Apache Cassandra : Miscellaneous Tools & Troubleshooting

Cassandra Stress

Cassandra Stress

IMPORTANT	The `cassandra-stress` tool is deprecated. A better tool for benchmarking Cassandra is NoSQLBench.

The `cassandra-stress` tool is used to benchmark and load-test a Cassandra cluster. `cassandra-stress` supports testing arbitrary CQL tables and queries, allowing users to benchmark their own data model.

This documentation focuses on user mode to test personal schema.

Usage

There are several operation types:

- write-only, read-only, and mixed workloads of standard data
- write-only and read-only workloads for counter columns
- user configured workloads, running custom queries on custom schemas

The syntax is `cassandra-stress <command> [options]`. For more information on a given command or options, run `cassandra-stress help <command|option>`.

Commands

read:
: Multiple concurrent reads - the cluster must first be populated by a write test

write:
: Multiple concurrent writes against the cluster

mixed:
: Interleaving of any basic commands, with configurable ratio and distribution - the cluster must first be populated by a write test

counter_write:
: Multiple concurrent updates of counters.

counter_read:
: Multiple concurrent reads of counters. The cluster must first be populated by a counterwrite test.

user:
: Interleaving of user provided queries, with configurable ratio and distribution.

help:
: Print help for a command or option

print:

Inspect the output of a distribution definition

Primary Options

-pop:

Population distribution and intra-partition visit order

-insert:

Insert specific options relating to various methods for batching and splitting partition updates

-col:

Column details such as size and count distribution, data generator, names, comparator and if super columns should be used

-rate:

Thread count, rate limit or automatic mode (default is auto)

-mode:

Additional options for authentication and connection properties. Also, "simplenative" can be selected as an alternative to the standalone Java driver

-errors:

How to handle errors when encountered during stress

-sample:

Specify the number of samples to collect for measuring latency

-schema:

Replication settings, compression, compaction, etc.

-node:

Nodes to connect to

-log:

Where to log progress to, and the interval at which to do it

-transport:

Custom transport factories

-port:

The port to connect to cassandra nodes on

-graph:

Graph recorded metrics

-tokenrange:

Token range settings

-jmx:

Username and password for JMX connection

-credentials-file <path>:

Credentials file to specify for CQL, JMX and transport

-reporting:

Frequency of printing statistics and header for stress output

Suboptions

Every command and primary option has its own collection of suboptions. These are too numerous to list here. For information on the suboptions for each command or option, please use the help command, `cassandra-stress help <command|option>`.

User mode

User mode allows you to stress your own schemas, to save you time in the long run. Find out if your application can scale using stress test with your schema.

Profile

User mode defines a profile using YAML. Multiple YAML files may be specified, in which case operations in the ops argument are referenced as specname.opname.

An identifier for the profile:

```
specname: staff_activities
```

The keyspace for the test:

```
keyspace: staff
```

CQL for the keyspace. Optional if the keyspace already exists:

```
keyspace_definition: |
 CREATE KEYSPACE stresscql WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 3};
```

The table to be stressed:

```
table: staff_activities
```

CQL for the table. Optional if the table already exists:

```
table_definition: |
  CREATE TABLE staff_activities (
      name text,
      when timeuuid,
      what text,
      PRIMARY KEY(name, when, what)
  )
```

Optional meta-information on the generated columns in the above table. The min and max only apply to text and blob types. The distribution field represents the total unique population distribution of that column across rows:

```
columnspec:
  - name: name
    size: uniform(5..10) # The names of the staff members are between 5-10 characters
    population: uniform(1..10) # 10 possible staff members to pick from
  - name: when
    cluster: uniform(20..500) # Staff members do between 20 and 500 events
  - name: what
    size: normal(10..100,50)
```

Supported types are:

An exponential distribution over the range [min..max]:

```
EXP(min..max)
```

An extreme value (Weibull) distribution over the range [min..max]:

```
EXTREME(min..max,shape)
```

A gaussian/normal distribution, where mean=(min+max)/2, and stdev is (mean-min)/stdvrng:

```
GAUSSIAN(min..max,stdvrng)
```

A gaussian/normal distribution, with explicitly defined mean and stdev:

```
GAUSSIAN(min..max,mean,stdev)
```

A uniform distribution over the range [min, max]:

```
UNIFORM(min..max)
```

A fixed distribution, always returning the same value:

```
FIXED(val)
```

If preceded by ~, the distribution is inverted

Defaults for all columns are size: uniform(4..8), population: uniform(1..100B), cluster: fixed(1)

Insert distributions:

```
insert:
  # How many partition to insert per batch
  partitions: fixed(1)
  # How many rows to update per partition
  select: fixed(1)/500
  # UNLOGGED or LOGGED batch for insert
  batchtype: UNLOGGED
```

Currently all inserts are done inside batches.

Read statements to use during the test:

```
queries:
   events:
      cql: select *  from staff_activities where name = ?
      fields: samerow
   latest_event:
      cql: select * from staff_activities where name = ?  LIMIT 1
      fields: samerow
```

Running a user mode test:

```
cassandra-stress user profile=./example.yaml duration=1m
```

```
"ops(insert=1,latest_event=1,events=1)" truncate=once
```

This will create the schema then run tests for 1 minute with an equal number of inserts, latest_event queries and events queries. Additionally the table will be truncated once before the test.

The full example can be found here:

```
spacenam: example # idenitifier for this spec if running with multiple yaml files
keyspace: example

# Would almost always be network topology unless running something locally
keyspace_definition: |
  CREATE KEYSPACE example WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 3};

table: staff_activities

# The table under test. Start with a partition per staff member
# Is this a good idea?
table_definition: |
  CREATE TABLE staff_activities (
        name text,
        when timeuuid,
        what text,
        PRIMARY KEY(name, when)
  )

columnspec:
  - name: name
    size: uniform(5..10) # The names of the staff members are between 5-10 characters
    population: uniform(1..10) # 10 possible staff members to pick from
  - name: when
    cluster: uniform(20..500) # Staff members do between 20 and 500 events
  - name: what
    size: normal(10..100,50)

insert:
  # we only update a single partition in any given insert
  partitions: fixed(1)
  # we want to insert a single row per partition and we have between 20 and 500
  # rows per partition
  select: fixed(1)/500
  batchtype: UNLOGGED             # Single partition unlogged batches are essentially
noops

queries:
```

```
   events:
      cql: select *  from staff_activities where name = ?
      fields: samerow
   latest_event:
      cql: select * from staff_activities where name = ?  LIMIT 1
      fields: samerow
```

Running a user mode test with multiple yaml files

cassandra-stress user profile=./example.yaml,./example2.yaml duration=1m "ops(ex1.insert=1,ex1.latest_event=1,ex2.insert=2)" truncate=once This will run operations as specified in both the example.yaml and example2.yaml files. example.yaml and example2.yaml can reference the same table, although care must be taken that the table definition is identical (data generation specs can be different).

Lightweight transaction support

cassandra-stress supports lightweight transactions. To use this feature, the command will first read current data from Cassandra, and then uses read values to fulfill lightweight transaction conditions.

Lightweight transaction update query:

```
queries:
  regularupdate:
      cql: update blogposts set author = ? where domain = ? and published_date = ?
      fields: samerow
  updatewithlwt:
      cql: update blogposts set author = ? where domain = ? and published_date = ? IF
body = ? AND url = ?
      fields: samerow
```

The full example can be found here:

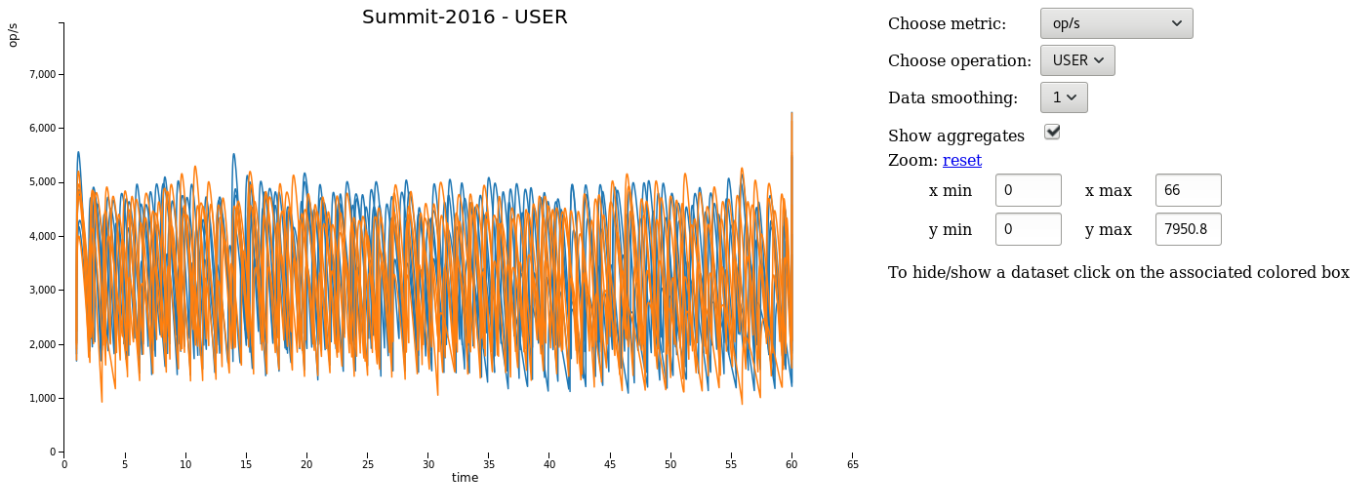
```
# Keyspace Name
keyspace: stresscql

# The CQL for creating a keyspace (optional if it already exists)
# Would almost always be network topology unless running something locall
keyspace_definition: |
  CREATE KEYSPACE stresscql WITH replication = {'class': 'SimpleStrategy',
'replication_factor': 1};

# Table name
table: blogposts
```

```
# The CQL for creating a table you wish to stress (optional if it already exists)
table_definition: |
  CREATE TABLE blogposts (
        domain text,
        published_date timeuuid,
        url text,
        author text,
        title text,
        body text,
        PRIMARY KEY(domain, published_date)
  ) WITH CLUSTERING ORDER BY (published_date DESC)
    AND compaction = { 'class':'LeveledCompactionStrategy' }
    AND comment='A table to hold blog posts'

### Column Distribution Specifications ###

columnspec:
  - name: domain
    size: gaussian(5..100)       #domain names are relatively short
    population: uniform(1..10M)  #10M possible domains to pick from

  - name: published_date
    cluster: fixed(1000)         #under each domain we will have max 1000 posts

  - name: url
    size: uniform(30..300)

  - name: title                  #titles shouldn't go beyond 200 chars
    size: gaussian(10..200)

  - name: author
    size: uniform(5..20)         #author names should be short

  - name: body
    size: gaussian(100..5000)    #the body of the blog post can be long

### Batch Ratio Distribution Specifications ###

insert:
  partitions: fixed(1)            # Our partition key is the domain so only insert one
per batch

  select:    fixed(1)/1000        # We have 1000 posts per domain so 1/1000 will allow 1
post per batch

  batchtype: UNLOGGED             # Unlogged batches
```

```
#
# A list of queries you wish to run against the schema
#
queries:
   singlepost:
      cql: select * from blogposts where domain = ? LIMIT 1
      fields: samerow
   regularupdate:
      cql: update blogposts set author = ? where domain = ? and published_date = ?
      fields: samerow
   updatewithlwt:
      cql: update blogposts set author = ? where domain = ? and published_date = ? IF
body = ? AND url = ?
      fields: samerow
```

Graphing

Graphs can be generated for each run of stress.




```
unknown
----------------------------------------
op rate                   : 4,869 op/s [events: 1,615 op/s, insert: 1,612 op/s, latest_event: 1,642 op/s]
partition rate            : 4,869 pk/s [events: 1,615 pk/s, insert: 1,612 pk/s, latest_event: 1,642 pk/s]
row rate                  : 811,056 row/s [events: 807,584 row/s, insert: 1,831 row/s, latest_event: 1,642 row/s]
latency mean              : 7.4 ms [events: 10.4 ms, insert: 5.9 ms, latest_event: 5.8 ms]
latency median            : 3.4 ms [events: 6.0 ms, insert: 2.1 ms, latest_event: 2.0 ms]
latency 95th percentile   : 27.9 ms [events: 32.4 ms, insert: 24.9 ms, latest_event: 24.5 ms]
latency 99th percentile   : 41.3 ms [events: 46.5 ms, insert: 37.8 ms, latest_event: 37.2 ms]
latency 99.9th percentile : 60.5 ms [events: 68.4 ms, insert: 53.6 ms, latest_event: 54.3 ms]
latency max               : 104.1 ms [events: 104.1 ms, insert: 100.1 ms, latest_event: 94.2 ms]
total gc count            : 0
total gc memory           : 0.000 KiB
total gc time             : 0.0 seconds
avg gc time               : NaN ms
stddev gc time            : 0.0 ms
Total operation time      : 00:01:00
cmd: user profile=./tlp-users.yaml duration=1m ops(insert=1,latest_event=1,events=1) truncate=once -graph file=tlp.html title=Summit-2016

2
----------------------------------------
op rate                   : 5,141 op/s [events: 1,723 op/s, insert: 1,699 op/s, latest_event: 1,719 op/s]
partition rate            : 5,141 pk/s [events: 1,723 pk/s, insert: 1,699 pk/s, latest_event: 1,719 pk/s]
row rate                  : 865,344 row/s [events: 861,698 row/s, insert: 1,934 row/s, latest_event: 1,719 row/s]
latency mean              : 7.0 ms [events: 9.6 ms, insert: 5.7 ms, latest_event: 5.5 ms]
latency median            : 3.1 ms [events: 5.3 ms, insert: 2.1 ms, latest_event: 1.9 ms]
latency 95th percentile   : 26.6 ms [events: 30.8 ms, insert: 23.9 ms, latest_event: 23.3 ms]
latency 99th percentile   : 40.9 ms [events: 46.0 ms, insert: 37.9 ms, latest_event: 37.1 ms]
latency 99.9th percentile : 60.0 ms [events: 66.2 ms, insert: 54.8 ms, latest_event: 55.1 ms]
latency max               : 131.0 ms [events: 107.3 ms, insert: 131.0 ms, latest_event: 119.0 ms]
total gc count            : 0
total gc memory           : 0.000 KiB
total gc time             : 0.0 seconds
avg gc time               : NaN ms
stddev gc time            : 0.0 ms
Total operation time      : 00:00:59
cmd: user profile=./tlp-users.yaml duration=1m ops(insert=1,latest_event=1,events=1) truncate=once -graph file=tlp.html title=Summit-2016 revision=2
```

To create a new graph:

```
cassandra-stress user profile=./stress-example.yaml
"ops(insert=1,latest_event=1,events=1)" -graph file=graph.html title="Awesome graph"
```

To add a new run to an existing graph point to an existing file and add a revision name:

```
cassandra-stress user profile=./stress-example.yaml duration=1m
"ops(insert=1,latest_event=1,events=1)" -graph file=graph.html title="Awesome graph"
revision="Second run"
```

FAQ

How do you use NetworkTopologyStrategy for the keyspace?

Use the schema option making sure to either escape the parenthesis or enclose in quotes:

```
cassandra-stress write -schema
"replication(strategy=NetworkTopologyStrategy,datacenter1=3)"
```

How do you use SSL?

Use the transport option:

```
cassandra-stress "write n=100k cl=ONE no-warmup" -transport
"truststore=$HOME/jks/truststore.jks truststore-password=cassandra"
```

Is Cassandra Stress a secured tool?

Cassandra stress is not a secured tool. Serialization and other aspects of the tool offer no security guarantees.

Generate tokens

Generate tokens

Pre-generates tokens for a datacenter with the given number of nodes using the token allocation algorithm. Useful in edge-cases when generated tokens needs to be known in advance of bootstrapping nodes. In nearly all cases it is best to just let the bootstrapping nodes automatically generate their own tokens. Added with: CASSANDRA-16205.

Usage

```
generatetokens -n NODES -t TOKENS \
--rf REPLICATION_FACTOR \
[--partitioner PARTITIONER] [--racks RACK_NODE_COUNTS]
```

-n,--nodes <arg>	Number of nodes.
-t,--tokens <arg>	Number of tokens/vnodes per node.
--rf <arg>	Replication factor.
-p,--partitioner <arg>	Database partitioner, either Murmur3Partitioner or RandomPartitioner.
--racks <arg>	Number of nodes per rack, separated by commas. Must add up to the total node count.

For example:

```
generatetokens -n 30 -t 8 --rf 3 --racks 10,10,10
```

will generate tokens for three racks of 10 nodes each.

This command, if used, is expected to be run before the Cassandra node is first started. The output from the command is used to configure the nodes `num_tokens` setting in the `cassandra.yaml` file.

If the following command is run:

```
tools/bin/generatetokens -n 9 -t 4 --rf 3 --racks 3,3,3
```

The output is:

```
    Generating tokens for 9 nodes with 4 vnodes each for replication factor 3 and
partitioner Murmur3Partitioner
    Node 0 rack 0: [-6270077235120413733, -1459727275878514299, 2887564907718879562,
5778609289102954400]
    Node 1 rack 1: [-8780789453057732897, -3279530982831298765, 1242905755717369197,
8125606440590916903]
    Node 2 rack 2: [-7240625170832344686, -4453155190605073029, 74749827226930055,
4615117940688406403]
    Node 3 rack 0: [-5361616212862743381, -2369629129354906532, 2065235331718124379,
6952107864846935651]
    Node 4 rack 1: [-8010707311945038792, -692488724325792122, 3751341424203642982,
7538857152718926277]
    Node 5 rack 2: [-7625666241388691739, -3866343086718185897, 5196863614895680401,
8895780530621367810]
    Node 6 rack 0: [-5815846723991578557, -1076108000102153211, 1654070543717746788,
8510693485606142356]
    Node 7 rack 1: [-2824580056093102649, 658827791472149626, 3319453165961261272,
6365358576974945025]
    Node 8 rack 2: [-4159749138661629463, -1914678202616710416, 4905990777792043402,
6658733220910940338]
```

Hash password tool

Hash password tool

The `hash_password` tool is used to get the jBcrypt hash of a password. This hash can be used in CREATE/ALTER ROLE/USER statements for improved security.

This feature can be useful if we want to make sure no intermediate system, logging or any other possible plain text password leak can happen.

Usage

hash_password <options>

-h,--help	Displays help message
-e,--environment-var <arg>	Use value of the specified environment variable as the password
-i,--input <arg>	Input is a file (or - for stdin) to read the password from. Make sure that the whole input including newlines is considered. For example, the shell command `echo -n foobar | hash_password -i -` will work as intended and just hash 'foobar'.
-p,--plain <arg>	Argument is the plain text password
-r,--logrounds <arg>	Number of hash rounds (default: 10).

One of the options --environment-var, --plain or --input must be used.

Troubleshooting

Troubleshooting

As any distributed database does, sometimes Cassandra breaks and you will have to troubleshoot what is going on. Generally speaking you can debug Cassandra like any other distributed Java program, meaning that you have to find which machines in your cluster are misbehaving and then isolate the problem using logs and tools. Luckily Cassandra had a great set of instrospection tools to help you.

These pages include a number of command examples demonstrating various debugging and analysis techniques, mostly for Linux/Unix systems. If you don't have access to the machines running Cassandra, or are running on Windows or another operating system you may not be able to use the exact commands but there are likely equivalent tools you can use.

- **Finding nodes**
- **Reading logs**
- **Using nodetool**
- **Using tools**

Find The Misbehaving Nodes

Find The Misbehaving Nodes

The first step to troubleshooting a Cassandra issue is to use error messages, metrics and monitoring information to identify if the issue lies with the clients or the server and if it does lie with the server find the problematic nodes in the Cassandra cluster. The goal is to determine if this is a systemic issue (e.g. a query pattern that affects the entire cluster) or isolated to a subset of nodes (e.g. neighbors holding a shared token range or even a single node with bad hardware).

There are many sources of information that help determine where the problem lies. Some of the most common are mentioned below.

Client Logs and Errors

Clients of the cluster often leave the best breadcrumbs to follow. Perhaps client latencies or error rates have increased in a particular datacenter (likely eliminating other datacenter's nodes), or clients are receiving a particular kind of error code indicating a particular kind of problem. Troubleshooters can often rule out many failure modes just by reading the error messages. In fact, many Cassandra error messages include the last coordinator contacted to help operators find nodes to start with.

Some common errors (likely culprit in parenthesis) assuming the client has similar error names as the Datastax `drivers <client-drivers>`:

- `SyntaxError` (**client**). This and other `QueryValidationException` indicate that the client sent a malformed request. These are rarely server issues and usually indicate bad queries.
- `UnavailableException` (**server**): This means that the Cassandra coordinator node has rejected the query as it believes that insufficent replica nodes are available. If many coordinators are throwing this error it likely means that there really are (typically) multiple nodes down in the cluster and you can identify them using `nodetool status <nodetool-status>` If only a single coordinator is throwing this error it may mean that node has been partitioned from the rest.
- `OperationTimedOutException` (**server**): This is the most frequent timeout message raised when clients set timeouts and means that the query took longer than the supplied timeout. This is a *client side* timeout meaning that it took longer than the client specified timeout. The error message will include the coordinator node that was last tried which is usually a good starting point. This error usually indicates either aggressive client timeout values or latent server coordinators/replicas.
- `ReadTimeoutException` or `WriteTimeoutException` (**server**): These are raised when clients do not specify lower timeouts and there is a *coordinator* timeouts based on the values supplied in the `cassandra.yaml` configuration file. They usually indicate a serious server side problem as the default values are usually multiple seconds.

Metrics

If you have Cassandra `metrics` reporting to a centralized location such as Graphite or Grafana you can typically use those to narrow down the problem. At this stage narrowing down the issue to a particular datacenter, rack, or even group of nodes is the main goal. Some helpful metrics to look at are:

Errors

Cassandra refers to internode messaging errors as "drops", and provided a number of `Dropped Message Metrics` to help narrow down errors. If particular nodes are dropping messages actively, they are likely related to the issue.

Latency

For timeouts or latency related issues you can start with operating/metrics.adoc#table-metrics[`table metrics`] by comparing Coordinator level metrics e.g. `CoordinatorReadLatency` or `CoordinatorWriteLatency` with their associated replica metrics e.g. `ReadLatency` or `WriteLatency`. Issues usually show up on the `99th` percentile before they show up on the `50th` percentile or the `mean`. While `maximum` coordinator latencies are not typically very helpful due to the exponentially decaying reservoir used internally to produce metrics, `maximum` replica latencies that correlate with increased `99th` percentiles on coordinators can help narrow down the problem.

There are usually three main possibilities:

1. Coordinator latencies are high on all nodes, but only a few node's local read latencies are high. This points to slow replica nodes and the coordinator's are just side-effects. This usually happens when clients are not token aware.
2. Coordinator latencies and replica latencies increase at the same time on the a few nodes. If clients are token aware this is almost always what happens and points to slow replicas of a subset of token ranges (only part of the ring).
3. Coordinator and local latencies are high on many nodes. This usually indicates either a tipping point in the cluster capacity (too many writes or reads per second), or a new query pattern.

It's important to remember that depending on the client's load balancing behavior and consistency levels coordinator and replica metrics may or may not correlate. In particular if you use `TokenAware` policies the same node's coordinator and replica latencies will often increase together, but if you just use normal `DCAwareRoundRobin` coordinator latencies can increase with unrelated replica node's latencies. For example:

- `TokenAware` + `LOCAL_ONE`: should always have coordinator and replica latencies on the same node rise together
- `TokenAware` + `LOCAL_QUORUM`: should always have coordinator and multiple replica latencies rise together in the same datacenter.

- `TokenAware` + `QUORUM`: replica latencies in other datacenters can affect coordinator latencies.
- `DCAwareRoundRobin` + `LOCAL_ONE`: coordinator latencies and unrelated replica node's latencies will rise together.
- `DCAwareRoundRobin` + `LOCAL_QUORUM`: different coordinator and replica latencies will rise together with little correlation.

Query Rates

Sometimes the **`table metric`** query rate metrics can help narrow down load issues as "small" increase in coordinator queries per second (QPS) may correlate with a very large increase in replica level QPS. This most often happens with `BATCH` writes, where a client may send a single `BATCH` query that might contain 50 statements in it, which if you have 9 copies (RF=3, three datacenters) means that every coordinator `BATCH` write turns into 450 replica writes! This is why keeping `BATCH's to the same partition is so critical, otherwise you can exhaust significant CPU capacitity with a "single" query.

Next Step: Investigate the Node(s)

Once you have narrowed down the problem as much as possible (datacenter, rack , node), login to one of the nodes using SSH and proceed to debug using **`logs`**, **`nodetool`**, and **`os tools`**. If you are not able to login you may still have access to `logs` and `nodetool` remotely.

Cassandra Logs

Cassandra Logs

Cassandra has rich support for logging and attempts to give operators maximum insight into the database while at the same time limiting noise to the logs.

Common Log Files

Cassandra has three main logs, the `system.log`, `debug.log` and `gc.log` which hold general logging messages, debugging logging messages, and java garbage collection logs respectively.

These logs by default live in `${CASSANDRA_HOME}/logs`, but most Linux distributions relocate logs to `/var/log/cassandra`. Operators can tune this location as well as what levels are logged using the provided `logback.xml` file.

`system.log`

This log is the default Cassandra log and is a good place to start any investigation. Some examples of activities logged to this log:

- Uncaught exceptions. These can be very useful for debugging errors.
- `GCInspector` messages indicating long garbage collector pauses. When long pauses happen Cassandra will print how long and also what was the state of the system (thread state) at the time of that pause. This can help narrow down a capacity issue (either not enough heap or not enough spare CPU).
- Information about nodes joining and leaving the cluster as well as token metadata (data ownersip) changes. This is useful for debugging network partitions, data movements, and more.
- Keyspace/Table creation, modification, deletion.
- `StartupChecks` that ensure optimal configuration of the operating system to run Cassandra
- Information about some background operational tasks (e.g. Index Redistribution).

As with any application, looking for `ERROR` or `WARN` lines can be a great first step:

```
$ # Search for warnings or errors in the latest system.log
$ grep 'WARN\|ERROR' system.log | tail
...

$ # Search for warnings or errors in all rotated system.log
$ zgrep 'WARN\|ERROR' system.log.* | less
...
```

debug.log

This log contains additional debugging information that may be useful when troubleshooting but may be much noiser than the normal `system.log`. Some examples of activities logged to this log:

- Information about compactions, including when they start, which sstables they contain, and when they finish.
- Information about memtable flushes to disk, including when they happened, how large the flushes were, and which commitlog segments the flush impacted.

This log can be *very* noisy, so it is highly recommended to use `grep` and other log analysis tools to dive deep. For example:

```
# Search for messages involving a CompactionTask with 5 lines of context
$ grep CompactionTask debug.log -C 5

# Look at the distribution of flush tasks per keyspace
$ grep "Enqueuing flush" debug.log | cut -f 10 -d ' ' | sort | uniq -c
    6 compaction_history:
    1 test_keyspace:
    2 local:
    17 size_estimates:
    17 sstable_activity:
```

gc.log

The gc log is a standard Java GC log. With the default `jvm.options` settings you get a lot of valuable information in this log such as application pause times, and why pauses happened. This may help narrow down throughput or latency issues to a mistuned JVM. For example you can view the last few pauses:

```
$ grep stopped gc.log.0.current | tail
2018-08-29T00:19:39.522+0000: 3022663.591: Total time for which application threads were
stopped: 0.0332813 seconds, Stopping threads took: 0.0008189 seconds
2018-08-29T00:19:44.369+0000: 3022668.438: Total time for which application threads were
stopped: 0.0312507 seconds, Stopping threads took: 0.0007025 seconds
2018-08-29T00:19:49.796+0000: 3022673.865: Total time for which application threads were
stopped: 0.0307071 seconds, Stopping threads took: 0.0006662 seconds
2018-08-29T00:19:55.452+0000: 3022679.521: Total time for which application threads were
stopped: 0.0309578 seconds, Stopping threads took: 0.0006832 seconds
2018-08-29T00:20:00.127+0000: 3022684.197: Total time for which application threads were
stopped: 0.0310082 seconds, Stopping threads took: 0.0007090 seconds
2018-08-29T00:20:06.583+0000: 3022690.653: Total time for which application threads were
stopped: 0.0317346 seconds, Stopping threads took: 0.0007106 seconds
2018-08-29T00:20:10.079+0000: 3022694.148: Total time for which application threads were
```

```
stopped: 0.0299036 seconds, Stopping threads took: 0.0006889 seconds
2018-08-29T00:20:15.739+0000: 3022699.809: Total time for which application threads were
stopped: 0.0078283 seconds, Stopping threads took: 0.0006012 seconds
2018-08-29T00:20:15.770+0000: 3022699.839: Total time for which application threads were
stopped: 0.0301285 seconds, Stopping threads took: 0.0003789 seconds
2018-08-29T00:20:15.798+0000: 3022699.867: Total time for which application threads were
stopped: 0.0279407 seconds, Stopping threads took: 0.0003627 seconds
```

This shows a lot of valuable information including how long the application was paused (meaning zero user queries were being serviced during the e.g. 33ms JVM pause) as well as how long it took to enter the safepoint. You can use this raw data to e.g. get the longest pauses:

```
$ grep stopped gc.log.0.current | cut -f 11 -d ' ' | sort -n  | tail | xargs -IX grep X
gc.log.0.current | sort -k 1
2018-08-28T17:13:40.520-0700: 1.193: Total time for which application threads were
stopped: 0.0157914 seconds, Stopping threads took: 0.0000355 seconds
2018-08-28T17:13:41.206-0700: 1.879: Total time for which application threads were
stopped: 0.0249811 seconds, Stopping threads took: 0.0000318 seconds
2018-08-28T17:13:41.638-0700: 2.311: Total time for which application threads were
stopped: 0.0561130 seconds, Stopping threads took: 0.0000328 seconds
2018-08-28T17:13:41.677-0700: 2.350: Total time for which application threads were
stopped: 0.0362129 seconds, Stopping threads took: 0.0000597 seconds
2018-08-28T17:13:41.781-0700: 2.454: Total time for which application threads were
stopped: 0.0442846 seconds, Stopping threads took: 0.0000238 seconds
2018-08-28T17:13:41.976-0700: 2.649: Total time for which application threads were
stopped: 0.0377115 seconds, Stopping threads took: 0.0000250 seconds
2018-08-28T17:13:42.172-0700: 2.845: Total time for which application threads were
stopped: 0.0475415 seconds, Stopping threads took: 0.0001018 seconds
2018-08-28T17:13:42.825-0700: 3.498: Total time for which application threads were
stopped: 0.0379155 seconds, Stopping threads took: 0.0000571 seconds
2018-08-28T17:13:43.574-0700: 4.247: Total time for which application threads were
stopped: 0.0323812 seconds, Stopping threads took: 0.0000574 seconds
2018-08-28T17:13:44.602-0700: 5.275: Total time for which application threads were
stopped: 0.0238975 seconds, Stopping threads took: 0.0000788 seconds
```

In this case any client waiting on a query would have experienced a 56ms latency at 17:13:41.

Note that GC pauses are not _*only* garbage collection, although generally speaking high pauses with fast safepoints indicate a lack of JVM heap or mistuned JVM GC algorithm. High pauses with slow safepoints typically indicate that the JVM is having trouble entering a safepoint which usually indicates slow disk drives (Cassandra makes heavy use of memory mapped reads which the JVM doesn't know could have disk latency, so the JVM safepoint logic doesn't handle a blocking memory mapped read particularly well).

Using these logs you can even get a pause distribution with something like histogram.py:

```
$ grep stopped gc.log.0.current | cut -f 11 -d ' ' | sort -n | histogram.py
# NumSamples = 410293; Min = 0.00; Max = 11.49
# Mean = 0.035346; Variance = 0.002216; SD = 0.047078; Median 0.036498
# each ∎ represents a count of 5470
    0.0001 -     1.1496 [410255]: ∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎∎
    1.1496 -     2.2991 [    15]:
    2.2991 -     3.4486 [     5]:
    3.4486 -     4.5981 [     1]:
    4.5981 -     5.7475 [     5]:
    5.7475 -     6.8970 [     9]:
    6.8970 -     8.0465 [     1]:
    8.0465 -     9.1960 [     0]:
    9.1960 -    10.3455 [     0]:
   10.3455 -    11.4949 [     2]:
```

We can see in this case while we have very good average performance something is causing multi second JVM pauses … In this case it was mostly safepoint pauses caused by slow disks:

```
$ grep stopped gc.log.0.current | cut -f 11 -d ' ' | sort -n | tail | xargs -IX grep X
gc.log.0.current| sort -k 1
2018-07-27T04:52:27.413+0000: 187831.482: Total time for which application threads were
stopped: 6.5037022 seconds, Stopping threads took: 0.0005212 seconds
2018-07-30T23:38:18.354+0000: 514582.423: Total time for which application threads were
stopped: 6.3262938 seconds, Stopping threads took: 0.0004882 seconds
2018-08-01T02:37:48.380+0000: 611752.450: Total time for which application threads were
stopped: 10.3879659 seconds, Stopping threads took: 0.0004475 seconds
2018-08-06T22:04:14.990+0000: 1113739.059: Total time for which application threads were
stopped: 6.0917409 seconds, Stopping threads took: 0.0005553 seconds
2018-08-14T00:04:06.091+0000: 1725730.160: Total time for which application threads were
stopped: 6.0141054 seconds, Stopping threads took: 0.0004976 seconds
2018-08-17T06:23:06.755+0000: 2007670.824: Total time for which application threads were
stopped: 6.0133694 seconds, Stopping threads took: 0.0006011 seconds
2018-08-23T06:35:46.068+0000: 2526830.137: Total time for which application threads were
stopped: 6.4767751 seconds, Stopping threads took: 6.4426849 seconds
2018-08-23T06:36:29.018+0000: 2526873.087: Total time for which application threads were
stopped: 11.4949489 seconds, Stopping threads took: 11.4638297 seconds
2018-08-23T06:37:12.671+0000: 2526916.741: Total time for which application threads were
stopped: 6.3867003 seconds, Stopping threads took: 6.3507166 seconds
2018-08-23T06:37:47.156+0000: 2526951.225: Total time for which application threads were
stopped: 7.9528200 seconds, Stopping threads took: 7.9197756 seconds
```

Sometimes reading and understanding java GC logs is hard, but you can take the raw GC files and visualize them using tools such as GCViewer which take the Cassandra GC log as input and show you detailed visual information on your garbage collection performance. This includes pause analysis as well as throughput information. For a stable Cassandra JVM you probably want to aim for pauses less

than 200ms and GC throughput greater than 99%.

Java GC pauses are one of the leading causes of tail latency in Cassandra (along with drive latency) so sometimes this information can be crucial while debugging tail latency issues.

Getting More Information

If the default logging levels are insuficient, `nodetool` can set higher or lower logging levels for various packages and classes using the `nodetool setlogginglevel` command. Start by viewing the current levels:

```
$ nodetool getlogginglevels

Logger Name                                        Log Level
ROOT                                                    INFO
org.apache.cassandra                                   DEBUG
```

Perhaps the `Gossiper` is acting up and we wish to enable it at `TRACE` level for even more insight:

```
$ nodetool setlogginglevel org.apache.cassandra.gms.Gossiper TRACE

$ nodetool getlogginglevels

Logger Name                                        Log Level
ROOT                                                    INFO
org.apache.cassandra                                   DEBUG
org.apache.cassandra.gms.Gossiper                      TRACE

$ grep TRACE debug.log | tail -2
TRACE [GossipStage:1] 2018-07-04 17:07:47,879 Gossiper.java:1234 - Updating
heartbeat state version to 2344 from 2343 for 127.0.0.2:7000 ...
TRACE [GossipStage:1] 2018-07-04 17:07:47,879 Gossiper.java:923 - local
heartbeat version 2341 greater than 2340 for 127.0.0.1:7000
```

Note that any changes made this way are reverted on next Cassandra process restart. To make the changes permanent add the appropriate rule to `logback.xml`.

```
diff --git a/conf/logback.xml b/conf/logback.xml
index b2c5b10..71b0a49 100644
--- a/conf/logback.xml
+++ b/conf/logback.xml
@@ -98,4 +98,5 @@ appender reference in the root level section below.
   </root>

   <logger name="org.apache.cassandra" level="DEBUG"/>
```

```
+  <logger name="org.apache.cassandra.gms.Gossiper" level="TRACE"/>
 </configuration>
```

Note that if you want more information than this tool provides, there are other live capture options available such as `packet-capture`.

Use Nodetool

Use Nodetool

Cassandra's `nodetool` allows you to narrow problems from the cluster down to a particular node and gives a lot of insight into the state of the Cassandra process itself. There are dozens of useful commands (see `nodetool help` for all the commands), but briefly some of the most useful for troubleshooting:

Cluster Status

You can use `nodetool status` to assess status of the cluster:

```
$ nodetool status <optional keyspace>

Datacenter: dc1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens       Owns (effective)  Host ID
Rack
UN  127.0.1.1  4.69 GiB   1            100.0%            35ea8c9f-b7a2-40a7-b9c5-
0ee8b91fdd0e  r1
UN  127.0.1.2  4.71 GiB   1            100.0%            752e278f-b7c5-4f58-974b-
9328455af73f  r2
UN  127.0.1.3  4.69 GiB   1            100.0%            9dc1a293-2cc0-40fa-a6fd-
9e6054da04a7  r3
```

In this case we can see that we have three nodes in one datacenter with about 4.6GB of data each and they are all "up". The up/down status of a node is independently determined by every node in the cluster, so you may have to run `nodetool status` on multiple nodes in a cluster to see the full view.

You can use `nodetool status` plus a little grep to see which nodes are down:

```
$ nodetool status | grep -v '^UN'
Datacenter: dc1
===============
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens       Owns (effective)  Host ID
Rack
Datacenter: dc2
===============
Status=Up/Down
```

```
|/ State=Normal/Leaving/Joining/Moving
--  Address    Load       Tokens       Owns (effective)  Host ID
Rack
DN  127.0.0.5  105.73 KiB  1            33.3%             df303ac7-61de-46e9-ac79-
6e630115fd75  r1
```

In this case there are two datacenters and there is one node down in datacenter `dc2` and rack `r1`. This may indicate an issue on `127.0.0.5` warranting investigation.

Coordinator Query Latency

You can view latency distributions of coordinator read and write latency to help narrow down latency issues using `nodetool proxyhistograms`:

```
$ nodetool proxyhistograms
Percentile       Read Latency      Write Latency      Range Latency   CAS Read Latency
CAS Write Latency View Write Latency
                     (micros)          (micros)          (micros)           (micros)
(micros)           (micros)
50%                   454.83            219.34              0.00               0.00
0.00               0.00
75%                   545.79            263.21              0.00               0.00
0.00               0.00
95%                   654.95            315.85              0.00               0.00
0.00               0.00
98%                   785.94            379.02              0.00               0.00
0.00               0.00
99%                  3379.39           2346.80              0.00               0.00
0.00               0.00
Min                    42.51            105.78              0.00               0.00
0.00               0.00
Max                 25109.16          43388.63              0.00               0.00
0.00               0.00
```

Here you can see the full latency distribution of reads, writes, range requests (e.g. `select * from keyspace.table`), CAS read (compare phase of CAS) and CAS write (set phase of compare and set). These can be useful for narrowing down high level latency problems, for example in this case if a client had a 20 millisecond timeout on their reads they might experience the occasional timeout from this node but less than 1% (since the 99% read latency is 3.3 milliseconds < 20 milliseconds).

Local Query Latency

If you know which table is having latency/error issues, you can use `nodetool tablehistograms` to get a

better idea of what is happening locally on a node:

```
$ nodetool tablehistograms keyspace table
Percentile  SSTables     Write Latency      Read Latency    Partition Size        Cell
Count
                              (micros)          (micros)           (bytes)
50%             0.00             73.46            182.79             17084
103
75%             1.00             88.15            315.85             17084
103
95%             2.00            126.93            545.79             17084
103
98%             2.00            152.32            654.95             17084
103
99%             2.00            182.79            785.94             17084
103
Min             0.00             42.51             24.60             14238
87
Max             2.00          12108.97          17436.92             17084
103
```

This shows you percentile breakdowns particularly critical metrics.

The first column contains how many sstables were read per logical read. A very high number here indicates that you may have chosen the wrong compaction strategy, e.g. `SizeTieredCompactionStrategy` typically has many more reads per read than `LeveledCompactionStrategy` does for update heavy workloads.

The second column shows you a latency breakdown of *local* write latency. In this case we see that while the p50 is quite good at 73 microseconds, the maximum latency is quite slow at 12 milliseconds. High write max latencies often indicate a slow commitlog volume (slow to fsync) or large writes that quickly saturate commitlog segments.

The third column shows you a latency breakdown of *local* read latency. We can see that local Cassandra reads are (as expected) slower than local writes, and the read speed correlates highly with the number of sstables read per read.

The fourth and fifth columns show distributions of partition size and column count per partition. These are useful for determining if the table has on average skinny or wide partitions and can help you isolate bad data patterns. For example if you have a single cell that is 2 megabytes, that is probably going to cause some heap pressure when it's read.

Threadpool State

You can use `nodetool tpstats` to view the current outstanding requests on a particular node. This is

useful for trying to find out which resource (read threads, write threads, compaction, request response threads) the Cassandra process lacks. For example:

```
$ nodetool tpstats
Pool Name                         Active   Pending      Completed   Blocked  All time
blocked
ReadStage                              2         0             12         0
0
MiscStage                              0         0              0         0
0
CompactionExecutor                     0         0           1940         0
0
MutationStage                          0         0              0         0
0
GossipStage                            0         0          10293         0
0
Repair-Task                            0         0              0         0
0
RequestResponseStage                   0         0             16         0
0
ReadRepairStage                        0         0              0         0
0
CounterMutationStage                   0         0              0         0
0
MemtablePostFlush                      0         0             83         0
0
ValidationExecutor                     0         0              0         0
0
MemtableFlushWriter                    0         0             30         0
0
ViewMutationStage                      0         0              0         0
0
CacheCleanupExecutor                   0         0              0         0
0
MemtableReclaimMemory                  0         0             30         0
0
PendingRangeCalculator                 0         0             11         0
0
SecondaryIndexManagement               0         0              0         0
0
HintsDispatcher                        0         0              0         0
0
Native-Transport-Requests              0         0            192         0
0
MigrationStage                         0         0             14         0
0
PerDiskMemtableFlushWriter_0           0         0             30         0
```

```
0
Sampler                             0         0              0         0
0
ViewBuildExecutor                   0         0              0         0
0
InternalResponseStage               0         0              0         0
0
AntiEntropyStage                    0         0              0         0
0

Message type           Dropped                  Latency waiting in queue (micros)
                                             50%               95%               99%
Max
READ                         0               N/A               N/A               N/A
N/A
RANGE_SLICE                  0              0.00              0.00              0.00
0.00
_TRACE                       0               N/A               N/A               N/A
N/A
HINT                         0               N/A               N/A               N/A
N/A
MUTATION                     0               N/A               N/A               N/A
N/A
COUNTER_MUTATION             0               N/A               N/A               N/A
N/A
BATCH_STORE                  0               N/A               N/A               N/A
N/A
BATCH_REMOVE                 0               N/A               N/A               N/A
N/A
REQUEST_RESPONSE             0              0.00              0.00              0.00
0.00
PAGED_RANGE                  0               N/A               N/A               N/A
N/A
READ_REPAIR                  0               N/A               N/A               N/A
N/A
```

This command shows you all kinds of interesting statistics. The first section shows a detailed breakdown of threadpools for each Cassandra stage, including how many threads are current executing (Active) and how many are waiting to run (Pending). Typically if you see pending executions in a particular threadpool that indicates a problem localized to that type of operation. For example if the `RequestResponseState` queue is backing up, that means that the coordinators are waiting on a lot of downstream replica requests and may indicate a lack of token awareness, or very high consistency levels being used on read requests (for example reading at `ALL` ties up RF `RequestResponseState` threads whereas `LOCAL_ONE` only uses a single thread in the `ReadStage` threadpool). On the other hand if you see a lot of pending compactions that may indicate that your compaction threads cannot keep up with the volume of writes and you may need to tune either the compaction strategy or the

`concurrent_compactors` or `compaction_throughput` options.

The second section shows drops (errors) and latency distributions for all the major request types. Drops are cumulative since process start, but if you have any that indicate a serious problem as the default timeouts to qualify as a drop are quite high (~5-10 seconds). Dropped messages often warrants further investigation.

Compaction State

As Cassandra is a LSM datastore, Cassandra sometimes has to compact sstables together, which can have adverse effects on performance. In particular, compaction uses a reasonable quantity of CPU resources, invalidates large quantities of the OS page cache, and can put a lot of load on your disk drives. There are great `os tools <os-iostat>` to determine if this is the case, but often it's a good idea to check if compactions are even running using `nodetool compactionstats`:

```
$ nodetool compactionstats
pending tasks: 2
- keyspace.table: 2

id                                   compaction type keyspace table completed total
unit  progress
2062b290-7f3a-11e8-9358-cd941b956e60 Compaction      keyspace table 21848273  97867583
bytes 22.32%
Active compaction remaining time :   0h00m04s
```

In this case there is a single compaction running on the `keyspace.table` table, has completed 21.8 megabytes of 97 and Cassandra estimates (based on the configured compaction throughput) that this will take 4 seconds. You can also pass `-H` to get the units in a human readable format.

Generally each running compaction can consume a single core, but the more you do in parallel the faster data compacts. Compaction is crucial to ensuring good read performance so having the right balance of concurrent compactions such that compactions complete quickly but don't take too many resources away from query threads is very important for performance. If you notice compaction unable to keep up, try tuning Cassandra's `concurrent_compactors` or `compaction_throughput` options.

Paths used for data files

Cassandra is persisting data on disk within the configured directories. Data files are distributed among the directories configured with `data_file_directories`. Resembling the structure of keyspaces and tables, Cassandra is creating subdirectories within `data_file_directories`. However, directories aren't removed even if the tables and keyspaces are dropped. While these directories are kept with the reason of holding snapshots, they are subject to removal. This is where operators need to know which directories are still in use. Running the `nodetool datapaths` command is an easy way to list in which

directories Cassandra is actually storing sstable data on disk.

```
% nodetool datapaths -- system_auth
Keyspace: system_auth
    Table: role_permissions
    Paths:
        /var/lib/cassandra/data/system_auth/role_permissions-
3afbe79f219431a7add7f5ab90d8ec9c

    Table: network_permissions
    Paths:
        /var/lib/cassandra/data/system_auth/network_permissions-
d46780c22f1c3db9b4c1b8d9fbc0cc23

    Table: resource_role_permissons_index
    Paths:
        /var/lib/cassandra/data/system_auth/resource_role_permissons_index-
5f2fbdad91f13946bd25d5da3a5c35ec

    Table: roles
    Paths:
        /var/lib/cassandra/data/system_auth/roles-5bc52802de2535edaeab188eecebb090

    Table: role_members
    Paths:
        /var/lib/cassandra/data/system_auth/role_members-0ecdaa87f8fb3e6088d174fb36fe5c0d
```

By default all keyspaces and tables are listed, however, a list of `keyspace` and `keyspace.table` arguments can be given to query specific data paths. Using the `--format` option the output can be formatted as YAML or JSON.

Diving Deep, Use External Tools

Diving Deep, Use External Tools

Machine access allows operators to dive even deeper than logs and `nodetool` allow. While every Cassandra operator may have their personal favorite toolsets for troubleshooting issues, this page contains some of the most common operator techniques and examples of those tools. Many of these commands work only on Linux, but if you are deploying on a different operating system you may have access to other substantially similar tools that assess similar OS level metrics and processes.

JVM Tooling

The JVM ships with a number of useful tools. Some of them are useful for debugging Cassandra issues, especially related to heap and execution stacks.

NOTE: There are two common gotchas with JVM tooling and Cassandra:

1. By default Cassandra ships with `-XX:+PerfDisableSharedMem` set to prevent long pauses (see `CASSANDRA-9242` and `CASSANDRA-9483` for details). If you want to use JVM tooling you can instead have `/tmp` mounted on an in memory `tmpfs` which also effectively works around `CASSANDRA-9242`.
2. Make sure you run the tools as the same user as Cassandra is running as, e.g. if the database is running as `cassandra` the tool also has to be run as `cassandra`, e.g. via `sudo -u cassandra <cmd>`.

Garbage Collection State (jstat)

If you suspect heap pressure you can use `jstat` to dive deep into the garbage collection state of a Cassandra process. This command is always safe to run and yields detailed heap information including eden heap usage (E), old generation heap usage (O), count of eden collections (YGC), time spend in eden collections (YGCT), old/mixed generation collections (FGC) and time spent in old/mixed generation collections (FGCT):

```
jstat -gcutil <cassandra pid> 500ms
 S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
 0.00   0.00  81.53  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  82.36  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  82.36  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  83.19  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  83.19  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  84.19  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  84.19  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  85.03  31.16  93.07  88.20     12    0.151     3    0.257    0.408
```

```
 0.00   0.00  85.03  31.16  93.07  88.20     12    0.151     3    0.257    0.408
 0.00   0.00  85.94  31.16  93.07  88.20     12    0.151     3    0.257    0.408
```

In this case we see we have a relatively healthy heap profile, with 31.16% old generation heap usage and 83% eden. If the old generation routinely is above 75% then you probably need more heap (assuming CMS with a 75% occupancy threshold). If you do have such persistently high old gen that often means you either have under-provisioned the old generation heap, or that there is too much live data on heap for Cassandra to collect (e.g. because of memtables). Another thing to watch for is time between young garbage collections (YGC), which indicate how frequently the eden heap is collected. Each young gc pause is about 20-50ms, so if you have a lot of them your clients will notice in their high percentile latencies.

Thread Information (jstack)

To get a point in time snapshot of exactly what Cassandra is doing, run `jstack` against the Cassandra PID. **Note** that this does pause the JVM for a very brief period (<20ms).:

```
$ jstack <cassandra pid> > threaddump

# display the threaddump
$ cat threaddump

# look at runnable threads
$grep RUNNABLE threaddump -B 1
"Attach Listener" #15 daemon prio=9 os_prio=0 tid=0x00007f829c001000 nid=0x3a74 waiting
on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
--
"DestroyJavaVM" #13 prio=5 os_prio=0 tid=0x00007f82e800e000 nid=0x2a19 waiting on
condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
--
"JPS thread pool" #10 prio=5 os_prio=0 tid=0x00007f82e84d0800 nid=0x2a2c runnable
[0x00007f82d0856000]
   java.lang.Thread.State: RUNNABLE
--
"Service Thread" #9 daemon prio=9 os_prio=0 tid=0x00007f82e80d7000 nid=0x2a2a runnable
[0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
--
"C1 CompilerThread3" #8 daemon prio=9 os_prio=0 tid=0x00007f82e80cc000 nid=0x2a29 waiting
on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE
--
```

```
# Note that the nid is the Linux thread id
```

Some of the most important information in the threaddumps are waiting/blocking threads, including what locks or monitors the thread is blocking/waiting on.

Basic OS Tooling

A great place to start when debugging a Cassandra issue is understanding how Cassandra is interacting with system resources. The following are all resources that Cassandra makes heavy uses of:

- CPU cores. For executing concurrent user queries
- CPU processing time. For query activity (data decompression, row merging, etc.)
- CPU processing time (low priority). For background tasks (compaction, streaming, etc ...)
- RAM for Java Heap. Used to hold internal data-structures and by default the Cassandra memtables. Heap space is a crucial component of write performance as well as generally.
- RAM for OS disk cache. Used to cache frequently accessed SSTable blocks. OS disk cache is a crucial component of read performance.
- Disks. Cassandra cares a lot about disk read latency, disk write throughput, and of course disk space.
- Network latency. Cassandra makes many internode requests, so network latency between nodes can directly impact performance.
- Network throughput. Cassandra (as other databases) frequently have the so called "incast" problem where a small request (e.g. `SELECT * from foo.bar`) returns a massively large result set (e.g. the entire dataset). In such situations outgoing bandwidth is crucial.

Often troubleshooting Cassandra comes down to troubleshooting what resource the machine or cluster is running out of. Then you create more of that resource or change the query pattern to make less use of that resource.

High Level Resource Usage (top/htop)

Cassandra makes signifiant use of system resources, and often the very first useful action is to run `top` or `htop` (website)to see the state of the machine.

Useful things to look at:

- System load levels. While these numbers can be confusing, generally speaking if the load average is greater than the number of CPU cores, Cassandra probably won't have very good (sub 100 millisecond) latencies. See Linux Load Averages for more information.
- CPU utilization. `htop` in particular can help break down CPU utilization into `user` (low and normal priority), `system` (kernel), and `io-wait` . Cassandra query threads execute as normal priority `user`

threads, while compaction threads execute as low priority `user` threads. High `system` time could indicate problems like thread contention, and high `io-wait` may indicate slow disk drives. This can help you understand what Cassandra is spending processing resources doing.

- Memory usage. Look for which programs have the most resident memory, it is probably Cassandra. The number for Cassandra is likely inaccurately high due to how Linux (as of 2018) accounts for memory mapped file memory.

IO Usage (iostat)

Use iostat to determine how data drives are faring, including latency distributions, throughput, and utilization:

```
$ sudo iostat -xdm 2
Linux 4.13.0-13-generic (hostname)     07/03/2018     _x86_64_    (8 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.28    0.32    5.42     0.01     0.13    48.55     0.01
2.21    0.26    2.32   0.64   0.37
sdb               0.00     0.00    0.00    0.00     0.00     0.00    79.34     0.00
0.20    0.20    0.00   0.16   0.00
sdc               0.34     0.27    0.76    0.36     0.01     0.02    47.56     0.03
26.90    2.98   77.73   9.21   1.03

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz
await r_await w_await  svctm  %util
sda               0.00     0.00    2.00   32.00     0.01     4.04   244.24     0.54
16.00    0.00   17.00   1.06   3.60
sdb               0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00
0.00    0.00    0.00   0.00   0.00
sdc               0.00    24.50    0.00  114.00     0.00    11.62   208.70     5.56
48.79    0.00   48.79   1.12  12.80
```

In this case we can see that `/dev/sdc1` is a very slow drive, having an `await` close to 50 milliseconds and an `avgqu-sz` close to 5 ios. The drive is not particularly saturated (utilization is only 12.8%), but we should still be concerned about how this would affect our p99 latency since 50ms is quite long for typical Cassandra operations. That being said, in this case most of the latency is present in writes (typically writes are more latent than reads), which due to the LSM nature of Cassandra is often hidden from the user.

Important metrics to assess using iostat:

- Reads and writes per second. These numbers will change with the workload, but generally speaking the more reads Cassandra has to do from disk the slower Cassandra read latencies are. Large numbers of reads per second can be a dead giveaway that the cluster has insufficient

memory for OS page caching.

- Write throughput. Cassandra's LSM model defers user writes and batches them together, which means that throughput to the underlying medium is the most important write metric for Cassandra.
- Read latency (`r_await`). When Cassandra missed the OS page cache and reads from SSTables, the read latency directly determines how fast Cassandra can respond with the data.
- Write latency. Cassandra is less sensitive to write latency except when it syncs the commit log. This typically enters into the very high percentiles of write latency.

Note that to get detailed latency breakdowns you will need a more advanced tool such as **`bcc-tools`**.

OS page Cache Usage

As Cassandra makes heavy use of memory mapped files, the health of the operating system's Page Cache is crucial to performance. Start by finding how much available cache is in the system:

```
$ free -g
              total        used        free      shared  buff/cache   available
Mem:             15           9           2           0           3           5
Swap:             0           0           0
```

In this case 9GB of memory is used by user processes (Cassandra heap) and 8GB is available for OS page cache. Of that, 3GB is actually used to cache files. If most memory is used and unavailable to the page cache, Cassandra performance can suffer significantly. This is why Cassandra starts with a reasonably small amount of memory reserved for the heap.

If you suspect that you are missing the OS page cache frequently you can use advanced tools like **cachestat** or **vmtouch** to dive deeper.

Network Latency and Reliability

Whenever Cassandra does writes or reads that involve other replicas, `LOCAL_QUORUM` reads for example, one of the dominant effects on latency is network latency. When trying to debug issues with multi machine operations, the network can be an important resource to investigate. You can determine internode latency using tools like `ping` and `traceroute` or most effectively `mtr`:

```
$ mtr -nr www.google.com
Start: Sun Jul 22 13:10:28 2018
HOST: hostname                     Loss%   Snt   Last   Avg  Best  Wrst StDev
  1.|-- 192.168.1.1                0.0%    10    2.0   1.9   1.1   3.7   0.7
  2.|-- 96.123.29.15               0.0%    10   11.4  11.0   9.0  16.4   1.9
  3.|-- 68.86.249.21               0.0%    10   10.6  10.7   9.0  13.7   1.1
  4.|-- 162.141.78.129             0.0%    10   11.5  10.6   9.6  12.4   0.7
```

```
  5.|-- 162.151.78.253             0.0%    10   10.9  12.1  10.4  20.2   2.8
  6.|-- 68.86.143.93               0.0%    10   12.4  12.6   9.9  23.1   3.8
  7.|-- 96.112.146.18              0.0%    10   11.9  12.4  10.6  15.5   1.6
  9.|-- 209.85.252.250             0.0%    10   13.7  13.2  12.5  13.9   0.0
 10.|-- 108.170.242.238            0.0%    10   12.7  12.4  11.1  13.0   0.5
 11.|-- 74.125.253.149             0.0%    10   13.4  13.7  11.8  19.2   2.1
 12.|-- 216.239.62.40              0.0%    10   13.4  14.7  11.5  26.9   4.6
 13.|-- 108.170.242.81             0.0%    10   14.4  13.2  10.9  16.0   1.7
 14.|-- 72.14.239.43               0.0%    10   12.2  16.1  11.0  32.8   7.1
 15.|-- 216.58.195.68              0.0%    10   25.1  15.3  11.1  25.1   4.8
```

In this example of `mtr`, we can rapidly assess the path that your packets are taking, as well as what their typical loss and latency are. Packet loss typically leads to between `200ms` and `3s` of additional latency, so that can be a common cause of latency issues.

Network Throughput

As Cassandra is sensitive to outgoing bandwidth limitations, sometimes it is useful to determine if network throughput is limited. One handy tool to do this is iftop which shows both bandwidth usage as well as connection information at a glance. An example showing traffic during a stress run against a local `ccm` cluster:

```
$ # remove the -t for ncurses instead of pure text
$ sudo iftop -nNtP -i lo
interface: lo
IP address is: 127.0.0.1
MAC address is: 00:00:00:00:00:00
Listening on lo
   # Host name (port/service if enabled)            last 2s   last 10s   last 40s
cumulative
--------------------------------------------------------------------------------------------
---
   1 127.0.0.1:58946                          =>      869Kb      869Kb      869Kb
217KB
     127.0.0.3:9042                           <=         0b         0b         0b
0B
   2 127.0.0.1:54654                          =>      736Kb      736Kb      736Kb
184KB
     127.0.0.1:9042                           <=         0b         0b         0b
0B
   3 127.0.0.1:51186                          =>      669Kb      669Kb      669Kb
167KB
     127.0.0.2:9042                           <=         0b         0b         0b
0B
   4 127.0.0.3:9042                           =>     3.30Kb     3.30Kb     3.30Kb
```

```
845B
     127.0.0.1:58946                          <=         0b         0b         0b
0B
   5 127.0.0.1:9042                           =>     2.79Kb     2.79Kb     2.79Kb
715B
     127.0.0.1:54654                          <=         0b         0b         0b
0B
   6 127.0.0.2:9042                           =>     2.54Kb     2.54Kb     2.54Kb
650B
     127.0.0.1:51186                          <=         0b         0b         0b
0B
   7 127.0.0.1:36894                          =>     1.65Kb     1.65Kb     1.65Kb
423B
     127.0.0.5:7000                           <=         0b         0b         0b
0B
   8 127.0.0.1:38034                          =>     1.50Kb     1.50Kb     1.50Kb
385B
     127.0.0.2:7000                           <=         0b         0b         0b
0B
   9 127.0.0.1:56324                          =>     1.50Kb     1.50Kb     1.50Kb
383B
     127.0.0.1:7000                           <=         0b         0b         0b
0B
  10 127.0.0.1:53044                          =>     1.43Kb     1.43Kb     1.43Kb
366B
     127.0.0.4:7000                           <=         0b         0b         0b
0B
------------------------------------------------------------------------------
---
Total send rate:                                    2.25Mb     2.25Mb     2.25Mb
Total receive rate:                                    0b         0b         0b
Total send and receive rate:                        2.25Mb     2.25Mb     2.25Mb
------------------------------------------------------------------------------
---
Peak rate (sent/received/total):                    2.25Mb         0b     2.25Mb
Cumulative (sent/received/total):                    576KB         0B      576KB
==============================================================================
===
```

In this case we can see that bandwidth is fairly shared between many peers, but if the total was getting close to the rated capacity of the NIC or was focussed on a single client, that may indicate a clue as to what issue is occurring.

Advanced tools

Sometimes as an operator you may need to really dive deep. This is where advanced OS tooling can

come in handy.

bcc-tools

Most modern Linux distributions (kernels newer than `4.1`) support bcc-tools for diving deep into performance problems. First install `bcc-tools`, e.g. via `apt` on Debian:

```
$ apt install bcc-tools
```

Then you can use all the tools that `bcc-tools` contains. One of the most useful tools is `cachestat` (cachestat examples) which allows you to determine exactly how many OS page cache hits and misses are happening:

```
$ sudo /usr/share/bcc/tools/cachestat -T 1
TIME        TOTAL   MISSES     HITS  DIRTIES   BUFFERS_MB  CACHED_MB
18:44:08       66       66        0       64           88       4427
18:44:09       40       40        0       75           88       4427
18:44:10     4353       45     4308      203           88       4427
18:44:11       84       77        7       13           88       4428
18:44:12     2511       14     2497       14           88       4428
18:44:13      101       98        3       18           88       4428
18:44:14    16741        0    16741       58           88       4428
18:44:15     1935       36     1899       18           88       4428
18:44:16       89       34       55       18           88       4428
```

In this case there are not too many page cache `MISSES` which indicates a reasonably sized cache. These metrics are the most direct measurement of your Cassandra node's "hot" dataset. If you don't have enough cache, `MISSES` will be high and performance will be slow. If you have enough cache, `MISSES` will be low and performance will be fast (as almost all reads are being served out of memory).

You can also measure disk latency distributions using `biolatency` (biolatency examples) to get an idea of how slow Cassandra will be when reads miss the OS page Cache and have to hit disks:

```
$ sudo /usr/share/bcc/tools/biolatency -D 10
Tracing block device I/O... Hit Ctrl-C to end.


disk = 'sda'
     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 12       |****************************************|
```

```
        32 -> 63         : 9        |********************************        |
        64 -> 127        : 1        |***                                     |
       128 -> 255        : 3        |**********                              |
       256 -> 511        : 7        |************************                |
       512 -> 1023       : 2        |******                                  |

disk = 'sdc'
     usecs               : count     distribution
         0 -> 1          : 0        |                                        |
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 41       |************                            |
       128 -> 255        : 17       |*****                                   |
       256 -> 511        : 13       |***                                     |
       512 -> 1023       : 2        |                                        |
      1024 -> 2047       : 0        |                                        |
      2048 -> 4095       : 0        |                                        |
      4096 -> 8191       : 56       |******************                      |
      8192 -> 16383      : 131      |****************************************|
     16384 -> 32767      : 9        |**                                      |
```

In this case most ios on the data drive (`sdc`) are fast, but many take between 8 and 16 milliseconds.

Finally `biosnoop` (examples) can be used to dive even deeper and see per IO latencies:

```
$ sudo /usr/share/bcc/tools/biosnoop | grep java | head
0.000000000    java           17427  sdc     R  3972458600 4096      13.58
0.000818000    java           17427  sdc     R  3972459408 4096       0.35
0.007098000    java           17416  sdc     R  3972401824 4096       5.81
0.007896000    java           17416  sdc     R  3972489960 4096       0.34
0.008920000    java           17416  sdc     R  3972489896 4096       0.34
0.009487000    java           17427  sdc     R  3972401880 4096       0.32
0.010238000    java           17416  sdc     R  3972488368 4096       0.37
0.010596000    java           17427  sdc     R  3972488376 4096       0.34
0.011236000    java           17410  sdc     R  3972488424 4096       0.32
0.011825000    java           17427  sdc     R  3972488576 16384      0.65
... time passes
8.032687000    java           18279  sdc     R  10899712  122880     3.01
8.033175000    java           18279  sdc     R  10899952  8192       0.46
8.073295000    java           18279  sdc     R  23384320  122880     3.01
8.073768000    java           18279  sdc     R  23384560  8192       0.46
```

With `biosnoop` you see every single IO and how long they take. This data can be used to construct the

latency distributions in `biolatency` but can also be used to better understand how disk latency affects performance. For example this particular drive takes ~3ms to service a memory mapped read due to the large default value (`128kb`) of `read_ahead_kb`. To improve point read performance you may may want to decrease `read_ahead_kb` on fast data volumes such as SSDs while keeping the a higher value like `128kb` value is probably right for HDs. There are tradeoffs involved, see queue-sysfs docs for more information, but regardless `biosnoop` is useful for understanding *how* Cassandra uses drives.

vmtouch

Sometimes it's useful to know how much of the Cassandra data files are being cached by the OS. A great tool for answering this question is vmtouch.

First install it:

```
$ git clone https://github.com/hoytech/vmtouch.git
$ cd vmtouch
$ make
```

Then run it on the Cassandra data directory:

```
$ ./vmtouch /var/lib/cassandra/data/
           Files: 312
     Directories: 92
  Resident Pages: 62503/64308  244M/251M  97.2%
         Elapsed: 0.005657 seconds
```

In this case almost the entire dataset is hot in OS page Cache. Generally speaking the percentage doesn't really matter unless reads are missing the cache (per e.g. **cachestat** in which case having additional memory may help read performance.

CPU Flamegraphs

Cassandra often uses a lot of CPU, but telling *what* it is doing can prove difficult. One of the best ways to analyze Cassandra on CPU time is to use CPU Flamegraphs which display in a useful way which areas of Cassandra code are using CPU. This may help narrow down a compaction problem to a "compaction problem dropping tombstones" or just generally help you narrow down what Cassandra is doing while it is having an issue. To get CPU flamegraphs follow the instructions for Java Flamegraphs.

Generally:

1. Enable the `-XX:+PreserveFramePointer` option in Cassandra's `jvm.options` configuation file. This has a negligible performance impact but allows you actually see what Cassandra is doing.

2. Run `perf` to get some data.
3. Send that data through the relevant scripts in the FlameGraph toolset and convert the data into a pretty flamegraph. View the resulting SVG image in a browser or other image browser.

For example just cloning straight off github we first install the `perf-map-agent` to the location of our JVMs (assumed to be `/usr/lib/jvm`):

```
$ sudo bash
$ export JAVA_HOME=/usr/lib/jvm/java-8-oracle/
$ cd /usr/lib/jvm
$ git clone --depth=1 https://github.com/jvm-profiling-tools/perf-map-agent
$ cd perf-map-agent
$ cmake .
$ make
```

Now to get a flamegraph:

```
$ git clone --depth=1 https://github.com/brendangregg/FlameGraph
$ sudo bash
$ cd FlameGraph
$ # Record traces of Cassandra and map symbols for all java processes
$ perf record -F 49 -a -g -p <CASSANDRA PID> -- sleep 30; ./jmaps
$ # Translate the data
$ perf script > cassandra_stacks
$ cat cassandra_stacks | ./stackcollapse-perf.pl | grep -v cpu_idle | \
    ./flamegraph.pl --color=java --hash > cassandra_flames.svg
```

The resulting SVG is searchable, zoomable, and generally easy to introspect using a browser.

Packet Capture

Sometimes you have to understand what queries a Cassandra node is performing *right now* to troubleshoot an issue. For these times trusty packet capture tools like `tcpdump` and Wireshark can be very helpful to dissect packet captures. Wireshark even has native CQL support although it sometimes has compatibility issues with newer Cassandra protocol releases.

To get a packet capture first capture some packets:

```
$ sudo tcpdump -U -s0 -i <INTERFACE> -w cassandra.pcap -n "tcp port 9042"
```

Now open it up with wireshark:

```
$ wireshark cassandra.pcap
```

If you don't see CQL like statements try telling to decode as CQL by right clicking on a packet going to 9042 → `Decode as` → select CQL from the dropdown for port 9042.

If you don't want to do this manually or use a GUI, you can also use something like cqltrace to ease obtaining and parsing CQL packet captures.

Third-Party Plugins

Third-Party Plugins

Available third-party plugins for Apache Cassandra

CAPI-Rowcache

The Coherent Accelerator Process Interface (CAPI) is a general term for the infrastructure of attaching a Coherent accelerator to an IBM POWER system. A key innovation in IBM POWER8's open architecture is the CAPI. It provides a high bandwidth, low latency path between external devices, the POWER8 core, and the system's open memory architecture. IBM Data Engine for NoSQL is an integrated platform for large and fast growing NoSQL data stores. It builds on the CAPI capability of POWER8 systems and provides super-fast access to large flash storage capacity and addresses the challenges associated with typical x86 server based scale-out deployments.

The official page for the CAPI-Rowcache plugin contains further details how to build/run/download the plugin.

Stratio's Cassandra Lucene Index

Stratio's Lucene index is a Cassandra secondary index implementation based on Apache Lucene. It extends Cassandra's functionality to provide near real-time distributed search engine capabilities such as with ElasticSearch or Apache Solr, including full text search capabilities, free multivariable, geospatial and bitemporal search, relevance queries and sorting based on column value, relevance or distance. Each node indexes its own data, so high availability and scalability is guaranteed.

The official Github repository Cassandra Lucene Index contains everything you need to build/run/configure the plugin.