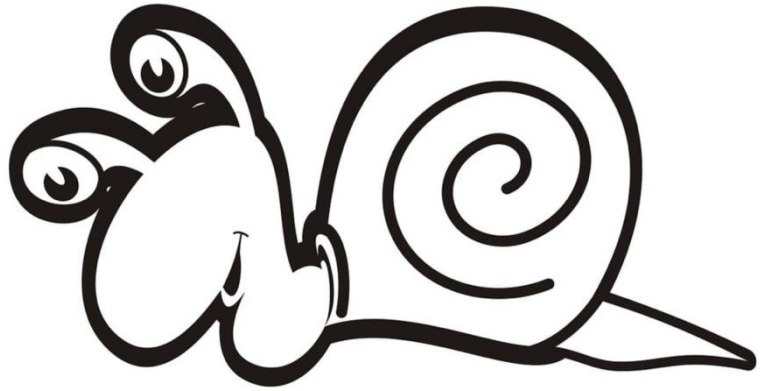


Оптимизируем Java код на примере Cassandra

JPoint 2026

Дмитрий Константинов

About me

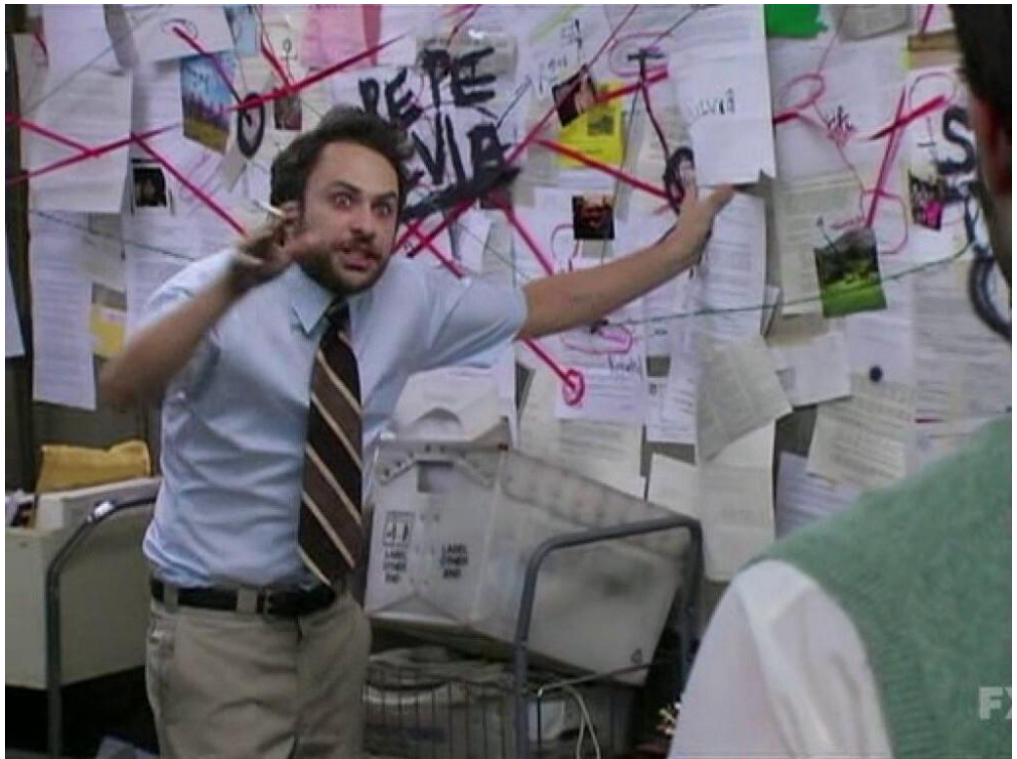
- Дмитрий Константинов
- Системный архитектор, пишу на Java
- Мне интересны Apache Cassandra, Zookeeper, Kafka, Hazelcast, etc
- Более 10 лет опыта с Cassandra
- Apache Cassandra committer
- https://t.me/cassandra_beyond

To tune, but why?

- I want lower latency / higher throughput for my Java app
- Less hardware -> cheaper and easier to manage

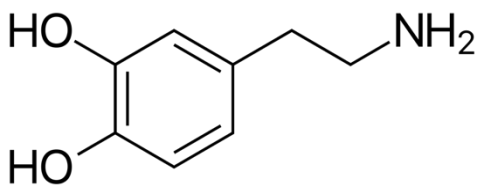
To tune, but why?

- I want lower latency / higher throughput for my Java app
- Less hardware -> cheaper and easier to manage
- **Performance analysis and tuning is a good way to study how something works**



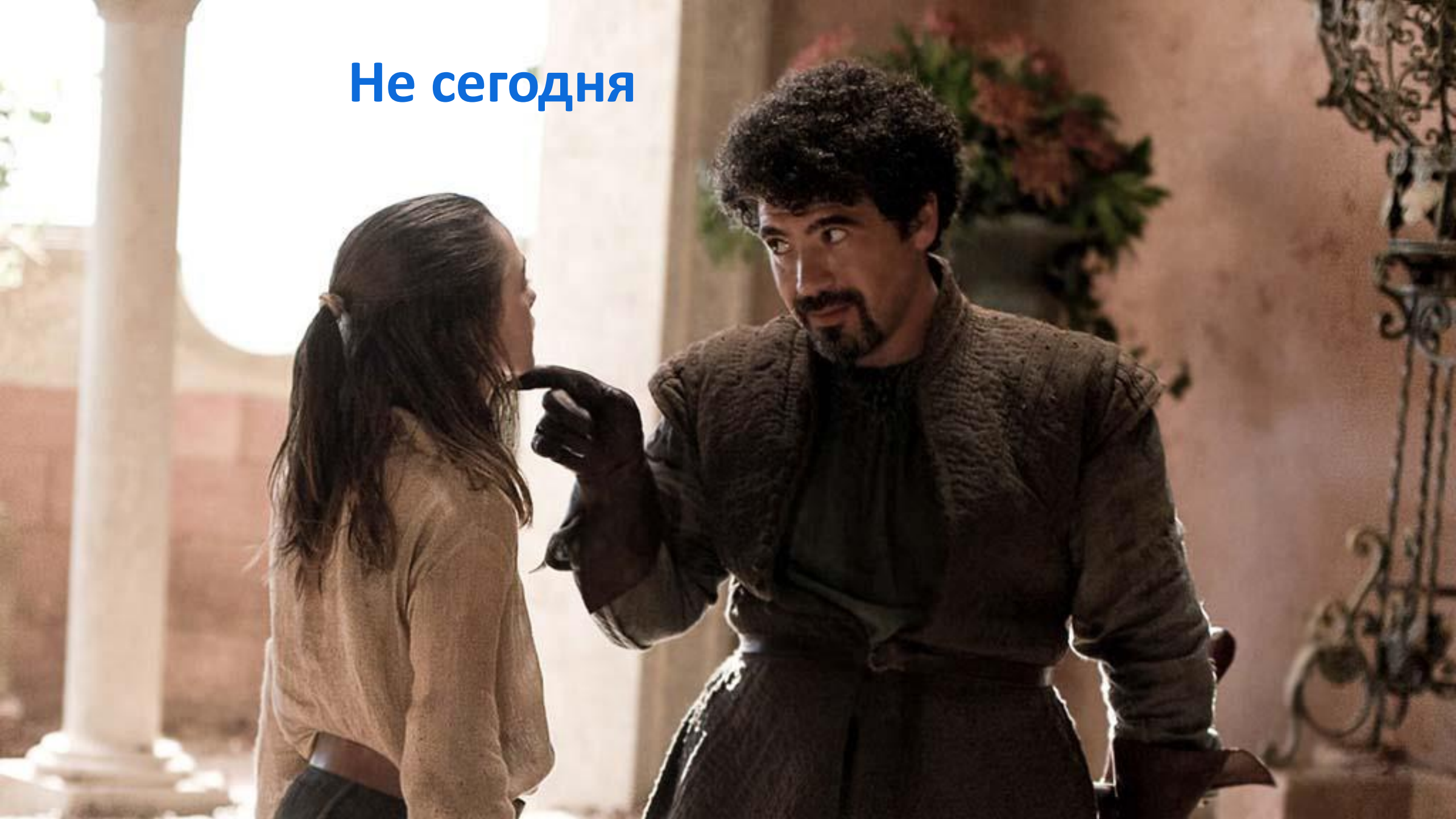
To tune, but why?

- I want lower latency / higher throughput for my Java app
- Less hardware -> cheaper and easier to manage
- Performance analysis and tuning is a good way to study how something works
- **I just like the process of perf tuning itself (good dopamine ☺)**



Java is slow, let’s rewrite everything in Rust

Не сегодня



But if I still want..

Доклад

Вызовите Rust — у нас тут медленно!

Как внедрить нативный движок на Rust в Java-приложение через Project Panama (FFI). Практика и подводные камни.

Денис Габайдулин
01.tech

JVM Languages

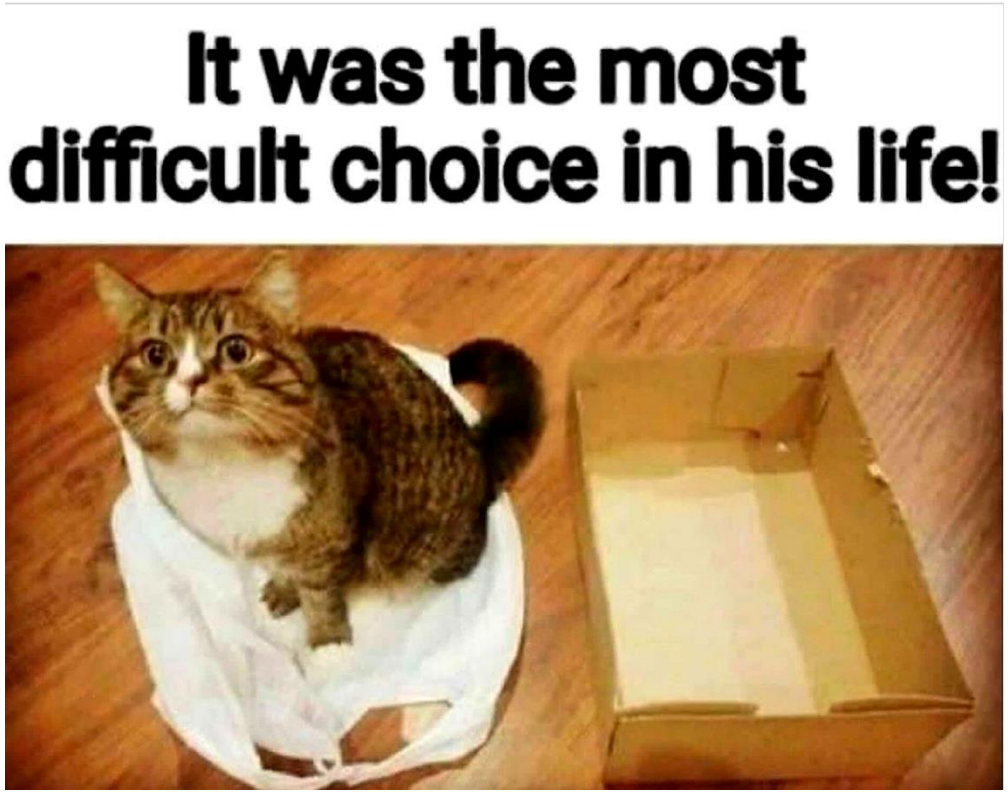

Intro
ROYAL

Let’s select a Java app to optimize…

- Well-known
- A large codebase
- Mature

Let’s select a Java app to optimize…

- Well-known
- A large codebase
- Mature
- I know internals good enough ☺



Let’s select a Java app to optimize…

- Apache Cassandra, JDK 17/21
- Trunk-based version (6.0-alpha) – able to contribute perf changes here

Let’s select a Java app to optimize…

- Apache Cassandra, JDK 17/21
- Trunk-based version (6.0-alpha)
- A single node - to make it simpler
- CPU-bound workload
  - Small rows/columns
  - Compaction/commit log – disabled (to not focus on IO today)

Workload

- Let’s focus on writes
- Inserted rows – it is rare we have just 1 column in real life
  - 1 partition text column (size = 15 symbols)
  - 1 clustering text column (size = 10 symbols)
  - 5 value text columns (size = 10 symbols each)

partition key	clustering key	value	value	value	value	value
…	…	…	…	…	…	…
	…	…	…	…	…	…
	…	…	…	…	…	…

Workload

- Let’s focus on writes
- Inserted rows – it is rare we have just 1 column in real life
  - 1 partition text column (size = 15 symbols)
  - 1 clustering text column (size = 10 symbols)
  - 5 value text columns (size = 10 symbols each)
- **2 kind of tests – real-time vs batch processing patterns**
  - individual inserts
  - batches (10 rows, single partition)

Workload

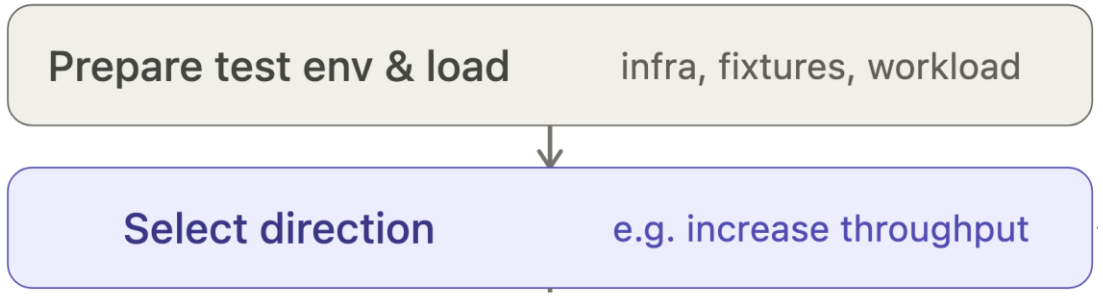
- Cassandra stress – OOB test load generator
- A modern alternative: easy-stress

```
./tools/bin/cassandra-stress \
  "user profile=./batch_profile_seq.yaml no-warmup ops(insert=1,partition-select=0) n=15m"
\
  -rate threads=300 -node <IP> \
  -mode native cql3 maxPending=256 connectionsPerHost=16
```

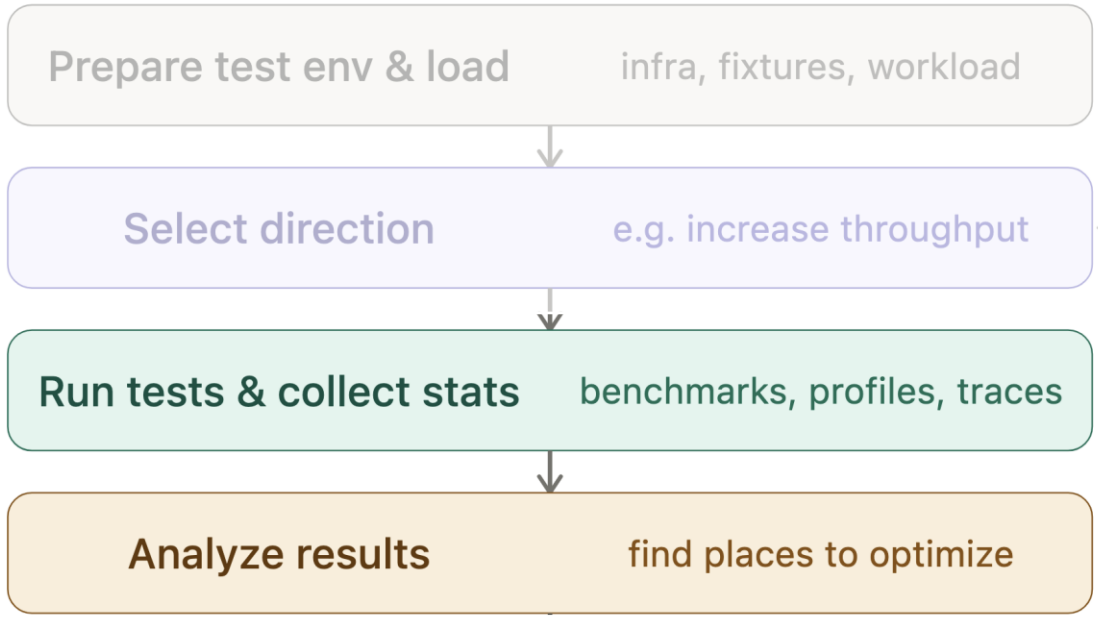
Setup - Hardware

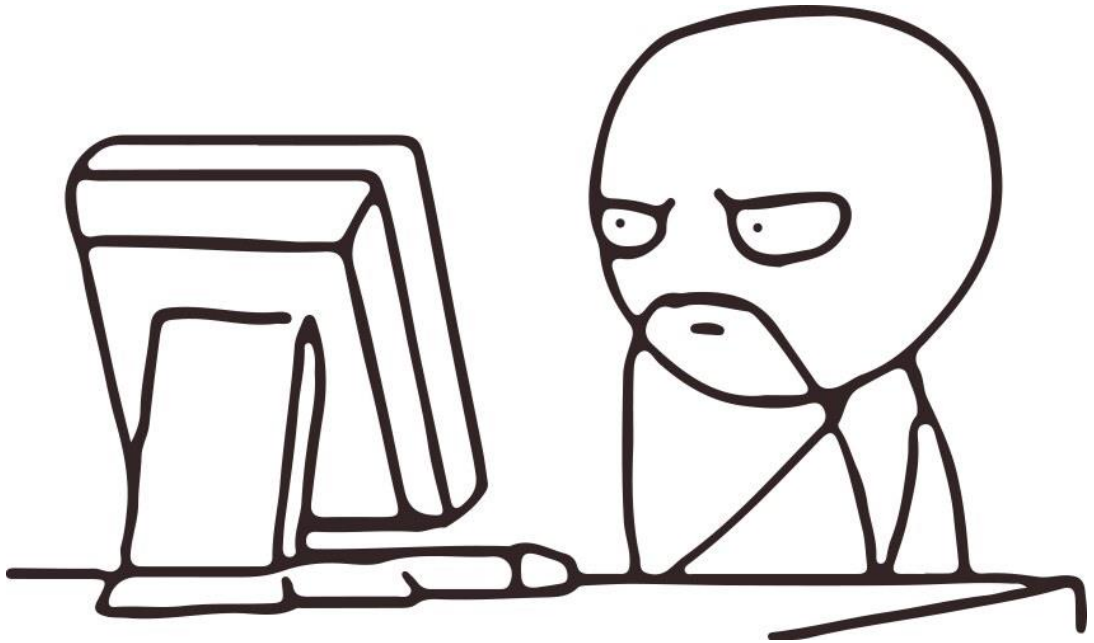
- AWS EC2
- Cassandra node:
  - m8i.4xlarge, CPU: Intel Xeon 16 vCPU, RAM: 64GiB
  - disk: gp3, IOPS limit: 3000, throughput: 200
- Cassandra stress node:
  - c5.9xlarge, CPU: Intel Xeon 36 vCPU, RAM: 72 GiB

Methodology

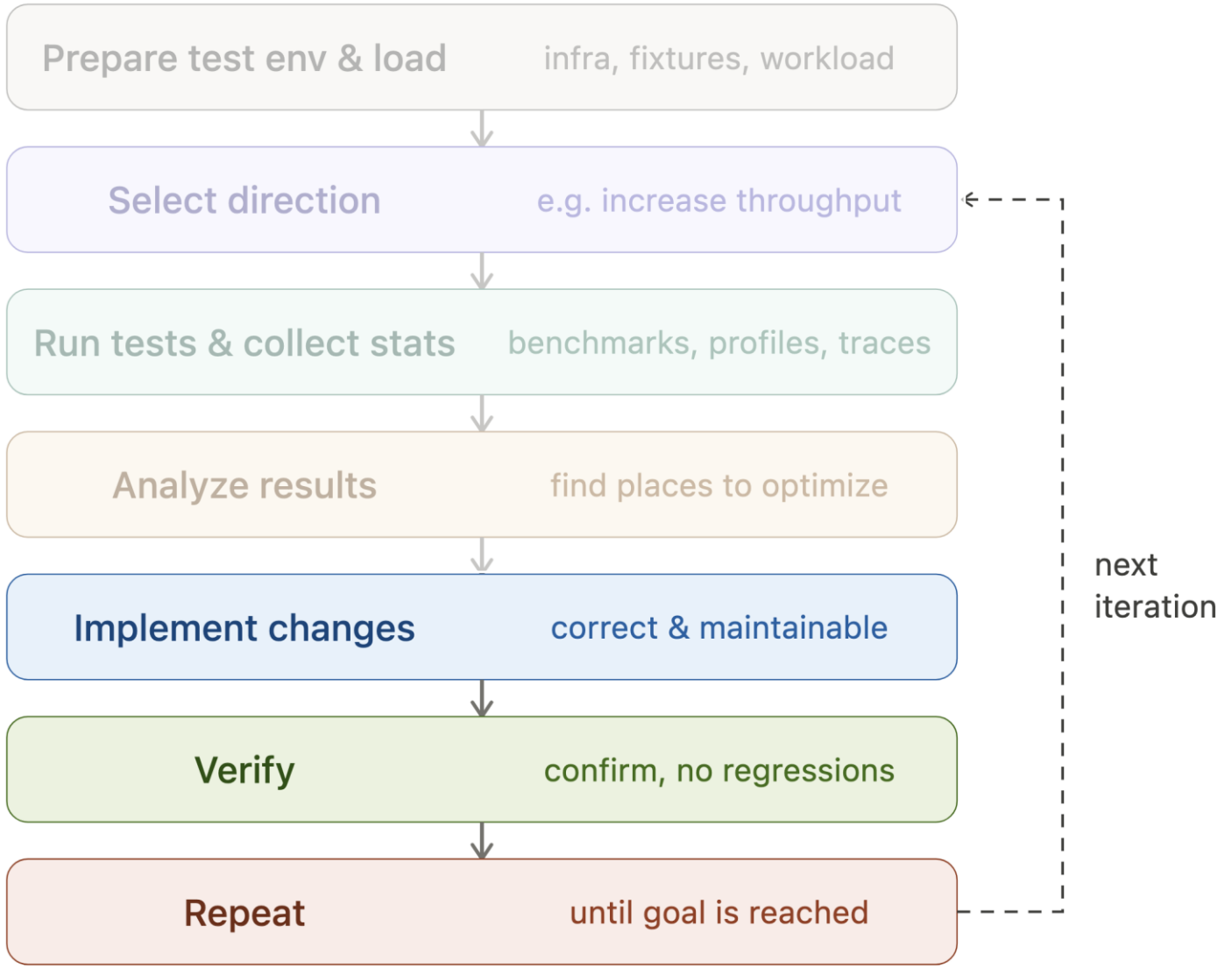


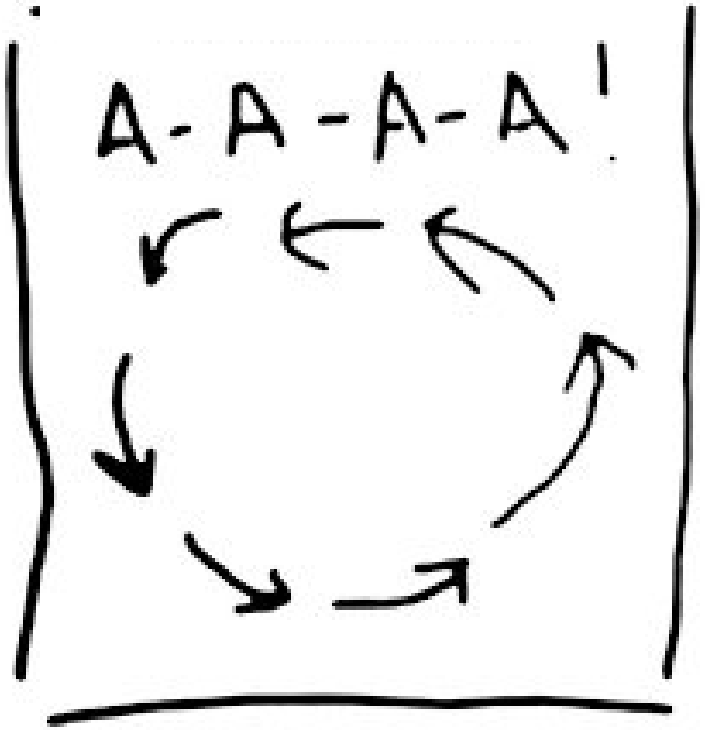
Methodology





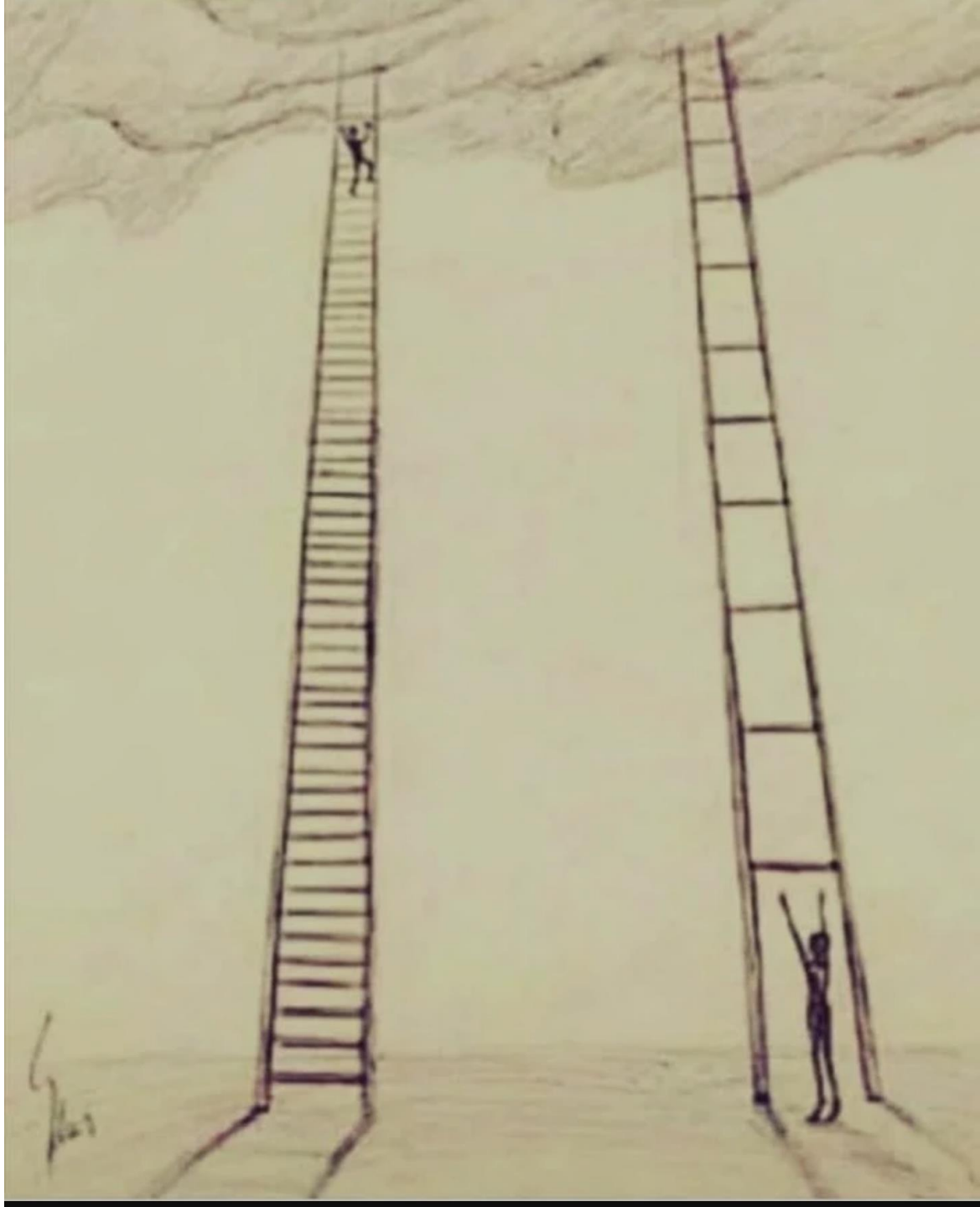
Methodology





Methodology

For a big project
it is rare a single round
rather a set of small incremental steps



Current results

~20 tickets to improve in different areas

CASSANDRA-20226	Reduce contention in MemtableAllocator.allocate
CASSANDRA-21083	Optimize memtable flush logic
CASSANDRA-21080	Switch LatencyMetrics to use ThreadLocalTimer/ThreadLocalCounter
CASSANDRA-21075	Optimize UTF8Validator.validate for ASCII prefixed Strings
CASSANDRA-21040	Optimize ModificationStatement#requiresRead
CASSANDRA-20885	Optimize org.apache.cassandra.audit.AuditLogManager#batchSuccess
CASSANDRA-20816	Optimize MessagingService.getVersionOrdinal
CASSANDRA-20804	Optimize DataPlacement lookup by ReplicationParams
CASSANDRA-20760	Optimize TrieMemtable#getFlushSet
CASSANDRA-20465	Reduce runtime overhead of org.apache.cassandra.schema.TableMetadataRef#get usage
CASSANDRA-21141	Reduce memory allocation during transformation of BatchStatement to Mutation
CASSANDRA-21088	Minor perf optimizations around memtable put logic
CASSANDRA-20250	Optimize Counter, Meter and Histogram metrics using thread local counters
CASSANDRA-20173	Avoid new ByteBuffer allocation for each NativeCell/NativeClustering during flushing of offheap_objects memtable
CASSANDRA-20167	Reduce memory allocations in miscellaneous places along write path
CASSANDRA-20166	Avoid ByteBuffer allocation during decoding of prepared CQL write requests
CASSANDRA-20162	Avoid memory allocation in NativeCell.valueSize() and NativeClustering.dataSize()
CASSANDRA-18831	JDK21 support

Results – individual writes

```
5.0.8        : 594,643 row/s
6.0-latest   : 877,457 row/s
```

Throughput: x1.48 higher (better)

Results – 10 row batches

```
5.0.8        :  78,804 op/s      788,037 row/s
6.0-latest   : 273,312 op/s    2,733,117 row/s
```

Throughput: x3.4 higher (better)

Story #1
Allocator and many threads

Profiling

- I have a set of tests within an increased load
- From some point – the load is not increasing

Profiling

- I have a set of tests within an increased load
- From some point – the load is not increasing
- CPU is not fully utililized, disk is not a bottleneck
- Where is the bottleneck?

Profiling

- Async-profiler
  - cpu/wall/allocations/etc modes
  - Capture results in JFR file (several options at the same time)
  - Flamegraph / heatmap views

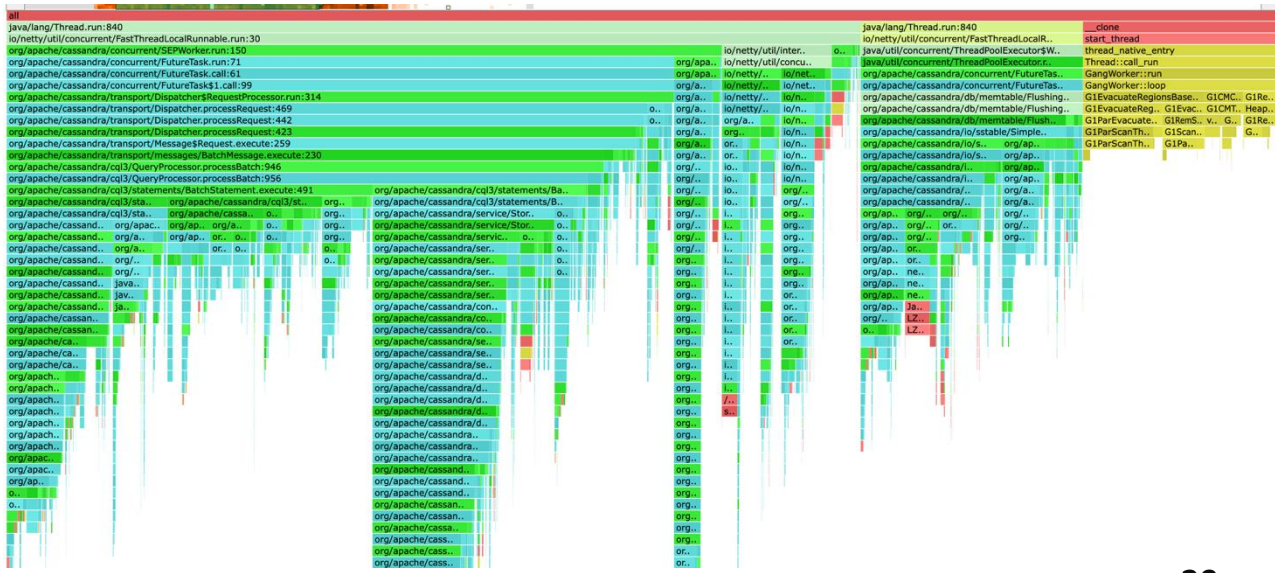
https://www.youtube.com/watch?v=u7-S-Hn-7Do

Profiling

[CASSANDRA-20854] Support low-overhead async profiling via JMX/Nodetool with the async-profiler library

```
<dependency>
    <groupId>tools.profiler</groupId>
    <artifactId>async-profiler</artifactId>
</dependency>
```

```
import one.profiler.AsyncProfiler;
AsyncProfiler asyncProfiler = AsyncProfiler.getInstance();
asyncProfiler.execute(command);
```

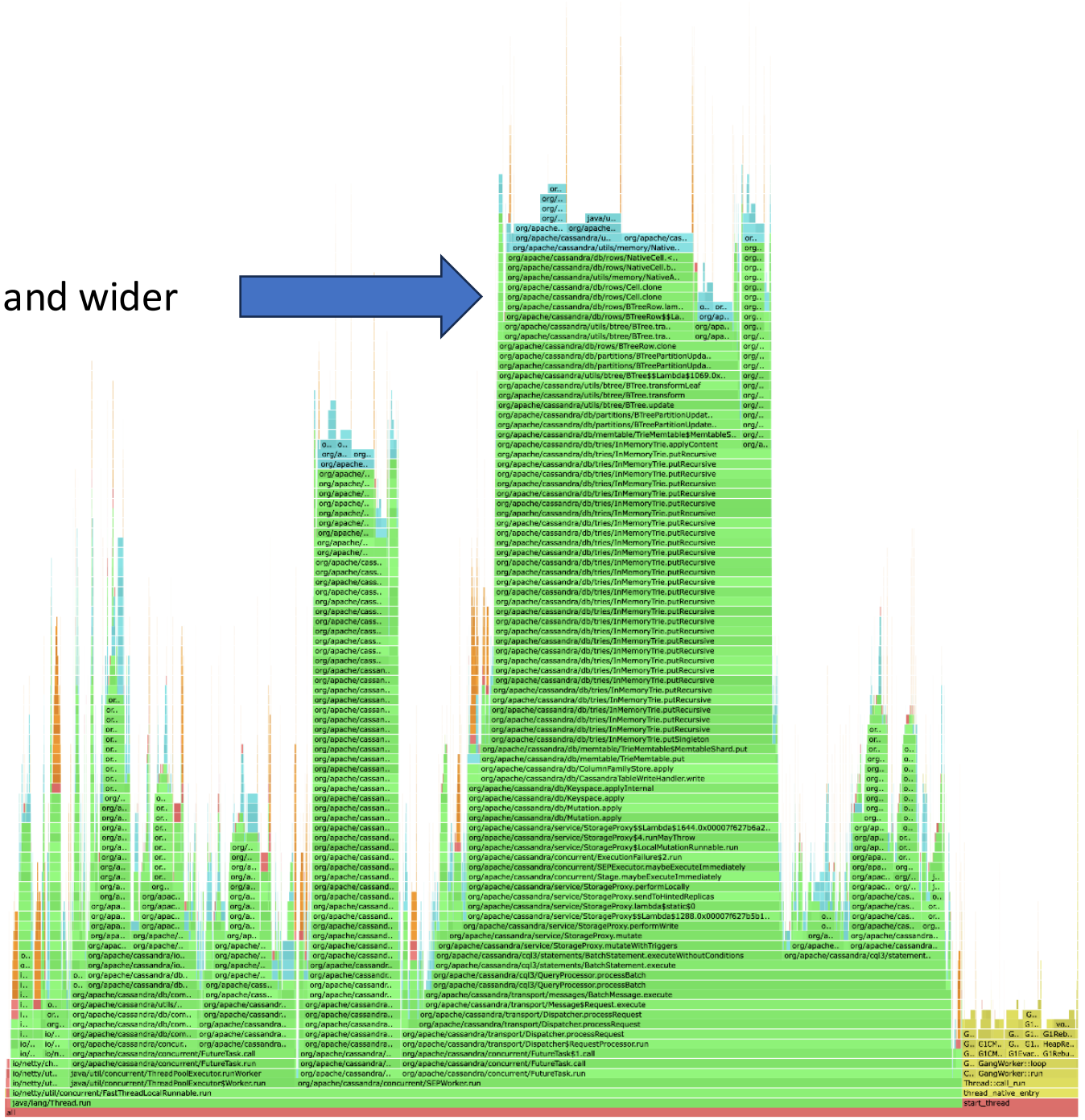


Profiling

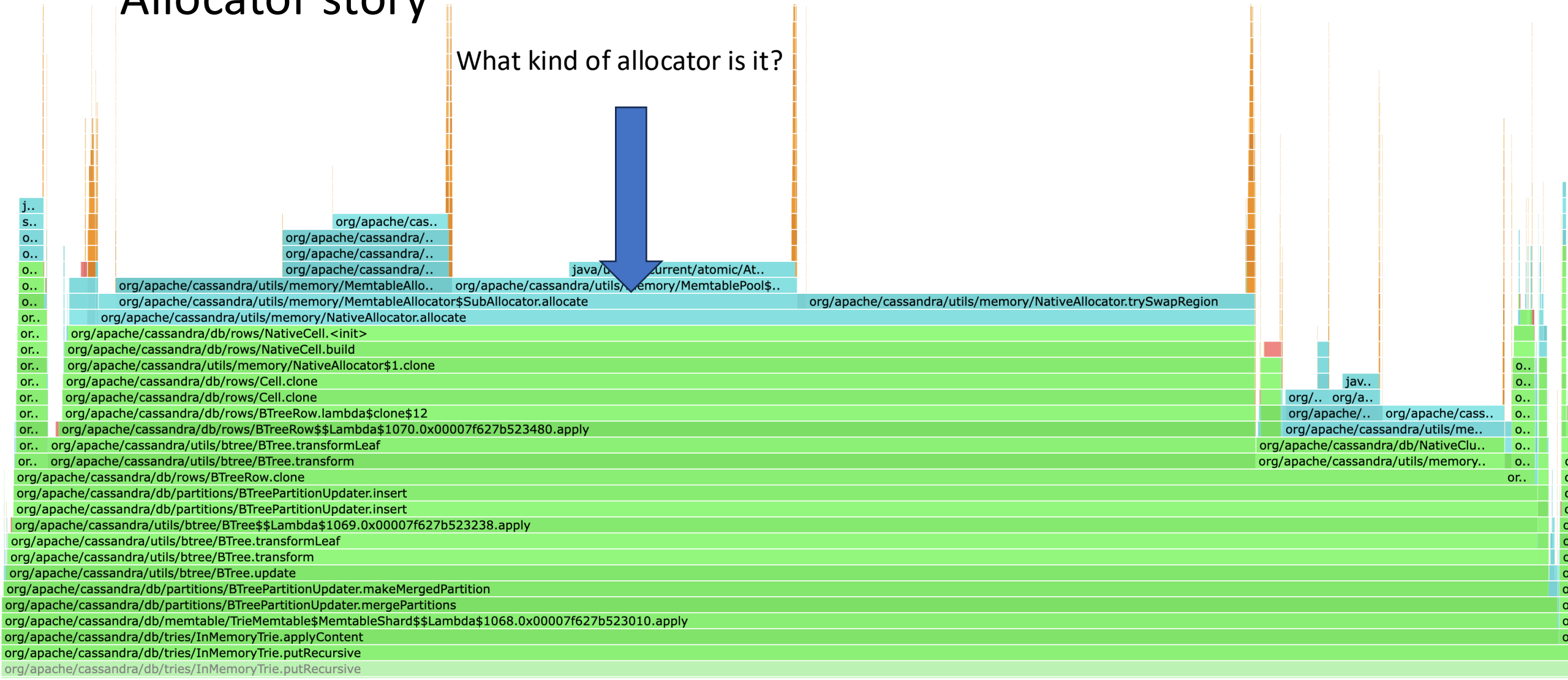
```
#!/bin/bash

PROFILE_DURATION_SEC_DEFAULT=1200
PROFILE_NAME_DEFAULT=cass

PROFILE_DURATION_SEC=${2:-$PROFILE_DURATION_SEC_DEFAULT}
PROFILE_NAME=${1:-$PROFILE_NAME_DEFAULT}
CASSANDRA_PID=$(ps uax | grep CassandraDaemon | grep -v grep | awk '{print $2}')

echo "run profiling for PID=$CASSANDRA_PID, duration: $PROFILE_DURATION_SEC sec, name: $PROFILE_NAME"

./bin/asprof -d $PROFILE_DURATION_SEC -i 5ms -f ${PROFILE_NAME}.jfr -e cpu,wall $CASSANDRA_PID
./bin/jfrconv --cpu -o heatmap $PROFILE_NAME.jfr  ${PROFILE_NAME}_cpu.html
./bin/jfrconv --wall -o heatmap $PROFILE_NAME.jfr  ${PROFILE_NAME}_wall.html
```

Allocator story

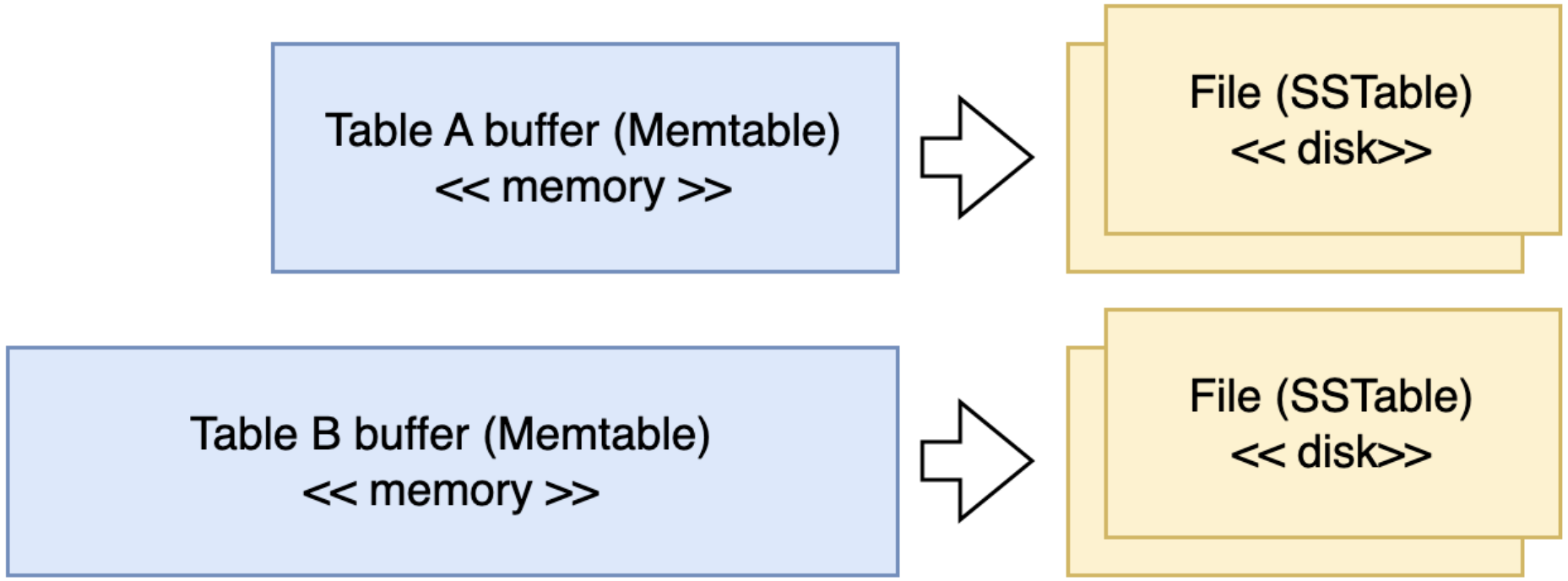
More load – this area is getting wider and wider
And no throughput improvement



Allocator story

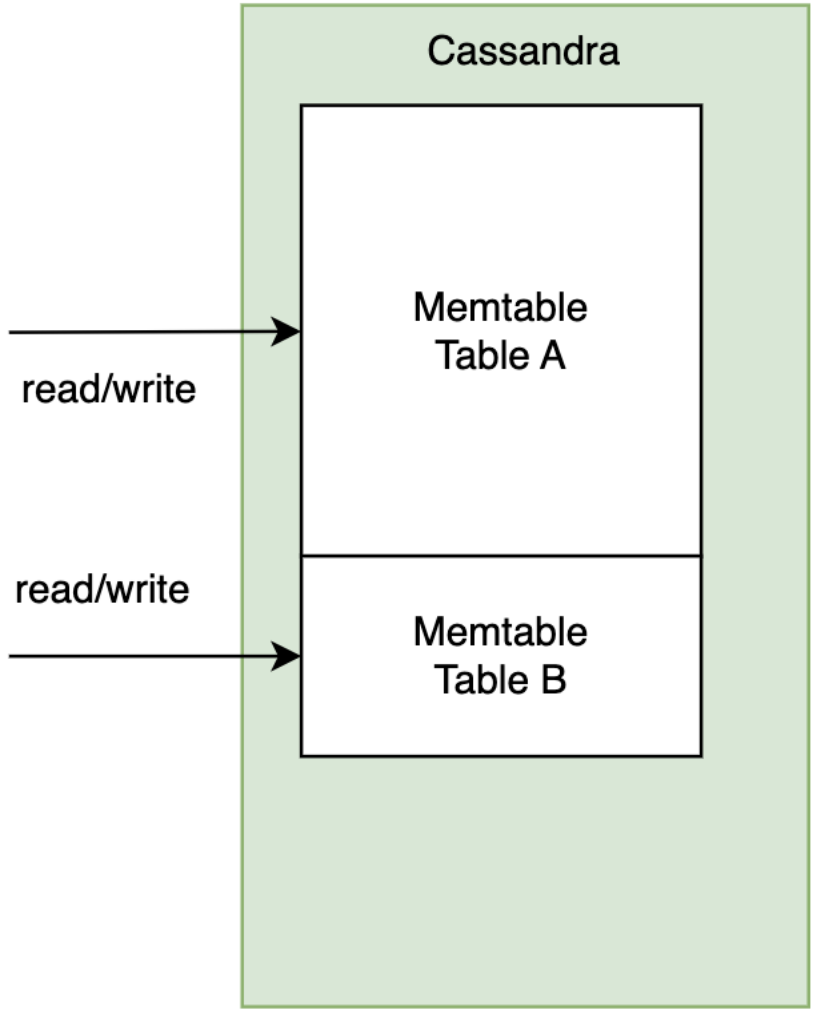


Memtable lifecycle

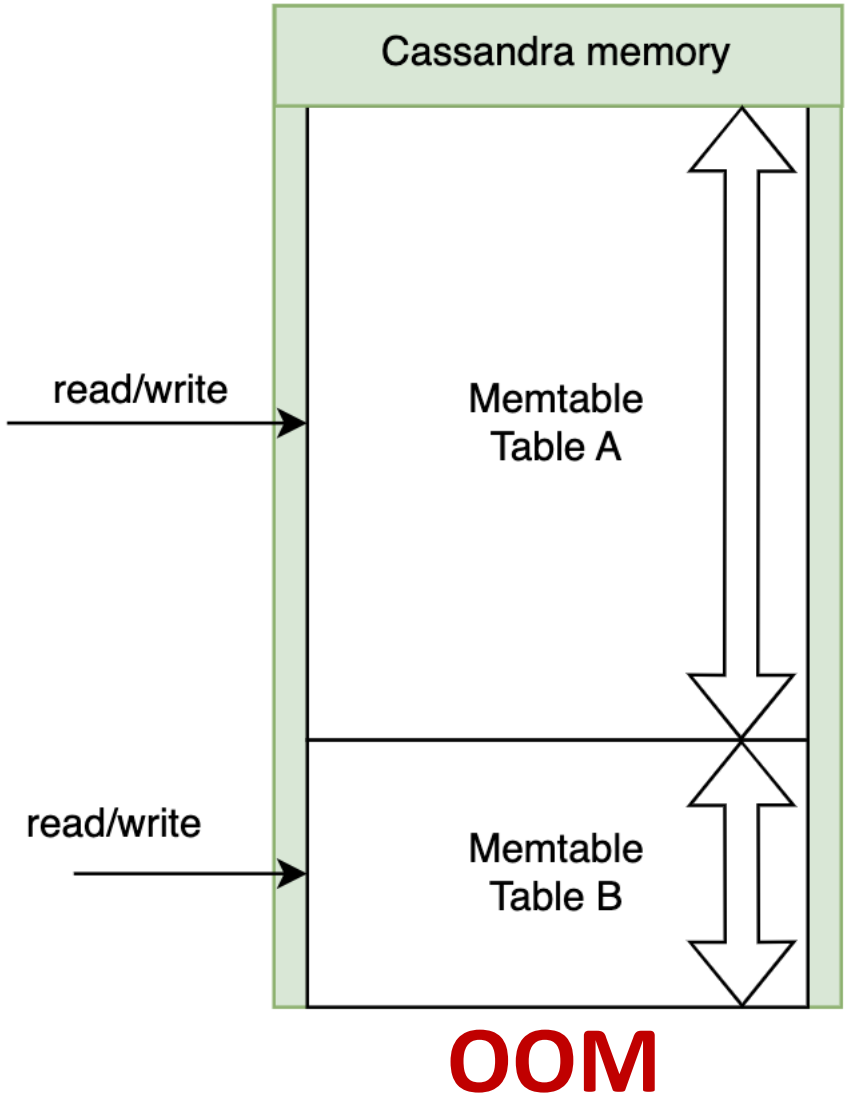


We write data to memory buffers and then flush it to disk

Memtable lifecycle



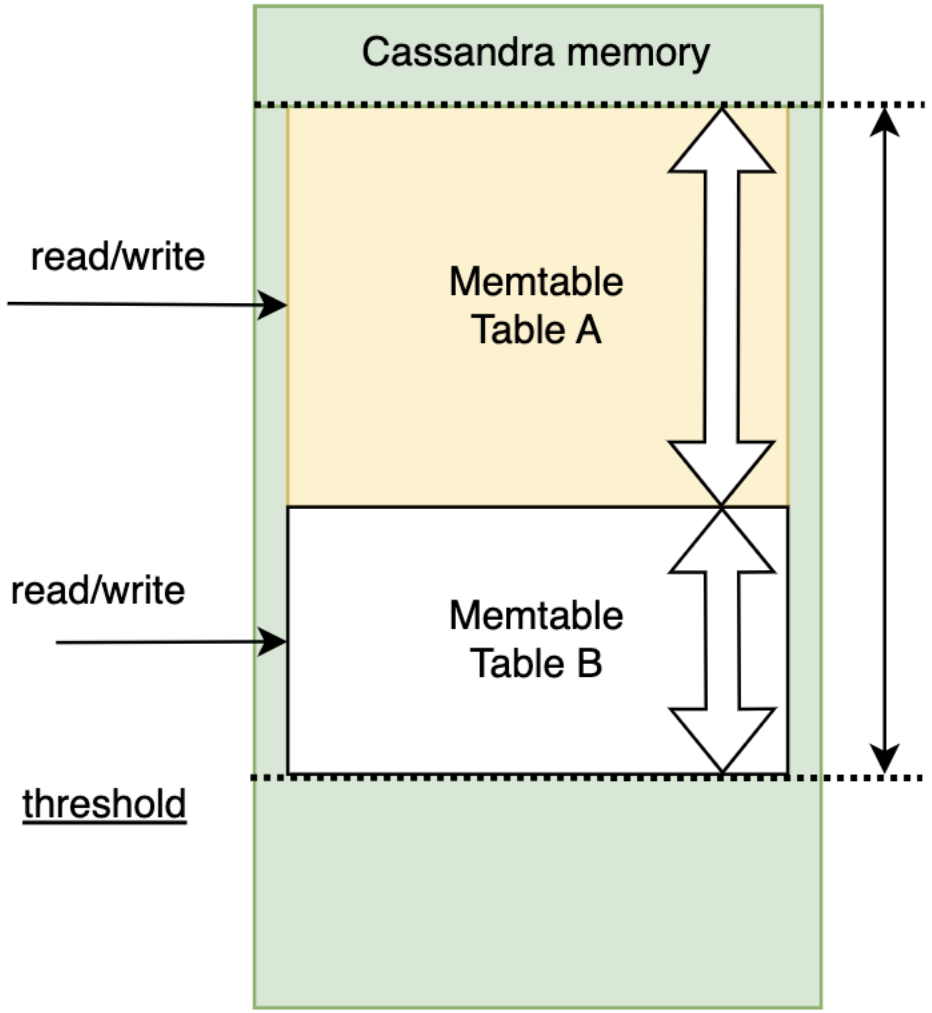
Memtable lifecycle



Memtable lifecycle

- Define a memory threshold
- When total used > threshold

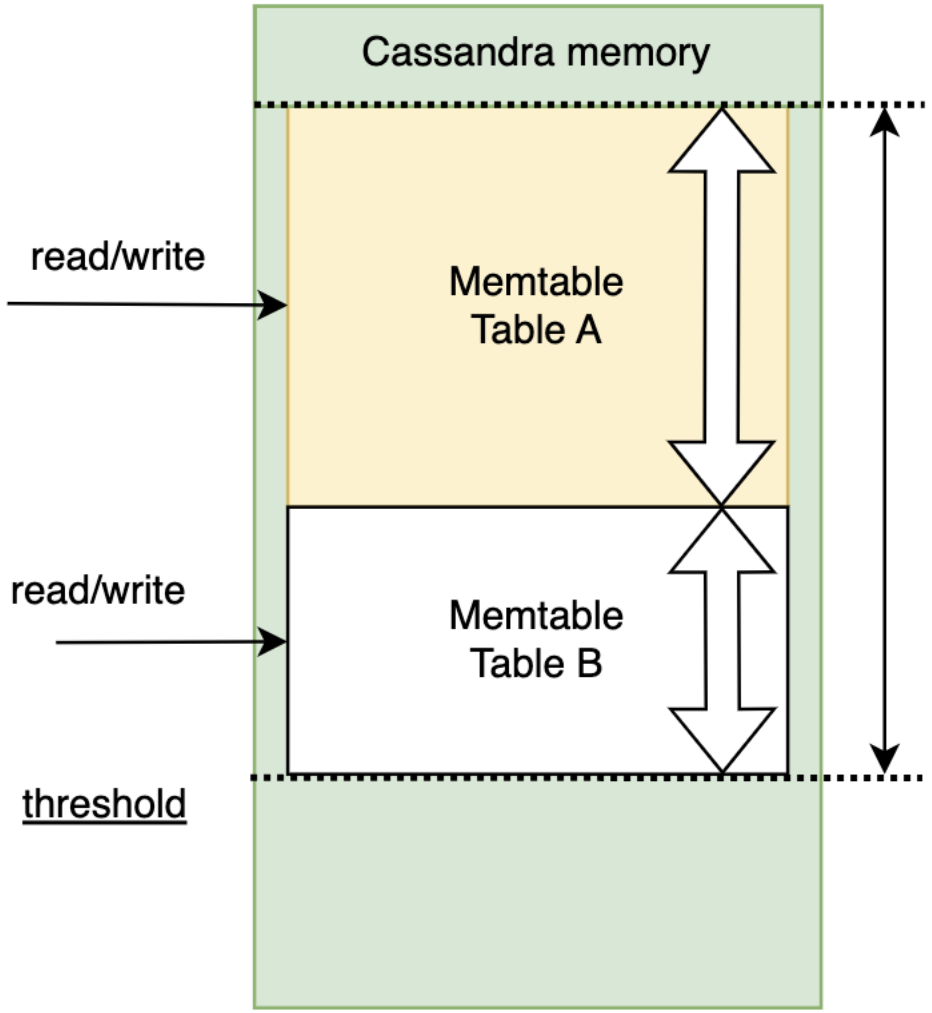
  flush the largest memtable to disk



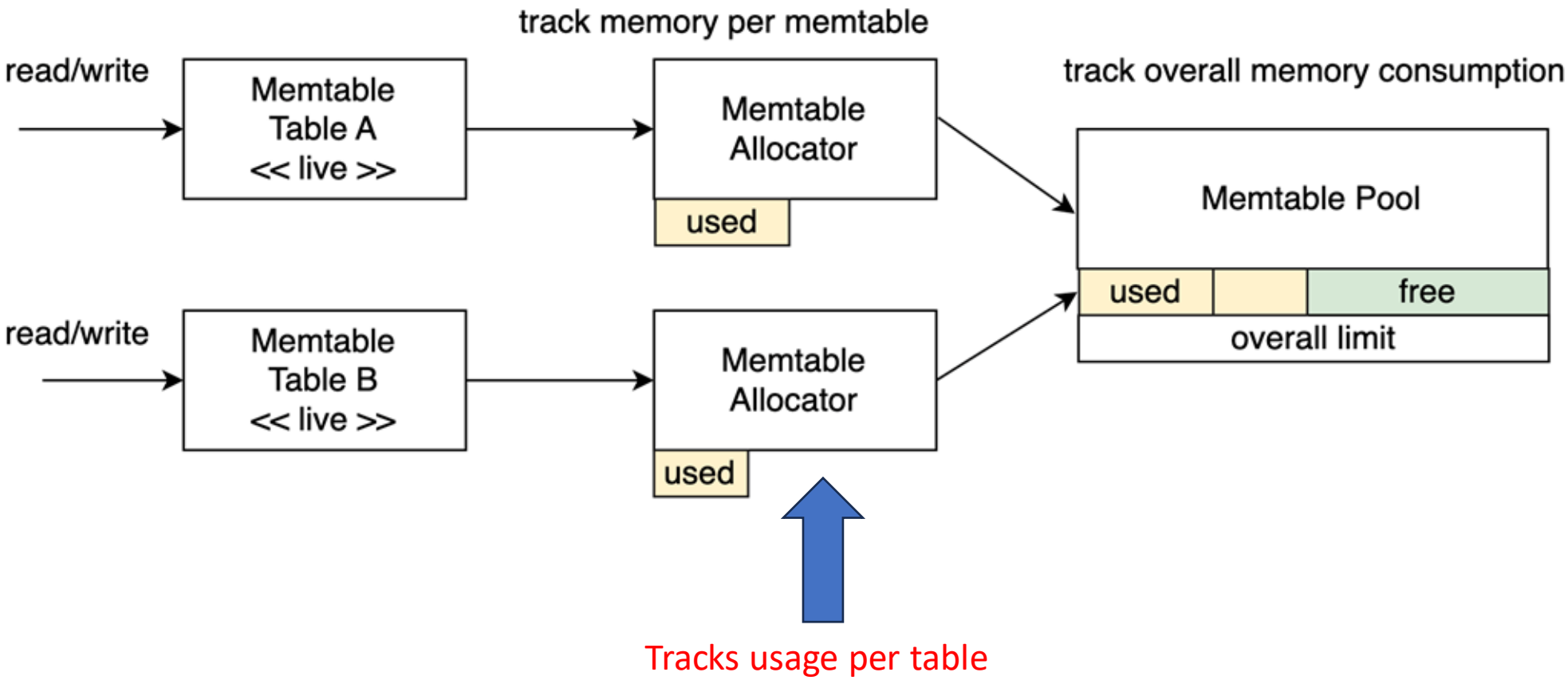
Memtable lifecycle

- Define a memory threshold
- When **total used** > threshold

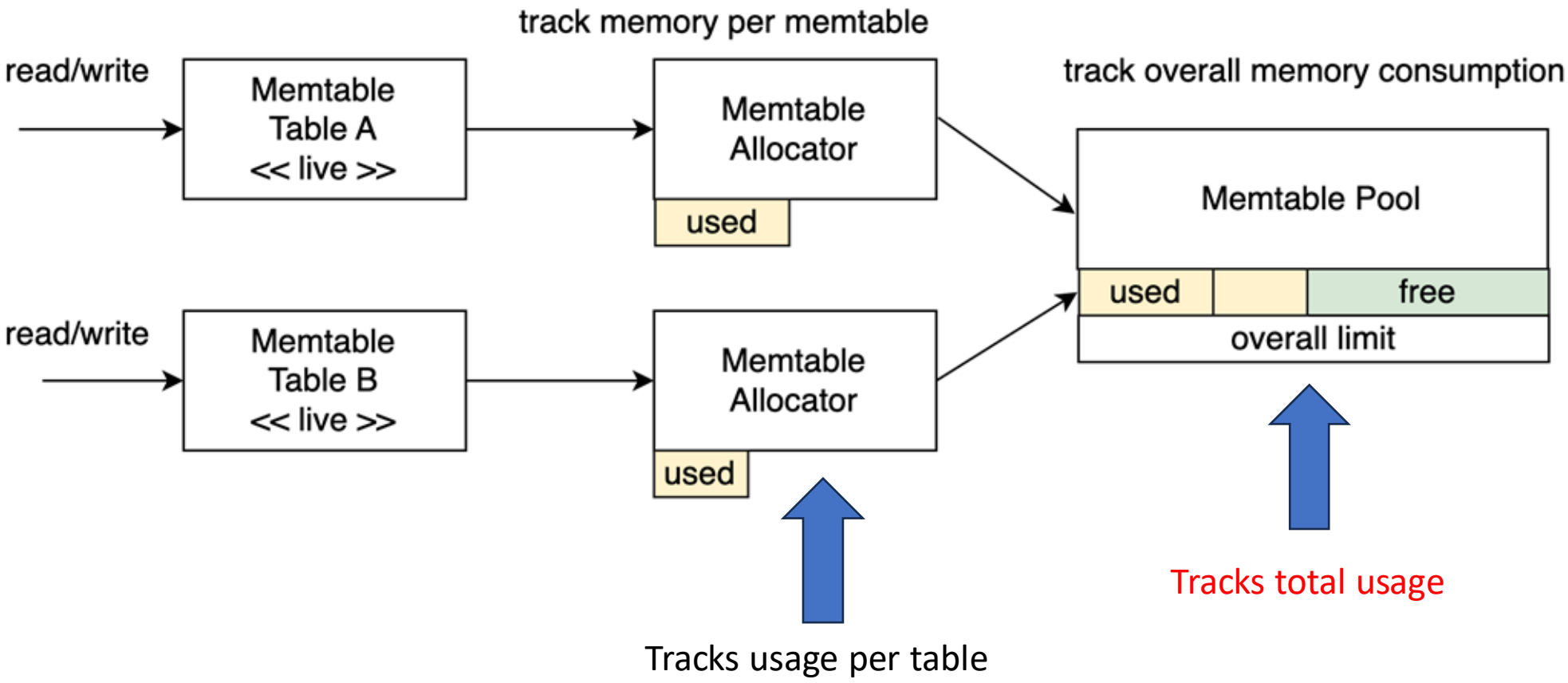
  flush the **largest** memtable to disk
- **So, we need to track memory usage**
  1. per memtable – to select largest
  2. total – to check threshold



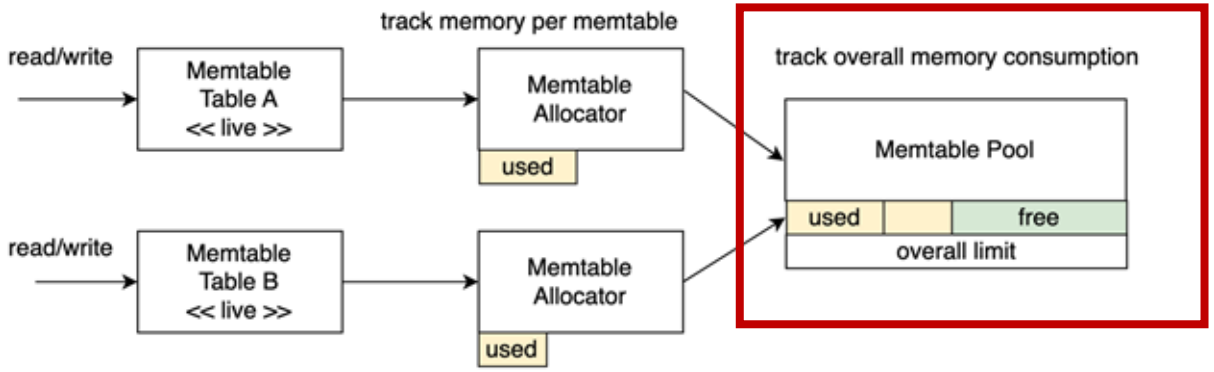
Memtable lifecycle



Memtable lifecycle



Total memtable consumption tracking




```
AtomicLong allocated = new AtomicLong();

boolean tryAllocate(long size) {
   while (true) {
      long cur = allocated.get();
      long next = cur + size;
      if (next > limit)
         return false;
      if (allocated.compareAndSet(cur, next))
         return true;
   }
}
```

Total memtable consumption tracking

```
AtomicLong allocated = new AtomicLong();

boolean tryAllocate(long size) {
   while (true) {
      long cur = allocated.get();
      long next = cur + size;
      if (next > limit)
         return false;
      if (allocated.compareAndSet(cur, next))
         return true;
   }
}
```

<==== CAS (Compare And Set) loop

<== contention

Total memtable consumption tracking

Reduce CAS loop with AtomicLong.addAndGet to reduce contention

```
AtomicLong allocated = new AtomicLong();

boolean tryAllocate(long size) {
   while (true) {
      long cur = allocated.get();
      long next = cur + size;
      if (next > limit)
         return false;
      if (allocated.compareAndSet(cur, next))
         return true;
   }
}
```

```
AtomicLong allocated = new AtomicLong();

boolean tryAllocate(long size) {
   long next = allocated.addAndGet(size);
   if (next > limit) {
      // a long comment with an analysis
      // why the related logic is ok with it
      allocated.addAndGet(-size);
      return false;
   }
   return true;
}
```

No loop, constant number of operations
BUT semantically different

Total memtable consumption tracking

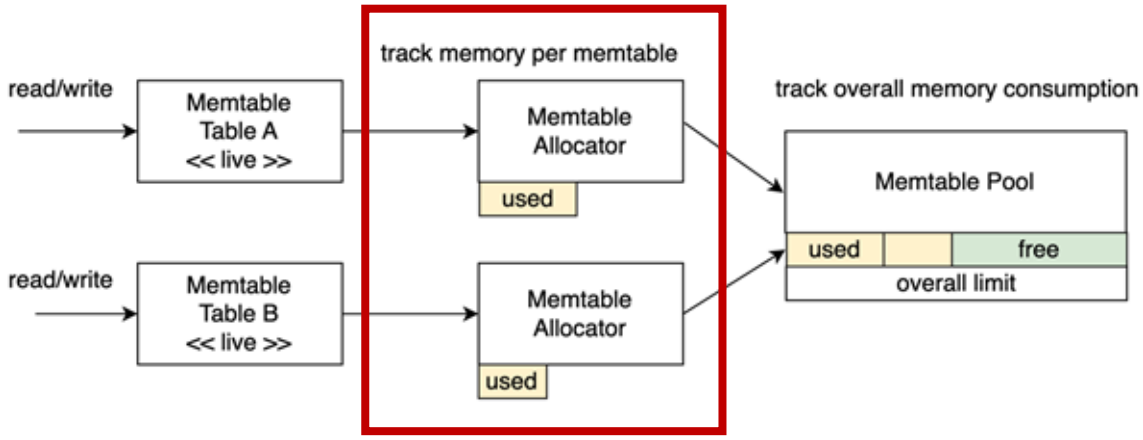
```
// We have switched from CAS loop and a strict limit check
// to addAndGet with a possible post-correction for perf reasons.
// Why this is OK:
// - We may temporarily exceed the limit here, but that also happens in case of blocking op order.
// - We decrease the allocated value, but that was also possible as part of the adjustment logic.
//
// We don't call released() here because it triggers hasRoom.signalAll(), which would
//   immediately wake up the current thread before memory is reclaimed and cause a busy loop.
// In a rare case, an unsuccessful attempt of a larger allocation near a limit by one thread
// may temporarily block progress on a smaller concurrent allocation by another thread,
// but both threads will be signalled and be able to proceed once memory is reclaimed.
```

```
AtomicLong allocated = new AtomicLong();

boolean tryAllocate(long size) {
    long next = allocated.addAndGet(size);
    if (next > limit) {
        // a long comment with an analysis
        // why the related logic is ok with it
        allocated.addAndGet(-size);
        return false;
    }
    return true;
}
```

No loop, constant number of operations

BUT semantically different

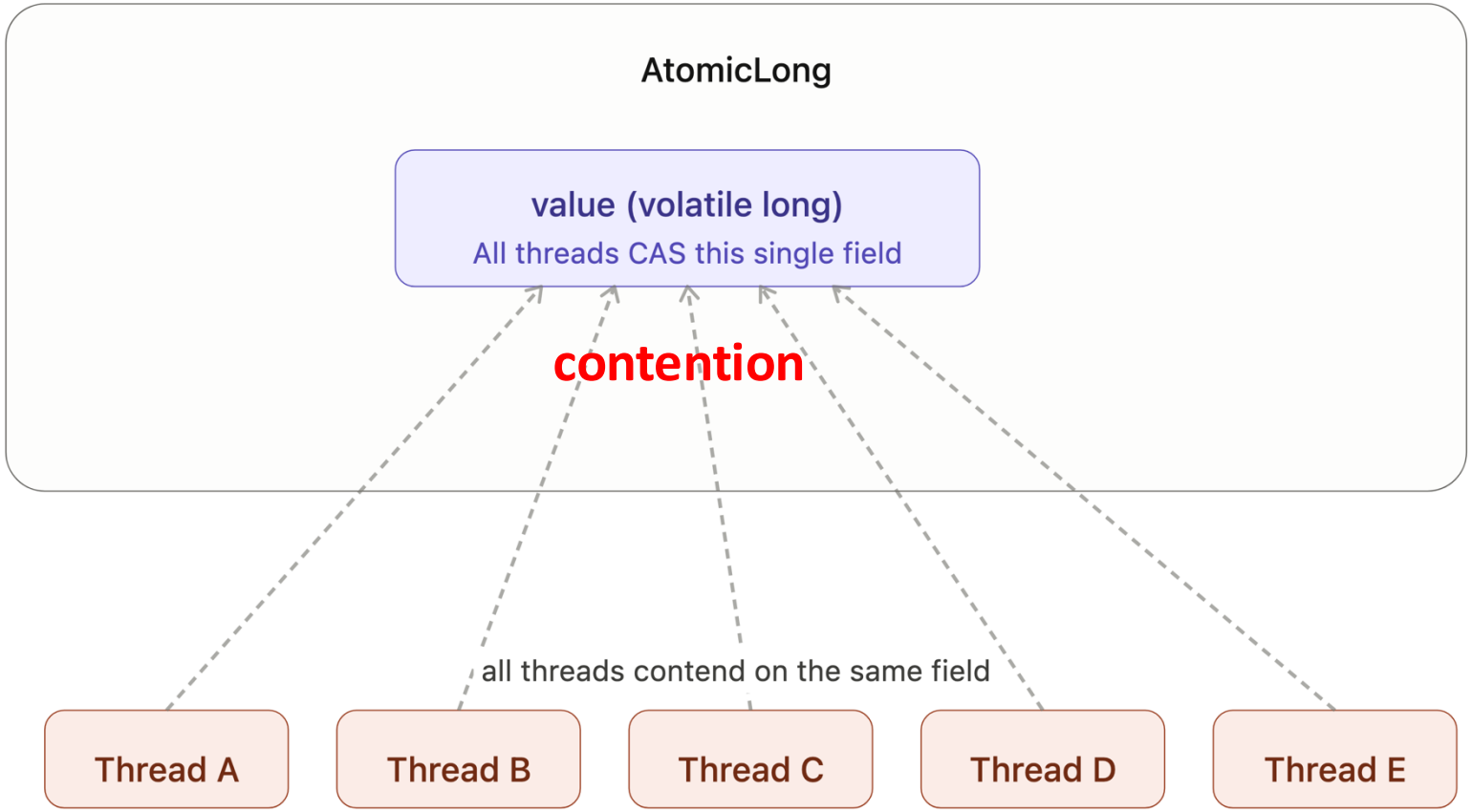
Per table usage tracking



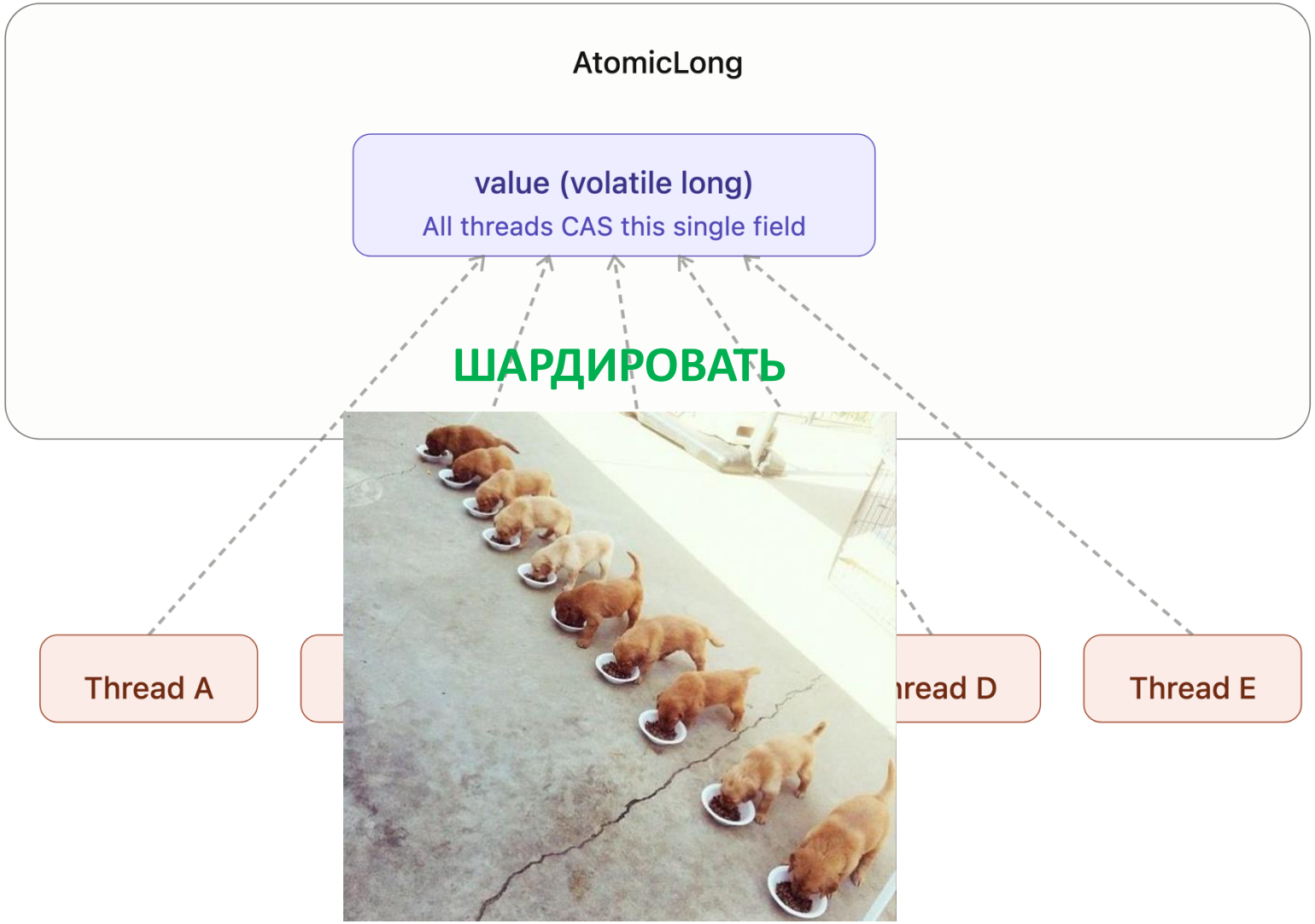

```
AtomicLong ownsByTable = new AtomicLong();

void allocated(int size)
{
   ownsByTable.addAndGet(size);
}
// used to decide what table to flush
public long owns()
{
   return ownsByTable.get();
}
```

<== contention by different threads

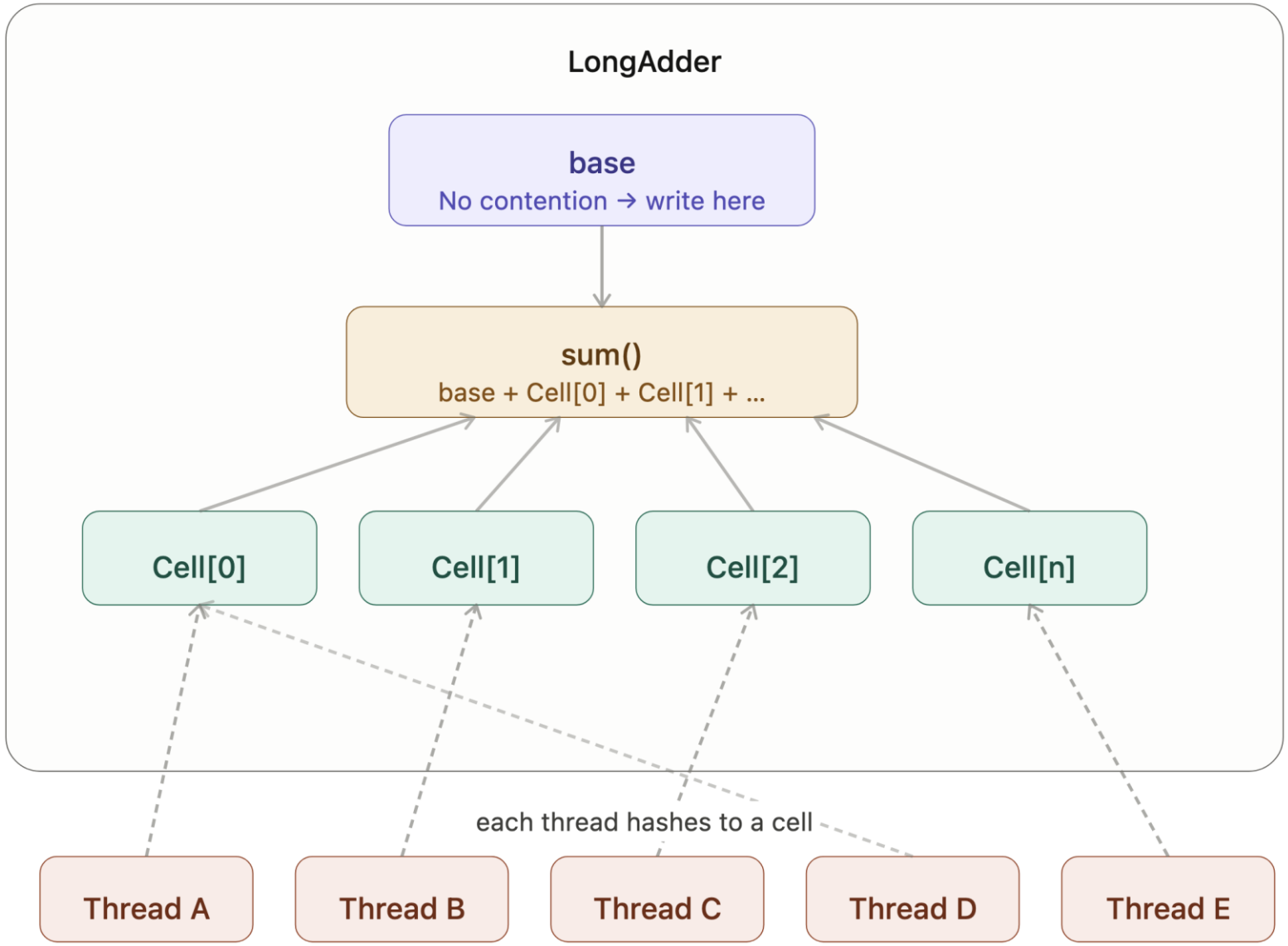
Per table usage tracking



Per table usage tracking



Per table usage tracking



Per table usage tracking

AtomicLong => LongAdder replacement

```
AtomicLong ownsByTable = new AtomicLong();

void allocated(int size) // hot
{
   ownsByTable.addAndGet(size);
}

public long owns() // cold
{
   return ownsByTable.get();
}
```

<== contention

```
LongAdder ownsByTable = new LongAdder();

void allocated(int size)
{
   ownsByTable.add(size);
}

public long owns()
{
   return ownsByTable.sum();
}
```

<== sharded write

Per table usage tracking

AtomicLong => LongAdder replacement

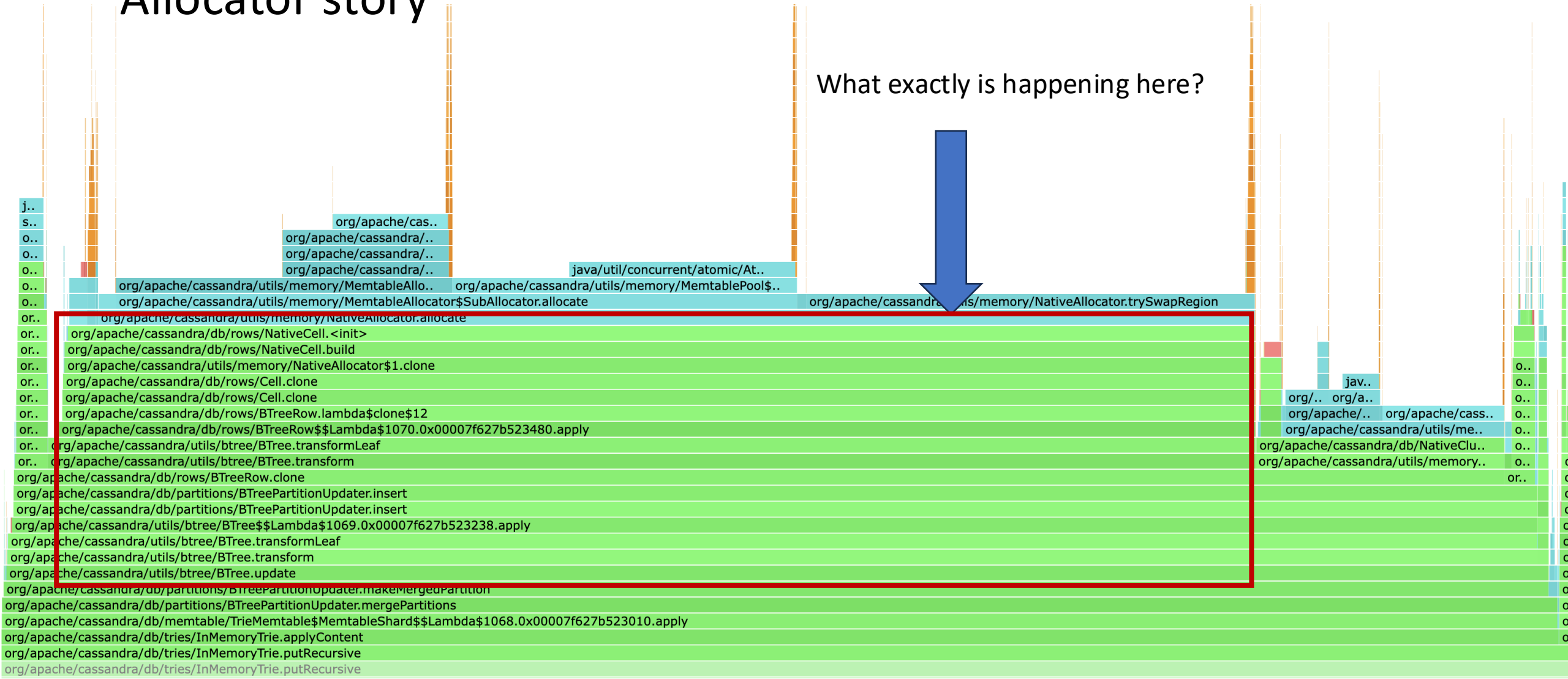
```
AtomicLong ownsByTable = new AtomicLong();

void allocated(int size)
{
   ownsByTable.addAndGet(size);
}

public long owns()
{
   return ownsByTable.get();
}
```

```
LongAdder ownsByTable = new LongAdder();

void allocated(int size)
{
   ownsByTable.add(size);
}

public long owns()
{
   return ownsByTable.sum();
}
```

We can do it because this method is not used in a hot path

Allocator story

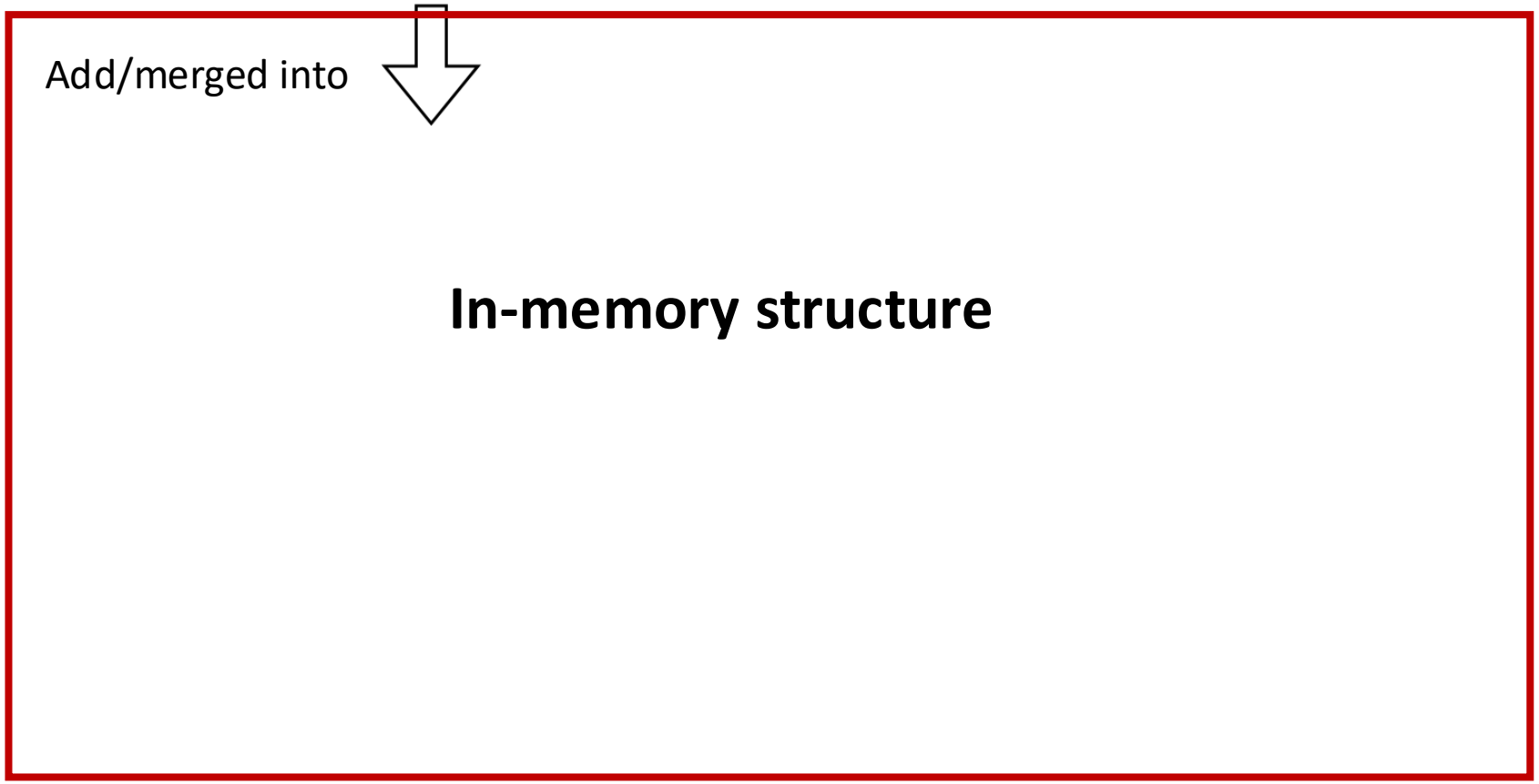
What exactly is happening here?



Allocator story

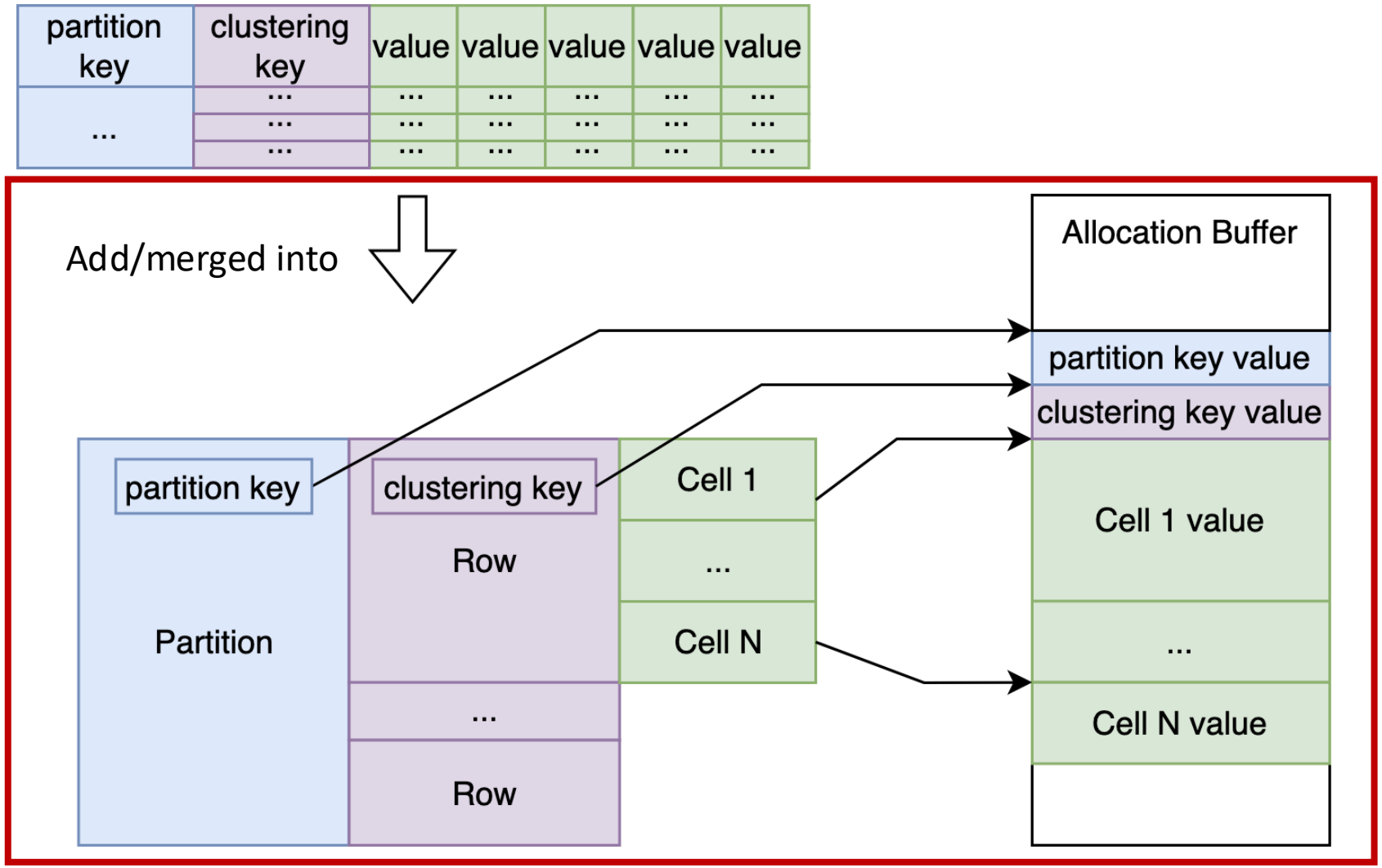
From a network request

partition key	clustering key	value	value	value	value	value
...	...	...	...	...	...	...
	...	...	...	...	...	...
	...	...	...	...	...	...

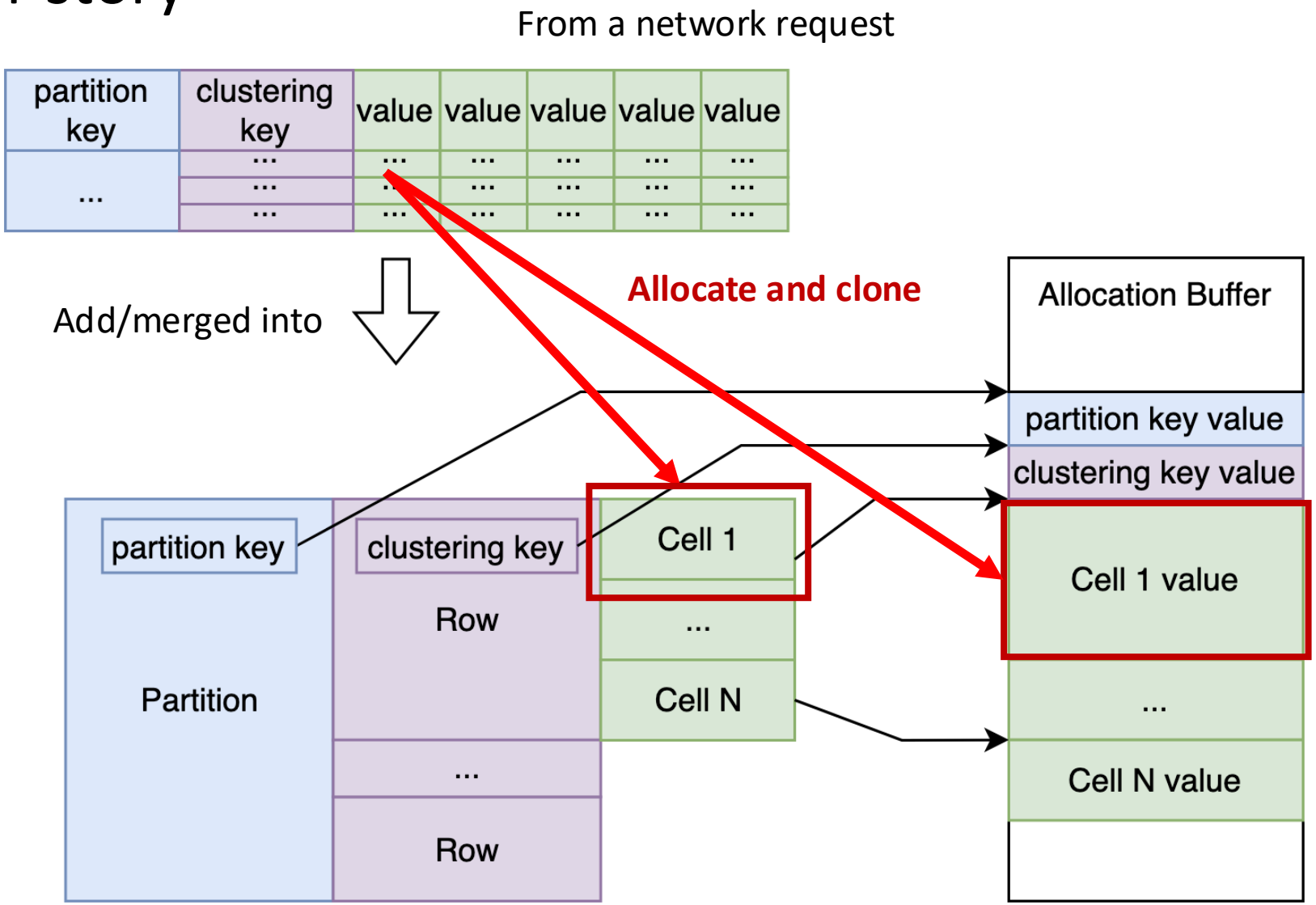


Allocator story

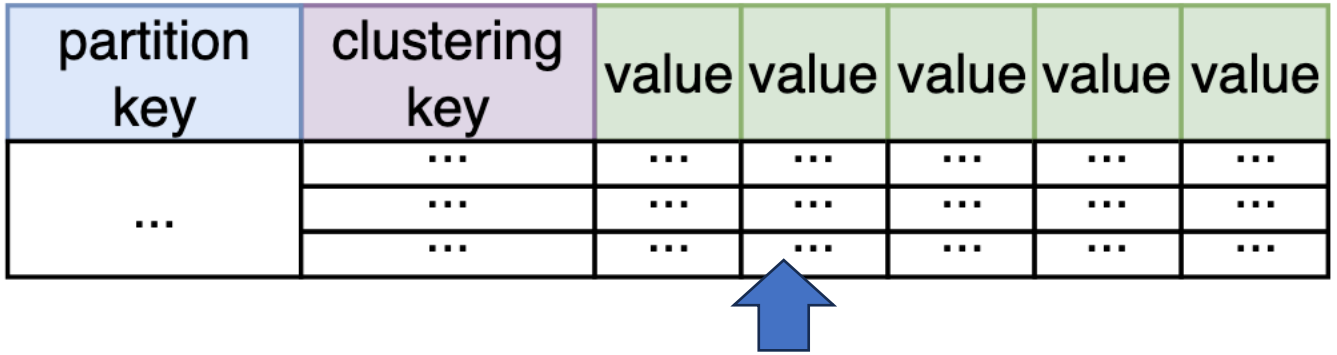
From a network request



Allocator story



Allocation batching




```
for each cell {
  allocateCellFrom(commonAllocator);
}
```

Allocation batching

partition key	clustering key	value	value	value	value	value
...	...	...	...	...	...	...
	...	...	...	...	...	...
	...	...	...	...	...	...

It is a cell

```
for each cell {
  allocateCellFrom(commonAllocator);
}
```

10 rows in a batch
1 clustering kes + 5 value columns = 6 items per row
6 x 10 rows = 60 times per batch request

800k r/sec x 60 = 48 mln allocation requests/sec

Allocation batching

```
for each cell {
  allocateCellFrom(commonAllocator);
}
```

It is not precise

```
int allocationSize = 0;
for each cell {
 allocationSize += cell.estimateAllocationSize();
}

requestLocalAllocator = allocateFrom(
                          commonAllocator,
                          allocationSize
                        );

for each cell {
  allocateCell(requestLocalAllocator);
}
```

Allocation batching

```
for each cell {
  allocateCellFrom(commonAllocator);
}
```

800k r/sec x 60 = 48 mln allocation requests/sec

```
int allocationSize = 0;
for each cell {
 allocationSize += cell.estimateAllocationSize();
}

requestLocalAllocator = allocateFrom(
                          commonAllocator,
                          allocationSize
                        );

for each cell {
  allocateCell(requestLocalAllocator);
}
```

800k r/sec x 1 = 800k allocation requests/sec

Allocation batching

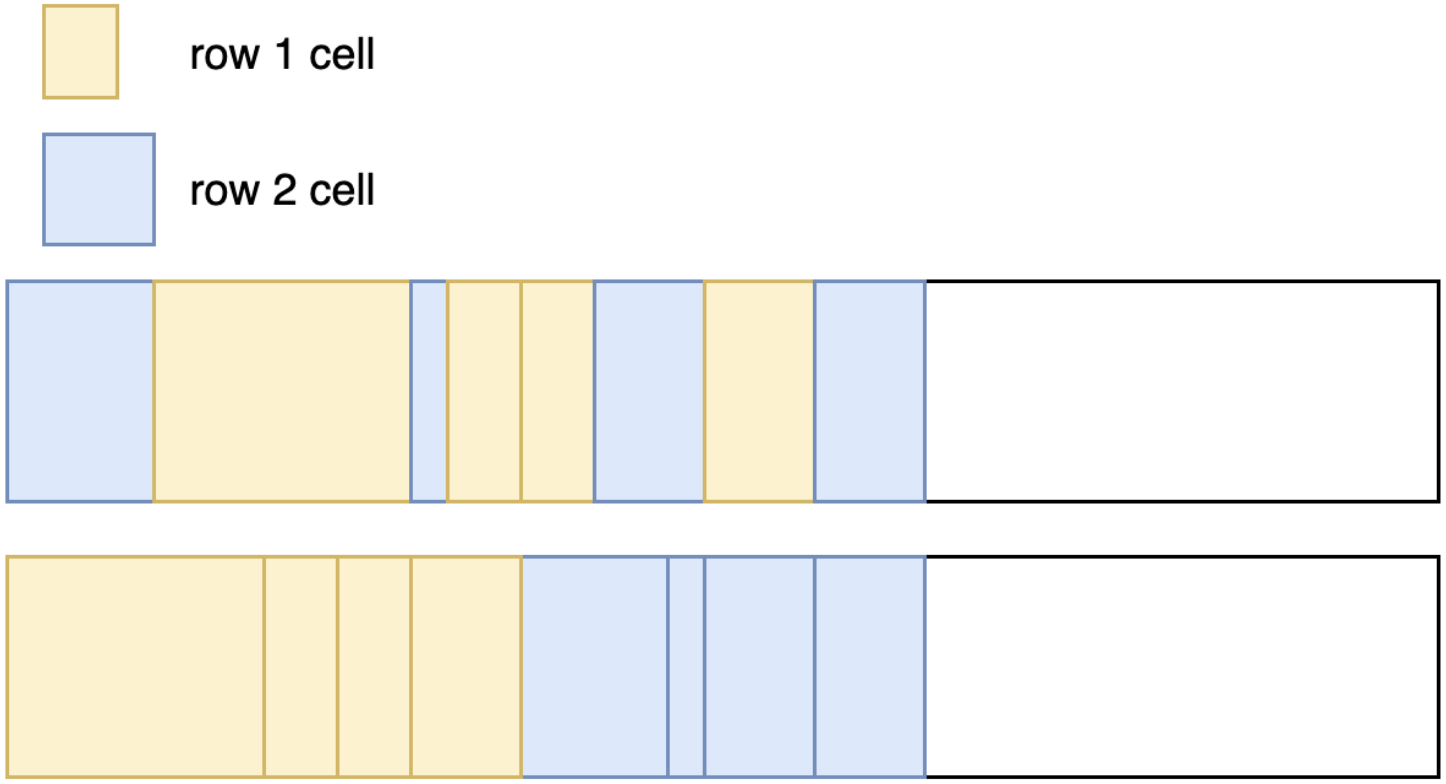
```
for each cell {
  allocateCellFrom(commonAllocator);
}
```

800k r/sec x 60 = 48 mln allocation requests/sec

```
int allocationSize = 0;
for each cell {
 allocationSize += cell.estimateAllocationSize();
}

requestLocalAllocator = allocateFrom(
                          commonAllocator,
                          allocationSize
                        );

for each cell {
  allocateCell(requestLocalAllocator);
}
```

800k r/sec x 1 = 800k allocation requests/sec

Allocation batching



Extra bonus: memory defragmentation

Reduce contention in MemtableAllocator.allocate

- Test results

op rate	: 78,804 op/s.	: 159,068 op/s
row rate	: 788,037 row/s.	: 1,590,681 row/s
latency mean	: 1.3 ms	: 0.9 ms
latency median	: 1.0 ms	: 0.8 ms
latency 95th percentile	: 2.3 ms	: 1.3 ms
latency 99th percentile	: 4.3 ms	: 1.9 ms
latency 99.9th percentile	: 21.5 ms	: 14.5 ms

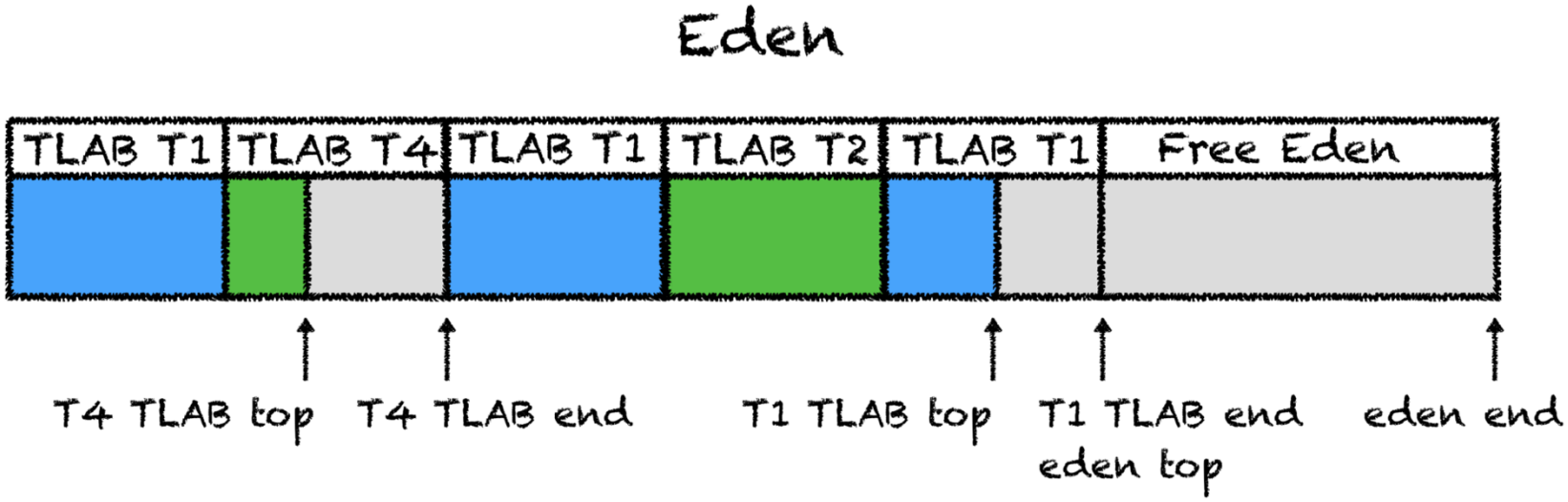
Alternatives

- What is about per-thread allocators?
  Like Thread-Local Allocation Buffers (TLABs) in JVM

Alternatives

- What is about per-thread allocators?
  Like Thread-Local Allocation Buffers (TLABs) in JVM

  https://habr.com/ru/articles/332708/
  https://shipilev.net/jvm/anatomy-quarks/4-tlab-allocation/



Alternatives

- What is about per-thread allocators?
  Like Thread-Local Allocation Buffers (TLABs) in JVM

☹ too many buffers (tables count x number of threads) => fragmentation, have to implement a complex stealing logic

Alternatives

- TLAB-like approach
  - ☹ too many buffers (tables count x number of threads)
- **Serialize to a temporary buffer and then copy once we know the full size**
  - ☹ Extra memory copies

Story summary

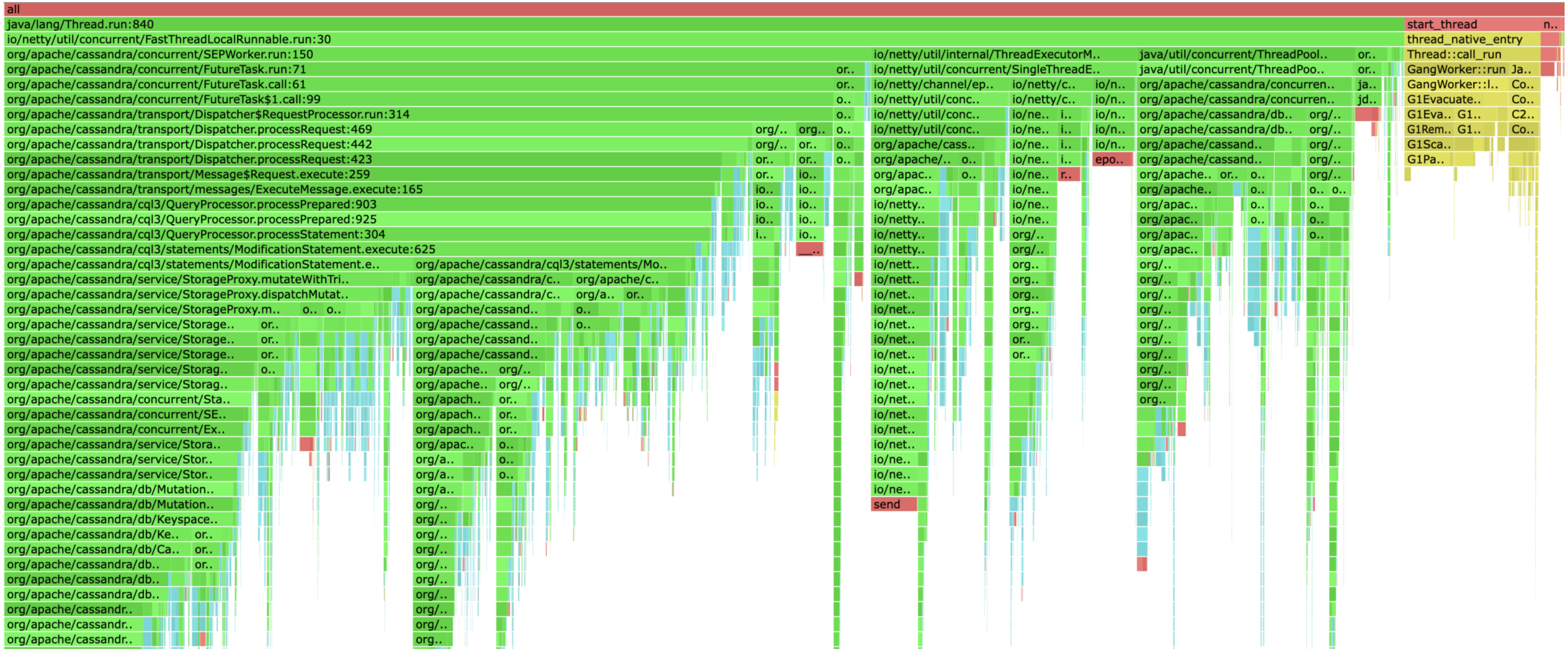
- Do not need a result immediately + high contention: AtomicLong –> LongAdder
- addAndGet can be more efficient that CAS loop

Story summary

- Do not need a result immediately + high contention: AtomicLong –> LongAdder
- addAndGet can be more efficient that CAS loop
- **Batch frequent contended operations to amortize overhead**

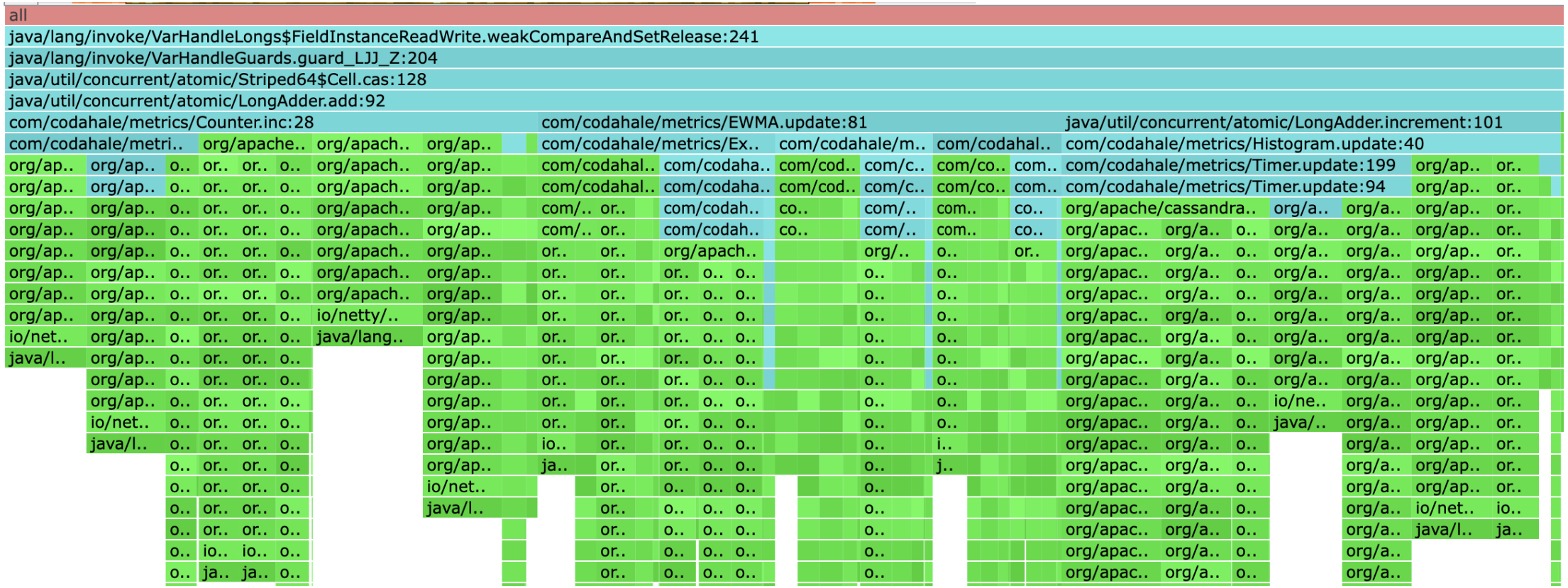

Story #2
Metrics are not free

Sometimes overheads are spread



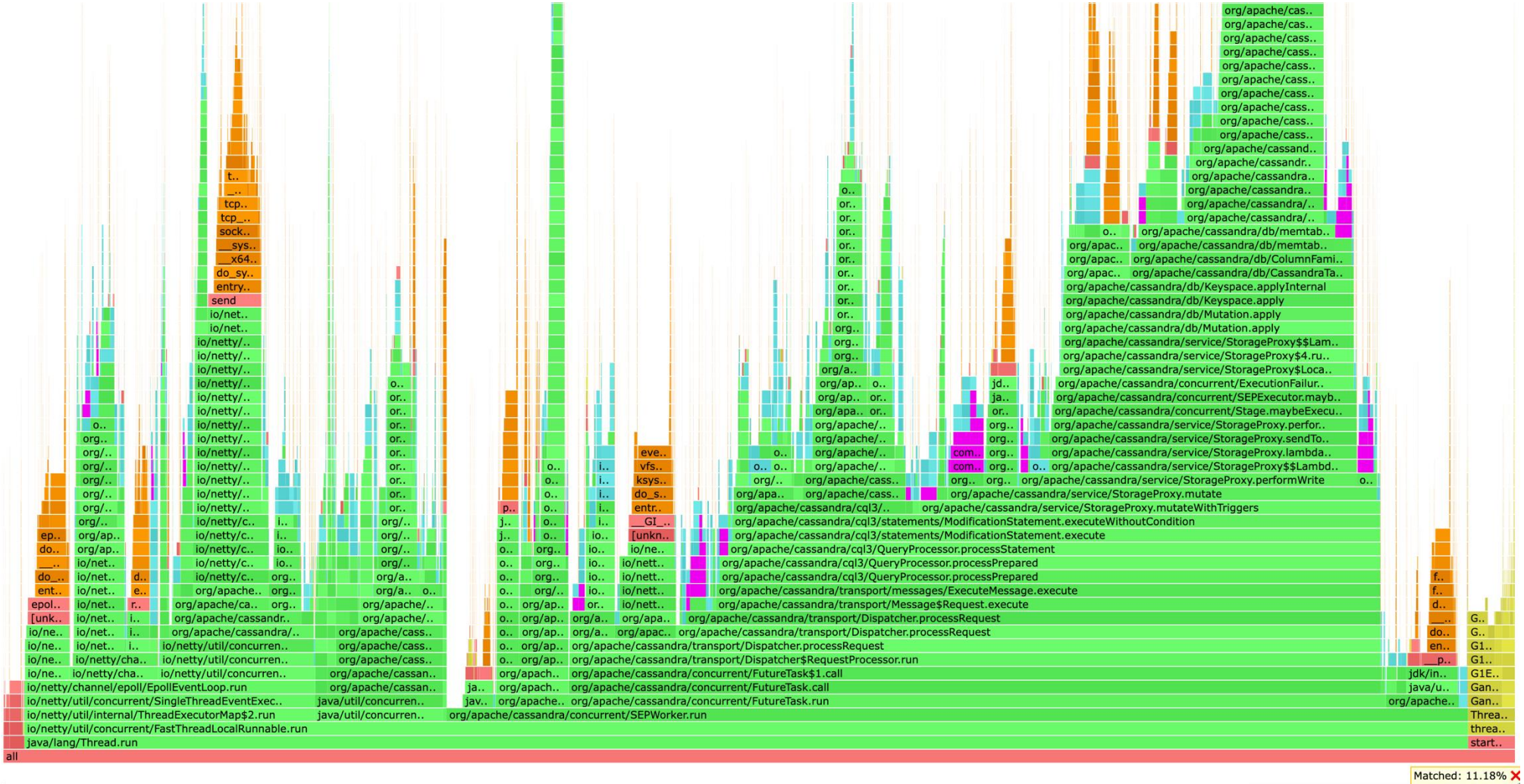
Sometimes overheads are spread

Use reverse view (heatmap HTML format) – “invoked by”



Metrics overhead

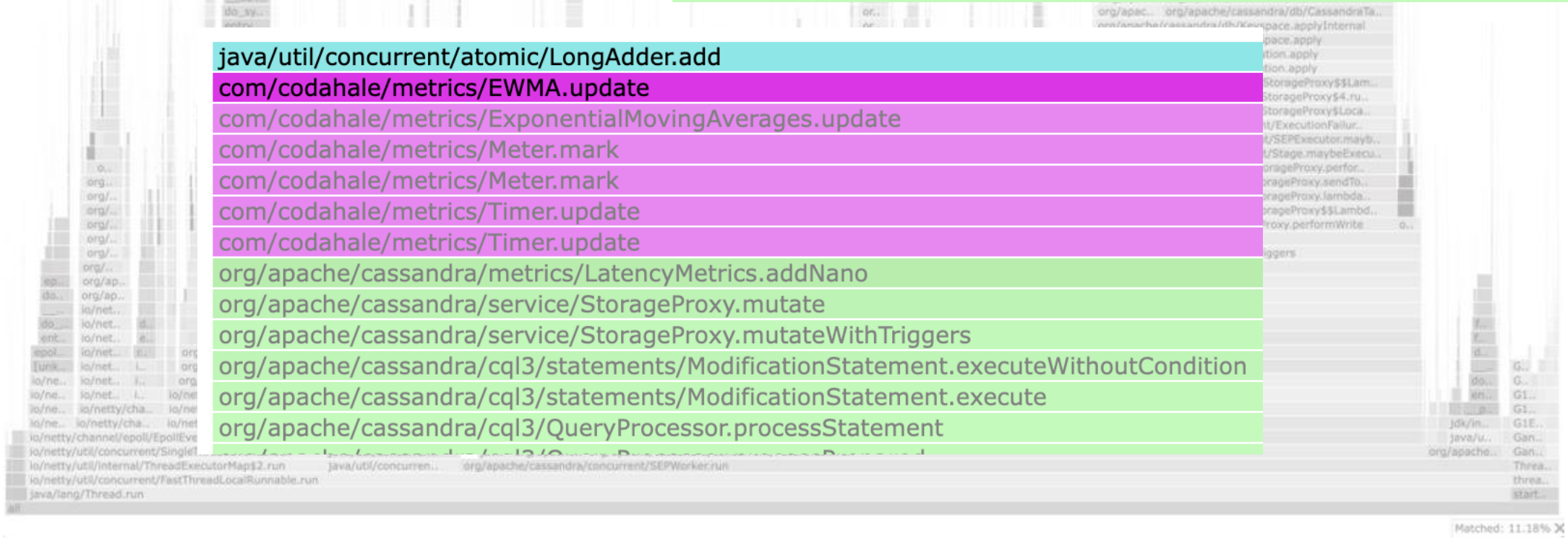
Use search



Metrics overhead

Matched: 11.18% ✗

java/util/concurrent/atomic/LongAdder.add
java/util/concurrent/atomic/LongAdder.increment
com/codahale/metrics/Histogram.update
com/codahale/metrics/Timer.update
com/codahale/metrics/Timer.update
org/apache/cassandra/metrics/LatencyMetrics.addNano
org/apache/cassandra/service/StorageProxy.mutate
org/apache/cassandra/service/StorageProxy.mutateWithTriggers
org/apache/cassandra/cql3/statements/ModificationStatement.executeWithoutCondition
org/apache/cassandra/cql3/statements/ModificationStatement.execute
org/apache/cassandra/cql3/QueryProcessor.processStatement

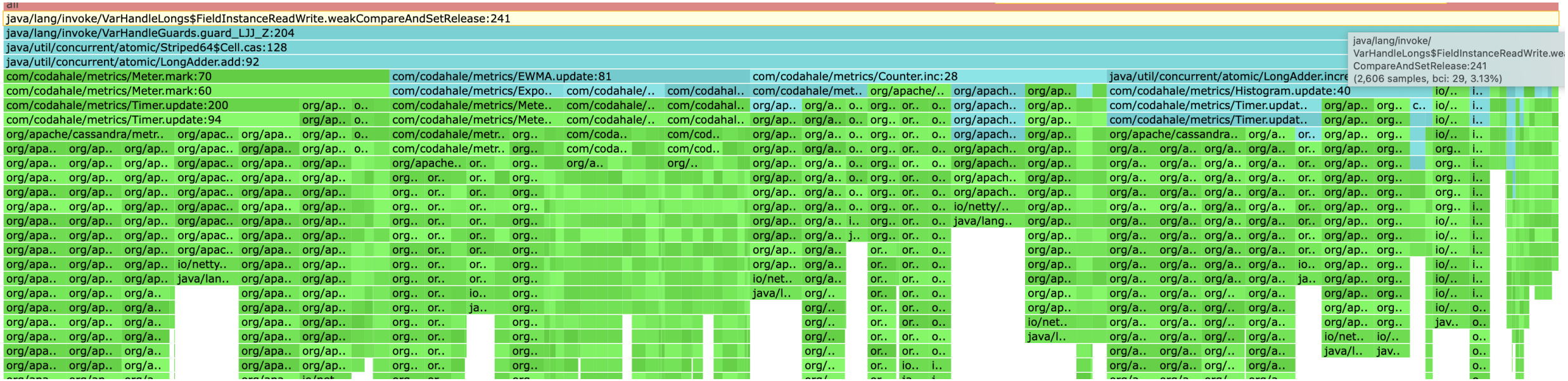


Metrics

- Cassadra uses Codehale/Dropwizard metrics
  - Counter: LongAdder wrapper
  - Meter: 4 LongAdder (total count + 1m/5m/15m rate)
  - Timer/Histogram – to skip due to lack of time

Metrics

LongAdder – is already optimized for multi-threaded contended code, can we do better?



Metrics

Thread-local counters

- Each thread has its own counter and increment it locally, cheap write
- Sum across threads to read

Metrics

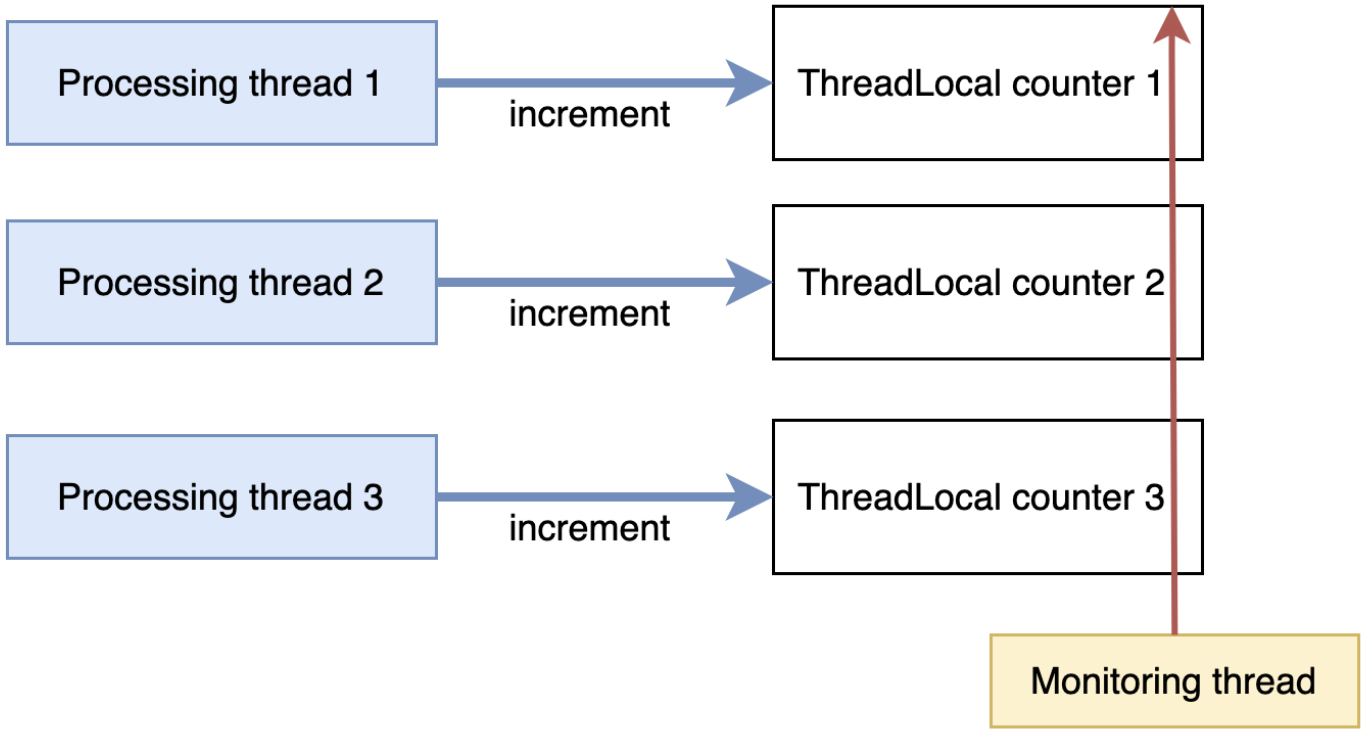
Thread-local counters

- Each thread has its own counter and increment it locally, cheap write
- Sum across threads to read

Metrics

Thread-local counters

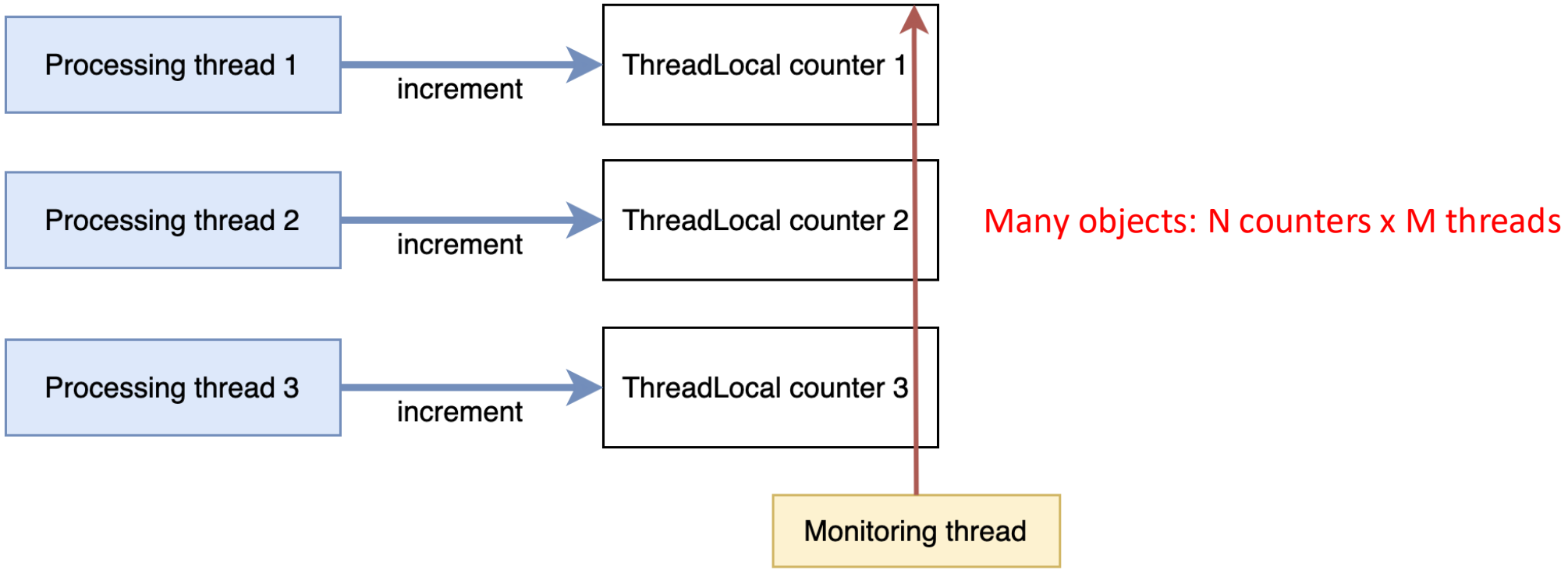
- Each thread has its own counter and increment it locally, cheap write
- Sum across threads to read (similar to LongAdder)



Metrics

Thread-local counters

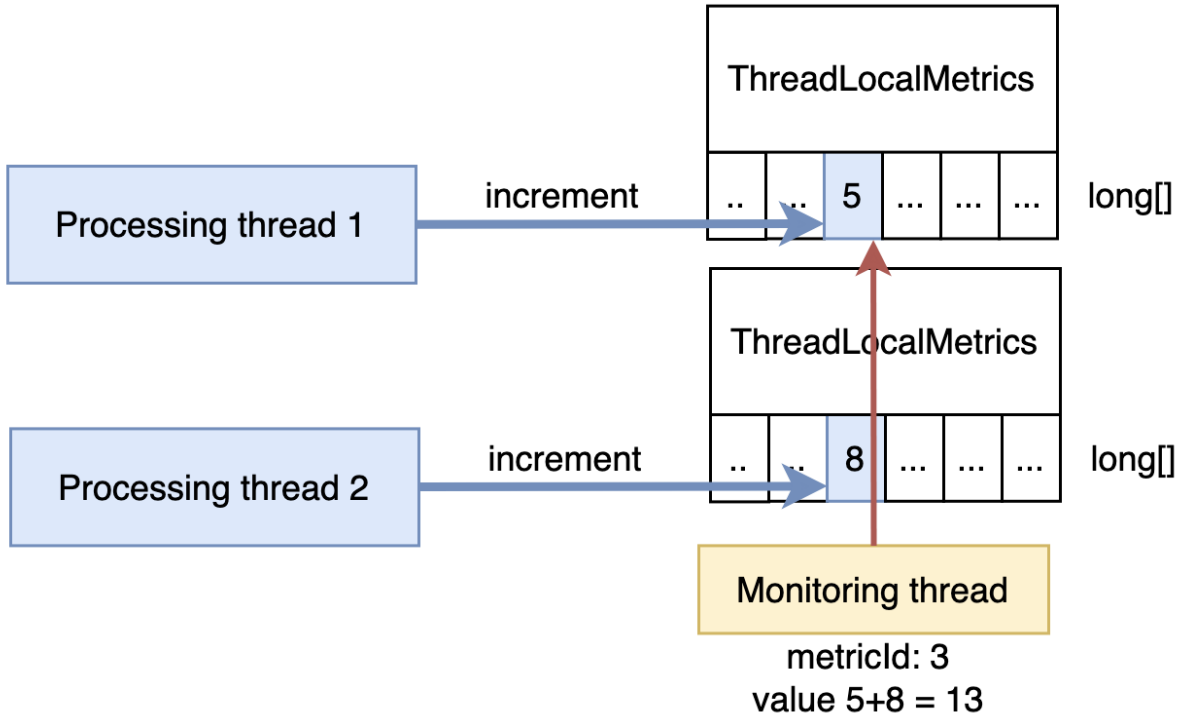
- Each thread has its own counter and increment it locally, cheap write
- Sum across threads to read



Metrics

Thread-local counters

- Each thread has its own counter and increment it locally, cheap write
- Sum across threads to read



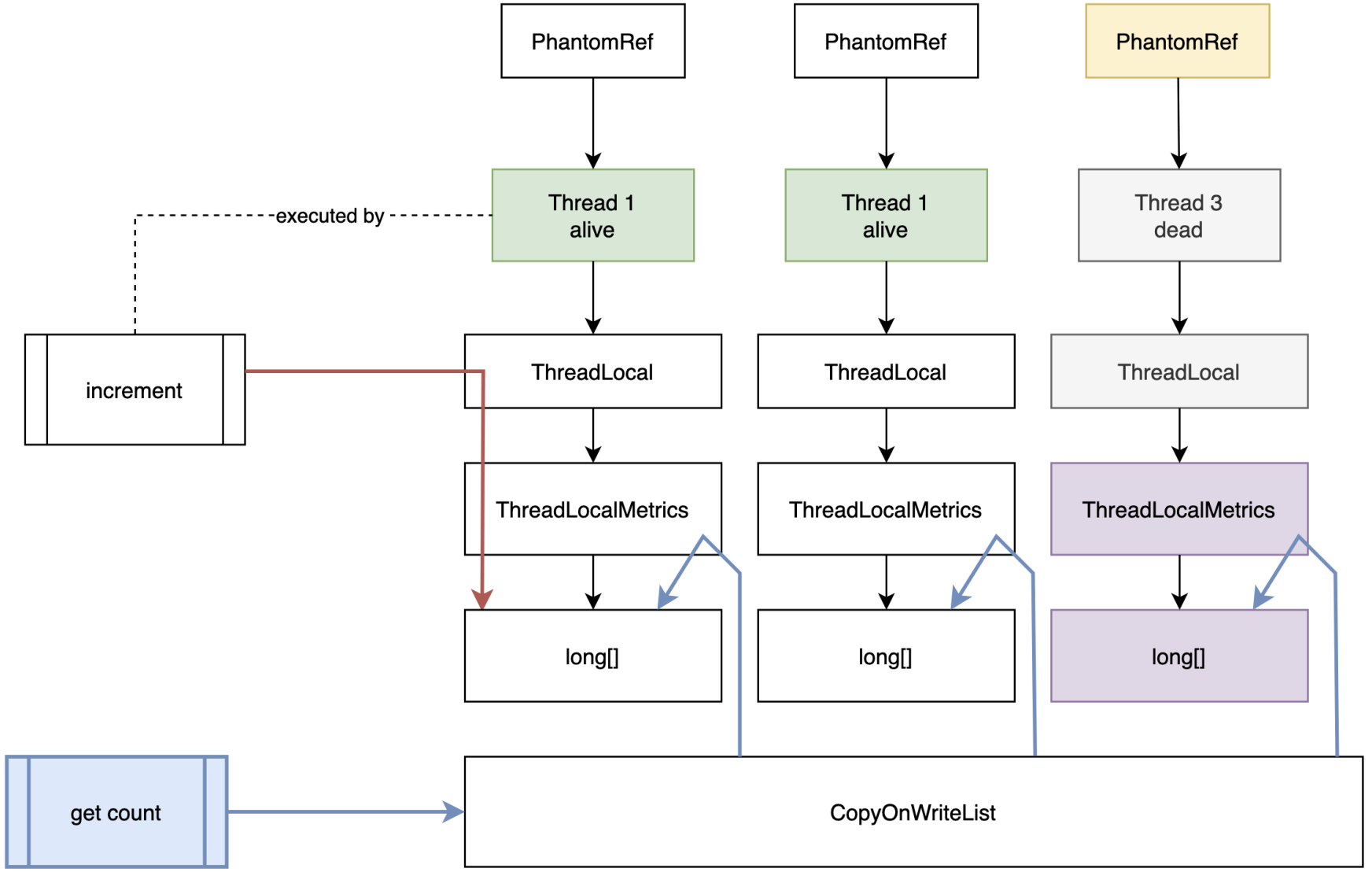
1 long per counter per thread

Metrics

Thread-local counters

- Each thread has its own counter and increment it locally, cheap write
- Sum across threads to read
- **Suggested by Benedict Elliott Smith in in CASSANDRA-20250**
- **Similar ideas are shared by Nitsan Wakart in his blog**

Real design is more complicated



Metrics

Thread-local counters

- Each thread has its own counter and increment it locally, cheap write
- Sum across threads to read
- **Suggested by Benedict Elliott Smith in CASSANDRA-20250**
- **Similar ideas are shared by Nitsan Wakart in his blog**

Microbenchmarks, JMH obviously

Server (VM, Linux, OpenJdk-11.0.26+4, Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz, 16 cores)

Benchmark	(type)	Mode	Cnt	Score	Error	Units
ThreadLocalMetricsBench	ThreadLocal	avgt	16	**4.299**	± 0.017	ns/op
ThreadLocalMetricsBench	LongAdder	avgt	16	**11.136**	± 0.003	ns/op

Time: **2.5x** lower (better 👍)

Microbenchmarks, JMH obviously

We increment usually more that one counter, especially for Meter object (total count + 1m/5m/15m rate)

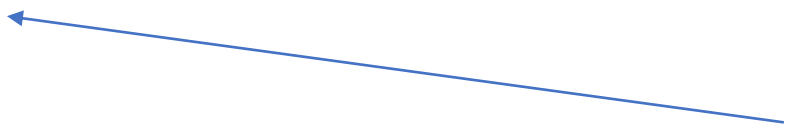
Memory is polluted by processing of requests themselves, the counters are evicted from CPU caches

Microbenchmarks, JMH obviously

We increment usually more that one counter, especially for Meter object (total count + 1m/5m/15m rate)

Memory is polluted by processing of requests themselves, the counters are evicted from CPU caches

```
Intel(R) Xeon(R) CPU E5-2680 v4
L1 Instruction Cache: 32 KB per core
L1 Data Cache: 32 KB per core
L2 Cache: 256 KB per core
L3 Cache: 35 MB
```

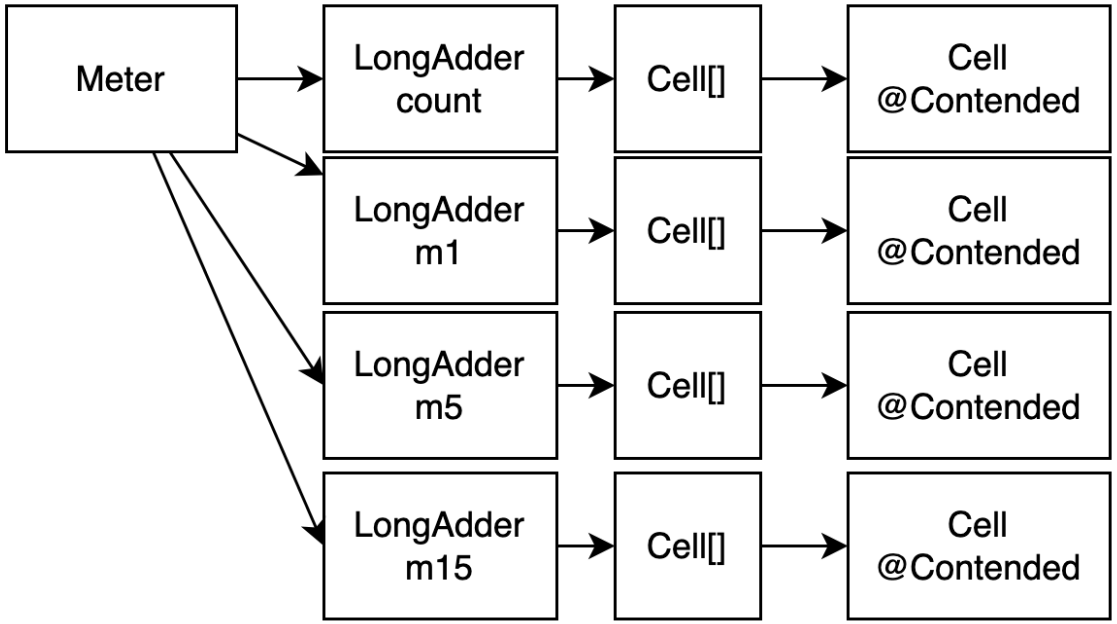


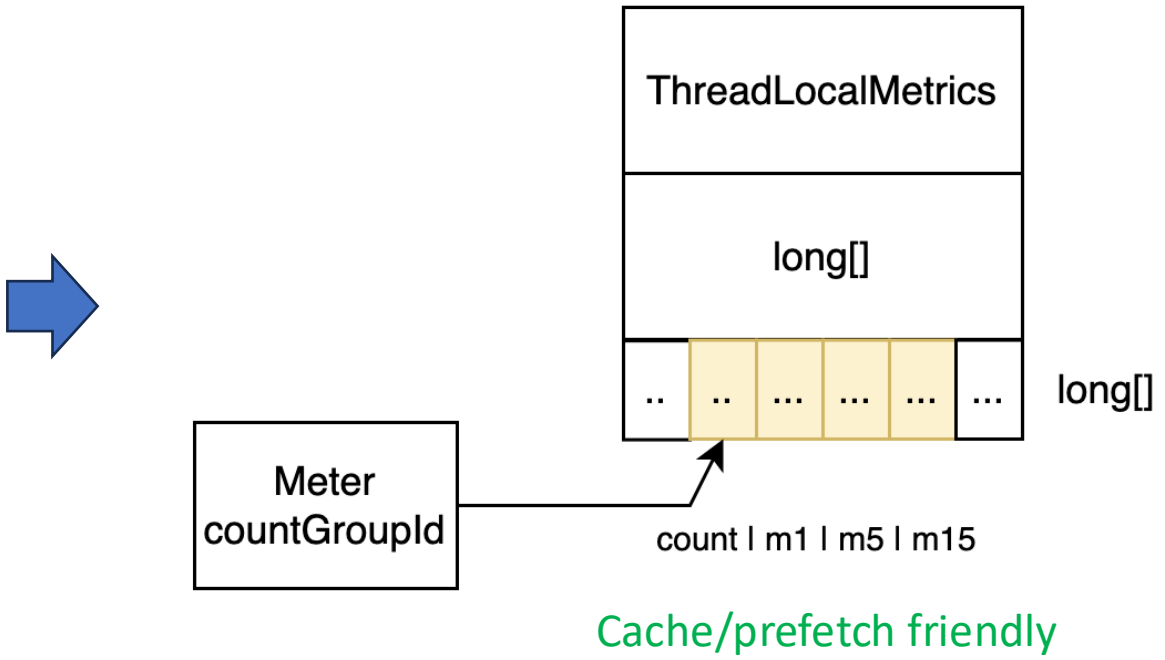
```
AtomicLongArray anotherMemory = new AtomicLongArray(256 * 1024);

@Setup(Level.Invocation)
public void polluteCpuCaches()
{
   for (int i = 0; i < anotherMemory.length(); i++)
      anotherMemory.incrementAndGet(i);
}
```

Microbenchmarks, JMH obviously

We increment usually more that one counter, especially for Meter object (total count + 1m/5m/15m rate)

Memory is polluted by processing of requests themselves, the counters are evicted from CPU caches





Cache/prefetch friendly

Microbenchmarks, JMH obviously

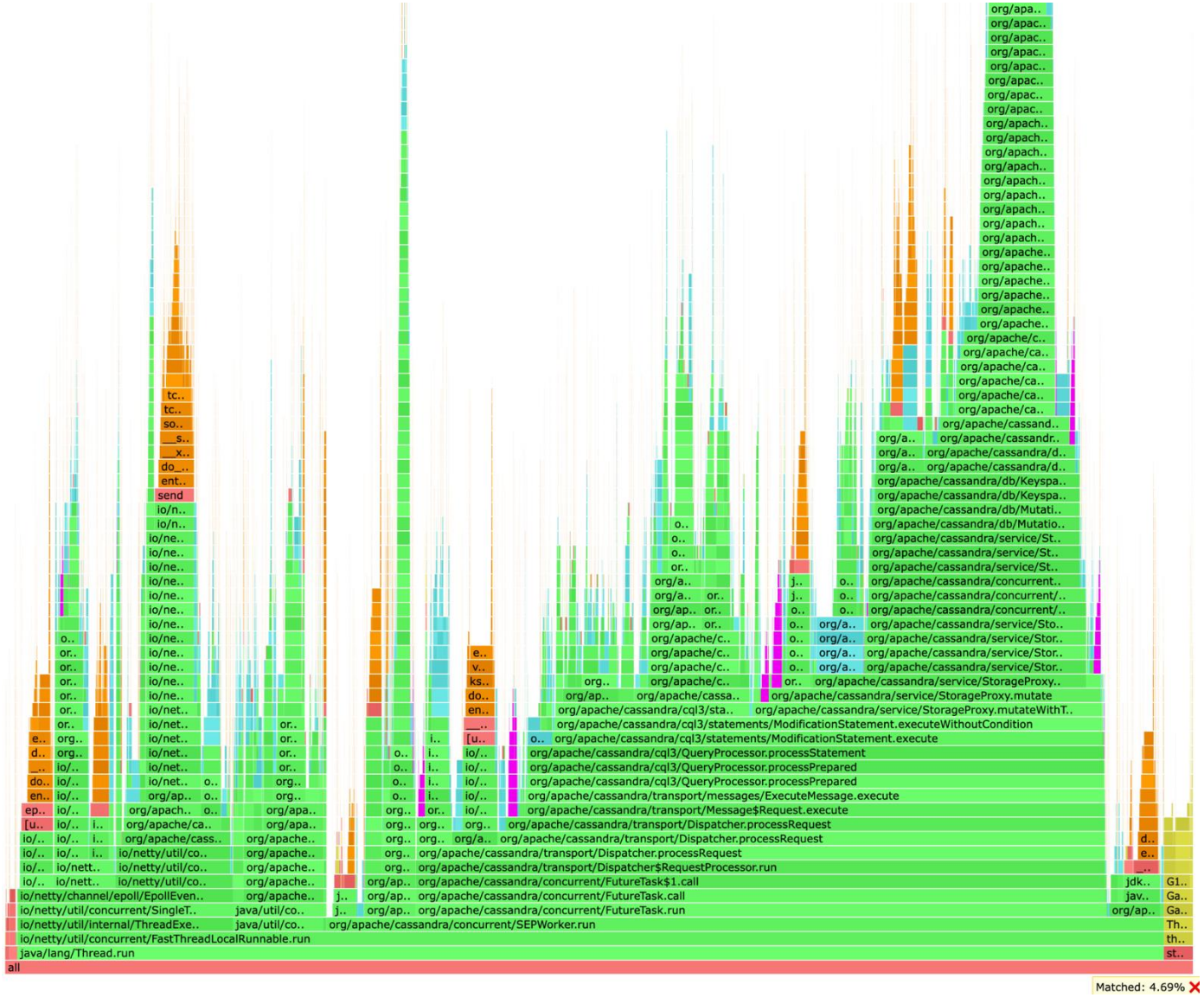
We increment usually more that one counter, especially for Meter object (total count + 1m/5m/15m rate)

Memory is polluted by processing of requests themselves, the counters are evicted from CPU caches

Benchmark	(type)	Mode	Cnt	Score	Error	Units
MetersBench.mark	ThreadLocal	avgt	16	**484.250**	± 14.694	ns/op
MetersBench.mark	Dropwizard	avgt	16	**2792.963**	± 215.250	ns/op

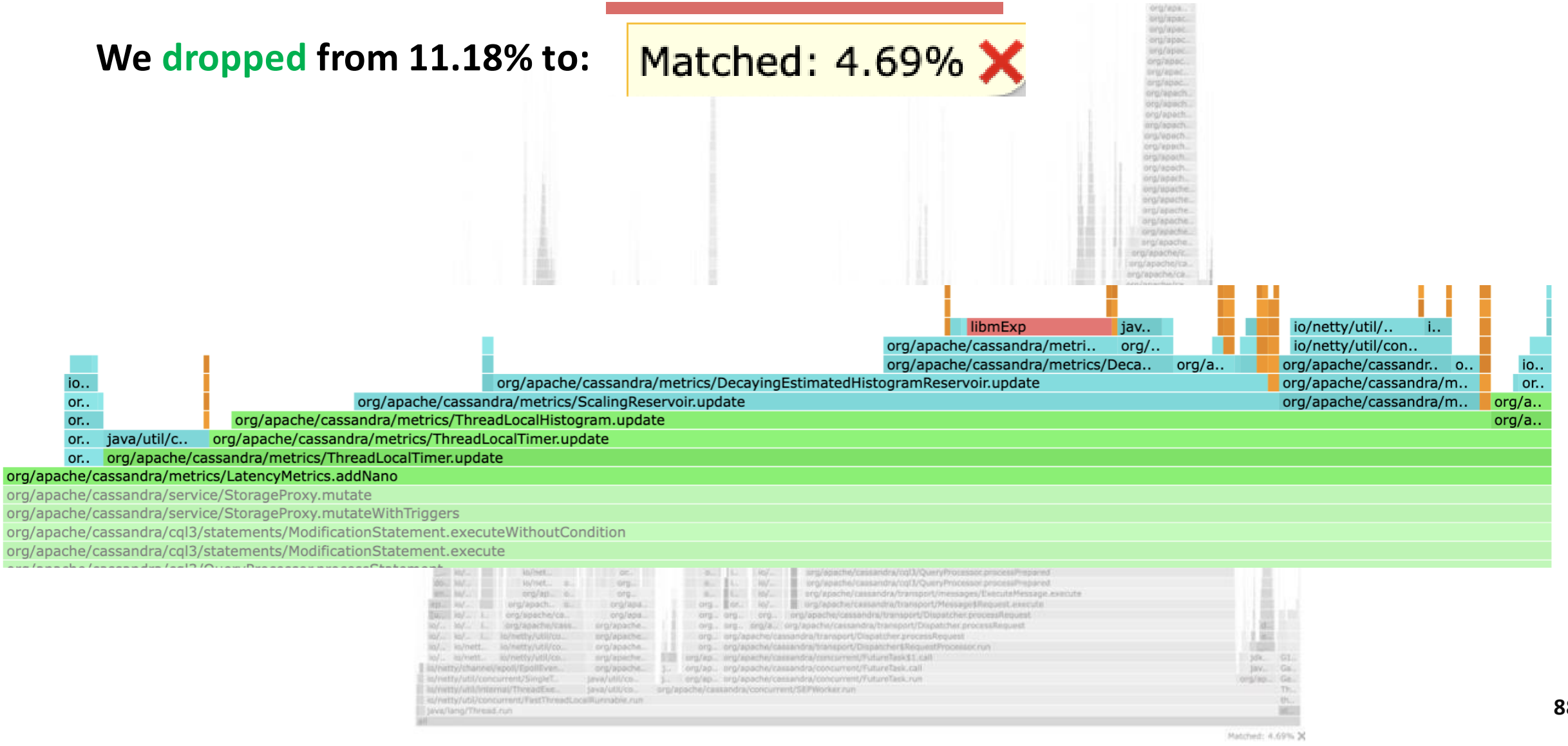
Time: **2.5x** lower (better 👍)

E2E test results



E2E test results

We dropped from 11.18% to:

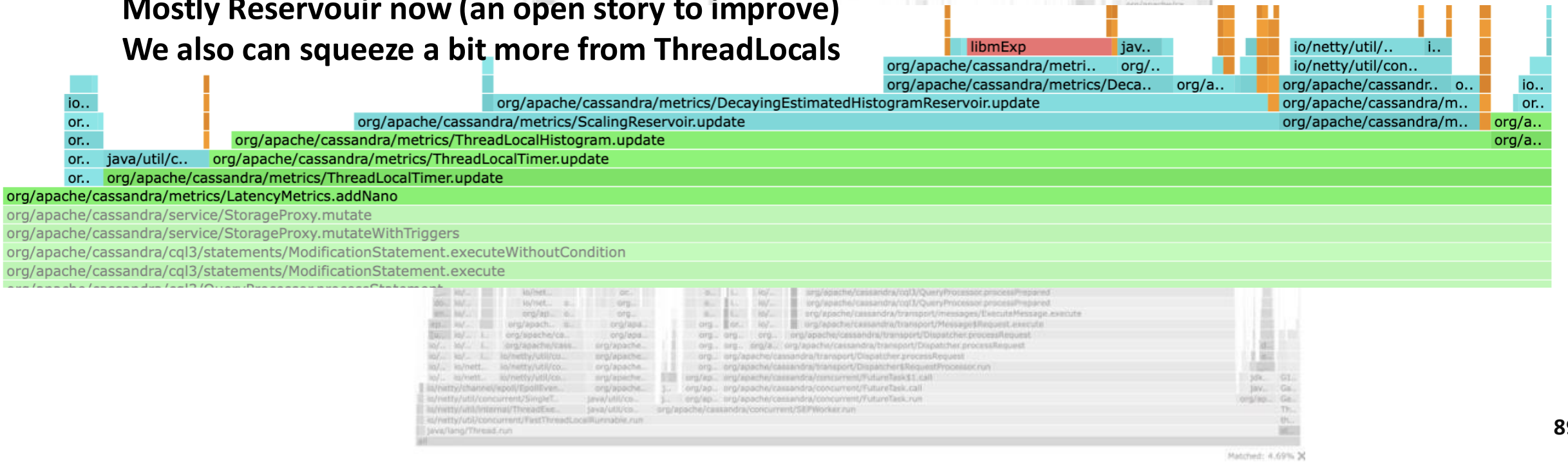


E2E test results

We **dropped** from 11.18% to:

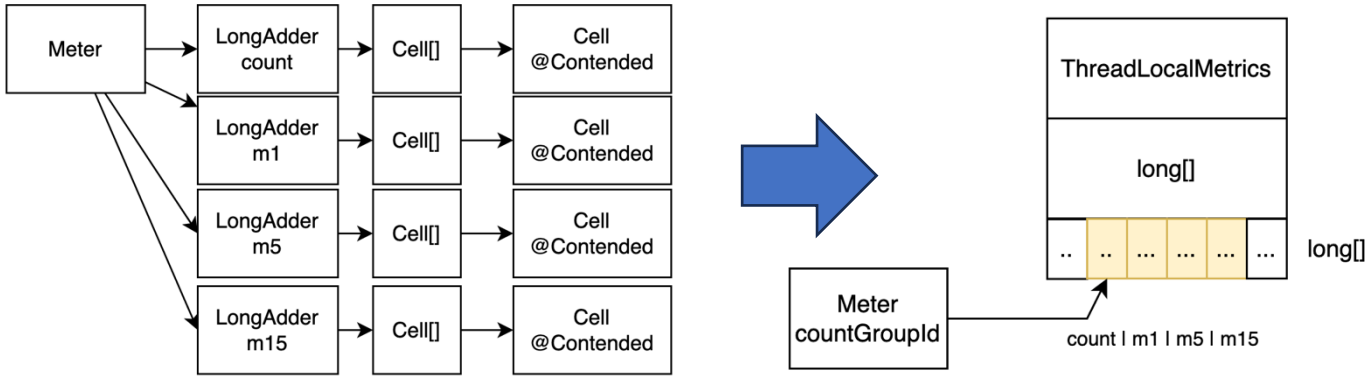
Matched: 4.69% ✗

Mostly Reservouir now (an open story to improve)
We also can squeeze a bit more from ThreadLocals



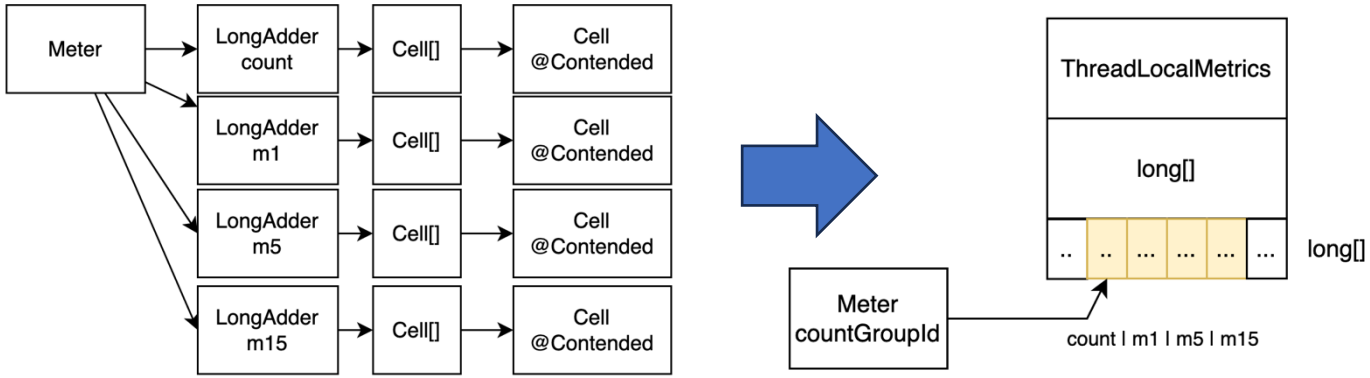
Summary

- What can be better that LongAdder?
  Thread local increment in long[] ☺



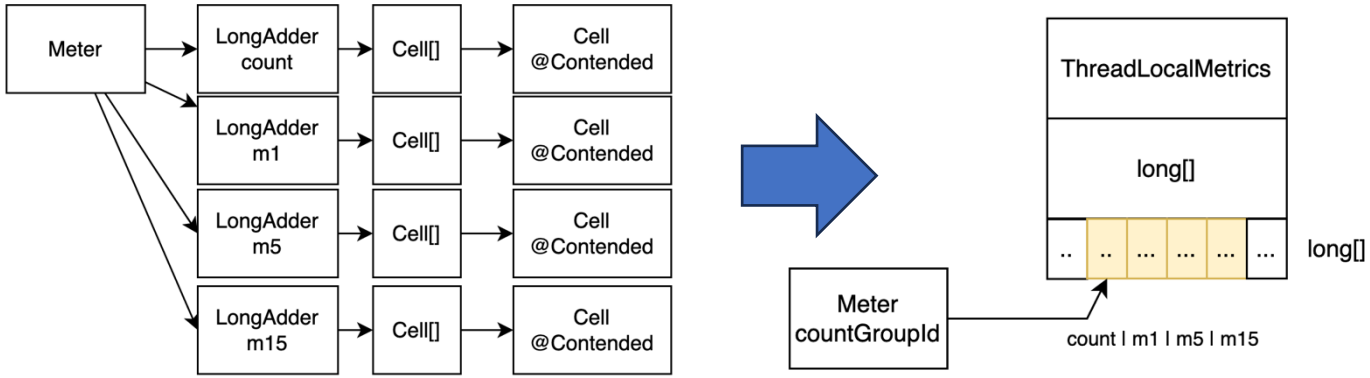
Summary

- What can be better that LongAdder?
  Thread local increment in long[] ☺



- If you have metric-like logic in batch processing
  you can use thread-local counters

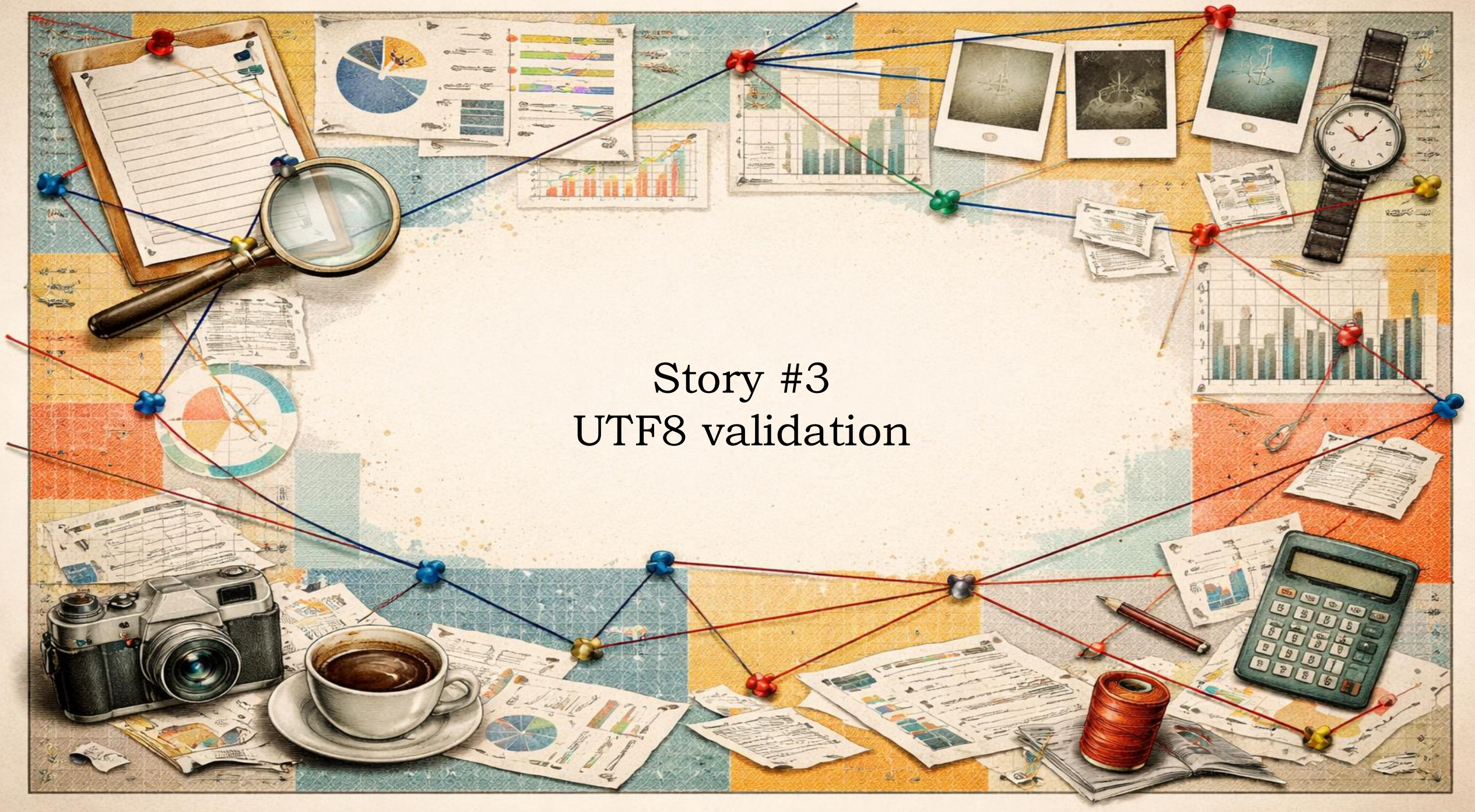
Summary



- What can be better that LongAdder?
  Thread local increment in long[] ☺

- If you have metric-like logic in batch processing
  you can use thread-local counters

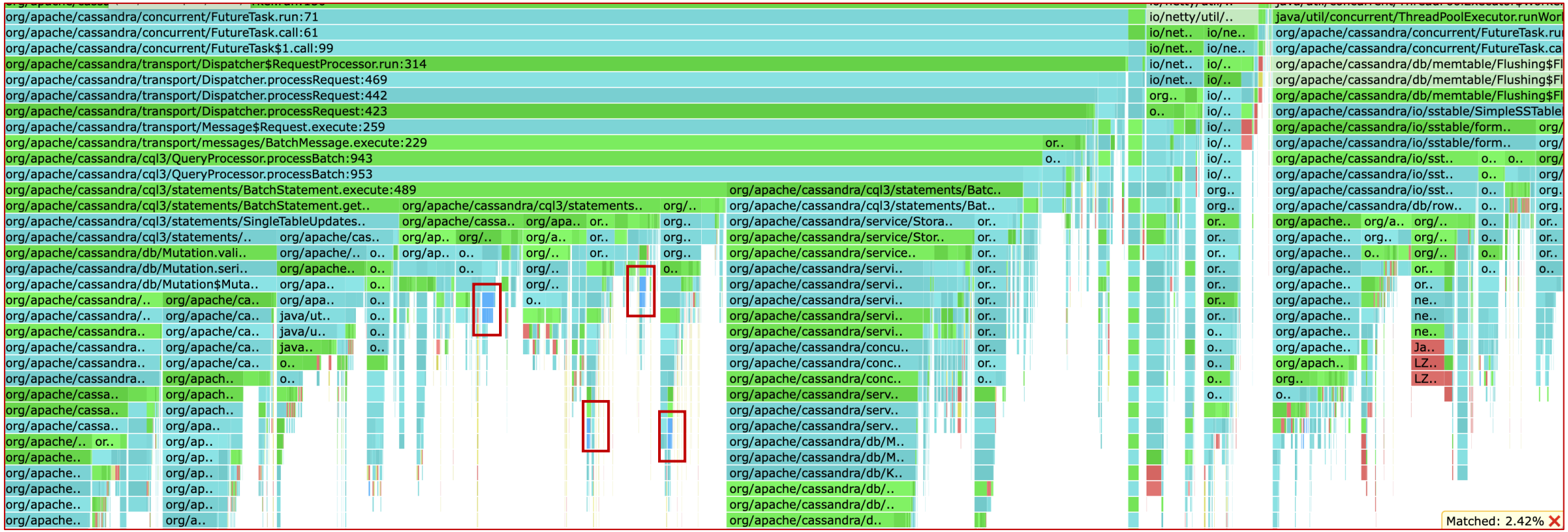
- In general: batch, collocate & avoid contention



Story #3
UTF8 validation

Optimize UTF8Validator.validate for ASCII prefixed Strings

- A user sends UTF-8 strings to CQL socket
- We need to validate incoming data – if it is really UTF-8



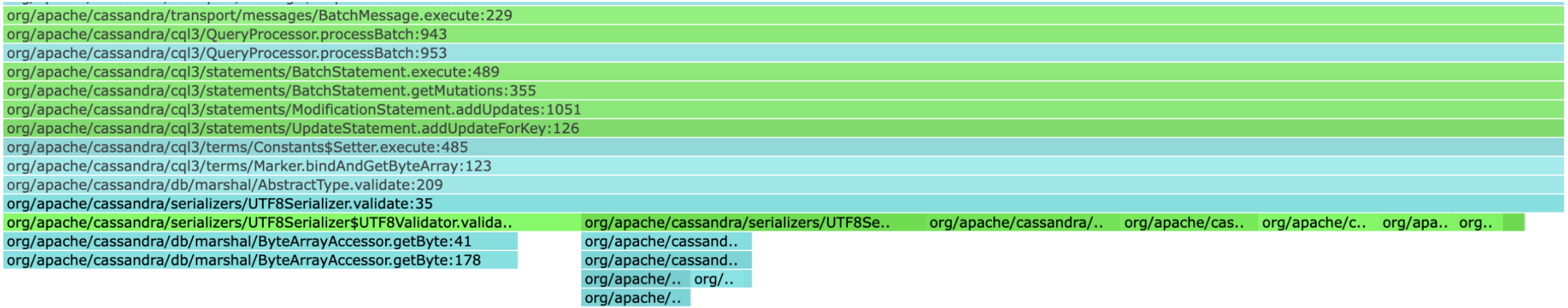
Optimize UTF8Validator.validate for ASCII prefixed Strings

```
static <V> boolean validate(V value, ValueAccessor<V> accessor)
{
    if (value == null)
        return false;

    int b = 0;
    int offset = 0;
    State state = State.START;
    while (!accessor.isEmptyFromOffset(value, offset))
    {
        b = accessor.getByte(value, offset++);
        switch (state)
        {
            case START:
                if (b >= 0)
                {
                    // ascii, state stays start.
                    if (b > 127)
                        return false;
                }
                else if ((b >> 5) == -2)
                {
                    // validate first byte of 2-byte char, 0xc2-0xdf
                    if (b == (byte) 0xc0)
                        // special case: modified utf8 null is 0xc080.
                        state = State.TWO_80;
                    else if ((b & 0x1e) == 0)
                        return false;
                    else
                        state = State.TWO;
                }
                else if ((b >> 4) == -2)
                {
                    // 3 bytes. first byte will be 0xe0 or 0xe1-0xef. handling of second byte will differ.
                    // so 0xe0,0xa0-0xbf,0x80-0xbf or 0xe1-0xef,0x80-0xbf,0x80-0xbf.
                    if (b == (byte)0xe0)
                        state = State.THREE_a0bf;
                    else
                        state = State.THREE_80bf_2;
                    break;
                }
                else if ((b >> 3) == -2)
                {
                    // 4 bytes. this is where the fun starts.
                    if (b == (byte)0xf0)
                        // 0xf0, 0x90-0xbf, 0x80-0xbf, 0x80-0xbf
                        state = State.FOUR_90bf;
                    else
                        // 0xf4, 0x80-0xbf, 0x80-0xbf, 0x80-0xbf
                        // 0xf1-0xf3, 0x80-0xbf, 0x80-0xbf, 0x80-0xbf
                        state = State.FOUR_80bf_3;
                    break;
                }
                else
                    return false; // malformed.
                break;
            case TWO:
                // validate second byte of 2-byte char, 0x80-0xbf
                if ((b & 0xc0) != 0x80)
                    return false;
                state = State.START;
                break;
            case TWO_80:
                if (b != (byte)0x80)
                    return false;
                state = State.START;
                break;
            case THREE_a0bf:
```

Много, сложно

Optimize UTF8Validator.validate for ASCII prefixed Strings

- Complex switch logic
- Not so good inlining (green vs blue colours) – too big method



Optimize UTF8Validator.validate for ASCII prefixed Strings

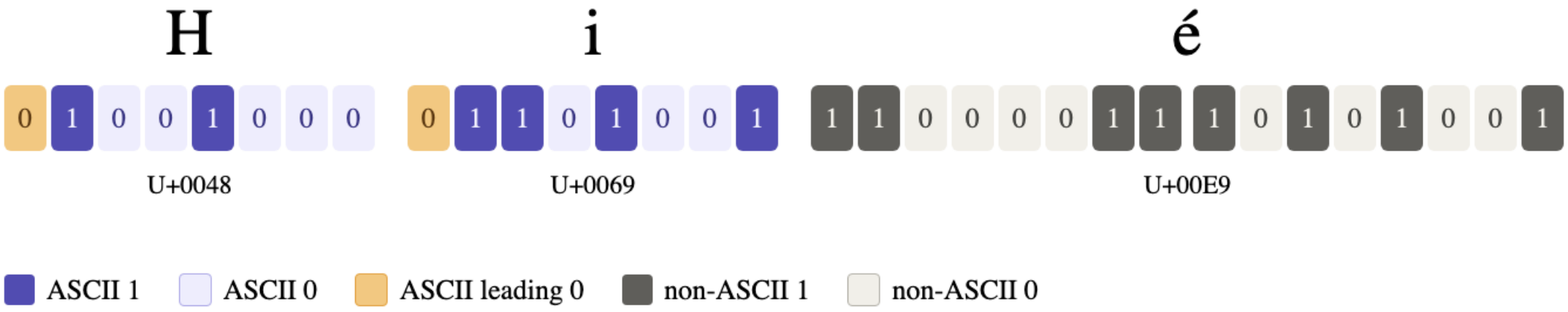
- Frequently for real data UTF8 strings are actually ASCII strings

ASCII, sign bit is 0

Byte 1	Byte 2	Byte 3	Byte 4
0yyyzzzz			
110xxxyy	10yyzzzz		
1110wwww	10xxxxyy	10yyzzzz	
11110uvv	10vvwwww	10xxxxyy	10yyzzzz

Optimize UTF8Validator.validate for ASCII prefixed Strings

- Frequently for real data UTF8 strings are actually ASCII strings



Optimize UTF8Validator.validate for ASCII prefixed Strings

- Frequently for real data UTF8 strings are actually ASCII strings
- ASCII strings/prefixes can be easy checked:

```
for (int i = off; i < end; i++) {
    if (bytes[i] < 0)
        return validateSlowPath(bytes, i, end);
}
```

Optimize UTF8Validator.validate for ASCII prefixed Strings

- Frequently for real data UTF8 strings are actually ASCII strings
- ASCII strings/prefixes can be easy checked:

```
for (int i = off; i < end; i++) {
    if (bytes[i] < 0)
        return validateSlowPath(bytes, i, end);
}
```

- https://lemire.me/blog/2018/10/16/validating-utf-8-bytes-java-edition/
- https://github.com/google/guava/blob/master/guava/src/com/google/common/base/Utf8.java#L123
- Similar ideas are in java.lang.StringCoding#decodeUTF8

Microbenchmark

Benchmark	(stringType)	Mode	Cnt	Score		Error	Units
UTF8ValidatorBench.old	short ASCII	avgt	15	**18.870**	±	2.499	ns/op
UTF8ValidatorBench.new	short ASCII	avgt	15	**9.554**	±	0.106	ns/op

Time: **2x** lower (better 👍)

Microbenchmark

Benchmark	(stringType)	Mode	Cnt	Score	Error	Units
UTF8ValidatorBench.old	short ASCII	avgt	15	18.870 ±	2.499	ns/op
UTF8ValidatorBench.new	short ASCII	avgt	15	9.554 ±	0.106	ns/op
UTF8ValidatorBench.old	long ASCII	avgt	15	**776.241** ±	12.015	ns/op
UTF8ValidatorBench.new	long ASCII	avgt	15	**193.148** ±	4.039	ns/op

Time: **4x** lower (better 👍)

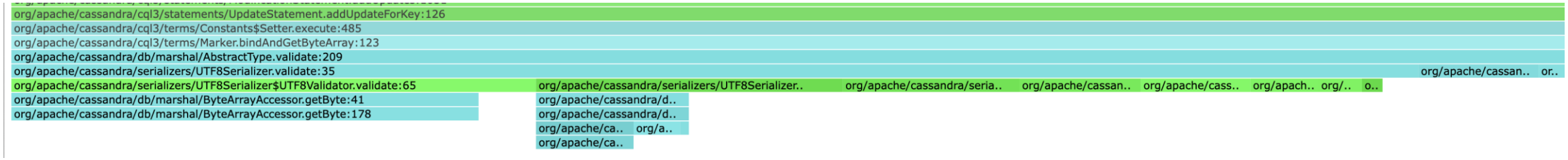
Microbenchmark

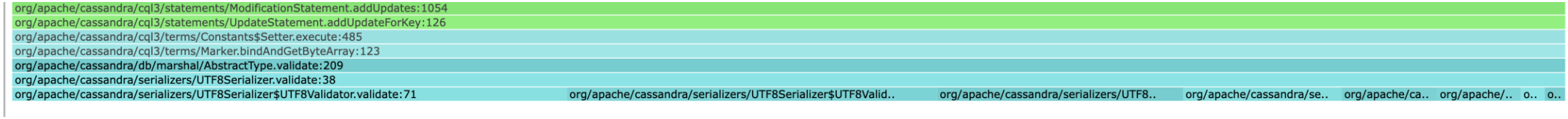
Benchmark	(stringType)	Mode	Cnt	Score	Error	Units
UTF8ValidatorBench.old	short ASCII	avgt	15	18.870 ±	2.499	ns/op
UTF8ValidatorBench.new	short ASCII	avgt	15	9.554 ±	0.106	ns/op
UTF8ValidatorBench.old	long ASCII	avgt	15	776.241 ±	12.015	ns/op
UTF8ValidatorBench.new	long ASCII	avgt	15	193.148 ±	4.039	ns/op
UTF8ValidatorBench.old	short ASCII prefix non-ASCII	avgt	15	**182.673** ±	1.329	ns/op
UTF8ValidatorBench.new	short ASCII prefix non-ASCII	avgt	15	**159.818** ±	9.852	ns/op
UTF8ValidatorBench.old	short non-ASCII	avgt	15	**142.486** ±	32.870	ns/op
UTF8ValidatorBench.new	short non-ASCII	avgt	15	**146.139** ±	7.165	ns/op
UTF8ValidatorBench.old	long non-ASCII	avgt	15	**2048.347** ±	33.241	ns/op
UTF8ValidatorBench.new	long non-ASCII	avgt	15	**2183.468** ±	254.497	ns/op

small degradation within an error range

E2e results

- -1% CPU
- It does not look a lot but it is just a single method change in a mature codebase
- 10 tickets like this -> significant improvement





Can we do better?

Can we do better?

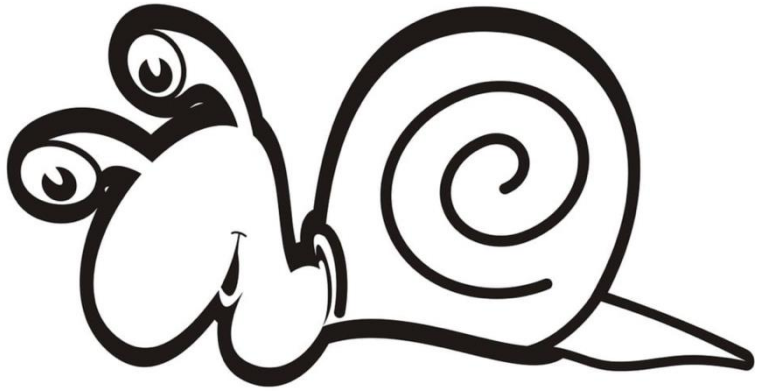
- SWAR
- Vector API

Can we do better?

- SWAR
- Vector API


Summary
VOYAGES
ÉCONOMIE
HISTOIRE

Summary



- Even mature projects have a lot of potential to optimize
- Define a goal, move in increments
- Reduce contention & improve data locality in hot paths
- Monitoring is a double-edged sword – it is essential for perf but also costs
- Check your data patterns, optimize for a typical scenario

Additional slides

Epilogue
JDK 21

Setup – Cassandra configuration

For those who curious..

- **JVM related settings**
  - **jdk-17.0.15+6 and jdk-21.0.10+7**
  - **heap size: 33082572800 = 30.813GiB (default, calculated by Cassandra scripts)**
  - GC: in case of G1 the following adjustments are made compared to the default configuration
    - -XX:ParallelGCThreads=16
    - -XX:ConcGCThreads=4
- Memtable / SSTable
  - trie memtable (64 shards)
  - memtable_allocation_type: offheap_objects
  - memtable_flush_writers: 8
  - SSTable format: big
- IO activities
  - compaction is disabled
  - commit log is disabled (due to lack of extra disk on my env to keep the IO rate)
- Others
  - native_transport_max_request_data_in_flight/native_transport_max_request_data_in_flight_per_ip increased to 2GiB
  - -Dcassandra.set_sep_thread_name=false
  - -Dio.netty.eventLoopThreads=2 (by default it is equal to number of cores we do not need that many ones, they only contend with other threads)

JDK 21 vs JDK 17, G1

JDK17, G1	JDK21, G1
op rate : 213,229 op/s [insert: 213,229 op/s] partition rate : 213,229 pk/s [insert: 213,229 pk/s] row rate : 2,132,292 row/s [insert: 2,132,292 row/s] latency mean : 1.4 ms [insert: 1.4 ms] latency median : 1.0 ms [insert: 1.0 ms] latency 95th percentile : 2.2 ms [insert: 2.2 ms] latency 99th percentile : 5.7 ms [insert: 5.7 ms] latency 99.9th percentile : 23.9 ms [insert: 23.9 ms] latency max : 389.5 ms [insert: 389.5 ms] total gc count : 38 total gc memory : 333.640 GiB total gc time : 6.8 seconds avg gc time : 177.8 ms stddev gc time : 87.6 ms Total operation time : 00:01:10	op rate : 230,427 op/s [insert: 230,427 op/s] partition rate : 230,427 pk/s [insert: 230,427 pk/s] row rate : 2,304,270 row/s [insert: 2,304,270 row/s] latency mean : 1.3 ms [insert: 1.3 ms] latency median : 0.9 ms [insert: 0.9 ms] latency 95th percentile : 1.9 ms [insert: 1.9 ms] latency 99th percentile : 4.3 ms [insert: 4.3 ms] latency 99.9th percentile : 23.0 ms [insert: 23.0 ms] latency max : 802.2 ms [insert: 802.2 ms] total gc count : 30 total gc memory : 358.018 GiB total gc time : 7.1 seconds avg gc time : 236.2 ms stddev gc time : 106.0 ms Total operation time : 00:01:05

JDK 21 vs JDK 17, G1

+8% throughput

Shenandoah (non-gen), JDK 21, 100 client threads	Shenandoah (non-gen), JDK 21, 200 client threads
Op rate : 141,221 op/s [insert: 141,221 op/s] Partition rate : 141,221 pk/s [insert: 141,221 pk/s] Row rate : 1,412,214 row/s [insert: 1,412,214 row/s] Latency mean : 0.7 ms [insert: 0.7 ms] Latency median : 0.6 ms [insert: 0.6 ms] Latency 95th percentile : 1.1 ms [insert: 1.1 ms] Latency 99th percentile : 1.6 ms [insert: 1.6 ms] Latency 99.9th percentile : 12.7 ms [insert: 12.7 ms] Latency max : 39.0 ms [insert: 39.0 ms] Total partitions : 15,000,000 [insert: 15,000,000] Total errors : 0 [insert: 0] Total GC count : 162 Total GC memory : 114.859 GiB Total GC time : 74.2 seconds Avg GC time : 458.3 ms StdDev GC time : 938.6 ms Total operation time : 00:01:46	Op rate : 157,846 op/s [insert: 159,524 op/s] Partition rate : 157,846 pk/s [insert: 159,524 pk/s] Row rate : 1,578,456 row/s [insert: 1,595,243 row/s] Latency mean : 1.2 ms [insert: 1.2 ms] Latency median : 1.0 ms [insert: 1.0 ms] Latency 95th percentile : 2.4 ms [insert: 2.4 ms] Latency 99th percentile : 4.9 ms [insert: 4.9 ms] Latency 99.9th percentile : 19.7 ms [insert: 19.7 ms] Latency max : 54.8 ms [insert: 54.8 ms] Total partitions : 15,000,000 [insert: 15,000,000] Total errors : 0 [insert: 0] Total GC count : 160 Total GC memory : 28.846 GiB Total GC time : 87.6 seconds Avg GC time : 547.6 ms StdDev GC time : 1,115.2 ms Total operation time : 00:01:35

JDK 21 genZGC, first attempt

JDK 21, GenZGC	
op rate	90,155 op/s [insert: 90,155 op/s]
partition rate	90,155 pk/s [insert: 90,155 pk/s]
row rate	901,547 row/s [insert: 901,547 row/s]
latency mean	0.5 ms [insert: 0.5 ms]
latency median	0.5 ms [insert: 0.5 ms]
latency 95th percentile	0.8 ms [insert: 0.8 ms]
latency 99th percentile	1.0 ms [insert: 1.0 ms]
latency 99.9th percentile	4.8 ms [insert: 4.8 ms]
latency max	26.1 ms [insert: 26.1 ms]
total gc count	247
total gc memory	52.568 GiB (wrong value, ignore it)
total gc time	211.1 seconds
avg gc time	854.5 ms
stddev gc time	2,033.0 ms
Total operation time	00:02:46

```
[2026-02-03T18:32:04.578+0000][66.750s][5831][6108][info
] Allocation Stall (SharedPool-Worker-25) 450.500ms
```

GC paused individual app threads if it cannot collect in time

Recall: ZGC does NOT support compressed references

JDK 21 genZGC, second attempt

```
memtable_heap_space: 2048MiB # by default Cassandra uses 1/4 of heap
size, in our case it is 7887MiB
```

ZGC, JDK21, 100 client threads

```
Op rate                   :  163,608 op/s
Row rate                  : 1,636,084 row/s
Latency mean              :    0.6 ms
Latency median            :    0.5 ms
Latency 95th percentile   :    0.9 ms
Latency 99th percentile   :    1.2 ms
Latency 99.9th percentile :   11.6 ms
Latency max               :   35.9 ms
Total partitions          : 15,000,000
Total errors              :          0
Total GC count            : 178
Total GC memory           : 220.252 GiB
Total GC time             :   51.9 seconds
Avg GC time               :  291.5 ms
StdDev GC time            :  571.7 ms
Total operation time      : 00:01:31
```

ZGC, JDK21, 200 client threads

```
Op rate                   :  193,924 op/s
Row rate                  : 1,939,237 row/s
Latency mean              :    1.0 ms
Latency median            :    0.8 ms
Latency 95th percentile   :    1.8 ms
Latency 99th percentile   :    3.7 ms
Latency 99.9th percentile :   20.5 ms
Latency max               :   76.1 ms
Total partitions          : 15,000,000
Total errors              :          0
Total GC count            : 224
Total GC memory           : 80.476 GiB
Total GC time             :   71.0 seconds
Avg GC time               :  317.0 ms
StdDev GC time            :  646.6 ms
Total operation time      : 00:01:17
```