
# Cassandra memory footprint

## Память Cassandra по полочкам

Дмитрий Константинов

# About me

- Дмитрий Константинов
- Системный архитектор и Java разработчик
- Разбираюсь с различными OpenSource технологиями, такими как Apache Cassandra, Zookeeper, Kafka, Hazelcast
- Опыт работы с Apache Cassandra: около 7 лет
- Профессиональные интересы:
  - распределенные системы
  - производительность
  - отказоустойчивость

# Java OutOfMemory (OOM)

ERROR [CompactionExecutor:2] CassandraDaemon.java:207 - Exception in thread Thread[CompactionExecutor:2,1,main]
java.lang.OutOfMemoryError: Direct buffer memory at java.nio.Bits.reserveMemory(Bits.java:694) at java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123) ~[na:1.8.0_171] at java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:311) at org.apache.cassandra.utils.memory.BufferPool.allocate(BufferPool.java:108) at org.apache.cassandra.utils.memory.BufferPool.access$1000(BufferPool.java:45) at

# Java OutOfMemory (OOM)

```
ERROR [CompactionExecutor:2] CassandraDaemon.java:207 - Exception in thread
Thread[CompactionExecutor:2,1,main]
java.lang.OutOfMemoryError: Direct buffer memory at
java.nio.Bits.reserveMemory(Bits.java:694) at
java.nio.DirectByteBuffer.<init>(DirectByteBuffer.java:123) ~[na:1.8.0_171] at
java.nio.ByteBuffer.allocateDirect(ByteBuffer.java:311) at
org.apache.cassandra.utils.memory.BufferPool.allocate(BufferPool.java:108) at
org.apache.cassandra.utils.memory.BufferPool.access$1000(BufferPool.java:45) at
```

# Linux OOM Killer

```
dmesg -T | grep -i killed
[Aug 12 16:42:24 2023] Killed process 31211 (java) total-vm:8651940kB, anon-
rss:6635464kB, file-rss:153648kB, shmem-rss:0kB
```


JAVA OOM
LINUX OOM KILLER

# Reasons

Goal

- Avoid Java OOM
- Avoid OS OOM Killer
- Optimal performance

# Reasons

Goal

- Avoid Java OOM
- Avoid OS OOM Killer
- Optimal performance

Background

- Understand how memory is used
- Memory vs X trade-offs

# Reasons

Goal

- Avoid Java OOM
- Avoid OS OOM Killer
- Optimal performance

Background

- Understand how memory is used
- Memory vs X trade-offs

How

- Control/limit memory usage
- Monitor

# Reasons

- Avoid Java OOM
- Avoid OS OOM Killer
- Optimal performance

- Understand how memory is used
- Memory vs X trade-offs

- Control/limit memory usage
- Monitor

+ bonus: how Cassandra devs solved some of memory related problems

# Environment

- Linux, 4.17.14
- Cassandra 4.1.4, 3-node cluster
- OpenJDK 11.x, HotSpot x64
- G1 GC
- heap size = 8GiB, 16GiB
- Total RAM: 48 GiB
- 28 CPU cores


- JVM

Cassandra memory = Java heap…
I can set –Xmx
and it is the maximum
which my Cassandra can consume

Let’s set –Xmx = all/90% of RAM

# Memory structure

- Heap
- Off-heap
- Threads
- JVM/GC internals


https://www.youtube.com/watch?v=kKigibHrV5I

# Native Memory Tracking

- Add conf/jvm-server.options: **-XX:NativeMemoryTracking=summary**
- Run basic write workload + repair to warmup a Cassandra process

# Native Memory Tracking

- Add conf/jvm-server.options: -XX:NativeMemoryTracking=summary
- Run basic write workload + repair to warmup a Cassandra process
- **$JAVA_HOME/bin/jcmd <pid> VM.native_memory summary**

# Native Memory Tracking

- Add conf/jvm-server.options: -XX:NativeMemoryTracking=summary
- Run basic write workload + repair to warmup a Cassandra process
- $JAVA_HOME/bin/jcmd <pid> VM.native_memory summary
- See also:
  - https://shipilev.net/jvm/anatomy-quarks/12-native-memory-tracking/
  - https://blog.arkey.fr/2020/11/30/off-heap-reconnaissance/
  - https://stackoverflow.com/questions/65157496/what-goes-into-the-other-section-of-javas-native-memory-tracking-output

# Native Memory Tracking

Total: reserved=11272268KB, committed=9993272KB
- Java Heap (reserved=8388608KB, committed=8388608KB)
  (mmap: reserved=8388608KB, committed=8388608KB)

- Class (reserved=1114854KB, committed=73230KB)
  (classes #11353)
  ( instance classes #10596, array classes #757)
  (malloc=2790KB #40947)
  (mmap: reserved=1112064KB, committed=70440KB)
  ( Metadata: )
  ( reserved=63488KB, committed=62160KB)
  ( used=60595KB)
  ( free=1565KB)
  ( waste=0KB =0.00%)
  ( Class space:)
  ( reserved=1048576KB, committed=8280KB)
  ( used=7229KB)
  ( free=1051KB)
  ( waste=0KB =0.00%)

...

# Native Memory Tracking

| Category | Size (committed) | % of heap |
|---|---|---|
| Heap | 8192 MiB | 100% |
| Others | 967 MiB | 12% |
| GC | 384 MiB | 5% |
| Class | 71 MiB | |
| Thread | 54 MiB | |
| Code | 53 MiB | |
| Symbol | 13 MiB | |
| Internal | 2 MiB | |
| Compiler | 1 MiB | |
| Total, non-heap | 1550 MiB | 19% |

**-Xmx8G**

# Native Memory Tracking

| Category | Size (committed) | % of heap |
|---|---|---|
| Heap | 8192 MiB | 100% |
| Others | 967 MiB | 12% |
| GC | 384 MiB | 5% |
| Class | 71 MiB | |
| Thread | 54 MiB | |
| Code | 53 MiB | |
| Symbol | 13 MiB | |
| Internal | 2 MiB | |
| Compiler | 1 MiB | |
| Total, non-heap | 1550 MiB | 19% |

**-Xmx8G**

| Category | Size (committed) | % of heap |
|---|---|---|
| Heap | 16384 MiB | 100% |
| Others | 966 MiB | 6% |
| GC | 694 MiB | 4% |
| Class | 71 MiB | |
| Thread | 54 MiB | |
| Code | 53 MiB | |
| Symbol | 14 MiB | |
| Internal | 2 MiB | |
| Compiler | 1 MiB | |
| Total, non-heap | 1863 MiB | 11% |

**-Xmx16G**

# Native Memory Tracking

| Category | Size (committed) | % of heap |
|---|---|---|
| Heap | 8192 MiB | 100% |
| Others | 967 MiB | 12% |
| GC | 384 MiB | 5% |
| Class | 71 MiB | |
| Thread | 54 MiB | |
| Code | 53 MiB | |
| Symbol | 13 MiB | |
| Internal | 2 MiB | |
| Compiler | 1 MiB | |
| Total, non-heap | 1550 MiB | 19% |

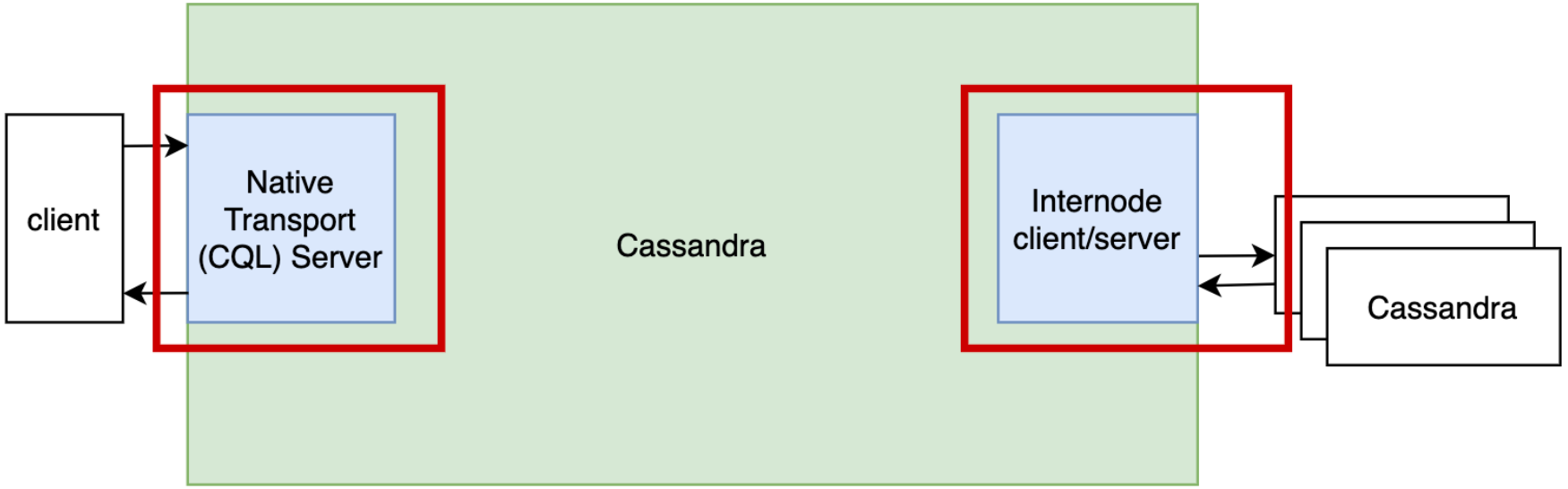
| Category | Size (committed) | % of heap |
|---|---|---|
| Heap | 16384 MiB | 100% |
| Others | 966 MiB | 6% |
| GC | 694 MiB | 4% |
| Class | 71 MiB | |
| Thread | 54 MiB | |
| Code | 53 MiB | |
| Symbol | 14 MiB | |
| Internal | 2 MiB | |
| Compiler | 1 MiB | |
| Total, non-heap | 1863 MiB | 11% |

Let’s discuss a bit later …

# Interim summary

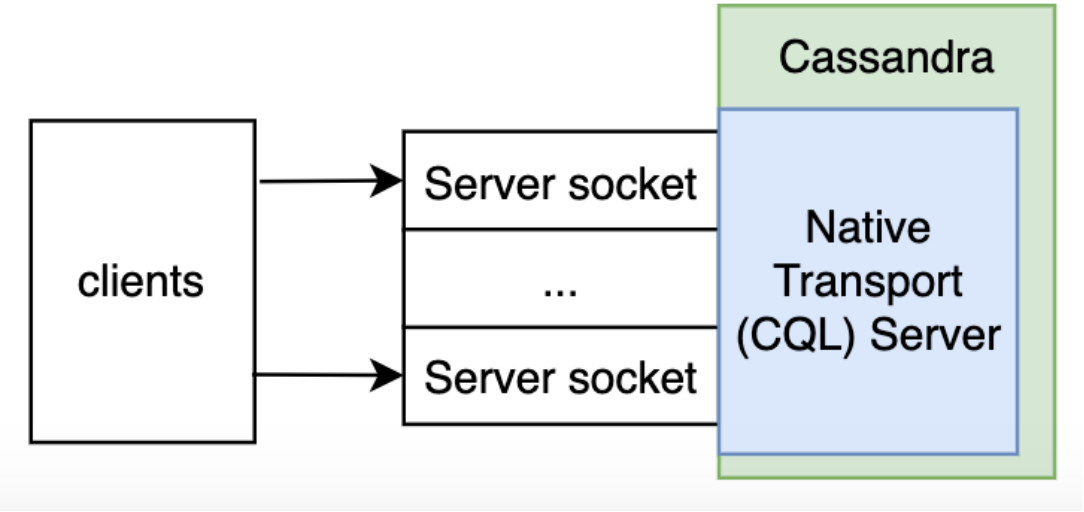
| Structures type | Heap/offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| Total | | | 119% | | |


- JVM
- **Network**
  - **Sockets**



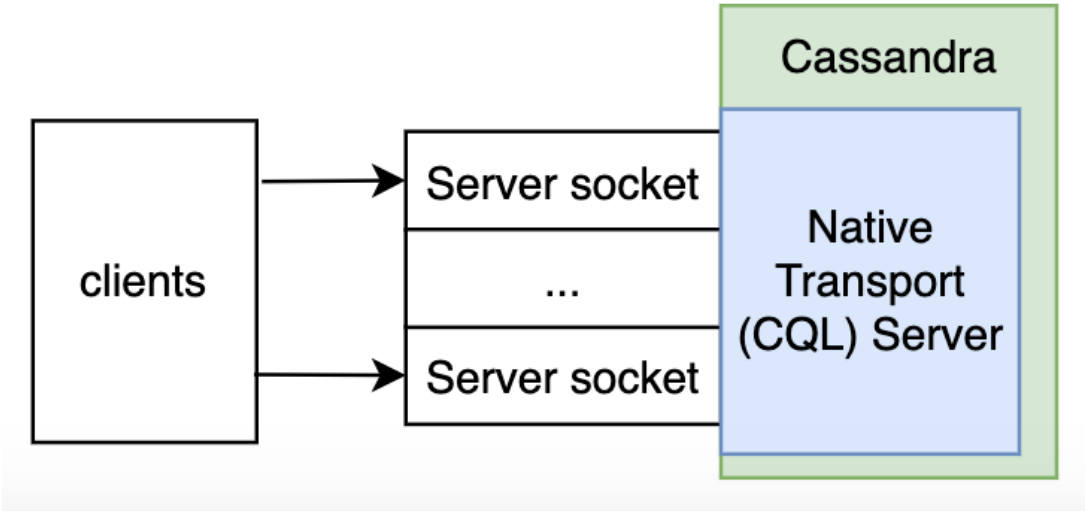
# Network sockets

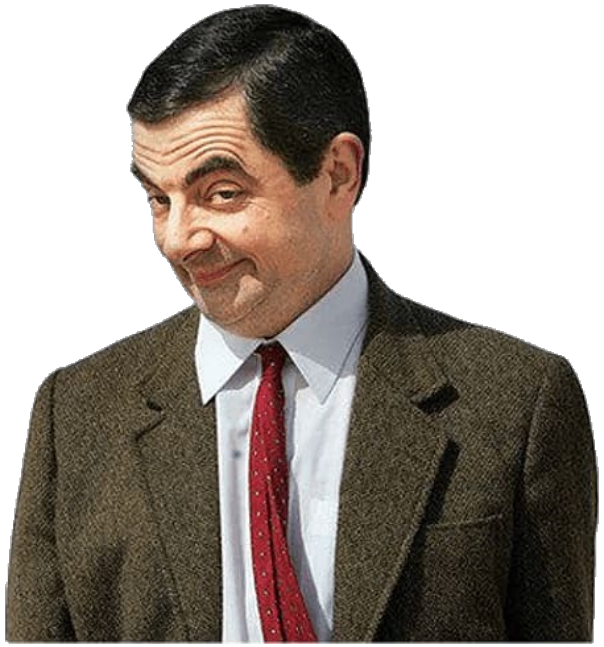
- I have Socket objects in my Java heap, this is all what I need to take in account..



# Network sockets

- I have Socket objects in my Java heap, this is all what I need to take in account..



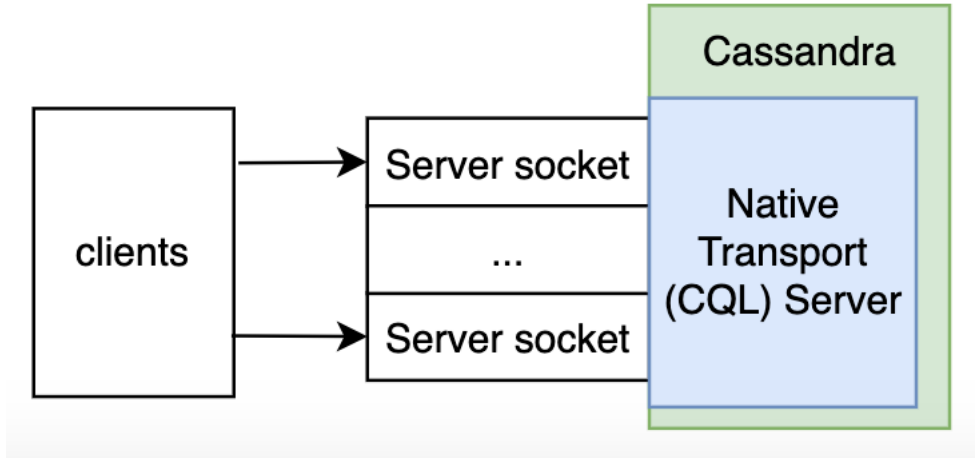



**IT'S FREE**

# Network - sockets

Test:

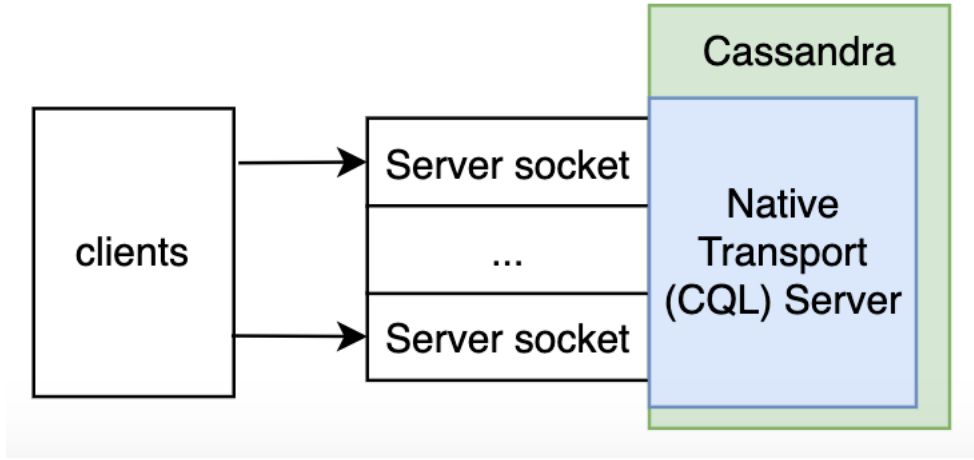
- open a lot (100) of CQL connections
- Send many large INSERT requests (20MiB)
- Slow down Cassandra server



# Network - sockets

Test:

- open a lot (100) of CQL connections
- Send many large INSERT requests (20MiB)
- Slow down Cassandra server




```
cat /proc/net/sockstat
sockets: used 661
TCP: inuse 251 orphan 0 tw 12 alloc 380 mem 372331
UDP: inuse 7 mem 770
UDPLITE: inuse 0
RAW: inuse 23
FRAG: inuse 0 memory 0
```

In 4KiB pages = 1.4 GiB

# Network - sockets

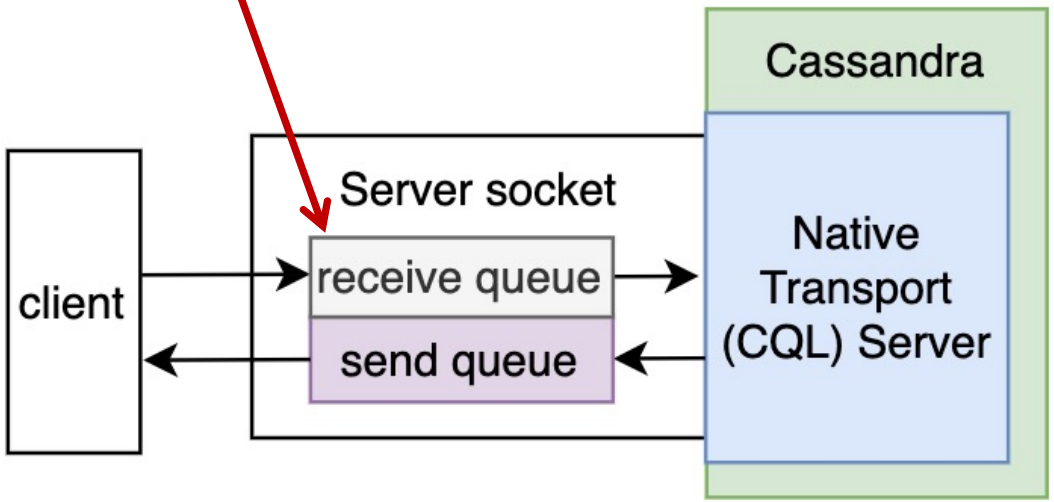
```
ss -atnmp | grep pid=10614 -A 1

State    Recv-Q Send-Q   Local Address:Port  Peer Address:Port
ESTAB  15732304      0         x.x.x.x:9042      y.y.y.y:57954  users:(("java",pid=29510,
skmem:(r15985904,rb16777216,t0,tb3151872,f784,w0,o0,bl0)
...
```

# Network - sockets

```
ss -atnmp | grep pid=10614 -A 1

State    Recv-Q Send-Q   Local Address:Port  Peer Address:Port
ESTAB  15732304      0         x.x.x.x:9042      y.y.y.y:57954  users:(("java",pid=29510,
skmem:(r15935904,rb16777216,t0,tb3151872,f784,w0,o0,bl0)
...
```



# Network - sockets

```
ss –atnmp | grep pid=10614 –A 1

State    Recv-Q Send-Q   Local Address:Port  Peer Address:Port
ESTAB  15732304      0         x.x.x.x:9042      y.y.y.y:57954  users:(("java",pid=29510,..
skmem:(r15985904,rb16777216,t0,tb3151872,f784,w0,o0,bl0)
...
```

skmem:(r<rmem_alloc>,rb<rcv_buf>,t<wmem_alloc>,tb<snd_buf>,f<fwd_alloc>,w<wmem_queued>,o<opt_mem>,bl<back_log>)

<rmem_alloc> the memory allocated for receiving packet
<rcv_buf> the total memory can be allocated for receiving packet
<wmem_alloc> the memory used for sending packet (which has been sent to layer 3)
…

https://man7.org/linux/man-pages/man8/ss.8.html

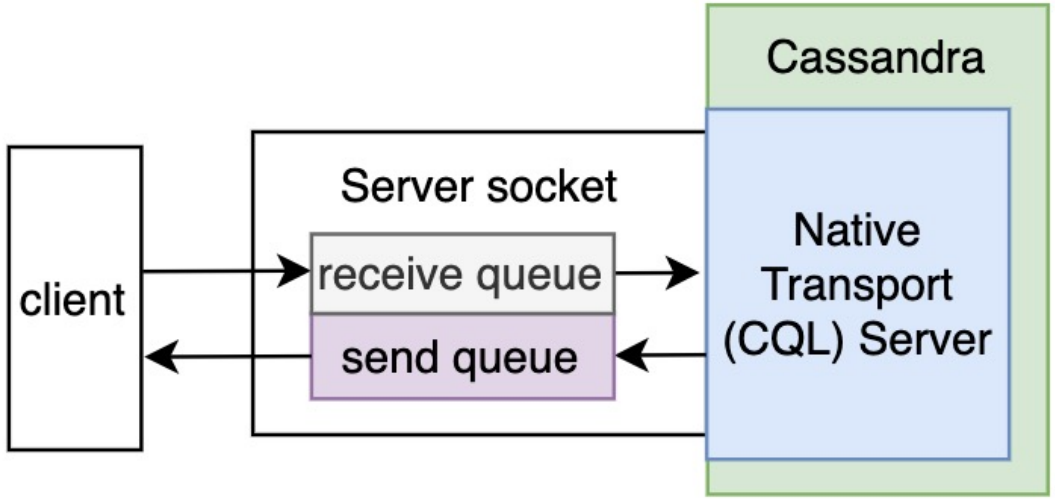
# Network - sockets

OS level:

net.core.rmem_max=16777216
net.core.wmem_max=16777216

net.core.rmem_default=16777216
net.core.wmem_default=16777216

net.ipv4.tcp_rmem='4096 87380 16777216'
net.ipv4.tcp_wmem='4096 65536 16777216'




from: https://docs.datastax.com/en/dse/6.8/docs/managing/configure/recommended-settings.html

Bonus, an interesting kernel issue:
https://blog.cloudflare.com/unbounded-memory-usage-by-tcp-for-receive-buffers-and-how-we-fixed-it

# Network - sockets

- Java:
  - Socket.setSendBufferSize(int size)
  - Socket.setReceiveBufferSize(int)

- Netty:
  - bootstrap.option(ChannelOption.SO_SNDBUF, ...);
  - bootstrap.childOption(ChannelOption.SO_RCVBUF, ...);

- Cassandra configuration:
  - internode_socket_receive_buffer_size
  - internode_socket_send_buffer_size
  - There are no such settings for CQL connections

# Network – number of CQL connections

- Cassandra allows to limit number of CQL connections (since 2.x, CASSANDRA-8086 )

- To configure (cassandra.yaml):
  - `native_transport_max_concurrent_connections (default: -1)`
  - `native_transport_max_concurrent_connections_per_ip (default: -1)`

- To monitor:
  - JMX: `org.apache.cassandra.metrics:type=Client name=connectedNativeClients`
  - Nodetool: `clientstats`
  - Virtual table: `select * from system_views.clients;`

# Network – number of CQL connections

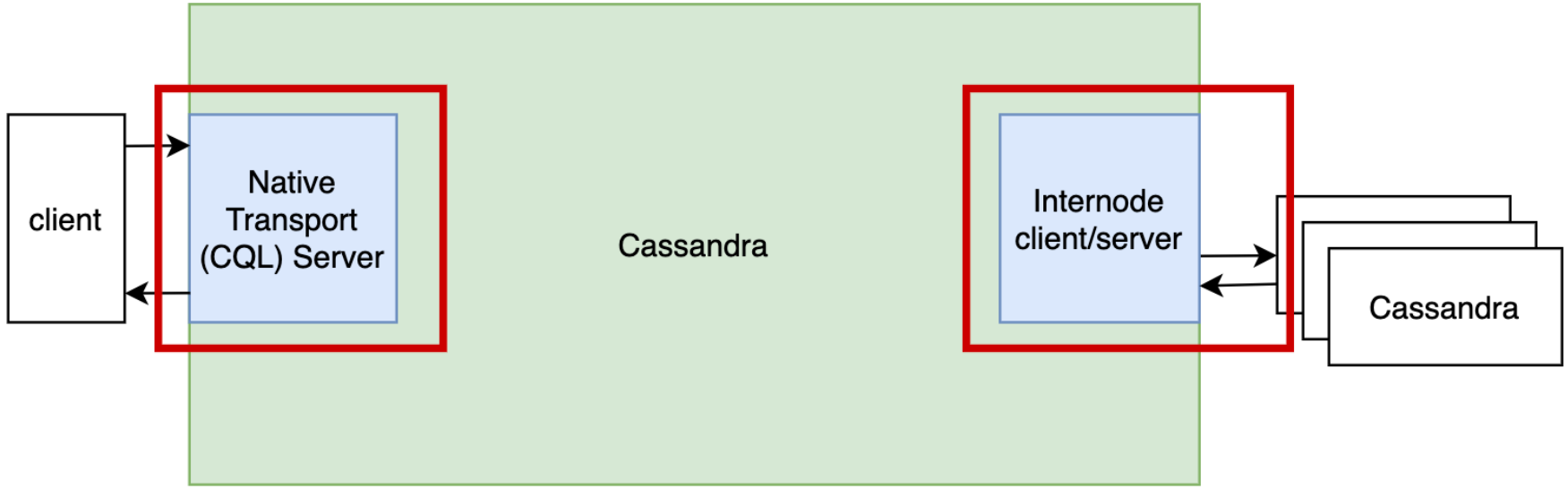
```
$ nodetool clientstats --all
Address          SSL   Cipher    Protocol  Version User      Keyspace  Requests Driver-Name    Driver-Version
/x.x.x.x:47610   false undefined undefined 5       cass_user           33870    DataStax Java  3.11.0
/y.y.y.y:54724   false undefined undefined 4       cass_user system    259      DataStax Java  3.7.1
...
Total connected clients: 116

User         Connections
Alice        23
Bob          2
...
```

# Interim summary

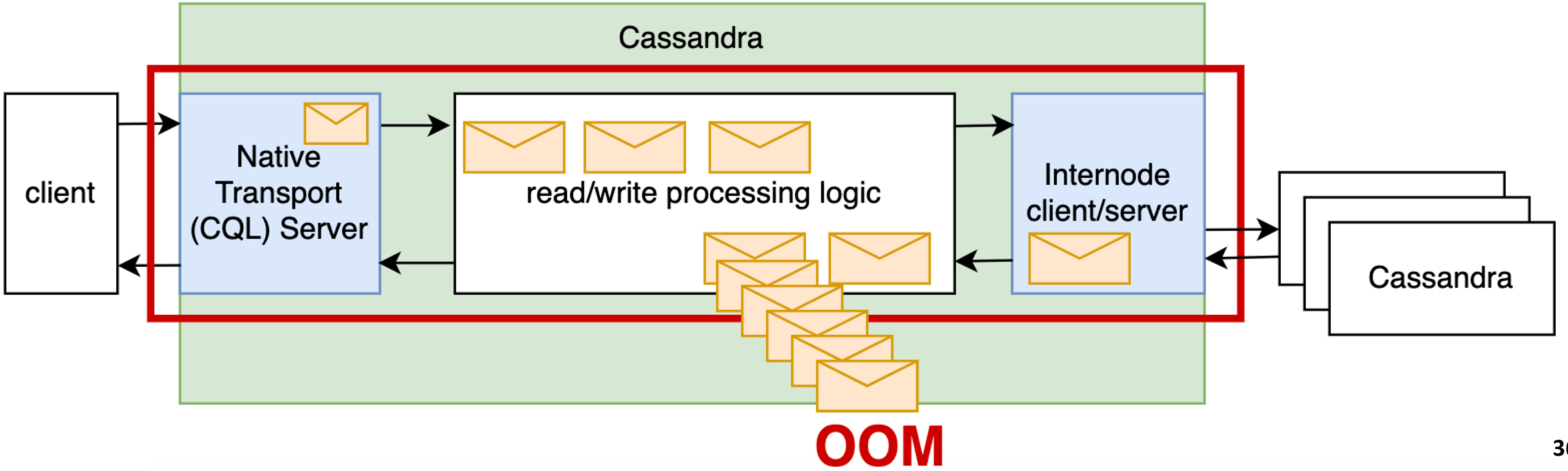
| Structures type | Heap/offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |


- JVM
- **Network**
  - Sockets
  - **Overload**



# Network - in-app memory consumption

Problem: too many concurrent requests may consume all memory and we will get an OOM



# Network - in-app memory consumption

Basic protection:

- native_transport_max_frame_size = 16 MiB (cannot be more than 256)
- native_transport_max_threads = 128

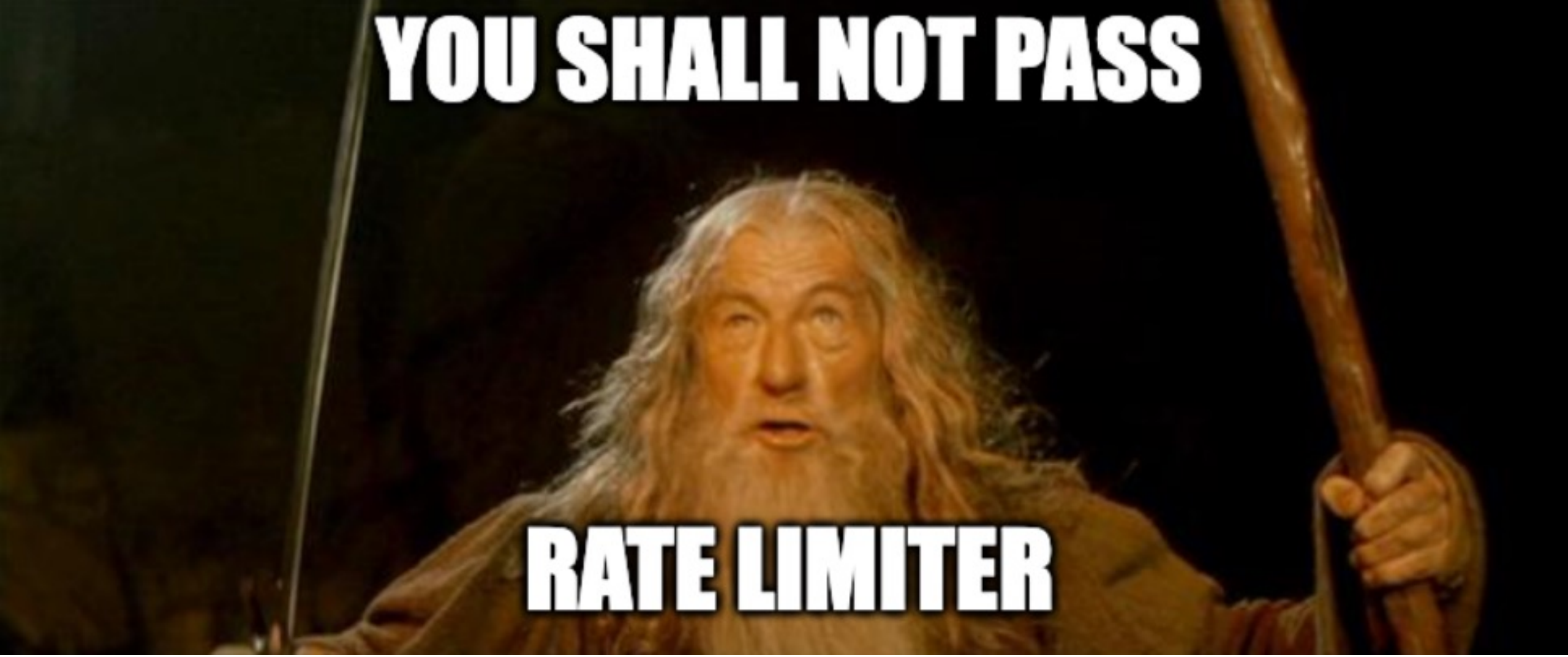
# Network - in-app memory consumption

Basic protection:

- native_transport_max_frame_size = 16 MiB (cannot be more than 256)
- native_transport_max_threads = 128

Not enough:

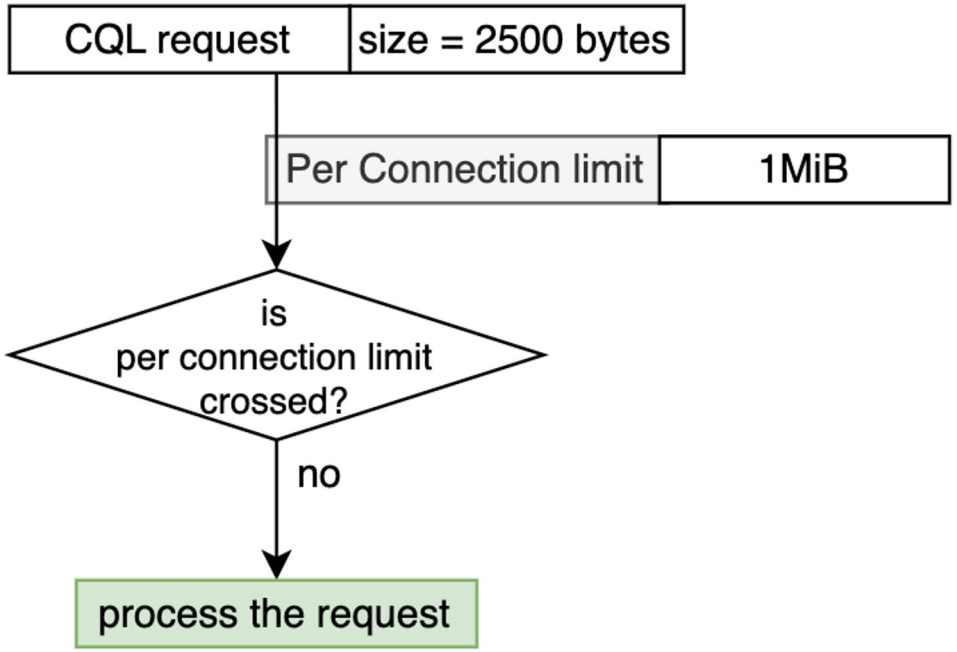
- Processing is complex and executed in multiple threads
- Requests may consume different amount of memory


YOU SHALL NOT PASS
RATE LIMITER

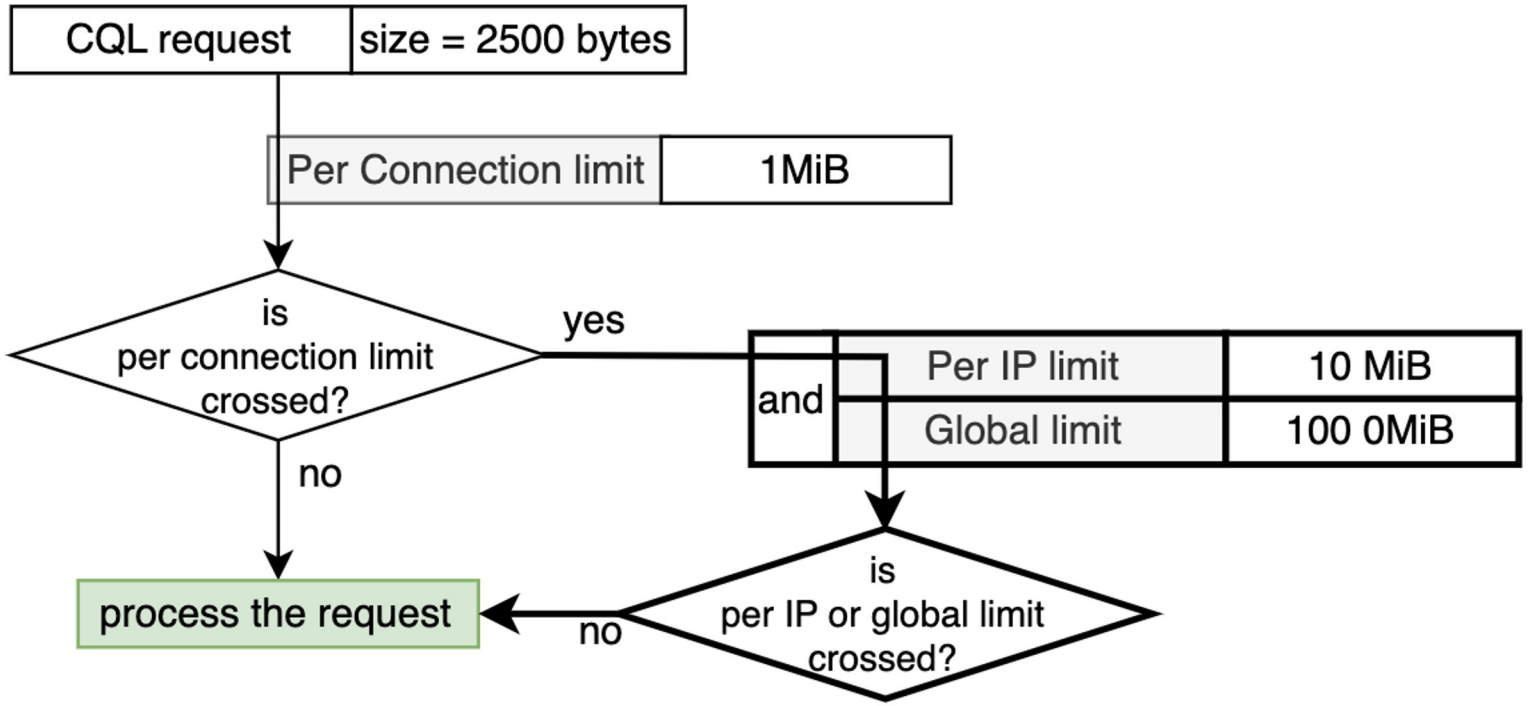
# Network - in-app memory consumption

- Rate limiting for in-flight request data
- Since 3.11.5, CASSANDRA-15013

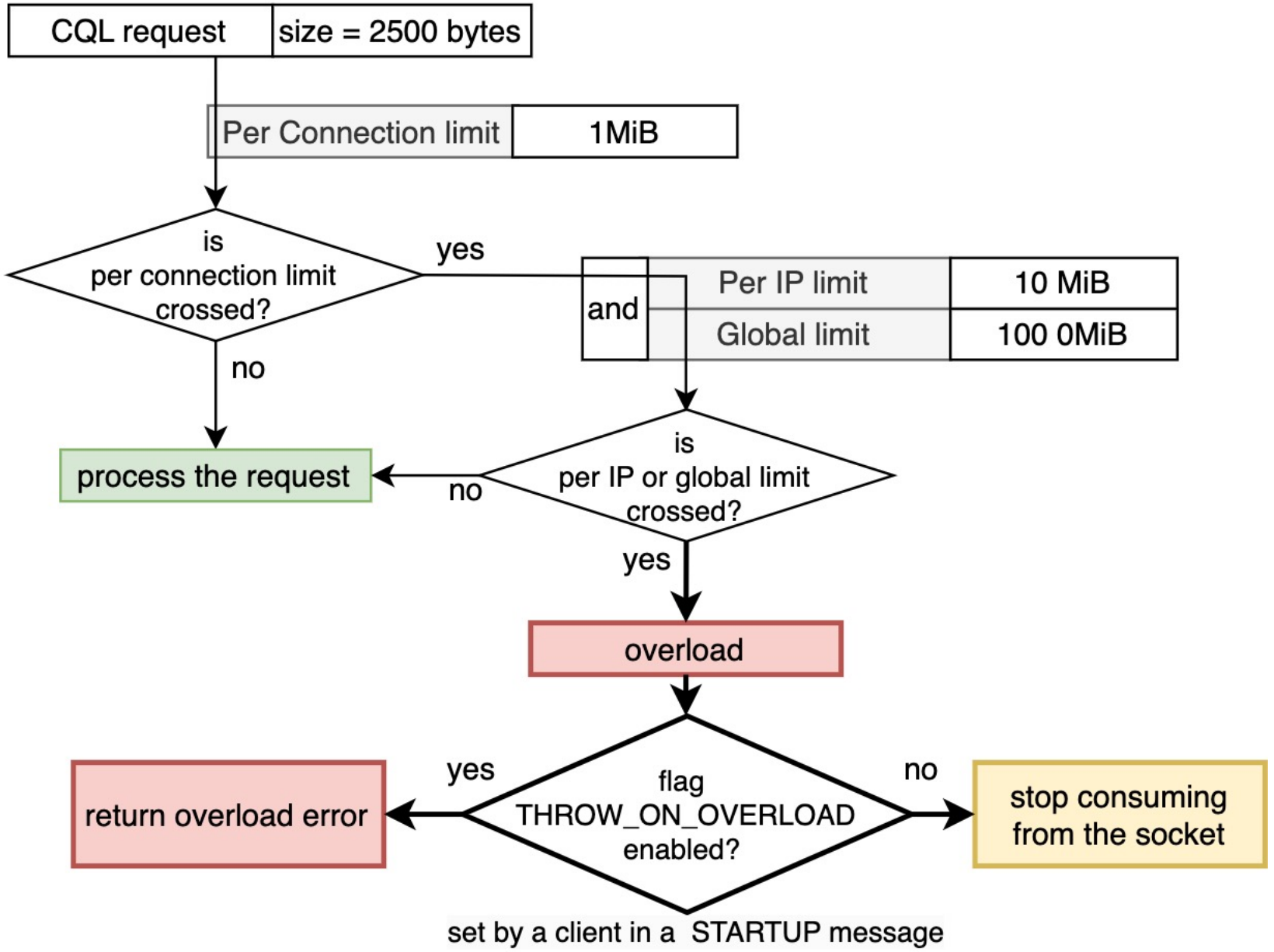
# Network - in-app memory consumption



# Network - in-app memory consumption



# Network - in-app memory consumption



# Network - in-app memory consumption

Configuration (cassandra.yaml):

- native_transport_receive_queue_capacity  (default: 1MiB)
- native_transport_max_request_data_in_flight_per_ip
- native_transport_max_request_data_in_flight

# Network - in-app memory consumption

Monitoring - JMX (`org.apache.cassandra.metrics:`):

- `type=Client,name=RequestDiscarded`
- `type=Client,name=RequestsSizeByIpDistribution`
- `type=Client,name=RequestsSize`
- `type=Client,name=PausedConnections`
- `type=ClientMessageSize,name=BytesReceived`
- `type=ClientMessageSize,name=BytesReceivedPerRequest`

# Network - in-app memory consumption

See also:

- https://www.instaclustr.com/blog/understanding-the-impacts-of-the-native-transport-requests-change-introduced-in-cassandra-3-11-5/ (perf tests)

- https://datastax-oss.atlassian.net/browse/JAVA-2589
  THROW_ON_OVERLOAD handling (not implemented)

# Network - in-app memory consumption

Similar logic exists for internode communications:

- internode_application_send_queue_capacity (default: 4 MiB)
- internode_application_send_queue_reserve_endpoint_capacity (default: 128 MiB)
- internode_application_send_queue_reserve_global_capacity (default: 512 MiB)

- internode_application_receive_queue_capacity (default: 4MiB)
- internode_application_receive_queue_reserve_endpoint_capacity (default: 128 MiB)
- internode_application_receive_queue_reserve_global_capacity (default: 512 MiB)

Metrics:

org.apache.cassandra.metrics:type=InboundConnection,scope=<remote host_port>,name=ReceivedCount
org.apache.cassandra.metrics:type=InboundConnection,scope=<remote host_port>,name=ThrottledCount
org.apache.cassandra.metrics:type=InboundConnection,scope=<remote host_port>,name=ThrottledNanos
…

# Network - in-app memory consumption

Classical rate limiting, as without it?

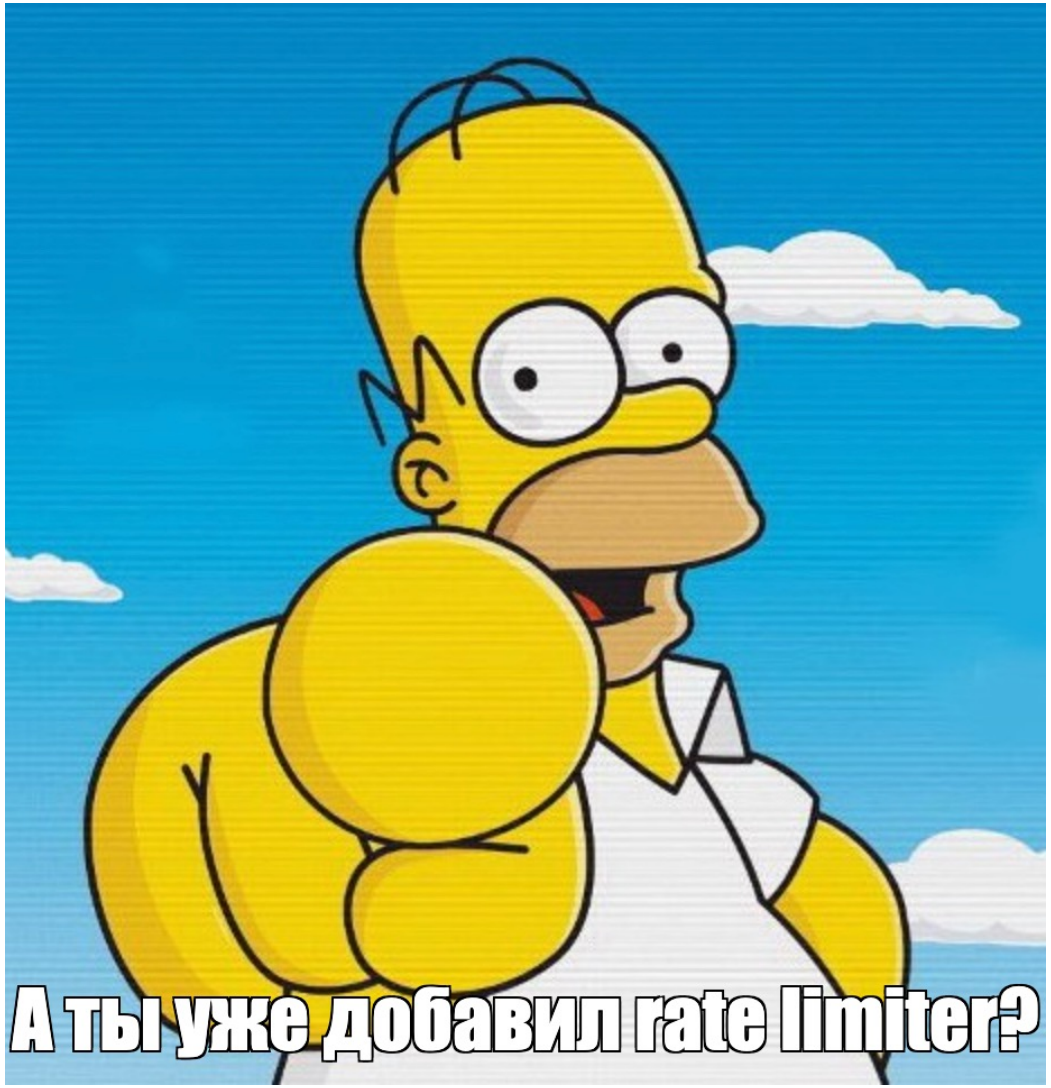
- CASSANDRA-16663 , since 4.1
- Configuration (cassandra.yml)
  - native_transport_rate_limiting_enabled (default: false)
  - native_transport_max_requests_per_second (default: 1’000’000)

# Network - in-app memory consumption

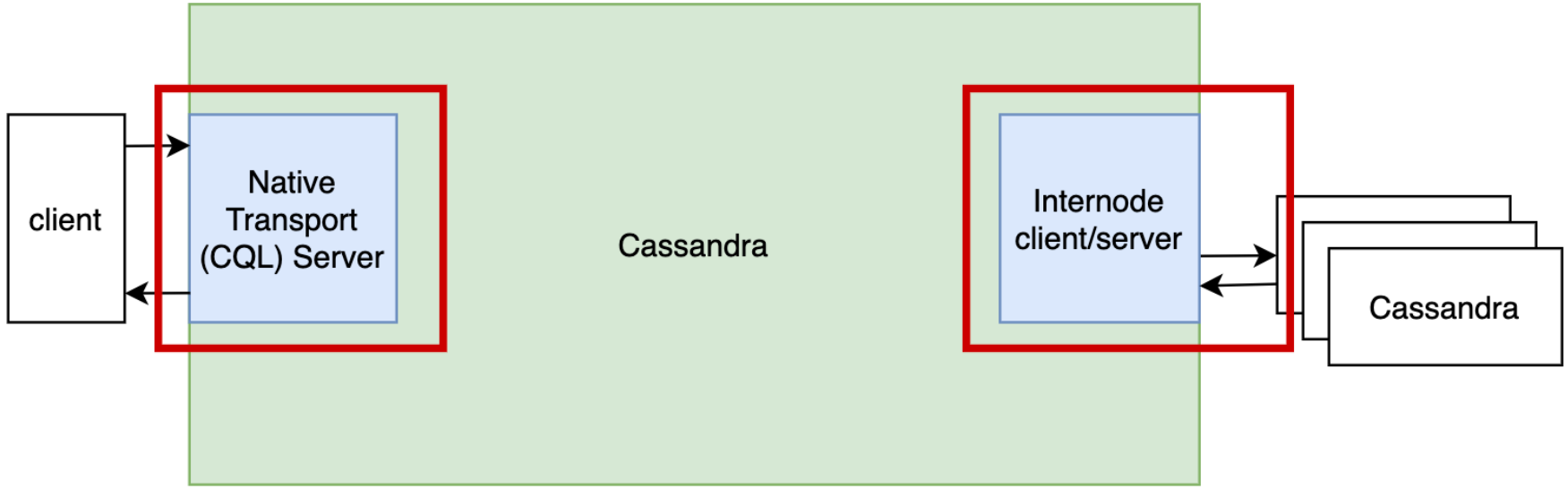
And it is still not the end, there are operation specific limits:

- Per write request
  - max_mutation_size
  - batch_size_fail_threshold
  - batch_size_warn_threshold
- Per read request
  - tombstone_warn_threshold
  - tombstone_failure_threshold
  - partition_keys_in_select_warn_threshold
  - partition_keys_in_select_fail_threshold
  - coordinator_read_size_warn_threshold
  - coordinator_read_size_fail_threshold
  - Page size (4.1.x, CASSANDRA-17189)
    - page_size: warn_threshold: -1
    - abort_threshold: -1
  - CASSANDRA-16896

# Interim summary




- JVM
- **Network**
  - Sockets
  - Overload
  - **Memory usage**



# Others - ?

| Category | Size (committed) | % of heap |
|---|---|---|
| Heap | 8 GiB | 100% |
| Others | 967 MiB | 12% |
| GC | 384 MiB | 5% |
| Class | 71 MiB | |
| Thread | 54 MiB | |
| Code | 53 MiB | |
| Symbol | 13 MiB | |
| Internal | 2 MiB | |
| Compiler | 1 MiB | |
| Total, non-heap | 1550 MiB | 19% |

What is it?

# Others - ?

| Category | Size (committed) | % of heap |
|---|---|---|
| Heap | 8 GiB | 100% |
| Others | 967 MiB | 12% |
| GC | 384 MiB | 5% |
| Class | 71 MiB | |
| Thread | 54 MiB | |
| Code | 53 MiB | |
| Symbol | 13 MiB | |
| Internal | 2 MiB | |
| Compiler | 1 MiB | |
| Total, non-heap | 1550 MiB | 19% |


Unsafe.allocateMemory
DirectByteBuffer

# Others - ?

- Let’s analyze it in more details
- Options:
  - NMT: Others section
  - JMX Metric: MemoryUsed in java.nio:type=BufferPool,name=direct
  - Heap dump: find DirectByteBuffer

# Others – JMX metric

```
nodetool sjk mxdump -q java.nio:type=BufferPool,name=direct
{
  "beans" : [ {
    "name" : "java.nio:name=direct,type=BufferPool",
    "Count" : 179,
    "TotalCapacity" : 57098341,
    "MemoryUsed" : 57098343,
    "Name" : "direct",
    "ObjectName" : "java.nio:type=BufferPool,name=direct"
  } ]
}
```

JSK (Java Swiss Knife) - https://github.com/aragozin/jvm-tools

# Others – heap dump

Find all DirectByteBuffers in a heap dump using Eclipse Memory Analyzer

```
SELECT capacity FROM java.nio.DirectByteBuffer
```

| capacity |
| --- |
| <Numeric> |
| 142,478,673 |
| 33,554,432 |
| 33,554,432 |
| 33,554,432 |
| 16,777,216 |
| … |

**X 56**

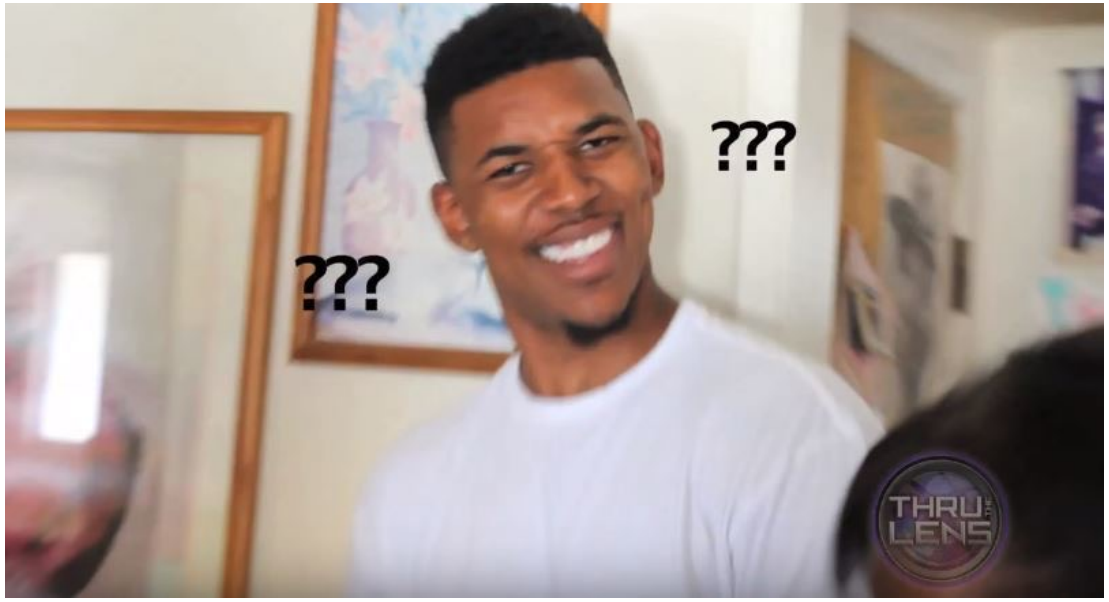
Total: 1199 MiB

# Others – ?

NMT: 967 MiB

JMX: 54.4 MiB

Heap dump: 1199 MiB




What is the real memory consumption by Direct Byte Buffers ??!

# Others – heap dump

```
SELECT capacity FROM java.nio.DirectByteBuffer
```

| capacity |
| --- |
| <Numeric> |
| 142,478,673 |
| 33,554,432 |
| 33,554,432 |
| 33,554,432 |
| 16,777,216 |
| ... |

**X 56**

# Others – JImage DirectByteBuffer

| Class Name | Ref. Objects | Shallow Heap | Ref. Shallow Heap | Retained Heap |
|---|---|---|---|---|
| <Regex> | <Numeric> | <Numeric> | <Numeric> | <Numeric> |
| class jdk.internal.module.SystemModuleFinders$SystemImage @ 0x600585c48 Systen | 1 | 8 | 64 | 8 |
| **READER** jdk.internal.jimage.ImageReader @ 0x600586f50 | 1 | 24 | 64 | 24 |
| **reader** jdk.internal.jimage.ImageReader$SharedImageReader @ 0x600586bc0 | 1 | 96 | 64 | 944 |
| **memoryMap** java.nio.DirectByteBufferR @ 0x600586c30 | 1 | 64 | 64 | 128 |
| **att** java.nio.DirectByteBuffer @ 0x600586c70 | 1 | 64 | 64 | 64 |

Statics | Attributes | Class Hierarchy | Value

| Type | Name | Value |
|---|---|---|
| ref | cleaner | null |
| ref | att | null |
| ref | fd | null |
| boolean | nativeByteOrder | false |
| boolean | bigEndian | true |
| boolean | isReadOnly | false |
| int | offset | 0 |
| ref | hb | null |
| long | address | 140478114242560 |
| int | capacity | 142478673 |
| int | limit | 28 |
| int | position | 0 |
| int | mark | -1 |

# Others – JImage DirectByteBuffer

| Class Name | Ref. Objects | Shallow Heap | Ref. Shallow Heap | Retained Heap |
|---|---|---|---|---|
| <Regex> | <Numeric> | <Numeric> | <Numeric> | <Numeric> |
| class jdk.internal.module.SystemModuleFinders$SystemImage @ 0x600585c48 Systen | 1 | 8 | 64 | 8 |
| **READER** jdk.internal.jimage.ImageReader @ 0x600586f50 | 1 | 24 | 64 | 24 |
| **reader** jdk.internal.jimage.ImageReader$SharedImageReader @ 0x600586bc0 | 1 | 96 | 64 | 944 |
| **memoryMap** java.nio.DirectByteBufferR @ 0x600586c30 | 1 | 64 | 64 | 128 |
| **att** java.nio.DirectByteBuffer @ 0x600586c70 | 1 | 64 | 64 | 64 |

- src/java.base/share/classes/jdk/internal/jimage/BasicImageReader.java - L73
- src/java.base/share/native/libjimage/NativeImageBuffer.cpp - L55
- src/java.base/share/native/libjimage/imageFile.cpp - L400

```
[host] ls -all jdk-11.0.15+10/lib/modules
-rw-r--r- 1 142478673 jdk-11.0.15+10/lib/modules
```

It is actually a memory mapped buffer

Statics | Attributes | Class Hierarchy | Value

| Type | Name | Value |
|---|---|---|
| ref | cleaner | null |
| ref | att | null |
| ref | fd | null |
| boolean | nativeByteOrder | false |
| boolean | bigEndian | true |
| boolean | isReadOnly | false |
| int | offset | 0 |
| ref | hb | null |
| long | address | 140478114242560 |
| int | capacity | 142478673 |
| int | limit | 28 |
| int | position | 0 |
| int | mark | -1 |

# Others – heap dump

```
SELECT capacity FROM java.nio.DirectByteBuffer
```

| capacity |
| --- |
| <Numeric> |
| ~~142,478,673~~ |
| 33,554,432 |
| 33,554,432 |
| 33,554,432 |
| 16,777,216 |
| ... |

Не считается, это ~~не настоящая память~~ на самом деле memory-mapped buffer

**X 56**

# Others – heap dump

```
SELECT capacity FROM java.nio.DirectByteBuffer
```

| capacity |
| --- |
| <Numeric> |
| 142,478,673 |
| 33,554,432 |
| 33,554,432 |
| 33,554,432 |
| 16,777,216 |
| ... |

**X 56**

# Others – Netty pool

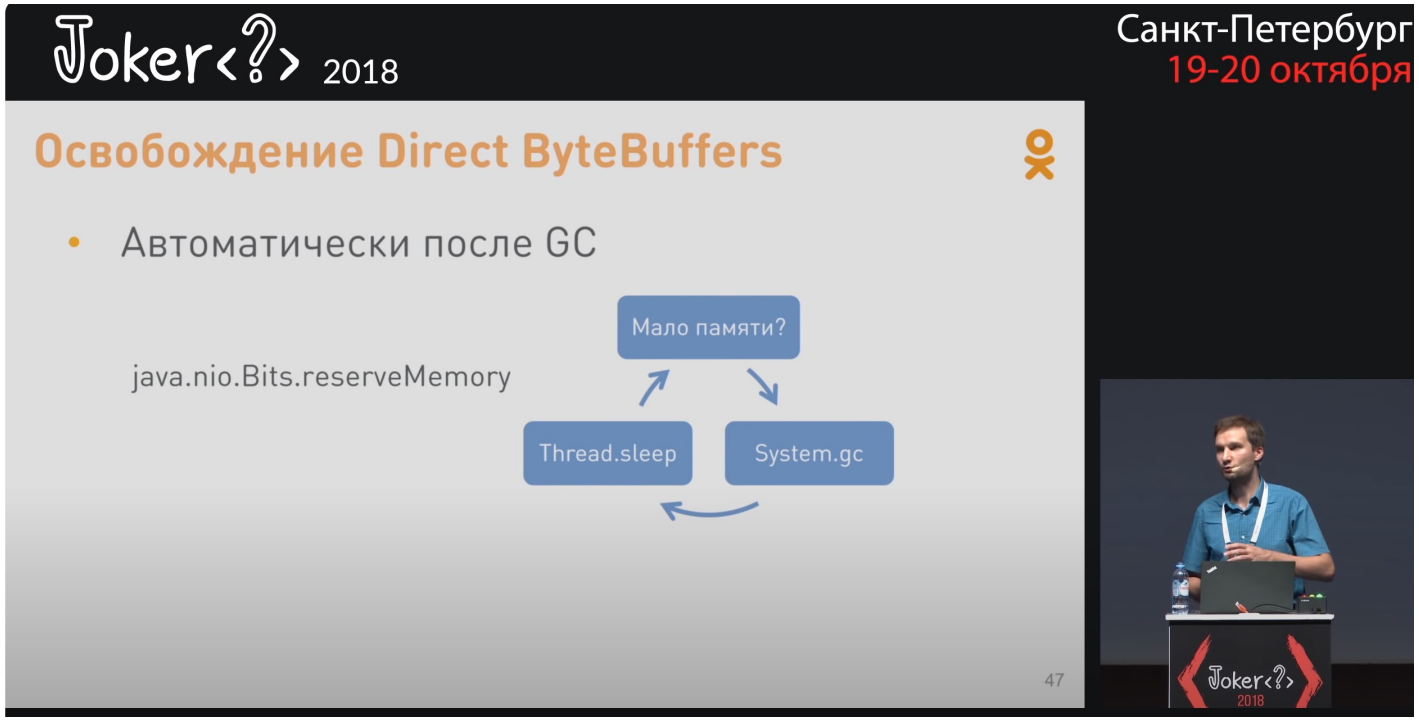
| Class Name | Shallow Heap | Retained Heap |
|---|---|---|
| <Regex> | <Numeric> | <Numeric> |
| java.nio.DirectByteBuffer @ 0x601ca8648 | 64 | 64 |
| **memory** io.netty.buffer.PoolChunk @ 0x601ca8600 | 72 | 13,392 |
| **head** io.netty.buffer.PoolChunkList @ 0x6038ab618 | 48 | 48 |
| **qInit** io.netty.buffer.PoolArena$DirectArena @ 0x6038a9da8 | 160 | 7,192 |
| **directArena** io.netty.buffer.PoolThreadCache @ 0x601599968 | 48 | 138,592 |
| **[36]** java.lang.Object[64] @ 0x601be0080 | 272 | 1,864 |
| **indexedVariables** io.netty.util.internal.InternalThreadLocalMap @ 0x6 | 136 | 2,000 |
| **threadLocalMap** io.netty.util.concurrent.FastThreadLocalThread @ | 120 | 3,448 |
| **directArena** io.netty.buffer.PoolThreadCache @ 0x601b363e8 | 48 | 139,096 |
| **[12]** io.netty.buffer.PoolArena[56] @ 0x603948a38 | 240 | 240 |
| **Total: 3 entries** | | |

# Others – Netty pool

- Why do we need a pool – just allocate and GC will collect it ?!

# Others – Netty pool

- Why do we need a pool – just allocate and GC will collect it ?!
- Expensive to allocate and collect by GC



# Network - in-app memory consumption

- **PooledByteBufAllocator -** Netty byte buffer allocator
- Pools direct byte buffers (Netty specific: ByteBuf)

# Network - in-app memory consumption

- **PooledByteBufAllocator -** Netty byte buffer allocator
- Pools direct byte buffers (Netty specific: ByteBuf)
- Memory is allocated using Unsafe.allocateMemory -> not visible via JMX for direct byte buffer pool
- io.netty.util.internal.PlatformDependent#usedDirectMemory

# Network - in-app memory consumption

- **PooledByteBufAllocator -** Netty byte buffer allocator
- Pools direct byte buffers (Netty specific: ByteBuf)
- Memory is allocated using Unsafe.allocateMemory -> not visible via JMX for direct byte buffer pool
- io.netty.util.internal.PlatformDependent#usedDirectMemory
- 28 cores => 2 arenas per core = 56 pool chunks x 16 MiB = **896 MiB**

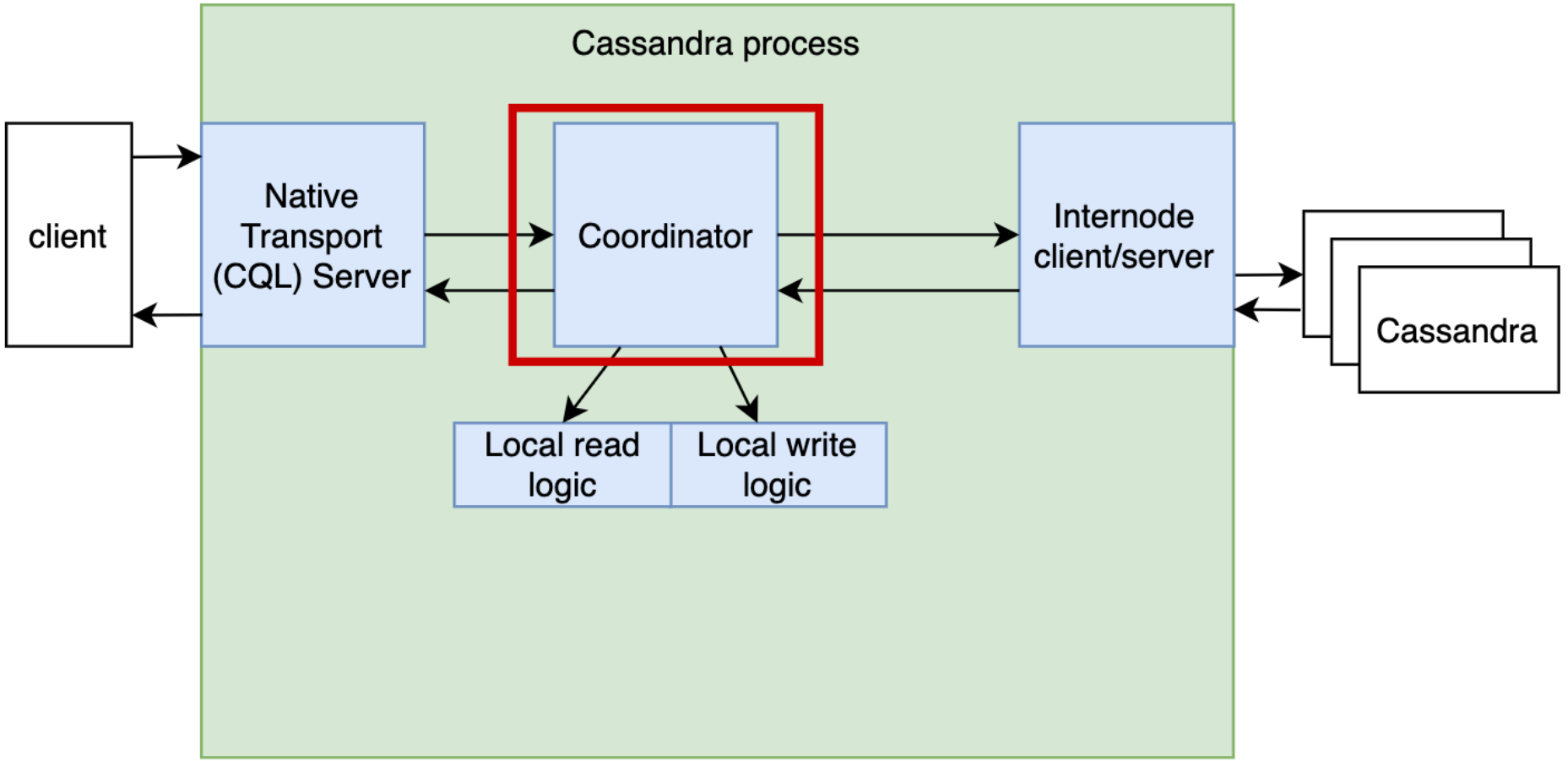
# Network – Netty allocator

- https://speakerdeck.com/trustin/buffer-allocation-and-leak-detection-in-netty
- Netty, the IO framework that propels them all By Stephane LANDELLE
- https://github.com/netty/netty/issues/3910

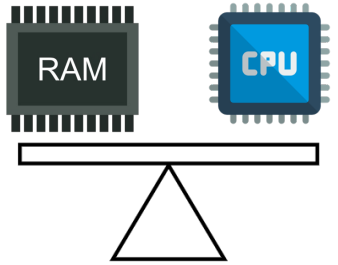
# Interim summary

| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |


- JVM
- Network
- **Coordinator**

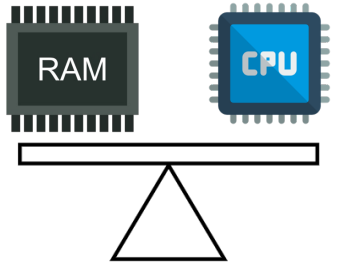


# Coordinator – prepared statement cache




- Parsing of CQL statement takes time and CPU
- Cassandra allows to prepare a CQL statement and store a result into a local memory cache

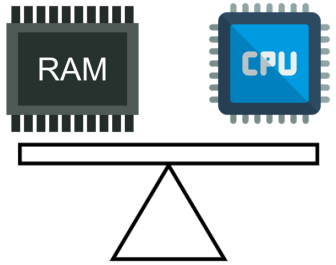
# Coordinator – prepared statement cache




- Parsing of CQL statement takes time and CPU
- Cassandra allows to prepare a CQL statement and store a result into a local memory cache
- Key = MD5 digest (CQL statement text)
- Stored in heap, Caffeine cache library

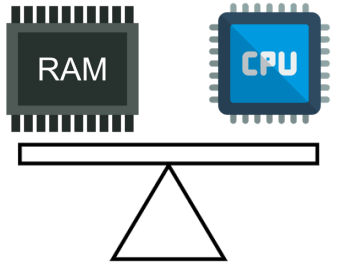
| Caffeine cache | |
|---|---|
| Key | Value |
| MD5 Digest | Parsed statement |

# Coordinator – prepared statement cache



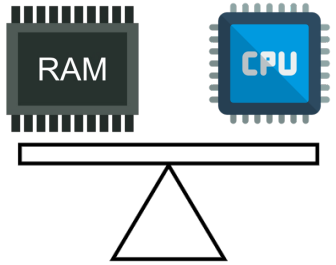

- Caffeine cache library (https://github.com/ben-manes/caffeine)
- Mature
- Sized-based eviction
- Advanced caching algorithm (TinyLRU)
- High performance
- Check https://github.com/ben-manes/caffeine?tab=readme-ov-file#in-the-news for more details

# Coordinator – prepared statement cache




- prepared_statements_cache_size (default: 32 MiB, usually is ok)
- Average cache entry size ~ 10-20 KiB => ~1600-3200 statements

# Coordinator – prepared statement cache

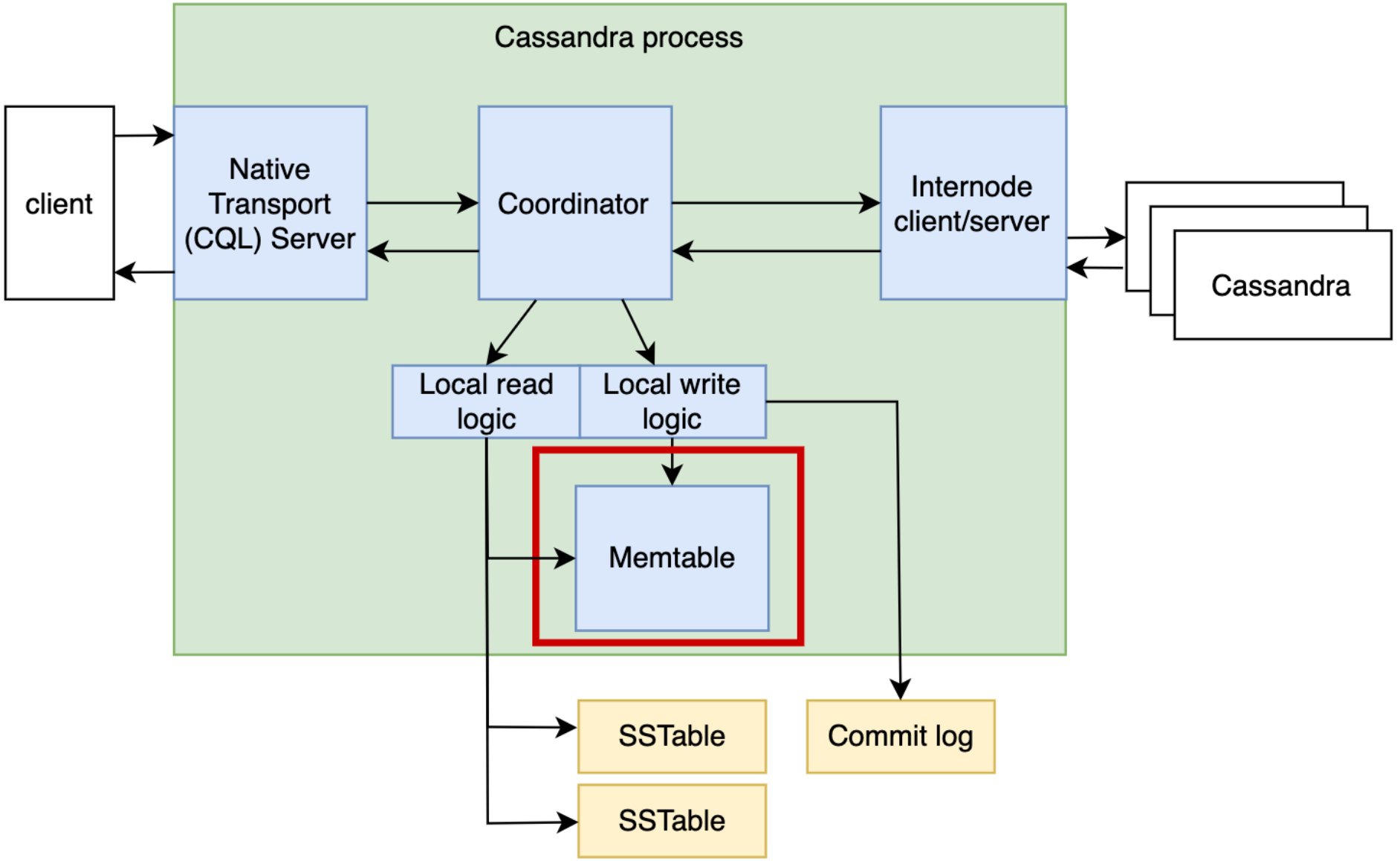



- prepared_statements_cache_size (default: 32 MiB, usually is ok)
- Average cache entry size ~ 10-20 KiB => ~1600-3200 statements
- JMX metrics:
  - PreparedStatementsCount
  - PreparedStatementsEvicted
  - RegularStatementsExecuted
- select * from system.prepared_statements;

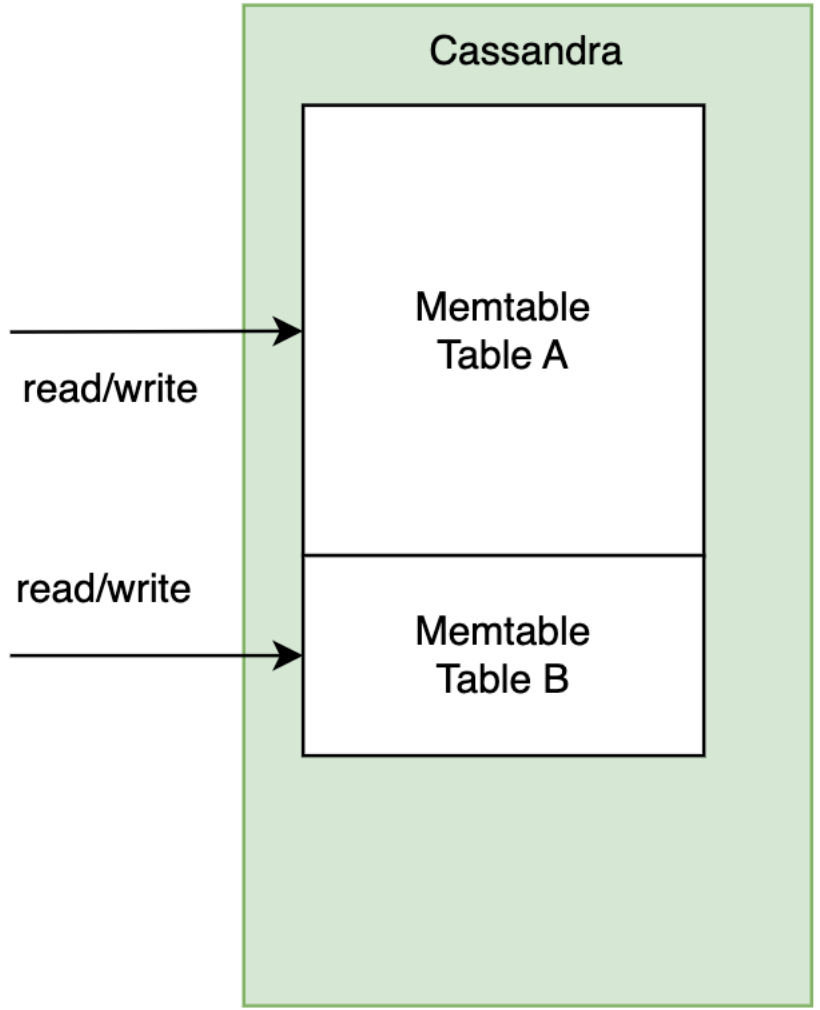
# Interim summary

| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1% | cassandra.yaml | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

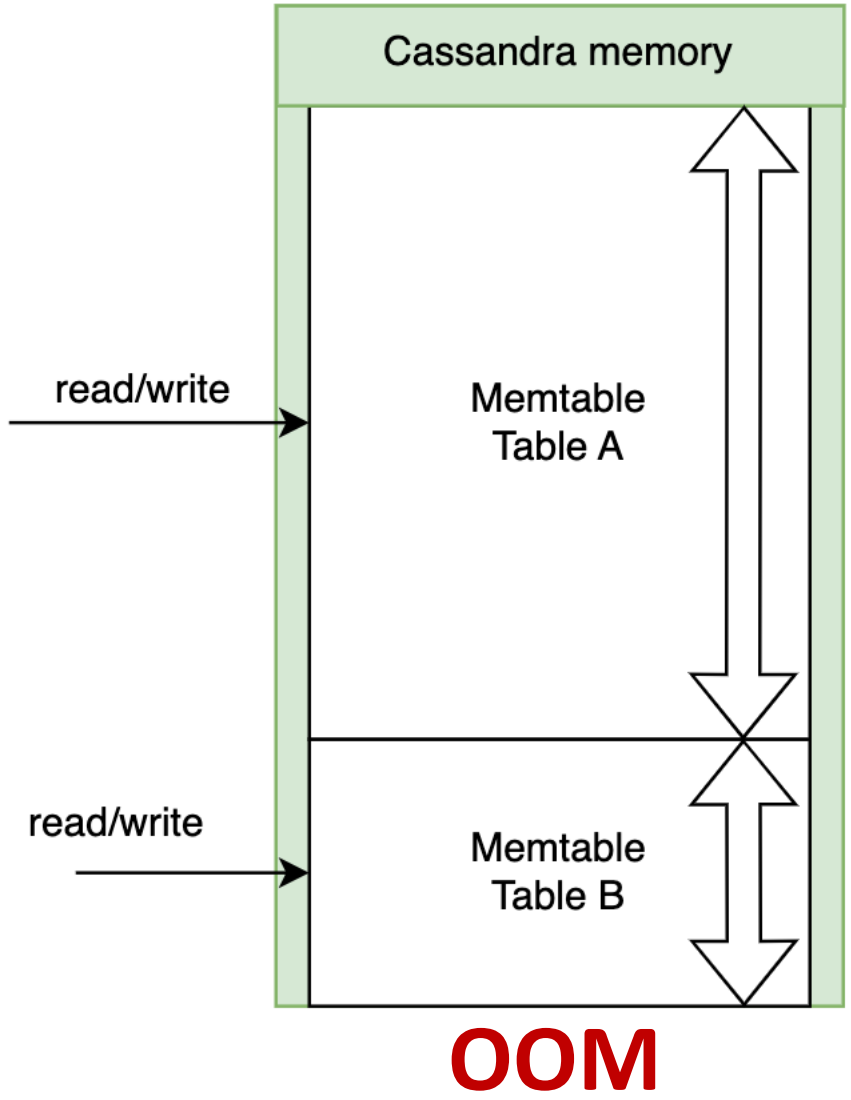

- JVM
- Network
- Coordinator
- **Memtables**
  - **Lifecycle**



# Memtable lifecycle



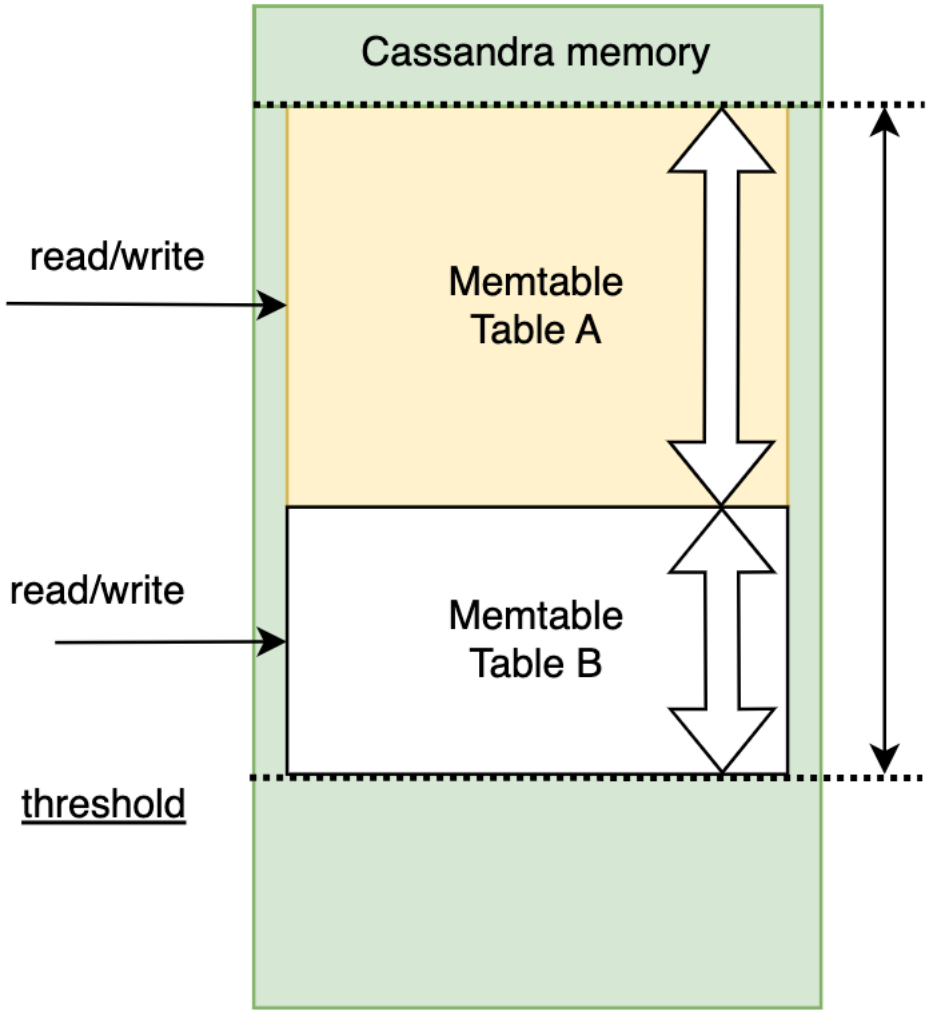
# Memtable lifecycle



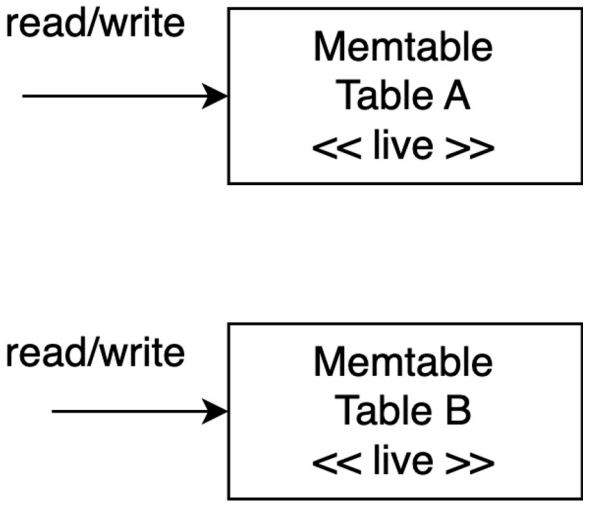
# Memtable lifecycle

- Define a memory threshold
- Track memory usage
  - per memtable
  - sum
- When sum > threshold

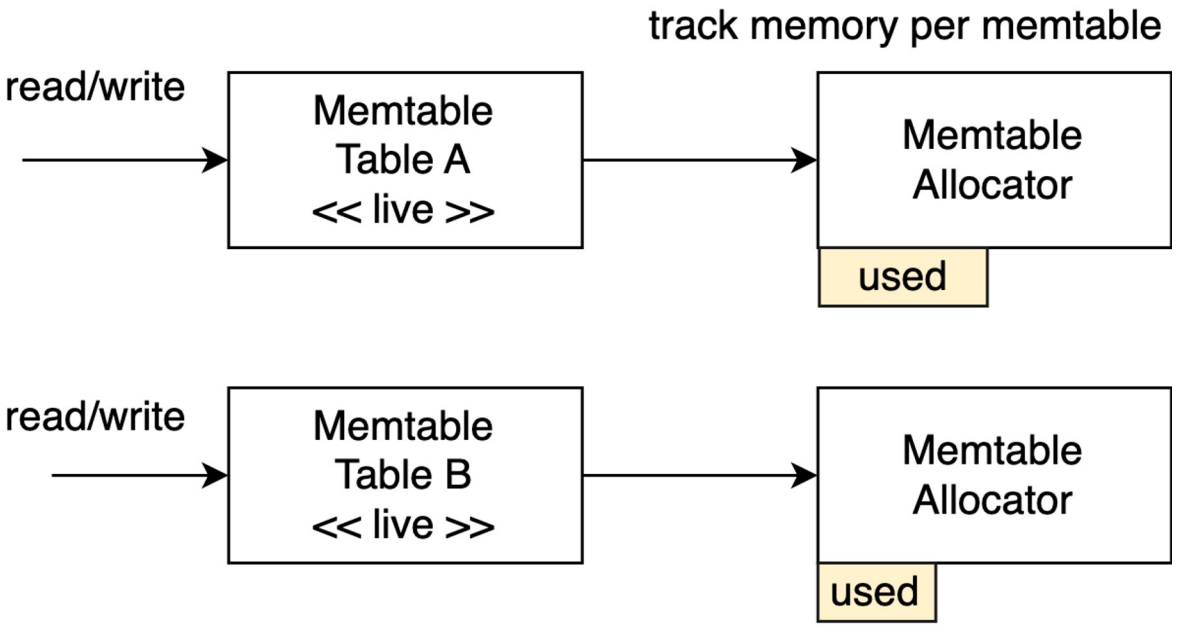
  flush the largest memtable to disk



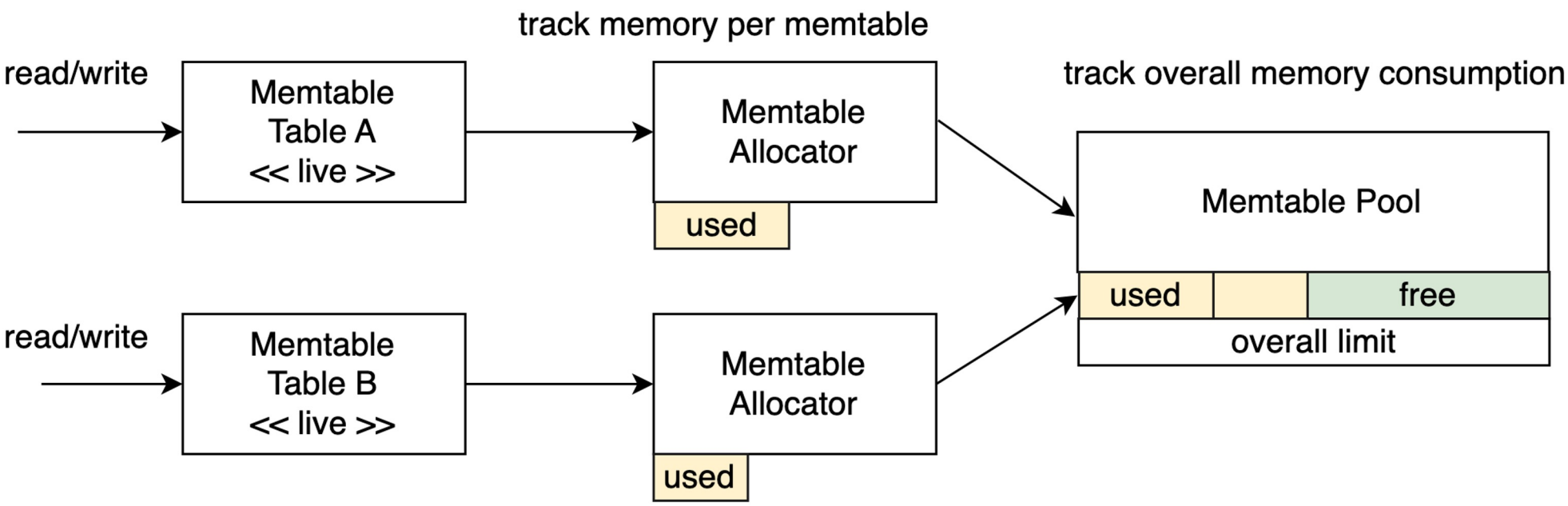
# Memtable lifecycle



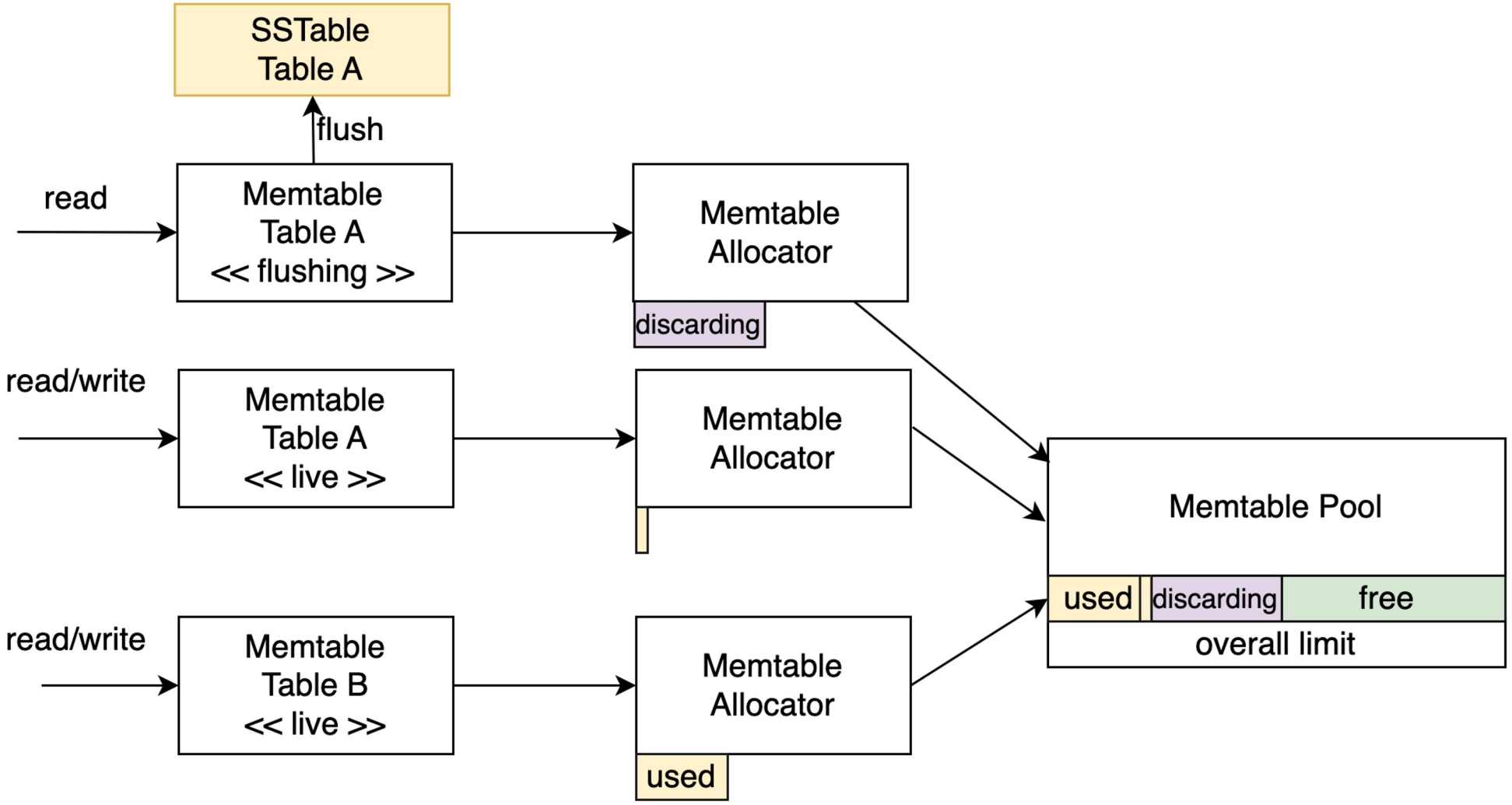
# Memtable lifecycle



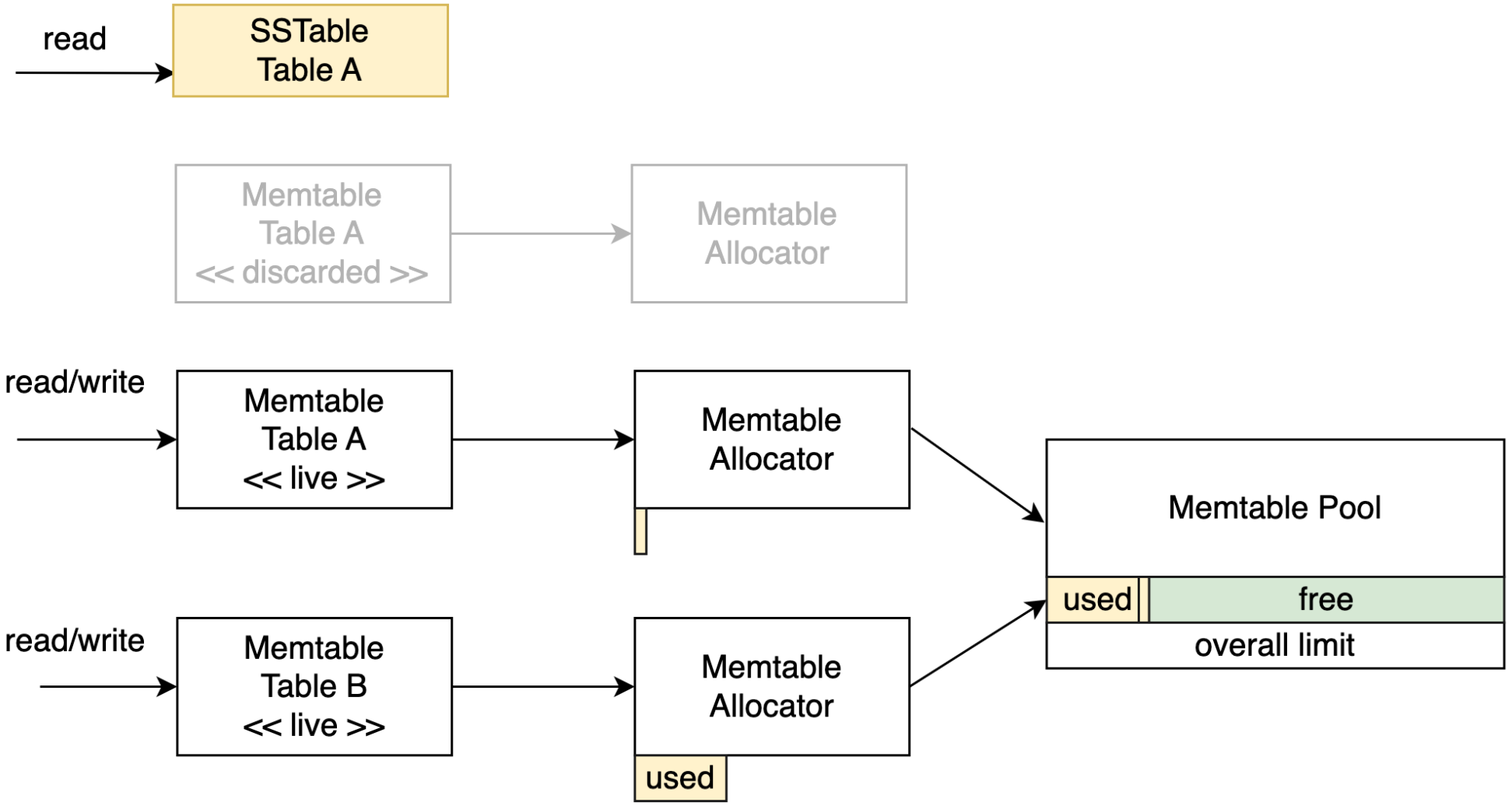
# Memtable lifecycle



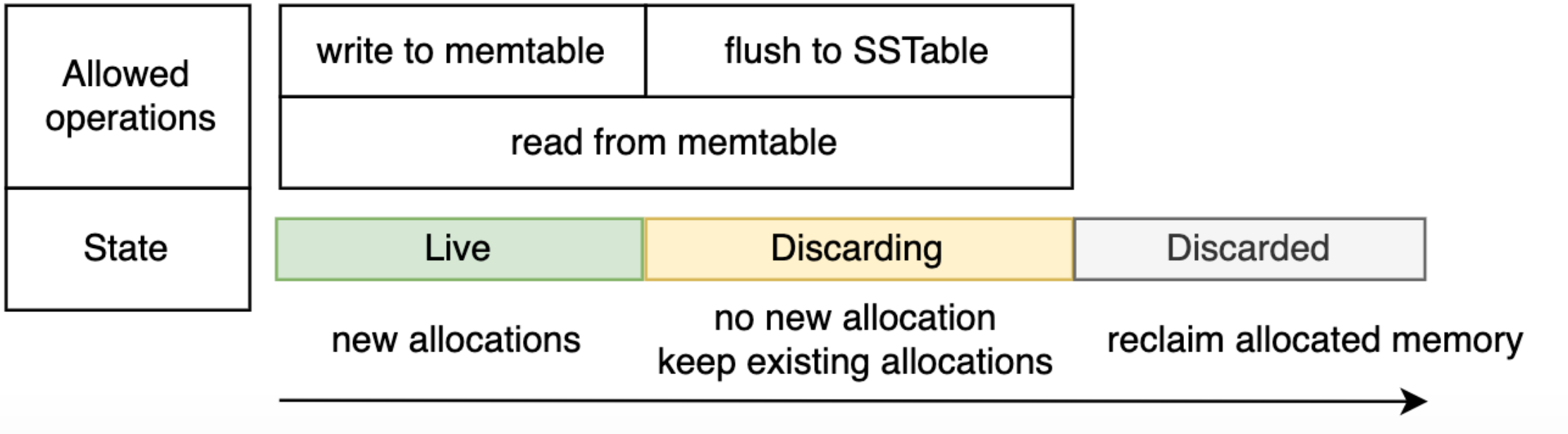
# Memtable lifecycle



# Memtable lifecycle

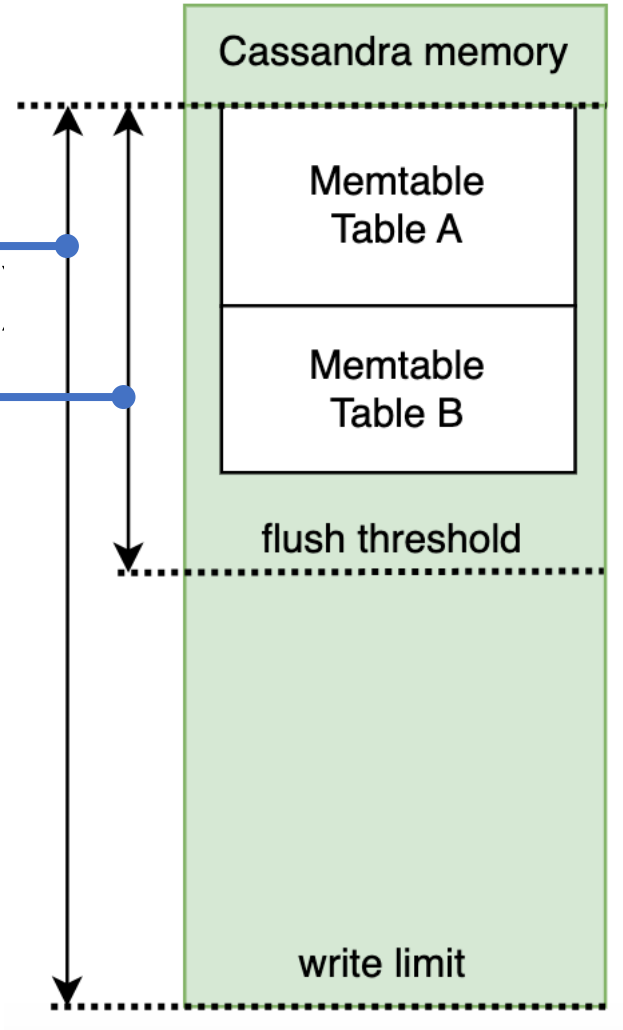


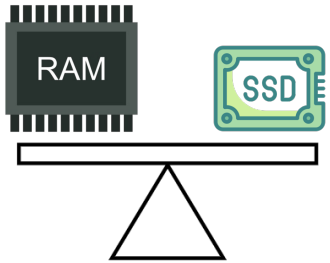
# Memtable lifecycle



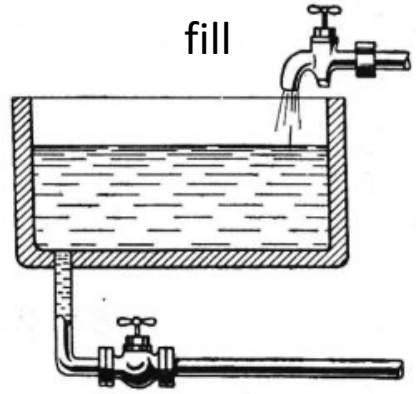
# Memtable lifecycle

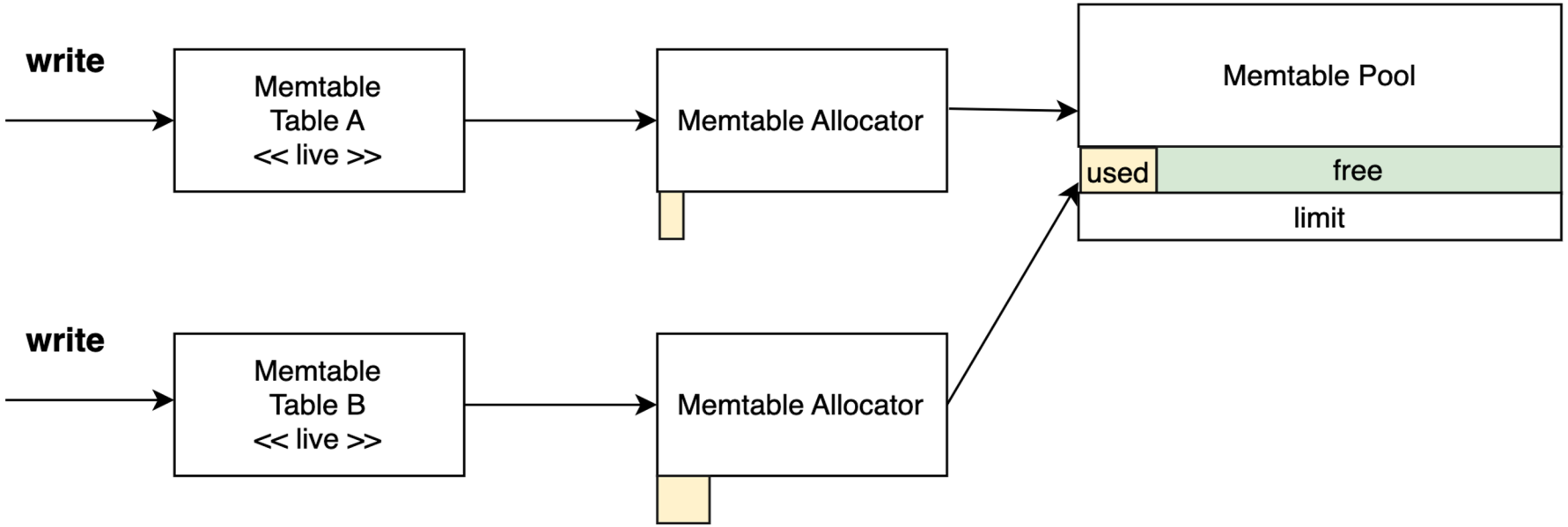
- memtable_heap_space (default: 1/4 of heap size)
- memtable_offheap_space (default: 1/4 of heap size)
- memtable_cleanup_threshold
  (default: 1 + memtable_flush_writers = 11%)



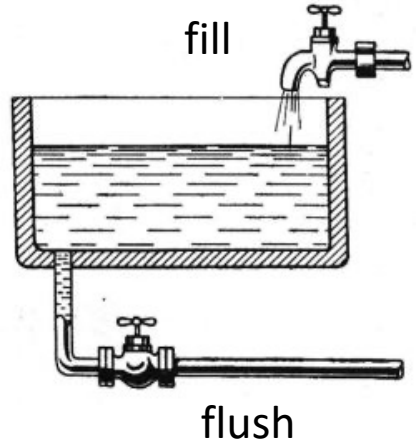



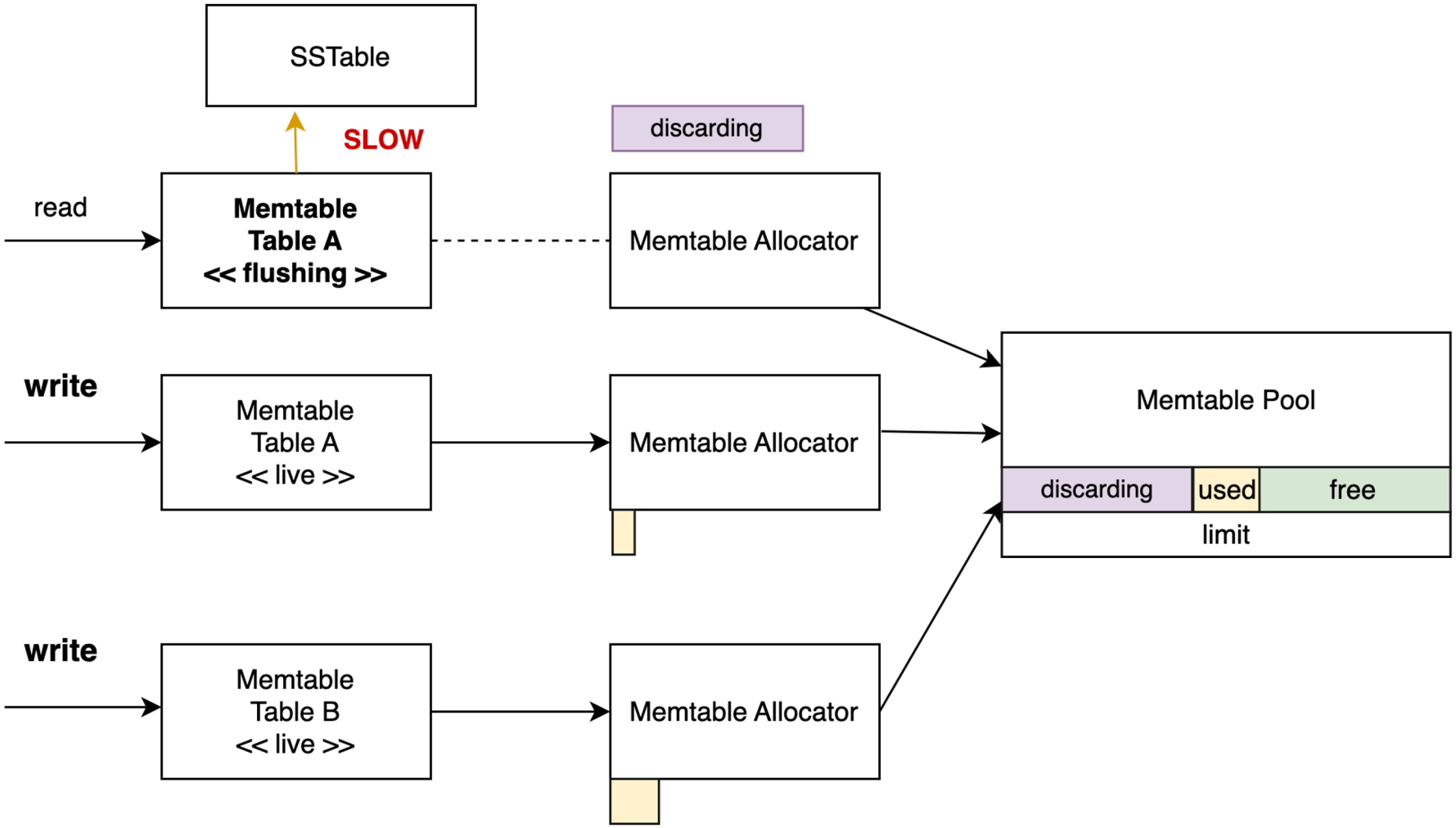
# Memtable lifecycle – slow flush



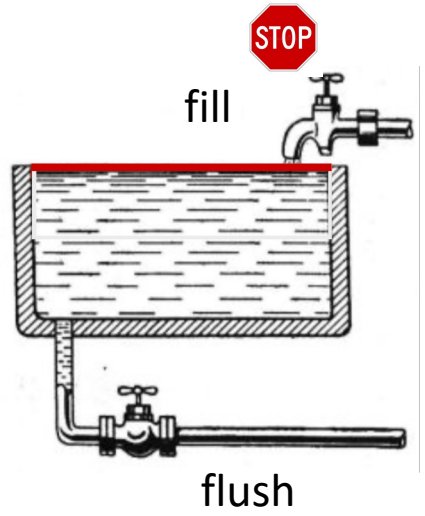



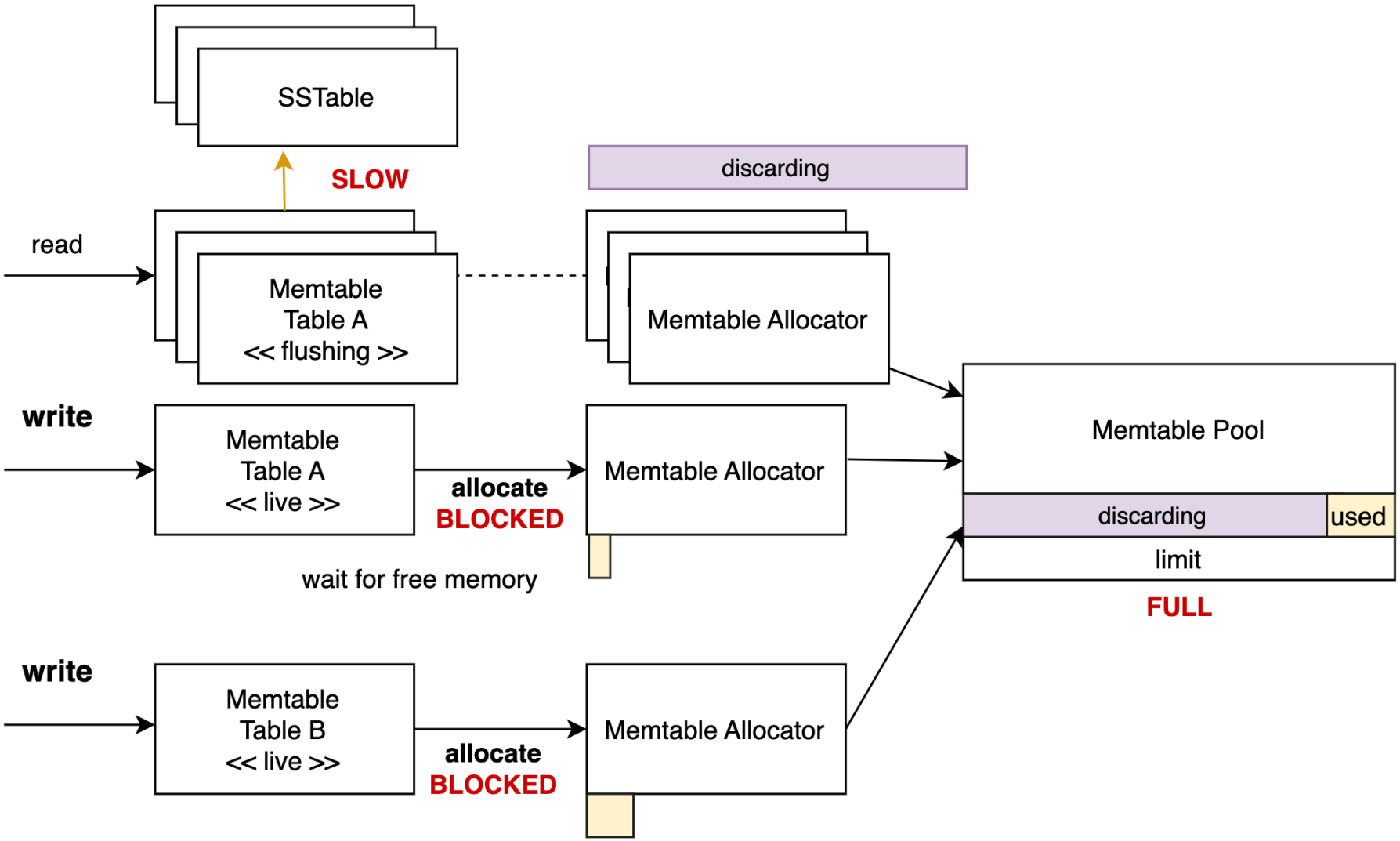
# Memtable lifecycle – slow flush



# Memtable lifecycle – slow flush

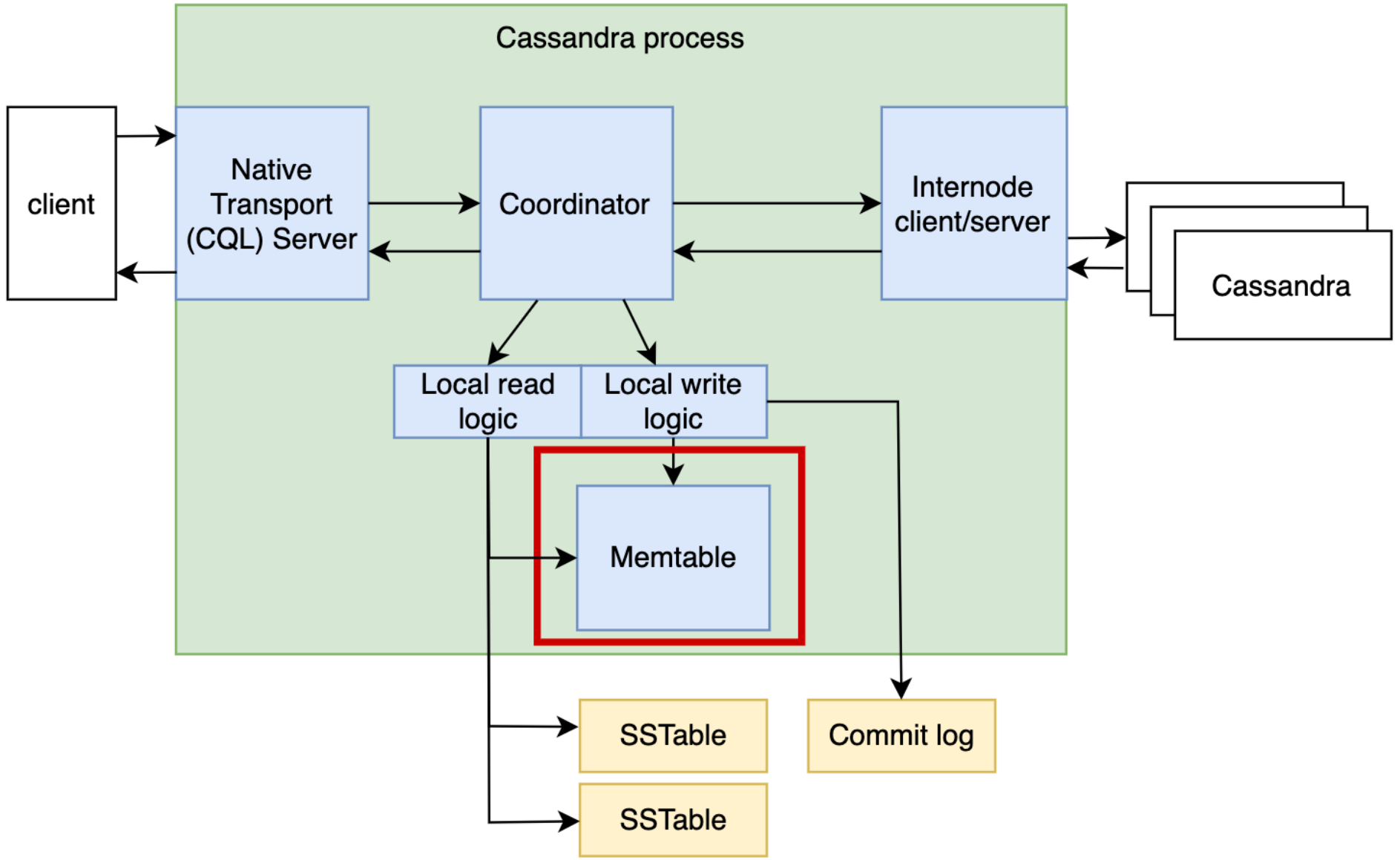


# Memtable lifecycle – blocked writes

- It is a kind of backpressure
- Too high memtable_cleanup_threshold is dangerous
- Important JMX metrics to track:
  - BlockedOnAllocation
  - PendingFlushTasks


- JVM
- Network
- Coordinator
- **Memtables**
  - Lifecycle
  - **Allocation**



# Memtable - allocation

```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1, VALUE_2)
VALUES ('PART_KEY_1', 'CUST_KEY_1', 'VALUE_A', 'VALUE_B');
```

```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1)
VALUES ('PART_KEY_1', 'CUST_KEY_2', 'VALUE_C');
```

# Memtable - allocation

```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1, VALUE_2)
VALUES ('PART_KEY_1', 'CUST_KEY_1', 'VALUE_A', 'VALUE_B');
```

```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1)
VALUES ('PART_KEY_1', 'CUST_KEY_2', 'VALUE_C');
```

- How is it stored in a memtable?
- How much memory do we need for it?
- What kind of memory is used?

# Memtable - allocation

```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1, VALUE_2)
VALUES ('PART_KEY_1', 'CUST_KEY_1', 'VALUE_A', 'VALUE_B');
```

```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1)
VALUES ('PART_KEY_1', 'CUST_KEY_2', 'VALUE_C');
```

- How is it stored in a memtable?
- How much memory do we need for it?
- What kind of memory is used?

- Less memory -> more data to fit into a memtable -> less flushing -> less compaction -> less write amplification factor

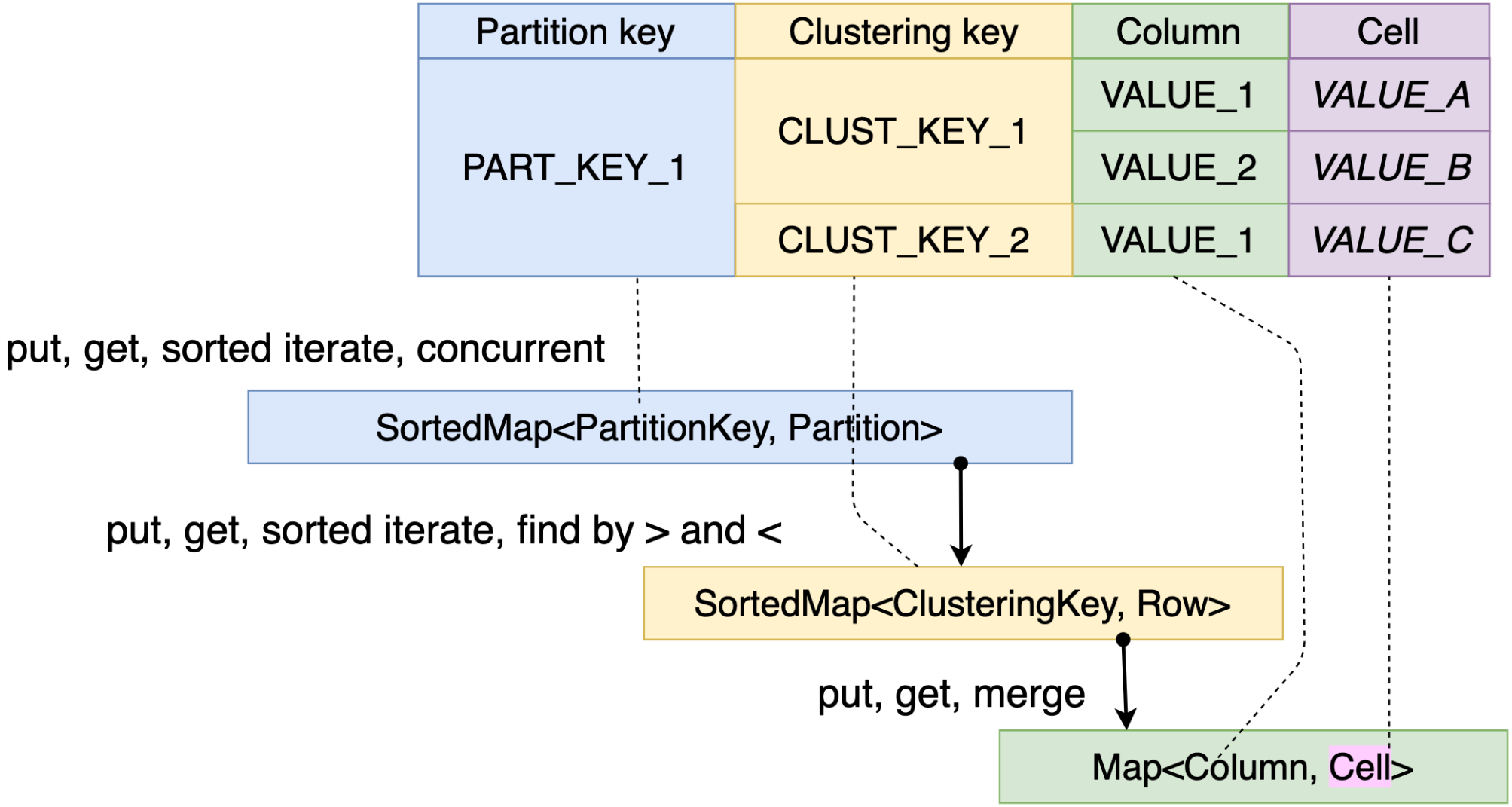
# Memtable - allocation

```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1, VALUE_2)
VALUES ('PART_KEY_1', 'CUST_KEY_1', 'VALUE_A', 'VALUE_B');
```

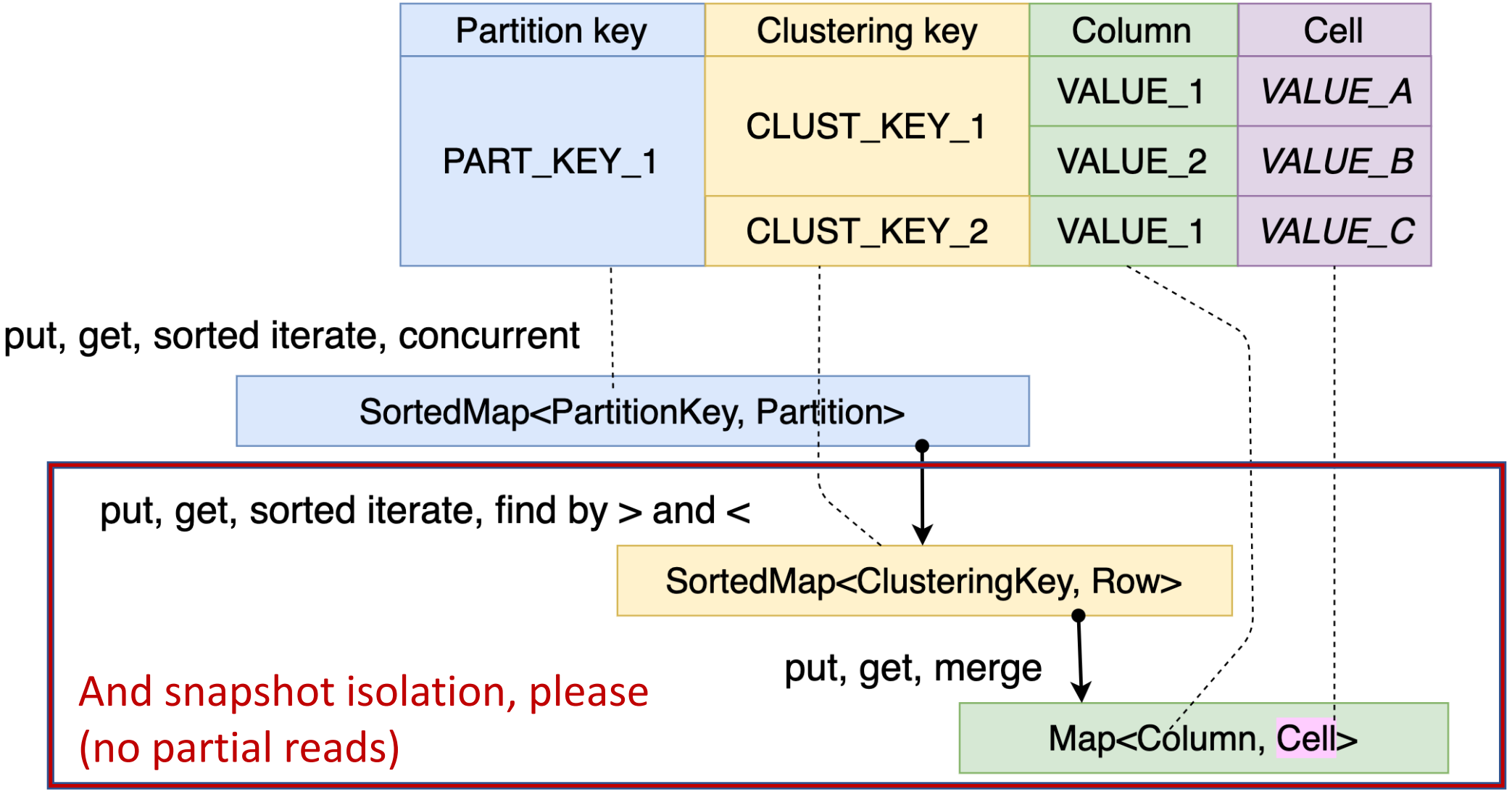
```
INSERT INTO TABLE A
        (PART_KEY, CLUST_KEY, VALUE_1)
VALUES ('PART_KEY_1', 'CUST_KEY_2', 'VALUE_C');
```

| Partition key | Clustering key | Column | Cell |
|---|---|---|---|
| PART_KEY_1 | CLUST_KEY_1 | VALUE_1 | *VALUE_A* |
| | | VALUE_2 | *VALUE_B* |
| | CLUST_KEY_2 | VALUE_1 | *VALUE_C* |

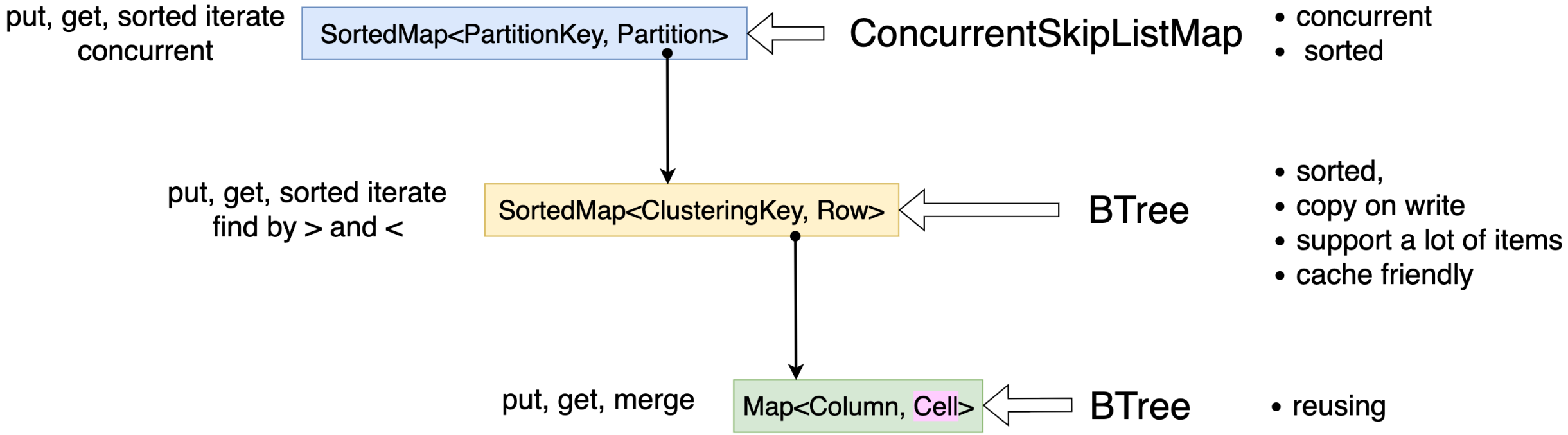
# Memtable - allocation



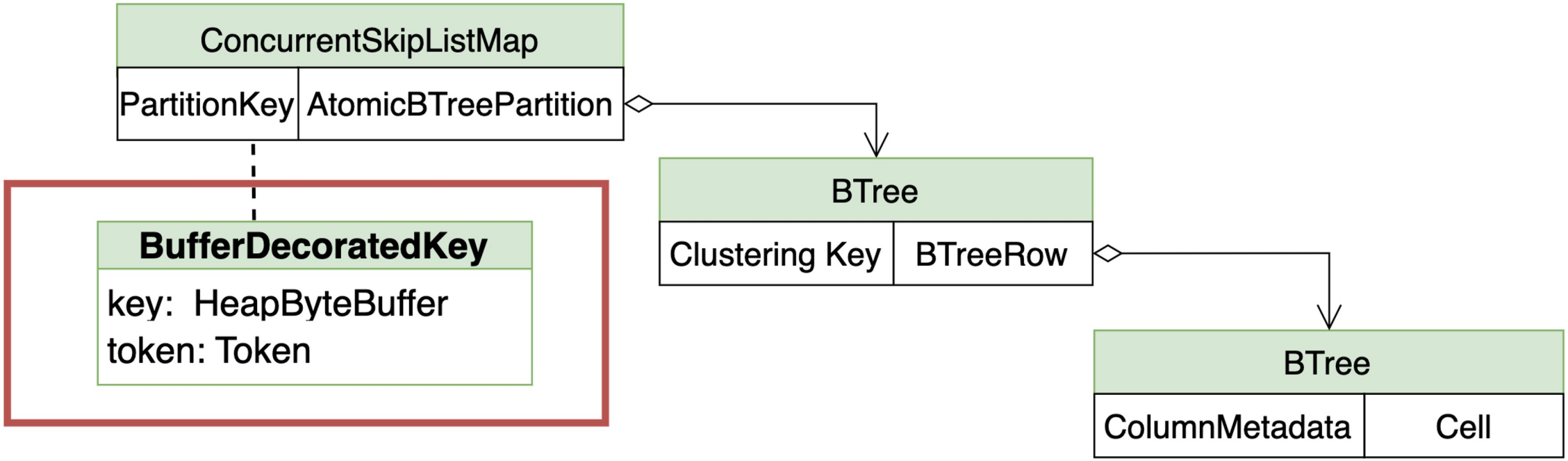
# Memtable - allocation



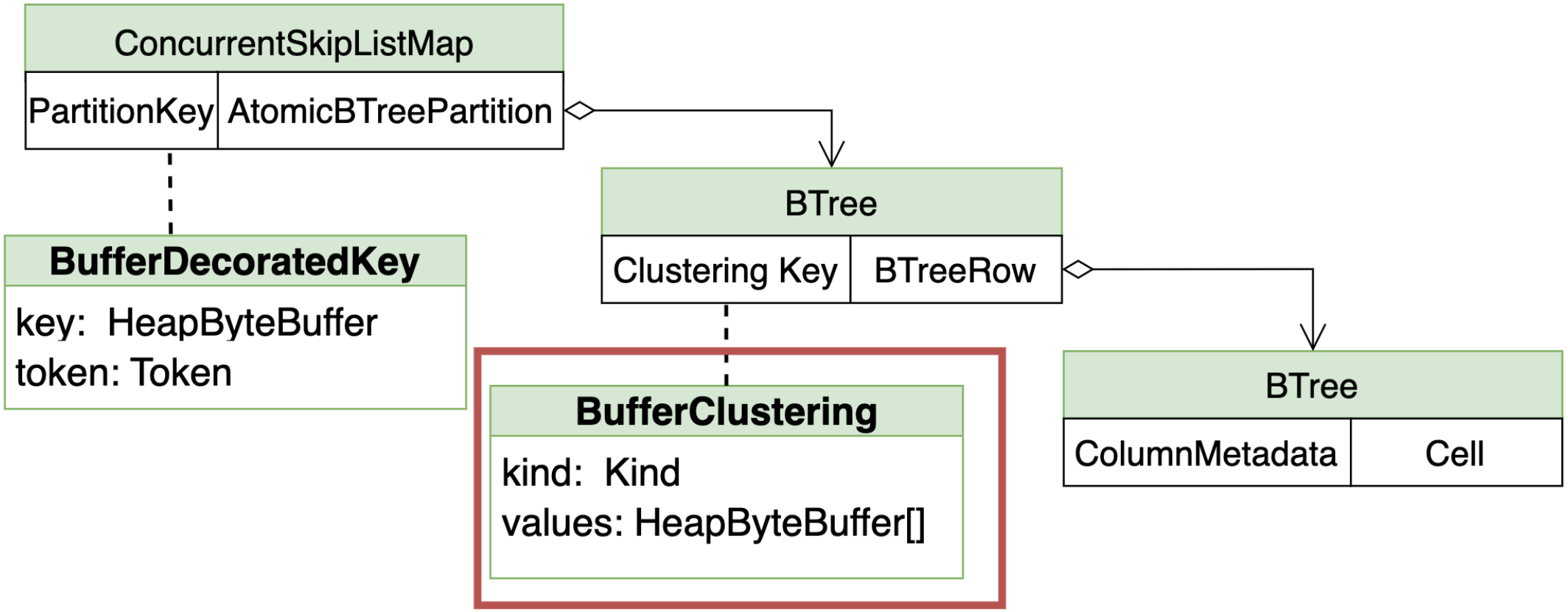
# Memtable - allocation



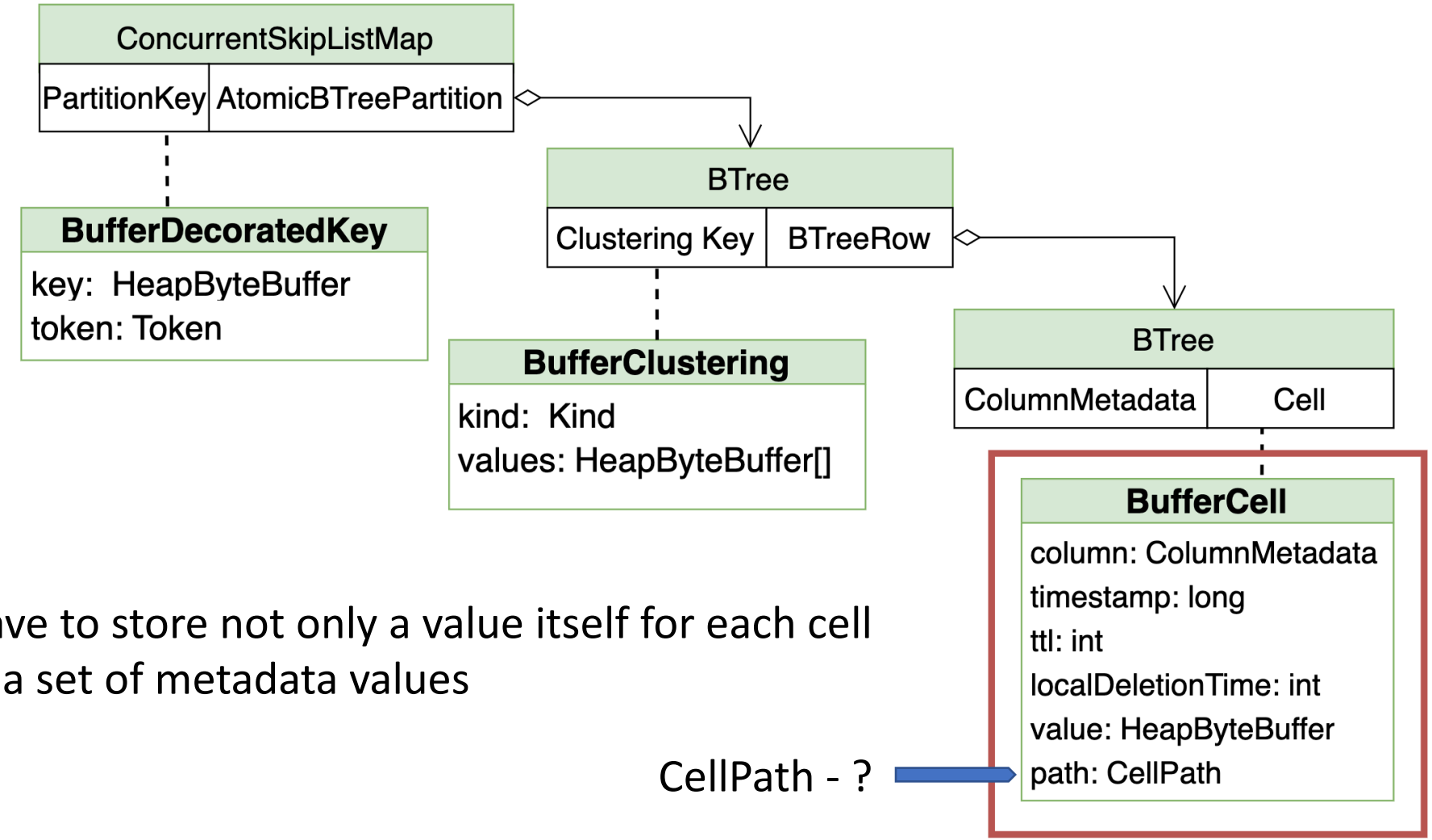
# Memtable - allocation



# Memtable - allocation



# Memtable - allocation




We have to store not only a value itself for each cell but a set of metadata values

CellPath - ?

# Composite types

- Cassandra supports composite types:
  - User Defined Types (UDT) ---
  - Collections (Map, Set, List) ---

```
CREATE TABLE user (
    name text PRIMARY KEY,
    addresses map<text, address>
);
```

```
CREATE TYPE address (
    street text,
    city text,
    zip text,
    phones set<text>
);
```

# Composite types

- Cassandra supports composite types:
  - User Defined Types (UDT)
  - Collections (Map, Set, List)
- These types can be frozen and non-frozen:
  - Frozen – the whole UDT/connection can be re-written only
  - Non-frozen – individual fields/elements can be modified

```
CREATE TABLE user (
    name text PRIMARY KEY,
    addresses map<text, address>
);
```

vs

```
CREATE TABLE user (
    name text PRIMARY KEY,
    addresses map<text, frozen<address>>
);
```

# Composite types

- Cell path – id of a field/element for a non-frozen type
- Each field/element of a non-frozen type is stored as a separate cell

# Composite types

- Cell path – id of a field/element for a non-frozen type
- Each field/element of a non-frozen type is stored as a separate cell

```
CREATE TABLE user (
    name text PRIMARY KEY,
    addresses map<text, address>
);
```

```
CREATE TYPE address (
    street text,
    city text,
    zip text,
    phones set<text>
);
```

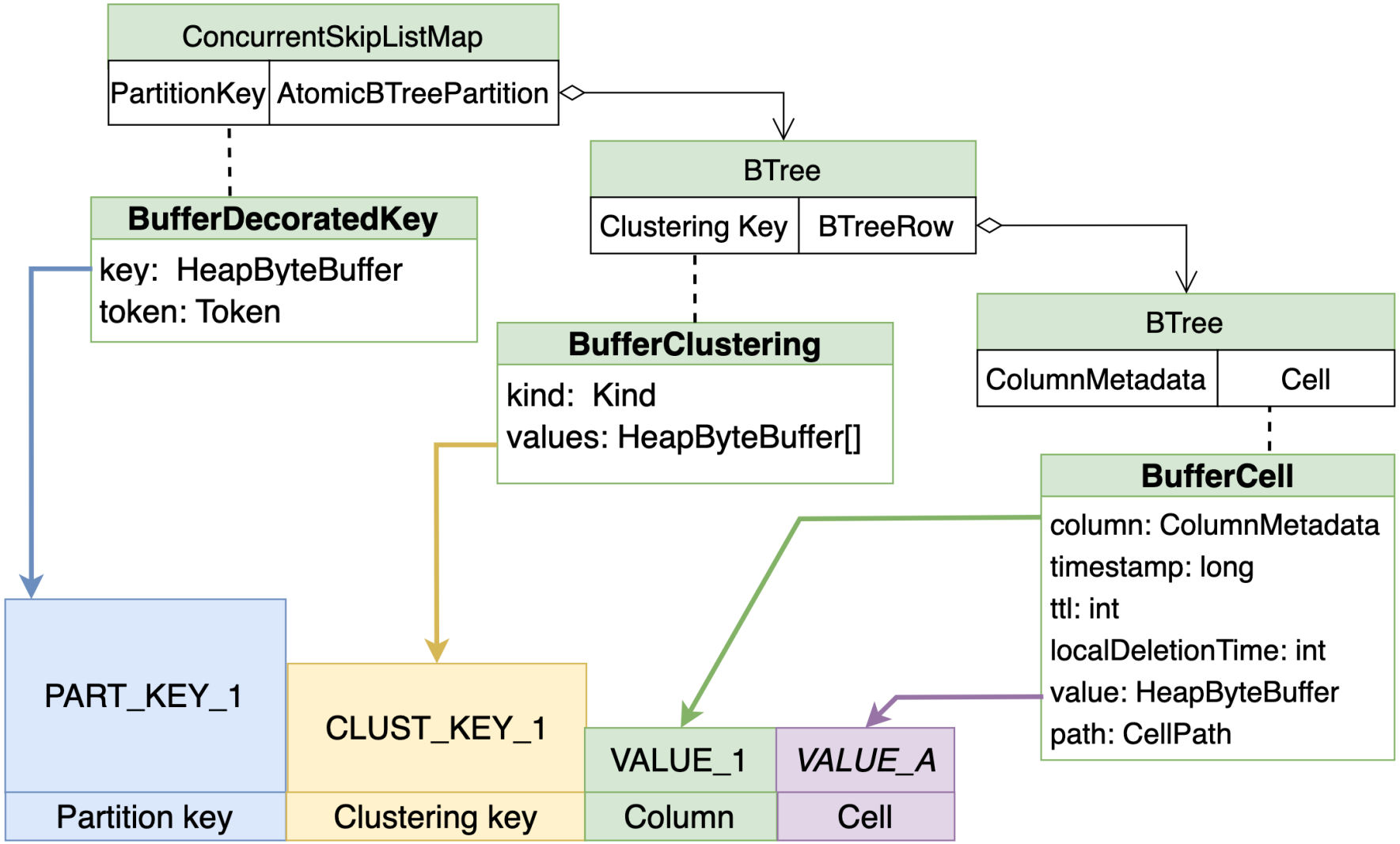
| BufferCell |
| --- |
| **path: "home/street"**<br>value: "Baker Street 221B" |

| BufferCell |
| --- |
| **path: "home/zip"**<br>value: "NW1 6XE" |

| BufferCell |
| --- |
| **path: "work/street"**<br>value: "..." |

...

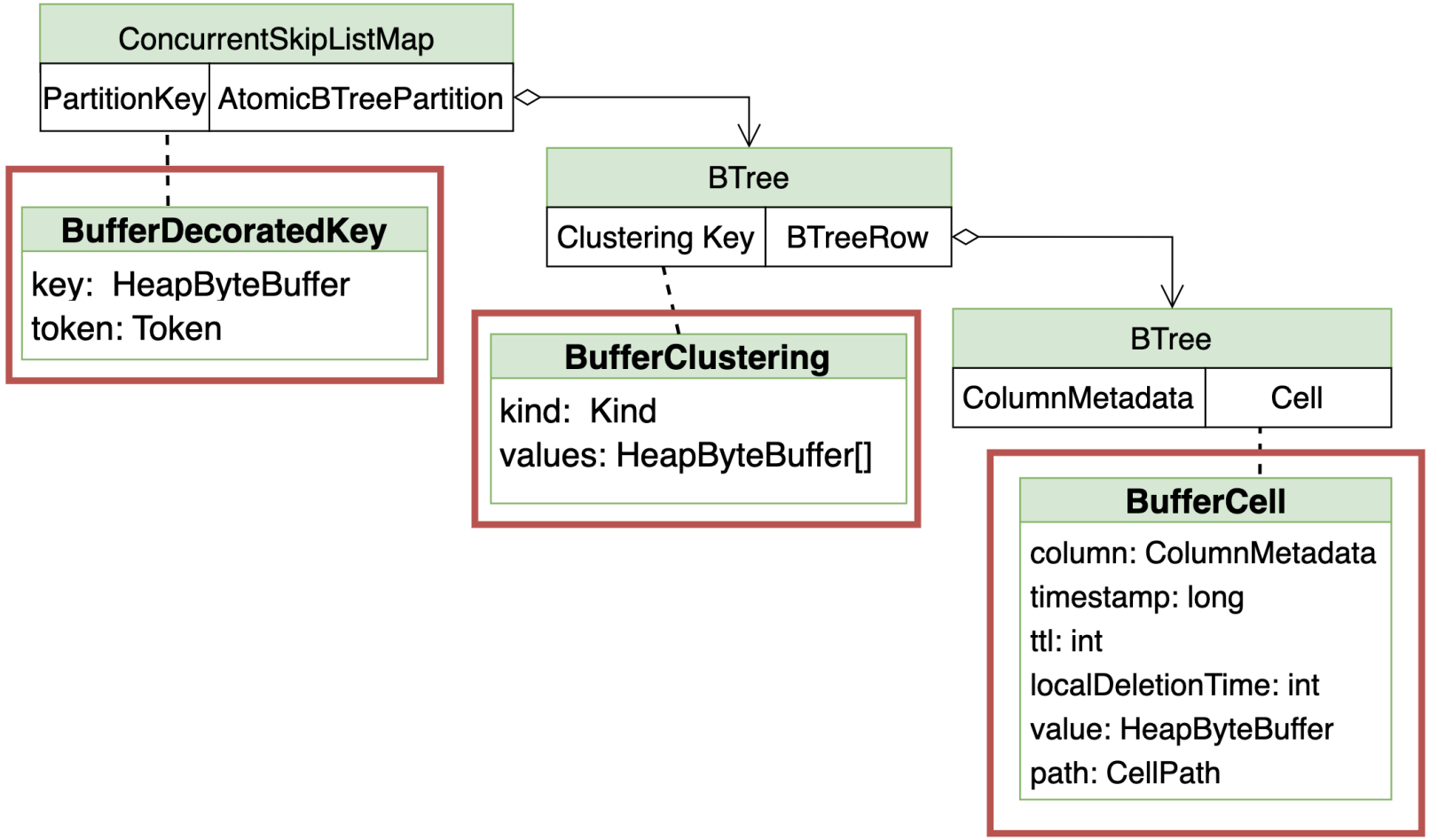
# Memtable - allocation



# Memtable - allocation

# Memtable - allocation

Due to lack of time let's analyze only keys and cell objects



# Memtable - allocation

- JOL (Java Object Layout)
  - https://github.com/openjdk/jol
  - https://shipilev.net/jvm/objects-inside-out/
  - Joker 2021: Java-объекты наизнанку

# Memtable - allocation

Partition key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes to encoded

ClassLayout.*parseClass*(BufferDecoratedKey.class).toPrintable()

# Memtable - allocation

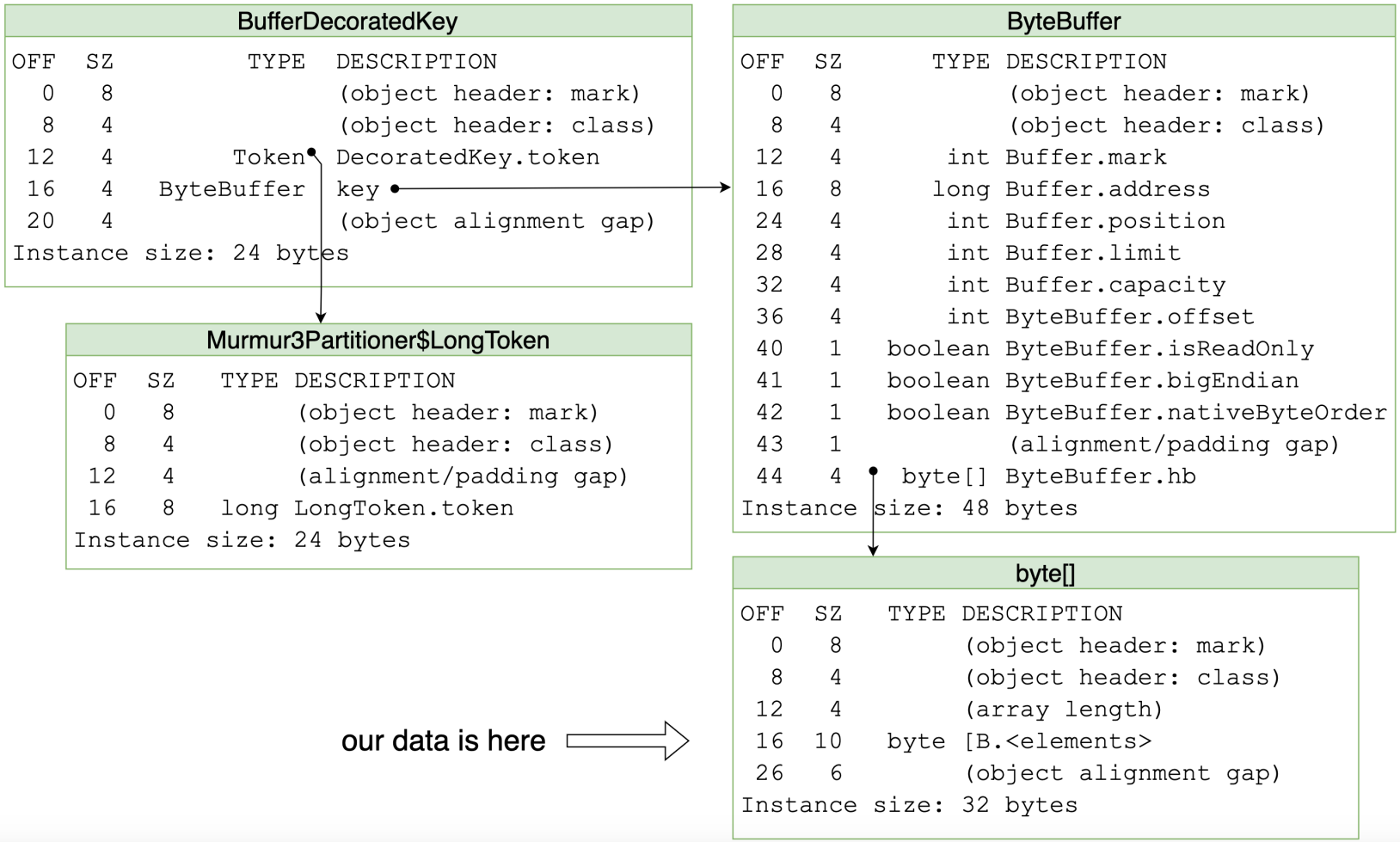
Partition key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

ClassLayout.*parseClass*(BufferDecoratedKey.class).toPrintable()

```
org.apache.cassandra.db.BufferDecoratedKey object internals:
OFF  SZ          TYPE  DESCRIPTION
  0   8                (object header: mark)
  8   4                (object header: class)
 12   4         Token  DecoratedKey.token
 16   4    ByteBuffer  BufferDecoratedKey.key
 20   4                (object alignment gap)
Instance size: 24 bytes
```

# Memtable - allocation

Partition key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded




BufferDecoratedKey – 24 bytes
HeapByteBuffer – 48 bytes
Token – 24 bytes
byte[] – 32 bytes
Total: 128 bytes

# Memtable - allocation

Partition key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

**BufferDecoratedKey**

```
OFF  SZ          TYPE  DESCRIPTION
  0   8                (object header: mark)
  8   4                (object header: class)
 12   4         Token  DecoratedKey.token
 16   4    ByteBuffer  key
 20   4                (object alignment gap)
Instance size: 24 bytes
```

**Murmur3Partitioner$LongToken**

```
OFF  SZ   TYPE DESCRIPTION
  0   8        (object header: mark)
  8   4        (object header: class)
 12   4        (alignment/padding gap)
 16   8   long LongToken.token
Instance size: 24 bytes
```

**ByteBuffer**

```
OFF  SZ      TYPE DESCRIPTION
  0   8           (object header: mark)
  8   4           (object header: class)
 12   4       int Buffer.mark
 16   8      long Buffer.address
 24   4       int Buffer.position
 28   4       int Buffer.limit
 32   4       int Buffer.capacity
 36   4       int ByteBuffer.offset
 40   1   boolean ByteBuffer.isReadOnly
 41   1   boolean ByteBuffer.bigEndian
 42   1   boolean ByteBuffer.nativeByteOrder
 43   1           (alignment/padding gap)
 44   4    byte[] ByteBuffer.hb
Instance size: 48 bytes
```

**byte[]**

```
OFF  SZ   TYPE DESCRIPTION
  0   8        (object header: mark)
  8   4        (object header: class)
 12   4        (array length)
 16  10   byte [B.<elements>
 26   6        (object alignment gap)
Instance size: 32 bytes
```

our data is here ⇨

BufferDecoratedKey – 24 bytes
HeapByteBuffer – 48 bytes
Token – 24 bytes
byte[] – 32 bytes
Total: 128 bytes

# Memtable - allocation

**Clustering** key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

**BufferClustering**

```
OFF  SZ        TYPE    DESCRIPTION
  0   8                (object header: mark)
  8   4                (object header: class)
 12   4          Kind  kind
 16   4 ByteBuffer[]   values
 20   4                (object alignment gap)
Instance size: 24 bytes
```

**ByteBuffer[]**

```
OFF  SZ   TYPE        DESCRIPTION
  0   8               (object header: mark)
  8   4               (object header: class)
 12   4               (array length)
 16   4   ByteBuffer ByteBuffer;.<elements>
 20   4                (object alignment gap)
Instance size: 24 bytes
```

**HeapByteBuffer**

```
OFF  SZ      TYPE DESCRIPTION
  0   8           (object header: mark)
  8   4           (object header: class)
 12   4       int Buffer.mark
 16   8      long Buffer.address
 24   4       int Buffer.position
 28   4       int Buffer.limit
 32   4       int Buffer.capacity
 36   4       int ByteBuffer.offset
 40   1   boolean ByteBuffer.isReadOnly
 41   1   boolean ByteBuffer.bigEndian
 42   1   boolean ByteBuffer.nativeByteOrder
 43   1           (alignment/padding gap)
 44   4    byte[] ByteBuffer.hb
Instance size: 48 bytes
```

**byte[]**

```
OFF  SZ   TYPE DESCRIPTION
  0   8        (object header: mark)
  8   4        (object header: class)
 12   4        (array length)
 16  10   byte [B.<elements>
 26   6        (object alignment gap)
Instance size: 32 bytes
```

our data is here

BufferClustering – 24 bytes
Object[] – 24 bytes
HeapByteBuffer – 48 bytes
byte[] – 32 bytes
Total: 128 bytes

# Memtable - allocation

**Cell** value = ”**0123456789**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

**BufferCell**

```
OFF  SZ              TYPE  DESCRIPTION
  0   8                    (object header: mark)
  8   4                    (object header: class)
 12   4   ColumnMetadata  column
 16   8              long  timestamp
 24   4               int  BufferCell.ttl
 28   4               int  localDeletionTime
 32   4        ByteBuffer  value
 36   4          CellPath  path
Instance size: 40 bytes
```

**HeapByteBuffer**

```
OFF  SZ       TYPE DESCRIPTION
  0   8            (object header: mark)
  8   4            (object header: class)
 12   4        int Buffer.mark
 16   8       long Buffer.address
 24   4        int Buffer.position
 28   4        int Buffer.limit
 32   4        int Buffer.capacity
 36   4        int ByteBuffer.offset
 40   1    boolean ByteBuffer.isReadOnly
 41   1    boolean ByteBuffer.bigEndian
 42   1    boolean ByteBuffer.nativeByteOrder
 43   1            (alignment/padding gap)
 44   4    byte[] ByteBuffer.hb
Instance size: 48 bytes
```

**byte[]**

```
OFF  SZ   TYPE DESCRIPTION
  0   8        (object header: mark)
  8   4        (object header: class)
 12   4        (array length)
 16  10   byte [B.<elements>
 26   6        (object alignment gap)
Instance size: 32 bytes
```

our data is here

BufferCell – 40 bytes
HeapByteBuffer – 48 bytes
byte[] – 32 bytes
Total: 120 bytes

# Memtable - allocation

Partition key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

Clustering key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

Cell value = ”**0123456789**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

| Entity | Objects | Total size, bytes |
|---|---|---|
| Partition key | 4 | 128 |
| Clustering key | 4 | 128 |
| Cell (value column) | 3 | 120 |
| Total | 11 | 376 |

# Memtable - allocation

Partition key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

Clustering key = ”**0000000001**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

Cell value = ”**0123456789**” (TEXT), 10 symbols, UTF8 => 10 bytes encoded

| Entity | Objects | Total size, bytes |
|---|---|---|
| Partition key | 4 | 128 |
| Clustering key | 4 | 128 |
| Cell (value column) | 3 | 120 |
| Total | 11 | 376 |

# Memtable - allocation

How can we reduce the amount data in heap?

- **Idea 1: move ByteBuffer values to off-heap -> DirectByteBuffer**

# Memtable - allocation

## Let’s move data to offheap - DirectByteBuffer

**BufferCell**

```
OFF  SZ              TYPE  DESCRIPTION
  0   8                    (object header: mark)
  8   4                    (object header: class)
 12   4   ColumnMetadata  column
 16   8              long  timestamp
 24   4               int  BufferCell.ttl
 28   4               int  localDeletionTime
 32   4        ByteBuffer  value
 36   4          CellPath  path
Instance size: 40 bytes
```

**HeapByteBuffer**

```
OFF  SZ       TYPE DESCRIPTION
  0   8            (object header: mark)
  8   4            (object header: class)
 12   4        int Buffer.mark
 16   8       long Buffer.address
 24   4        int Buffer.position
 28   4        int Buffer.limit
 32   4        int Buffer.capacity
 36   4        int ByteBuffer.offset
 40   1    boolean ByteBuffer.isReadOnly
 41   1    boolean ByteBuffer.bigEndian
 42   1    boolean ByteBuffer.nativeByteOrder
 43   1            (alignment/padding gap)
 44   4    byte[] ByteBuffer.hb
Instance size: 48 bytes
```

**byte[]**

```
OFF  SZ   TYPE DESCRIPTION
  0   8        (object header: mark)
  8   4        (object header: class)
 12   4        (array length)
 16  10   byte [B.<elements>
 26   6        (object alignment gap)
Instance size: 32 bytes
```

our data is here

# Memtable - allocation

## Let’s move data to offheap - DirectByteBuffer

**BufferCell**

```
OFF  SZ              TYPE  DESCRIPTION
  0   8                    (object header: mark)
  8   4                    (object header: class)
 12   4   ColumnMetadata  column
 16   8              long  timestamp
 24   4               int  BufferCell.ttl
 28   4               int  localDeletionTime
 32   4        ByteBuffer  value
 36   4          CellPath  path
Instance size: 40 bytes
```

**DirectByteBuffer**

```
OFF  SZ              TYPE DESCRIPTION
  0   8                   (object header: mark)
  8   4                   (object header: class)
 12   4               int Buffer.mark
 16   8              long Buffer.address
 24   4               int Buffer.position
 28   4               int Buffer.limit
 32   4               int Buffer.capacity
 36   4               int ByteBuffer.offset
 40   1           boolean ByteBuffer.isReadOnly
 41   1           boolean ByteBuffer.bigEndian
 42   1           boolean ByteBuffer.nativeByteOrder
 43   1                   (alignment/padding gap)
 44   4            byte[] ByteBuffer.hb
 48   4   FileDescriptor MappedByteBuffer.fd
 52   4            Object DirectByteBuffer.att
 56   4           Cleaner DirectByteBuffer.cleaner
 60   4                   (object alignment gap)
Instance size: 64 bytes
```

HeapByteBuffer was 48 bytes ☹

# Memtable - allocation

Let’s move data to offheap - DirectByteBuffer

| Entity | Objects | Heap size, bytes | Off-heap, bytes | Total, bytes |
|---|---|---|---|---|
| Partition key | 4 -> 3 | 128 -> 112 | 10 | 128 -> 122 |
| Clustering key | 4 -> 3 | 128 -> 112 | 10 | 128 -> 122 |
| Cell (value column) | 3 -> 2 | 120 -> 104 | 10 | 120 -> 114 |
| Total | 11 -> 8 | 376 -> 328 | 30 | 376 -> 358 |
| | | **13% better** | | **5% better** |

# Memtable - allocation

Let’s move data to offheap - DirectByteBuffer

| Entity | Objects | Heap size, bytes | Off-heap, bytes | Total, bytes |
|---|---|---|---|---|
| Partition key | 4 -> 3 | 128 -> 112 | 10 | 128 -> 122 |
| Clustering key | 4 -> 3 | 128 -> 112 | 10 | 128 -> 122 |
| Cell (value column) | 3 -> 2 | 120 -> 104 | 10 | 120 -> 114 |
| Total | 11 -> 8 | 376 -> 328 | 30 | 376 -> 358 |

memtable_allocation_type:

- heap_buffers
- **offheap_buffers <==**

# Memtable - allocation

How can we reduce the amount data in heap?

- Idea 1: move ByteBuffer values to off-heap -> DirectByteBuffer
- **Idea 2: get rid of ByteBuffer objects at all**

# Memtable - allocation

Git rid of DirectByteBuffer objects:

1. **Allocate memory using a native allocator**
2. **Store a pointer to it as long**

# Memtable - allocation

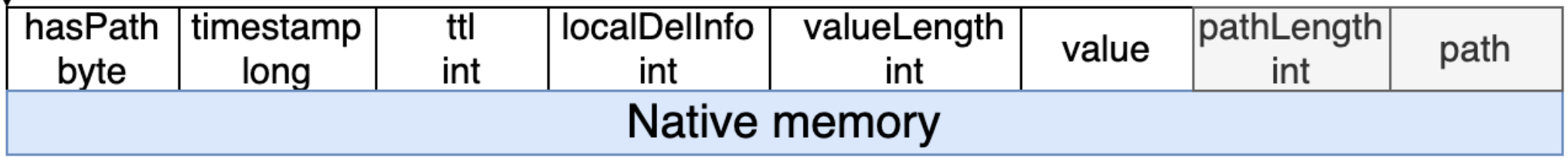
Git rid of DirectByteBuffer objects:

1. Allocate memory using a native allocator
2. Store a pointer to it as long
3. **Read/write the memory using sun.misc.Unsafe**
4. **Encode data into the memory**
5. ?!
6. Profit

# Memtable - allocation

**NativeCell**

```
OFF  SZ                TYPE  DESCRIPTION
  0   8                      (object header: mark)
  8   4                      (object header: class)
 12   4   ColumnMetadata  ColumnData.column
 16   8              long  NativeCell.peer
Instance size: 24 bytes
```




1 + 8 + 4 + 4 + 4 + 10 = 31 bytes

# Memtable - allocation

Git rid of DirectByteBuffer objects

| Entity | Objects | Heap size, bytes | Off-heap, bytes | Total, bytes |
|---|---|---|---|---|
| Partition key | 4 -> 3 -> 2 | 128 -> 112 -> 48 | 10 -> 14 | 128 -> 122 -> 62 |
| Clustering key | 4 -> 3 -> 1 | 128 -> 112 -> 24 | 10 -> 17 | 128 -> 122 -> 41 |
| Cell | 3 -> 2 -> 1 | 120 -> 104 -> 24 | 10 -> 31 | 120 -> 114 -> 55 |
| Total | 11 -> 8 -> 4 | 376 -> 328 -> 96 | 30 -> 62 | 376 -> 358 -> 158 |

**74% better!**

**58% better!**

# Memtable - allocation

Git rid of DirectByteBuffer objects

| Entity | Objects | Heap size, bytes | Off-heap, bytes | Total, bytes |
|---|---|---|---|---|
| Partition key | 4 -> 3 -> 2 | 128 -> 112 -> 48 | 10 -> 14 | 128 -> 122 -> 62 |
| Clustering key | 4 -> 3 -> 1 | 128 -> 112 -> 24 | 10 -> 17 | 128 -> 122 -> 41 |
| Cell | 3 -> 2 -> 1 | 120 -> 104 -> 24 | 10 -> 31 | 120 -> 114 -> 55 |
| Total | 11 -> 8 -> 4 | 376 -> 328 -> 96 | 30 -> 62 | 376 -> 358 -> 158 |

memtable_allocation_type:

- heap_buffers
- offheap_buffers
- **offheap_objects <==**

# Memtable

- memtable_allocation_type:
  - heap_buffers
  - offheap_buffers
  - offheap_objects
- memtable_heap_space (default: 1/4 of heap size)
- memtable_offheap_space (default: 1/4 of heap size)

# Memtable - allocation

How can we reduce the amount data in heap?

- Idea 1: move ByteBuffer values to off-heap -> DirectByteBuffer
- Idea 2: get rid of ByteBuffer objects at all
- **Idea 3: reduce objects overhead**

# Objects overhead

- Currently: each Java object has a header = 12 bytes

**NativeCell**

```
OFF  SZ               TYPE  DESCRIPTION
  0   8                     (object header: mark)
  8   4                     (object header: class)
 12   4   ColumnMetadata  ColumnData.column
 16   8               long  NativeCell.peer
Instance size: 24 bytes
```

# Objects overhead

- Currently: each Java object has a header = 12 bytes
- Project Lilliput: 12 bytes -> 8 bytes (or, potentially, even 4 bytes)

**NativeCell**

```
OFF   SZ                TYPE   DESCRIPTION
  0    8                       (object header: mark)
  8    4                       (object header: class)
 12    4   ColumnMetadata   ColumnData.column
 16    8              long   NativeCell.peer
Instance size: 24 bytes
```

# Lilliput, additional info

- https://wiki.openjdk.org/display/lilliput
- Roman Kennke, Project Lilliput - Compressed Object Headers
- Joker 2023: Максим Дегтярёв — Лилипут на горизонте
- https://builds.shipilev.net/openjdk-jdk-lilliput/
- We can estimate results with JOL:
  - https://github.com/openjdk/jol?tab=readme-ov-file#heapdump-estimates

# objects overhead - Lilliput

- Heap: 16 Gb
- offheap_objects
- 100’000 partitions x 10 clustering keys = 1'000’000 rows
- 10 TEXT columns, 10 symbols – 10’000’000 cells

```
java –jar jol–cli–latest.jar heapdump–stats dump.hprof
```

# objects overhead - Lilliput

Hotspot Layout Simulation (JDK 11, Current VM: 12-byte object headers, 4-byte references, 8-byte aligned objects, 8-byte aligned array bases)

| INSTANCES | SIZE | SUM SIZE | CLASS |
|---|---|---|---|
| 10,000,125 | 24 | 240,003,000 | NativeCell |
| 1,200,189 | 64 | 76,812,096 | Object[11] |
| 1,000,069 | 32 | 32,002,208 | BTreeRow |
| 1,000,057 | 24 | 24,001,368 | LivenessInfo |
| 1,000,046 | 24 | 24,001,104 | NativeClustering |
| 100,868 | 24 | 2,420,832 | Murmur3Partitioner$LongToken |
| 100,404 | 24 | 2,409,696 | ConcurrentSkipListMap$Node |
| 100,244 | 24 | 2,405,856 | RegularAndStaticColumns |
| 100,158 | 24 | 2,403,792 | Columns |
| 100,150 | 32 | 3,204,800 | EncodingStats |
| 100,051 | 32 | 3,201,632 | AbstractBTreePartition$Holder |
| 100,050 | 24 | 2,401,200 | NativeDecoratedKey |
| 100,050 | 32 | 3,201,600 | AtomicBTreePartition |

# objects overhead - Lilliput

- Heap: 16 Gb
- offheap_objects
- 100’000 partitions x 10 clustering keys = 1'000’000 rows
- 10 TEXT columns, 10 symbols – 10’000’000 cells

```
java –jar jol–cli–latest.jar heapdump–estimates dump.hprof
```

# objects overhead - Lilliput

```
Heap Dump: 16gb_offheap_objects_100kPart.hprof

'Overhead' comes from additional metadata, representation and alignment losses.
'JVM mode' is the relative footprint change compared to the best JVM mode in this JDK.
'Upgrade From' is the relative footprint change against the same mode in other JDKs.

Read progress: 269M... 538M... DONE
=== Overall Statistics


    15715K,     Total objects
      265M,     Total data size
     16.88,     Average data per object

=== Stock 32-bit OpenJDK


 Footprint,   Overhead,     Description
      444M,     +67.6%,     32-bit (<4 GB heap)

=== Stock 64-bit OpenJDK (JDK < 15)

 Footprint,   Overhead,   JVM Mode,     Description
      667M,    +151.6%,     +40.3%,     64-bit, no comp refs (>32 GB heap, default align)
      475M,     +79.3%,         0%,     64-bit, comp refs (<32 GB heap, default align)
      578M,    +118.1%,     +21.7%,     64-bit, comp refs with large align (   32..64GB heap,  16-byte align)
      581M,    +119.1%,     +22.2%,     64-bit, comp refs with large align (  64..128GB heap,  32-byte align)
     1042M,    +292.9%,    +119.1%,     64-bit, comp refs with large align ( 128..256GB heap,  64-byte align)
     2044M,    +670.7%,    +329.8%,     64-bit, comp refs with large align ( 256..512GB heap, 128-byte align)
     4053M,   +1428.3%,    +752.4%,     64-bit, comp refs with large align (512..1024GB heap, 256-byte align)
…
```

# objects overhead - Lilliput

| Mode | Used heap, Mb | Change, % |
|---|---|---|
| Stock 64-bit OpenJDK (JDK < 15), 12-byte headers | 475 | 0 |
| Experimental 64-bit OpenJDK: Lilliput, 8-byte headers, array base improvements | 444 | -6.5 |
| Experimental 64-bit OpenJDK: Lilliput, 4-byte headers | 349 | -26.4 |

# objects overhead - Lilliput

| Mode | Used heap, Mb | Change, % |
|---|---|---|
| Stock 64-bit OpenJDK (JDK < 15), 12-byte headers | 475 | 0 |
| Experimental 64-bit OpenJDK: Lilliput, 8-byte headers, array base improvements | 444 | -6.5 |
| Experimental 64-bit OpenJDK: Lilliput, 4-byte headers | 349 | -26.4 |

small

**NativeCell** (the most frequent object)

```
OFF  SZ               TYPE  DESCRIPTION
  0   8                     (object header: mark)
  8   4                     (object header: class)
 12   4   ColumnMetadata  ColumnData.column
 16   8               long  NativeCell.peer
Instance size: 24 bytes
```

# objects overhead - Lilliput

| Mode | Used heap, Mb | Change, % |
|---|---|---|
| Stock 64-bit OpenJDK (JDK < 15), 12-byte headers | 475 | 0 |
| Experimental 64-bit OpenJDK: Lilliput, 8-byte headers, array base improvements | 444 | -6.5 |
| Experimental 64-bit OpenJDK: Lilliput, 4-byte headers | 349 | -26.4 |

small

**NativeCell**

```
OFF  SZ               TYPE  DESCRIPTION
  0   8                     (object header: mark)
  8   4                     (object header: class)
 12   4   ColumnMetadata  ColumnData.column
 16   8               long  NativeCell.peer
Instance size: 24 bytes
```

Objects are 8-byte aligned

24 – 4 == align ==> 24

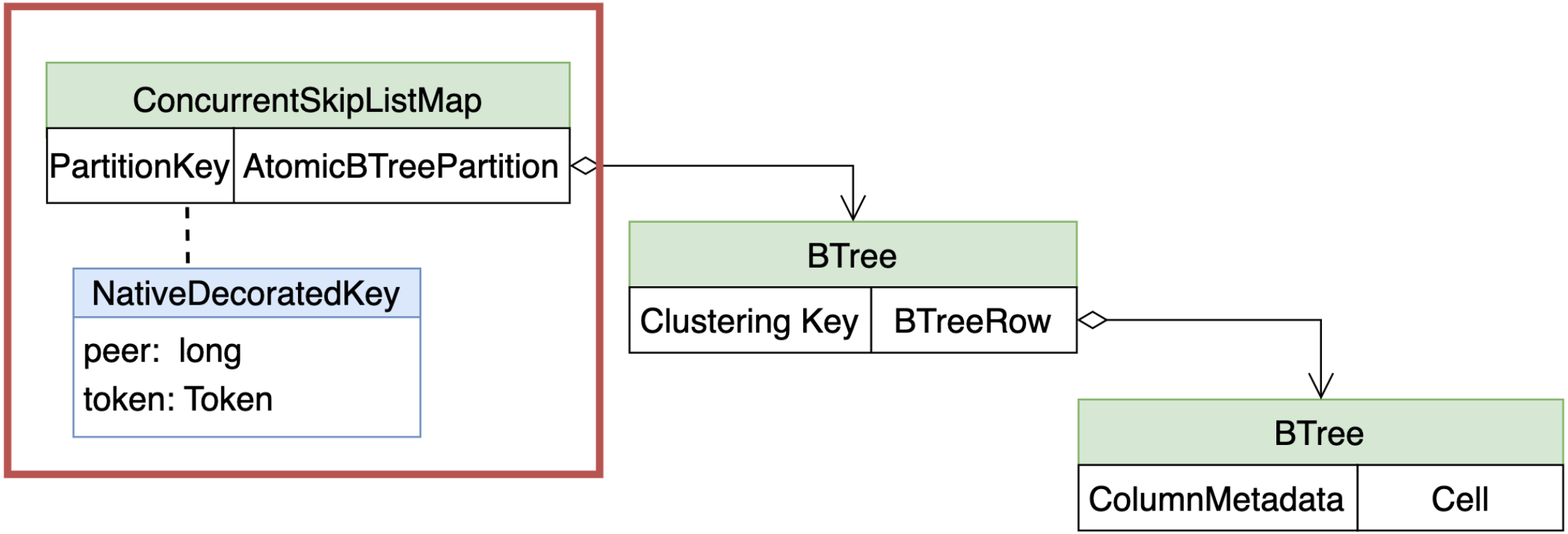
24 – 8 == align ==> 16

# Memtable - allocation

How can we reduce the amount data in heap?

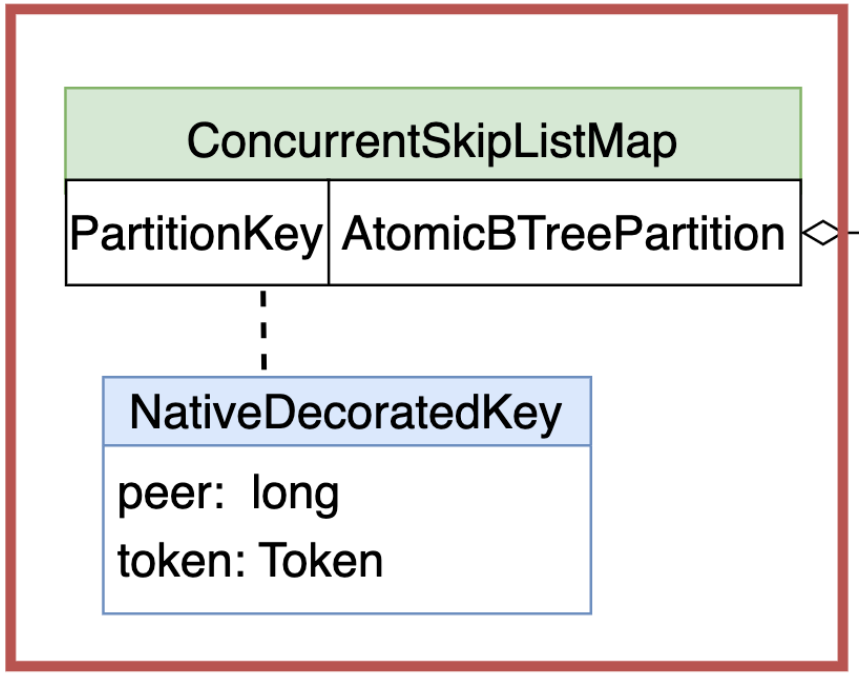
- Idea 1: move ByteBuffer values to off-heap -> DirectByteBuffer
- Idea 2: get rid of ByteBuffer objects at all
- Idea 3: reduce objects overhead
- **Idea 4: get rid of other objects**

# Partition keys




We still have a lot of objects here

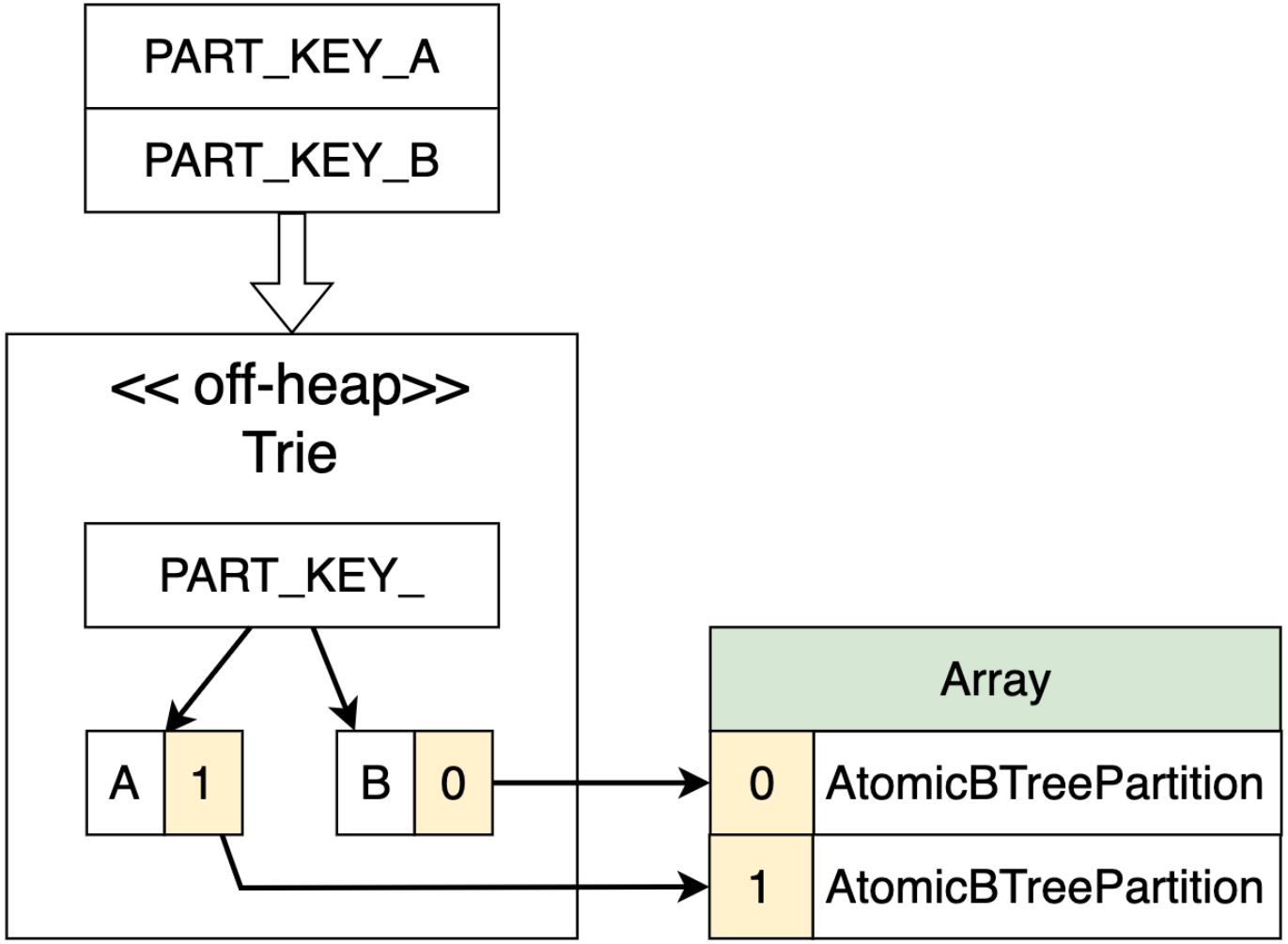
# Partition keys



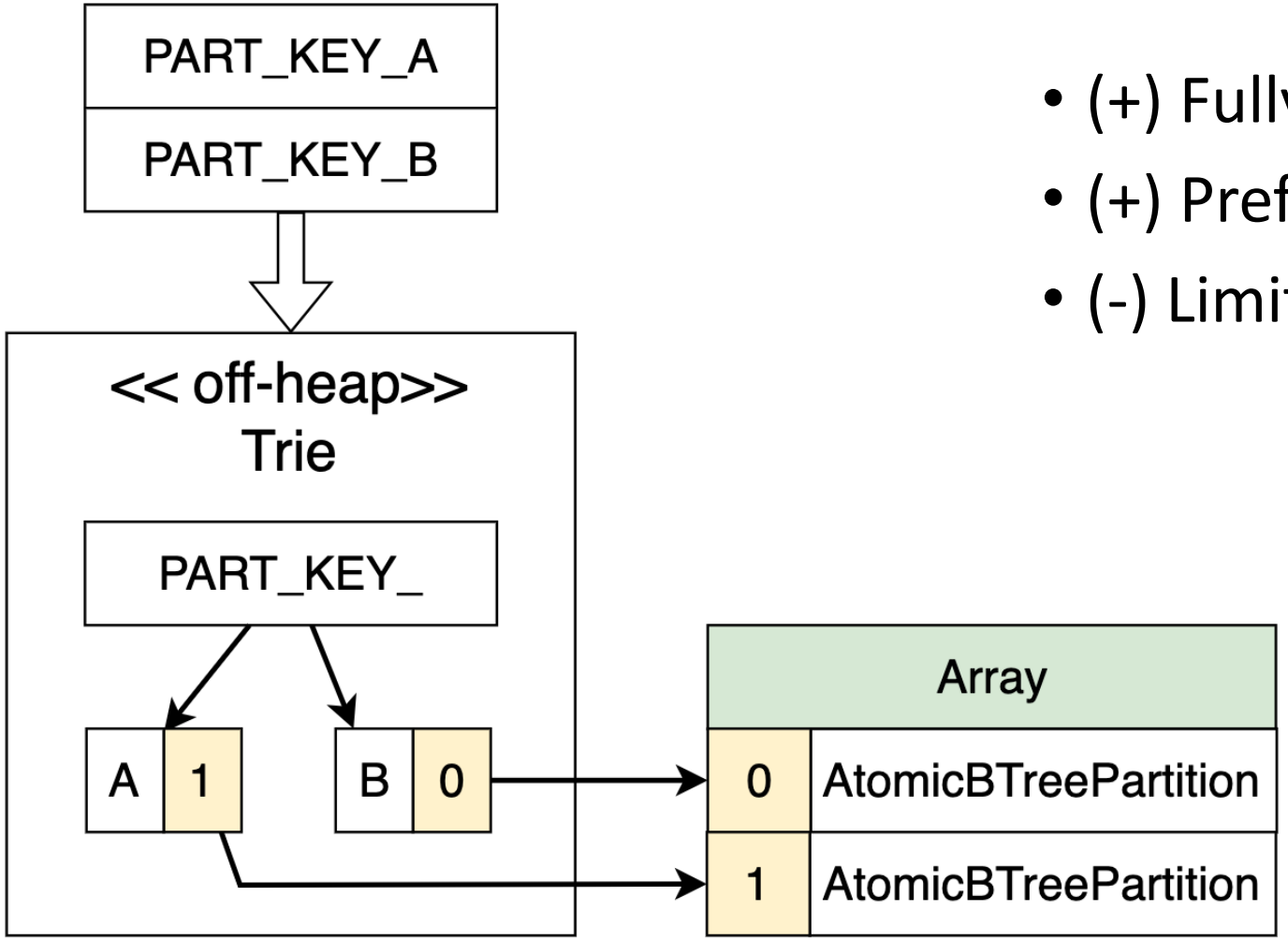


Let’s move this whole structure to off-heap!

# Let’s try a trie for partition keys

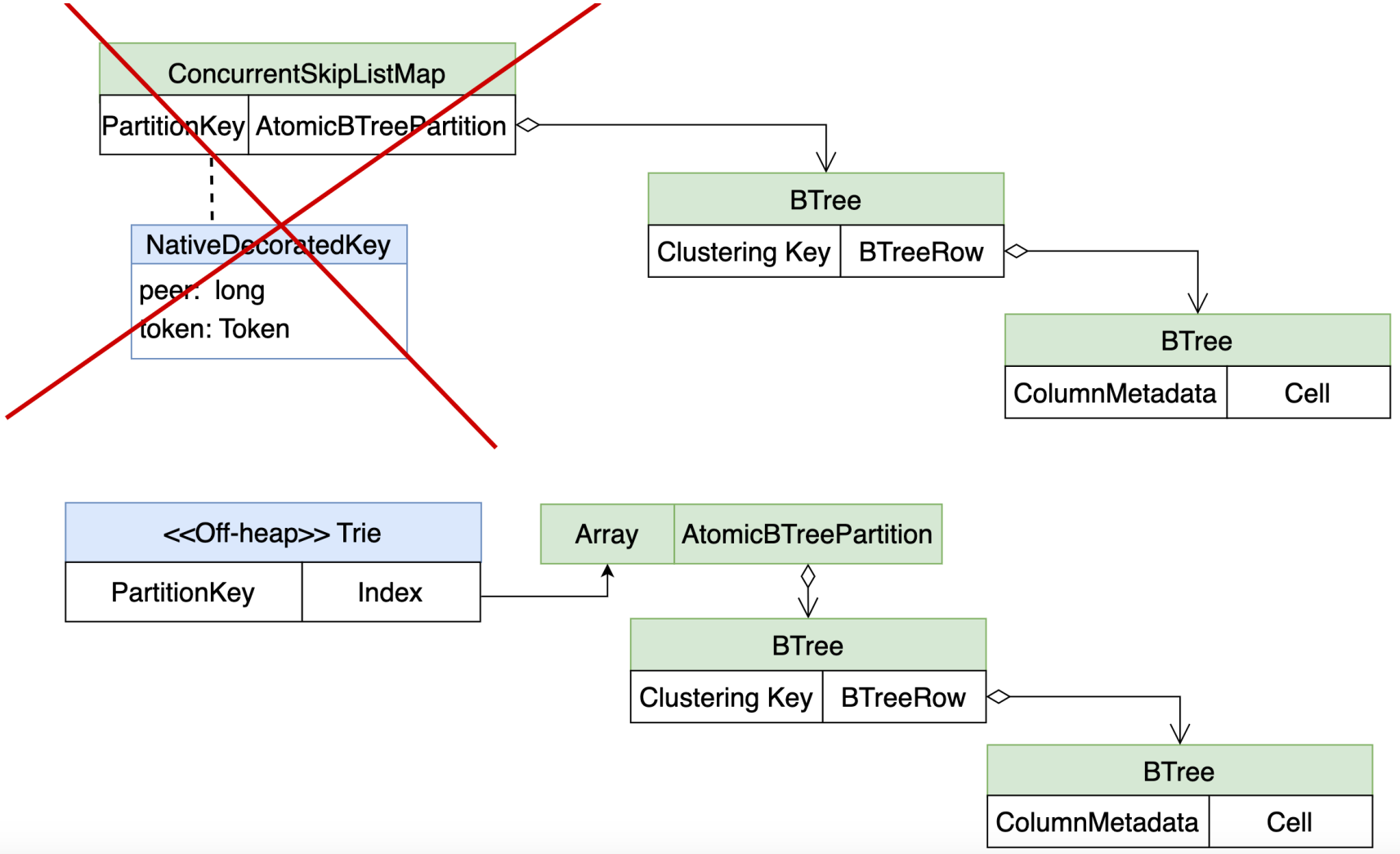


# Let’s try a trie for partition keys




- (+) Fully off-heap
- (+) Prefix deduplication
- (-) Limited concurrency

# Let’s try a trie for partition keys



# Tries for partition keys

- Available since 5.0
- https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-19%3A+Trie+memtable+implementation
- https://cassandra.apache.org/_/blog/Apache-Cassandra-5.0-Features-Trie-Memtables-and-Trie-Indexed-SSTables.html
- https://www.vldb.org/pvldb/vol15/p3359-lambov.pdf
  - You can find benchmark results here

# Memtable

Partition key, clustering keys, cells, objects, tries..

Ok, but what is the total amount of memory required to store data in memtable?

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes |
|---|---|
| 1 TEXT column, 100 symbols | |
| 10 TEXT columns, 10 symbols | |
| | |
| | |
| | |
| | |

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes |
|---|---|
| 1 TEXT column, 100 symbols | |
| 10 TEXT columns, 10 symbols | |
| 10 TIMESTAMP columns | |
| 10 INT columns | |
| | |
| | |

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes |
|---|---|
| 1 TEXT column, 100 symbols | |
| 10 TEXT columns, 10 symbols | |
| 10 TIMESTAMP columns | |
| 10 INT columns | |
| UDT, 10 TEXT fields, 10 symbols, non-frozen | |
| UDT, 10 TEXT fields, 10 symbols, frozen | |

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes |
|---|---|
| 1 TEXT column, 100 symbols | 296 |
| 10 TEXT columns, 10 symbols | 741 |
| 10 TIMESTAMP columns | |
| 10 INT columns | |
| UDT, 10 TEXT fields, 10 symbols, non-frozen | |
| UDT, 10 TEXT fields, 10 symbols, frozen | |

1 cell vs 1 cells

2.5x diff

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes |
|---|---|
| 1 TEXT column, 100 symbols | 296 |
| 10 TEXT columns, 10 symbols | 741 |
| 10 TIMESTAMP columns | 721 |
| 10 INT columns | 681 |
| UDT, 10 TEXT fields, 10 symbols, non-frozen | |
| UDT, 10 TEXT fields, 10 symbols, frozen | |

8-byte vs 4-byte data
the size is almost the same
overheads dominate

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes |
|---|---|
| 1 TEXT column, 100 symbols | 296 |
| 10 TEXT columns, 10 symbols | 741 |
| 10 TIMESTAMP columns | 721 |
| 10 INT columns | 681 |
| UDT, 10 TEXT fields, 10 symbols, non-frozen | 849 |
| UDT, 10 TEXT fields, 10 symbols, frozen | 336 |

Frozen UDT = 1 cell

55% better

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes |
|---|---|
| 1 TEXT column, 100 symbols | 296 |
| 10 TEXT columns, 10 symbols | 741 |
| 10 TIMESTAMP columns | 721 |
| 10 INT columns | 681 |
| UDT, 10 TEXT fields, 10 symbols, non-frozen | 849 |
| UDT, 10 TEXT fields, 10 symbols, frozen | 336 |

If you have several values
and all of them are changed together

group them into a frozen UDT

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- offheap_objects

| Case | Size, bytes | On disk (LZ4) |
|---|---|---|
| 1 TEXT column, 100 symbols | 296 | 112 |
| 10 TEXT columns, 10 symbols | 741 | 130 |
| 10 TIMESTAMP columns | 721 | 101 |
| 10 INT columns | 681 | 61 |
| UDT, 10 TEXT fields, 10 symbols, non-frozen | 849 | 146 |
| UDT, 10 TEXT fields, 10 symbols, frozen | 336 | 132 |

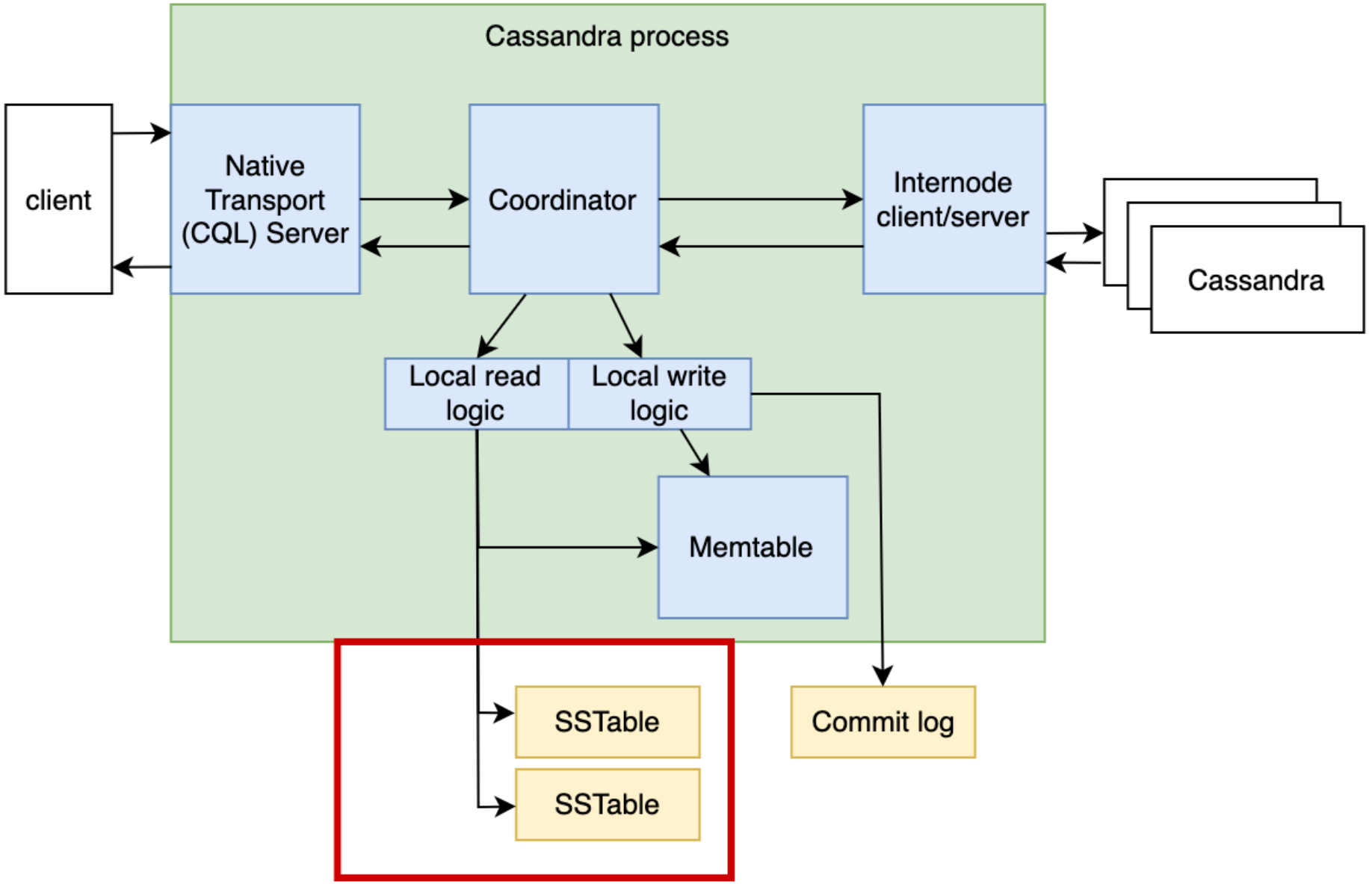
Disk representation is more compact, reasons:

1) no objects
2) immutable
3) compressed

# Interim summary

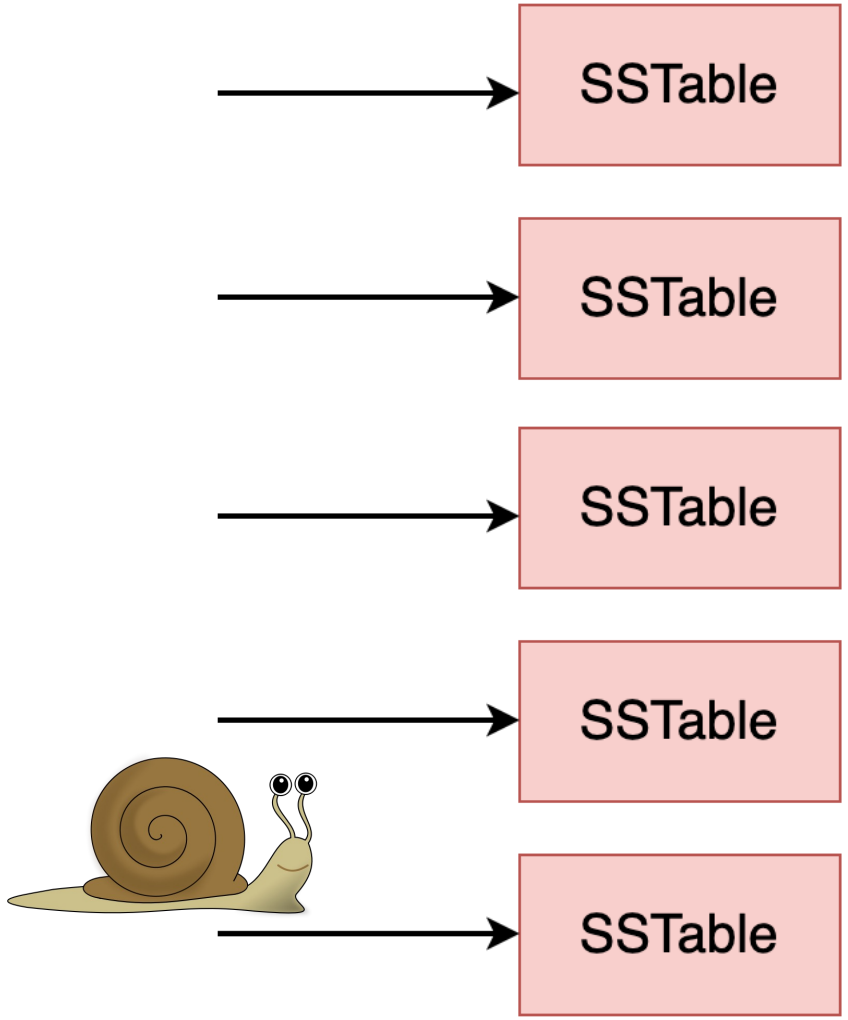
| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1% | cassandra.yaml | JMX, nodetool |
| Memtable | Heap | 2048 | 25 | memtable_heap_space | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| Memtable | Offheap | 2048 | 25 | memtable_offheap_space | JMX, nodetool |


- JVM
- Network
- Coordinator
- Memtables
- **SSTables**
  - **Bloom filter**

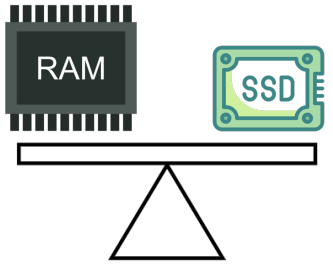


# SSTables read

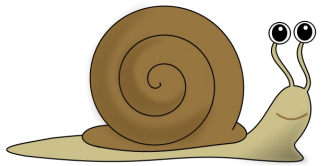
- Do we need to read all of them ?!



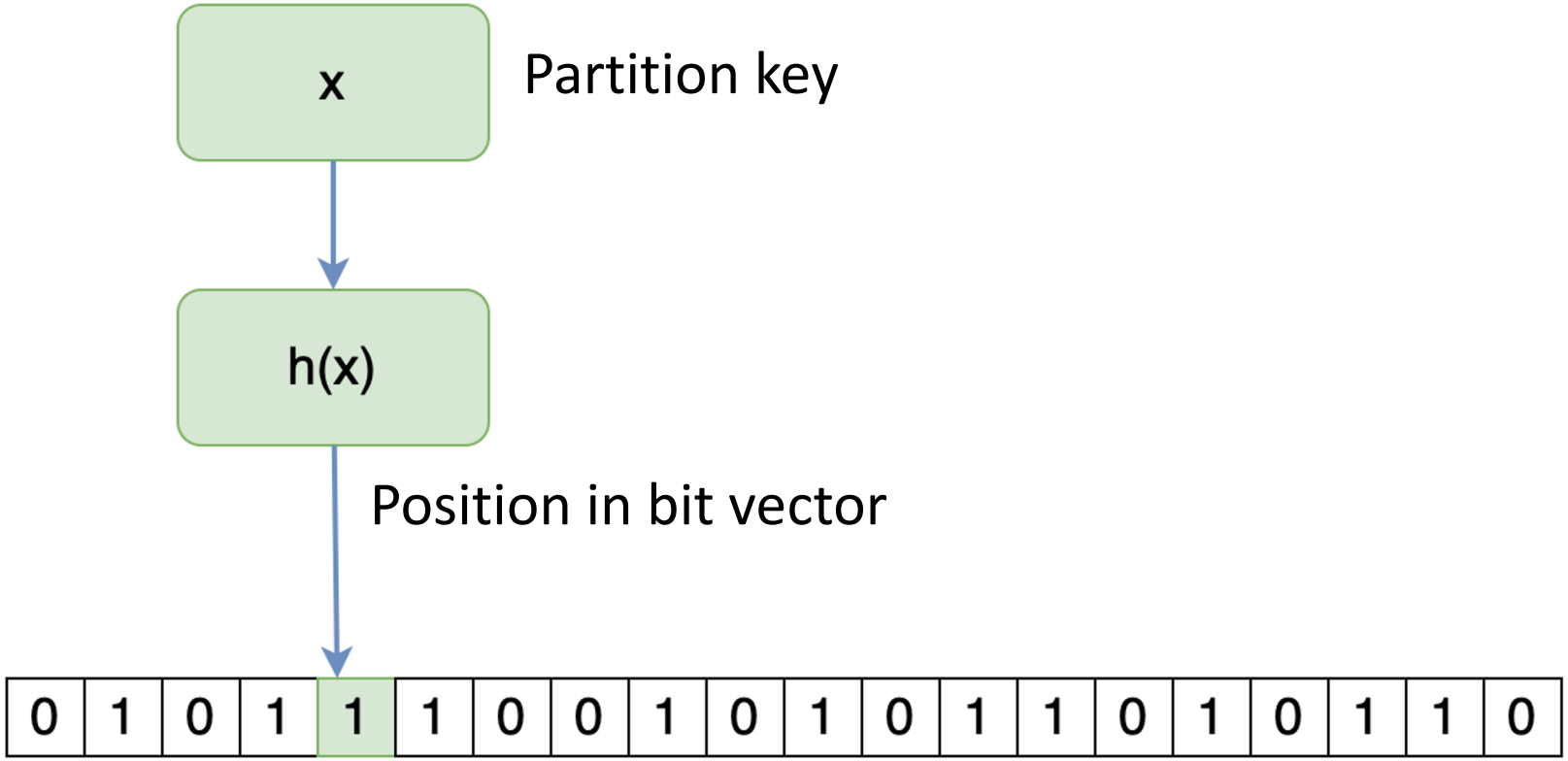
# SSTables read




- Do we need to read all of them ?! – no
- Bloom filter – a probabilistic data structure
  - Maybe yes..
  - Absolutely not!
- Decides based on a partition key

# SSTable – Bloom filter

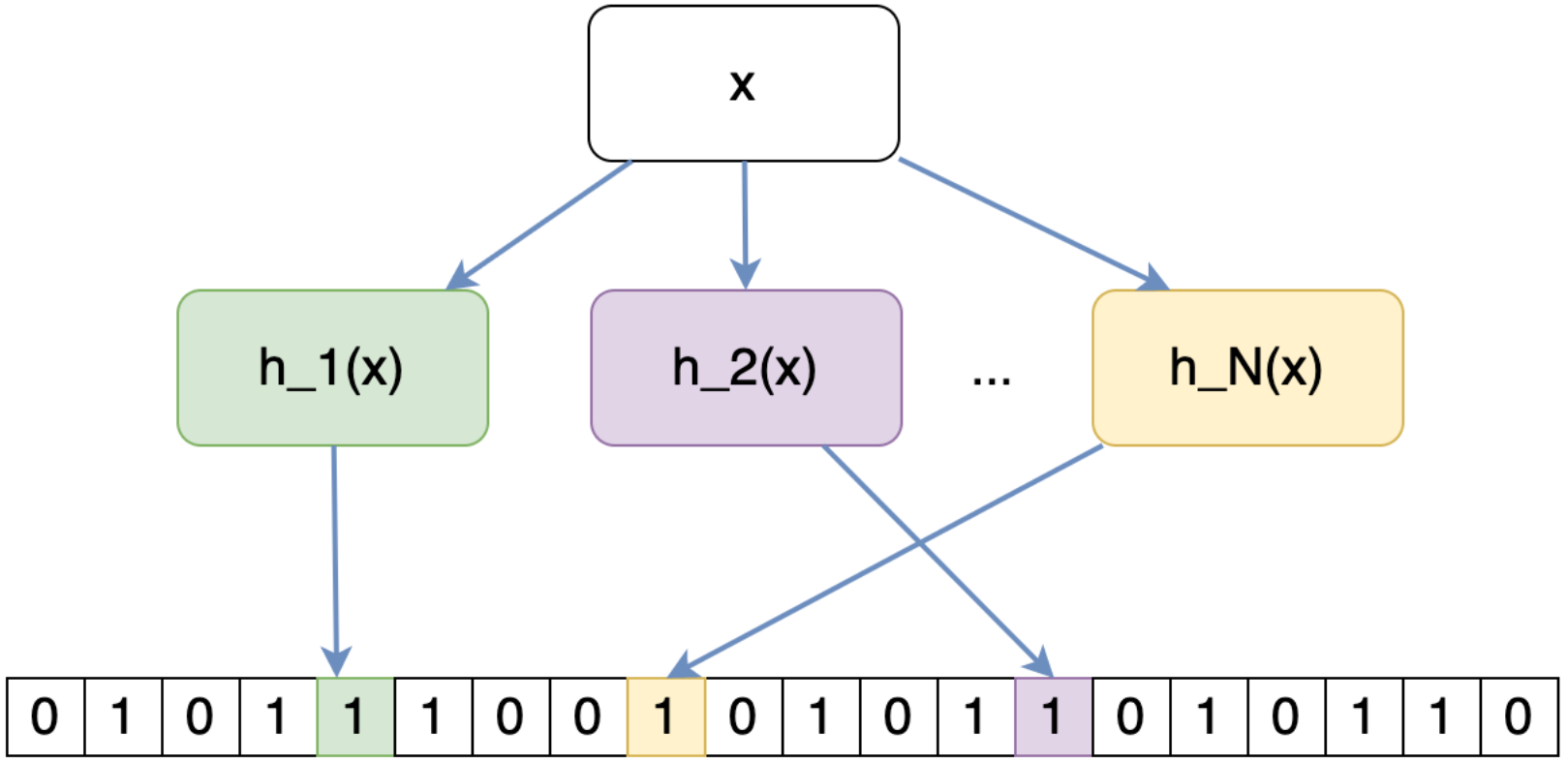



h(x):

1 => maybe yes

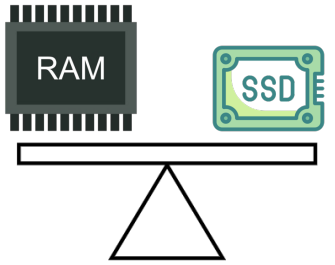
0 => no

# SSTable – Bloom filter




https://www.slideshare.net/doanduyhai/cassandra-data-structures-and-algorithms

# Bloom filters




- Per SSTable, immutable, stored on disk, loaded to memory, created during flush & compaction
- Loaded into off-heap (Native.malloc)

# Bloom filters

- Size depends on the number of different partition keys (not rows!)
- Size does NOT depend on the length of partition keys

# Bloom filters

- Size depends on the number of different partition keys (not rows!)
- Size does NOT depend on the length of partition keys
- Size depends on a configured error probability

  (parameter: `bloom_filter_fp_chance`), it is defined per table

# Bloom filters

- Size depends on the number of different partition keys (not rows!)
- Size does NOT depend on the length of partition keys
- Size depends on a configured error probability

  (parameter: `bloom_filter_fp_chance`), it is defined per table
- For Leveled compaction strategy: bloom_filter_fp_chance = 0.1
- For other compaction strategies: bloom_filter_fp_chance = 0.01

# Bloom filters

Monitoring:

- nodetool tablestats (“Bloom filter off heap memory used” property)
- JMX (`org.apache.cassandra.metrics`):
  - `type=Table,keyspace=KEYSPACE,scope=TABLE,`
    `name=BloomFilterOffHeapMemoryUsed`

# Bloom filters

| Number of partition keys | Size, MiB bloom_filter_fp_chance = 0.1 |
|---:|---:|
| 1’000’000 | 0.6 |
| 10’000’000 | 6.0 |
| 100’000’000 | 59.6 |
| 1’000’000’000 | 596 |

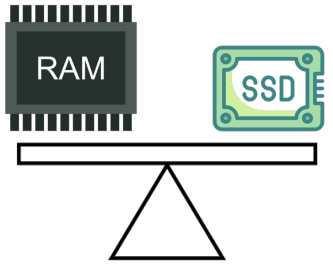
# Bloom filters

| Number of partition keys | Size, MiB bloom_filter_fp_chance = 0.1 | Size, MiB bloom_filter_fp_chance = 0.01 |
|---|---|---|
| 1’000’000 | 0.6 | 1.2 |
| 10’000’000 | 6.0 | 11.9 |
| 100’000’000 | 59.6 | 119.2 |
| 1’000’000’000 | 596 | 1192.1 |

Error 10x ⬇

Size 2x ⬆

# Bloom filters



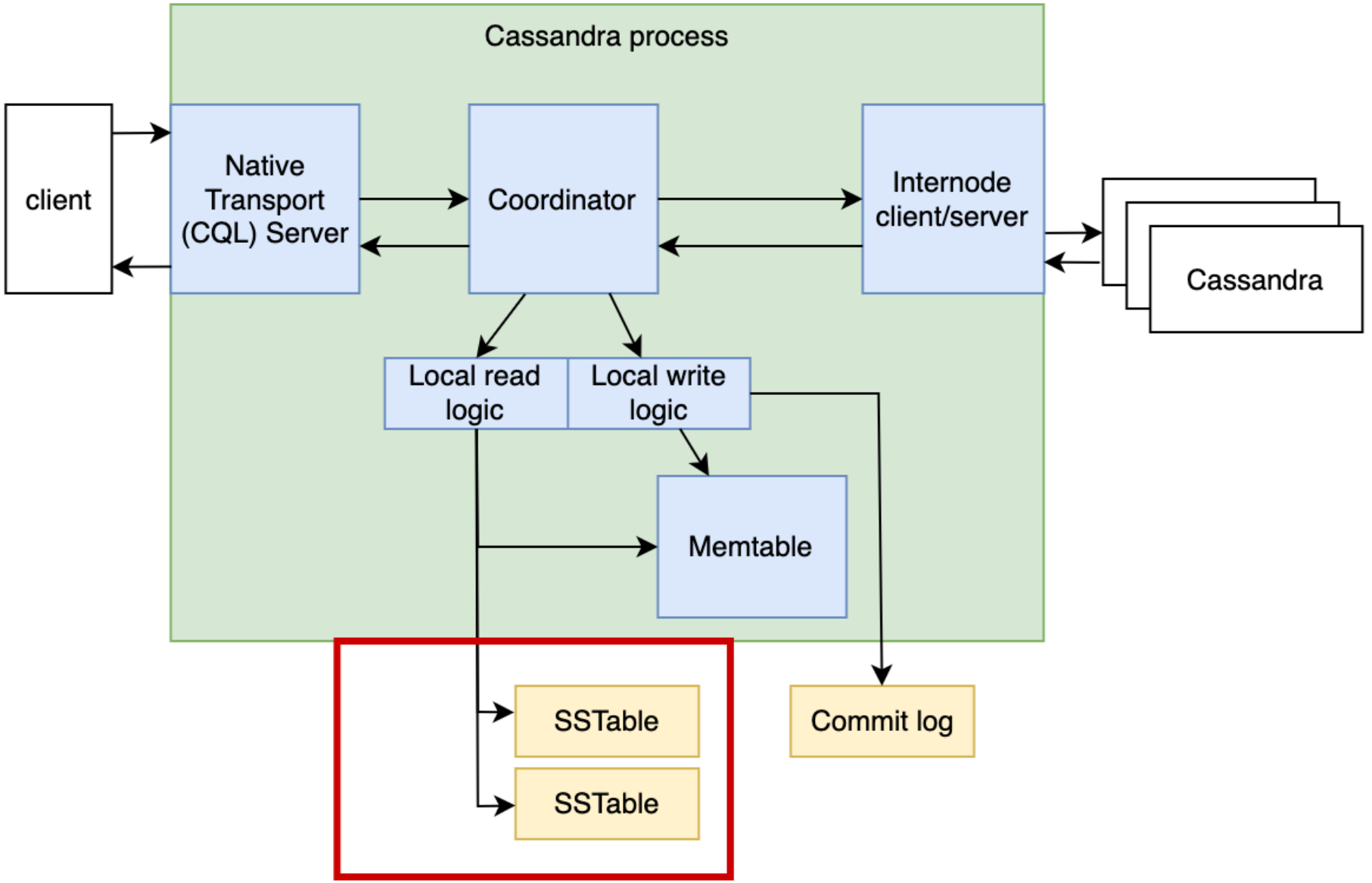

| Number of partition keys | Size, MiB bloom_filter_fp_chance = 0.1 | Size, MiB bloom_filter_fp_chance = 0.01 |
|---|---|---|
| 1’000’000 | 0.6 | 1.2 |
| 10’000’000 | 6.0 | 11.9 |
| 100’000’000 | 59.6 | 119.2 |
| 1’000’000’000 | 596 | 1192.1 |

- Calculator: https://hur.st/bloomfilter/?n=10000&p=1.0E-7&m=&k=
- org.apache.cassandra.utils.FilterFactory#createFilter

# Interim summary

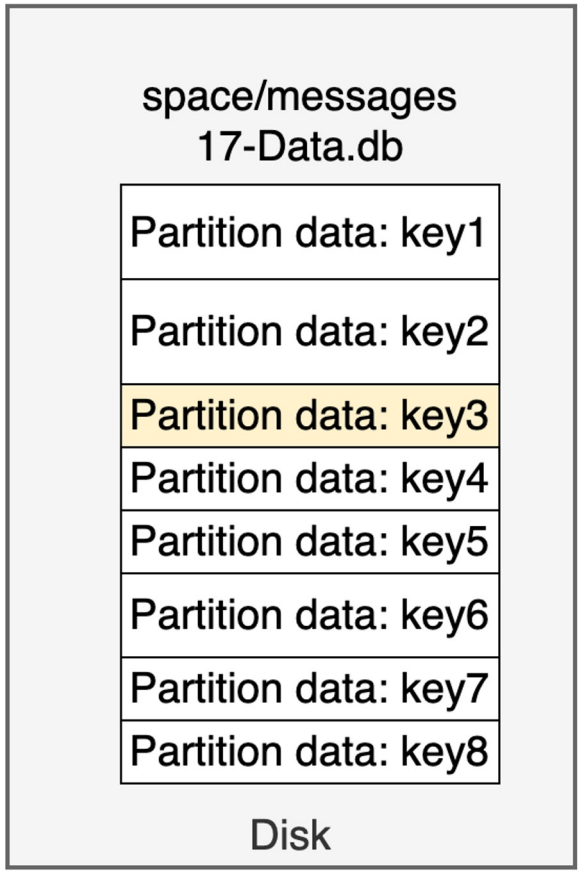
| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1% | cassandra.yaml | JMX, nodetool |
| Memtable | Heap | 2048 | 25 | memtable_heap_space | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| Memtable | Offheap | 2048 | 25 | memtable_offheap_space | JMX, nodetool |
| Bloom filters | Offheap | vary | vary | DDL | JMX, nodetool |
| | | | | | |
| | | | | | |


- JVM
- Network
- Coordinator
- Memtables
- **SSTables**
- Bloom filter
- **Index Summary**



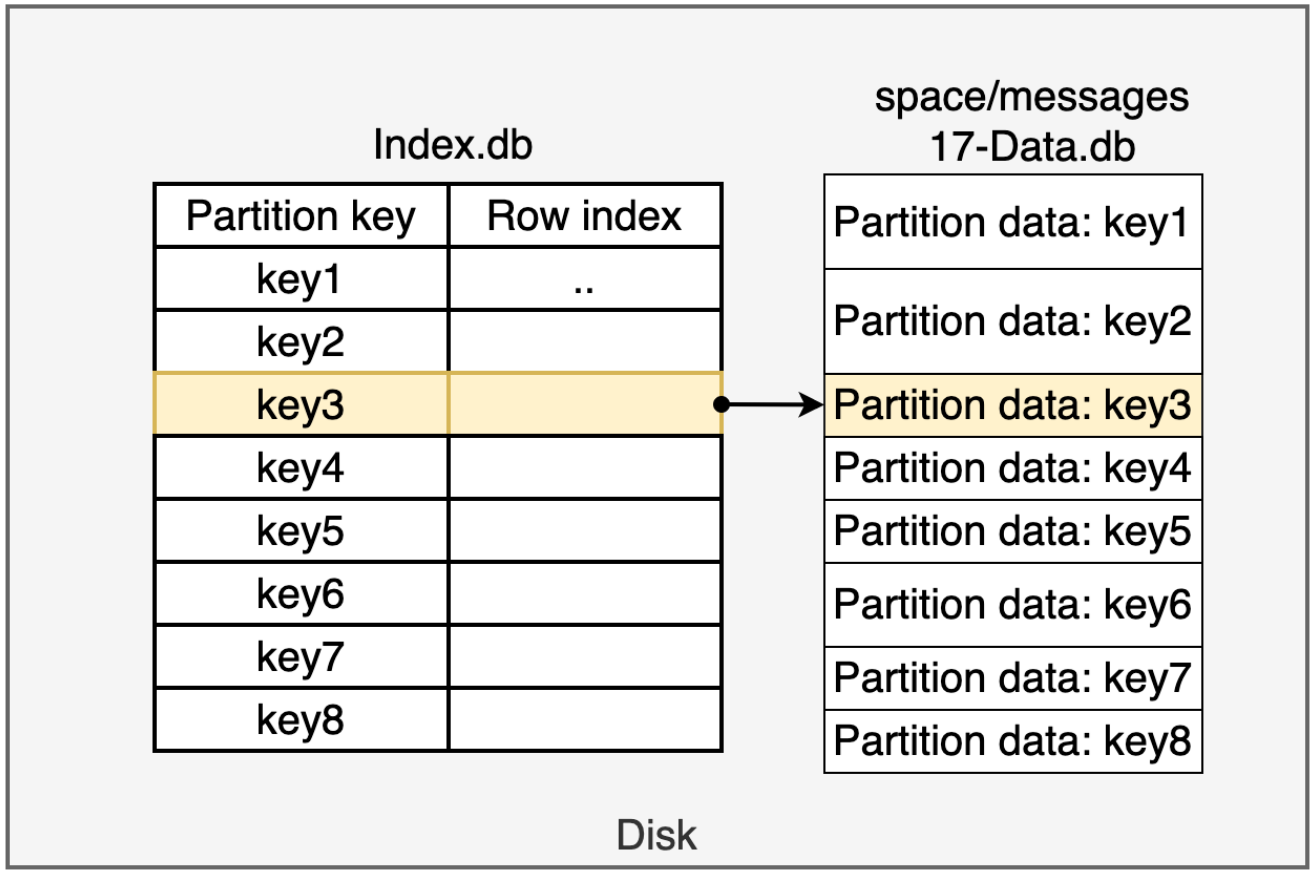
# SSTable read

We need to lookup data for our partition key



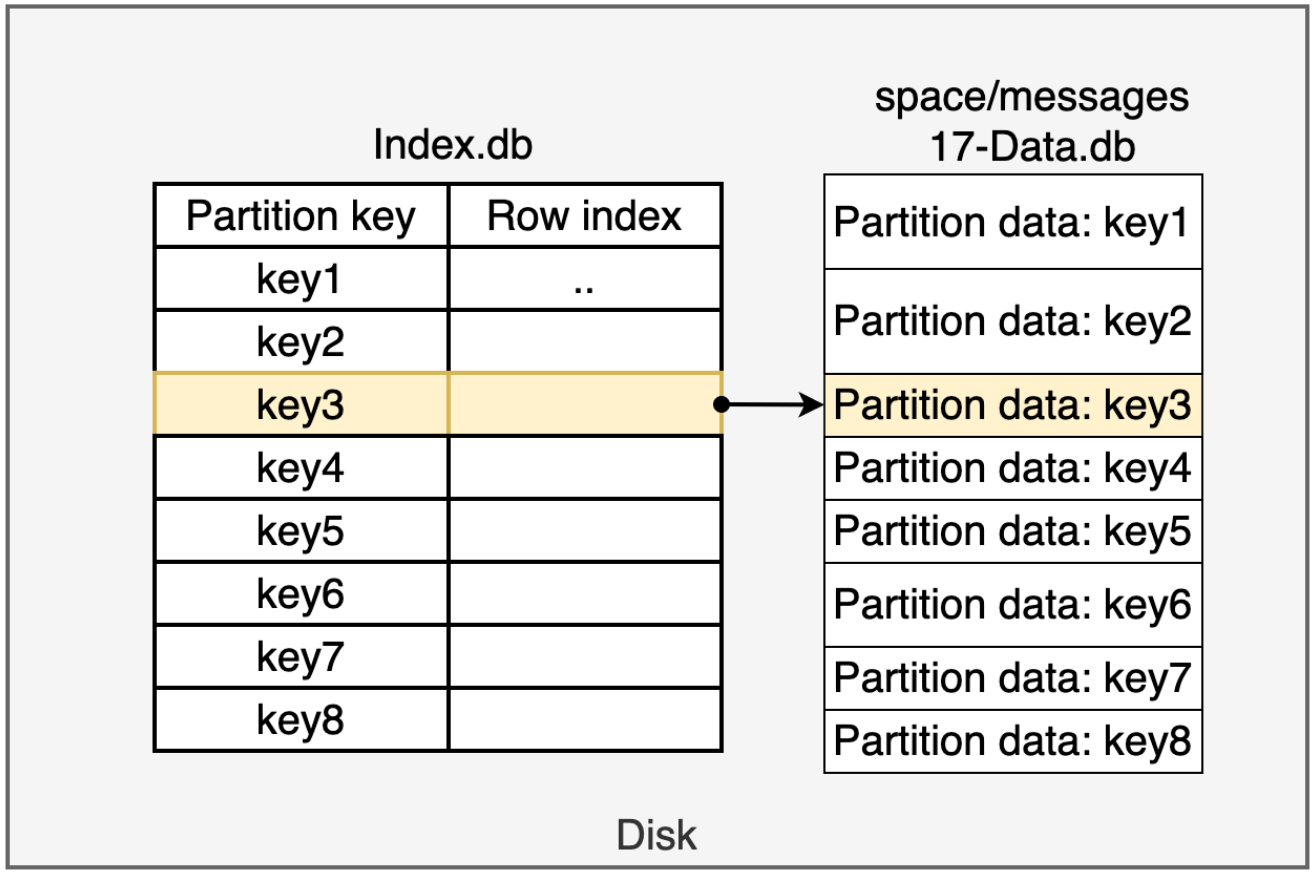
# SSTable index

Let’s add a sorted index

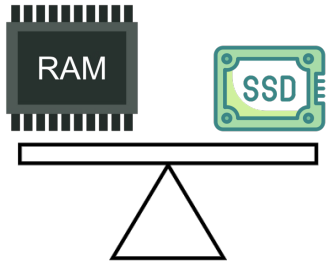


# SSTable indexed read

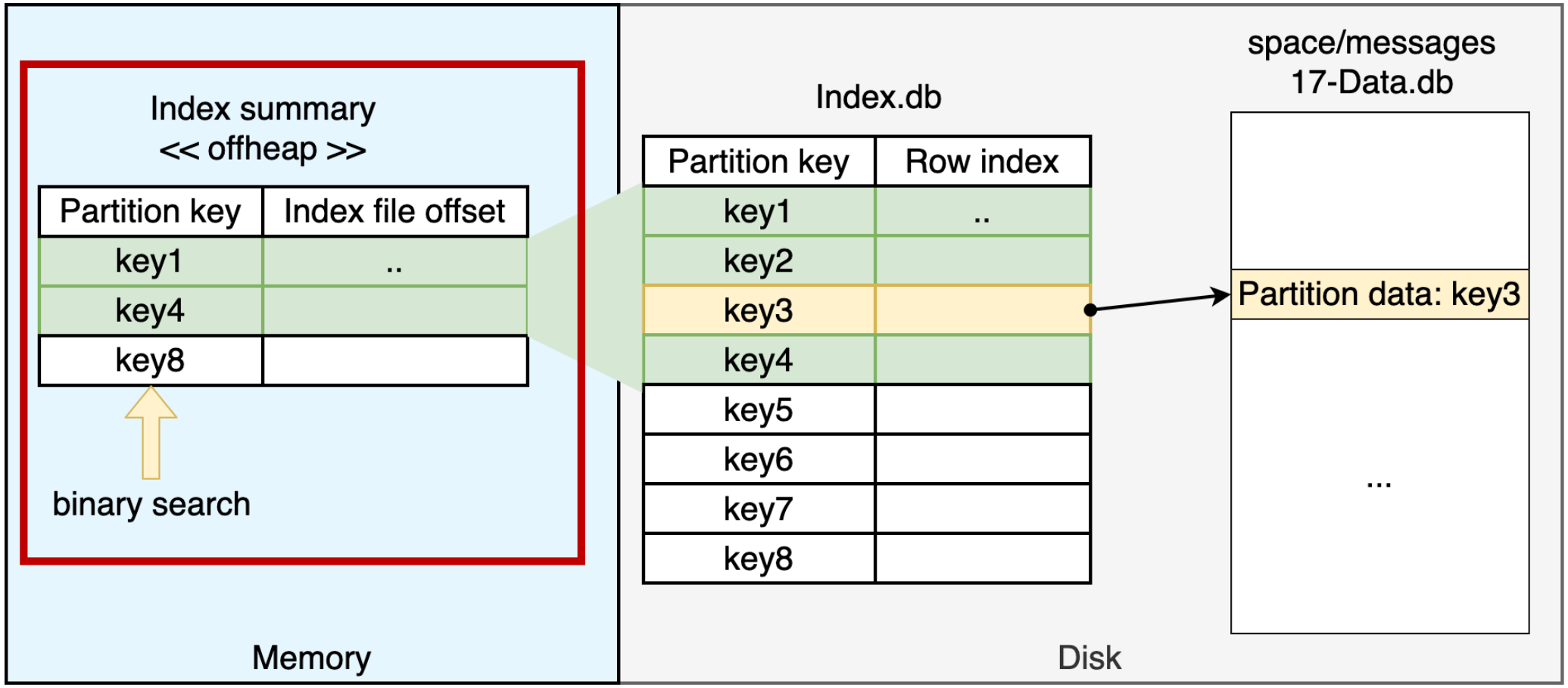
- We have to do log N random reads from disk to find the key (index is sorted)
- Index is smaller than Data file but still too large to load fully to memory



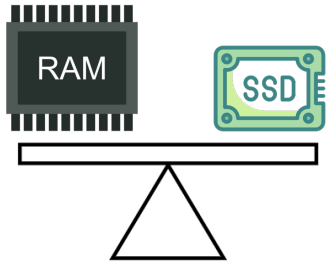
# SSTable Index Summary



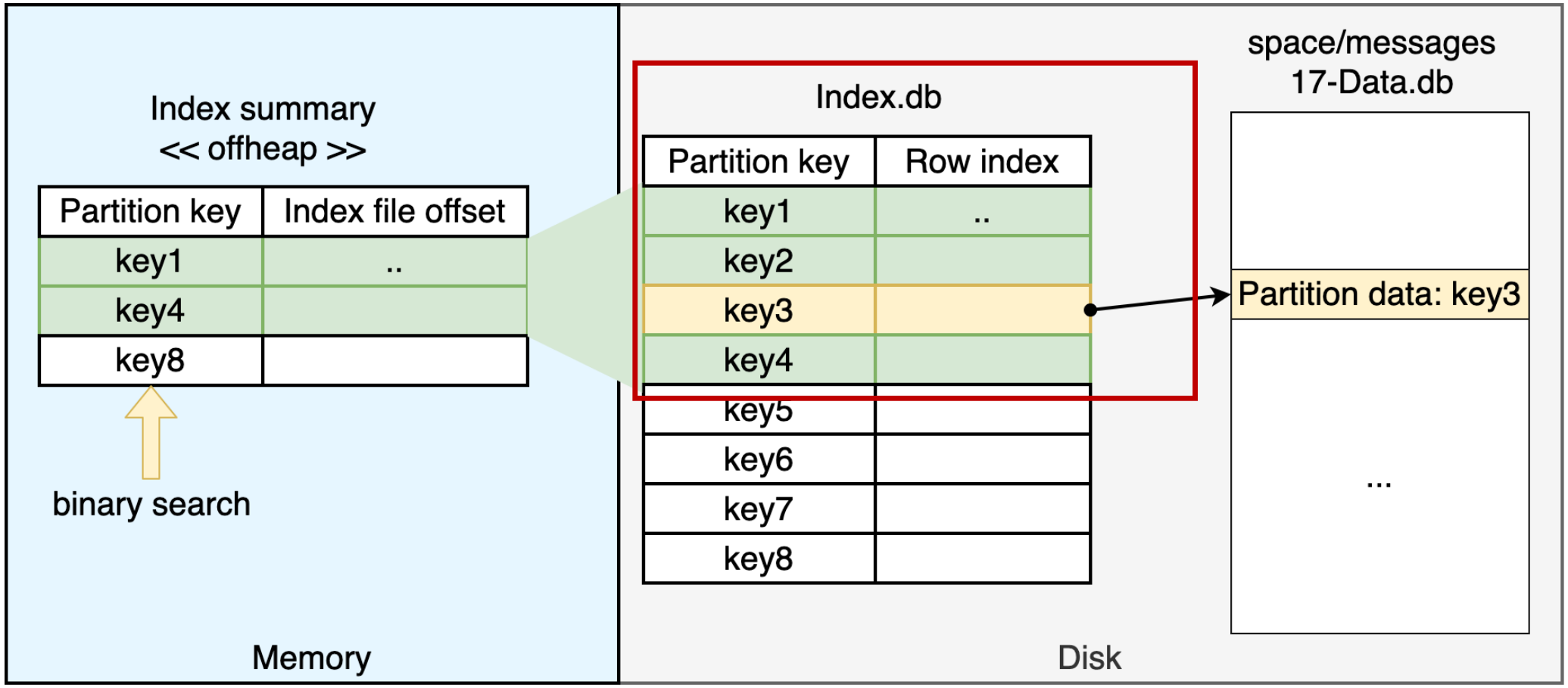

- Let's load to memory every N keys from Index ("index summary") to identify an approximate location of the exact index record
- Than scan the index to find the exact key



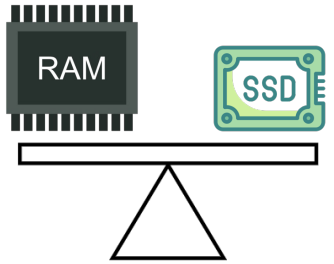
# SSTable Index Summary



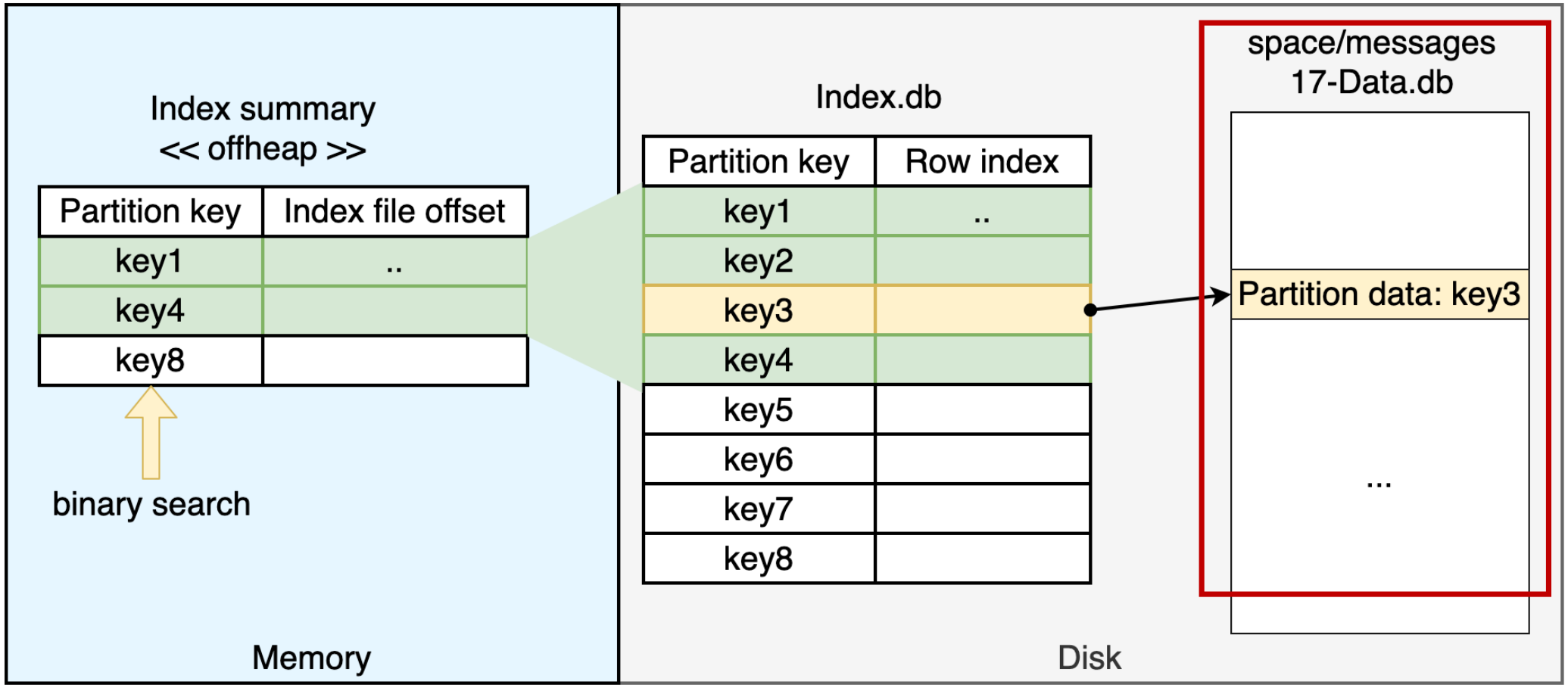

- Let's load to memory every N keys from Index ("index summary") to identify an approximate location of the exact index record
- Than scan the index to find the exact key



# SSTable Index Summary




- Let's load to memory every N keys from Index ("index summary") to identify an approximate location of the exact index record
- Than scan the index to find the exact key



# SSTable Index Summary

- Index summary is stored in off-heap (Native.malloc) + on disk (to load on startup)
- In case of a memory lack – index interval for colder SSTables (read rate) is increasing

# SSTable Index Summary

- Index summary is stored in off-heap (Native.malloc) + on disk (to load on startup)
- In case of a memory lack – index interval for colder SSTables (read rate) is increasing
- Cassandra.yaml:
  - index_summary_capacity_in_mb (default: 5% of heap size)
  - index_summary_resize_interval (default: 60 min)
- Table level settings: min_index_interval = 128 / max_index_interval = 2048

# SSTable Index Summary

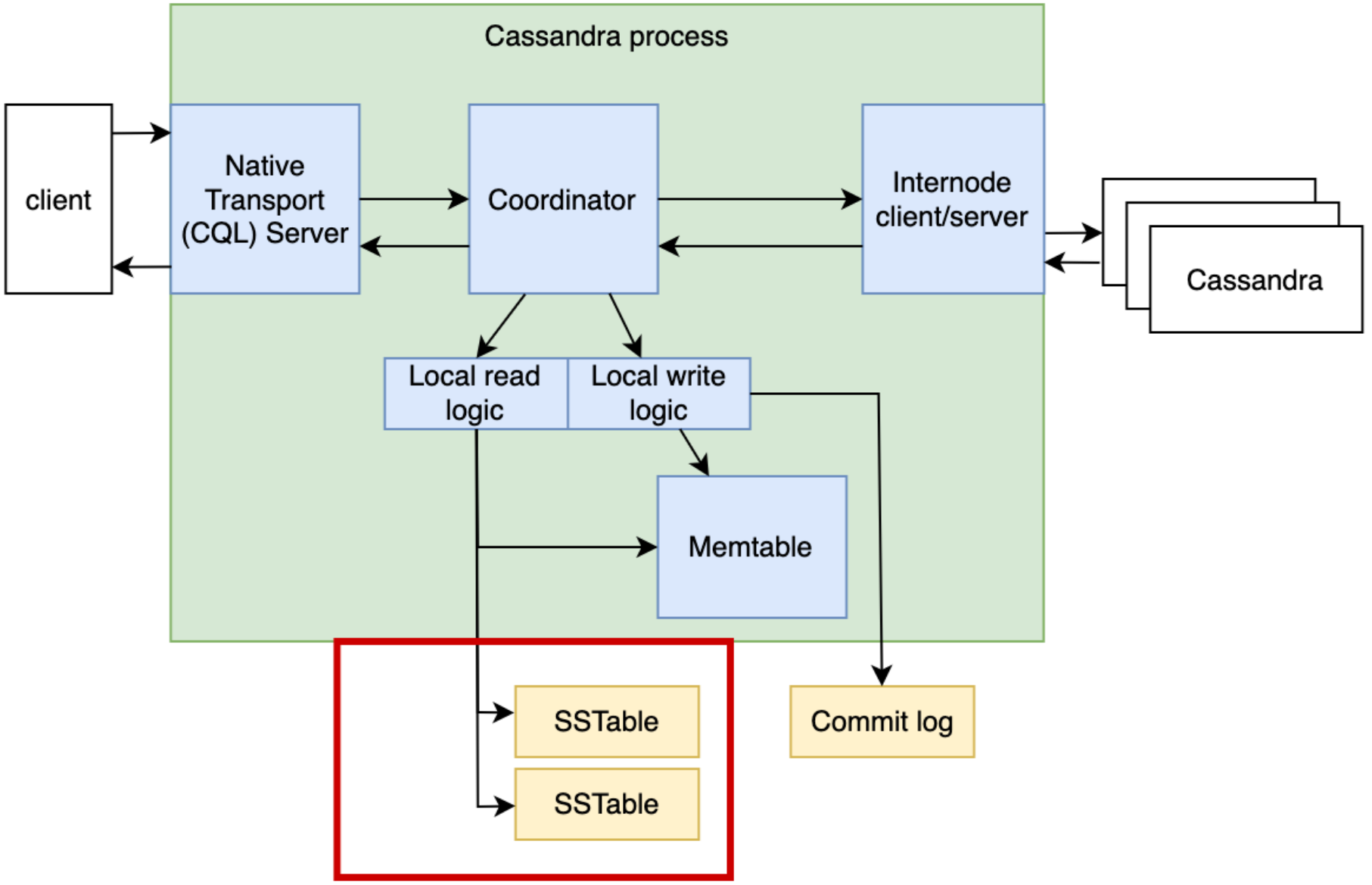
Monitoring:

- nodetool tablestats (“Index summary off heap memory used” property)
- JMX (`apache.cassandra.metrics`):
  - `type=Table,keyspace=KEYSPACE,scope=TABLE,name=IndexSummaryOffHeapMemoryUsed`
  - `type=Compaction,name=IndexSummaryRedistributionTime`

# Interim summary

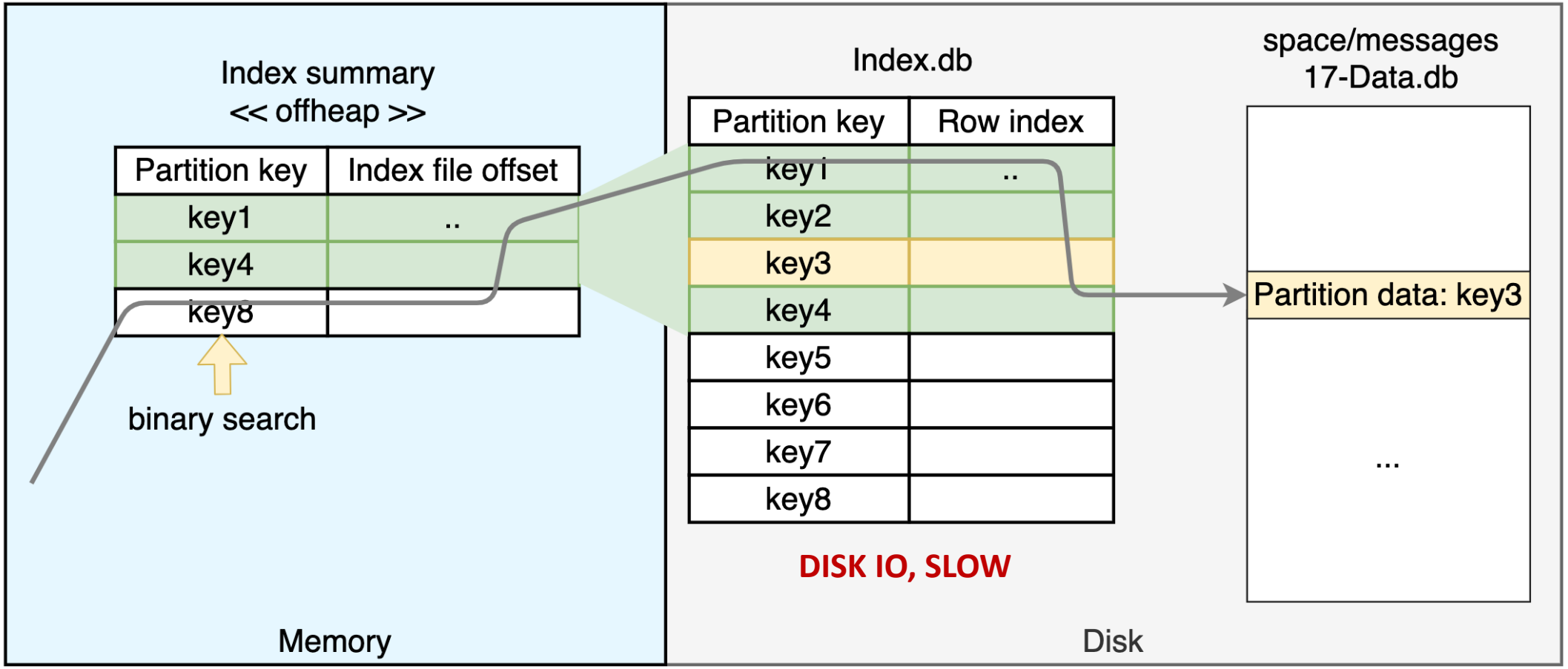
| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1 | cassandra.yaml | JMX, nodetool |
| Memtable | Heap | 2048 | 25 | memtable_heap_space | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| Memtable | Offheap | 2048 | 25 | memtable_offheap_space | JMX, nodetool |
| Bloom filters | Offheap | vary | vary | DDL | JMX, nodetool |
| Index Summary | Offheap | 410 | 5 | cassandra.yaml + DDL | JMX, nodetool |


- JVM
- Network
- Coordinator
- Memtables
- **SSTables**
- Bloom filter
- Index Summary
- **Key cache**



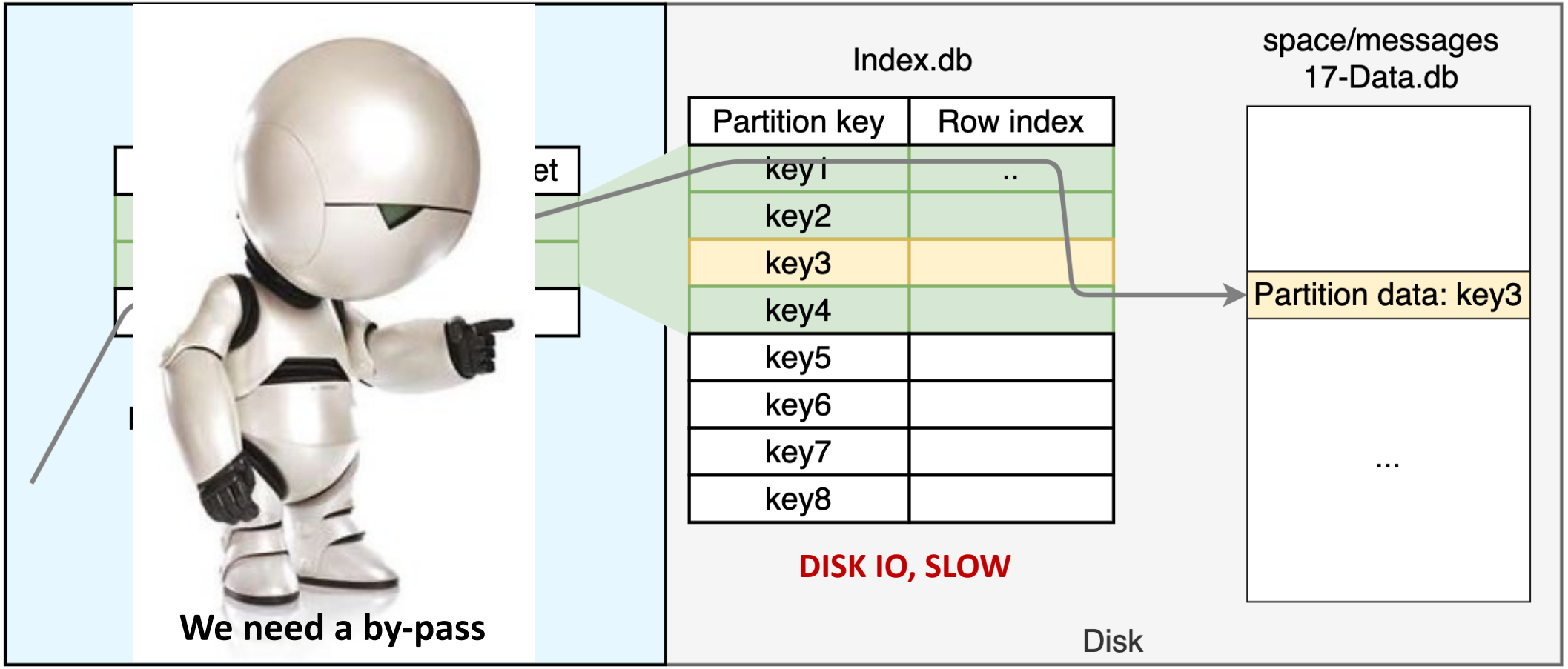
# SSTable read

Even with an index summary we have to do extra disk reads from Index

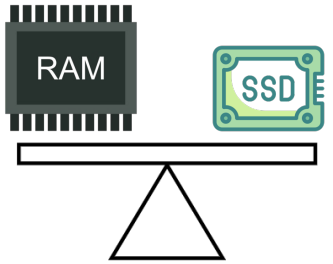


# SSTable read

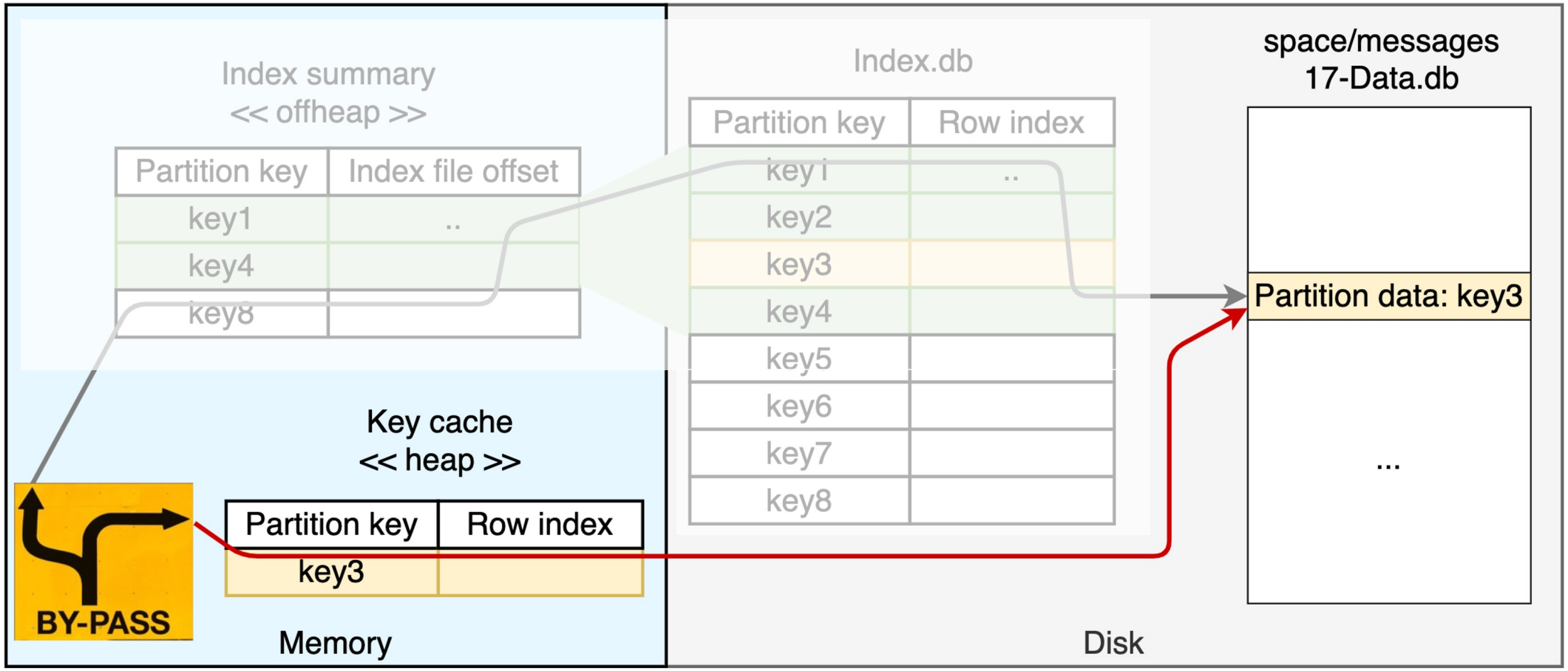
Even with an index summary we have to do extra disk reads from Index



# SSTable – key cache




Let’s cache the result of index lookup



# SSTable – key cache

- Stored in heap
- Periodically flushed to disk (to load on startup)
- Based on Caffeine library again

# SSTable – key cache

- Stored in heap
- Periodically flushed to disk (to load on startup)
- Based on Caffeine library again
- Configuration:
  - Cassandra.yaml:
    - key_cache_size_in_mb ( default: min(5% of Heap (in MB), 100MB) )
    - key_cache_save_period
    - key_cache_keys_to_save
  - Table level: caching = {'keys': ‘ALL | NONE’}

# SSTable – key cache

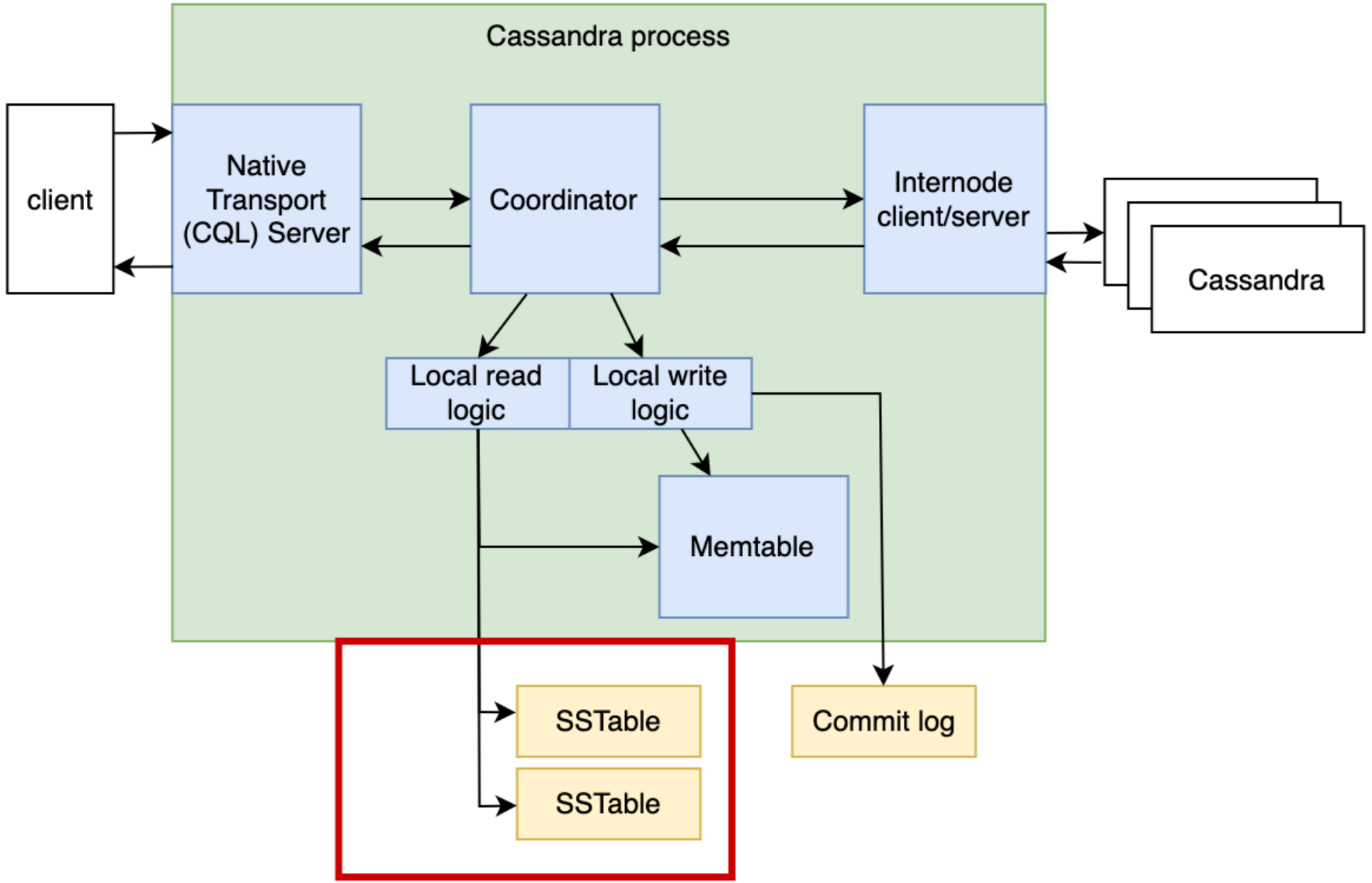
Monitoring:

- nodetool info (“Key Cache” property)
- JMX (`org.apache.cassandra.metrics:`)
  - `type=Table,keyspace=KEYSPACE,scope=TABLE,name=KeyCacheHitRate`
  - `type=Cache,scope=KeyCache,name=Size`
  - `type=Cache,scope=KeyCache,name=HitRate`
  - `type=Cache,scope=KeyCache,name=Misses`
  - `type=Cache,scope=KeyCache,name=Requests`
    - …

# Interim summary

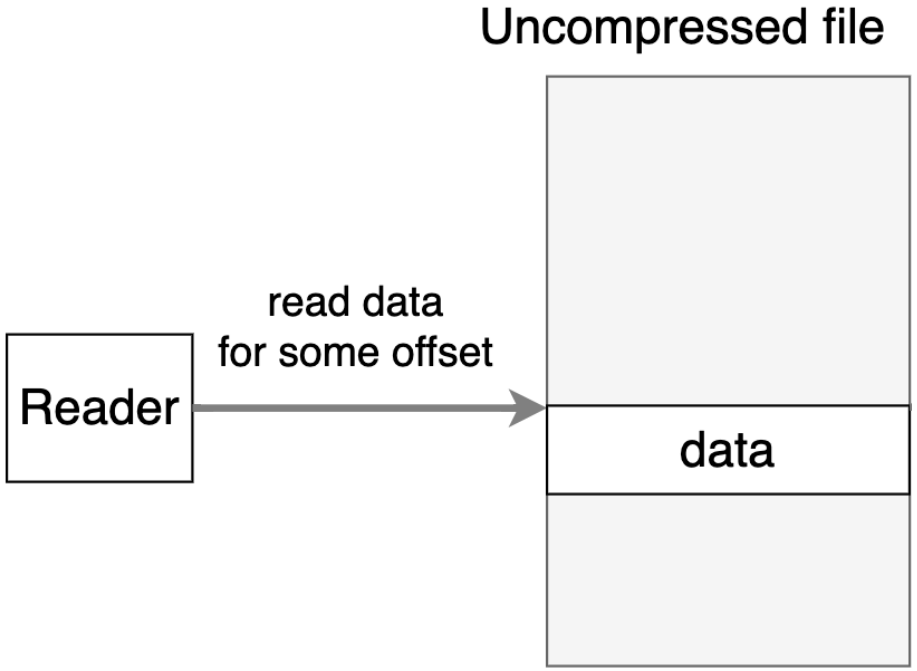
| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1 | cassandra.yaml | JMX, nodetool |
| Memtable | Heap | 2048 | 25 | memtable_heap_space | JMX, nodetool |
| Key cache | Heap | 100 | 1 | Cassandra.yaml, DDL | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| Memtable | Offheap | 2048 | 25 | memtable_offheap_space | JMX, nodetool |
| Bloom filters | Offheap | vary | vary | DDL | JMX, nodetool |
| Index Summary | Offheap | 410 | 5 | cassandra.yaml + DDL | JMX, nodetool |


- JVM
- Network
- Coordinator
- Memtables
- **SSTables**
- Bloom filter
- Index Summary
- Key cache
- **Compression metadata**



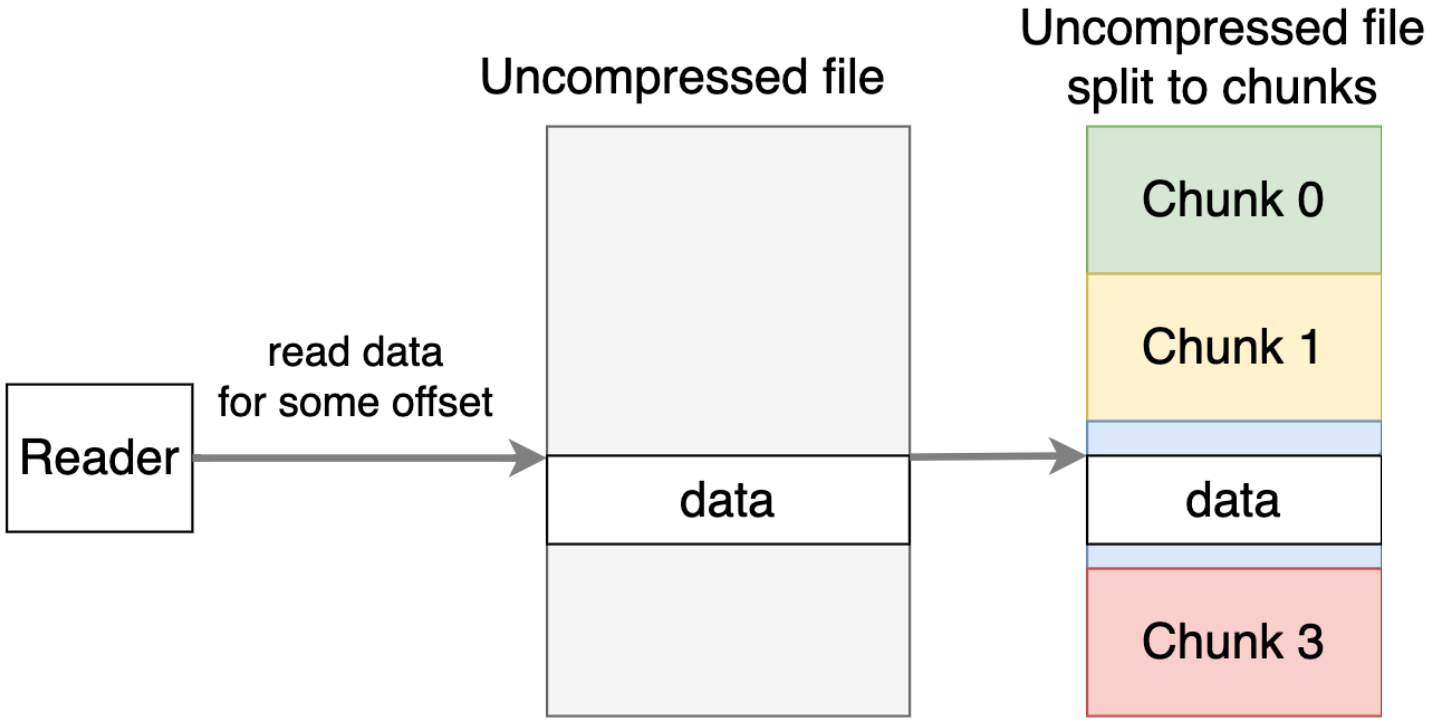
# Compressed SSTable data read

- SSTable Data files are compressed by chunks
- We know a logical position in non-compressed data - how to find the correspondent position to read in a compressed file?



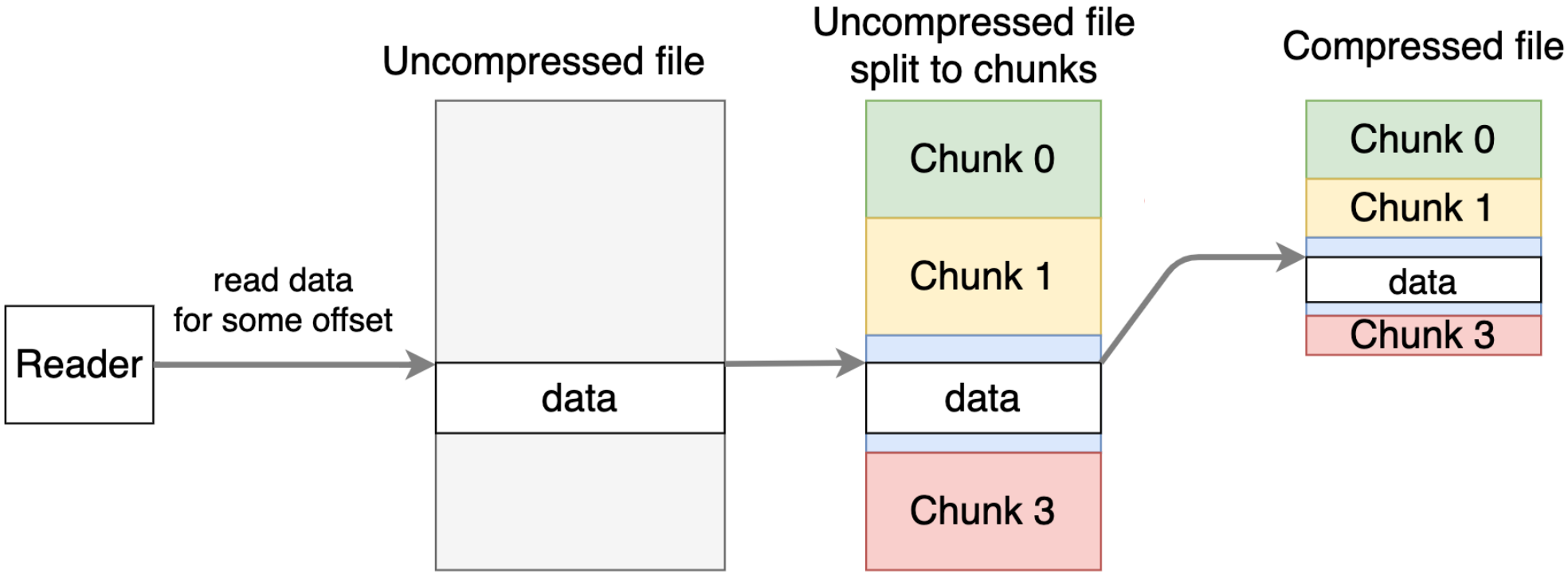
# Compressed SSTable data read

- SSTable Data files are compressed by chunks
- We know a logical position in non-compressed data - how to find the correspondent position to read in a compressed file?



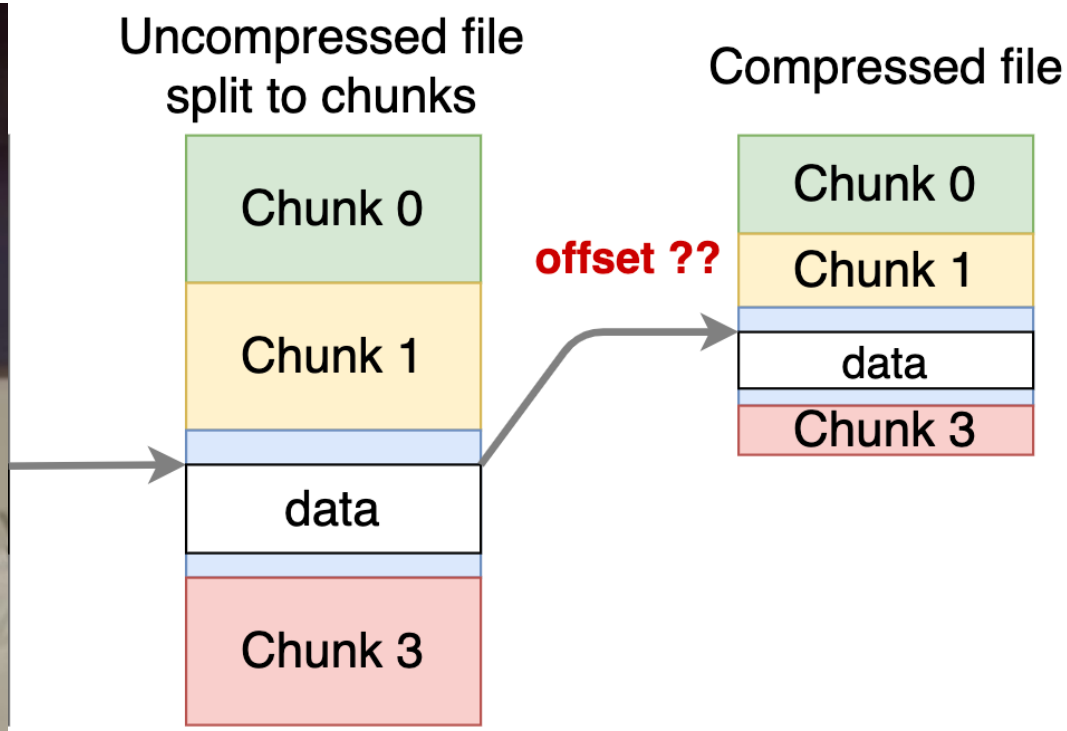
# Compressed SSTable data read

- SSTable Data files are compressed by chunks
- We know a logical position in non-compressed data - how to find the correspondent position to read in a compressed file?



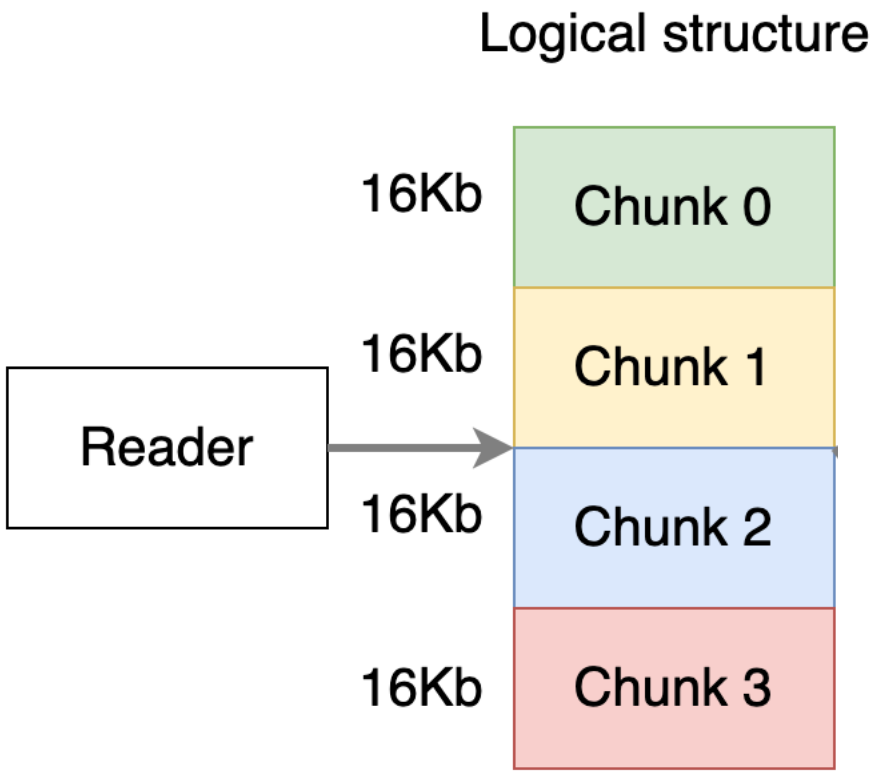
# Compressed SSTable data read

- SSTable Data files are compressed by chunks
- We know a logical position in non-compressed data - how to find the correspondent position to read in a compressed file?



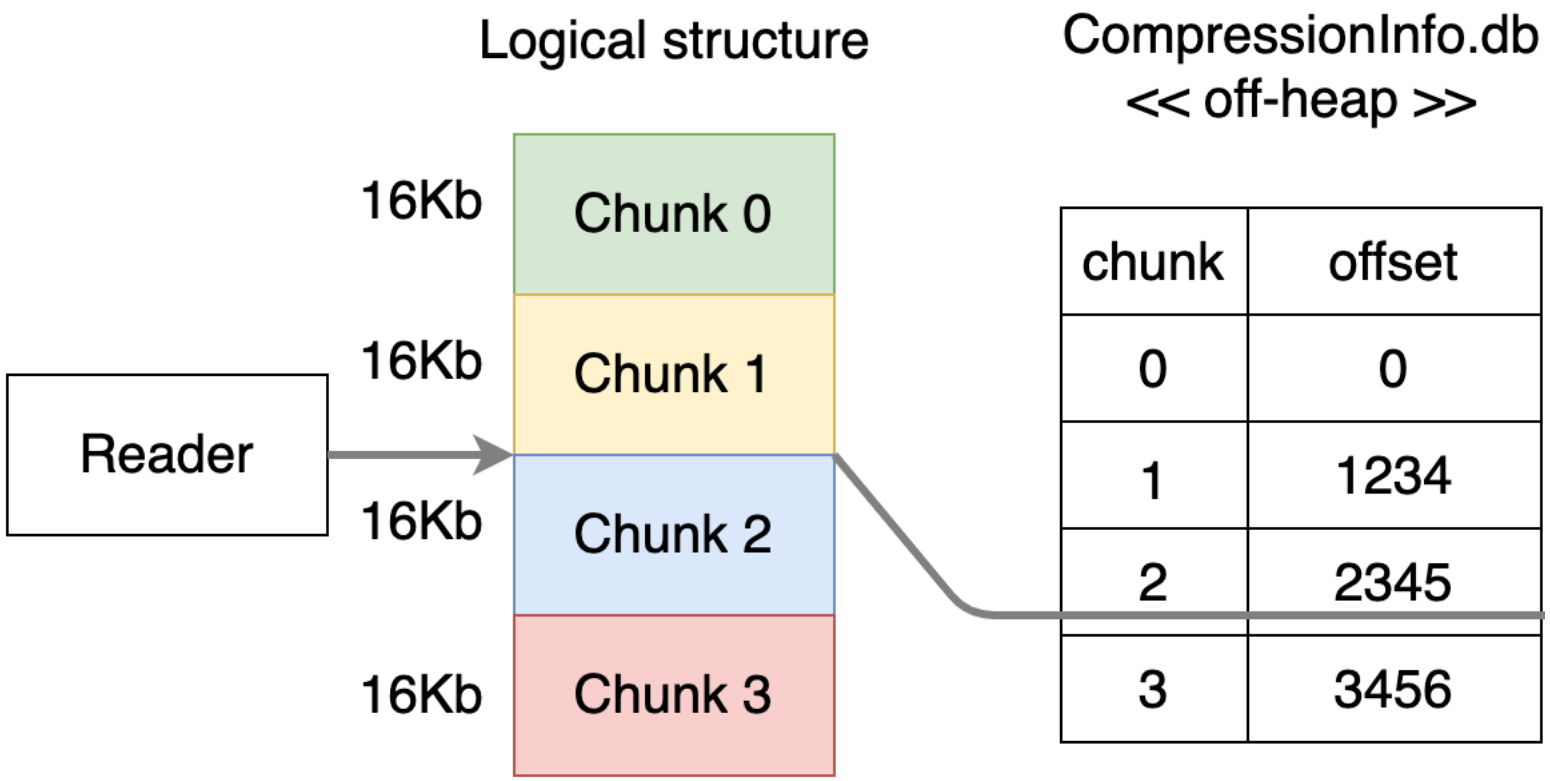
# Compression info

- We need a mapping to do a translation



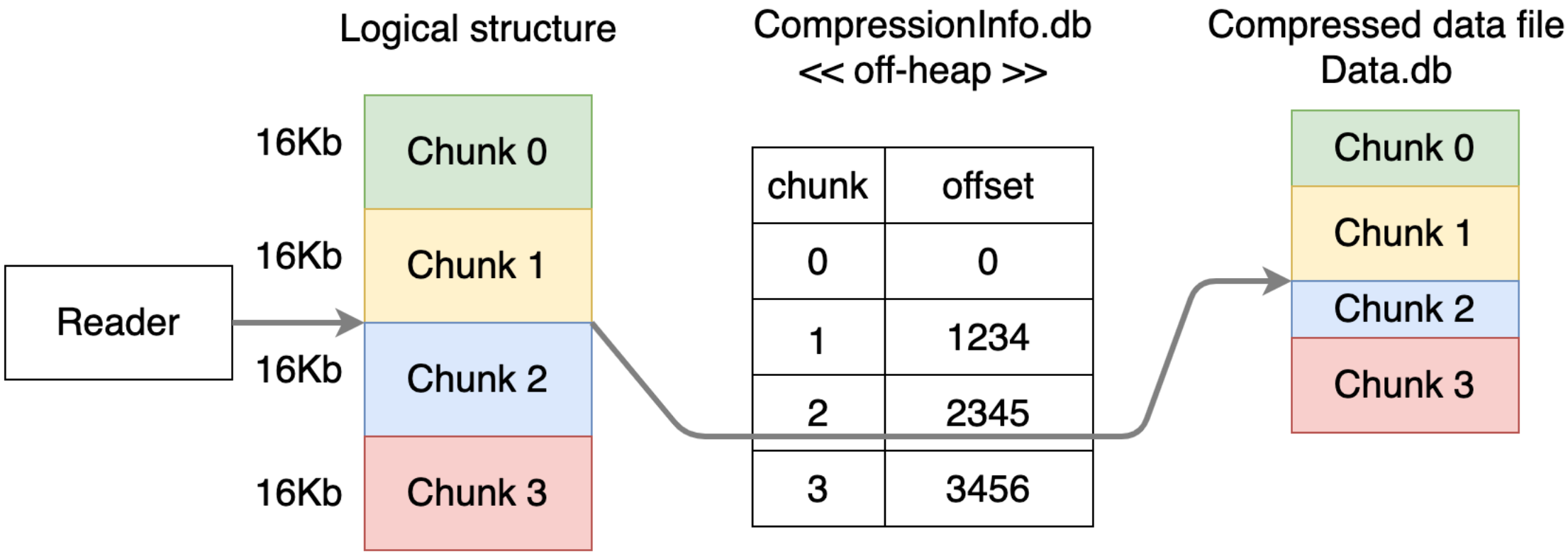
# Compression info

- We need a mapping to do a translation

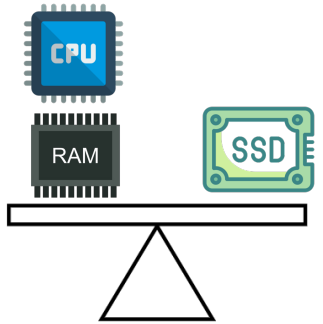


# Compression info

- We need a mapping to do a translation

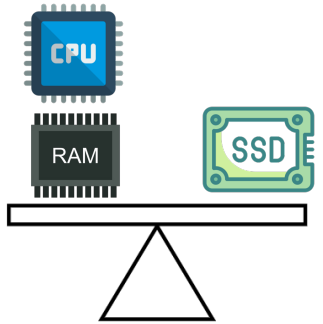


# Compression info




- Stored in off-heap (Native.malloc) + persisted on disk
- Amount of data: 8 bytes per chunk
- Not required if SSTable compression disabled
- Depends on a chunk size, smaller chunk -> more chunks -> more offsets to store

# Compression info

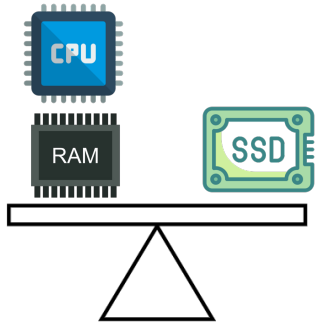



- Stored in off-heap (Native.malloc) + persisted on disk
- Amount of data: 8 bytes per chunk
- Not required if SSTable compression disabled
- Depends on a chunk size, smaller chunk -> more chunks -> more offsets to store

| SSTable size* | Chunk length = 16 KiB |
|---:|---:|
| 1 TiB | 64 MiB |
| 100 GiB | 6.4 MiB |
| 10 GiB | 640 KiB |
| 1GiB | 64 KiB |
| 100 MiB | 6.4 KiB |

*uncompressed

# Compression info




- Stored in off-heap (Native.malloc) + persisted on disk
- Amount of data: 8 bytes per chunk
- Not required if SSTable compression disabled
- Depends on a chunk size, smaller chunk -> more chunks -> more offsets to store

| SSTable size* | Chunk length = 16 KiB | Chunk length = 4 KiB |
|---|---|---|
| 1 TiB | 64 MiB | 256 MiB |
| 100 GiB | 6.4 MiB | 25.5 MiB |
| 10 GiB | 640 KiB | 2560 KiB |
| 1GiB | 64 KiB | 256 KiB |
| 100 MiB | 6.4 KiB | 25.6 KiB |

x4 more

*uncompressed

Trade-off: disk read size vs memory usage

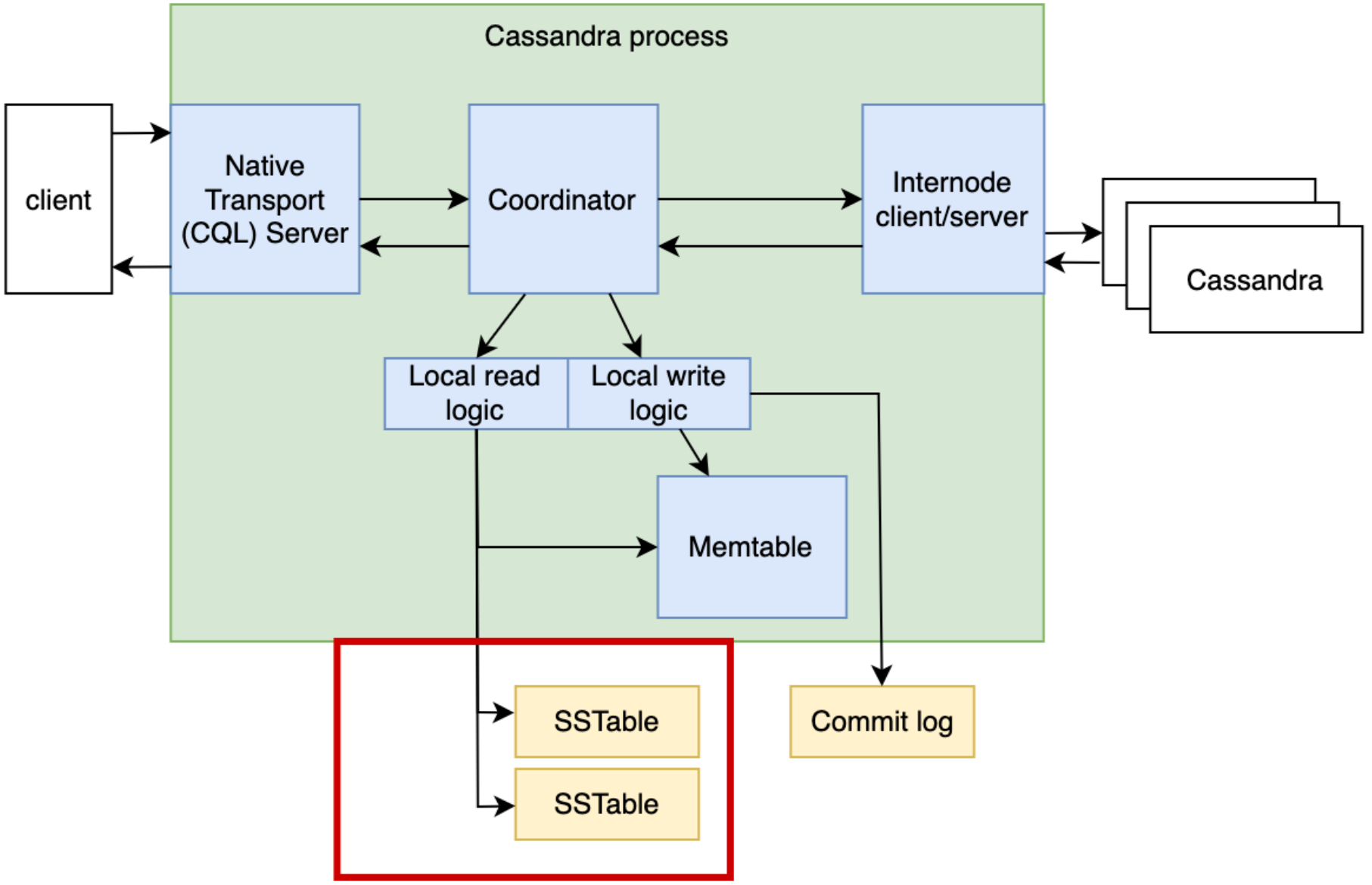
# Compression info

- Configuration:
  - Table level: compression = {'chunk_length_in_kb': '4', 'class': '...LZ4Compressor’}
- Monitoring:
  - nodetool tablestats (”Compression metadata off heap memory used” property)
  - JMX: `org.apache.cassandra.metrics:type=Table,keyspace=KEYSPACE,scope=TABLE,name=CompressionMetadataOffHeapMemoryUsed`

# Interim summary

| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1 | cassandra.yaml | JMX, nodetool |
| Memtable | Heap | 2048 | 25 | memtable_heap_space | JMX, nodetool |
| Key cache | Heap | 100 | 1 | Cassandra.yaml, DDL | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| Memtable | Offheap | 2048 | 25 | memtable_offheap_space | JMX, nodetool |
| Bloom filters | Offheap | vary | vary | DDL | JMX, nodetoool |
| Index Summary | Offheap | 410 | 5 | cassandra.yaml + DDL | JMX, nodetool |
| Compression metadata | Offheap | vary | vary | DDL | JMX, nodetool |


- JVM
- Network
- Coordinator
- Memtables
- **SSTables**
- ...
- **OS Page Cache**



# OS page cache

- Cassandra keeps OS file caching (OS page cache) enabled
- It is a primary caching level for data in Cassandra now

# OS page cache

- Cassandra keeps OS file caching (OS page cache) enabled
- It is a primary caching level for a data in Cassandra now
- Cassandra evicts old SSTables from the cache after compaction

  native int posix_fadvise(fd, offset, len, POSIX_FADV_DONTNEED)
- Keep it as much as you can (of course, <= live data set)

# OS page cache

- Monitoring
  - free (“buff/cache” column), vmstat
  - How to view which tables are cached - https://github.com/tobert/pcstat
  - eBPF/bcc
    - https://www.brendangregg.com/linuxperf.html
    - https://www.brendangregg.com/blog/2021-08-30/high-rate-of-paging.html

```
./pcstat ./data/data/*/*/*-Data.db
|----------------------------------------+--------+-------+--------+---------|
| Name                                   | Size   | Pages | Cached | Percent |
|----------------------------------------+--------+-------+--------+---------|
| system_auth/roles-/nb-1-big-Data.db    | 102    | 1     | 1      | 100.000 |
| system/compaction_history/nb-486-big-Data.db  | 21814  | 6     | 6      | 100.000 |
...
```

# Summary

| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1 | cassandra.yaml | JMX, nodetool |
| Memtable | Heap | 2048 | 25 | memtable_heap_space | JMX, nodetool |
| Key cache | Heap | 100 | 1 | Cassandra.yaml, DDL | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| Memtable | Offheap | 2048 | 25 | memtable_offheap_space | JMX, nodetool |
| Bloom filters | Offheap | vary | vary | DDL | JMX, nodetoool |
| Index Summary | Offheap | 410 | 5 | cassandra.yaml + DDL | JMX, nodetool |
| Compression metadata | Offheap | vary | vary | DDL | JMX, nodetool |
| OS page cache | Off-heap | Vary | vary | OS level | free, pcstat, eBPF |

# Summary

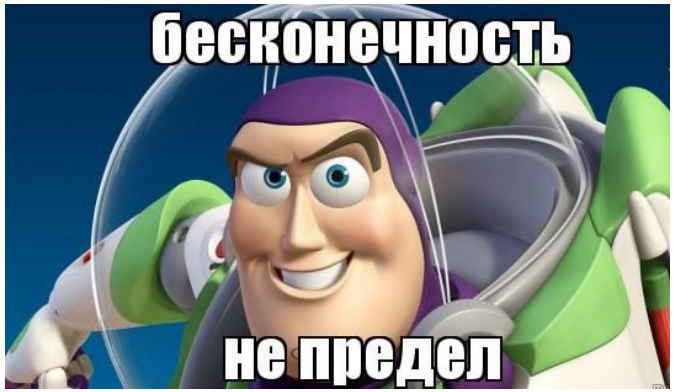
- Cassandra memory is much more than just Java heap

# Summary

- Cassandra memory is much more than just Java heap
- Memory footprint depends on
  - Configuration
  - Schema
  - Amount of stored data
  - Network activity

# Summary

- Cassandra memory is much more than just Java heap
- Memory footprint depends on
  - Configuration
  - Schema
  - Amount of stored data
  - Network activity
- Cassandra memory usage is improving, try newer versions



# Summary

- Cassandra memory is much more than just Java heap
- Memory footprint depends on
  - Configuration
  - Schema
  - Amount of stored data
  - Network activity
- Cassandra memory usage is improving, try newer versions
- Some of the described techiniques can be applicable in your applications (such as rate limiting, Caffeine, Bloom filters, reducing the number of objects, etc)

# The end

# It is not the end of the story

We have not covered all memory consumers due to lack of time:

- Chunk cache
- Row cache
- Commit log
- Hints
- Repair
- LWT
- Security
- …

# Additional slides

# Memory allocation in application code

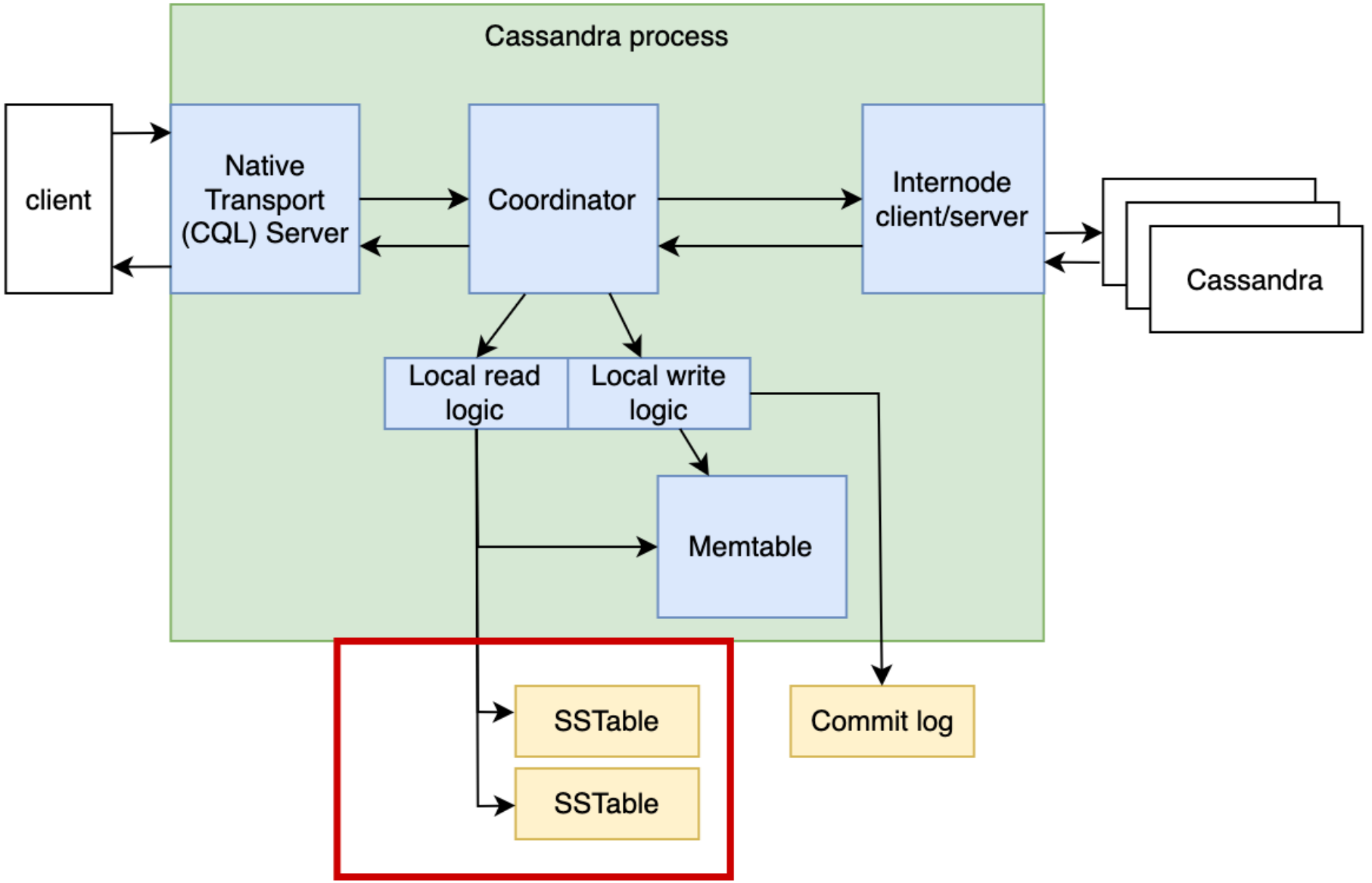
- In-heap allocation: new Object(); new int[10];
- Off-heap allocation:
  - DirectByteBuffer (+)
  - Memory mapped files (+)
    - Commit log write
    - SSTable read (disk_access_mode)
  - Unsafe.allocateMemory (+)
    - Netty uses it
    - JEP draft: Deprecate Memory-Access Methods in sun.misc.Unsafe for Removal
  - Native (JNI, JNA)
    - com.sun.jna.Native#malloc (+)
  - Memory segment (JEP 454: Foreign Function & Memory API ) (-)
  - OS-level structures: sockets, file descriptors, etc (+)

# Memory usage in memtables

- 20’000 partitions x 10 clustering keys = 200’000 rows
- Partition key: TEXT, 10 symbols
- Clustering key: TEXT, 10 symbols
- Live data: data + metadata

| Case | Raw data | Live data | Heap objects | Offheap_buffers | | Offheap_objects | | On disk (LZ4) |
|---|---|---|---|---|---|---|---|---|
| | | | Heap | Heap | Offheap | Heap | Offheap | |
| 10 TEXT columns, 10 symbols | 100 + 10 + 1 | 291 | 1236 | 1301 | 111 | 413 | 328 | 130 |
| 1 TEXT column, 100 symbols | 100 + 10 + 1 | 147 | 404 | 325 | 111 | 157 | 139 | 112 |
| 10 TIMESTAMP columns | 80 + 10 + 1 | 271 | 1216 | 1301 | 91 | 413 | 308 | 101 |
| 10 INT columns | 40 + 10 + 1 | 231 | 1176 | 1301 | 951 | 413 | 268 | 61 |
| UDT, 10 TEXT fields, 10 symbols, non-frozen | 100 + 10 + 1 | 323 | 1944 | 2149 | 131 | 461 | 388 | 146 |
| UDT, 10 TEXT fields, 10 symbols, frozen | 100 + 10 + 1 | 187 | 444 | 325 | 151 | 157 | 179 | 132 |


- JVM
- Network
- Coordinator
- Memtables
- **SSTables**
- ..
- Compression metadata
- **Chunk cache**



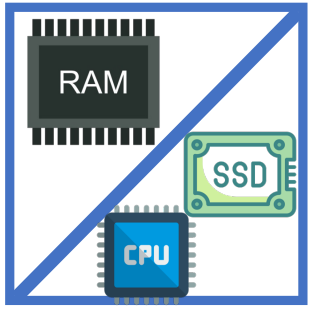
# SSTable – chunk cache

- To read data we need to:
  - Read compressed chunks from disk
  - Decompress
- Can we make it faster?

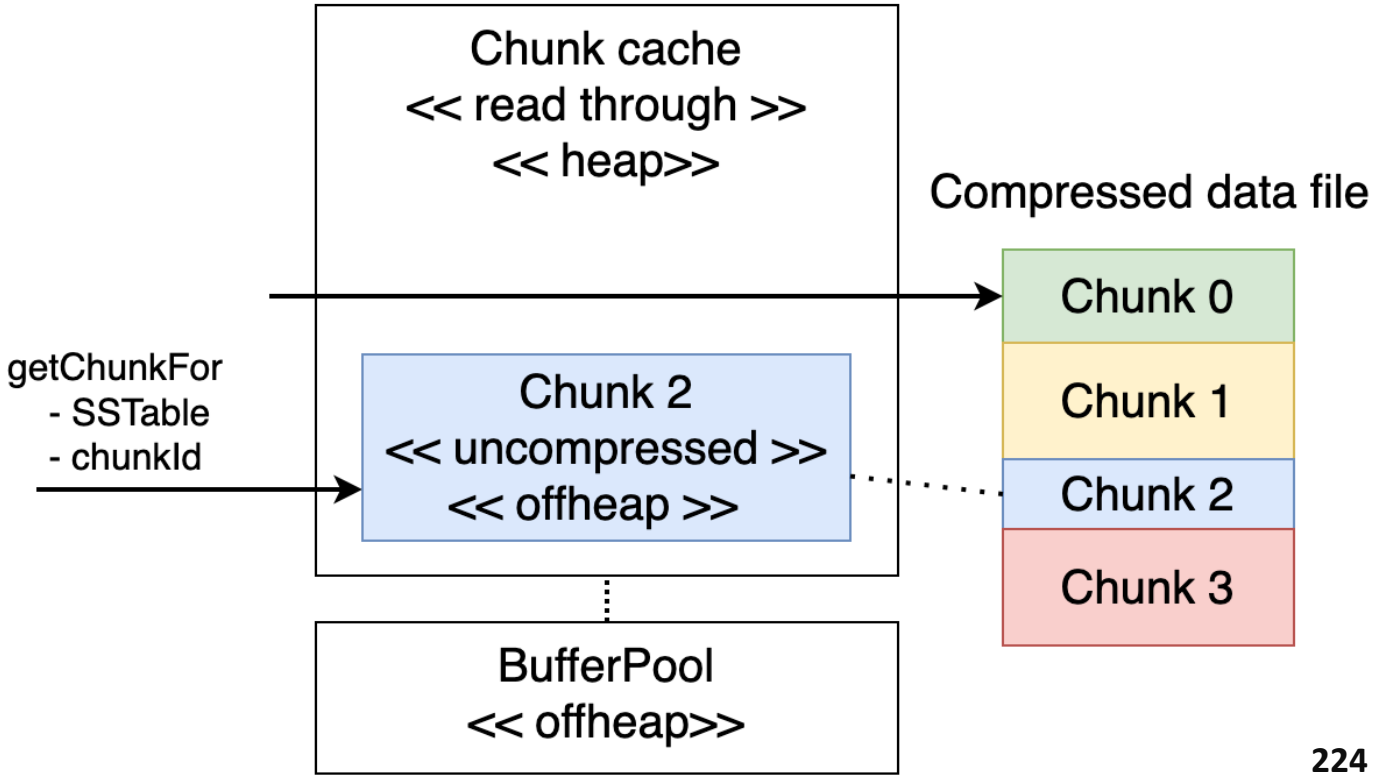
# SSTable – chunks reading/decompression

- To read data we need to:
  - Read compressed chunks from disk
  - Decompress
- Can we cache the results?

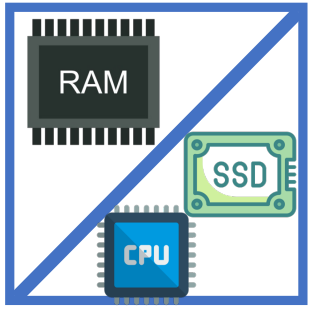
# SSTable – chunk cache



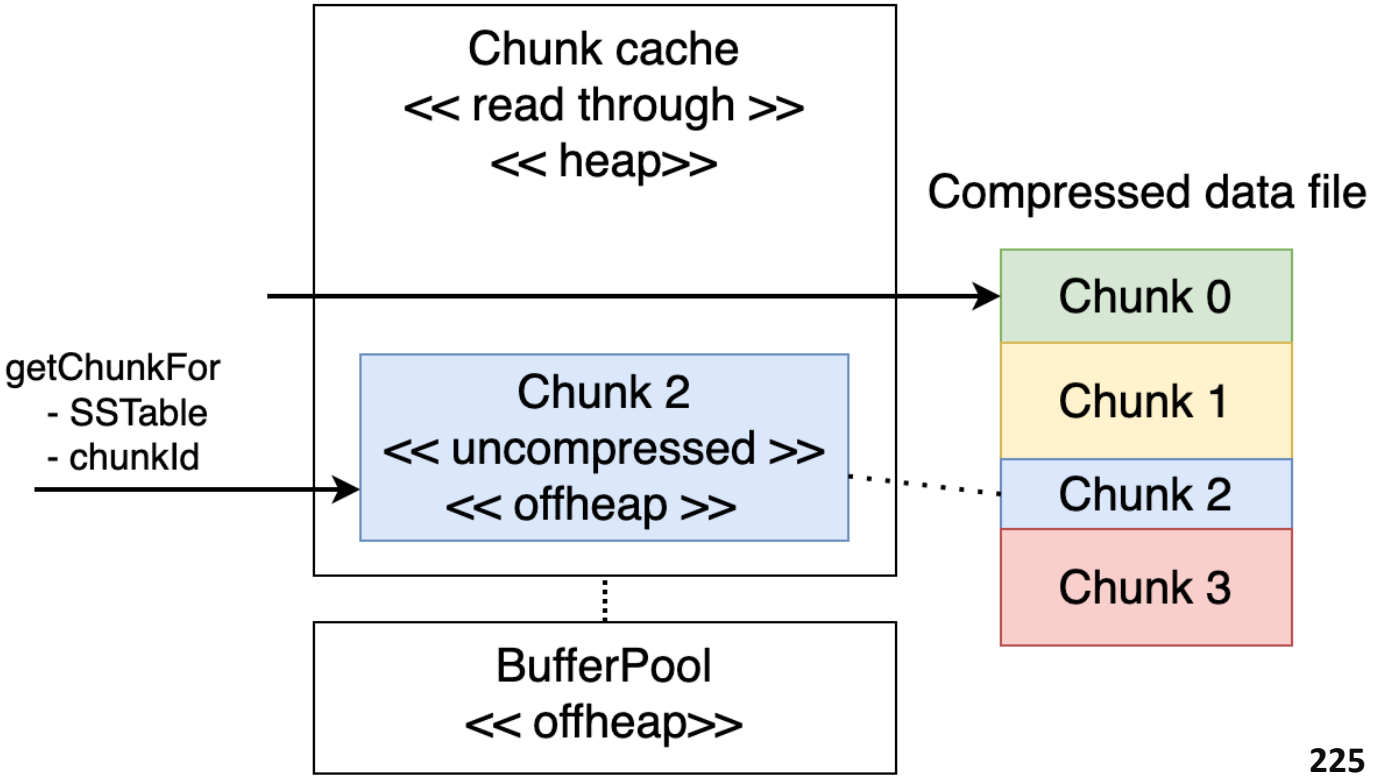

- CPU/disk vs memory trade-off
- Caffeine caching library
- Cache is shared for all SSTables



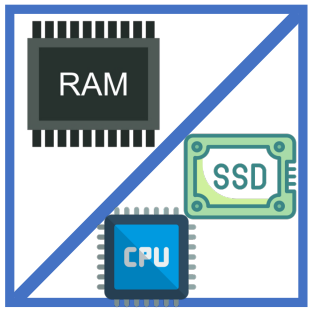
# SSTable – chunk cache




- CPU/disk vs memory trade-off
- Caffeine caching library
- Cache is shared for all SSTables
- Cache data – off-heap (Native.malloc)
- Cache structure – heap (128 bytes per chunk)
- Compactions evict data for old compacted SSTables

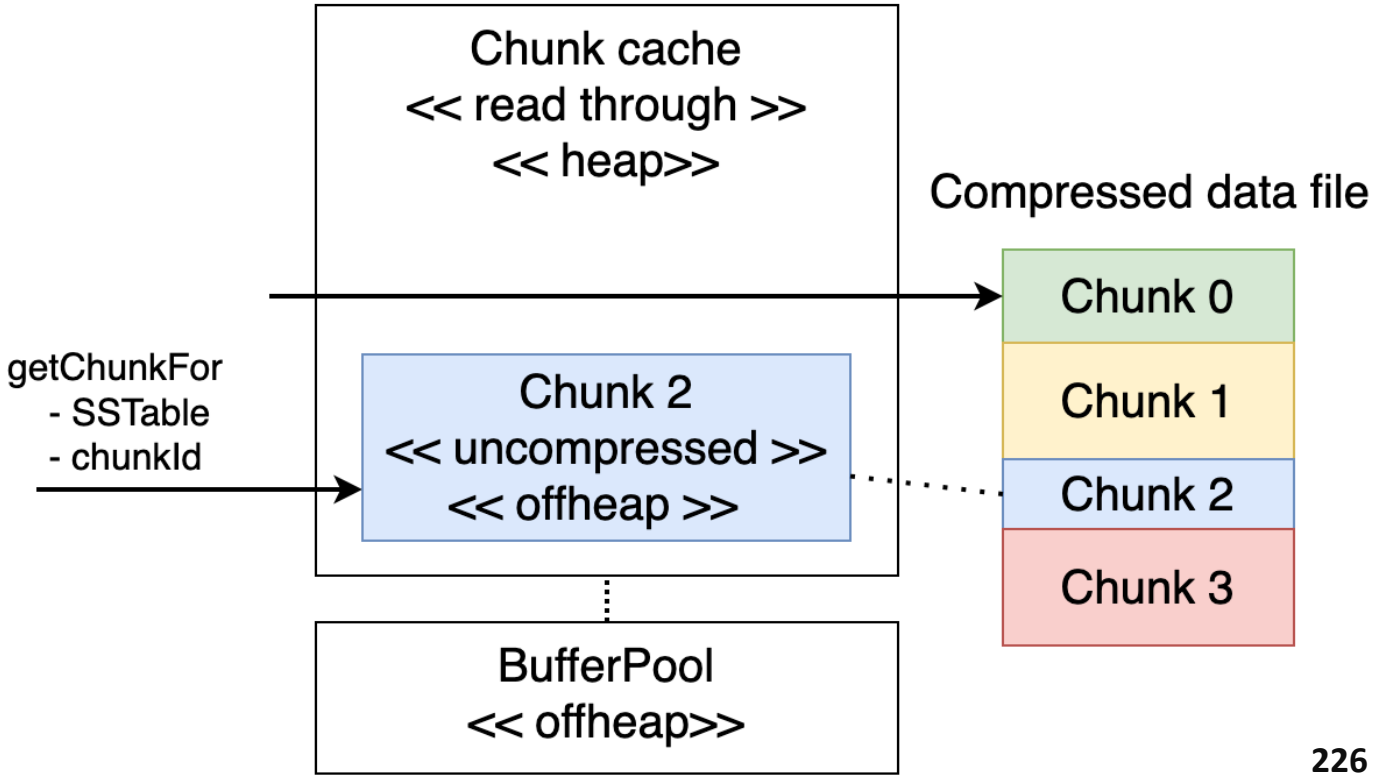


# SSTable – chunk cache




- Can make latency worser ☹

  see: CASSANDRA-16036
- Disabled by default since 4.0



# SSTable – chunk cache

- Configuration (cassandra.yaml)
  - file_cache_enabled (default: **false**, since 4.0)
  - file_cache_size (default: min(512Mb, 25% of heap size) ) – it is about off-heap data size
- Monitoring
  - nodetool info, “Chunk Cache” property
  - JMX
    - org.apache.cassandra.metrics:type=BufferPool,scope=chunk-cache,name=Capacity
    - org.apache.cassandra.metrics:type=BufferPool,scope=chunk-cache,name=Hits
    - org.apache.cassandra.metrics:type=BufferPool,scope=chunk-cache,name=Misses
    - org.apache.cassandra.metrics:type=BufferPool,scope=chunk-cache,name=OverflowSize
    - org.apache.cassandra.metrics:type=BufferPool,scope=chunk-cache,name=Size
    - org.apache.cassandra.metrics:type=BufferPool,scope=chunk-cache,name=UsedSize

# Interim summary

| Structures type | Heap / offheap | Size, MiB | Size, % of heap | How to configure | How to monitor |
|---|---|---|---|---|---|
| Heap | Heap | 8192 | 100 | JVM flags | JMX, NMT |
| Request processing | Heap | vary | vary | cassandra.yaml | JMX, nodetool |
| Prepared stmt cache | Heap | 32 | < 1 | cassandra.yaml | JMX, nodetool |
| Memtable | Heap | 2048 | 25 | memtable_heap_space | JMX, nodetool |
| Key cache | Heap | 100 | 1 | Cassandra.yaml, DDL | JMX, nodetool |
| Chunk cache | Heap | 0 | 0 | Cassandra.yaml | JMX, nodetool |
| JVM structures | Offheap | 583 | 7 | JVM flags | JMX, NMT |
| Others? | Offheap | 967 | 12 | TBD | NMT |
| Netty pool | Offheap | 896 | 11 | JVM system properties | Java API |
| Sockets | Offheap | vary | vary | OS level | ss, /proc, nodetool |
| Memtable | Offheap | 2048 | 25 | memtable_offheap_space | JMX, nodetool |
| Bloom filters | Offheap | vary | vary | DDL | JMX, nodetoool |
| Index Summary | Offheap | 410 | 5 | cassandra.yaml + DDL | JMX, nodetool |
| Compression metadata | Offheap | vary | vary | DDL | JMX, nodetool |
| Chunk cache | Off-heap | 0 | 0 | Cassandra.yaml | JMX, nodetool |