Cache & Concurrency considerations for a high performance Cassandra

SriSatish Ambati

Performance, Riptano, Cassandra

Azul Systems & OpenJDK

@srisatish

Trail ahead

Elements of Cache Performance

Metrics, Monitors

JVM goes to BigData Land!

Examples

Lucandra, Twissandra

Cassandra Performance with JVM Commentary

Runtime Views

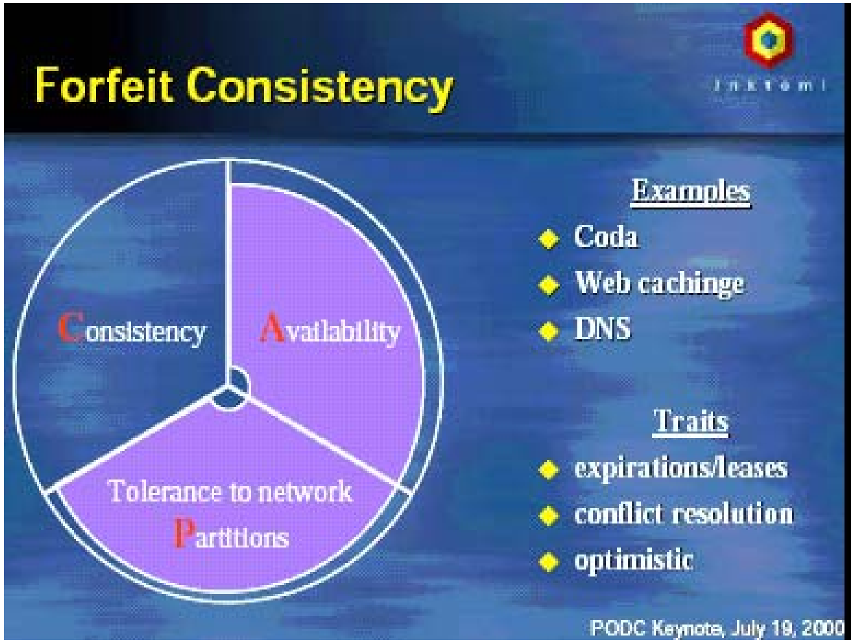
Non Blocking HashMap

Locking: concurrency

Garbage Collection

A feather in the CAP

- Eventual Consistency
  - Levels
  - Doesn’t mean data loss (journaled)
- SEDA
  - Partitioning, Cluster & Failure detection, Storage engine mod
  - Event driven & non-blocking io
  - Pure Java



Forfeit Consistency

Examples

- Coda
- Web cachinge
- DNS

Traits

- expirations/leases
- conflict resolution
- optimistic

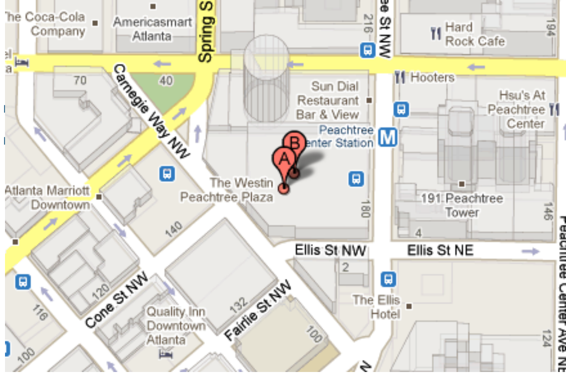
PODC Keynote, July 19, 2000

> Count what is countable, measure what is measurable, and what is not measurable, make measurable
>
> -Galileo

Elements of Cache Performance Metrics

- Operations:
  - Ops/s: Puts/sec, Gets/sec, updates/sec
  - Latencies, percentiles
  - Indexing
- # of nodes – scale, elasticity
- Replication
  - Synchronous, Asynchronous (fast writes)
- Tuneable Consistency
- Durability/Persistence
- Size & Number of Objects, Size of Cache
- # of user clients

Elements of Cache Performance: "Think Locality"



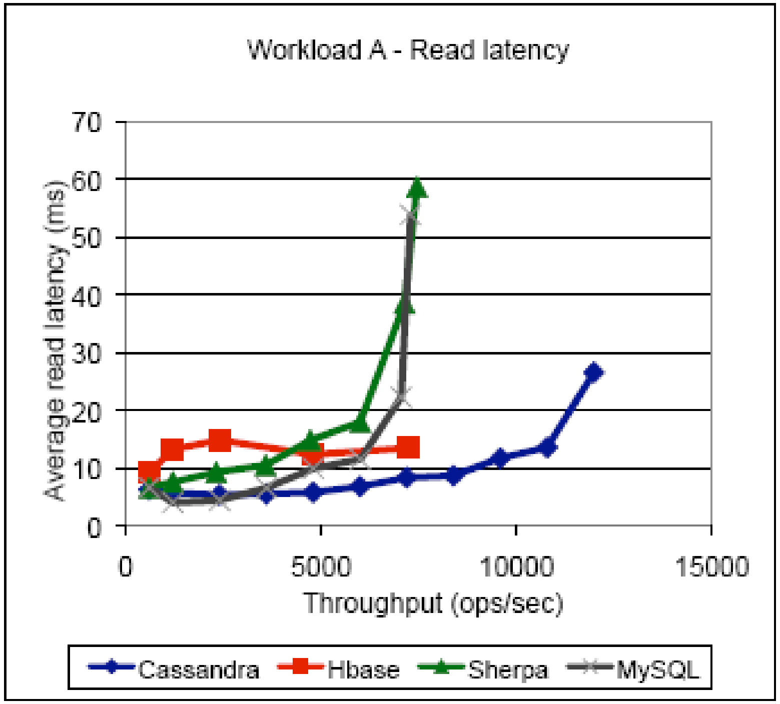
- Hot or Not: The 80/20 rule.
  - – A small set of objects are very popular!
  - – What is the most RT tweet?
- Hit or Miss: Hit Ratio
  - – How effective is your cache?
  - – LRU, LFU, FIFO.. Expiration
- Long-lived objects lead to better locality.
- Spikes happen
  - – Cascading events
  - – Cache Thrash: full table scans

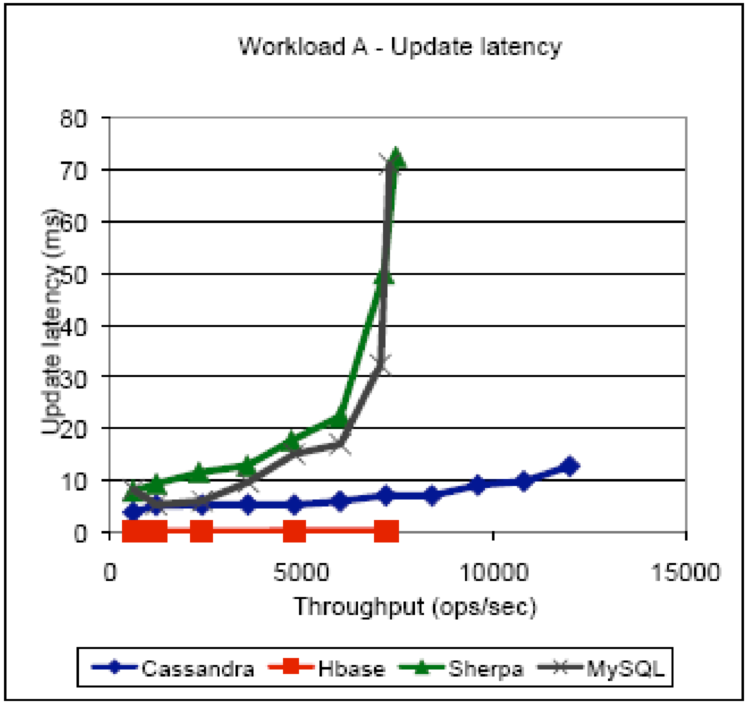
Real World Performance

- Facebook Inbox
  - Writes:0.12ms, Reads:15ms @ 50GB data
- Twitter performance
  - Twissandra (simulation)
- Cassandra for Search & Portals
  - Lucandra, solandra (simulation)
- ycbs/PNUTS benchmarks
  - 5ms read/writes @ 5k ops/s (50/50 Update heavy)
  - 8ms reads/5ms writes @ 5k ops/s (95/5 read heavy)
- Lab environment
  - ~5k writes per sec per node, <5ms latencies
  - ~10k reads per sec per node, <5ms latencies
- Performance has improved in newer versions

yahoo cloud store benchmark

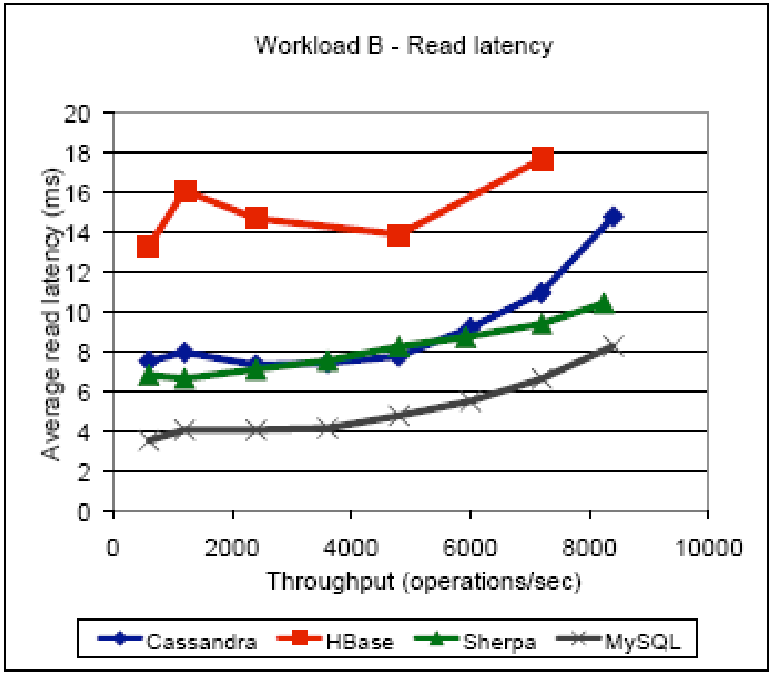
50/50 – Update Heavy

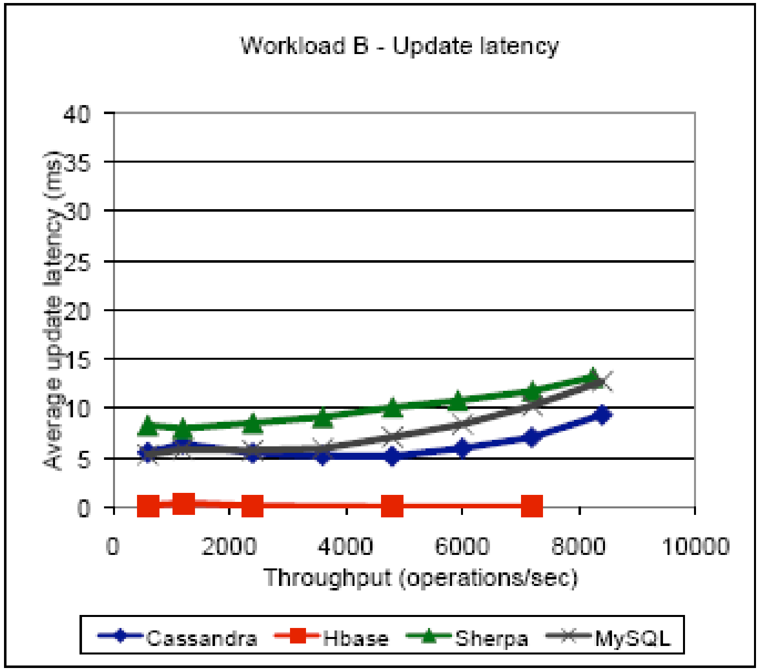




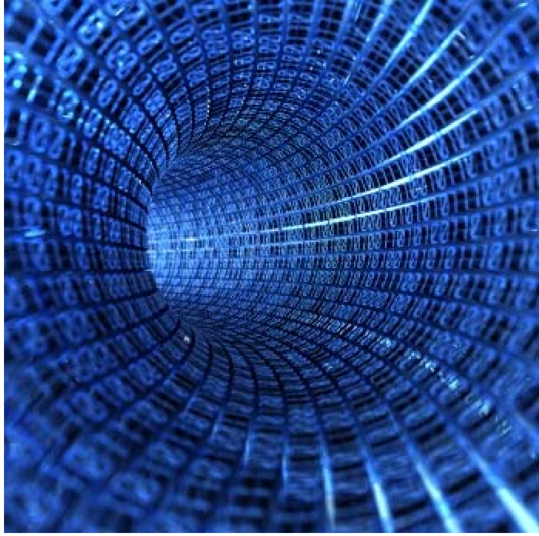
yahoo cloud store benchmark

95/5 – read heavy





Java in BigData Land!



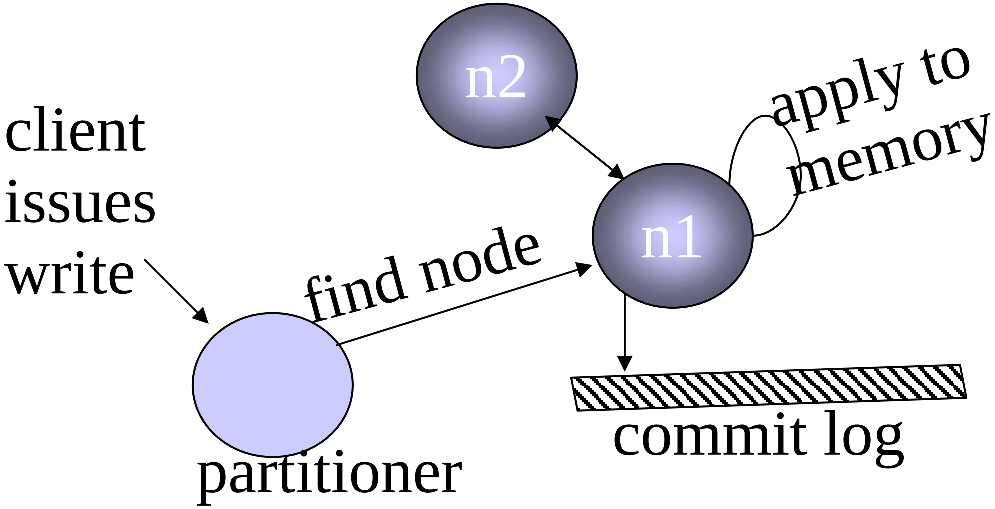
Limits for scale

- Locks : **`synchronized`**
  - Can’t use all my multi-cores!
  - `java.util.collections` also hold locks
  - **Use non-blocking collections!**
- (de)Serialization is expensive
  - Hampers object portability
  - **Use avro, thrift!**
- Object overhead
  - average enterprise collection has 3 elements!
  - **Use byte[ ], primitives where possible!**
- Garbage Collection
  - Can’t throw memory at the problem!
  - **Mitigate, Monitor, Measure foot print**

Tools

- What is the JVM doing:
  - – dtrace, hprof, introscope, jconsole, visualvm, yourkit, azul zvision
- Invasive JVM observation tools
  - – bci, jvmti, jvmdi/pi agents, jmx, logging
- What is the OS doing:
  - – dtrace, oprofile, vtune
- What is the network disk doing:
  - – Ganglia, iostat, lsof, netstat, nagios

furiously ***fast*** writes

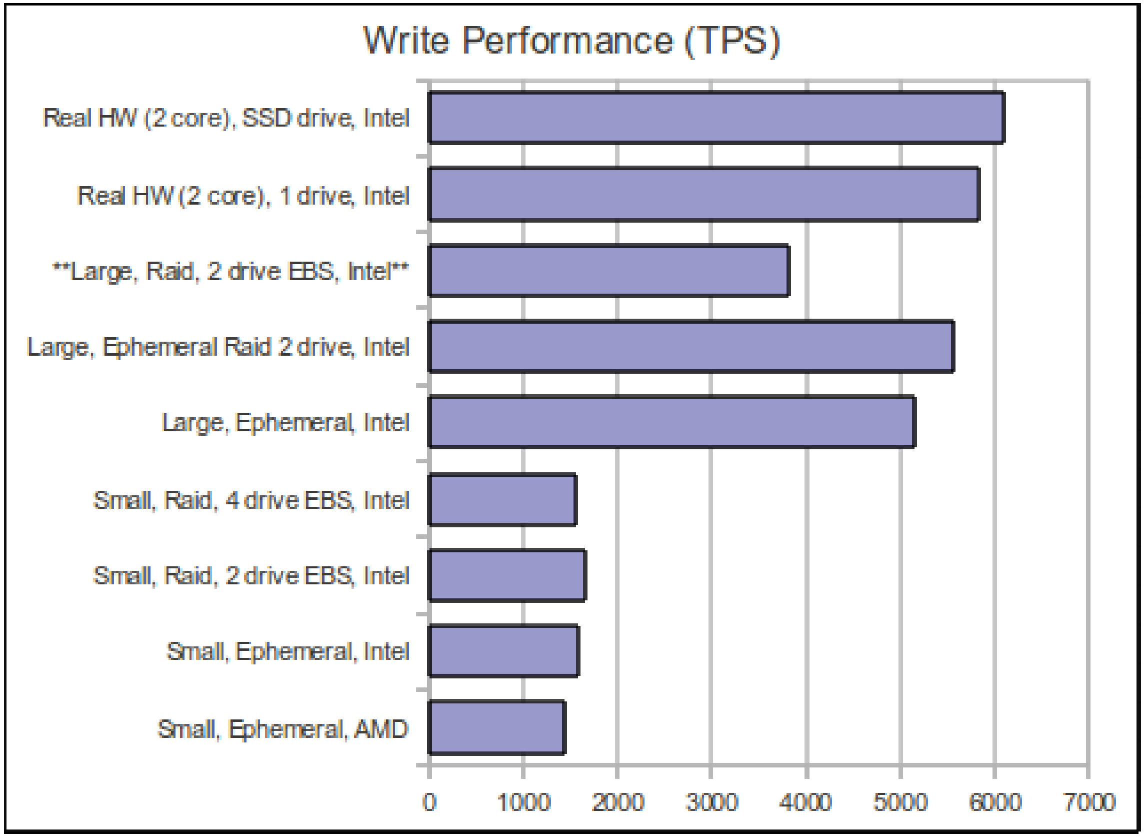


- Append only writes
  - Sequential disk access
- No locks in critical path
- Key based atomicity

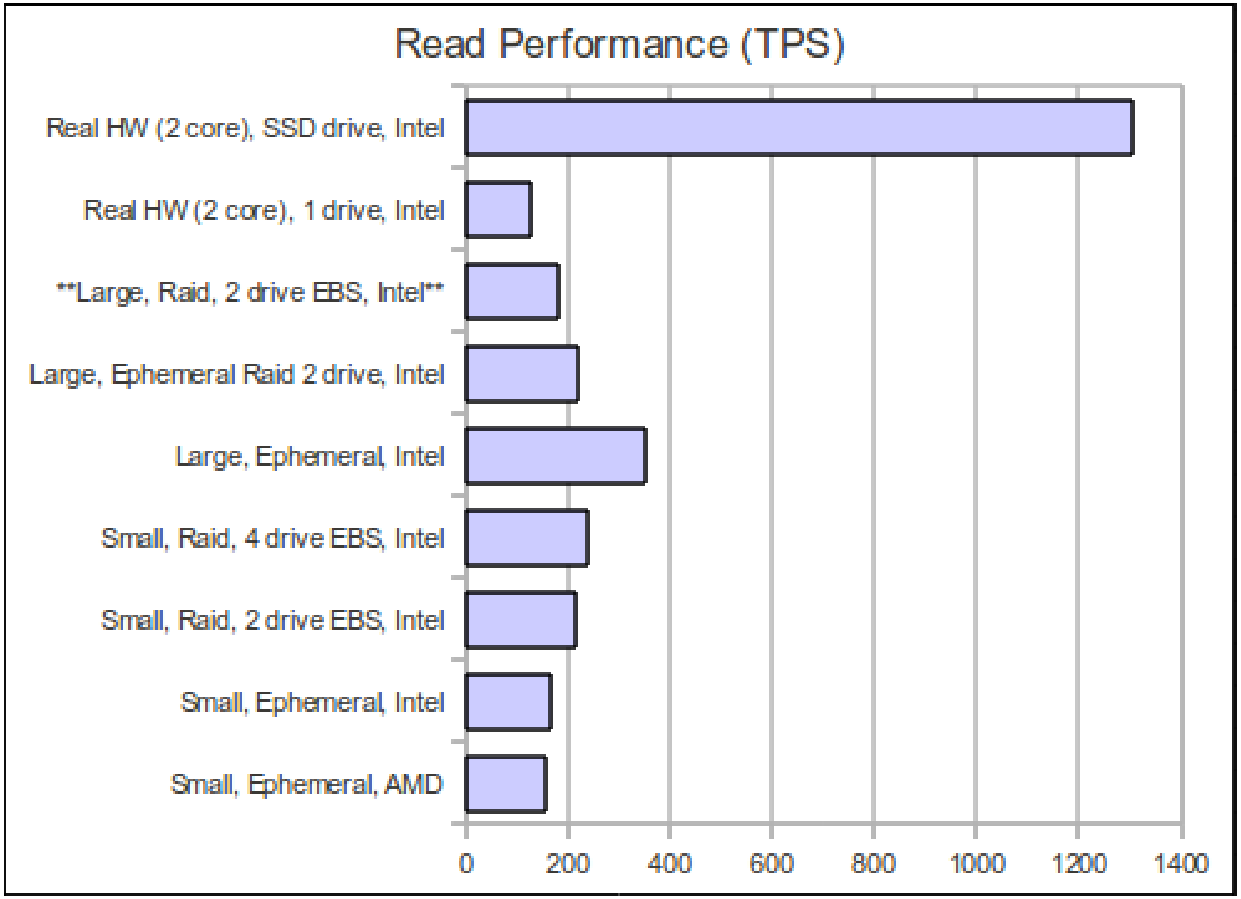
furiously ***`fast`*** writes

- Use separate disks for commitlog
  - – Don’t forget to size them well
  - – Isolation difficult in the cloud..
- Memtable/SSTable sizes
  - – Delicately balanced with GC
- **`memtable_throughput_in_mb`**

Cassandra on EC2 cloud

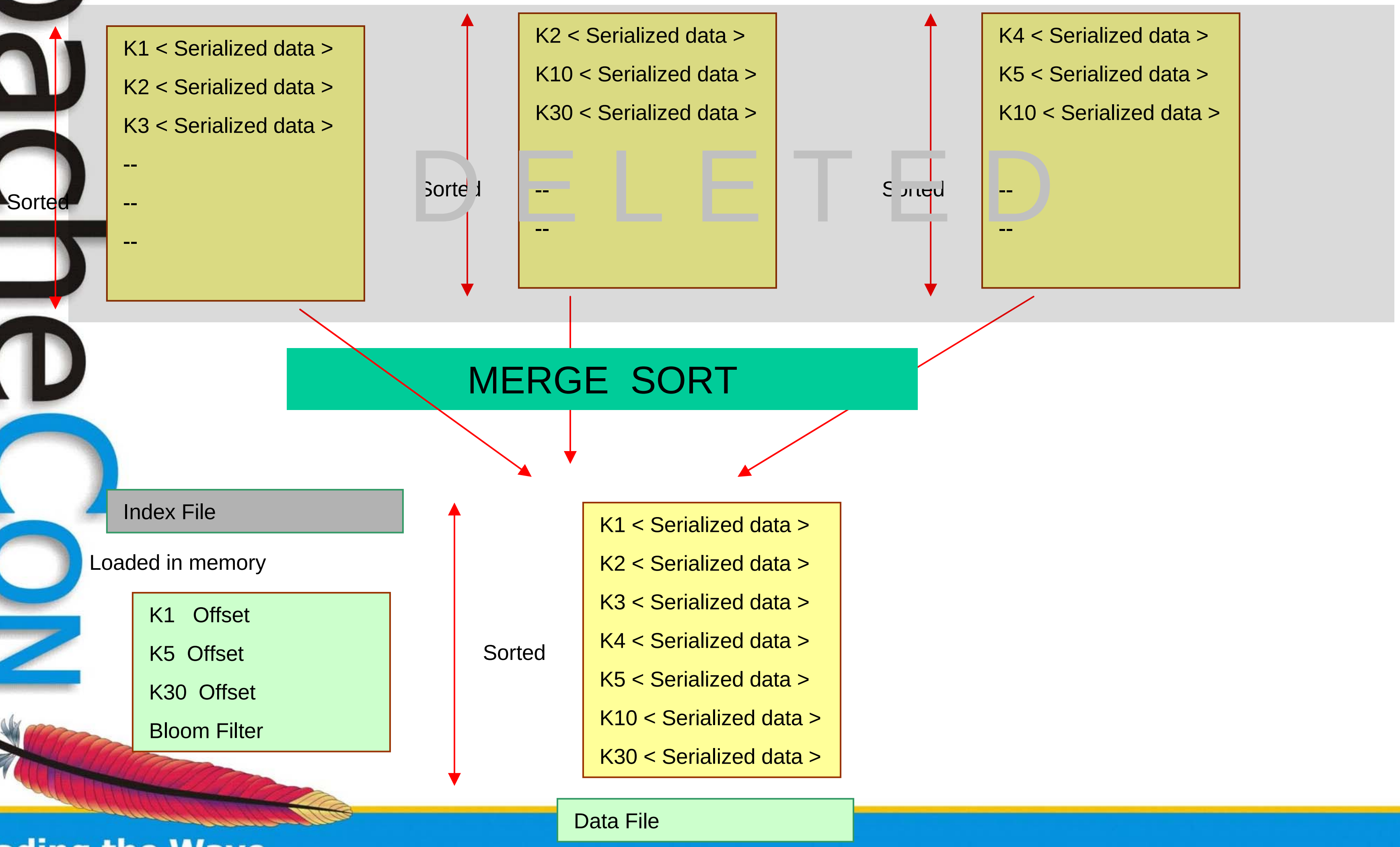


Cassandra on EC2 cloud



THE SIMPSONS

Compactions



K1 < Serialized data >
K2 < Serialized data >
K3 < Serialized data >
--
--
--

Sorted

K2 < Serialized data >
K10 < Serialized data >
K30 < Serialized data >
--
--

Sorted

K4 < Serialized data >
K5 < Serialized data >
K10 < Serialized data >
--
--

Sorted

DELETED

MERGE SORT

Index File

Loaded in memory

K1 Offset
K5 Offset
K30 Offset
Bloom Filter

Sorted

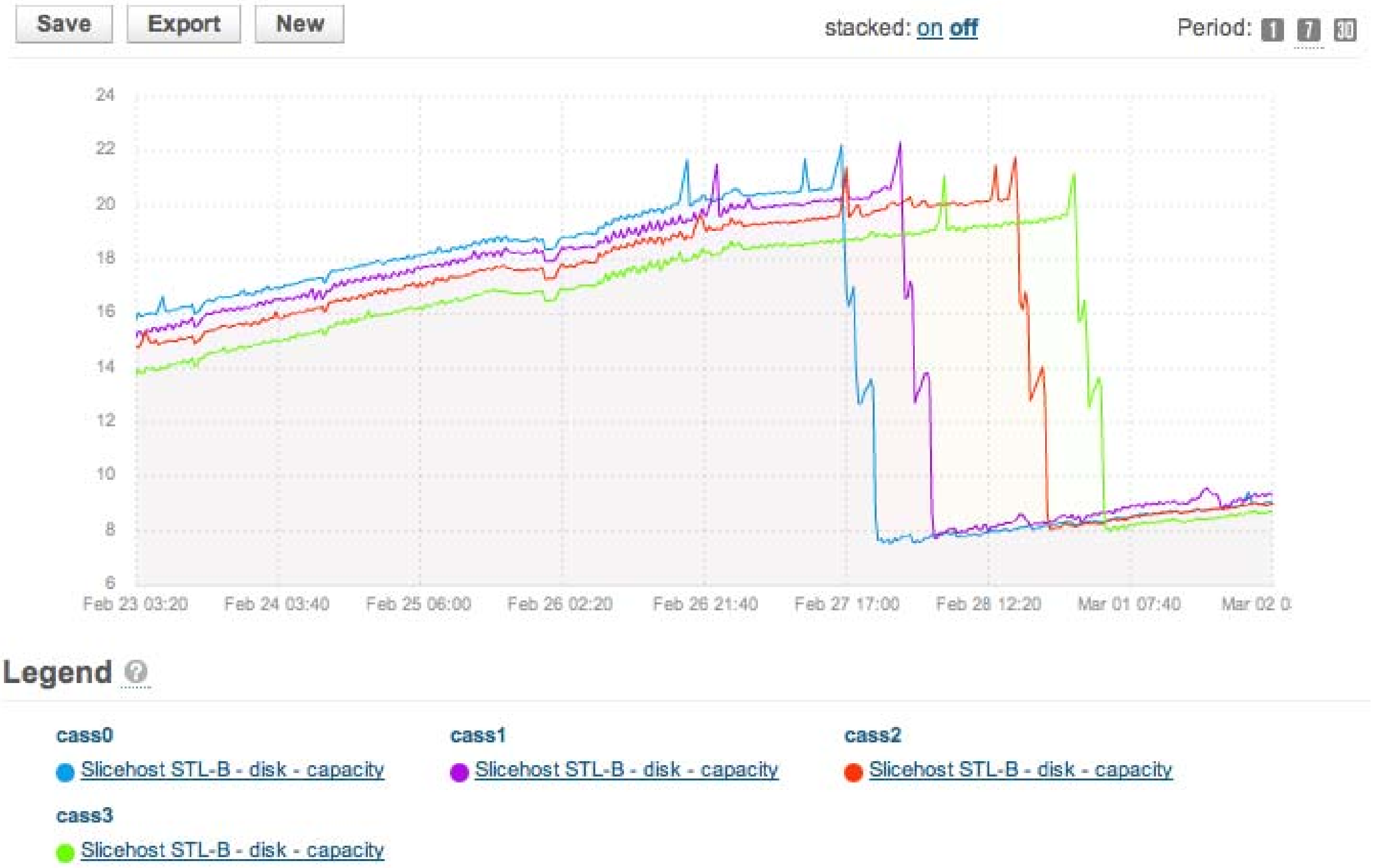
K1 < Serialized data >
K2 < Serialized data >
K3 < Serialized data >
K4 < Serialized data >
K5 < Serialized data >
K10 < Serialized data >
K30 < Serialized data >

Data File

Compactions

- Intense disk io & mem churn
- Triggers GC for tombstones
- Minor/Major Compactions
- Reduce priority for better reads
- Other Parameters -
  - `CompactionManager.minimumCompactionThreshold=xxxx`

Example: compaction in realworld, cloudkick



Legend

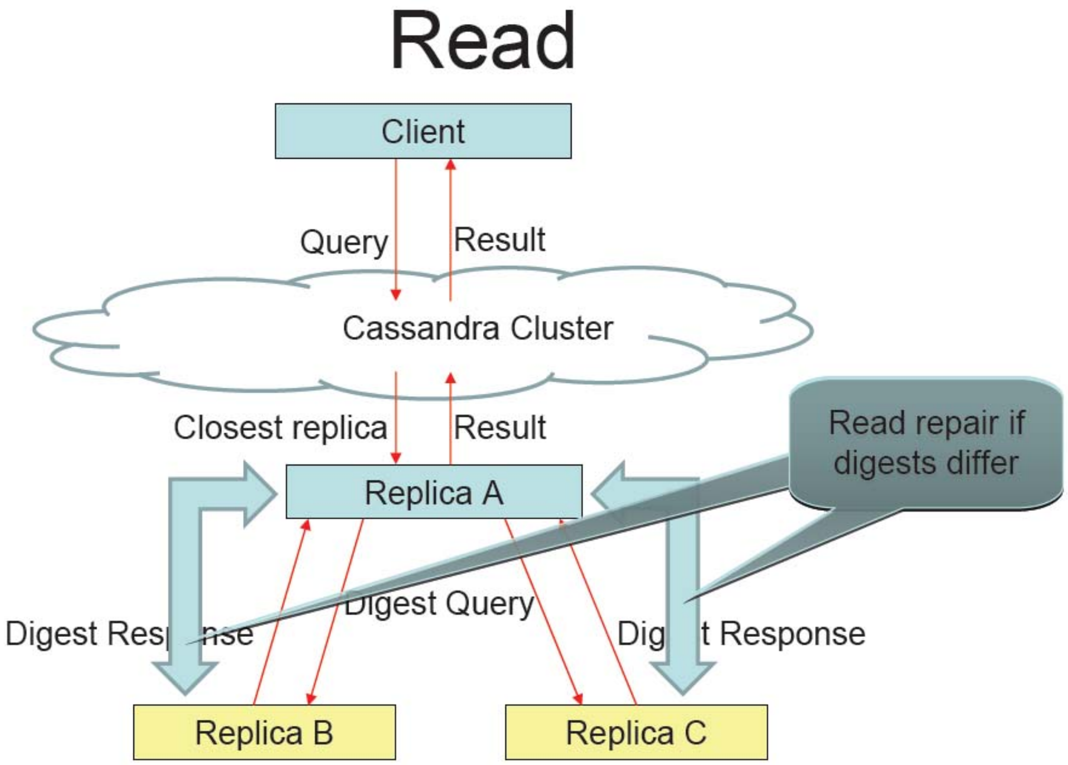
cass0
● Slicehost STL-B - disk - capacity

cass1
● Slicehost STL-B - disk - capacity

cass2
● Slicehost STL-B - disk - capacity

cass3
● Slicehost STL-B - disk - capacity

reads design



Read repair if digests differ

reads performance

- BloomFilter used to identify the right file
- Maintain column indices to look up columns
  - Which can span different SSTables
- Less io than typical b-tree
- Cold read: Two seeks
  - One for Key lookup, another row lookup
- Key Cache
  - Optimized in latest cassandra
- Row Cache
  - Improves read performance
  - GC sensitive for large rows.
- Most (google) applications require single row transactions*

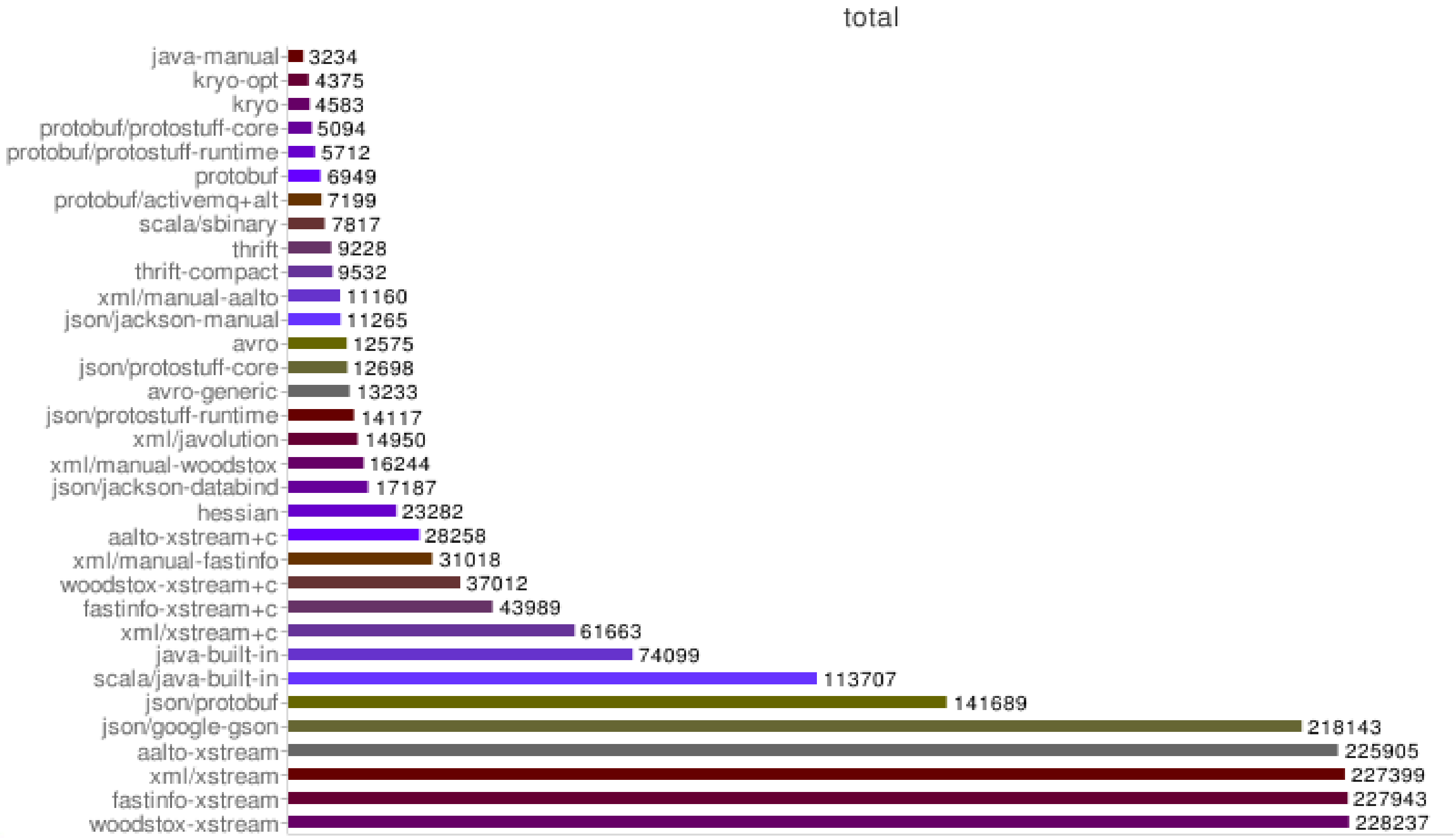
*Sanjay G, BigTable Design, Google.

Client Performance Marshal Arts: Ser/Deserialization

- Clients dominated by Thrift, Avro
  - Hector, Pelops
- Thrift: upgrade to latest: 0.5, 0.4
- No news: java.io.Serializable is S.L..O….W
- Use “transient”
- avro, thrift, proto-buf
- Common Patterns of Doom:
  - Death by a million gets

Serialization + Deserialization uBench

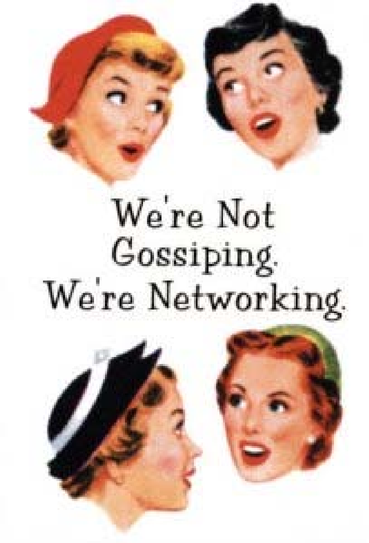
- http://code.google.com/p/thrift-protobuf-compare/wiki/BenchmarkingV2



Adding Nodes

- New nodes
  - Add themselves to busiest node
  - And then Split its Range
- Busy Node starts transmit to new node
- Bootstrap logic initiated from any node, cli, web
- Each node capable of ~40MB/s
  - Multiple replicas to parallelize bootstrap
- UDP for control messages
- TCP for request routing

inter-node comm

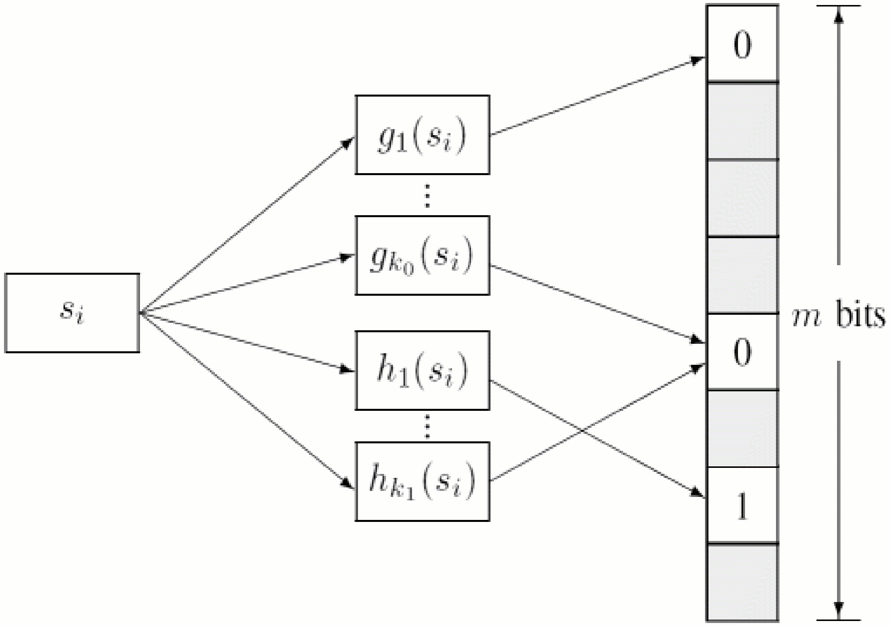


- Gossip Protocol
  - – It’s exponential
  - – (epidemic algorithm)
- Failure Detector
  - – Accrual rate phi
- Anti-Entropy
  - – Bringing replicas to uptodate

Bloom Filter: in full bloom

- “constant” time
- size:compact
- false positives
- Single lookup

  for key in file
- Deletion
- Improve
  - Counting BF
  - Bloomier filters



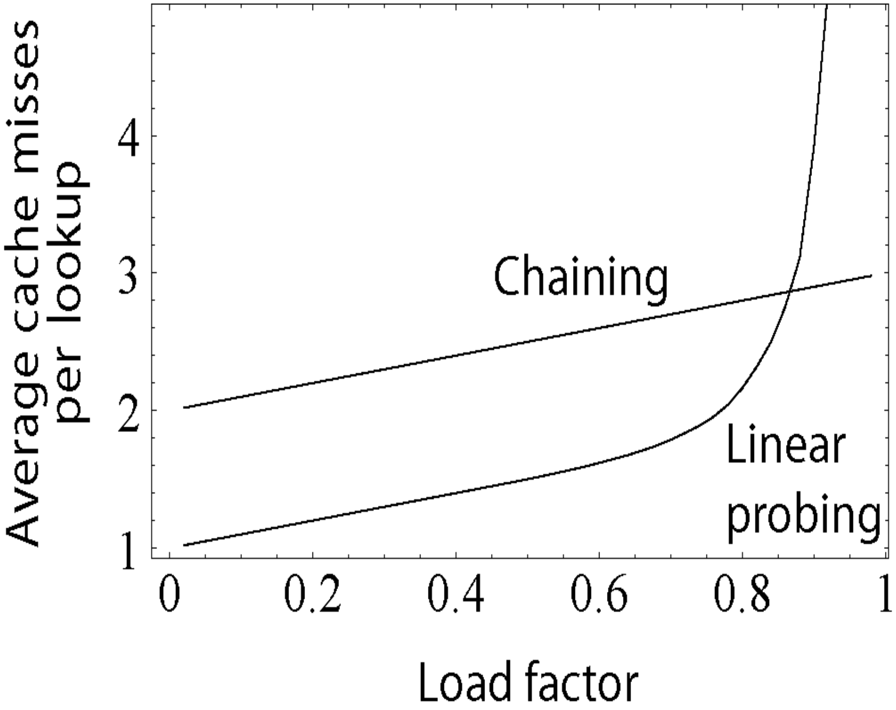
Birthdays, Collisions & Hashing functions

- Birthday Paradox

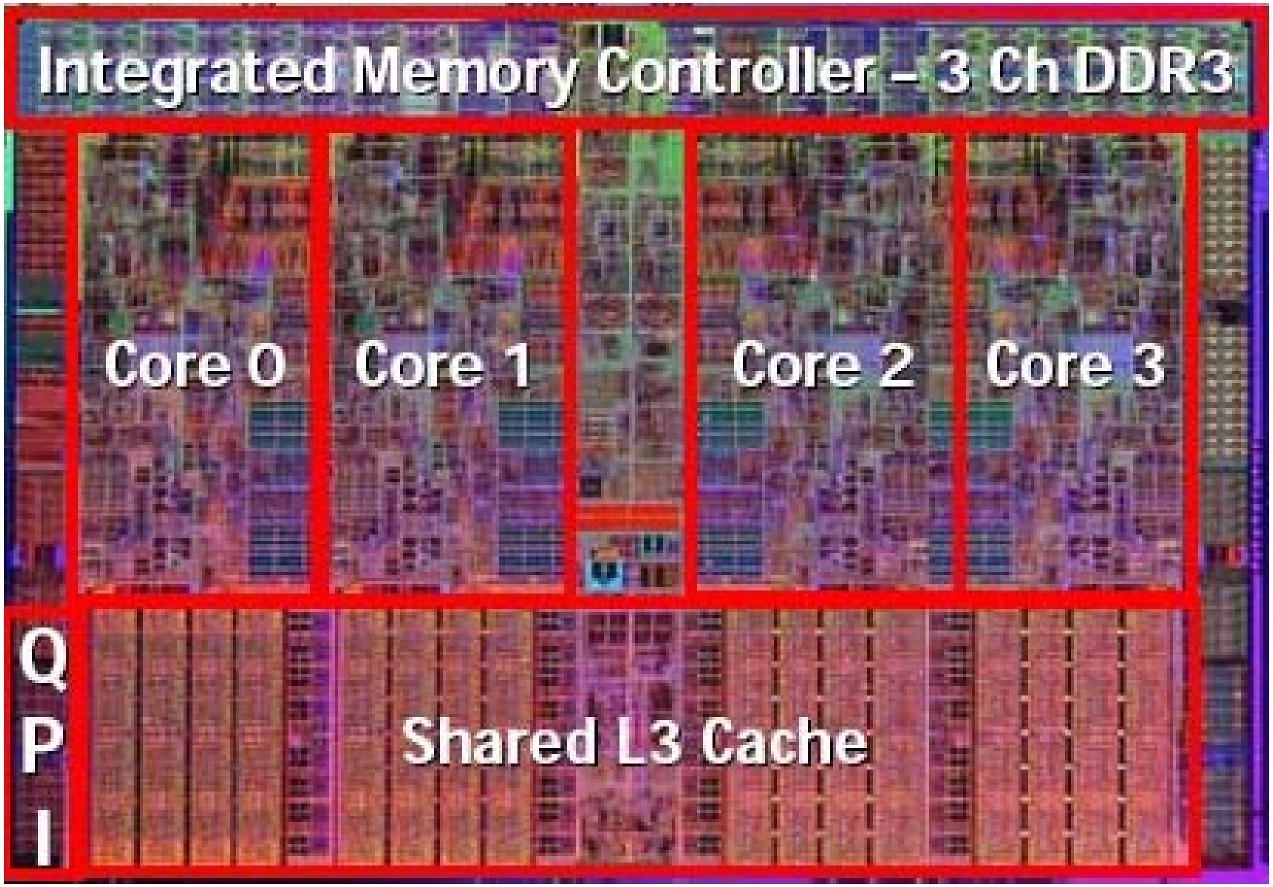
  For the N=21 people in this room

  Probability that at least 2 of them share same birthday is ~0.47
- Collisions are real!
- An unbalanced HashMap behaves like a list O(n) retrieval
- Chaining & Linear probing
- Performance Degrades
- with 80% table density

.

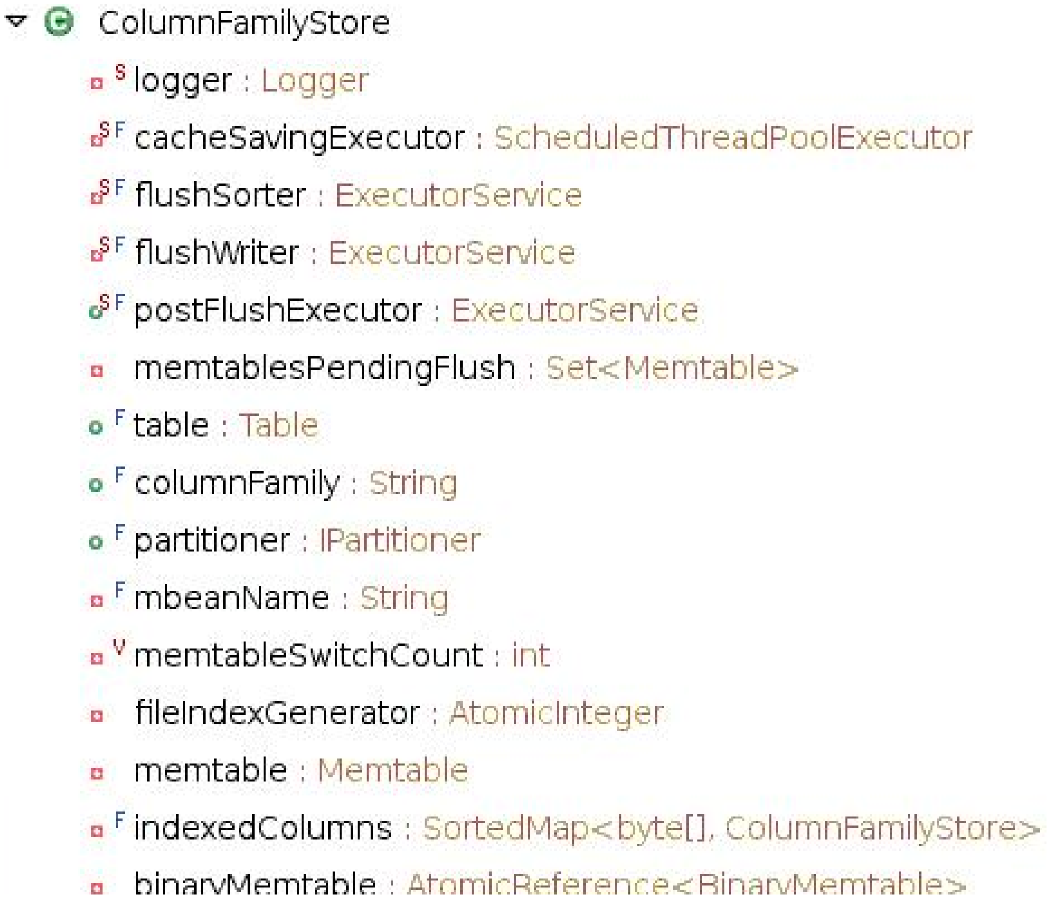


the devil’s in the details



CFS

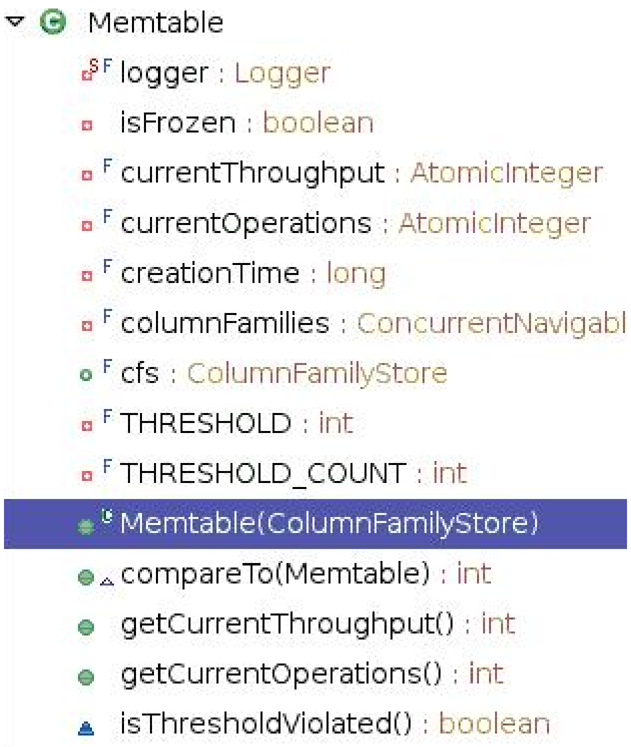
- All in the family!
- denormalize



```
ColumnFamilyStore
    logger : Logger
    cacheSavingExecutor : ScheduledThreadPoolExecutor
    flushSorter : ExecutorService
    flushWriter : ExecutorService
    postFlushExecutor : ExecutorService
    memtablesPendingFlush : Set<Memtable>
    table : Table
    columnFamily : String
    partitioner : IPartitioner
    mbeanName : String
    memtableSwitchCount : int
    fileIndexGenerator : AtomicInteger
    memtable : Memtable
    indexedColumns : SortedMap<byte[], ColumnFamilyStore>
    binaryMemtable : AtomicReference<BinaryMemtable>
```

Memtable

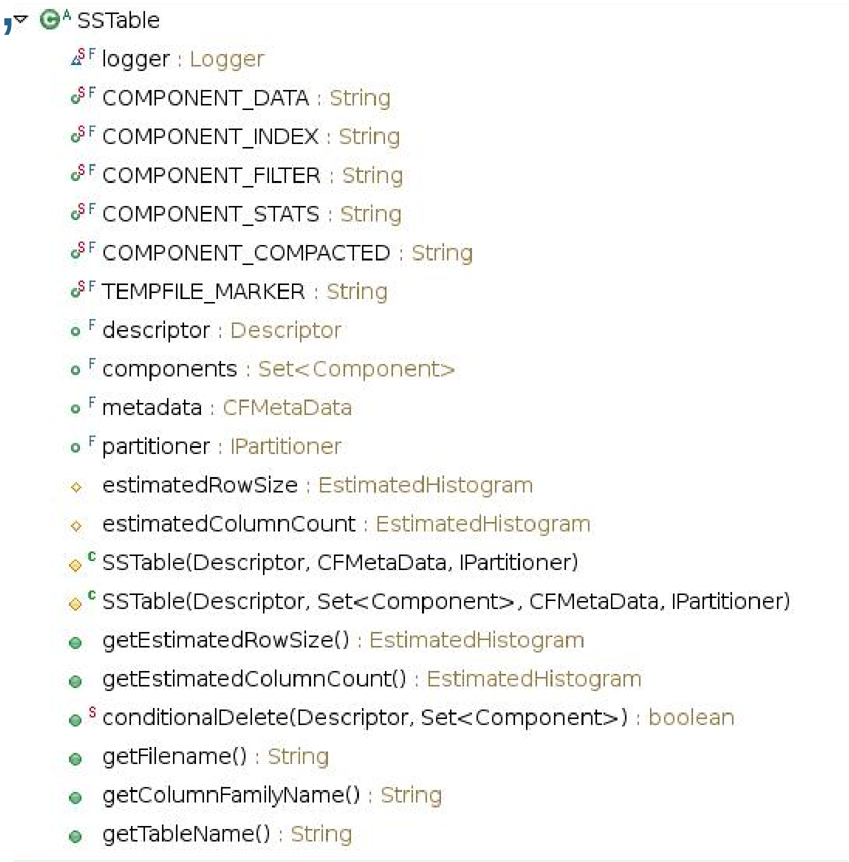
- In-memory
- ColumnFamily specific
- **`throughput`** determines size before flush
- Larger memtables can improve reads



```
Memtable
    logger : Logger
    isFrozen : boolean
    currentThroughput : AtomicInteger
    currentOperations : AtomicInteger
    creationTime : long
    columnFamilies : ConcurrentNavigabl
    cfs : ColumnFamilyStore
    THRESHOLD : int
    THRESHOLD_COUNT : int
    Memtable(ColumnFamilyStore)
    compareTo(Memtable) : int
    getCurrentThroughput() : int
    getCurrentOperations() : int
    isThresholdViolated() : boolean
```

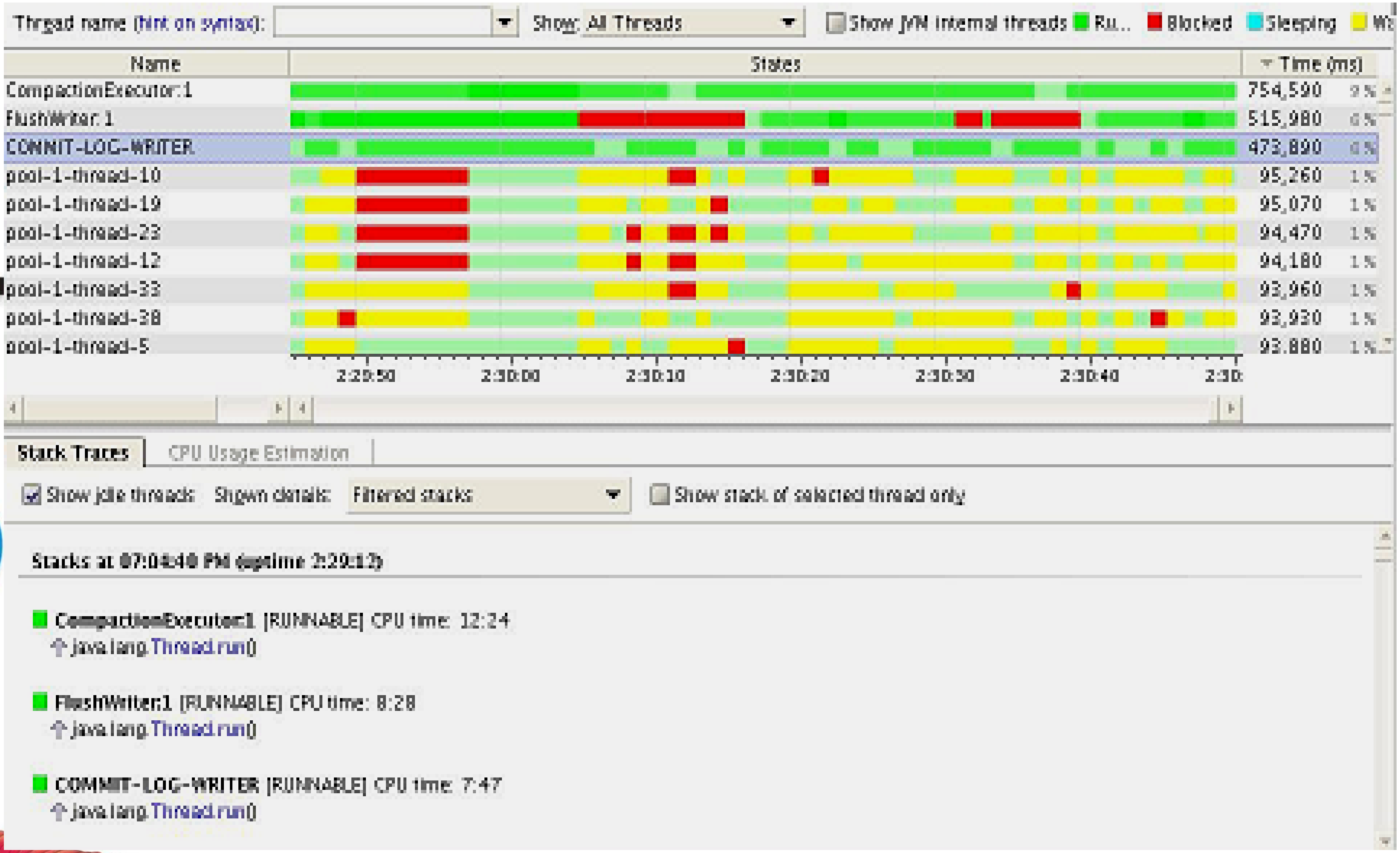
SSTable

- MemTable “flushes” to a SSTable
- Immutable after
- Read: Multiple SSTable lookups possible
- Chief Execs:
  - SSTableWriter
  - SSTableReader



```
SSTable
    logger : Logger
    COMPONENT_DATA : String
    COMPONENT_INDEX : String
    COMPONENT_FILTER : String
    COMPONENT_STATS : String
    COMPONENT_COMPACTED : String
    TEMPFILE_MARKER : String
    descriptor : Descriptor
    components : Set<Component>
    metadata : CFMetaData
    partitioner : IPartitioner
    estimatedRowSize : EstimatedHistogram
    estimatedColumnCount : EstimatedHistogram
    SSTable(Descriptor, CFMetaData, IPartitioner)
    SSTable(Descriptor, Set<Component>, CFMetaData, IPartitioner)
    getEstimatedRowSize() : EstimatedHistogram
    getEstimatedColumnCount() : EstimatedHistogram
    conditionalDelete(Descriptor, Set<Component>) : boolean
    getFilename() : String
    getColumnFamilyName() : String
    getTableName() : String
```

Write: Runtime threads



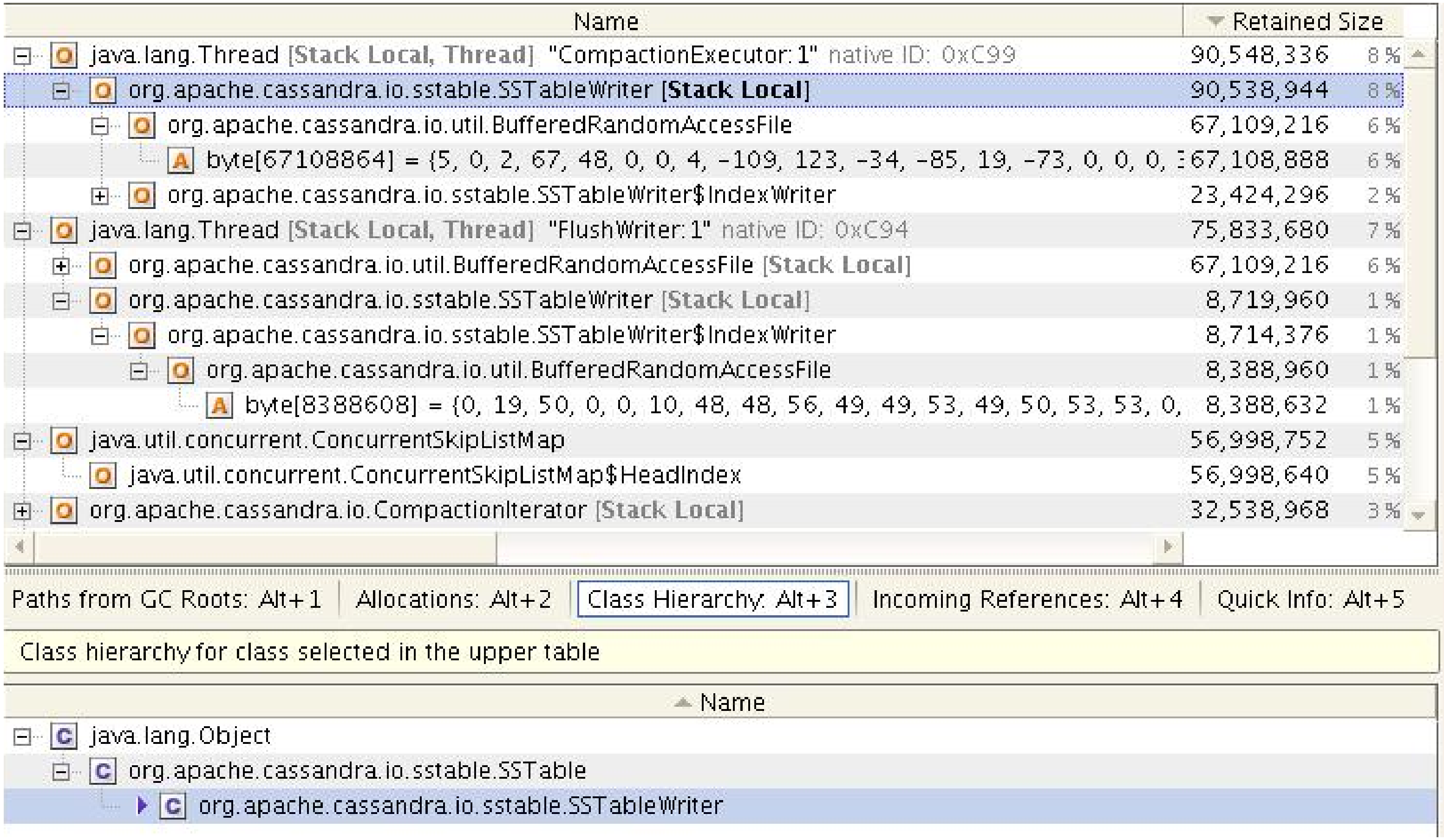
Name	States	Time (ms)	
CompactionExecutor:1		754,590	9 %
FlushWriter:1		515,980	6 %
COMMIT-LOG-WRITER		473,890	6 %
pool-1-thread-10		95,260	1 %
pool-1-thread-19		95,070	1 %
pool-1-thread-23		94,470	1 %
pool-1-thread-12		94,180	1 %
pool-1-thread-33		93,960	1 %
pool-1-thread-38		93,930	1 %
pool-1-thread-5		93,880	1 %

☑ Show idle threads Shown details: Filtered stacks □ Show stack of selected thread only

```
Stacks at 07:04:40 PM (uptime 2:29:12)

CompactionExecutor:1 [RUNNABLE] CPU time: 12:24
  java.lang.Thread.run()

FlushWriter:1 [RUNNABLE] CPU time: 8:28
  java.lang.Thread.run()

COMMIT-LOG-WRITER [RUNNABLE] CPU time: 7:47
  java.lang.Thread.run()
```

Leading the Wave
of Open Source

Writes: runtime mem

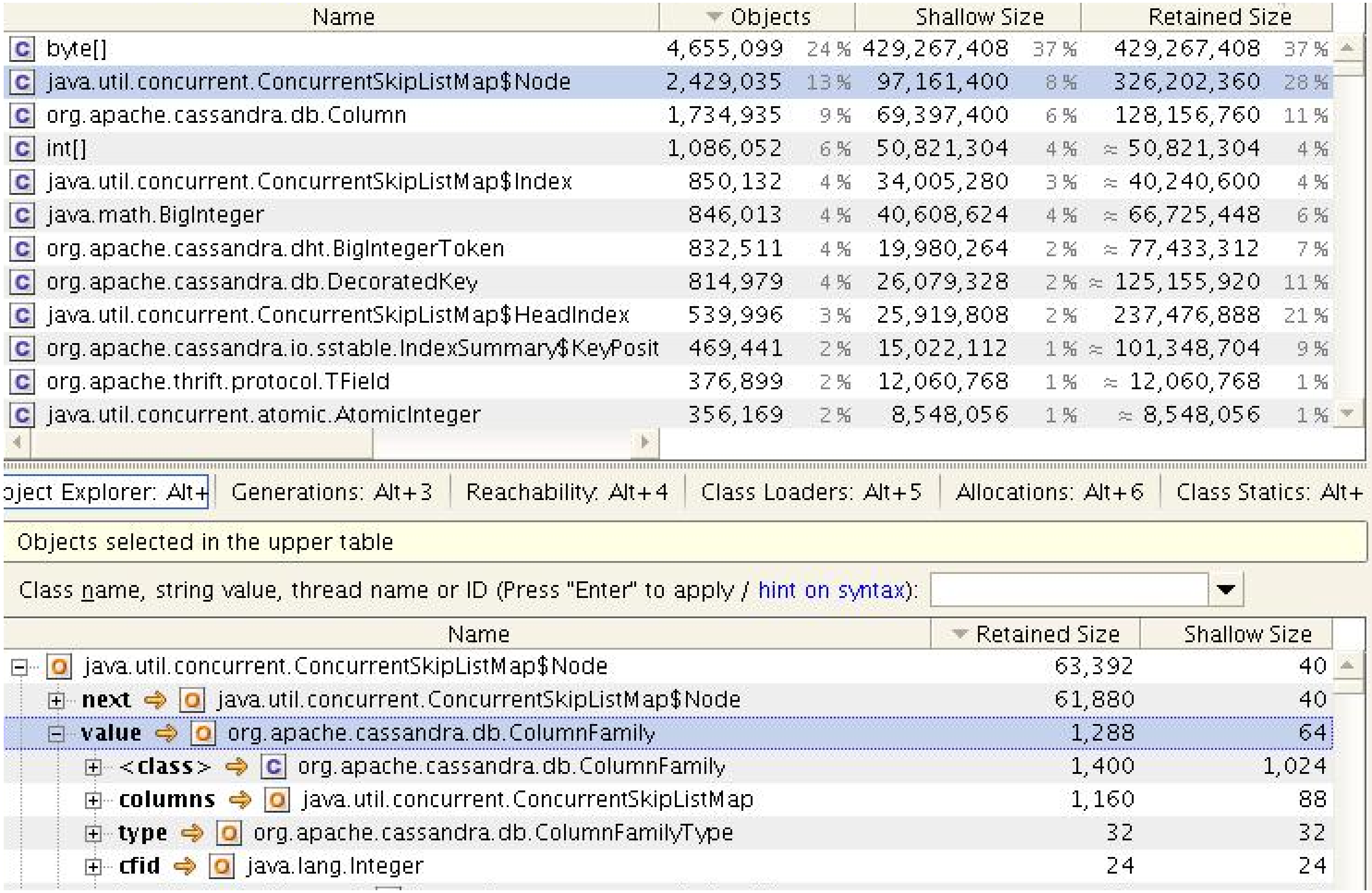


Name	Retained Size	
java.lang.Thread [Stack Local, Thread] "CompactionExecutor:1" native ID: 0xC99	90,548,336	8 %
org.apache.cassandra.io.sstable.SSTableWriter [Stack Local]	90,538,944	8 %
org.apache.cassandra.io.util.BufferedRandomAccessFile	67,109,216	6 %
byte[67108864] = {5, 0, 2, 67, 48, 0, 0, 4, -109, 123, -34, -85, 19, -73, 0, 0, 0,	67,108,888	6 %
org.apache.cassandra.io.sstable.SSTableWriter$IndexWriter	23,424,296	2 %
java.lang.Thread [Stack Local, Thread] "FlushWriter:1" native ID: 0xC94	75,833,680	7 %
org.apache.cassandra.io.util.BufferedRandomAccessFile [Stack Local]	67,109,216	6 %
org.apache.cassandra.io.sstable.SSTableWriter [Stack Local]	8,719,960	1 %
org.apache.cassandra.io.sstable.SSTableWriter$IndexWriter	8,714,376	1 %
org.apache.cassandra.io.util.BufferedRandomAccessFile	8,388,960	1 %
byte[8388608] = {0, 19, 50, 0, 0, 10, 48, 48, 56, 49, 49, 53, 49, 50, 53, 53, 0,	8,388,632	1 %
java.util.concurrent.ConcurrentSkipListMap	56,998,752	5 %
java.util.concurrent.ConcurrentSkipListMap$HeadIndex	56,998,640	5 %
org.apache.cassandra.io.CompactionIterator [Stack Local]	32,538,968	3 %

Class hierarchy for class selected in the upper table

Name
java.lang.Object
org.apache.cassandra.io.sstable.SSTable
org.apache.cassandra.io.sstable.SSTableWriter

Example: Java Overheads



Name	Objects		Shallow Size		Retained Size	
byte[]	4,655,099	24 %	429,267,408	37 %	429,267,408	37 %
java.util.concurrent.ConcurrentSkipListMap$Node	2,429,035	13 %	97,161,400	8 %	326,202,360	28 %
org.apache.cassandra.db.Column	1,734,935	9 %	69,397,400	6 %	128,156,760	11 %
int[]	1,086,052	6 %	50,821,304	4 %	≈ 50,821,304	4 %
java.util.concurrent.ConcurrentSkipListMap$Index	850,132	4 %	34,005,280	3 %	≈ 40,240,600	4 %
java.math.BigInteger	846,013	4 %	40,608,624	4 %	≈ 66,725,448	6 %
org.apache.cassandra.dht.BigIntegerToken	832,511	4 %	19,980,264	2 %	≈ 77,433,312	7 %
org.apache.cassandra.db.DecoratedKey	814,979	4 %	26,079,328	2 %	≈ 125,155,920	11 %
java.util.concurrent.ConcurrentSkipListMap$HeadIndex	539,996	3 %	25,919,808	2 %	237,476,888	21 %
org.apache.cassandra.io.sstable.IndexSummary$KeyPosit	469,441	2 %	15,022,112	1 %	≈ 101,348,704	9 %
org.apache.thrift.protocol.TField	376,899	2 %	12,060,768	1 %	≈ 12,060,768	1 %
java.util.concurrent.atomic.AtomicInteger	356,169	2 %	8,548,056	1 %	≈ 8,548,056	1 %

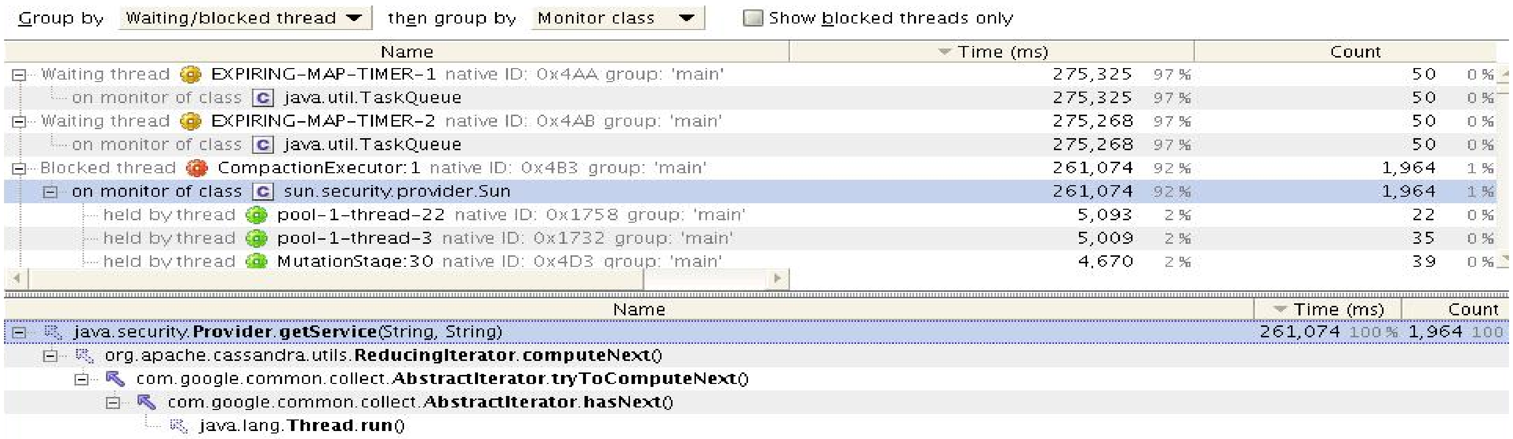
bject Explorer: Alt+ | Generations: Alt+3 | Reachability: Alt+4 | Class Loaders: Alt+5 | Allocations: Alt+6 | Class Statics: Alt+

Objects selected in the upper table

Class name, string value, thread name or ID (Press "Enter" to apply / hint on syntax):

Name	Retained Size	Shallow Size
java.util.concurrent.ConcurrentSkipListMap$Node	63,392	40
next ⇨ java.util.concurrent.ConcurrentSkipListMap$Node	61,880	40
value ⇨ org.apache.cassandra.db.ColumnFamily	1,288	64
<class> ⇨ org.apache.cassandra.db.ColumnFamily	1,400	1,024
columns ⇨ java.util.concurrent.ConcurrentSkipListMap	1,160	88
type ⇨ org.apache.cassandra.db.ColumnFamilyType	32	32
cfid ⇨ java.lang.Integer	24	24

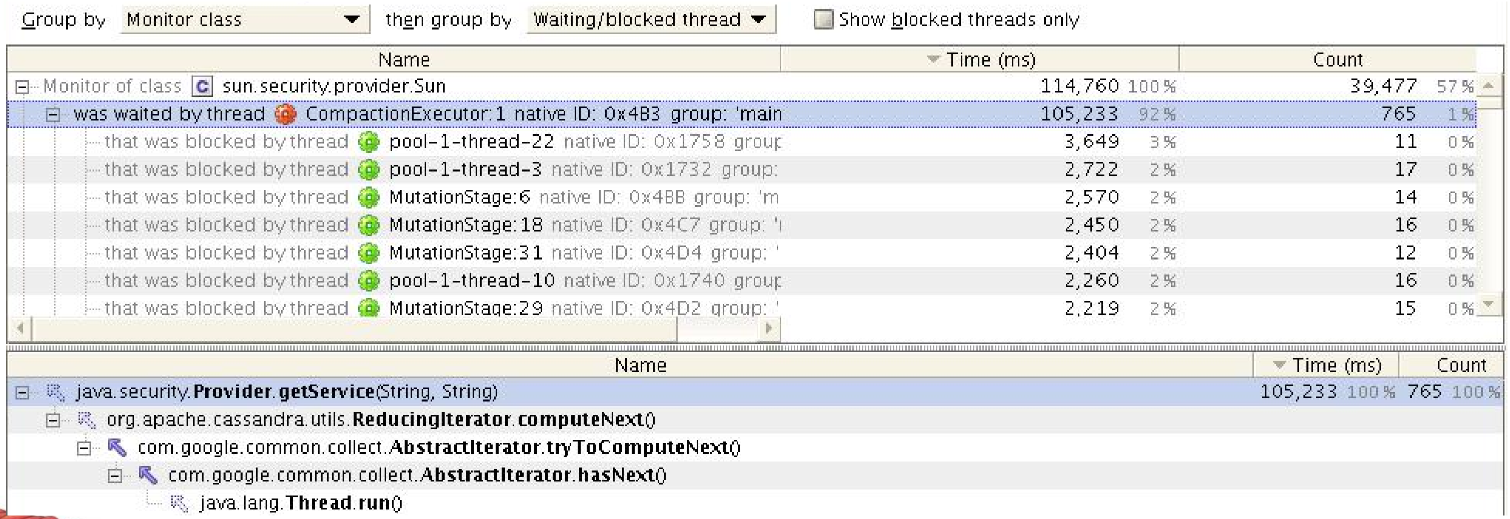
writes: monitors



Group by: Waiting/blocked thread ▾ then group by: Monitor class ▾ □ Show blocked threads only

Name	Time (ms)		Count	
Waiting thread EXPIRING-MAP-TIMER-1 native ID: 0x4AA group: 'main'	275,325	97 %	50	0 %
on monitor of class java.util.TaskQueue	275,325	97 %	50	0 %
Waiting thread EXPIRING-MAP-TIMER-2 native ID: 0x4AB group: 'main'	275,268	97 %	50	0 %
on monitor of class java.util.TaskQueue	275,268	97 %	50	0 %
Blocked thread CompactionExecutor:1 native ID: 0x4B3 group: 'main'	261,074	92 %	1,964	1 %
on monitor of class sun.security.provider.Sun	261,074	92 %	1,964	1 %
held by thread pool-1-thread-22 native ID: 0x1758 group: 'main'	5,093	2 %	22	0 %
held by thread pool-1-thread-3 native ID: 0x1732 group: 'main'	5,009	2 %	35	0 %
held by thread MutationStage:30 native ID: 0x4D3 group: 'main'	4,670	2 %	39	0 %

Name	Time (ms)		Count	
java.security.Provider.getService(String, String)	261,074	100 %	1,964	100
org.apache.cassandra.utils.ReducingIterator.computeNext()				
com.google.common.collect.AbstractIterator.tryToComputeNext()				
com.google.common.collect.AbstractIterator.hasNext()				
java.lang.Thread.run()				



Group by: Monitor class ▾ then group by: Waiting/blocked thread ▾ □ Show blocked threads only

Name	Time (ms)		Count	
Monitor of class sun.security.provider.Sun	114,760	100 %	39,477	57 %
was waited by thread CompactionExecutor:1 native ID: 0x4B3 group: 'main	105,233	92 %	765	1 %
that was blocked by thread pool-1-thread-22 native ID: 0x1758 grour	3,649	3 %	11	0 %
that was blocked by thread pool-1-thread-3 native ID: 0x1732 group:	2,722	2 %	17	0 %
that was blocked by thread MutationStage:6 native ID: 0x4BB group: 'm	2,570	2 %	14	0 %
that was blocked by thread MutationStage:18 native ID: 0x4C7 group: 'I	2,450	2 %	16	0 %
that was blocked by thread MutationStage:31 native ID: 0x4D4 group: '	2,404	2 %	12	0 %
that was blocked by thread pool-1-thread-10 native ID: 0x1740 grour	2,260	2 %	16	0 %
that was blocked by thread MutationStage:29 native ID: 0x4D2 group: '	2,219	2 %	15	0 %

Name	Time (ms)		Count	
java.security.Provider.getService(String, String)	105,233	100 %	765	100 %
org.apache.cassandra.utils.ReducingIterator.computeNext()				
com.google.common.collect.AbstractIterator.tryToComputeNext()				
com.google.common.collect.AbstractIterator.hasNext()				
java.lang.Thread.run()				

UUID

- java.util.UUID is slow
  - static use leads to contention

SecureRandom

- Uses /dev/urandom for seed initialization

  `-Djava.security.egd=file:/dev/urandom`
- PRNG without file is atleast 20%-40% better.
- Use TimeUUIDs where possible – much faster
- JUG – java.uuid.generator

- *http://github.com/cowtowncoder/java-uuid-generator*
- *http://jug.safehaus.org/*
- *http://johannburkard.de/blog/programming/java/Java-UUID-generators-compared.html*

synchronized

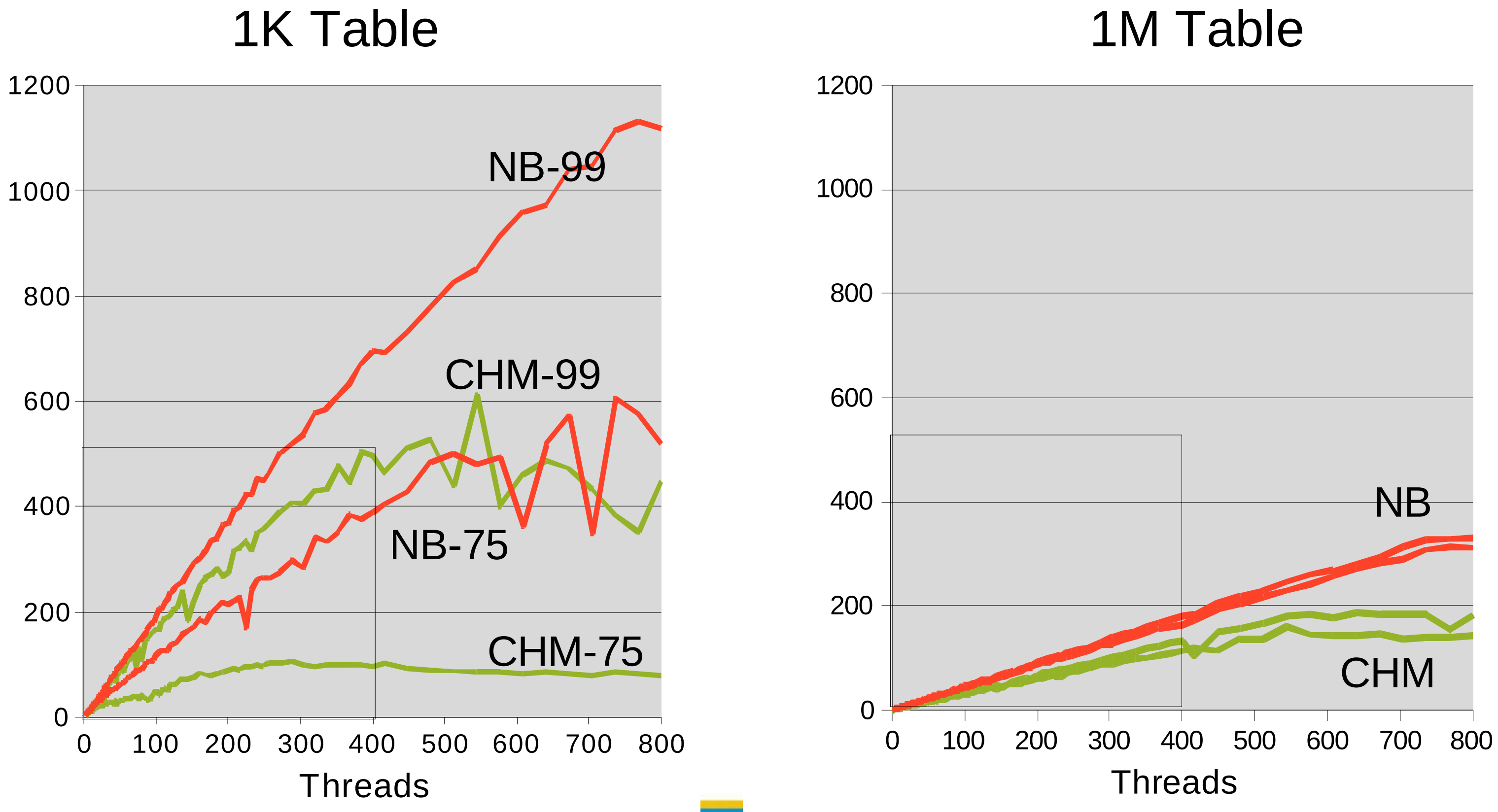
- Coarse grained locks
- io under lock
- Stop signal on a highway
- java.util.concurrent does not mean no locks
- Non Blocking, Lock free, Wait free collections

Scalable Lock-Free Coding Style

- Big Array to hold Data
- Concurrent writes via: CAS & Finite State Machine
  - No **locks**, no **volatile**
  - *Much* faster than locking under heavy load
  - Directly reach main data array in 1 step
- Resize as needed
  - Copy Array to a larger Array on demand
  - Use State Machine to help copy
  - “ Mark” old Array words to avoid missing late updates

Non-Blocking HashMap

Azul Vega2 – 768 cpus



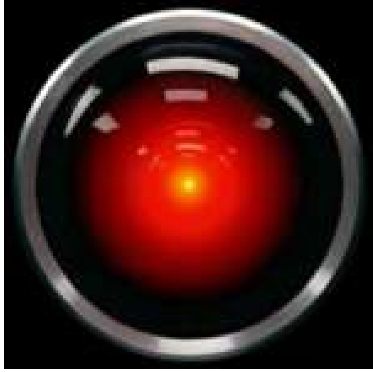
Cassandra uses High Scale Non-Blocking Hashmap

```
public class BinaryMemtable implements IFlushable
{
…
  private final Map<DecoratedKey,byte[]> columnFamilies =
   new NonBlockingHashMap<DecoratedKey, byte[]>();
/* Lock and Condition for notifying new clients about Memtable
    switches */
  private final Lock lock = new ReentrantLock(); Condition condition;
…
}
public class Table
{
…
  private static final Map<String, Table> instances =  new
    NonBlockingHashMap<String, Table>();
…
}
```

GC-sensitive elements within Cassandra

- Compaction triggers System.gc()
  - Tombstones from files
- “GCInspector”
- Memtable Threshold, sizes
- SSTable sizes
- Low overhead collection choices

Garbage Collection



- Pause Times

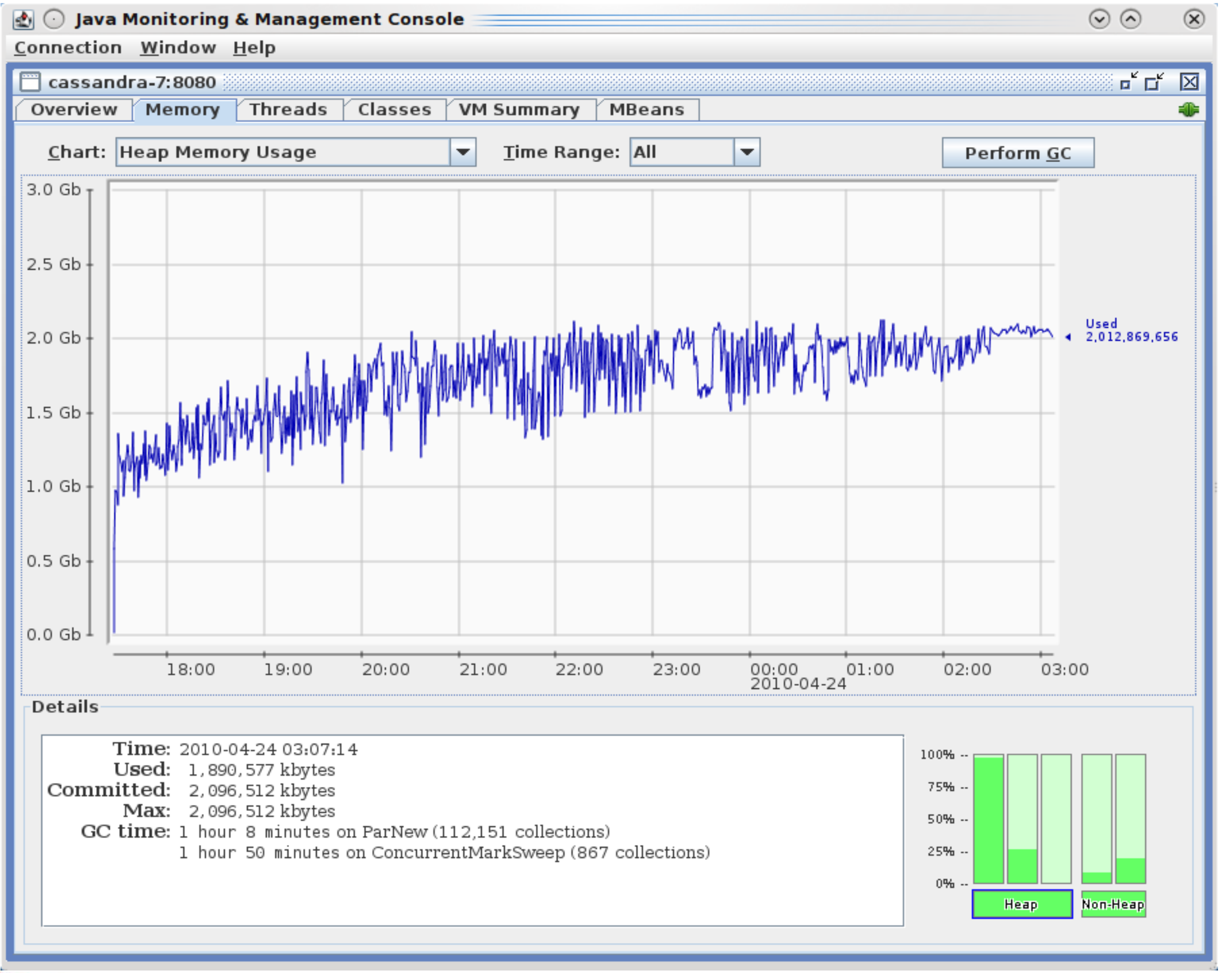
  if stop_the_word_FullGC > ttl_of_node

  => failed requests; failure accrual & node repair.
- Allocation Rate
  - New object creation, insertion rate
- Live Objects (residency)
  - if residency in heap > 50%
  - GC overheads dominate.
- Overhead
  - space, cpu cycles spent GC
- 64-bit not addressing pause times
  - Bigger is not better!

Memory Fragmentation

- Fragmentation
  - Performance degrades over time
  - Inducing “Full GC” makes problem go away
  - Free memory that cannot be used
- Reduce occurrence
  - Use a compacting collector
  - Promote less often
  - Use uniform sized objects
- Solution – *unsolved*
  - Use latest CMS with CR:6631166
  - Azul’s Zing JVM & Pauseless GC

CASSANDRA-1014



Time: 2010-04-24 03:07:14
Used: 1,890,577 kbytes
Committed: 2,096,512 kbytes
Max: 2,096,512 kbytes
GC time: 1 hour 8 minutes on ParNew (112,151 collections)
1 hour 50 minutes on ConcurrentMarkSweep (867 collections)

Best Practices: Garbage Collection

- GC Logs are cheap even in production

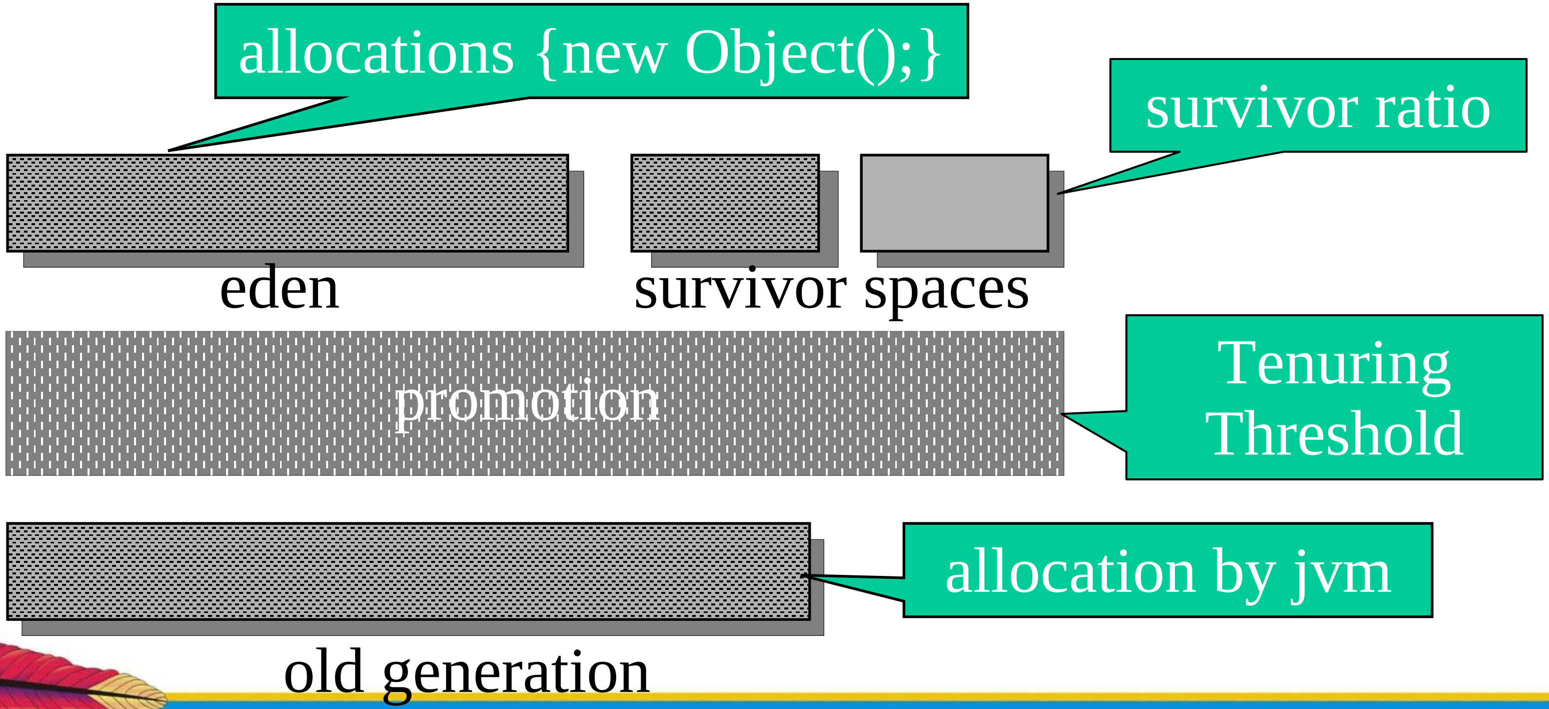
```
-Xloggc:/var/log/cassandra/gc.log
-XX:+PrintGCDetails
-XX:+PrintGCTimeStamps -XX:+PrintTenuringDistribution
-XX:+PrintHeapAtGC
```

- Slightly expensive ones:

```
-XX:PrintFLSStatistics=2 -XX:CMSStatistics=1
-XX:CMSInitiationStatistics
```

Sizing: Young Generation

- Should we set –Xms == -Xmx ?
- Use –Xmn (fixed eden)



Tuning CMS

- Don’t promote too often!
  - Frequent promotion causes fragmentation
- Size the generations
  - Min GC times are a function of Live Set
  - Old Gen should host steady state comfortably
- Parallelize on multicores:
  - -XX:ParallelCMSThreads=4
  - -XX:ParallelGCThreads=4
- Avoid CMS Initiating heuristic
  - -XX:+UseCMSInitiationOccupanyOnly
- Use Concurrent for System.gc()
  - -XX:+ExplicitGCInvokesConcurrent

Summary

Design & Implementation of Cassandra takes advantage of strengths while avoiding common JVM issues.

- **Locks:**
  - **Avoids locks in critical path**
  - **Uses non-blocking collections, TimeUUIDs!**
  - **Still Can’t use all my multi-cores..?**
    >> Other bottlenecks to find!
- **De/Serialization:**
  - **Uses avro, thrift!**
- **Object overhead**
  - **Uses mostly byte[ ], primitives where possible!**
- **Garbage Collection**
  - **Mitigate: Monitor, Measure foot print.**
  - **Work in progress by all jvm vendors!**

Cassandra starts from a great footing from a JVM standpoint and will reap the benefits of the platform!

Q&A

References

- ***Verner Wogels, Eventually Consistent http://www.allthingsdistributed.com/2008/12/eventually_consistent.html***
- ***Bloom, Burton H. (1970), "Space/time trade-offs in hash coding with allowable errors"***
- ***Avinash Lakshman, http://static.last.fm/johan/nosql-20090611/cassandra_nosql.pdf***
- ***Eric Brewer, CAP http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf***
- ***Tony Printzeis, Charlie Hunt, Javaone Talk http://www.scribd.com/doc/36090475/GC-Tuning-in-the-Java***
- ***http://github.com/digitalreasoning/PyStratus/wiki/Documentation***
- ***http://www.cs.cornell.edu/home/rvr/papers/flowgossip.pdf***
- Cassandra on Cloud, http://www.coreyhulen.org/?p=326

Count what is countable, measure what is measurable, and what is not measurable, make measurable

-Galileo