Cassandra

Monitoring and Performance Tuning

Performance Goals

- Concurrency level
- Usage pattern - Read/write ratio
- Expected peak hours
- Total volume
- Volume growth
- Read and Write Latency SLA
- Availability SLA

Example:
The cluster must support 30,000 read operations per second from the available_rooms_by_hotel_date *table with a 95th percentile read latency of 3 ms.*

Common techniques to improve performance

- Enabling SSTable compression in order to conserve disk space, at the cost of additional CPU processing.
- Throttling network usage and threads, which can be used to keep network and CPU utilization under control, at the cost of reduced throughput and increased latency.
  - Cassandra is based on a Staged Event Driven Architecture (SEDA). Cassandra separates different tasks into stages connected by a messaging service.
  - Increasing or decreasing number of threads allocated to specific tasks such as reads, writes, or compaction in order to affect the priority relative to other tasks or to support additional clients.
- Increasing heap size in order to decrease query times.

Hardware

- 8-16 physical cores
- 32 GB RAM
- Commit logs on SSD
- Data Drive SSD
  - multiple (JBOD)
- 10gbps Ethernet

Linux OS Tools

- dstat: Shows all system resources instantly. For example, you can compare disk usage in combination with interrupts from your IDE controller, or compare the network bandwidth numbers directly with the disk throughput (in the same interval).
- top: Provides an ongoing look at CPU processor activity in real time.
- System performance tools: Tools such as iostat, mpstat, iftop, sar, lsof, netstat, htop, vmstat, and similar can collect and report a variety of metrics about the operation of the system.
- vmstat: Reports information about processes, memory, paging, block I/O, traps, and CPU activity.
- iftop: Shows a list of network connections. Connections are ordered by bandwidth usage, with the pair of hosts responsible for the most traffic at the top of list. This tool makes it easier to identify the hosts causing network congestion.

Cassandra Logging

- Cassandra uses the Simple Logging Facade for Java (SLF4J) API for logging
- To change the logging level, open the file `conf/logback.xml`

```
<root level="INFO">
     <appender-ref ref="FILE" />
</root>
```

- Temporarily change the logging level using nodetool

```
$ bin/nodetool setlogginglevel org.apache.cassandra DEBUG
```

JMX Support

- Cassandra has built-in support for Java Management Extensions (JMX)
  - Low available memory detection, including the size of each generation space on the heap
  - Thread information such as deadlock detection, peak number of threads, and current live threads
  - Log level control
  - General information such as application uptime and the active classpath

$ nodetool tpstasts

```
$ bin/nodetool tpstats

Pool Name                    Active   Pending      Completed   Blocked  All time blocked
ReadStage                         0         0            216         0                 0
MutationStage                     1         0           3637         0                 0
CounterMutationStage              0         0              0         0                 0
ViewMutationStage                 0         0              0         0                 0
GossipStage                       0         0              0         0                 0
RequestResponseStage              0         0              0         0                 0
AntiEntropyStage                  0         0              0         0                 0
MigrationStage                    0         0              2         0                 0
MiscStage                         0         0              0         0                 0
InternalResponseStage             0         0              2         0                 0
ReadRepairStage                   0         0              0         0                 0


Message type           Dropped

READ                         0 RANGE_SLICE                  0 _TRACE                       0
HINT                         0 MUTATION                     0 COUNTER_MUTATION             0
BATCH_STORE                  0 BATCH_REMOVE                 0 REQUEST_RESPONSE             0
PAGED_RANGE                  0 READ_REPAIR                  0
```

$ nodetool tpstasts

- The top portion of the output presents data on tasks in each of Cassandra's thread pools.
- The bottom portion of the output indicates the number of dropped messages for the node. Dropped messages are an indicator of Cassandra's load shedding implementation, which each node uses to defend itself when it receives more requests than it can handle.
  - For example, internode messages that are received by a node but not processed within the rpc_timeout are dropped, rather than processed, as the coordinator node will no longer be waiting for a response.
- Seeing lots of zeros in the output for blocked tasks and dropped messages means that you either have very little activity on the server or that Cassandra is doing an exceptional job of keeping up with the load.

https://docs.datastax.com/en/cassandra/3.0/cassandra/tools/toolsTPstats.html

$ nodetool tablestats

- View the read and write latency and total number of reads and writes at the keyspace and table level.
- View detailed information about Cassandra’s internal structures for each table, including memtables, Bloom filters and SSTables.

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.

$ nodetool tablestats

```
bin/nodetool tablestats demo.users
Total number of tables: 39
----------------
Keyspace : demo
        Read Count: 0
        Read Latency: 9 ms.
        Write Count: 0
        Write Latency: 7 ms.
        Pending Flushes: 0
                Table: users
                SSTable count: 1
                Space used (live): 5041
(truncated output)
```

Compaction Metrics

```
$ nodetool compactionstats
```

Shows statistics about ongoing compactions. 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

$ nodetool proxyhistograms

The output of this command shows the full request latency recorded by the coordinator. The output includes the percentile rank of read and write latency values for inter-node communication. Typically, you use the command to see if requests encounter a slow node.

```
$ nodetool proxyhistograms

proxy histograms

Percentile      Read Latency     Write Latency     Range
Latency                   (micros)          (micros)          (micros)
50%                     654.95              0.00           1629.72
75%                     943.13              0.00           5839.59
95%                    4055.27              0.00          62479.63
98%                   62479.63              0.00         107964.79
99%                  107964.79              0.00         129557.75
Min                     263.21              0.00            545.79
Max                  107964.79              0.00         155469.30
```

$ bin/nodetool tablehistograms

```
$ bin/nodetool tablehistograms key_space table
```

- It omits the range latency statistics that proxyhistograms shows and instead provides counts of SSTables read per query.
- The partition size and cell count are provided, and this provides another way of identifying large partitions.

Detect Large Partitions

- In addition to the nodetool tablehistograms, you can detect large partitions by searching logs for WARN messages that reference “Writing large partition” or “Compacting large partition”. The threshold for warning on compaction of large partitions is set by the compaction_large_partition_warning_threshold_mb property in the *cassandra.yaml*file

Tracing

- If you can narrow your search down to a specific table and query of concern, you can use tracing to gain detailed insight

```
cqlsh:hotel> TRACING ON
cqlsh:hotel> SELECT * from hotels where id='AZ123';
```

- Tracing in driver program

https://github.com/abulbasar/cassandra-java-driver-examples/blob/master/src/com/example/TracingExample.java

Tuning Methodologies

- Once you've identified the root cause of performance issues related to one of your established goals, it's time to begin tuning performance.
- The suggested methodology for tuning Cassandra performance is to change one configuration parameter
- In some cases, it may be that you can get the performance back in line simply by adding more resources such as memory or extra nodes, but make sure that you aren't simply masking underlying design or configuration issues. at a time and test the results.

Caching

- Caches are used to improve responsiveness to read operations.
- Built cache types - the key cache, row cache, and counter cache

```
cql> ALTER TABLE demo.user WITH bloom_filter_fp_chance = 0.01 AND
caching = {'keys': 'ALL', 'rows_per_partition': ’100'} ;
```

Key Cache

- Cassandra’s key cache stores a map of partition keys to row index entries, facilitating faster read access into SSTables stored on disk.
- Key cache greatly increases reads without consuming a lot of additional memory, it is enabled by default
- The key_cache_size_in_mb setting indicates the maximum amount of memory that will be devoted to the key cache, which is shared across all tables. The default value is either 5% of the total JVM heap, or 100 MB, whichever is less.

Row Cache

- The row cache caches entire rows and can speed up read access for frequently accessed rows, at the cost of more memory usage.
- Caution: this can easily lead to more performance issues than it solves.
- You may want to explore file caching features supported by your operating system as an alternative to row caching.

Counter Cache

- The counter cache improves counter performance by reducing lock contention for the most frequently accessed counters. There is no per-table option for configuration of the counter cache.
- The counter_cache_size_in_mb setting determines the maximum amount of memory that will be devoted to the counter cache, which is shared across all tables. The default value is either 2.5% of the total JVM heap, or 50 MB, whichever is less.

Memtables

- Cassandra stores memtables either on the Java heap, off-heap (native) memory, or both.
- The limits on heap and off-heap memory can be set via the properties memtable_heap_space_in_mb and memtable_offheap_space_in_mb, respectively.
- By default, Cassandra sets each of these values to 1/4 of the total heap size set in the *cassandra-env.sh* file.
- Allocating memory for memtables reduces the memory available for caching and other internal Cassandra structures, so tune carefully and in small increments.

Memtable Flush

- memtable_flush_writers: 2 by default, indicates the number of threads used to write out the memtables when it becomes necessary. If your data directories are backed by SSD, you should increase this to the number of cores, without exceeding the maximum value of 8.
- If you have a very large heap, it can improve performance to set this count higher, as these threads are blocked during disk I/O.
- The memtable_flush_period_in_ms option sets the interval at which the memtable will be flushed to disk. Setting this property results in more predictable write I/O, but will also result in more SSTables and more frequent compactions, possibly impacting read performance.

Commit Logs

- You can set the value for how large the commit log is allowed to grow before it stops appending new writes to a file and creates a new one. This value is set with the commitlog_segment_size_in_mb property. By default, the value is 32 MB. This is similar to setting log rotation on a logging engine such as Log4J or Logback
- To increase the amount of writes that the commit log can hold, you'll want to enable log compression via the commitlog_compression property. The supported compression algorithms are LZ4, Snappy, and Deflate. Using compression comes at the cost of additional CPU time to perform the compression.
- Sync to disk - batch or periodic

SSTables

- Unlike the commit log, Cassandra writes SSTable files to disk asynchronously.
- If you're using hard disk drives, it's recommended that you store the datafiles and the commit logs on separate disks for maximum performance.
- If you're deploying on solid state drives (SSDs), it is fine to use the same disk.

Hinted Handoff

- We can control the bandwidth utilization of hint delivery using the property hinted_handoff_throttle_in_kb, default 1MB/sec
- Set a cap on the amount of disk space devoted to hints via the max_hints_file_size_in_mb property
- You can clear out any hints awaiting delivery to one or more nodes using the nodetool truncatehints command with a list of IP addresses or hostnames.
- Hints eventually expire after the value expressed by the max_hint_window_in_ms property.

Compaction

- SizeTieredCompactionStrategy (default)
  - Suitable for write heavy work load
- LeveledCompactionStrategy
  - This strategy should be used if there is a high ratio of reads to writes or predictable latency is required.
  - LCS will tend to not perform as well if a cluster is already I/O bound. If writes dominate reads, Cassandra may struggle to keep up.
- DateTieredCompactionStrategy
  - It is intended to improve read performance for time series data, specifically for access patterns that involve accessing the most recently written data.

Compaction

- [Table level set up] compaction threshold. The compaction threshold refers to the number of SSTables that are in the queue to be compacted before a minor compaction is actually kicked off. By default, the minimum number is 4 and the maximum is 32.
- `$ nodetool getcompactionthreshold demo user`
- Throttle compaction as across the cluster (compaction_throughput_mb_per_sec in the *cassandra.yaml* file). Setting this value to 0 disables throttling entirely, but the default value of 16 MB/s is sufficient for most non-write-intensive cases.
- If this does not fix the issue, you can increase the number of threads dedicated to compaction by setting the concurrent_compactors property

Compaction History

- The nodetool compactionhistory command prints statistics about completed compactions, including the size of data before and after compaction and the number of rows merged.

$ bin/nodetool compactionhistory

Concurrency and threading

- The concurrent_reads setting determines how many simultaneous read requests the node can service.
- This defaults to 32, but should be set to the number of drives used for data storage × 16.
- This is because when your data sets are larger than available memory, the read operation is I/O bound.

Concurrency and threading

- concurrent_writes should correlate to the number of clients that will write concurrently to the server.
- If Cassandra is backing a web application server, you can tune this setting from its default of 32 to match the number of threads the application server has available to connect to Cassandra.
- It is common in Java application servers to prefer database connection pools no larger than 20 or 30, but if you're using several application servers in a cluster, you'll need to factor that in as well.

Concurrency and threading

- max_hints_delivery_threads: Maximum number of threads devoted to hint delivery
- memtable_flush_writers: Number of threads devoted to flushing memtables to disk
- concurrent_compactors: Number of threads devoted to running compaction
- native_transport_max_threads: Maximum number of threads devoted to processing incoming CQL requests

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

- For heap sizes from 14 GB to 64 GB, G1 performs better than 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.

https://github.com/jbenninghoff/cluster-validation

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' }
```

Cassandra Stress

- The cassandra-stress tool is a Java-based stress testing utility for basic benchmarking and load testing a Cassandra cluster.
- Data modeling choices can greatly affect application performance. Significant load testing over several trials is the best method for discovering issues with a particular data model. The cassandra-stress tool is an effective tool for populating a cluster and stress testing CQL tables and queries.
- Use cassandra-stress to determine how a schema performs.
  - Understand how your database scales.
  - Optimize your data model and settings.
  - Determine production capacity.

$ cassandra-stress help

```
$ tools/bin/cassandra-stress help
$ tools/bin/cassandra-stress print
dist=FIXED\(10\)
$ tools/bin/cassandra-stress help -insert
$ tools/bin/cassandra-stress help user
$ tools/bin/cassandra-stress help -mode
```