DATASTAX

DataStax Enterprise 4.0 In-Memory Option

A look at performance, use cases, and anti-patterns

White Paper

Table of Contents

Abstract

DataStax Enterprise 4.0 introduces a new in-memory option that brings the power and flexibility of Cassandra to an in-memory database. This paper discusses a number of performance tests performed by DataStax with the in-memory option along with recommendations on when and when not to use 4.0's in-memory tables.

Introduction

DataStax's new in-memory option allows users to create memory-only database objects with faster performance over objects that are standard disk or SSD-based, while ensuring data protection so that data contained in memory-only objects is never lost. From a developer perspective, there is no difference in working with an in-memory database object over a traditional disk-based table.

With the in-memory option, DataStax Enterprise gives administrators the ability to tune the performance needed for a particular segment of data. 'Cold' data that does not require any stringent performance SLA's can be directed to traditional spinning disks. 'Hot' data needing faster performance can be placed on SSD's, while data needing the absolute fastest response times can be placed in-memory. All of this can be done in one database instance of DataStax Enterprise.

In-memory tables provide complete data durability. Data is not lost when a machine loses power or suffers a technical problem. All writes are written to a commit log as well as to a memory table so new data is fully protected. Users can customize how often memory tables are flushed to disk for backup or other purposes.

Read operations on memory tables only address data in memory unlike reads against standard disk or SSD tables that will reference data caches but also disk for either single or multiple SStable reads.

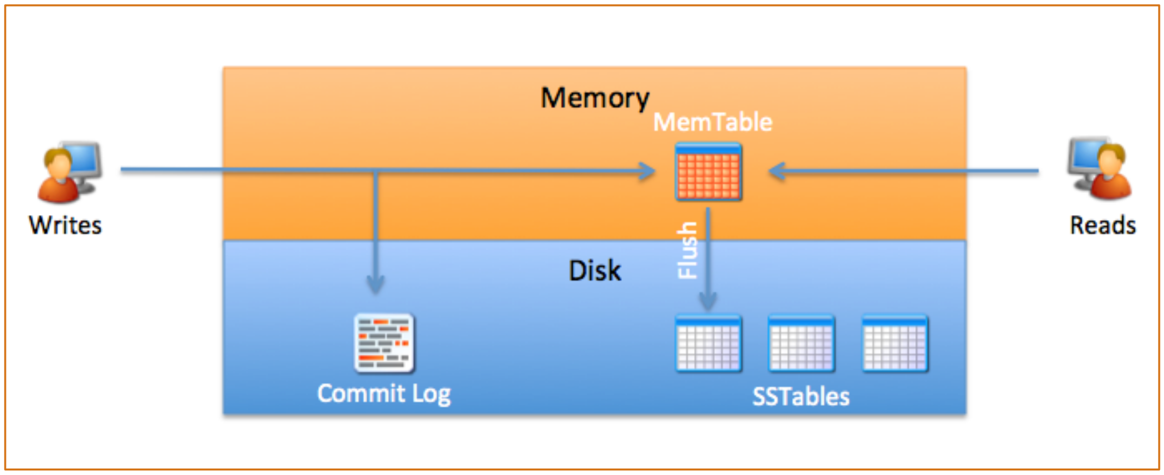

Figure 1 – DSE 4.0 In-Memory Option Architecture and Process

Creating an in-memory table is simple and is done through standard CQL statements (CREATE/ALTER):

```
CREATE TABLE users (
  uid text,
  fname text,
  lname text,
  PRIMARY KEY (uid)
) WITH compaction={‘class’: ‘MemoryOnlyStrategy’, ‘size_limit_in_mb’: 256}
AND caching = ‘NONE’;
```

The above example creates a memory-based table that is limited to 256MB in size. Once created, data can be inserted, modified, and queried like any other table.

Performance Implications of In-Memory Tables

Because disk IO is no longer an issue, in-memory tables can easily handle a workload with heavy and random overwrites which would have previously resulted in fragmenting a partition among many sstables on disk. This opens Cassandra up to a new range of applications that may previously have required a separate memory resident system although currently these memtables are still on the JVM heap. Workloads that require reading from on-disk sstables especially benefit from in-memory where the speedup can be anywhere from 10x to >100x for read operations.

Performance Benchmark Examples

The magnitude of speedup for in-memory tables is directly related to the amount of sstables Cassandra would have needed to access with on disk compaction. As an example, in a test performed with EC2 m1.large instances, the first sstable seems to add roughly 200~300mu of latency (at 50th percentile intra-process latency). This number grows under load with additional sstables. The increasing delay is most likely due to the amount of recently written files that can be kept in the OS file cache. Since actual disk spindle I/O requires somewhere between 1-10ms, the delay becomes quite large once the page cache is exhausted.

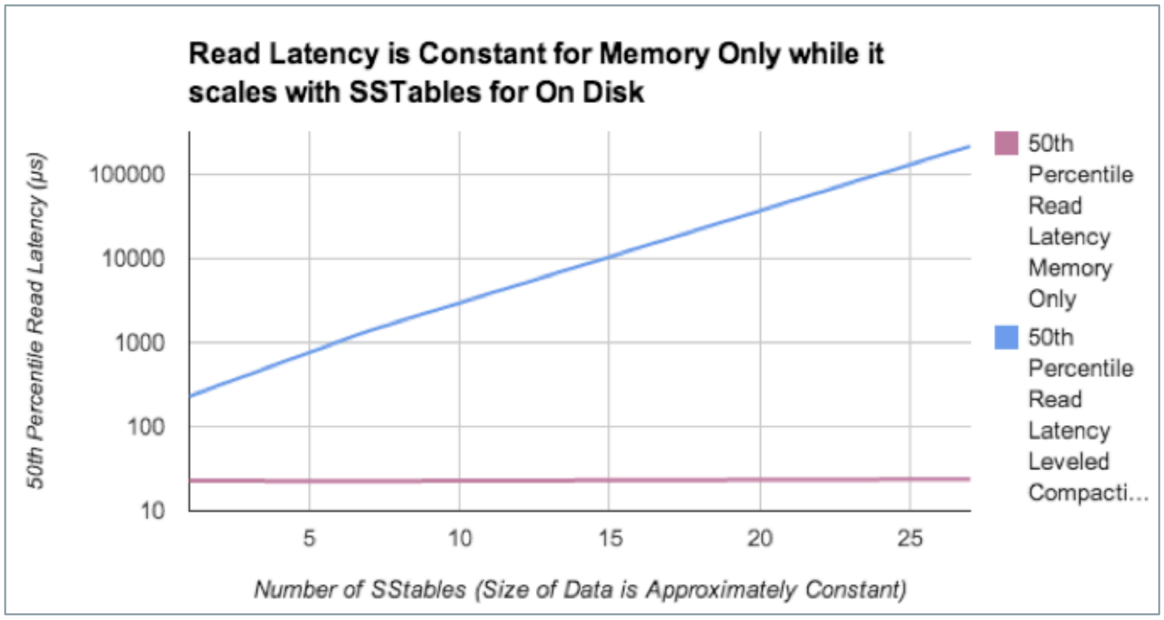

Figure 2 - Serial read / write (200K ops over 7K partitions). Reads against a single m1.large ec2 instance after a heavy overwrite workload

From Figure 2, it can be determined that for the exact same workload, the in-memory table's average latency (indicated in pink) is constant relative to the amount of sstables that would have been created in a comparative Leveled Compaction Strategy workload. Most normal applications will seldom access more than a single digit number of sstables (the left quarter of the graph), but the improvement is considerable even in that regime. Although the difference in microseconds may seem small on this chart, it changes by an order of magnitude with the number of read requests that a single CPU can perform on a table.

Effects of the Page Cache

Linux operating systems use a Page Cache to speed I/O from slow spindle disks. Page Caches retain blocks of files in memory after both read and write operations allowing them to be rapidly accessed again. Because of this feature, systems that are *not* using the majority of their RAM will find their sstables held in page cache and not benefit from in-memory tables.

The workloads run below have been designed to stress the operating system to the point that it is unable to successfully keep all of the sstable blocks in memory thus showing the effective slowdown caused by disk I/O. The maximum possible improvement that can be gained from in-memory tables is when the page cache has been exhausted completely.

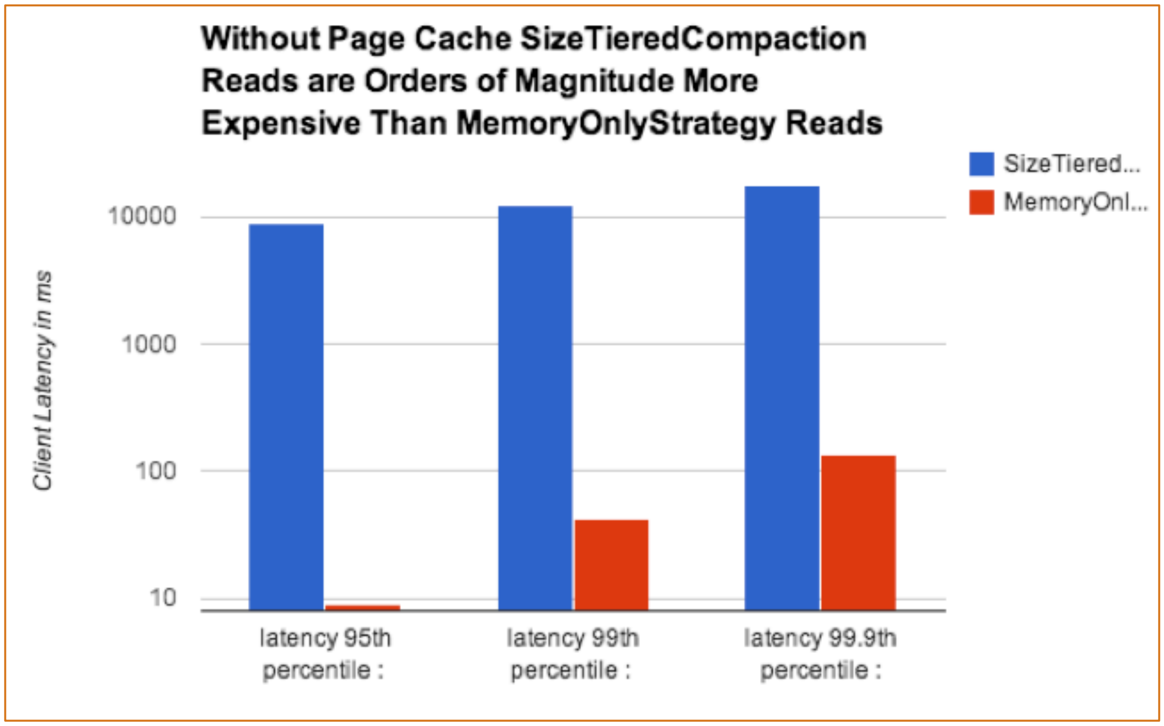

Figure 3 – Results of in-memory vs. disk-based tables with page cache exhaustion.

The 1,000-fold improvement seen in Figure 3 is unlikely without a difficult workload running in parallel, and gives a good indication of how much faster RAM access is as compared to disk I/O. In most systems there will always be some page-cache speeding up a portion of read requests so speedups on the order of 10~100x will be more likely.

Workload Recommendations and Cautions

The following sections contain information on workloads known to benefit from DSE 4.0 in-memory tables, as well as other workloads that will not benefit from in-memory tables.

Recommended Workloads

Semi-Static and Overwrite Workloads

Concurrent read / write loads that force multiple sstables will see the greatest improvement from DataStax's in-memory option. For example, holding the last known position of a set of GPS device in an in-memory table while using a separate on-disk table for persistence and logging is a viable workload for in-memory.

This "last location" table would require a write pattern which would have distributed partition keys amongst sstables with an on disk workload and required a great deal of disk I/O even though the size of information that is being updated is static. For an on-disk table, this would require continuous compaction to remain manageable, but an in-memory table does not require disk I/O nor will it actively change in size.

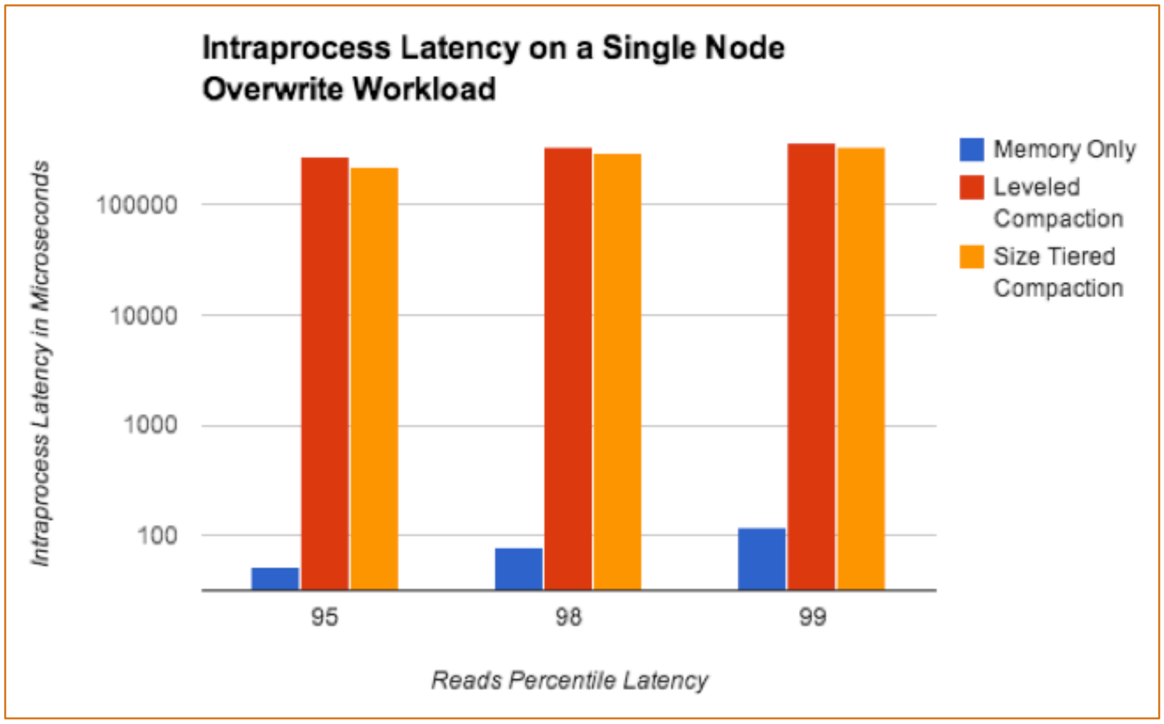

Figure 4 – Sample overwrite performance test. Interleaved read / write (1 million overwrite/read operations over 7K partitions).

The above graph reports the server side process latency (log scale) for read requests in a scenario with AWS M1.xlarge attempting to overwrite and read (3/1 ratio) to a table in a single M1.large Cassandra node. In blue, the ability of in-memory tables to skip costly disk IO is shown with orders of magnitude less latency than on disk-based tables. In this particular experiment read requests hit on average between 5 and 6 sstables.

Performing a similar experiment on a 5 node cluster with RF=3 and Consistency Quorum on all operations shows similar benefits, but to a lesser degree do to the additional latency added by network communications. Internode communications show that the difference in performance directly affects the ability of a coordinating node to execute a read operation. Coordinator Read Latency includes the time required to receive a response from replicas for a CL>ONE read request.

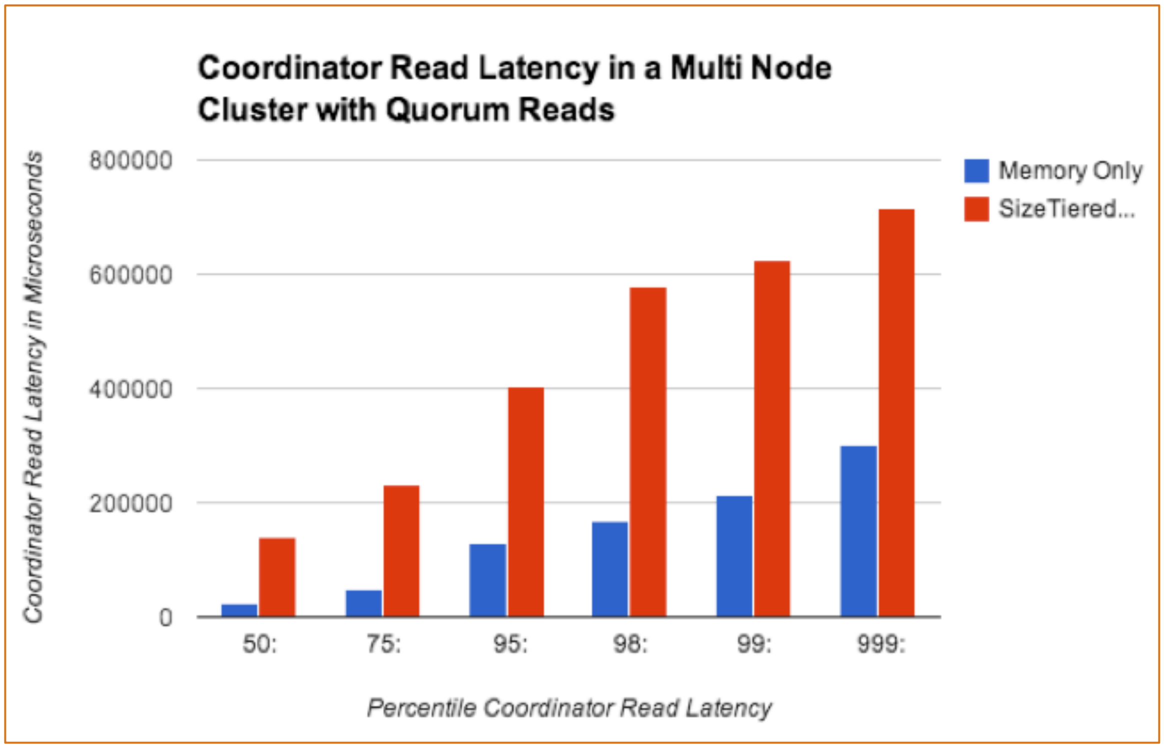

Figure 5 – Sample multi-node workload test.

Both processes seem to be greatly affected by the relative network latency in the cluster (visible in the upper percentiles of latency). Best performance in this situation seems to be requests at Consistency Level=One. Although the network is dominating latency in this situation, the nodes internally receive the same benefit as in the single node test and are able to process many more reads.

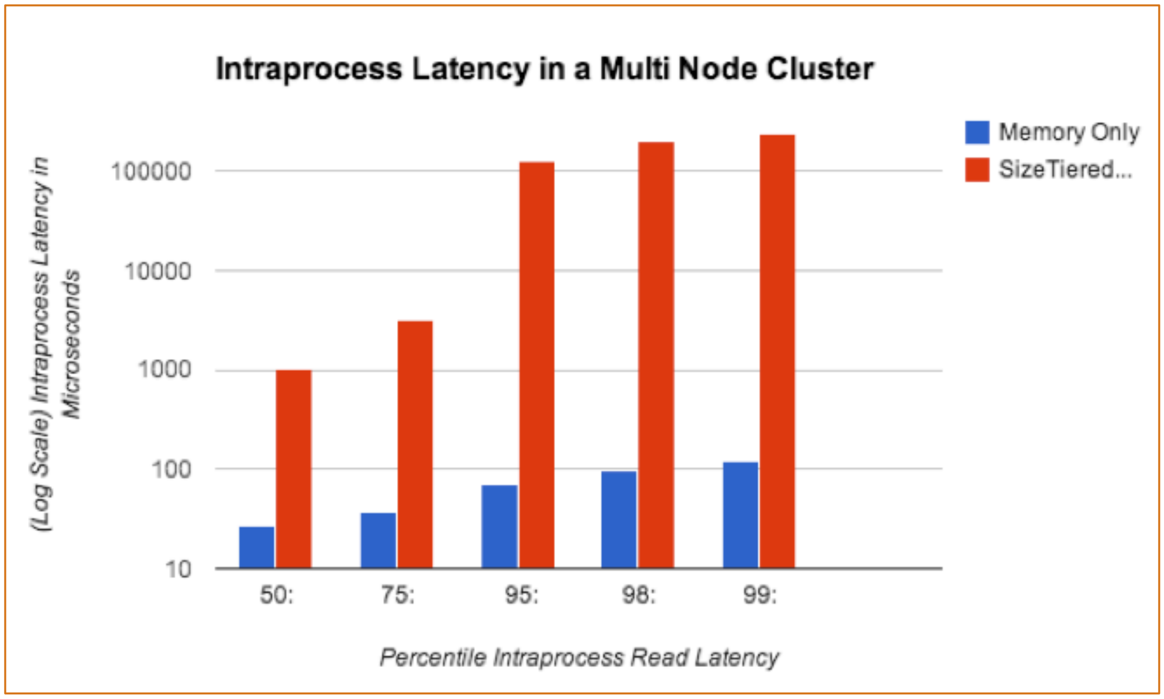

Figure 6 – Intraprocess latency on multi-node cluster test.

The scale on the Figure 6 graph is logarithmic because of the dramatic difference in latency times between on memory and on-disk compaction options.

Workloads Involving Counters

Because counters by design require a read for every write at RF > 1, it is important that a counter table be able to rapidly read its data. As such, they are ideal for in-memory tables. It is also important to set the *memtable_flush_period_in_ms* parameter for any counter table to a low value.

Workloads to Avoid

Heavy Delete and Time-To-Live Workloads

Delete operations create tombstones just as in normal on disk tables. The difference is that with in-memory tables, tombstones remain resident in memory until *GC Grace Seconds* pass. With current default settings every tombstone will remain in RAM for 10 days before it deleted. Hence, any workload using TTL's will make the RAM unavailable unless *GC_GRACE* is lowered. This makes the system vulnerable to dead data being restored. In either case the best solution most likely is to switch to an overwrite-only workload.

Monotonically-Growing Workloads

When an in-memory table has reached its *SIZE_LIMIT_IN_MB* all future mutations are dropped and the table will most likely convert to an on-disk strategy. This situation must be avoided. DataStax OpsCenter makes it easy to monitor such workloads with alerts that can proactively notify an administrator when an in-memory table is approaching is maximum size.

Other Considerations and Limitations

Because in-memory tables are stored on the JVM heap, the total SIZE_LIMIT_IN_MB for an in-memory table should currently be limited to 1GB per node, with amount also being dependent on the heap size and the need to leave room for normal Cassandra activity.

An upcoming version of DSE will greatly increase the amount of RAM an in-memory table can utilize per node.

Monitoring and Troubleshooting Performance of In-Memory Tables

The creation of too many in-memory tables can easily lead to GC spirals and cluster instability where badly formed workload may end up with a block of useless memory. The following best-practice recommendations ensure users get the most from the system when monitoring and troubleshooting in-memory performance.

Reaching Maximum In-Memory Table Size

Cassandra does not read before it writes so it is difficult to determine at write time whether a mutation is adding to the growth of an in-memory table. Once an in-memory table's *AllMemtablesDataSize*[1] has been reached, all future mutations are dropped even if they did not affect the table's size. In this situation, the administrator has the option to (1) truncate; (2) alter compaction strategy to an on-disk strategy or (3) only perform reads. DataStax OpsCenter can be used to monitor the size of in-memory table sizes and set alerts that inform administrators when a table is approaching its maximum size.

[1] JMX o.a.c.metrics.columnfamilies.keyspace.table.AllMemTablesDataSize

Performance Expectations Not Being Met

For smaller workloads and systems where a majority of RAM is not in use and being actively paged by other I/O activity, the performance of in-memory tables will be identical to an on-disk table. This means that the workload is easy and the system is already benefiting from in-memory performance.

Conclusion

DataStax Enterprise 4.0's in-memory option is simple to use and supplies another performance knob that can be used to increase performance for certain use application use cases. For downloads of DataStax Enterprise, online documentation, client drivers, getting started materials and more, visit www.datastax.com.

About DataStax

DataStax provides a massively scalable enterprise NoSQL platform to run mission-critical business applications for some of the world's most innovative and data-intensive enterprises. Powered by the open source Apache Cassandra™ database, DataStax delivers a fully distributed, continuously available platform that is faster to deploy and less expensive to maintain than other database platforms.

DataStax has more than 400 customers in 38 countries including leaders such as Netflix, Rackspace, Pearson Education, and Constant Contact, and spans verticals including web, financial services, telecommunications, logistics, and government. Based in San Mateo, Calif., DataStax is backed by industry-leading investors including Lightspeed Venture Partners, Meritech Capital, and Crosslink Capital.

For more information, visit www.datastax.com.

Appendix

Schema used for testing:

```
CREATE KEYSPACE ks WITH replication = {
 'class': 'SimpleStrategy',
 'replication_factor': '1'
};

USE ks;

CREATE TABLE mem_hl (
 key text,
 clusterkey text,
 col1 text,
 col2 text,
 col3 text,
 col4 text,
 PRIMARY KEY (key, clusterkey)
) WITH
 bloom_filter_fp_chance=0.010000 AND
 caching='KEYS_ONLY' AND
 comment='' AND
 dclocal_read_repair_chance=0.000000 AND
 gc_grace_seconds=864000 AND
 index_interval=128 AND
 read_repair_chance=0.100000 AND
 replicate_on_write='true' AND
 populate_io_cache_on_flush='false' AND
 default_time_to_live=0 AND
 speculative_retry='99.0PERCENTILE' AND
 memtable_flush_period_in_ms=3600000 AND
 compaction={'size_limit_in_mb': '800', 'class': 'MemoryOnlyStrategy'} AND
 compression={'sstable_compression': 'LZ4Compressor'};
```

The test pattern included inserts and reads where the test cycled over possible values for the clustering key and partition key in order to maximize the distribution of overwrites and the difficulty of the read task. Using relatively prime numbers means that it requires partition * clustering operations before a rewrite actually occurs.

For example:

Op 1 : Insert Partition 1 ClusteringKey 1 “String Op1” “String Op1” “String Op1” “String op1”
Op 2 : insert Partition 2 ClusteringKey 2
Op 3 : insert Partition 3 ClusteringKey 3
Op 4 : Insert Partition 4 ClusteringKey 1
Op 5 : insert Partition 5 ClusteringKey 2
Op 6 : insert Partition 6 ClusteringKey 3
Op 7 : Insert Partition 7 ClusteringKey 1
Op 8 : insert Partition 1 ClusteringKey 2
Op 9 : insert Partition 2 ClusteringKey 3
…
Op 22: Insert Partition 1 ClusteringKey 1 “String Op22” “String Op22” “String Op22” “String op22”