Apache Cassandra:
Core Concepts, Skills, and Tools

Working with the Cassandra read path
Exercise Workbook

Joe Chu
Leo Schuman
October, 2014

Exercise 1: Working with Bloom filters

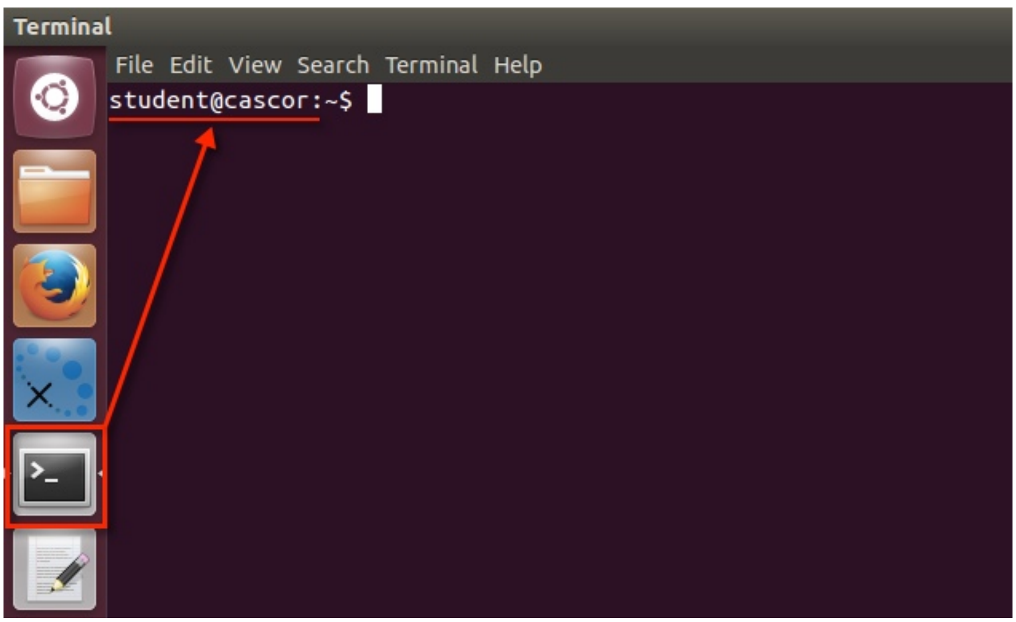
In this exercise, you will:

- Examine the factors that determines Bloom Filter size
- Look up statistics for the Bloom Filter

Steps

Examine the factors that determine Bloom Filter size

1. In the virtual machine, open a Terminal window or switch to an existing Terminal running the Linux shell.



2. From the Linux shell, navigate to the *data* directory for the keyspace *musicdb* and the table *album*.

```
cd ~/node1/data/musicdb/album-[table id]
```

3. In the data directory for the *album* table, list the files. Take note of the size of the *-filter.db* files.

```
ls -lh *Filter.db
```

```
student@cascor:~/node1/data/musicdb/album-b95732706f6f11e48cfc6b9a4e3f8a18$ ls -lh *Filter.db
-rw-rw-r-- 1 student student 4.0K Nov 18 14:12 musicdb-album-ka-1-Filter.db
student@cascor:~/node1/data/musicdb/album-b95732706f6f11e48cfc6b9a4e3f8a18$
```

The -Filter.db file is the Bloom Filter generated for each SSTable.

4. Navigate to the data directory for the Open another Terminal window.

5. In the new Terminal window, start up *cqlsh*.

```
ccm node1 cqlsh
```

6. In *cqlsh*, set *musicdb* as the default keyspace.

```
USE musicdb;
```

7. From *the musicdb* keyspace, DESCRIBE the table *album*.

```
DESCRIBE TABLE album
```

```
    tracks map<int, text>,
    PRIMARY KEY ((title, year))
) WITH bloom_filter_fp_chance = 0.01
    AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
    AND comment = ''
```

This shows that the bloom_filter_fp_chance setting is set to 0.01, or a 1% chance of encountering a false positive. This is the default value for a CQL table.

8. In *musicdb*, ALTER the *album* table to set *bloom_filter_fp_chance* to 0.0001.

```
ALTER TABLE musicdb.album
WITH bloom_filter_fp_chance = 0.0001;
```

9. Switch back to the other Terminal window, and execute the *nodetool upgradesstables* command for the *album* table in the *musicdb* keyspace.

```
ccm node1 nodetool "upgradesstables --include-all-sstables
musicdb album"
```

nodetool upgradesstables rebuilds SSTables for a specified keyspace and table. We are doing this here so that the Bloom Filters are also rebuilt. Normally this command will only upgrade sstables not at the most recent SSTable version; the --include-all-sstables *flag is needed to force the rebuild to occur.*

Normally you would need to run nodetool upgradesstables on each node. For the purposes of this exercise, running it on only one node is sufficient.

10. In the *album* data directory, list out the files again.

```
ls -lh *Filter.db
```

With the new bloom_filter_fp_chance setting, the size of the Bloom Filters have gotten much larger, in exchange for having a lower probability of a false positive.

11. In the virtual machine, switch back to the other Terminal window running *cqlsh*.

12. In the *musicdb* keyspace, ALTER the *album* table to change *bloom_filter_fp_chance* to 1.0.

```
ALTER TABLE musicdb.album
WITH bloom_filter_fp_chance = 1.0;
```

13. Switch to the other Terminal window and execute the *nodetool upgradesstables* command for the *album* table in *musicdb.*

```
ccm node1 nodetool "upgradesstables --include-all-sstables
musicdb album"
```

14. In the *album* data directory, list out the files again.

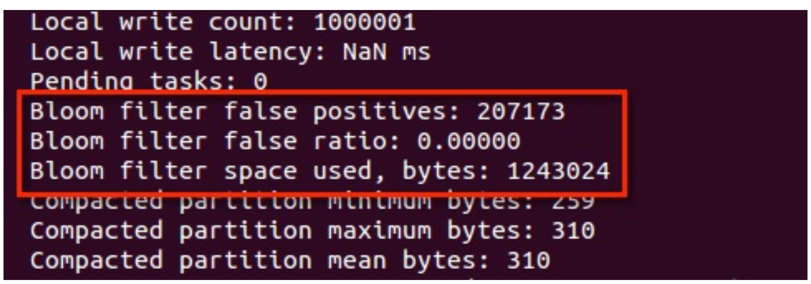
```
ls -lh *Filter.db
```

Since the bloom_filter_fp_chance setting is set 1.0, this effectively disables the use of Bloom Filters. Did you find any -Filter.db files?

Look up statistics for the Bloom Filter

15. In the terminal window, run *nodetool cfstats* for the *album* table on node1.

```
ccm node1 nodetool cfstats musicdb.album
```



Your results will vary based on other activity. Nodetool cfstats *show statistics about the Bloom Filter used for the album table including the number of false positives, the ratio of false positives to successful hits, and the total size of the bloom filter for the table.*

There are other statistics shown in cfstats, which will be described in a later section.

END OF EXERCISE

Exercise 2: Working with the key cache

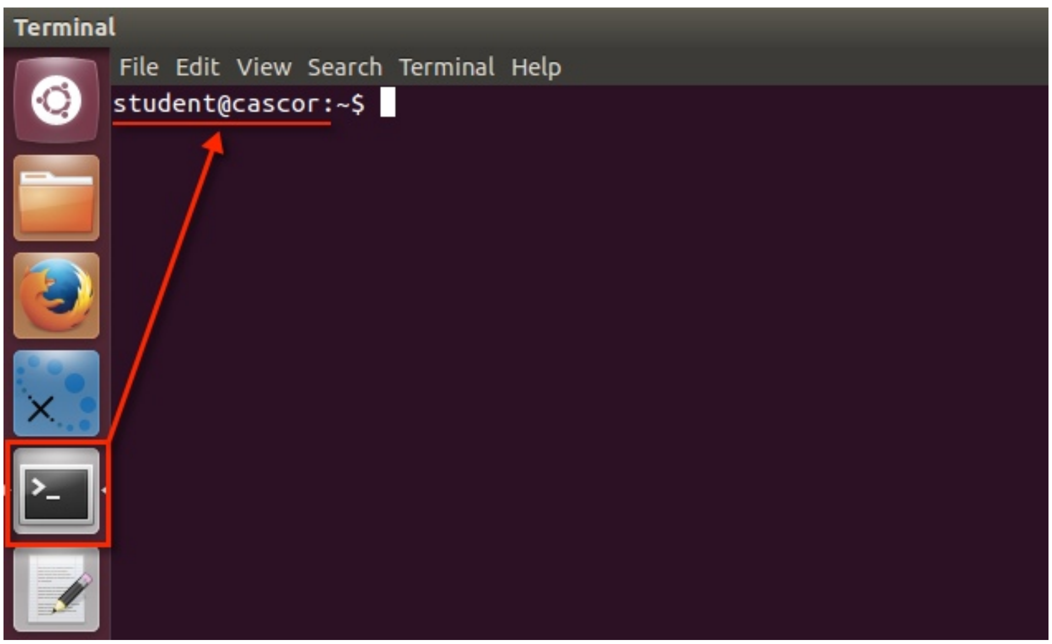
In this exercise, you will:

- Test for performance improvements using the key cache
- Examine the saved cache feature

Steps

Examine the benefits of using the key cache

1. In the virtual machine, open a Terminal window or switch to an existing Terminal running the Linux shell.



2. In the Terminal window, start up *cqlsh*.

```
ccm node1 cqlsh
```

3. In *cqlsh*, use the DESCRIBE command to look at the table properties for the performer table.

```
DESCRIBE TABLE musicdb.performer;
```

```
CREATE TABLE musicdb.performer (
    name text PRIMARY KEY,
    born int,
    country text,
    died int,
    founded int,
    style text,
    type text
) WITH bloom_filter_fp_chance = 0.01
    AND caching = '{"keys":"ALL", "rows_per_partition":"NONE"}'
    AND comment = ''
```

Here we can see that the key cache is enabled by default. It can only be turned on or off.

4. Exit from *cqlsh*.

```
EXIT
```

5. From the Linux shell, navigate to the *cascor/read-path/exercise-2* directory.

```
cd ~/cascor/read-path/exercise-2
```

6. In the *exercise-2* directory, run *cassandra-stress* using the *cqlstress.yaml* profile to perform 20,000 insert operations using 1 client thread with no warmup.

```
cassandra-stress user profile=cqlstress.yaml
ops\(insert=1\) no-warmup n=20000 -rate threads=1
```

7. After inserting the keys, run the CCM command for *nodetool flush*. This will flush any data remaining in the MemTables to disk.

```
ccm flush
```

8. Use *CCM* to set *key_cache_save_period* in the *cassandra.yaml* file to 120.

```
ccm updateconf 'key_cache_save_period: 120'
```

9. Stop and start the *CCM* cluster again.

```
ccm stop
ccm start
```

10. Drop the Linux page cache to clear memory caches.

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

11. Run *nodetool info* on node1 to check the key cache statistics for the *performer* table.

```
ccm node1 nodetool info
```

```
student@cascor:~$ ccm node1 nodetool info
ID               : 711e31d7-3b2c-4b02-ba51-7cdc926314a7
Gossip active    : true
Thrift active    : true
Native Transport active: true
Load             : 8.28 MB
Generation No    : 1416383012
Uptime (seconds) : 53
Heap Memory (MB) : 88.44 / 490.00
Data Center      : datacenter1
Rack             : rack1
Exceptions       : 0
Key Cache        : entries 33, size 2.73 KB, capacity 24 MB, 17 hits, 50 requests, 0.340 recent hit rate, 120 save period in seconds
Row Cache        : entries 0, size 0 bytes, capacity 0 bytes, 0 hits, 0 requests, NaN recent hit rate, 0 save period in seconds
Counter Cache    : entries 0, size 0 bytes, capacity 12 MB, 0 hits, 0 requests, NaN recent hit rate, 7200 save period in seconds
Token            : (invoke with -T/--tokens to see all 3 tokens)
```

Here we confirm that there are very few entries in the key cache after restarting the cluster. Also confirm that the save period has been changed.

12. In the *exercise-2* directory, run *cassandra-stress* using the *cqlstress.yaml* profile to perform 20,000 of the *simple1* query with 1 client thread and no warmup.

```
cassandra-stress user profile=cqlstress.yaml
ops\(simple1=1\) no-warmup n=20000 -rate threads=1
```

Make a note of the total operation time when cassandra-stress *completes.*

13. Run *nodetool info* on node1 again to check the key cache statistics for the *performer* table.

```
ccm node1 nodetool info
```

There should be changes to the key cache now, with significantly more entries now. Also make note of the recent hit rate.

14. Drop the Linux page cache again.

```
echo 3 | sudo tee /proc/sys/vm/drop_caches
```

15. In the *exercise-2* directory, run *cassandra-stress* using the *cqlstress.yaml* profile to perform 20,000 of the *simple1* query using 1 client thread and no warmup.

```
cassandra-stress user profile=cqlstress.yaml
ops\(simple1=1\) no-warmup n=20000 -rate threads=1
```

How does the total operation time for this cassandra-stress run compare to the previous? Was there a change in the read latency?

16. Run *nodetool info* on node1 again to check the key cache statistics for the *performer* table.

```
ccm node1 nodetool info
```

How does the recent hit rate for the key cache compare with the previous display?

Examine the saved cache feature

17. Navigate to the saved_cache directory for node1 and list the files there.

```
cd ~/node1/saved_caches
ls
```

```
student@cascor:~/cascor/read-path/exercise-2$ cd ~/node1/saved_caches/
student@cascor:~/node1/saved_caches$ ls
musicdb-performer-6ed41dc06fbb11e4a5b26b9a4e3f8a18-KeyCache-b.db
system-compaction_history-b4dbb7b4dc493fb5b3bfce6e434832ca-KeyCache-b.db
system-local-7ad54392bcdd35a684174e047860b377-KeyCache-b.db
system-peers-37f71aca7dc2383ba70672528af04d4f-KeyCache-b.db
system-schema_columnfamilies-45f5b36024bc3f83a3631034ea4fa697-KeyCache-b.db
system-schema_columns-296e9c049bec3085827dc17d3df2122a-KeyCache-b.db
system-sstable_activity-5a1ff267ace03f128563cfae6103c65e-KeyCache-b.db
```

At this time there should be a file saved for the musicdb-performer key cache.

18. Stop and start the cascor cluster again.

```
ccm stop
ccm start
```

19. Run *nodetool info* on node1 again to check the key cache statistics for the *performer* table.

```
ccm node1 nodetool info
```

Here we should still see about the same number of entries for the key cache, rather than starting again from 0.

END OF EXERCISE

Demo 3: Use the CLI to examine data storage

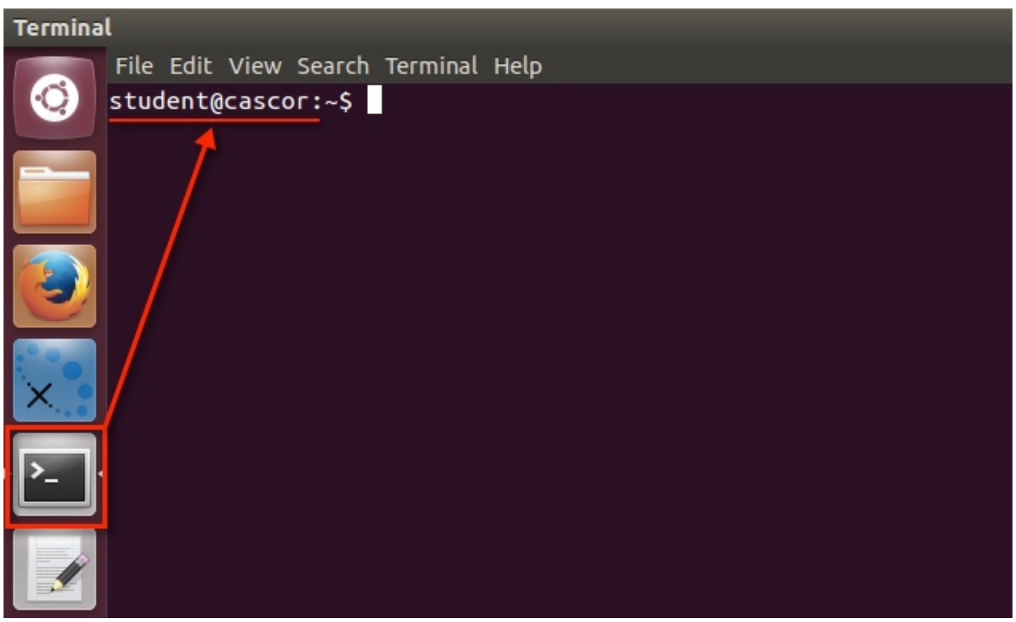
In this exercise, you will:

- View how data is stored for a table with a simple primary key
- View how data is stored for a table with a compound primary key
- View how data is stored for a table with a composite partition key

Steps

View how data is stored for a table with a simple primary key

1. In the virtual machine, open a Terminal window or switch to an existing Terminal running the Linux shell.



2. In the Terminal window, start up *cassandra-cli* using the CCM shortcut.

```
ccm node1 cli
```

3. In *cassandra-cli*, switch to the keyspace *musicdb*.

```
use musicdb;
```

4. In the *musicdb* keyspace, list 10 partitions from the *performer* table.

```
list performer limit 10;
```

```
=> (name=country, value=556e6974656420537461746573, timestamp=1398966778578000)
=> (name=style, value=556e6b6e6f776e, timestamp=1398966778578000)
=> (name=type, value=62616e64, timestamp=1398966778578000)
-------------------
RowKey: Lords of Acid
=> (name=, value=, timestamp=1398966778579000)
=> (name=country, value=42656c6769756d, timestamp=1398966778579000)
=> (name=style, value=556e6b6e6f776e, timestamp=1398966778579000)
=> (name=type, value=62616e64, timestamp=1398966778579000)

10 Rows Returned.
Elapsed time: 183 msec(s).
[default@musicdb]
```

The section for each RowKey, or partition key, represents a partition and each line would be a cell. The first cell has an empty cell name and value, but contains a timestamp used for that partition key. Other cells correspond to the column name and values we see in CQL.

You may also see some partitions that look like they have garbage data, which was automatically generated when running cassandra-stress *to write to the* performer *table.*

View how data is stored for a table with a compound primary key

5. In the *musicdb* keyspace, list 3 partitions from the *performers_by_style* table.

```
list performers_by_style limit 3;
```

```
-------------------
RowKey: New Wave
=> (name=Altered Images:, value=, timestamp=1398966783447002)
=> (name=Animotion:, value=, timestamp=1398966783447003)
=> (name=Arcadia:, value=, timestamp=1398966783448000)
=> (name=Baltimora:, value=, timestamp=1398966783448001)
=> (name=Berlin:, value=, timestamp=1398966783448002)
=> (name=Boy George:, value=, timestamp=1398966783448003)
=> (name=Culture Club:, value=, timestamp=1398966783448004)
=> (name=Naked Eyes:, value=, timestamp=1398966783449000)
=> (name=Nena:, value=, timestamp=1398966783449001)
=> (name=Re-Flex:, value=, timestamp=1398966783449002)
=> (name=The Blow Monkeys:, value=, timestamp=1398966783449003)
=> (name=The Boomtown Rats:, value=, timestamp=1398966783449004)
=> (name=The Call:, value=, timestamp=1398966783450000)
=> (name=The Waitresses:, value=, timestamp=1398966783450001)
-------------------
RowKey: Trance
=> (name=BT:, value=, timestamp=1398966783450002)
```

Here we see again that the partition key (style) is the same as the RowKey. With the clustering column (name), the value is saved in the cell name, and the cell values are not used.

6. In the *musicdb* keyspace, list 3 partitions from the *albums_by_genre* table.

```
list albums_by_genre limit 3;
```

```
-------------------
RowKey: Funk
=> (name=Blaze:2000:Blaze:, value=, timestamp=1398966790318003)
=> (name=Chaka Khan:1989:Life Is A Dance (The Remix project):, value=, timestamp=1398966790319000)
=> (name=James Vincent:1976:Space Traveler:, value=, timestamp=1398966790319001)
=> (name=James Vincent:1974:Culmination:, value=, timestamp=1398966790319002)
-------------------
RowKey: New Wave
=> (name=Boy George:2002:U Never Can B 2 Straight:, value=, timestamp=1398966790319003)
=> (name=Boy George:1998:Sold:, value=, timestamp=1398966790319004)
=> (name=Boy George:1989:Boyfriend:, value=, timestamp=1398966790320000)
-------------------
RowKey: Soundtrack
=> (name=Elton John:1999:Elton John and Tim Rice's Aida:, value=, timestamp=1398966790320001)

10 Rows Returned.
Elapsed time: 144 msec(s).
[default@musicdb]
```

With multiple clustering columns, the values are concatenated together in the cell name.

View how data is stored for a table with a composite partition key

7. In the *musicdb* keyspace, list 5 partitions from the *album* table.

```
list album limit 5;
```

```
-------------------
RowKey: Merry Christmas From Elvis Presley, A:1982
=> (name=, value=, timestamp=1398966786048000)
=> (name=genre, value=526f636b, timestamp=1398966786048000)
=> (name=performer, value=456c76697320507265736c6579, timestamp=1398966786048000)
=> (name=tracks:00000001, value=4f20436f6d6520416c6c205965204661697468660756c, timestamp=1398966786048000)
=> (name=tracks:00000002, value=5468652046697273742004e6f656c, timestamp=1398966786048000)
=> (name=tracks:00000003, value=4f6e204120536e6f77792043687269737446d6173204e69676874, timestamp=1398966786048000)
=> (name=tracks:00000004, value=57696e74657220576f6e6465726c616e64, timestamp=1398966786048000)
=> (name=tracks:00000005, value=54686520576f6e64657266756c20576f726c64204f66204368726973746d6173, timestamp=13989667860 4
8000)
```

The composite partition key is made up of album and year, which is shown together in the RowKey.

8. Use the *quit* command to close the CLI.

END OF DEMO

Exercise 4: Read data and examine its tracing output

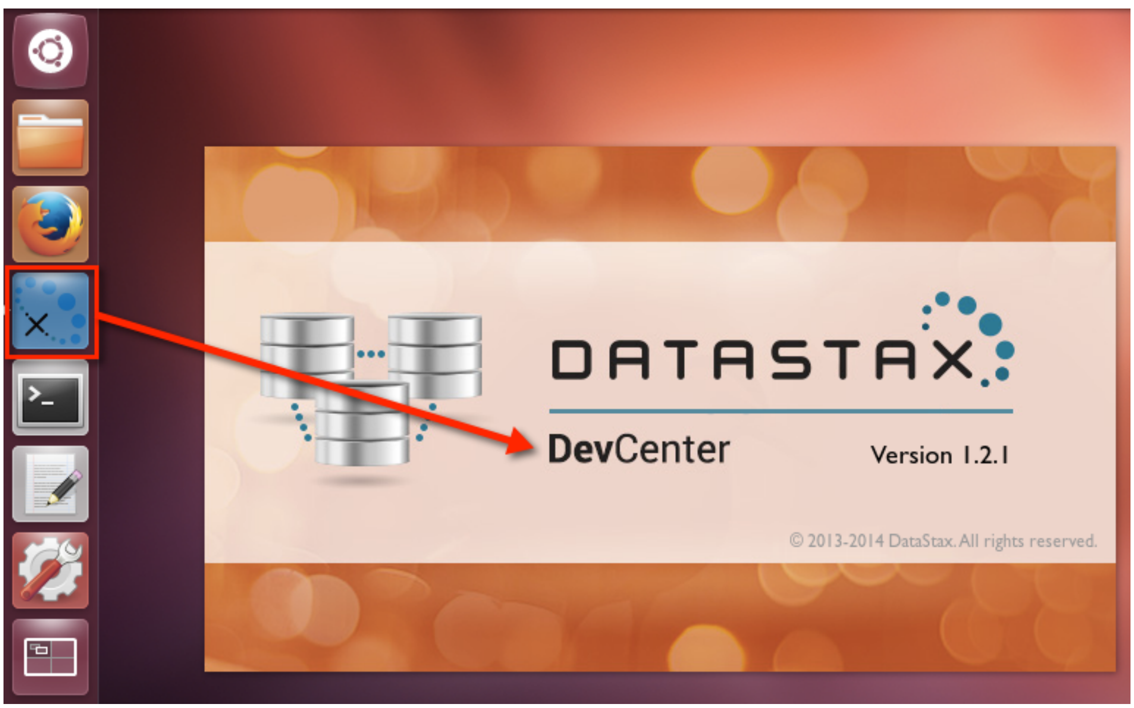
In this exercise, you will:

- Look over the traces when doing an INSERT and SELECT
- Study the trace for a query that cannot fulfill CL requirements
- Explore the trace for a COUNT aggregate query

Steps

Look over the traces when doing an INSERT and SELECT

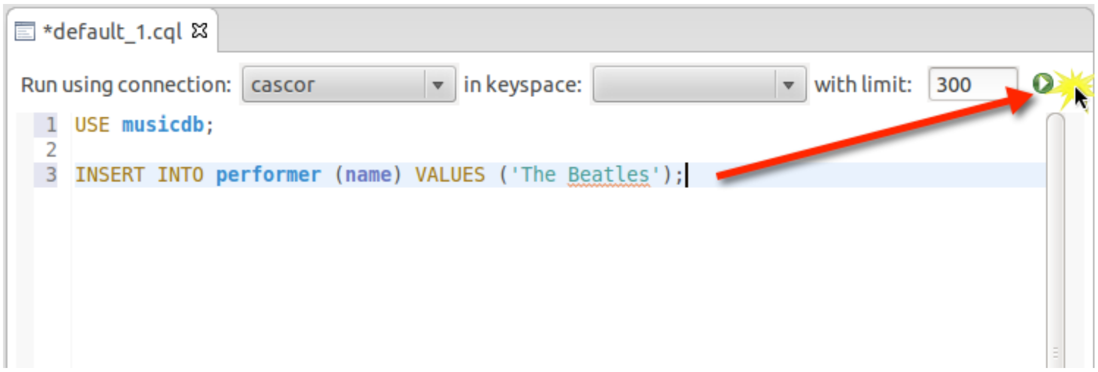
1. In the virtual machine, start up *DevCenter*.



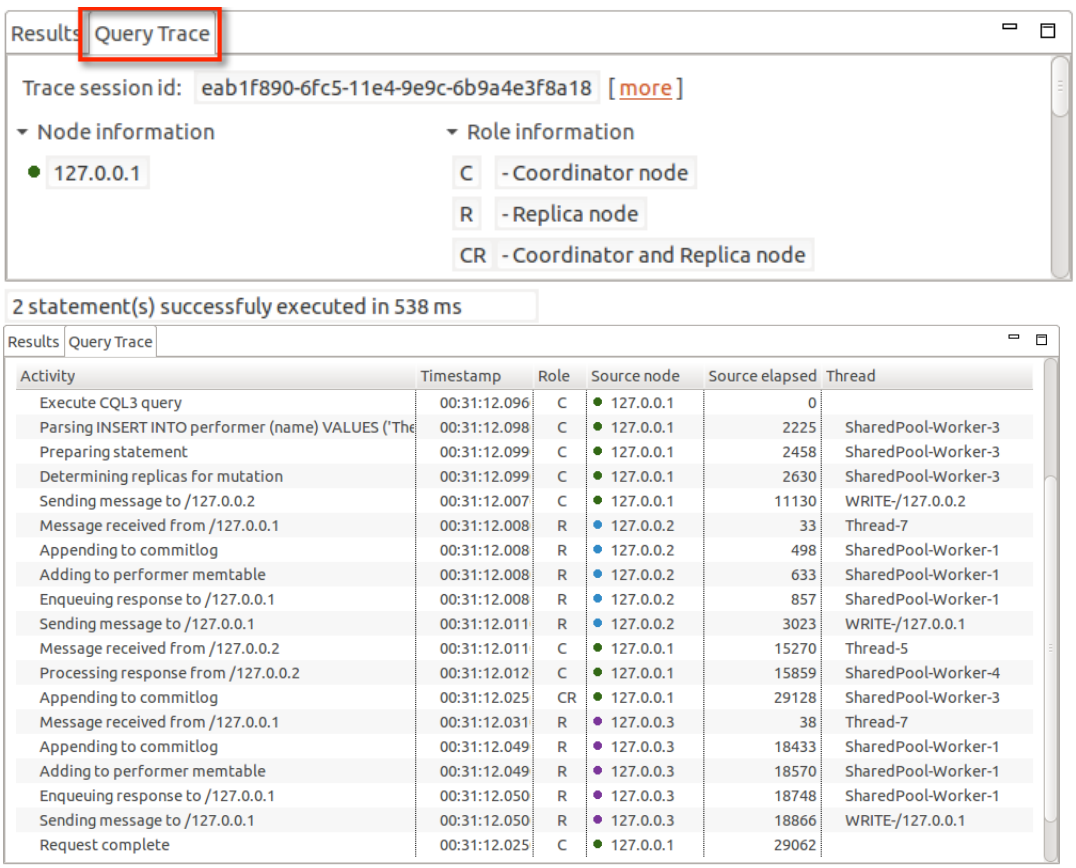
2. In *DevCenter*, set up and connect to the *cascor* cluster, if it does not automatically connect.

3. In the main editor window, write a script to insert a row into the *performer* table for the *musicdb* keyspace, and then execute it.

```
INSERT INTO performer (name) VALUES ('The Beatles');
```



4. In the *Results* window, switch over to the *Query Trace* tab.



Activity	Timestamp	Role	Source node	Source elapsed	Thread
Execute CQL3 query	00:31:12.096	C	127.0.0.1	0	
Parsing INSERT INTO performer (name) VALUES ('The	00:31:12.098	C	127.0.0.1	2225	SharedPool-Worker-3
Preparing statement	00:31:12.099	C	127.0.0.1	2458	SharedPool-Worker-3
Determining replicas for mutation	00:31:12.099	C	127.0.0.1	2630	SharedPool-Worker-3
Sending message to /127.0.0.2	00:31:12.007	C	127.0.0.1	11130	WRITE-/127.0.0.2
Message received from /127.0.0.1	00:31:12.008	R	127.0.0.2	33	Thread-7
Appending to commitlog	00:31:12.008	R	127.0.0.2	498	SharedPool-Worker-1
Adding to performer memtable	00:31:12.008	R	127.0.0.2	633	SharedPool-Worker-1
Enqueuing response to /127.0.0.1	00:31:12.008	R	127.0.0.2	857	SharedPool-Worker-1
Sending message to /127.0.0.1	00:31:12.011	R	127.0.0.2	3023	WRITE-/127.0.0.1
Message received from /127.0.0.2	00:31:12.011	C	127.0.0.1	15270	Thread-5
Processing response from /127.0.0.2	00:31:12.012	C	127.0.0.1	15859	SharedPool-Worker-4
Appending to commitlog	00:31:12.025	CR	127.0.0.1	29128	SharedPool-Worker-3
Message received from /127.0.0.1	00:31:12.031	R	127.0.0.3	38	Thread-7
Appending to commitlog	00:31:12.049	R	127.0.0.3	18433	SharedPool-Worker-1
Adding to performer memtable	00:31:12.049	R	127.0.0.3	18570	SharedPool-Worker-1
Enqueuing response to /127.0.0.1	00:31:12.050	R	127.0.0.3	18748	SharedPool-Worker-1
Sending message to /127.0.0.1	00:31:12.050	R	127.0.0.3	18866	WRITE-/127.0.0.1
Request complete	00:31:12.025	C	127.0.0.1	29062	

You can resize the Query Trace window if necessary to make it easier to view.

5. In the Query Trace window, use the *Role* and *Source node* columns to identify the coordinator node for this insert.

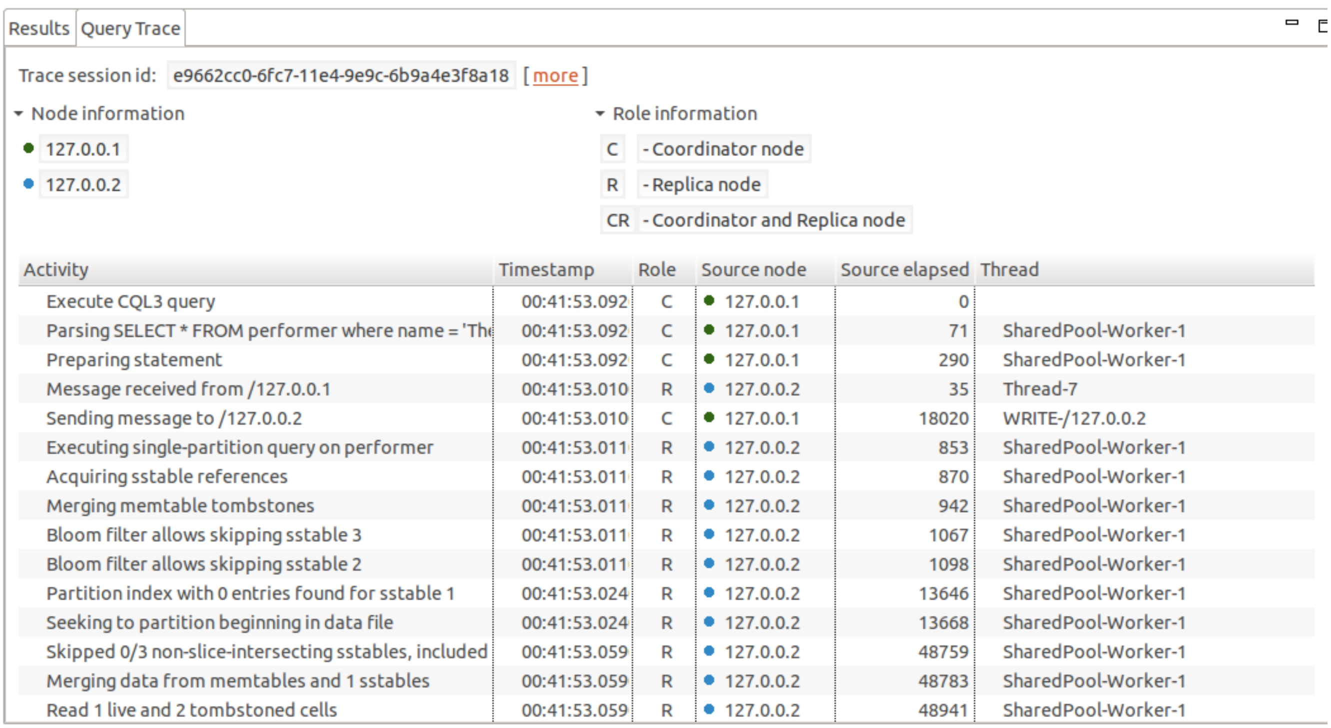
6. In the Query Trace window, look for the *Request complete* trace and use the *Source elapsed* column to determine how long it took for the insert to complete.

The time needed for the request to complete may be different from the trace with the highest Source elapsed. *Why is this?*

7. In the main editor window, write a script to query the same row from the *performer* table for the *musicdb* keyspace, and then execute it.

```
SELECT * FROM performer WHERE name = 'The Beatles';
```

8. Switch to the *Query Trace* window and review the Activity traces.



Results | Query Trace

Trace session id: e9662cc0-6fc7-11e4-9e9c-6b9a4e3f8a18 [more]

Node information
- 127.0.0.1
- 127.0.0.2

Role information
- C - Coordinator node
- R - Replica node
- CR - Coordinator and Replica node

Activity	Timestamp	Role	Source node	Source elapsed	Thread
Execute CQL3 query	00:41:53.092	C	127.0.0.1	0	
Parsing SELECT * FROM performer where name = 'Th	00:41:53.092	C	127.0.0.1	71	SharedPool-Worker-1
Preparing statement	00:41:53.092	C	127.0.0.1	290	SharedPool-Worker-1
Message received from /127.0.0.1	00:41:53.010	R	127.0.0.2	35	Thread-7
Sending message to /127.0.0.2	00:41:53.010	C	127.0.0.1	18020	WRITE-/127.0.0.2
Executing single-partition query on performer	00:41:53.011	R	127.0.0.2	853	SharedPool-Worker-1
Acquiring sstable references	00:41:53.011	R	127.0.0.2	870	SharedPool-Worker-1
Merging memtable tombstones	00:41:53.011	R	127.0.0.2	942	SharedPool-Worker-1
Bloom filter allows skipping sstable 3	00:41:53.011	R	127.0.0.2	1067	SharedPool-Worker-1
Bloom filter allows skipping sstable 2	00:41:53.011	R	127.0.0.2	1098	SharedPool-Worker-1
Partition index with 0 entries found for sstable 1	00:41:53.024	R	127.0.0.2	13646	SharedPool-Worker-1
Seeking to partition beginning in data file	00:41:53.024	R	127.0.0.2	13668	SharedPool-Worker-1
Skipped 0/3 non-slice-intersecting sstables, included	00:41:53.059	R	127.0.0.2	48759	SharedPool-Worker-1
Merging data from memtables and 1 sstables	00:41:53.059	R	127.0.0.2	48783	SharedPool-Worker-1
Read 1 live and 2 tombstoned cells	00:41:53.059	R	127.0.0.2	48941	SharedPool-Worker-1

9. In the Query Trace window, use the *Role* and *Source node* columns to identify the coordinator node and replica nodes for this query.

How many replica nodes needed to be contacted to complete this query?

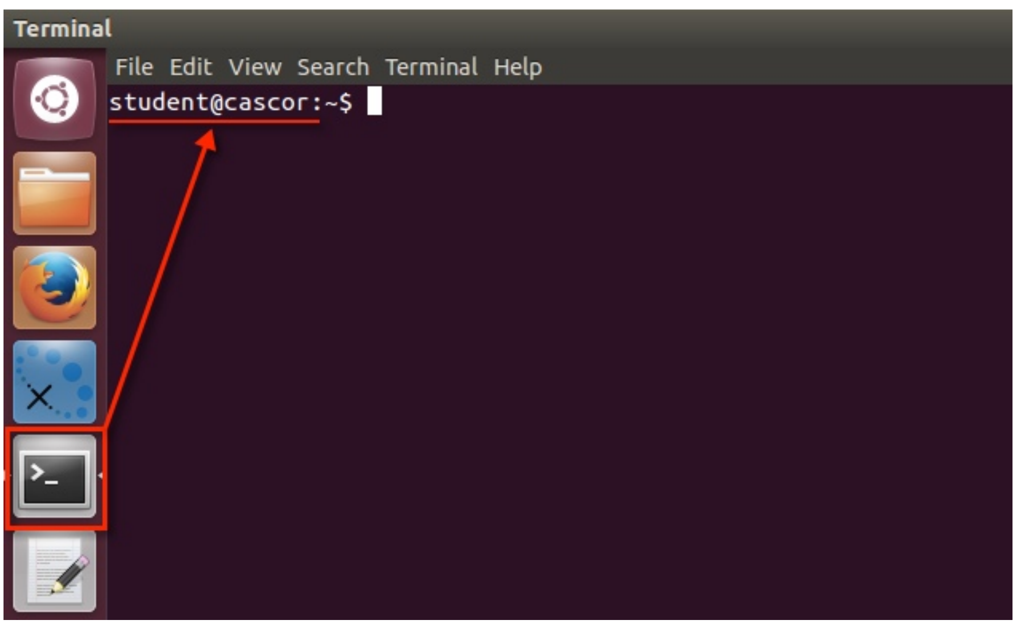
10. In the Query Trace window, find activity traces that pertain to the read path.

Did the bloom filters allow any of the SSTables to be skipped? How many SSTables have been read and merged with memtables? Were there any hits in the key cache?

11. In the Query Trace window, look for the *Request complete* trace and use the *Source elapsed* column to determine that indicates how long it took for the query to complete.

Study the trace for a query that cannot fulfill CL requirements

12. In the virtual machine, open a Terminal window or switch to an existing Terminal running the Linux shell.



13. In the Terminal window, use CCM to stop node2.

```
ccm node2 stop
```

14. In the Terminal, start up *cqlsh*.

```
ccm node1 cqlsh
```

15. In *cqlsh*, enable request tracing.

```
TRACING ON
```

16. In *cqlsh*, INSERT a row into the *performer* table in the *musicdb* keyspace.

```
USE musicdb;
INSERT INTO performer (name) VALUES ('The Beatles');
```

17. In the corresponding trace, check which nodes the row was written to.

```
cqlsh:musicdb> INSERT INTO performer (name) VALUES ('The Beatles');

Tracing session: ddaefd60-6fc9-11e4-9e9c-6b9a4e3f8a18

 activity                                                                 | timestamp                  | source    | source_elapsed
--------------------------------------------------------------------------+----------------------------+-----------+----------------
                                                        Execute CQL3 query | 2014-11-19 00:55:52.630000 | 127.0.0.1 |              0
 Parsing INSERT INTO performer (name) VALUES ('The Beatles'); [SharedPool-Worker-1] | 2014-11-19 00:55:52.630000 | 127.0.0.1 |             71
                                   Preparing statement [SharedPool-Worker-1] | 2014-11-19 00:55:52.630000 | 127.0.0.1 |            276
                       Determining replicas for mutation [SharedPool-Worker-1] | 2014-11-19 00:55:52.630000 | 127.0.0.1 |            466
                                  Appending to commitlog [SharedPool-Worker-1] | 2014-11-19 00:55:52.630000 | 127.0.0.1 |            699
                             Adding to performer memtable [SharedPool-Worker-1] | 2014-11-19 00:55:52.631000 | 127.0.0.1 |            854
                                  Appending to commitlog [SharedPool-Worker-1] | 2014-11-19 00:55:52.631000 | 127.0.0.1 |           1322
                                 Adding to hints memtable [SharedPool-Worker-1] | 2014-11-19 00:55:52.631001 | 127.0.0.1 |           1422
                          Sending message to /127.0.0.3 [WRITE-/127.0.0.3] | 2014-11-19 00:55:52.632000 | 127.0.0.1 |             --
                             Message received from /127.0.0.1 [Thread-7] | 2014-11-19 00:55:52.633000 | 127.0.0.3 |             34
                                  Appending to commitlog [SharedPool-Worker-1] | 2014-11-19 00:55:52.634000 | 127.0.0.3 |           1023
                             Adding to performer memtable [SharedPool-Worker-1] | 2014-11-19 00:55:52.634000 | 127.0.0.3 |           1134
                          Enqueuing response to /127.0.0.1 [SharedPool-Worker-1] | 2014-11-19 00:55:52.634000 | 127.0.0.3 |           1268
                          Sending message to /127.0.0.1 [WRITE-/127.0.0.1] | 2014-11-19 00:55:52.634000 | 127.0.0.3 |           1517
                             Message received from /127.0.0.3 [Thread-4] | 2014-11-19 00:55:52.635000 | 127.0.0.1 |             --
                    Processing response from /127.0.0.3 [SharedPool-Worker-2] | 2014-11-19 00:55:52.635000 | 127.0.0.1 |             --
                                                          Request complete | 2014-11-19 00:55:52.631063 | 127.0.0.1 |           1063
```

With a consistency level of ONE, it is still possible to insert data as long as one of the replica nodes is still up. Remember that it is possible for the coordinator to also be a replica node. Did the coordinator try to send a message to node2? Was there a hint saved?

Explore the trace for a COUNT aggregate query

18. Run a query to get a COUNT of all of the rows in the *performers_by_style* table.

```
SELECT COUNT(*) FROM performers_by_style;
```

19. From the trace, determine the amount of time for this query to complete.

How much time did it take to get a count of the rows? Why did it take as long as it did?

20. Exit cqlsh and start node2 again.

```
EXIT
ccm start
```

END OF EXERCISE

Exercise 5: Using *cfstats* to obtain and measure performance

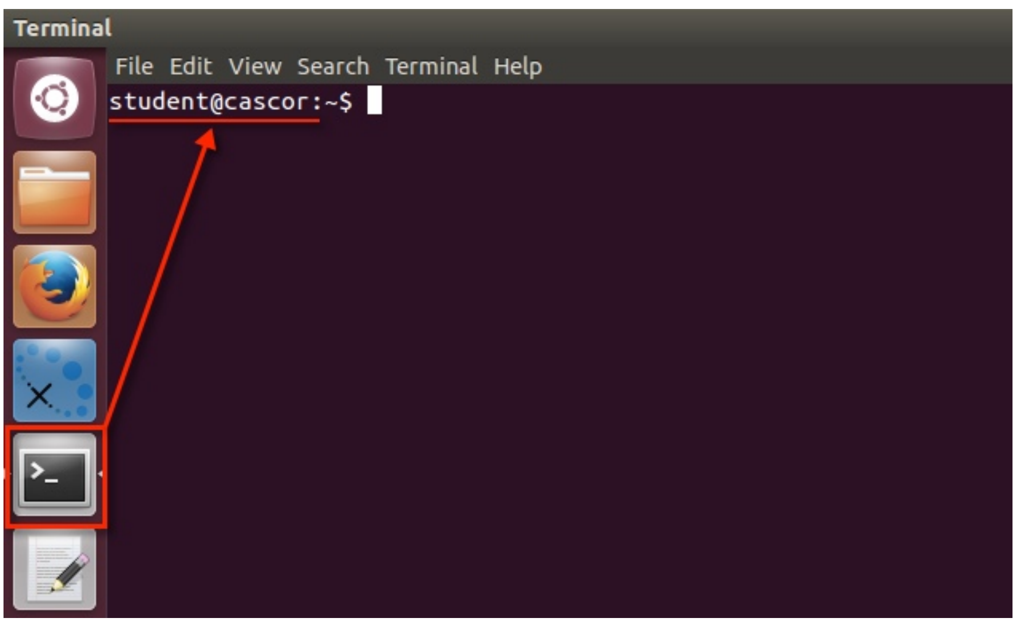
In this exercise, you will:

- Compare workload volume and type across multiple tables
- Examine key statistics in measuring performance

Steps

Compare workload volume and type across multiple tables

1. In the virtual machine, open a Terminal window or switch to an existing Terminal running the Linux shell.



2. From the Linux shell, navigate to the *exercise-5* directory for the *read-path* module.

```
cd ~/cascor/read-path/exercise-5
```

3. In the *exercise-5* directory, run the script *musicdb-workload.sh*.

```
./musicdb-workload.sh 1
```

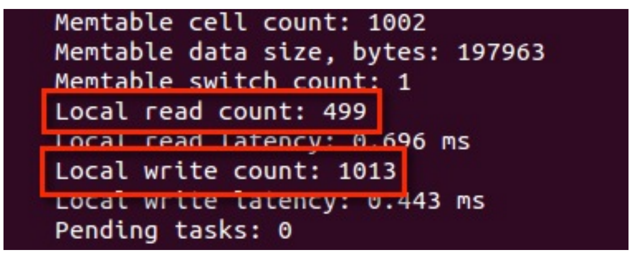
This script simulates a workload being run on the tables in the musicdb *keyspace. Since it will run indefinitely, a ctrl-c keypress will be needed to stop the script later.*

4. In the virtual machine, open another Terminal window.

5. From the new Terminal window, run *nodetool cfstats* for the *album* table in the *musicdb* keyspace.

```
ccm node1 nodetool cfstats musicdb.album
```

6. From the *nodetool cfstats* output, find the *local read count* and *local write count* for the *album* table.



```
Memtable cell count: 1002
Memtable data size, bytes: 197963
Memtable switch count: 1
Local read count: 499
Local read latency: 0.696 ms
Local write count: 1013
Local write latency: 0.443 ms
Pending tasks: 0
```

Based on the two counts, would you consider the workload being run against the album *table to be read-heavy, write-heavy, or mixed?*

7. Run *nodetool cfstats* for the *tracks_by_album* table in the *musicdb* keyspace.

```
ccm node1 nodetool cfstats musicdb.tracks_by_album
```

8. From the *nodetool cfstats* output, find the *local read count* and *local write count* for the *tracks_by_album* table.

```
Memtable cell count: 4000
Memtable data size, bytes: 4841600
Memtable switch count: 1
Local read count: 0
Local read latency: NaN ms
Local write count: 4000
Local write latency: 0.756 ms
Pending tasks: 0
```

Based on the two counts, would you consider the workload being run against the tracks_by_album table to be read-heavy, write-heavy, or mixed?

9. Run *nodetool cfstats* for the *performer* table in the *musicdb* keyspace.

```
ccm node1 nodetool cfstats musicdb.performer
```

10. From the *nodetool cfstats* output, find the local read count and local write count for the *performer* table.

```
Memtable cell count: 1002
Memtable data size, bytes: 197963
Memtable switch count: 1
Local read count: 499
Local read latency: 0.696 ms
Local write count: 1013
Local write latency: 0.443 ms
Pending tasks: 0
```

Based on the two counts, would you consider the workload being run against the performer table to be read-heavy, write-heavy, or mixed?

11. Run *nodetool cfstats* for the *musicdb* keyspace.

```
ccm node1 nodetool cfstats musicdb
```

12. From the output, compare the overall read and write count for the keyspace.

```
student@cascor:~$ ccm node1 nodetool cfstats musicdb
Keyspace: musicdb
        Read Count: 8844
        Read Latency: 17.305342718227045 ms.
        Write Count: 64920
        Write Latency: 0.16705177141096736 ms.
        Pending Flushes: 0
                Table: album
                SSTable count: 7
                Space used (live): 1956626
                Space used (total): 1956626
                Space used by snapshots (total): 0
                SSTable Compression Ratio: 0.5215163356115958
                Memtable cell count: 3006
                Memtable data size: 68001
```

How do the read and write counts for the individual tables compare to the total? Based on this, is it possible to determine the most active tables and tables least frequently used?

Examine key statistics in measuring performance

13. In the *Terminal*, run *nodetool cfstats* on the *album* table and take a look at the read and write latency.

```
ccm node1 nodetool cfstats musicdb.album
```

```
Memtable cell count: 3004
Memtable data size, bytes: 2507758
Memtable switch count: 107
Local read count: 171592
Local read latency: 0.581 ms
Local write count: 172414
Local write latency: 0.014 ms
Pending tasks: 0
```

Is the read operations or the write operations faster? Is this what you would expect with Cassandra?

14. From the *nodetool cfstats* output, take a look at the bloom filter statistics for the *album* table.

```
Local write latency: 0.246 ms
Pending flushes: 0
Bloom filter false positives: 43
Bloom filter false ratio: 0.02055
Bloom filter space used: 13024
Compacted partition minimum bytes: 104
Compacted partition maximum bytes: 5722
```

Checking the false positive ratio can also be helpful when tuning performance. Is the current ratio similar to the bloom_filter_fp_chance table property?

15. From the *nodetool cfstats output*, take a look at the tombstone statistics for the *album* table.

```
Average live cells per slice (last five minutes): 1.0
Maximum live cells per slice (last five minutes): 1.0
Average tombstones per slice (last five minutes): 0.0
Maximum tombstones per slice (last five minutes): 0.0
```

Keeping the number of tombstones low is also important for performance tuning. The more tombstones that exist for a partition or row, the slower the reads will be.

END OF EXERCISE