A REPORT

ON

Cassandra JDK version upgrade

By

Adarsh Rajesh 2019A7PS0230H

AT

Nutanix Technologies , Bangalore

A Practice School II Station of

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

June, 2023

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)

Practice School Division

Station: Nutanix Technologies **Centre:** Bangalore

Duration: 5.5-6 months **Date of Start:** 17/01/2023

Date of Submission: 21/06/2023

Title of the Project: Cassandra JDK version upgrade

ID No./Name(s)/: Adarsh Rajesh (2019A7PS0230H)

Discipline(s) of the Student(s): Computer Science

Name(s) and Designation(s) of the expert(s) : Swagat Borah (Director , Engineering) , Mangesh Kute (Senior Technical Manager) , Godithi Lakshmi Sruthi (Member of Technical Staff - 3)

Name(s) of the: Chandra Shekar RK
PS Faculty

Key Words: Java , Cassandra , JDK , database , garbage collection, benchmarking

Project Areas: Distributed Database , JDK upgrade

Abstract: The major objective of this project is to build and ship Cassandra with the latest version of JDK(17) and perform microbenchmarking to obtain performance metrics such as throughput , latency , CPU utilization and memory consumption . Compare and analyze the benefits of upgrading to JDK 17 while ensuring that there are no performance/functionality regression.

Signature(s) of Student(s) Signature of PS Faculty

Date

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE
PILANI (RAJASTHAN)

PRACTICE SCHOOL DIVISION

Response Option Sheet

Station: Nutanix Technologies Center: Bangalore

ID No. & Name(s): Adarsh Rajesh (2019A7PS0230H)

Title of the Project: Cassandra JDK version upgrade

Usefulness of the project to the on-campus courses of study in various disciplines. Project should be scrutinized keeping in view the following response options. Write Course No. and Course Name against the option under which the project comes.

Refer Bulletin for Course No. and Course Name.

Code No.	Response Option	Course No.(s) & Name
1.	A new course can be designed out of this project.	
2.	The project can help modification of the course content of some of the existing Courses	
3.	The project can be used directly in some of the existing Compulsory Discipline Courses (CDC)/ Discipline Courses Other than Compulsory (DCOC)/ Emerging Area (EA), etc. Courses	
4.	The project can be used in preparatory courses like Analysis and Application Oriented Courses (AAOC)/ Engineering Science (ES)/ Technical Art (TA) and Core Courses.	
5.	This project cannot come under any of the above mentioned options as it relates to the professional work of the host organization.	

_____

_____

Signature of Student
Date:

Signature of Faculty
Date:

Acknowledgements

I would like to express my sincere gratitude and heartfelt appreciation to Mangesh Kute , Godithi Lakshmi Sruthi , Arumugam Arumugam and Raghav Tulshibagwale and all of the NDB team members for their contributions and support. I am truly fortunate to have had the opportunity to work with such talented individuals, and I extend my deepest thanks for their exceptional efforts. Their collaboration and professionalism have made this journey a rewarding and memorable experience.

I would also like to thank Swagat Borah for giving me the opportunity to work on this project .

My gratitude to our PS Faculty Mr. Chandra Shekar RK for guiding us . Finally , I would like to thank BITS Pilani for giving me the opportunity to pursue a 6-month internship at Nutanix as a part of the Practice School programme.

CONTENTS

1. PROJECT OVERVIEW

In distributed file systems , the files and data are distributed/replicated/deduplicated across systems . In order to access the data , it becomes necessary to know the location of the files present. This information is captured as part form of metadata. All the metadata information is handled by Medusa/Cassandra. Nutanix has built some additional features on top of the open source cassandra in a very customized manner in order to cater to the needs of the company data. The Stargate is the data component which would ask medusa/cassandra for the metadata to get the exact location of the file/data and cater to client requests and access the data .

Cassandra daemon is the main cassandra java process which handles various requests such as random read , random write , sequential read and sequential write. The entire code and logic for the daemon is written in java. Currently the daemon is compiled with Java 8 version and all the services running under Nutanix currently use OpenJDK 8 runtime environment on the clusters.

This project aims to explore the possibility and outcomes of upgrading the java version for cassandra service to use the latest JDK version - JDK 17. Since the release of Java 10 , there has been a 6-month-rapid release cycle followed by Oracle. Starting with Java 11 , there were new LTS releases every 3 years. We aim to compare across the LTS versions of JDK to understand the benefits of migrating the Cassandra codebase from OpenJDK 8 to OpenJDK 11 or OpenJDK 17. The focus of the project is to validate if we get any improvements in Java garbage collection performance due to the better and optimized GC algorithms implemented in later release versions. This in turn should result in better throughput and faster execution times of Cassandra service.

Primarily this project intends to benchmark cassandra service to compare the performance by analyzing the following metrics - application throughput , application execution time , garbage collection pause time , CPU usage , memory consumption and heap usage.

Release	Released	Active Support	Security Support
8 (LTS)	18 Mar 2014	Ended on 31 Mar, 2022	Ends on 31 Dec, 2030
11 (LTS)	25 Sep 2018	Ends on 30 Sep, 2023	Ends on 20 Sep, 2026
17 (LTS)	14 Sep 2021	Ends on 30 Sep, 2026	Ends on 30 Sep, 2029

Table 1: Details regarding support and maintenance of various LTS releases

2. BACKGROUND STUDY

A theoretical study and research was conducted as part of this project to enlist the salient features introduced in OpenJDK 11 and OpenJDK 17 which would be relevant for this project.

OpenJDK 11

1. **Module system** : Modules are collections of packages / interfaces / classes and add a higher level of aggregation above the package. The key benefits of the introduction of module system are :
    - Leads to more efficient class loading mechanisms. The optimization techniques of JVM would be better utilized since all the packages required by a class would be encoded within the respective modules. This can lead to faster compilation and execution times.
    - Enforces strong encapsulation for developers. Developers need to explicitly declare and export the packages required by a module. This level of encapsulation makes the application more secure.
    - With modules , it is possible to hide certain packages for internal use only. The public access restriction of packages can be fine-tuned to improve security and reliability.
    - In order to migrate applications from OpenJDK 8 to OpenJDK 11 , the application libraries and modules from JDK 8 classpath should be moved to JDK 11's module class.

2. **Garbage collection**
    - OpenJDK 8 uses Parallel GC as its default garbage collection algorithm. OpenJDK 11 uses G1 Gc as its default garbage collector.
    - Parallel GC is a throughput collector that uses multiple threads to speed up garbage collection.
    - The aim of G1Gc is to strike a balance between throughput and pause times. G1 Gc aims to achieve higher throughputs while meeting the pause time goals with higher probability.
    - Epsilon : a no-op garbage collector introduced as an experimental feature in JDK 11.
    - ZGC : a scalable low latency garbage collector introduced as an experimental collector in JDK 11.
    - Currently the cassandra daemon process uses the Concurrent Mark and Sweep collector (CMS) for GC activities. However the CMS collector had been deprecated since Java 9.

3. **Profiling and diagnostics :** OpenJDK 11 introduces a new unified GC logging framework that works more effectively as compared to the older framework. Since the logging format has changed , any script that parses GC logs to obtain GC stats may have to be altered accordingly.

OpenJDK 17

1. **Garbage collection**
    - The Epsilon and ZGC garbage collectors have been productised in JDK 17.
    - Shenandoah : a low-pause time garbage collector has been introduced.
    - Both OpenJDK 11 and OpenJDK 17 use G1Gc as their default garbage collectors. There have been constant improvements and optimizations made in the G1 garbage collection algorithm to reduce pause times and lower overhead memory consumption .
    - The CMS garbage collector has been removed

2. The exploratory **Java-based early (AOT) and Graal just-in-time (JIT)** compiler has been removed. However, the exploratory JVMCI has been retained so that the developers can keep using the externally built variants of the compiler for JIT compiling.

3. **Strong encapsulation for JDK internals** : Java's internal API is accessed by many third-party applications , frameworks and libraries. In Java 16 , the encapsulation was made stronger and by default , it was not allowed to access the internal APIs of JDK. However this can be superseded using some JVM flags set during run time.

Note : All the features listed in JDK 11 would also be applicable and present in JDK 17 unless deprecated later .

State of JAVA ecosystem report

Based on the data gathered from millions of applications with performance benchmarked , a state of ecosystem report was published by New Relic in 2020. The report provides insights into the most used JDK version in production , most common heap size configurations and most commonly used GC algorithms. The findings of the report suggests the following :

1. A vast majority of applications continued running on JAVA 8 until 2020. Since then there has been a shift observed with more companies migrating their applications to use JAVA 11. JAVA 17 was not very widely used.4

2. There has been a distinct shift in preferences around which GC algorithm to use. Owing to the performance gains obtained with the G1 collector on Java 11 and later releases , it is the most widely used garbage collection algorithm.

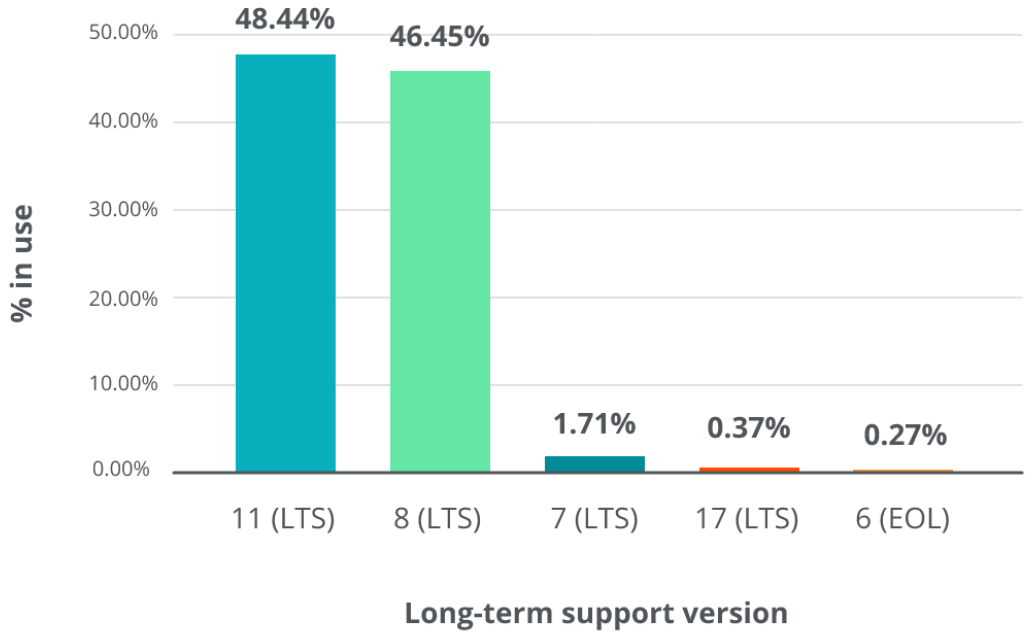


Figure 1: % usage of LTS versions by production level applications
(Source : 2022 State of the Java Ecosystem Report | New Relic)

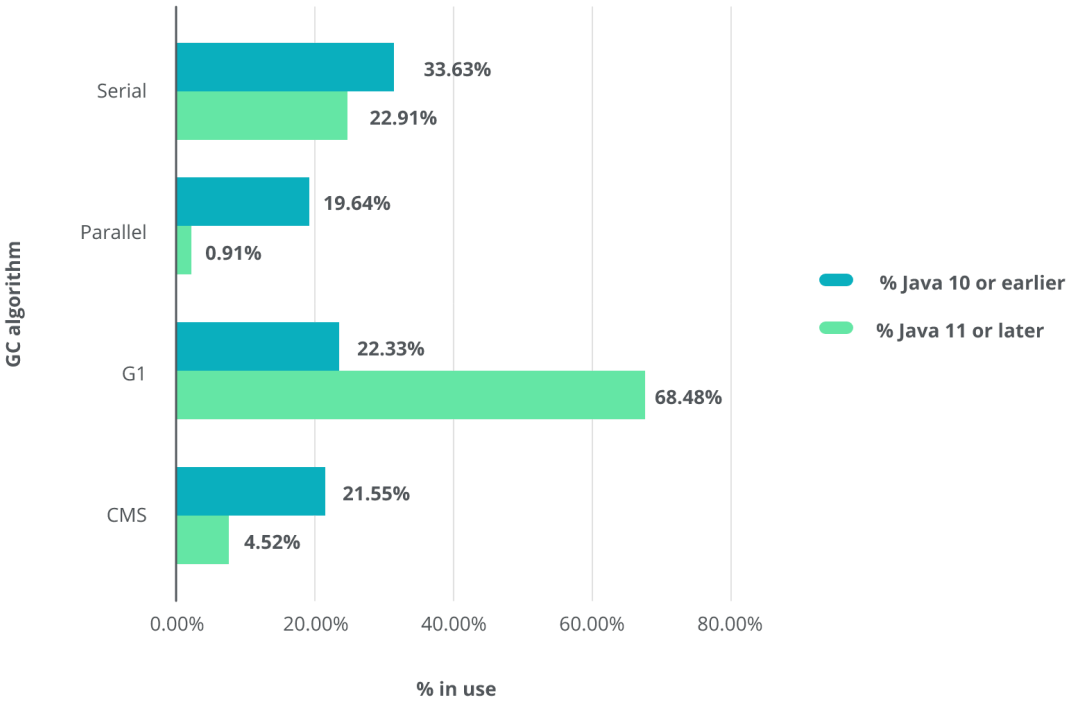


Figure 2: % usage of garbage collectors used in applications
(Source : 2022 State of the Java Ecosystem Report | New Relic)

3. CASSANDRA ARCHITECTURE

Cassandra is an open source distributed key-value store which has been customized to suit the needs and requirements of the company. In distributed file systems , the files and data are distributed/replicated/deduplicated across systems . In order to access the data , it becomes necessary to know the location of the files present. This information is captured as part form of metadata. All the metadata information is handled by Medusa/Cassandra. Nutanix has built some additional features on top of the open source cassandra in a very customized manner in order to cater to the needs of the company data. A lot of changes and modifications have been made on top of open-source cassandra , to a point where it is now completely diverged and cannot be integrated back into the open source version.

The key features of this distributed metadata layer are :

1. High availability and fault tolerance
2. Consistent read/writes : Open source cassandra did not support strict consistency instead supports eventual consistency. Paxos algorithm (distributed consensus algorithm) was implemented on top of it to ensure strict consistency.
3. Dynamically scales up/down

Components

Cassandra Daemon

- Java process running the cassandra service
- It is a distributed key-value store open sourced by Facebook.
- Follows row and column store model
  <Keyspace , Column Family , Row Name , Column Name>
- Uses Log-Structured merge trees to enable fast writes and slow reads
- Use consistent hashing for key distribution

Cassandra Monitor

- C++ process implemented by company to monitor the daemon.
- Implemented as a parent-child process. The child process is the one that monitors daemon. The only job of the parent process is to start/restart the child process.
- Periodically checks the health of the monitor - tracks fault tolerance status , checks memory and thread progress status , checks heap usage , monitors GC activities , compaction stats , initiates Cassandra self healing etc.
- Maintains and updates configuration and state of Cassandra daemon (like kNormal , kForwarding etc.)
- Sets up pre start up / post crash configurations for daemon

Medusa

- C++ API abstraction layer that converts metadata read/writes to Cassandra reads/writes
- Caches 3 major metadata maps for faster reads
- Synchronizes outstanding requests (read/writes)

Dynamic Ring Changer

- Used for node addition / node removal (dynamic scaling)
- Perform metadata disk replacement
- Perform self healing incase any node crashes
- Ring skew correction (in events of any node going down) to ensure each node has equal share of data.

Cassandra WAL - Write Ahead Log

- Used for crash recovery.
- Supports 2 types of records : Delta (intended change) and Checkpoint (state of application)
- Logs are stored in distributed fashion with Cassandra as backend

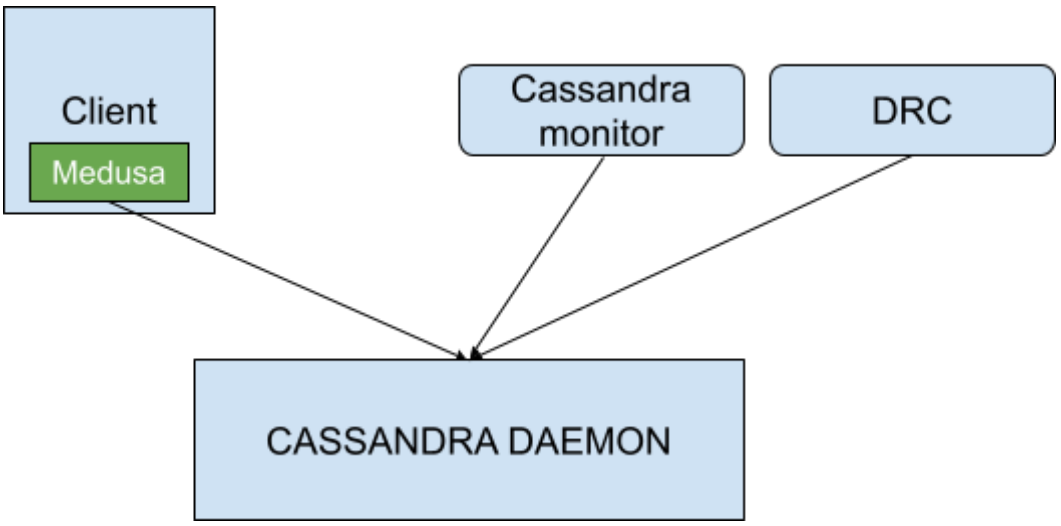


Figure 3: Cassandra inter-service communication

Cassandra Read-Write Workflow

Any LSM tree based database would typically consist of the following components :

1. **Commit-log** : Writes keep getting appended to the commit log. It is a persistent file on the disk.
2. **Memtable** : In memory data structure for absorbing writes. Helps in fast sequential write to disk when filled.
3. **SSTable** : Immutable sorted string table for storing key-value pairs on disk
4. **Bloom Filter** : A probabilistic data structure that checks the presence of key in O(1) time. A false positive match is possible however a false negative match would never occur. Helps improve read performance.

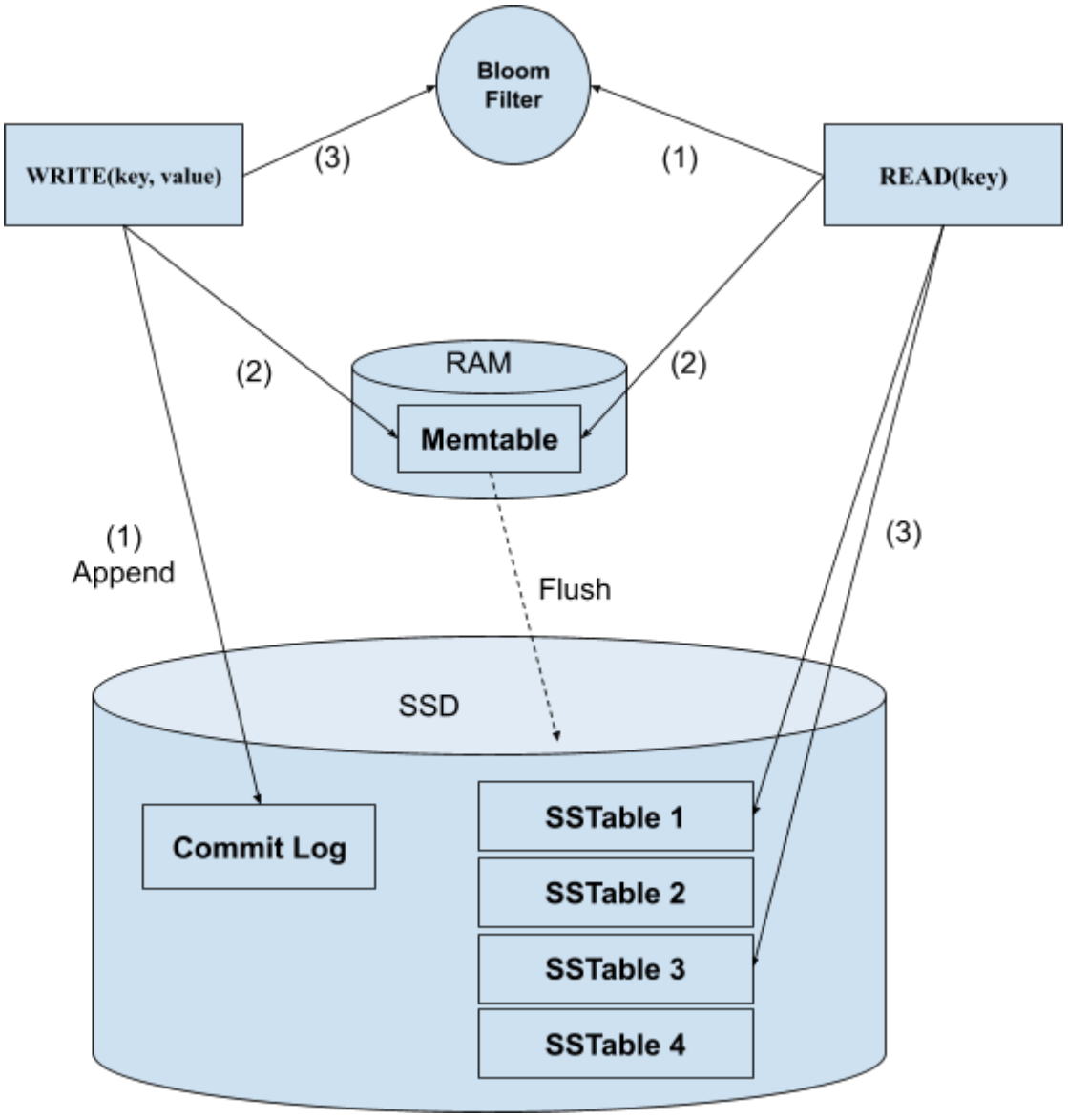


Figure 4: Cassandra Read-Write path

4. JAVA GARBAGE COLLECTION

Automatic garbage collection is one of the key features of the Java programming language. Garbage collection is the process of scanning the heap memory and identifying the objects that are in use and deleting the unused objects so that the memory used by unused objects can be reclaimed. An object is said to be in use/references if any part of the program still maintains a pointer to that object.

Working of java garbage collection

The garbage collection uses a mark-and-sweep method.

Phase 1: Marking - In this step , the garbage collector identifies the pieces of memory that are in use and not in use. Whenever a java object is created in heap , it has a mark bit that is set to 0. The garbage collector traverses the object tree and sets the mark bit of all reachable objects from root to 1.

Phase 2: Sweep - In this phase , the unused objects are deleted and memory is reclaimed. The deletion of objects can be done in two stages :

a) Normal deletion - Removes unreferenced objects leaving behind referenced objects in heap and pointers to free space
b) Deletion with compaction - To get improved performance , the remaining referenced objects are compacted so that memory allocation of newer objects is faster.

However, the above method of compaction of objects can be inefficient if the number of objects increases thereby leading to longer GC times. Thus the information from object allocation behavior can be used to enhance JVM performance.

Generational Garbage collection

The memory heap is divided into different regions that help make GC more efficient.

Young Generation (Eden space + Survivor space) : This is where new objects are allocated and aged. The **Eden** space is the region where objects are created. When the Eden space is full , the unreferenced objects are removed by the garbage collector and the used objects are moved to **Survivor** space. This event is called ‘Minor garbage collection’. All minor collections are ‘stop-the-world’ events i.e , the application threads are stopped until the collection is completed.

Old Generation (Tenured space) : The **tenured** space is the region where long-lived objects are stored. Generally, an age threshold (measured in terms of GC cycles survived) is set for objects in the young generation , upon crossing which the objects are moved to the old generation. Eventually the old generation objects would also be garbage collected. This is called ‘Major garbage collection’. Major collections are also ‘stop-the-world’ events and generally take longer time since it involves deleting the entire set of live objects.
Minor collections involve faster and simpler garbage collection cycles since they occur more frequently and act on smaller heap regions of objects.

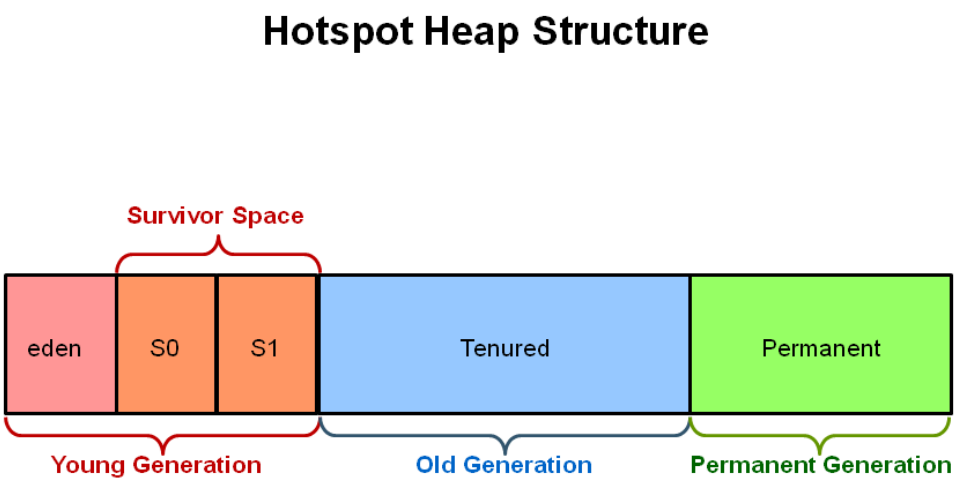


Figure 5 : JVM heap structure
(Source : Java Garbage Collection Basics (oracle.com))

Note : In versions prior to Java 8, there was a third region - The permanent generation which included the metadata required by JVM. This was removed in Java 8 and later release versions.

Garbage Collection Algorithms

Serial GC

Both minor and major collections are done serially. It uses the mark-and-compact strategy of garbage collection. Serial GC may be a good choice for applications that do not have low pause time requirements. Generally it is not a good idea to use this algorithm for production level services since the garbage collection may frequently take over application threads and freeze the other processes ('stop-the-world' events).

Parallel GC

Use multiple threads to carry out young generation GC. By default on a host using 'n' CPUs the parallel garbage collector uses 'n' parallel threads. The number of parallel threads to use can be controlled by JVM options too. It is also known as a 'throughput' collector since it can use multiple CPUs to increase application throughput. Parallel GC is the default garbage collector of the JVM in Java 8.

Concurrent Mark-and-Sweep (CMS)

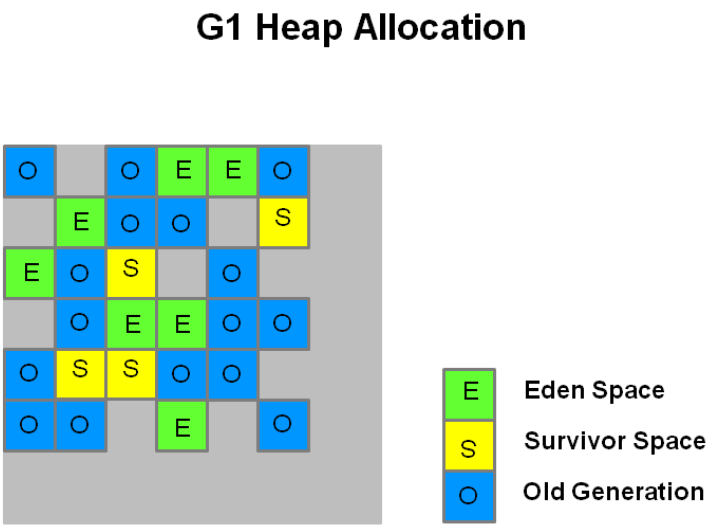
The CMS garbage collector is used to collect the tenured space. It aims to reduce GC pauses by carrying out most of the collection concurrently with application threads. Since it uses multiple concurrent threads , it requires more resources and processing power as compared to other garbage collectors. Typically applications that have a large set of long lived data and run on multiple machines can benefit from CMS. Currently the Cassandra JVM is configured to use the CMS garbage collector.
As of Java 9, the CMS garbage collector had been deprecated. From Java 14 onwards , it was completely removed.

G1 GC

The G1 garbage collector was designed and developed to be a long term replacement for the CMS garbage collector. It is available from JDK 7 and later releases. It is mainly targeted for multiprocessor machines. The aim of G1Gc is to strike a balance between throughput and pause times. G1 Gc aims to achieve higher throughputs while meeting the pause time goals with higher probability. The G1 algorithm takes a different approach with regards to dividing the heap regions as opposed to other generational garbage collectors that we discussed before.

G1 heap structure

The heap is partitioned into a set of equal sized regions (each with contiguous range of virtual memory). These regions are mapped to logical representations of Eden , Survivor and old generation spaces.



Young Generation Collection in G1

New objects are allocated to Eden regions. A young GC will be triggered when a threshold of Eden regions have been filled. Live objects are moved to survivor regions or to old generation space (if their aging threshold is met). Young GCs are ‘stop the world’ events.

Old Generation Collection in G1

Old generation garbage collection occurs in following phases :

1. Concurrent Marking phase : Live objects information of the old region is calculated concurrently while the application is running. The liveness information can then be used to determine which regions to reclaim during evacuation pause.
2. Remark Phase : Empty regions are removed and reclaimed. Region liveness is computed for each region. Uses the Snapshot-at-the-Beginning (SATB) algorithm which is much faster then what was used with CMS.
3. Cleanup Phase: G1 selects the old regions with lowest ‘liveness’ . This step occurs concurrently with young GC. So during this phase there is a mixed collection (young GC + old GC) going on. If the G1 does not find any old regions worth collecting then it would proceed with only young collections.

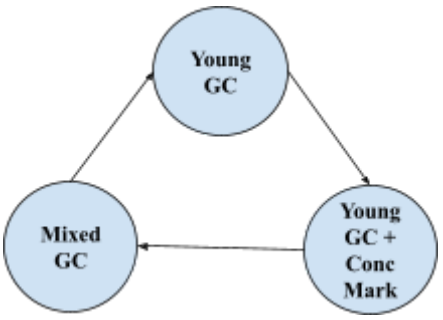


Figure 6: States of G1 collection

Major improvements to G1 GC since JDK 8

1. **Parallel Full GC**: In JDK 9, G1GC introduced parallel full garbage collection, which allows multiple threads to be utilized during the full GC phase. This improvement helps reduce the pause times during full garbage collection, resulting in improved application throughput.

2. **String Deduplication**: JDK 8 introduced the concept of String Deduplication in G1GC, which helps to reduce memory consumption by eliminating duplicate instances of strings in the heap. This feature was further improved in subsequent versions, including JDK 9 and JDK 11, leading to better memory utilization.

3. **Concurrent Class Unloading**: Starting from JDK 9, G1GC supports concurrent class unloading, which means classes that are no longer needed can be unloaded while the application is running, reducing the memory footprint and improving garbage collection efficiency.

4. **Ergonomics and Tuning**: With each JDK release, G1GC's default configuration and ergonomics have been fine-tuned to provide better performance out of the box. The heuristics and algorithms used for determining various G1GC parameters, such as heap sizing, region sizing, pause time goals, etc., have been improved to adapt better to different workloads.

5. **Mixed GC Algorithm Enhancements**: The mixed garbage collection algorithm used in G1GC has been improved in subsequent JDK releases. JDK 10 introduced the concept of concurrent refinement, which helps to reduce the impact of mixed GC pauses on application threads. JDK 11 further enhanced mixed GC by introducing the notion of initiating mixed GC cycles based on the number of young GC cycles rather than time-based triggering, resulting in more predictable and efficient mixed GC behavior.

6. **Performance and Reliability Enhancements**: Numerous performance optimizations and bug fixes have been implemented in G1GC across JDK versions. These improvements address issues related to pause times, throughput, memory fragmentation, and overall stability.

5. EXPERIMENTS CONDUCTED

Goals
The set of experiments aim to compare performance stats of Cassandra service across LTS versions of JDK. We analyze the following metrics to capture performance gain -

- Application Throughput - Medusa ops/sec
- Latency - Application execution time & GC pause times
- CPU usage
- Heap usage

Setup
We compile and build cassandra daemon with JDK 8, 11 and 17. The generated binaries are then pushed on a 4-node cluster with each CVM having 8 cores of CPU. We perform isolated testing of Cassandra service , meaning that we ensure that other services using java do not interfere with garbage collection .

Pre run configs

- Disable minor compaction to obtain consistent results across runs
- Disable leader only reads : Typically once the leader is elected , all read requests would be invariably directed to the leader only. We want to prevent this so that all nodes are equally pushed to serve requests
- Disable buffer pool
- Restrict CPU usage of Cassandra to 3 cores
- Run major compaction on each node before every run
- Run explicit full GC using jcmd on each node before every run

Experiments

After several iterations of runs , we have selected two sets of experiments with below mentioned config settings to help draw our conclusions.
Every result obtained for each JDK version is an average of 3 runs performed.

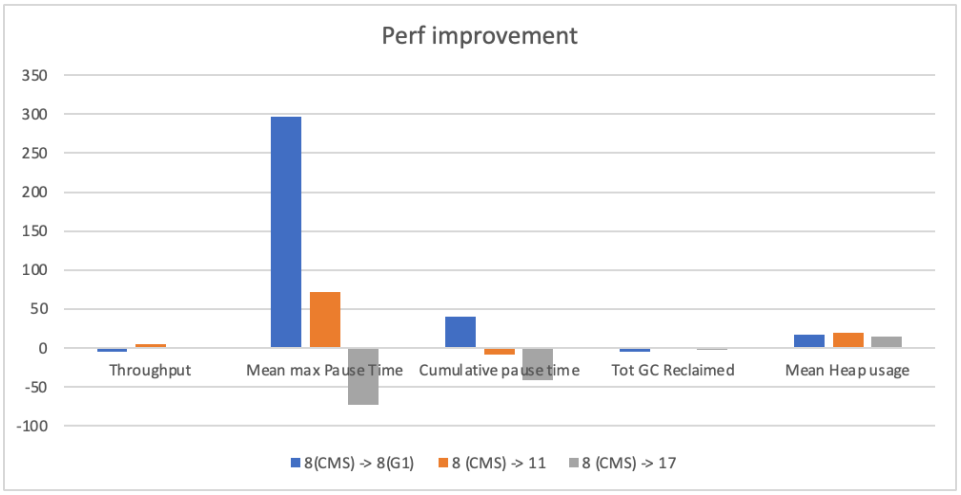
Exp 1 (Metadata load = ~100 GB)
num_vdisks = 5 ; vdisk_size = 5gb ; test_type = rw ; outstanding ops = 640 ;
runtime = 1800 secs ; CPU given to cassandra = 3 cores ; Heap size = 2560 M

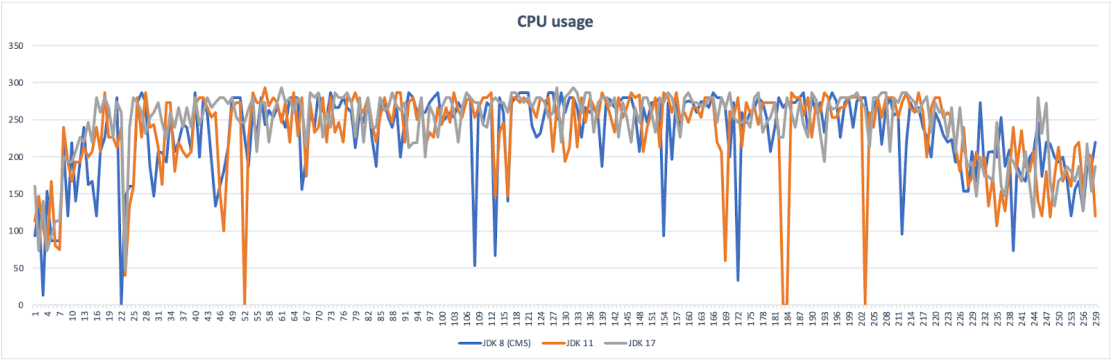
Exp 2 (Metadata load = ~ 50 GB) - Varying Heap sizes
num_vdisks = 5 ; vdisk_size = 5gb ; test_type = rw ; outstanding ops = 640 ;
runtime = 1800 secs ; CPU given to cassandra = 3 cores ;
Heap sizes = 2.5G , 1.5G , 1 G , 0.5G

6. RESULTS

EXP 1

	Throughput	Mean max Pause Time	Cumulative pause time	Tot GC Reclaimed	Mean Heap usage
8 (CMS)	9203	107.02	47942.75	6.08E+12	50.06%
8 (G1Gc)	8747	425.1	66912	5.75E+12	58.85%
11	9692	184.37	43665	6.14E+12	59.98%
17	9260	29.47	27843.67	5.96E+12	57.43%
8(CMS) -> 8(G1)	-4.954906 %	297.2154737 %	39.56646208 %	-5.427631579 %	17.55892928 %
8 (CMS) -> 11	5.3134847 %	72.27621005 %	-8.922621251 %	0.986842105 %	19.81622054 %
8 (CMS) -> 17	0.6193633 %	-72.46309101 %	-41.92308535 %	-1.973684211 %	14.7223332 %





EXP 2

8 (CMS)

	Throughput	Mean max Pause Time	Cumulative pause time	Tot GC Reclaimed	Heap usage
2.5 G	13111.8	118	62649	5.90E+12	50-70%
1.5 G	13605.6	86.31	69103.8	5.69E+12	Upto 80%
1 G	8094	73.36	67623	2.59E+12	Upto 80%
0.5 G	8544.3	223.55	192660	3.28E+12	Upto 90%

Note : For 0.5G heap size , the values are averaged over 3 nodes only since one node crashes due to long GC pause times and thereby causing timeout error for read-write load.

Some observations based on the runs with 8 (CMS) :

1. We see a throughput drop with heap sizes 1G and 0.5G
2. With 0.5G , we see very high cumulative GC pause time (~ 10% of runtime) . These significant GC events also cause cassandra to crash on one of the nodes due to high GC pause times.
3. Also the GC reclaimed is considerably lower with reduced heap size.

We now run the experiments with JDK 17 to check if we are able to sustain rw workload with consistent throughput on lower heap set-ups.

17 (default = G1Gc)

	Throughput	Mean max Pause Time	Cumulative pause time	Tot GC Reclaimed	Heap usage
2.5 G	13098	37.95	36152.5	5.76E+12	Upto 70%
1.5 G	14723.5	35.28	51584.25	6.27E+12	Upto 80%
1 G	10134.75	33.05	65721.825	5.96E+12	Upto 80%
0.5 G	11788.5	108.9	156724.5	5.01E+12	Upto 95%

7. CONCLUSIONS

EXP 1

1. As we can see from EXP 1 , there is a significant decrease in GC pause times from JDK 8 -> 17 , with no degradation in throughput or CPU utilization.
2. As far as throughput is concerned , we don't see a major improvement from JDK 8 -> 17. The % reduced pause time is not seen to be large enough to impact the throughput. Thus we went ahead with EXP 2 where we have performed the runs with varying heap sizes.

EXP 2

1. In EXP 2 , it is observed that restricting heap size leads to more significant GC events. With 0.5G on JAVA 8 (CMS) , we see very high cumulative GC pause time (~ 10% of runtime) . These significant GC events also cause cassandra to crash on one of the nodes due to high GC pause times.
2. When the runs were repeated with JAVA 17 , we did not see node crashes due to long GC pauses. Also the GC pause times were significantly lower with throughput and GC reclaimed being similar or higher in some cases.

Overall

- Clearly there is an improvement in Garbage collection performance while upgrading from Java 8 to Java 17.
- Java 17 is able to sustain similar/higher throughput even with restricted heap sizes
- The CPU / memory overhead of using Java 17 over Java 8 is marginal.

8. FUTURE WORK

1. Explore the way to enable a service (say Cassandra) to use a particular version of JDK without impacting the JDK versions used by other services on a production cluster.
2. Experiment with other GC algorithms of JDK 17 and compare the performance across them.
3. Profile cassandra service to identify the bottleneck step majorly impacting the throughput and performance.
4. Identify a cluster that is impacted by long cassandra GC pauses resulting in the nodes going into degraded mode or getting restarted
5. Check if upgrading to JDK 17 helps prevent degradation of nodes due to long GC pauses.

References

1. (n.d.). Oracle | Cloud Applications and Cloud Platform. Retrieved June 18, 2023, from https://www.oracle.com/
2. Balosin, I. (2019, December 14). *JVM Garbage Collectors Benchmarks Report 19.12 – Ionut Balosin*. Ionut Balosin. Retrieved June 18, 2023, from https://ionutbalosin.com/2019/12/jvm-garbage-collectors-benchmarks-report-19-12/
3. *Java 17 Updates. Since September 2021, Oracle provides… | by Divergent Software Labs*. (2021, October 12). Medium. Retrieved June 18, 2023, from https://medium.com/@emmabrown.divergentsl/java-17-updates-4d407ae2be4a
4. *Java/OpenJDK*. (2023, June 14). endoflife.date. Retrieved June 18, 2023, from https://endoflife.date/java
5. *JDK 11*. (2018, September 25). OpenJDK. Retrieved June 18, 2023, from https://openjdk.org/projects/jdk/11/
6. *JDK 17*. (n.d.). OpenJDK. Retrieved June 18, 2023, from https://openjdk.org/projects/jdk/17/
7. *JVM Garbage Collectors*. (2022, July 4). Baeldung. Retrieved June 18, 2023, from https://www.baeldung.com/jvm-garbage-collectors
8. Knupfer, F. (2022, December 8). *Java garbage collection: What is it and how does it work?* New Relic. Retrieved June 18, 2023, from https://newrelic.com/blog/best-practices/java-garbage-collection
9. *Reasons to move to Java 11 - Azure*. (2023, April 3). Microsoft Learn. Retrieved June 18, 2023, from https://learn.microsoft.com/en-us/java/openjdk/reasons-to-move-to-java-11
10. Salnikov, N. (2013, December 3). *G1 vs CMS vs Parallel GC*. DZone. Retrieved June 18, 2023, from https://dzone.com/articles/g1-vs-cms-vs-parallel-gc
11. Sasidharan, D. K. (2022, April 8). *Does Java 18 finally have a better alternative to JNI?* Okta Developer. Retrieved June 18, 2023, from https://developer.okta.com/blog/2022/04/08/state-of-ffi-java
12. *2022 State of the Java Ecosystem Report*. (2021, August 5). New Relic. Retrieved June 18, 2023, from https://newrelic.com/resources/report/2022-state-of-java-ecosystem
13. Vincent, J. P. (n.d.). *Log Structured Merge Trees. LSM tree is the heart of most storage… | by John Pradeep Vincent | The Startup*. Medium. Retrieved June 18, 2023, from https://medium.com/swlh/log-structured-merge-trees-9c8e2bea89e8
14. Walczak, M. (2022, September 5). *Java 17 Features: All You Need to Know*. Netguru. Retrieved June 18, 2023, from https://www.netguru.com/blog/java-17-features
15. *Deep Dive Into the New Java JIT Compiler - Graal*. (2022, June 7). Baeldung. Retrieved June 18, 2023, from https://www.baeldung.com/graal-java-jit-compiler