Benchmarking Apache Cassandra™ and DataStax Enterprise

Published: June 2018
https://zdatainc.com/

Table of Contents

Executive Summary

zData performed an evaluation of the performance differences between open source Apache Cassandra™ and DataStax Enterprise in May 2018. The overall conclusions of the performance tests were that DataStax Enterprise outperformed open source Cassandra by a wide margin in every test.

Overview

NoSQL databases have become the standard data platform for what can be characterized as “cloud” applications - that is, real-time applications that are distributed in nature and require constant uptime, instant elasticity and scale, and an ability to tackle multiple workloads and data formats in the same database. When it comes to performance, it should be noted that there is no single “winner takes all” in every use case or benchmark. Depending on the scenario, it is always possible for one database to outperform its competitors and yet lag them when the rules of engagement change.

When comparing performance among databases, it is always recommended that the engines in consideration be tested under the specific use case and deployment conditions intended for a particular production application. Still, general competitive benchmarks of usual-and-customary application workloads can be useful to those evaluating database performance.

This paper documents the results of benchmark tests that compared open source Apache Cassandra and DataStax Enterprise to verify performance optimizations in the latest DataStax Enterprise version that gives it an edge over Cassandra.

Configurations and Test Methodology

The benchmark tests were performed on Amazon Web Services EC2 instances, a recognized platform for hosting horizontally scalable software such as the tested databases. The tests were run on AWS Instance Types: i3.8xlarge for 5 database nodes and c3.xlarge for 10 stress client nodes. The test data was stored on NVmE local SSD and amounted to 2.7TB of data across each cluster. The operating system was Ubuntu 16.04.

The version of open source Apache Cassandra used was 3.0.16. The versions of DSE tested were 5.1.8 and DSE 6.0.0.

An IoT use case scenario was used for the benchmark using the DSE 6.0.0 cassandra-stress tool as the testing tool. A group of ten Cassandra Stress nodes were used to parallelize data generation across the target five-node database clusters and as the source of load for the tests.

Load Routine

The stress.yaml file (Appendix A) was placed on each of the 10 Stress instances. Each iteration of the following stress command yielded approximately 500GB of data.

Total Data Size: ~500G / DSE node | ~2.5TB total
Individual Row Size: ~64 Bytes / Row
Total rows to write: 1.4B / node | 14B Total
Total partitions to write: 14M / node | 140M Total

Each stress client will write a subset of the data, we do this by changing the seq= param in the command line below.

Client 1: seq=1..14000000
Client 2: seq=14000001..28000000
Client 3: seq=28000001..42000000
…

```
/cassandra-stress user profile=stress.yaml ops\(insert=1\) n=1400m cl=QUORUM no-warmup -mode native cql3 protocolVersion=3 -errors ignore -rate threads=350 -pop seq=1..14000000 contents=SORTED -insert visits=fixed\(100\) -node <host1>,<host2>,<host3> -log file=seq1.log hdrfile=seq1.hdr -graph file=seq1.html title=seq1
```

A snapshot was performed after the data load for each software version tested (Cassandra 3, DSE 5, and DSE 6). All data was restored from the snapshot between major test runs of each version.

The CQL Data Modeler can be used to visualize how data is stored on disk and to verify the data load. The CQL Data Modeler runs the nodetool status command and runs the following CQL queries.

Find a machine_id to query:

```
SELECT machine_id, sensor_value, time FROM iot_space.iot_table LIMIT 1;
```

Query that partition to see it's rows:

```
SELECT machine_id, sensor_value, time FROM iot_space.iot_table WHERE machine_id =your_machine_id_from_last_query LIMIT 1000;
```

Find Maximum Performance Level

There are a few ways to find optimal performance level depending on the use case. For this test, we replicated an IoT use case which often involves a fixed number of partitions with often written or read-to rows. We used a read-only test to find the optimal thread count because it determined the right number of threads for the heaviest workload (reads) and warms the cache.

We started by having the 10 Stress nodes run a 90% write, 10% read workload to measure how we were stressing the cluster and to work towards maximizing the peak performance of the cluster when no maintenance tasks were occurring and then backing off to a workload that is recommended customers run in a typical scenario.

Each Stress node ran the following stress test

```
cassandra-stress user profile=stress.yaml ops\(query_by_machine_id=1\) duration=10m
cl=ONE no-warmup -mode native cql3 protocolVersion=3 -errors ignore -rate threads=350
-pop seq=1..14000000  contents=SORTED -insert visits=fixed\(100\) -node
<host1>,<host2>,<host3> -log file=Thread_Test_10Mpop_seq1.log
hdrfile=Thread_Test_10Mpop_seq1.hdr -graph file=Thread_Test_10Mpop_seq1.html
title=Thread_Test_10Mpop_seq1
```

Ideally, the number of threads is tuned up or down in increments of 50 to achieve an optimal latency to match your SLA or until you're maxing out your DSE cluster. An easy way to monitor saturation levels is to watch CPU saturation levels increase with threadcounts. When 450 threads is reached without hitting the maximum (CPU's between 90 and 100% utilization), it's the signal to add additional Stress nodes.

IoT Workload Selection

It is impractical to test at maximum performance (and pretty unfair to older software versions), so tests were conducted at 75% of the max performance level. For the NVMe backed AWS instances, this put us at 350 threads per Stress node. Three workload tests show the range of the system included:

- 90% write and 10% read
- 50% write and 50% read
- 10% write and 90% read

This testing routine emphasizes a partially cached set of partitions that represents a typical IoT use case. Because each test involves some writes, the data was restored from snapshot between each test.

Workload Results

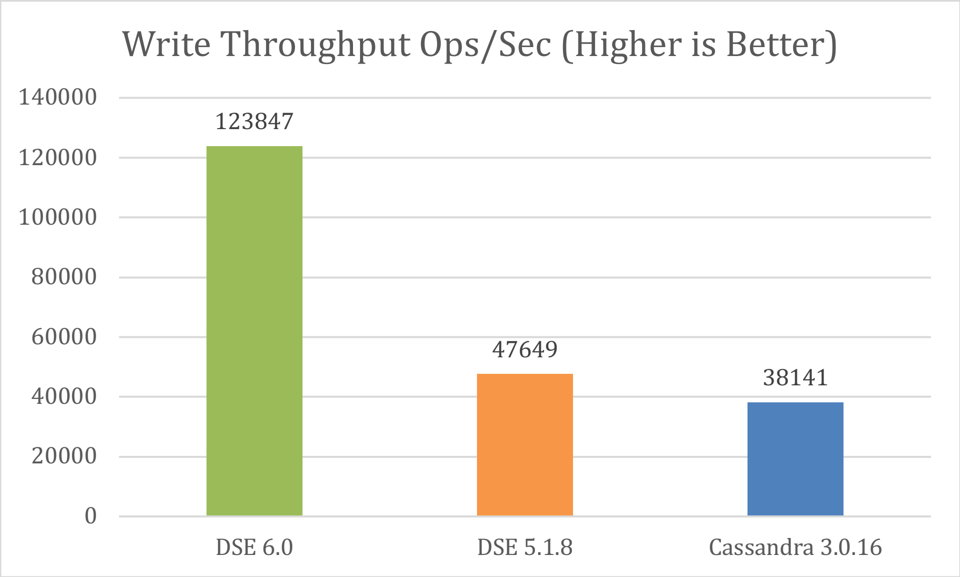
90% Write and 10% Read

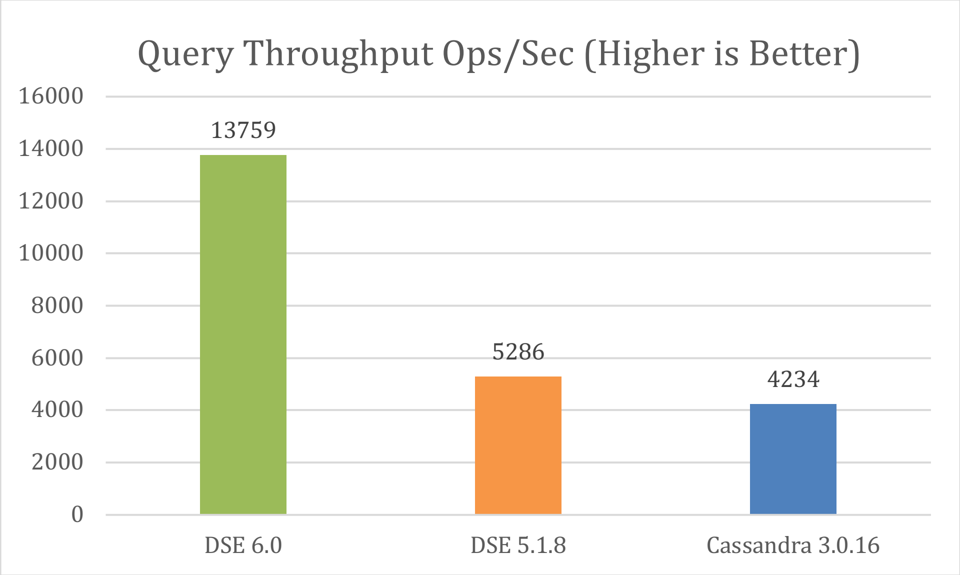
The following cassandra-stress command was used on all Stress nodes each of which lists (-n) all DSE/Cassandra nodes.

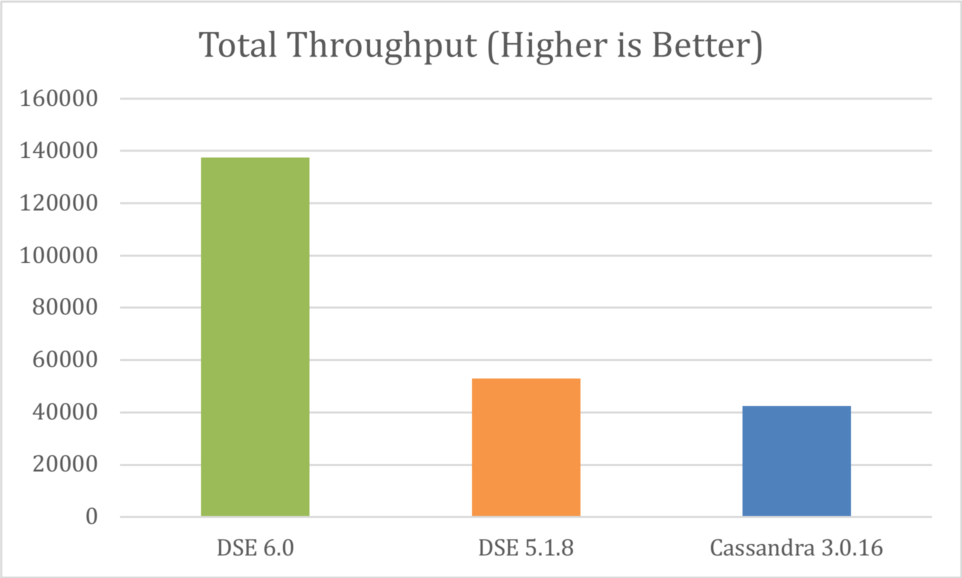
```
cassandra-stress user profile=stress.yaml ops\(insert=9,query_by_machine_id=1\)
duration=1hr  cl=ONE no-warmup -mode native cql3 protocolVersion=3 -errors ignore -
rate threads=350 -pop seq=1..14000000 contents=SORTED -insert visits=fixed\(100\) -
node <host1>,<host2>,<host3> -log file=90Write_10Read_10Mpop_seq1.log
hdrfile=90Write_10Read_10Mpop_seq1.hdr -graph file=90Write_10Read_10Mpop_seq1.html
title=90Write_10Read_10Mpop_seq1
```

This workload is displayed in more detail because it is more typical of the described use case. The following figures show the throughput/operations-per-second (**more is better**) graphed vertically and the time in which the test was run graphed horizontally.

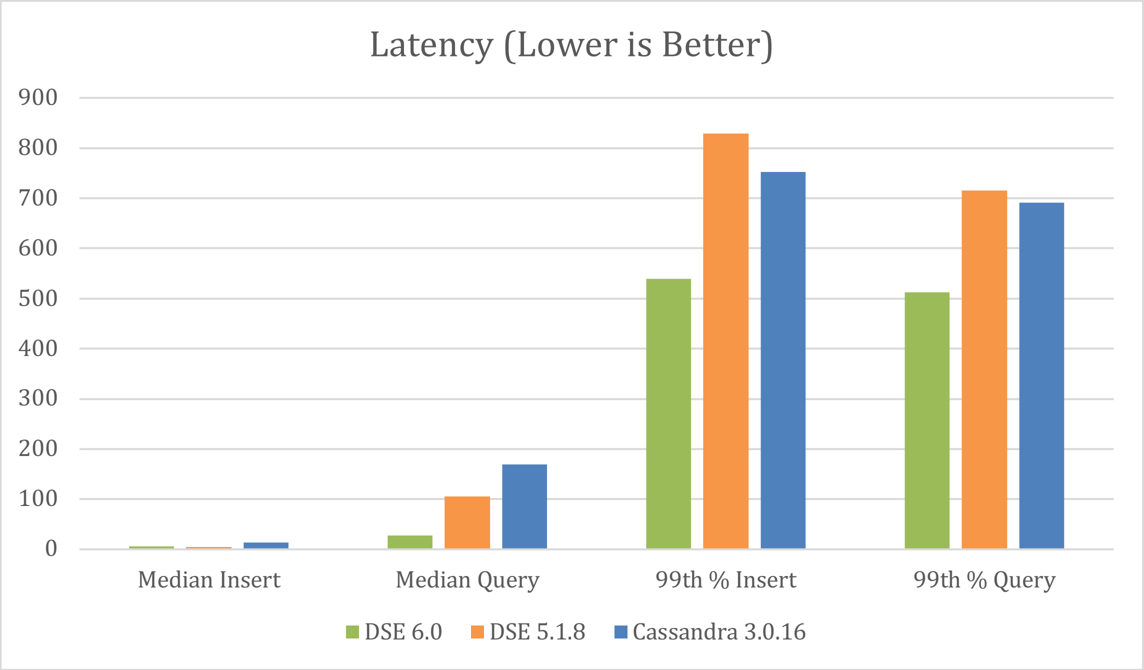
You can see in these figures that DSE 6 (green) delivered an average of 123,847 write operations per second for the same workload, Cassandra 3.0.16 averaged 38,141 ops/sec and DSE 5.1.8 delivered 47,679 ops/sec. For the query portion of the test, DSE 6 delivered 13,759 queries per second, Cassandra 3.0.16 averaged 4,234 ops/sec, and DSE 5.1.8 delivered 5,286 ops/sec.







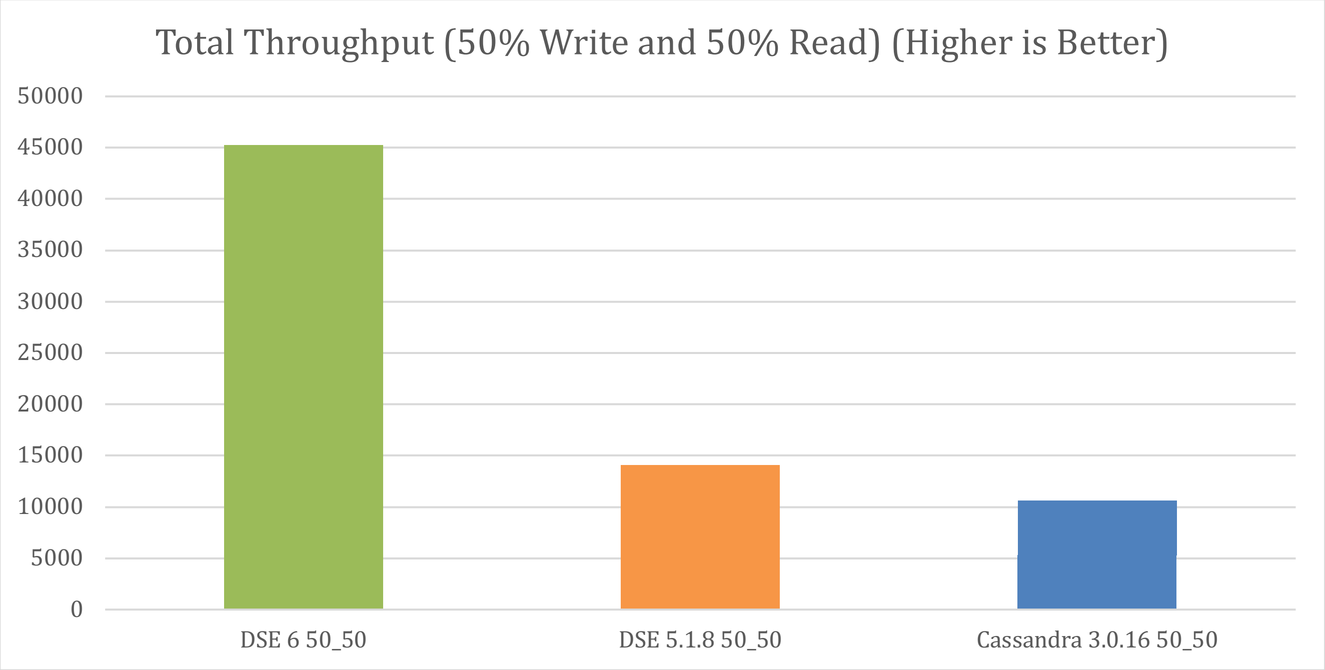
Despite this test being designed to maximize throughput, DSE 6 is generally able to show lower latency. A typical expectation is that lowering the thread count would widen the latency gaps.

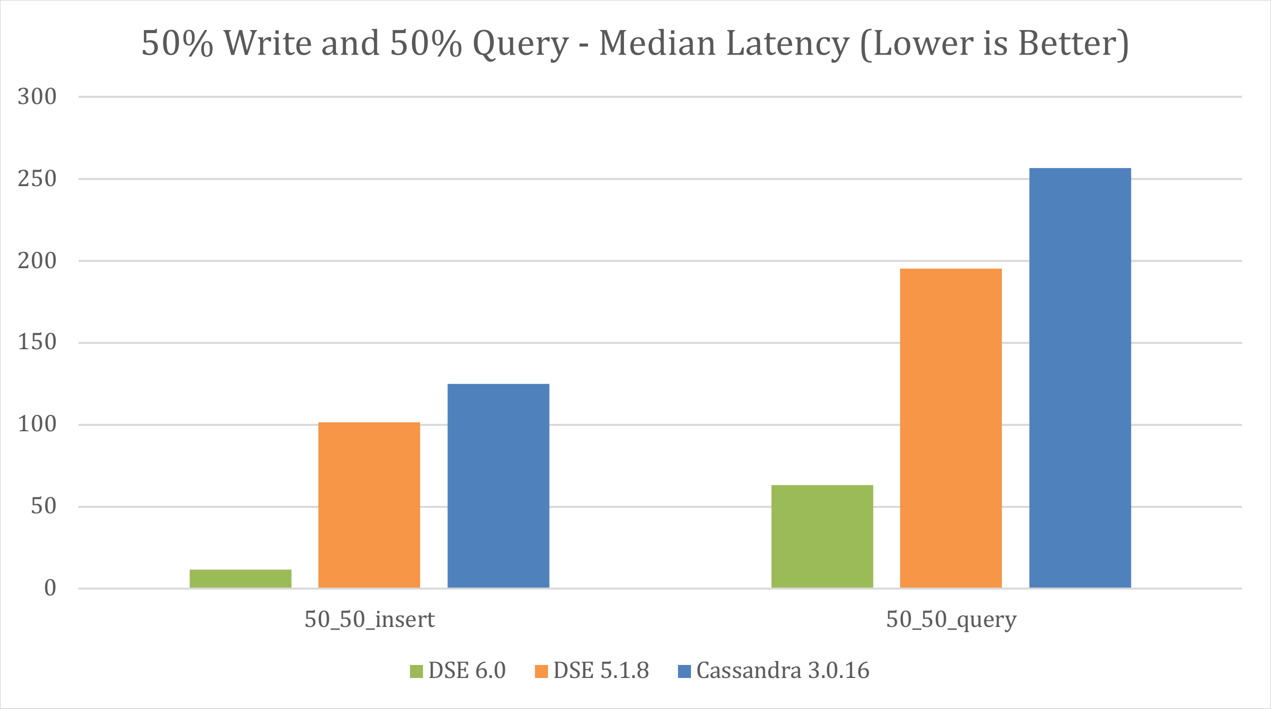


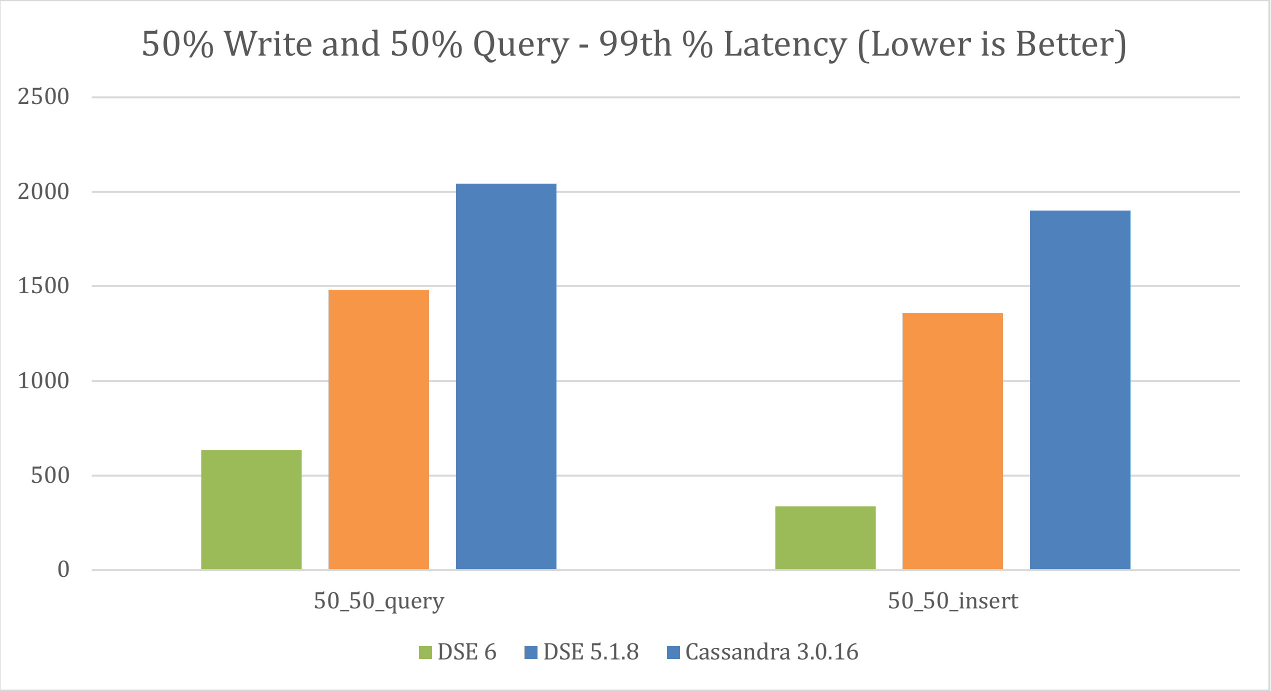
50% Write and 50% Read

Before running this test, we restored from snapshot and ran a 100% read 10-minute stress test.

```
cassandra-stress user profile=stress.yaml ops\(insert=5,query_by_machine_id=5\)
duration=1hr  cl=ONE no-warmup -mode native cql3 protocolVersion=3 -errors ignore -
rate threads=350 -pop seq=1..14000000 contents=SORTED -insert visits=fixed\(100\) -
node <host1>,<host2>,<host3> -log file=50Write_50Read_10Mpop_seq1.log
hdrfile=50Write_50Read_10Mpop_seq1.hdr -graph file=50Write_50Read_10Mpop_seq1.html
title=50Write_50Read_10Mpop_seq1
```



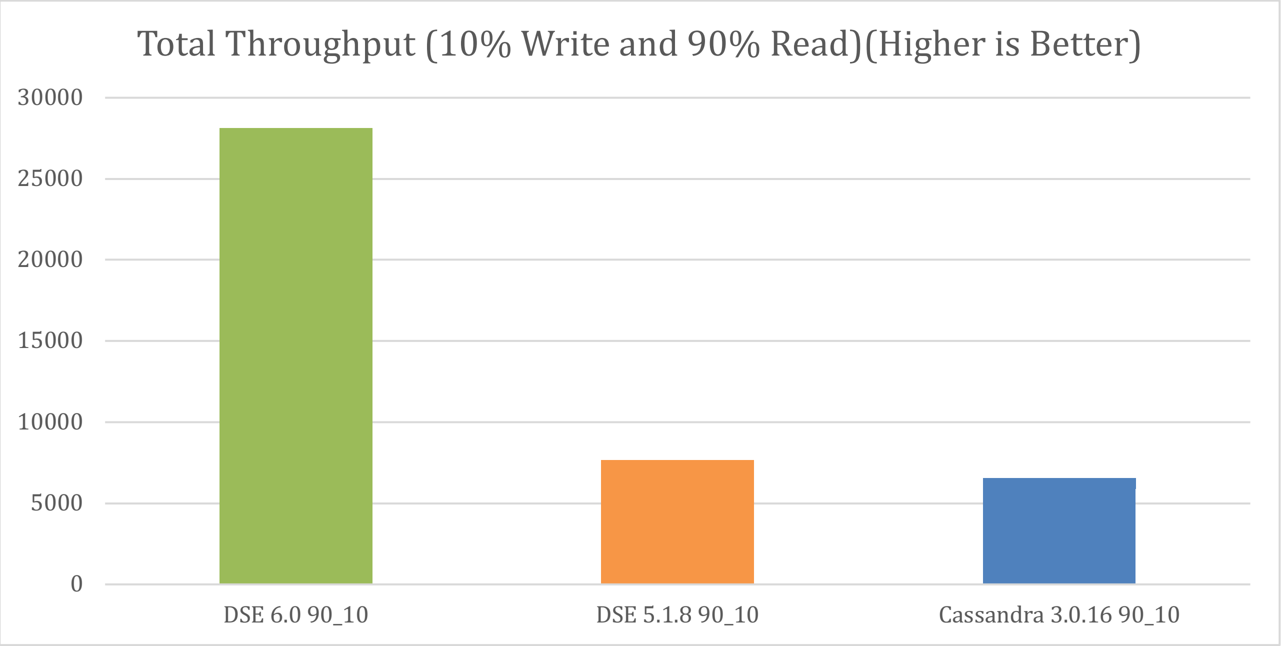

50% Write and 50% Query - Median Latency (Lower is Better)
300
250
200
150
100
50
0
50_50_insert
50_50_query
🟩 DSE 6.0
🟧 DSE 5.1.8
🟦 Cassandra 3.0.16

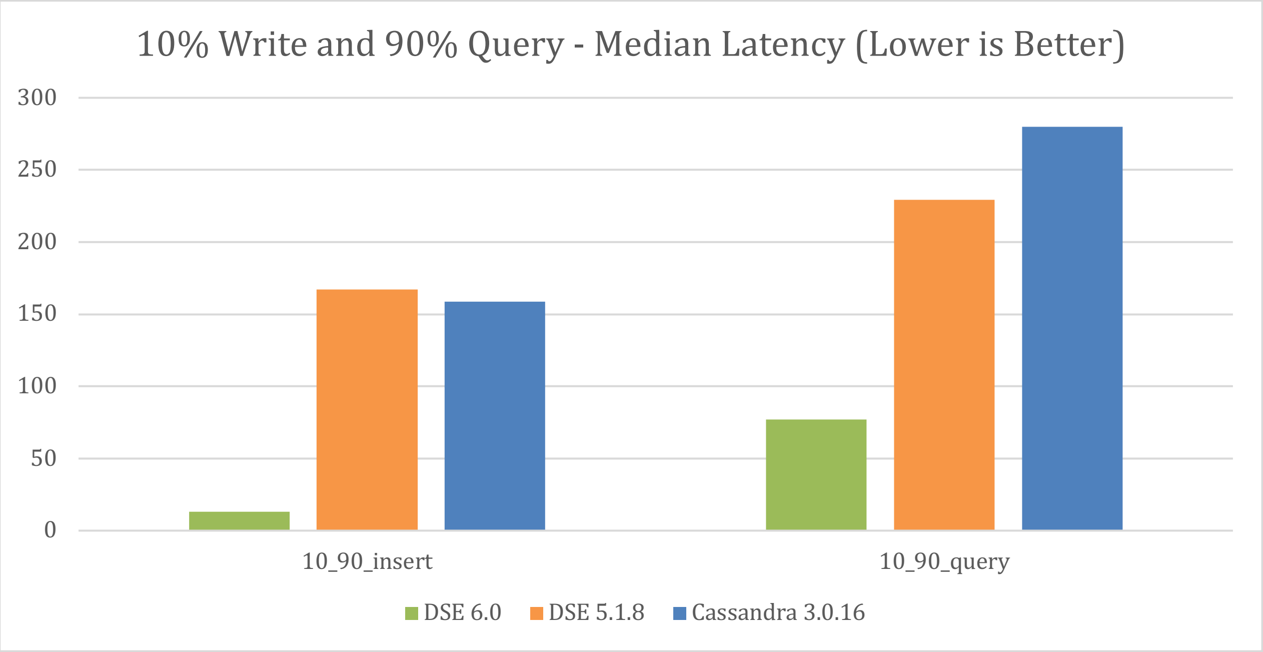

50% Write and 50% Query - 99th % Latency (Lower is Better)
2500
2000
1500
1000
500
0
50_50_query
50_50_insert
🟩 DSE 6
🟦 DSE 5.1.8
🟦 Cassandra 3.0.16

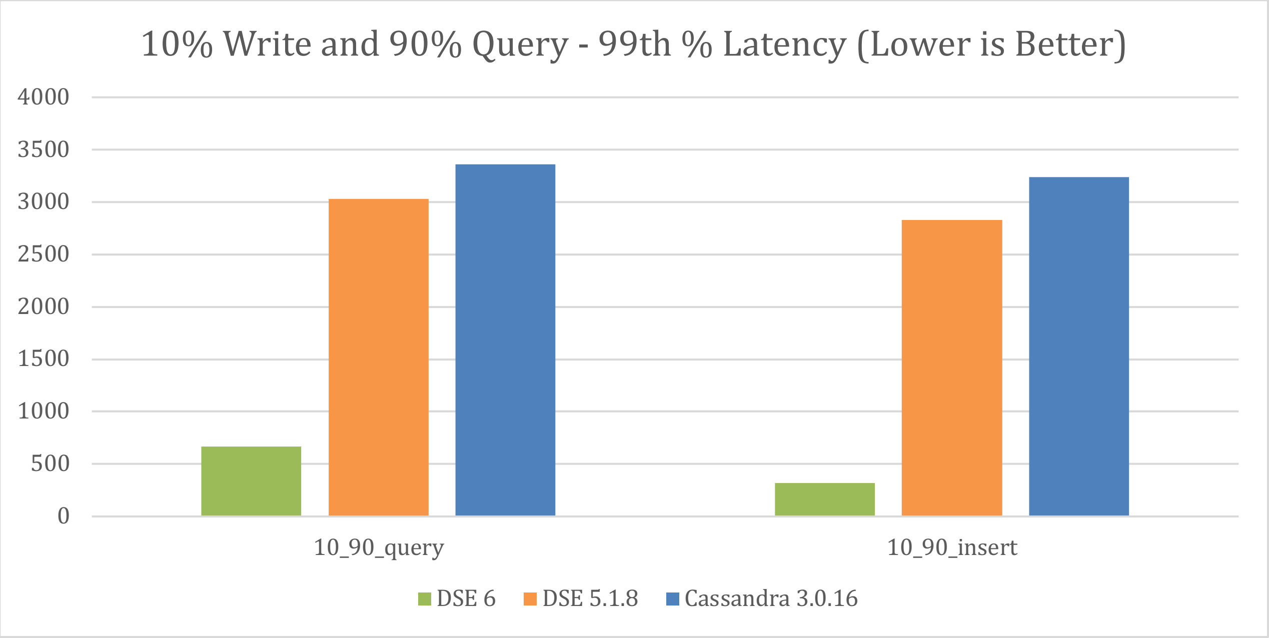
10% Write and 90% Read

Before running this test, we restored from snapshot and ran a 100% read 10-minute stress test.

```
cassandra-stress user profile=stress.yaml ops\(insert=1,query_by_machine_id=9\)
duration=1hr  cl=ONE no-warmup -mode native cql3 protocolVersion=3 -errors ignore -
rate threads=350 -pop seq=1..14000000 contents=SORTED -insert visits=fixed\(100\) -
node <host1>,<host2>,<host3> -log file=10Write_90Read_10Mpop_seq1.log
hdrfile=10Write_90Read_10Mpop_seq1.hdr -graph file=10Write_90Read_10Mpop_seq1.html
title=10Write_90Read_10Mpop_seq1
```







Conclusion

DataStax Enterprise 6 outperformed open source Cassandra and outperformed earlier versions of DataStax Enterprise. DSE 6 achieved up to 3x throughput and showed up to a 10x reduction in latency.

All tests are repeatable using the guidance provided in this document and document appendices.

Appendix A – Configuration Details

These configuration files were used for the databases, with comments and default values removed forbrevity:

- Host Configuration
  - Ubuntu 16.04 LTE
  - OS Settings:
    - See appendix C
  - Mount Info:
    - mdadm --create --verbose --force --run /dev/md1 --level=0 --raid-devices=4 /dev/nvme0n1 /dev/nvme1n1 /dev/nvme2n1 /dev/nvme3n1
      mkfs.xfs /dev/md1 -f -K
      mkdir /mnt/cass_data_disks/data1
      mount -t xfs -o noatime /dev/md1 /mnt/cass_data_disks/data1
      chmod -R 777 /mnt/cass_data_disks/data1
  - Host Config
    - /tarball_location/resources/cassandra/conf/cassandra.yaml:
      - memtable_flush_writers: 4
      - concurrent_compactors: 4
      - compaction_throughput_mb_per_sec: 0
      - hints_directory: /mnt/cass_data_disks/data1/hints
      - data_file_directories:
        - /mnt/cass_data_disks/data1/data
      - commitlog_directory: /mnt/cass_data_disks/data1/commitlog
      - saved_caches_directory: /mnt/cass_data_disks/data1/saved_caches
      - disk_access_mode: mmap_index_only
    - /tarball_location/resources/cassandra/conf/cassandra-env.sh
      - JVM_OPTS="$JVM_OPTS -XX:MaxDirectMemorySize=200g”

Appendix B – Cassandra-Stress from DSE 6 - stress.yaml template

```
### Schema Specifications ###
#
# user profile=stress.yaml ops\(insert=1\) n=10m cl=ONE no-warmup -rate threads=50
#   -insert visits=fixed\(100\) -pop seq=1..100k contents=SORTED
#
# ^ will write 100k partitions with 100 rows each in sorted order
#
```

```
# Keyspace Name
keyspace: iot_space

# The CQL for creating a keyspace (optional if it already exists)
keyspace_definition: |
  CREATE KEYSPACE iot_space WITH replication = {'class': 'SimpleStrategy', 'replication_factor': 3};
table: iot_table

# The CQL for creating a table you wish to stress (optional if it already exists)
table_definition: |
  CREATE TABLE iot_space.iot_table (
    station_id blob,
    machine_id blob,
    machine_type text,
    sensor_value double,
    time bigint,
    PRIMARY KEY (machine_id, time)
    ) WITH CLUSTERING ORDER BY (time DESC)
    AND compaction = {'class': 'org.apache.cassandra.db.compaction.TimeWindowCompactionStrategy',
    'compaction_window_unit': 'MINUTES', 'compaction_window_size': '1'};

### Column Distribution Specifications ###
columnspec:
  - name: station_id
    population: seq(1..5b)  # 5 Billion potential station_ids
    size: fixed(16)
  - name: machine_type
    size: uniform(10..20) # machine_type is 10-20 chars
    population: uniform(1..10) # there are 10 types of machine
  - name: machine_id
    size: fixed(16)
    population: seq(1..5b) # 5 Billion unique machines
  - name: sensor_value
    population: gaussian(0..1000) # sensor_values range from 0-1000 and follow a gaussian distribution
  - name: time
    cluster: fixed(100) # 100 sensor_values updates per machine
    population: seq(0..99)

### Batch Ratio Distribution Specifications ###
insert:
  partitions: fixed(1)
  select:  fixed(1)/100  # Inserts will be single row
  batchtype: UNLOGGED


# A list of queries you wish to run against the schema
#
queries:
  query_by_machine_id:
    cql: SELECT machine_id, sensor_value, time FROM iot_table WHERE machine_id = ? and time >= 90 LIMIT
10
    fields: samerow
```

Appendix C – osssetup.sh (should be run as root)

```
#!/bin/bash

#Install required/useful packages
DEBIAN_FRONTEND=noninteractive apt-get --assume-yes install libjemalloc1 numactl htop iftop iotop hwloc
libaio1

# Increase the socket listen() backlog
echo 4096 > /proc/sys/net/core/somaxconn

# Increase the maximum number of remembered connection requests, which are still
# did not receive an acknowledgment from connecting client.
echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog

# Scaling governor
if [ -e /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor ]; then
    NR_CPUS=`nproc`
    for (( i=0; i <$NR_CPUS; i++ )) do
        echo performance > /sys/devices/system/cpu/cpu${i}/cpufreq/scaling_governor
    done
else
    echo "No cpufreq"
fi

# Set clock source
echo tsc > /sys/devices/system/clocksource/clocksource0/current_clocksource

#Sysctl settings
#setup networking
echo 'net.core.wmem_max=12582912' >> /etc/sysctl.conf
echo 'net.core.rmem_max=12582912' >> /etc/sysctl.conf

#You also need to set minimum size, initial size, and maximum size in bytes:
echo 'net.ipv4.tcp_rmem= 10240 87380 12582912' >> /etc/sysctl.conf
echo 'net.ipv4.tcp_wmem= 10240 87380 12582912' >> /etc/sysctl.conf

#Turn on window scaling which can be an option to enlarge the transfer window:
echo 'net.ipv4.tcp_window_scaling = 1' >> /etc/sysctl.conf

#Enable timestamps as defined in RFC1323:
echo 'net.ipv4.tcp_timestamps = 1' >> /etc/sysctl.conf

#Enable select acknowledgments:
echo 'net.ipv4.tcp_sack = 1' >> /etc/sysctl.conf
```

```
#By default, TCP saves various connection metrics in the route cache when the connection closes, so that
connections established in the near future can use these to set initial conditions. Usually, this
#increases overall performance, but may sometimes cause performance degradation. If set, TCP will not
cache metrics on closing connections.
echo 'net.ipv4.tcp_no_metrics_save = 1' >> /etc/sysctl.conf

#Set maximum number of packets, queued on the INPUT side, when the interface receives packets faster than
kernel can process them.
echo 'net.core.netdev_max_backlog = 5000' >> /etc/sysctl.conf

# Disable slow start
echo 'net.ipv4.tcp_slow_start_after_idle = 0' >> /etc/sysctl.conf

#Enable HRTICK in scheduler
echo HRTICK > /sys/kernel/debug/sched_features

# Prevent auto-scaling from doing anything to our tunables
echo 'kernel.sched_tunable_scaling = 0' >> /etc/sysctl.conf

# Preempt sooner
echo 'kernel.sched_min_granularity_ns = 500000' >> /etc/sysctl.conf

# Don't delay unrelated workloads
echo 'kernel.sched_wakeup_granularity_ns = 500000' >> /etc/sysctl.conf

# Schedule all tasks in this period
echo 'kernel.sched_latency_ns = 1000000' >> /etc/sysctl.conf

# autogroup seems to prevent sched_latency_ns from being respected
echo 'kernel.sched_autogroup_enabled = 0' >> /etc/sysctl.conf

# Disable numa balancing
echo 'kernel.numa_balancing = 0' >> /etc/sysctl.conf

#reload
sysctl -p
```