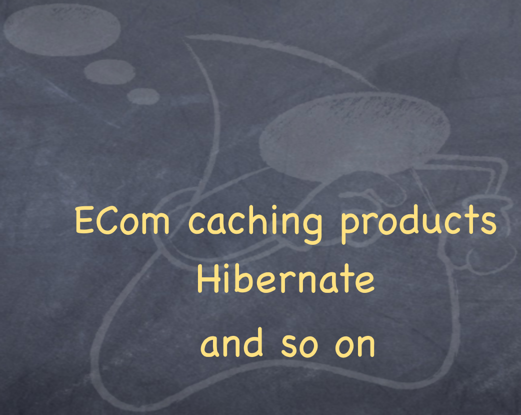
ECom caching products

DO YOU USE

ECom caching products

Hibernate

DO YOU USE

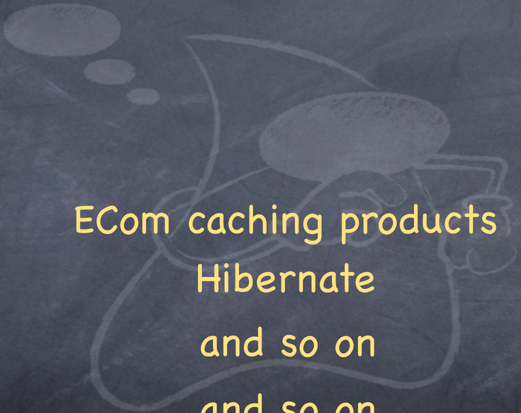


ECom caching products

Hibernate

and so on

DO YOU USE



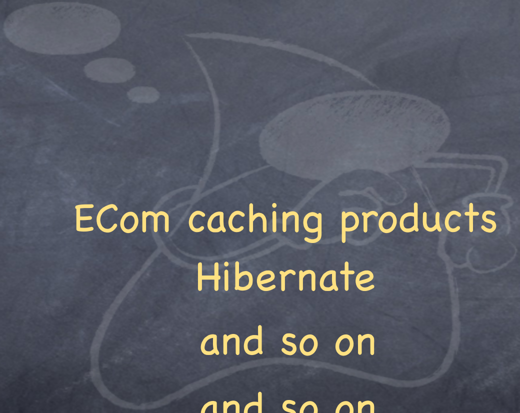
ECom caching products

Hibernate

and so on

and so on

DO YOU USE



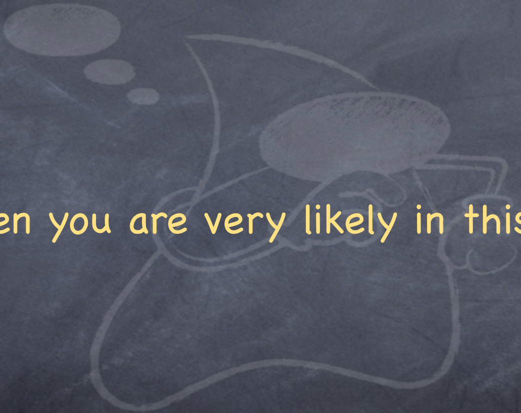
ECom caching products

Hibernate

and so on

and so on

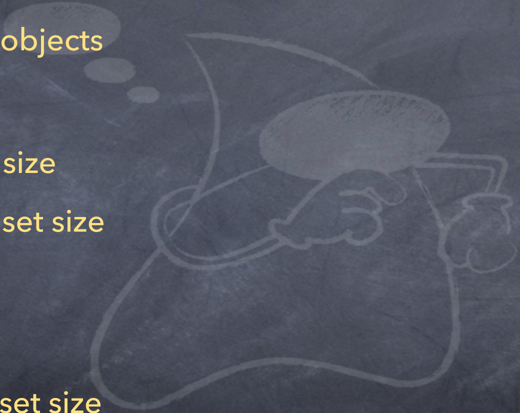
and so on



then you are very likely in this 70%

PROBLEMS

- High memory churn rates
  - many temporary objects
- Large live data set size
  - inflated live data set size
    - loitering
- Unstable live data set size
  - memory leak



WAR STORIES

- Reduced allocation rates from 1.8gb/sec to 0
  - tps jumped from 400,000 to 25,000,000!!!
- Stripped all logging our of a transactional engine
  - Throughput jumped by a factor of 4x
- Wrapped 2 logging statements in a web socket framework
  - Memory churn reduced by a factor of 2

ALLOCATION SITE

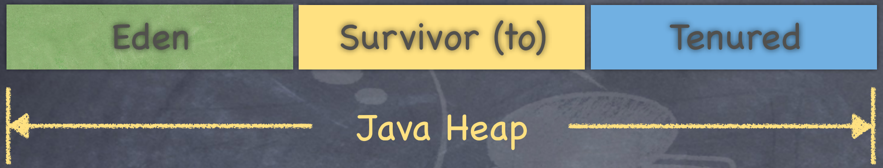
```
Foo foo = new Foo();
```

forms an allocation site

```
0: new           #2    // class java/lang/Object
2: dup
4: invokespecial #1    // Method java/lang/Object."<init>":()V
```

- Allocation will (mostly) occur in Java heap
  - fast path
  - slow path
  - small objects maybe optimized to an on-stack allocation

JAVA HEAP



Eden

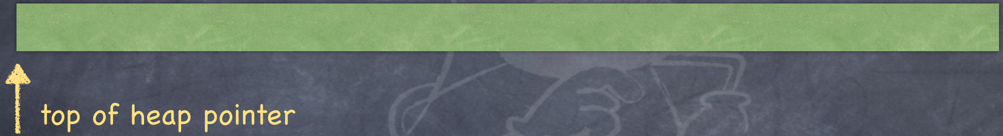
Survivor (to)

Tenured

Java Heap

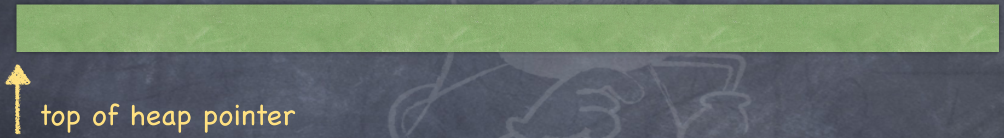
- Java Heap is made of;
  - Eden - nursery
  - Survivor - intermediate pool designed to delay promotion
  - Tenured - to hold long lived data
- Each space contributes to a different set of problems
  - All affect GC overhead

EDEN ALLOCATIONS



top of heap pointer

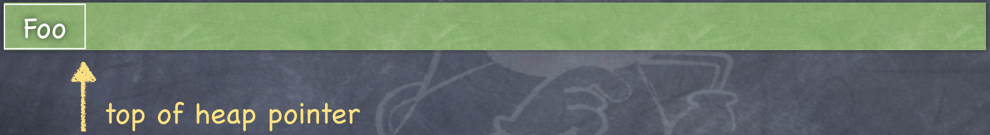
OBJECT ALLOCATION



top of heap pointer

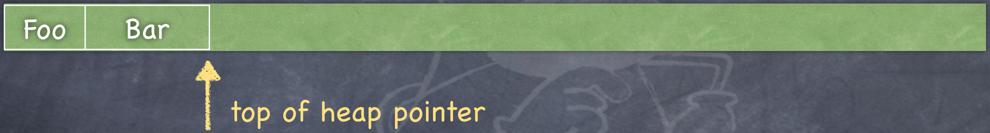
```
Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];
```

OBJECT ALLOCATION



```
Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];
```

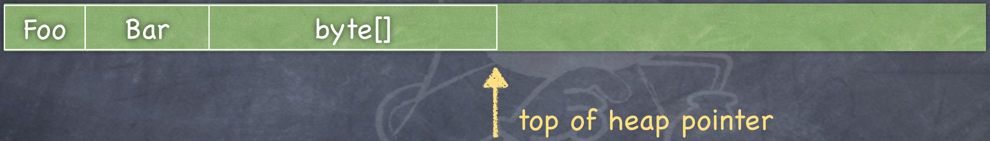
OBJECT ALLOCATION



top of heap pointer

```
Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];
```

OBJECT ALLOCATION



Foo

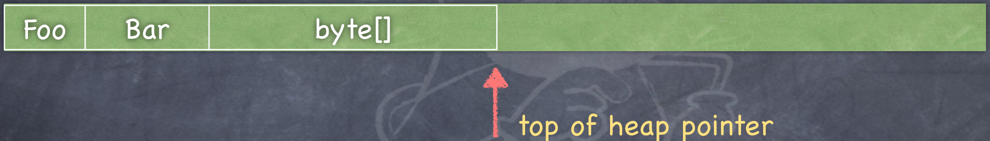
Bar

byte[]

top of heap pointer

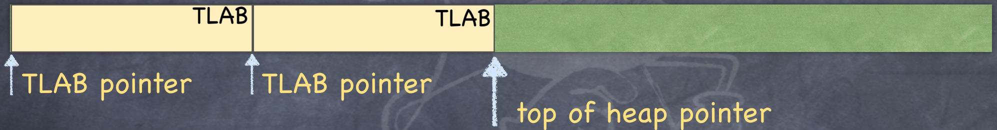
```
Foo foo = new Foo();
Bar bar = new Bar();
byte[] array = new byte[N];
```

OBJECT ALLOCATION



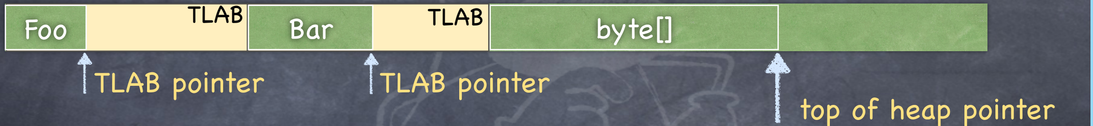
- In multi-threaded apps, top of heap pointer must be surrounded by barriers
  - single threads allocation
  - hot memory address
    - solved by stripping (Thread local allocation blocks)

TLAB ALLOCATION



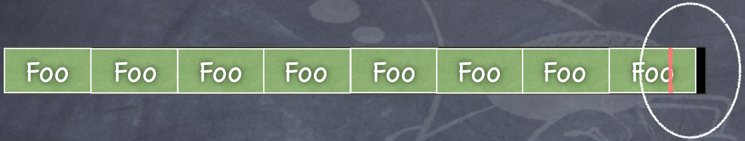
- Assume 2 threads
  - each thread will have it's own (set of) TLAB(s)

TLAB ALLOCATIONS



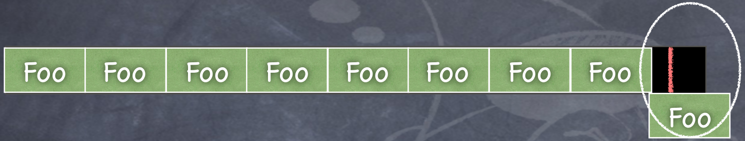
- Thread 1 -> Foo foo = new Foo(); byte[] array = new byte[N];
  - byte[] doesn't fit in a TLAB
- Thread 2 -> Bar bar = new Bar();

TLAB WASTE %



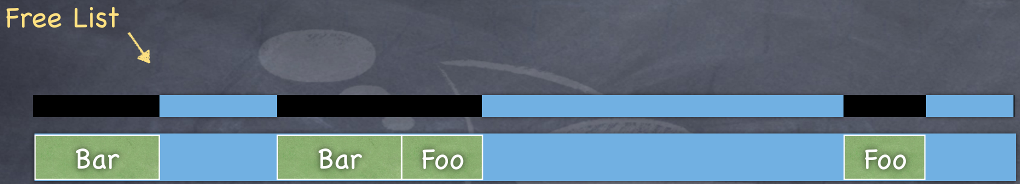
- Allocation failure to prevent buffer overflow
- waste up to 1% of a TLAB

TLAB WASTE %



- Allocation failure to prevent buffer overflow
- waste up to 1% of a TLAB

TENURED SPACE



- Allocations in tenured make use of a free list
  - free list allocation is ~10x the cost of bump and run
- Data in tenured tends to be long lived
  - amount of data in tenured do affect GC pause times

PROBLEMS

- High memory churn rates
  - many temporary objects

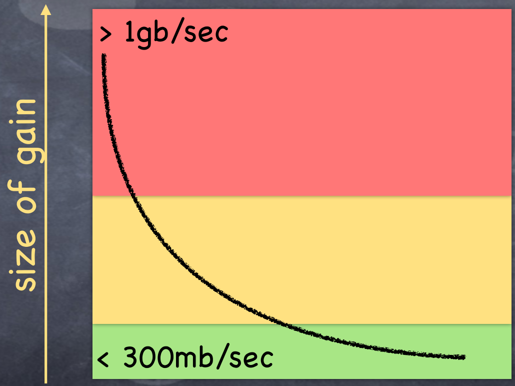
PROBLEMS

- High memory churn rates
  - many temporary objects

→

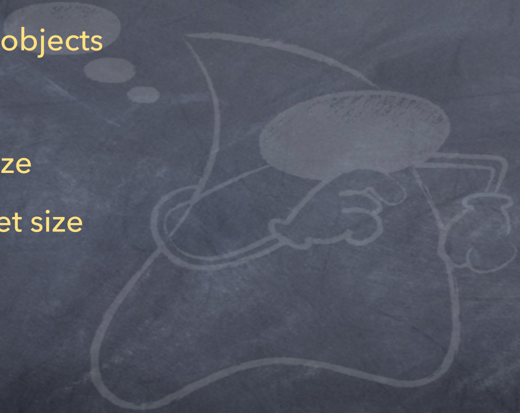
- Quickly fill Eden
  - frequent young gc cycles
  - speeds up aging
  - premature promotion
    - more frequent tenured cycles
    - increased copy costs
    - increased heap fragmentation
- Allocation is quick
  - quick * large number = slow

REDUCING ALLOCATIONS



PROBLEMS

- High memory churn rates
    - many temporary objects
- Large live data set size
    - inflated live data set size
        - loitering



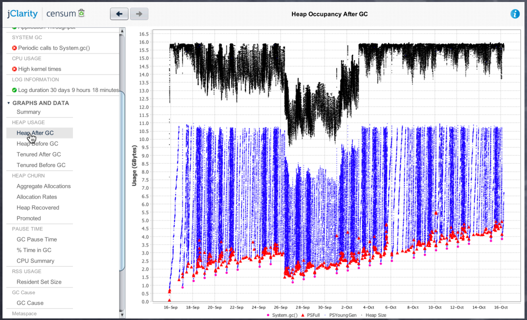
PROBLEMS

- High memory churn rates
  - many temporary objects
- Large live data set size
  - inflated live data set size
    - loitering

→

- inflated scan for root times
- reduced page locality
- Inflated compaction times
  - increase copy costs
  - likely less space to copy too

PAUSE VS OCCUPANCY



Heap Occupancy After GC

SYSTEM GC

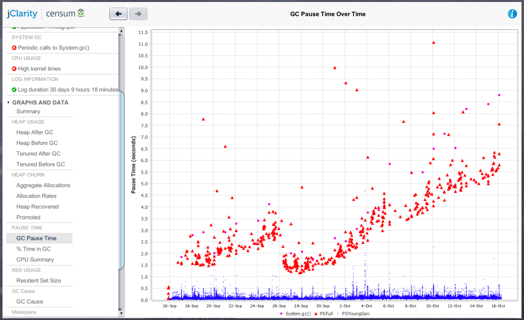
☒ Periodic calls to System.gc()

CPU USAGE

☒ High kernel times

LOG INFORMATION

✓ Log duration 30 days 9 hours 18 minutes



GC Pause Time Over Time

SYSTEM GC

☒ Periodic calls to System.gc()

CPU USAGE

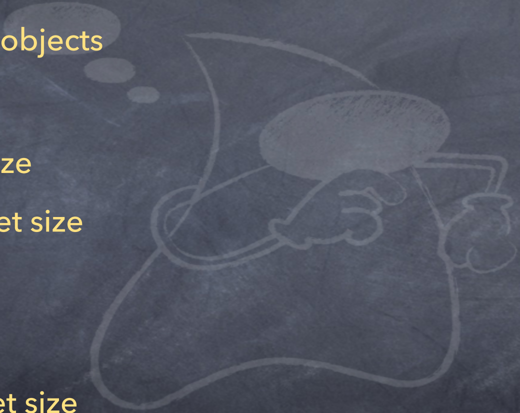
☒ High kernel times

LOG INFORMATION

✓ Log duration 30 days 9 hours 18 minutes

PROBLEMS

  - High memory churn rates
    - many temporary objects



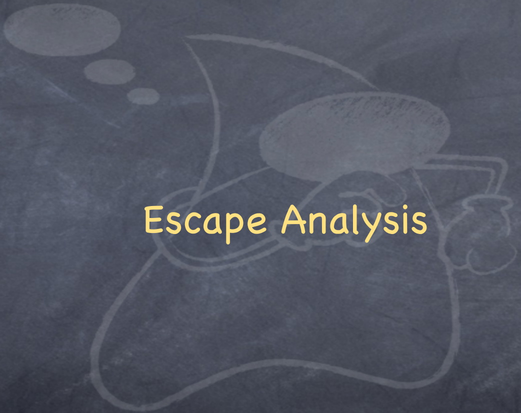
- Large live data set size
  - inflated live data set size
    - loitering

- Unstable live data set size
  - memory leak

PROBLEMS

- High memory churn rates
    - many temporary objects
- Large live data set size
    - inflated live data set size
        - loitering
- Unstable live data set size →
    - memory leak

- Eventually you run out of heap space
    - each app thread throws an OutOfMemoryError and terminates
        - JVM will shutdown with all non-daemon threads terminate



Escape Analysis

Demo time

Send us a Java 11 GC log or tweet about @jclarity #QConSF and #censum

and

get a free Censum License

Ask out my Java Performance Tuning Workshop