Cassandra

Recommended Settings

Garbage Collection

- Garbage collection is the process by which Java removes data that is no longer needed from memory.
- You definitely want to minimize is a garbage collection pause, also known as a stop-the-world event.
- A pause occurs when a region of memory is full and the JVM needs to make space to continue.
- During a pause all operations are suspended.
- Because a pause affects networking, the node can appear as down to other nodes in the cluster.
- Additionally, any Select and Insert statements will wait, which increases read and write latencies.

GC Pause Detection

The two most common log messages that indicate excessive pausing is occurring are:

```
INFO [ScheduledTasks:1] 2013-03-07 18:44:46,795
GCInspector.java (line 122) GC for
ConcurrentMarkSweep: 1835 ms for 3 collections,
2606015656 used; max is 10611589120
INFO [ScheduledTasks:1] 2013-03-07 19:45:08,029
GCInspector.java (line 122) GC for ParNew: 9866 ms
for 8 collections, 2910124308 used; max is
6358564864
```

Causes of garbage collection pause

- If the problem is recent, check for any recent applications changes.
- Excessive tombstone activity: often caused by heavy delete workloads.
- Large row updates or large batch updates: reduce the size of the individual write below 1 Mb (at the most).
- Extremely wide rows: manifests as problems in repairs, selects, caching, and elsewhere
- Swap is enabled (Disable swap at OS level by `$ sudo swapoff -a`)

G1 – Garbage Collector

- Heap sizes from 14 GB to 64 GB.G1 performs better than CMS for larger heaps because it scans the regions of the heap containing the most garbage objects first, and compacts the heap on-the-go, while CMS stops the application when performing garbage collection.
- The workload is variable, that is, the cluster is performing the different processes all the time.
- For future proofing, as CMS will be deprecated in Java 9
- `JVM_OPTS="$JVM_OPTS -XX:+UseG1GC”`

Size of Java Heap

- Heap size is usually between ¼ and ½ of system memory.
- Do not devote all memory to heap because it is also used for offheap cache and file system cache.
- Always enable GC logging when adjusting GC.
- Adjust settings gradually and test each incremental change.
- Enable parallel processing for GC, particularly when using DSE Search.
- Cassandra's GCInspector class logs information about any garbage collection that takes longer than 200 ms. Garbage collections that occur frequently and take a moderate length of time (seconds) to complete, indicate excessive garbage collection pressure on the JVM. In addition to adjusting the garbage collection options, other remedies include adding nodes, and lowering cache sizes.
- For a node using G1, the Cassandra community recommends a MAX_HEAP_SIZE as large as possible, up to 64 GB.
- Default : `max(min(1/2 ram, 1024MB), min(1/4 ram, 8GB)`

Turn On GC Logging

```
JVM_OPTS="$JVM_OPTS -XX:+PrintGCDetails"
JVM_OPTS="$JVM_OPTS -XX:+PrintGCDateStamps"
JVM_OPTS="$JVM_OPTS -XX:+PrintHeapAtGC"
JVM_OPTS="$JVM_OPTS -XX:+PrintTenuringDistribution"
JVM_OPTS="$JVM_OPTS -XX:+PrintGCApplicationStoppedTime"
JVM_OPTS="$JVM_OPTS -XX:+PrintPromotionFailure"
JVM_OPTS="$JVM_OPTS -XX:PrintFLSStatistics=1”
JVM_OPTS="$JVM_OPTS -Xloggc:/var/log/cassandra/gc-`date +%s`.log"
```

https://www.slideshare.net/aszegedi/everything-i-ever-learned-about-jvm-performance-tuning-twitter/27-Thrift_can_be_heavy_Thrift

IO Stat Check

- Watch the I/O utilization using iostat -x -t 10, which shows the averages for 10 second intervals and prints timestamps:
  - %iowait over 1 indicates that the node is starting to get I/O bound.
  - The acceptable bounds for await (Average Wait in Milliseconds) are:
    - Most SSDs: below 10 ms.
    - Most 7200 RPM spinning disks: below 200 ms.

Adjust Compaction Throughput

- Temporarily adjust the value using nodetool setcompactionthroughput
- Once you have found a good throughput for your system, set it permanently in cassandra.yaml (`compaction_throughput_mb_per_sec`).
- If your I/O is not able to keep up with the necessary compaction throughput, you probably need to get faster disks or add more nodes

Compaction Strategy

- **SizeTieredCompactionStrategy (STCS)**
  - Recommended for write-intensive workloads.
- **LeveledCompactionStrategy (LCS)**
  - Recommended for read-intensive workloads
- **TimeWindowCompactionStrategy (TWCS)**
  - **Suitable for time seriese data type**
- **DateTieredCompactionStrategy (DTCS)**
  - Deprecated in Cassandra 3.0.8/3.8

SizeTieredCompactionStrategy

- The SizeTieredCompactionStrategy (STCS) initiates compaction when Cassandra has accumulated a set number (default: 4) of similar-sized SSTables. STCS merges these SSTables into one larger SSTable. As these larger SSTables accumulate, STCS merges these into even larger SSTables. At any given time, several SSTables of varying sizes are present.
- While STCS works well to compact a write-intensive workload, it makes reads slower because the merge-by-size process does not group data by rows.

SizeTieredCompactionStrategy

- **Pros:** Compacts write-intensive workload very well.
- **Cons:** Can hold onto stale data too long. Amount of memory needed increases over time.

LeveledCompactionStrategy (LCS)

- LCS arranges data in SSTables on multiple levels
- Newly appeared SSTables (not from compaction) almost always show up in L0
- Only L0 is allowed to have SSTables of any size, L1+ levels can only have 160MB SSTables (with minor exceptions)
- Only L0 can allow SSTables to overlap, for all other levels, it's guaranteed to not overlap within the level
- When LCS is able to catch up (i.e. no SSTable in L0), a read in the worst case just need to read from N SSTables (N = the highest level that has data), i.e. at most one SSTable from each L(N>=1) level; ideally 90% of the read will hit the highest level and be satisfied by one SSTable
- Each L(N) level (N>=1) is supposed to accommodate 10^N SSTables. So going one level higher, you will be able to accommodate 10x more SSTables
- When a L0 SSTable is picked as a compaction candidate, it will find all other overlapping L0 SSTables (maximum 32), as well as all of the overlapping L1 SSTables (most likely all 10 L1s), to perform a compaction, and the result will be written into multiple SSTables in L1
- When the total size of all SSTables in L1 is greater than 10 x 160MB, the first L1 SSTable since the last compacted key could be picked as the next compaction candidate, and it will be compacted together with all overlapping L2 SSTables, and the result will be written into multiple SSTables in L2
- For higher levels, it will follow the same fashion, adding candidates from highest level to the lowes

LeveledCompactionStrategy (LCS)

- Pros: Hit ratio is high. Key can be found with max L (number of levels) SS table search.
- Cons: causes write application

Where it is suitable?

- Use cases needing very consistent read performance with much higher read to write ratio

TimeWindowCompactionStrategy (TWCS)

- This strategy is an alternative for time series data.
- TWCS compacts SSTables using a series of *time windows*.
- While with a time window, TWCS compacts all SSTables flushed from memory into larger SSTables using STCS.
- At the end of the time window, all of these SSTables are compacted into a single SSTable.
- Then the next time window starts and the process repeats. The duration of the time window is the only setting required.

Alter Compact Strategy

```
ALTER TABLE users WITH compaction = {'class' :
'SizeTieredCompactionStrategy', 'min_threshold'
: 6 }

ALTER TABLE users WITH compaction = { 'class' :
'LeveledCompactionStrategy' }
```

Compaction Metrics

Monitoring compaction performance is an important aspect of knowing when to add capacity to your cluster. The following attributes are exposed through CompactionManagerMBean:

Attribute	Description
BytesCompacted	Total number of bytes compacted since server [re]start
CompletedTasks	Number of completed compactions since server [re]start
PendingTasks	Estimated number of compactions remaining to perform
TotalCompactionsCompleted	Total number of compactions since server [re]start

Thread pool

```
$ nodetool tpstats
```

```
[training@localhost apache-cassandra-3.10]$ bin/nodetool tpstats
Pool Name                         Active   Pending      Completed   Blocked  All time blocked
ReadStage                              0         0          85363         0                 0
MiscStage                              0         0              0         0                 0
CompactionExecutor                     0         0           5601         0                 0
MutationStage                          0         0              0         0                 0
MemtableReclaimMemory                  0         0             29         0                 0
PendingRangeCalculator                 0         0              1         0                 0
GossipStage                            0         0              0         0                 0
SecondaryIndexManagement               0         0              0         0                 0
HintsDispatcher                        0         0              0         0                 0
RequestResponseStage                   0         0              0         0                 0
Native-Transport-Requests              0         0            745         0                 0
ReadRepairStage                        0         0              0         0                 0
CounterMutationStage                   0         0              0         0                 0
MigrationStage                         0         0              0         0                 0
MemtablePostFlush                      0         0             30         0                 0
PerDiskMemtableFlushWriter_0           0         0             29         0                 0
ValidationExecutor                     0         0              0         0                 0
Sampler                                0         0              0         0                 0
MemtableFlushWriter                    0         0             29         0                 0
InternalResponseStage                  0         0              0         0                 0
ViewMutationStage                      0         0              0         0                 0
AntiEntropyStage                       0         0              0         0                 0
CacheCleanupExecutor                   0         0              0         0                 0
```

Read/write latency statistics

```
$ jconsole
localhost:7199
```

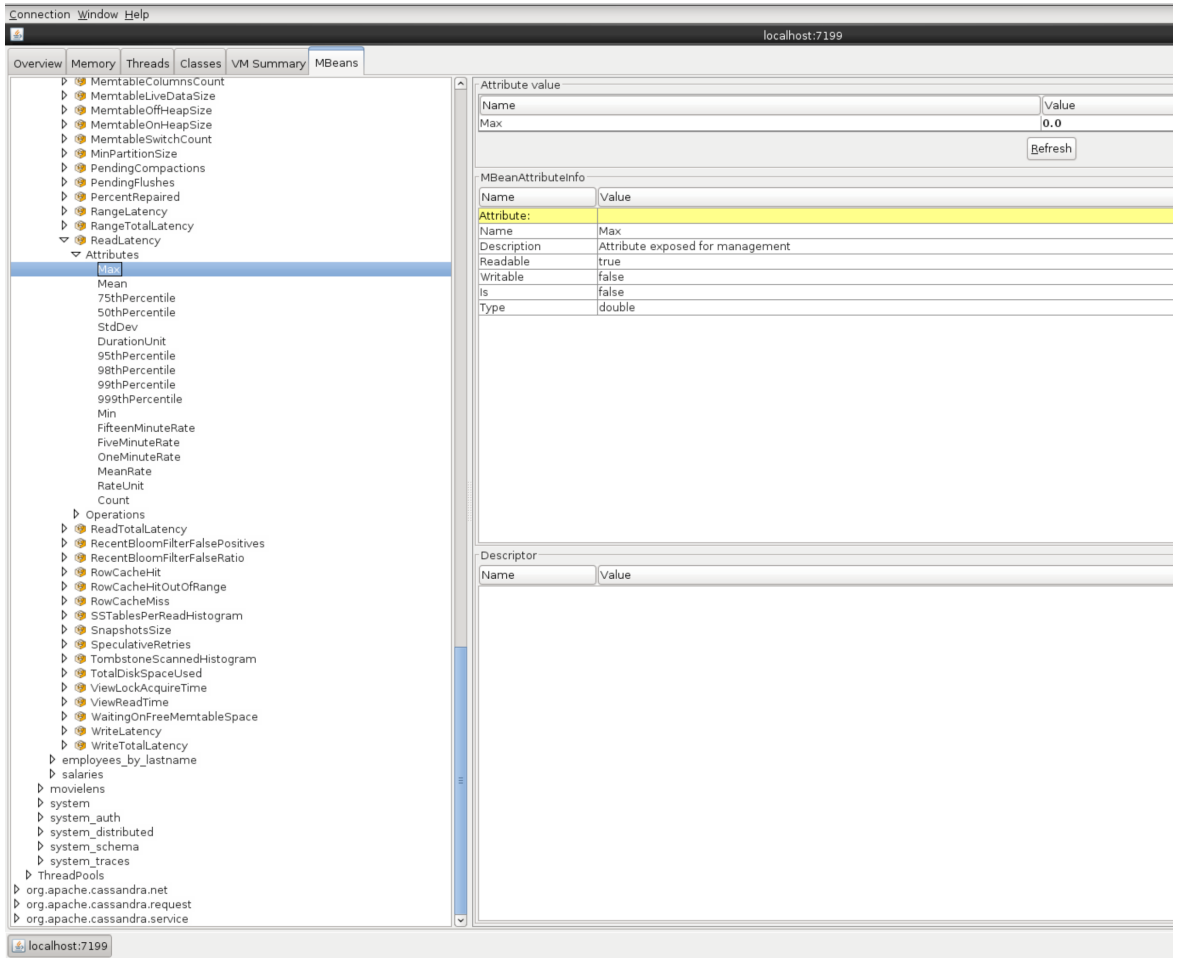

Table statistics

- For individual tables, ColumnFamilyStoreMBean provides the same general latency attributes as StorageProxyMBean. Unlike StorageProxyMBean, ColumnFamilyStoreMBean has a number of other statistics that are important to monitor for performance trends. The most important of these are:

Attribute	Description
MemtableDataSize	The total size consumed by this table's data (not including metadata).
MemtableColumnsCount	Returns the total number of columns present in the memtable(across all keys).
MemtableSwitchCount	How many times the memtable has been flushed out.
RecentReadLatencyMicros	The average read latency since the last call to this bean.
RecentWriterLatencyMicros	The average write latency since the last call to this bean.
LiveSSTableCount	The number of live SSTables for this table.