Java Garbage Collection

Shalini Mittal
Corporate Trainer

Garbage Collection

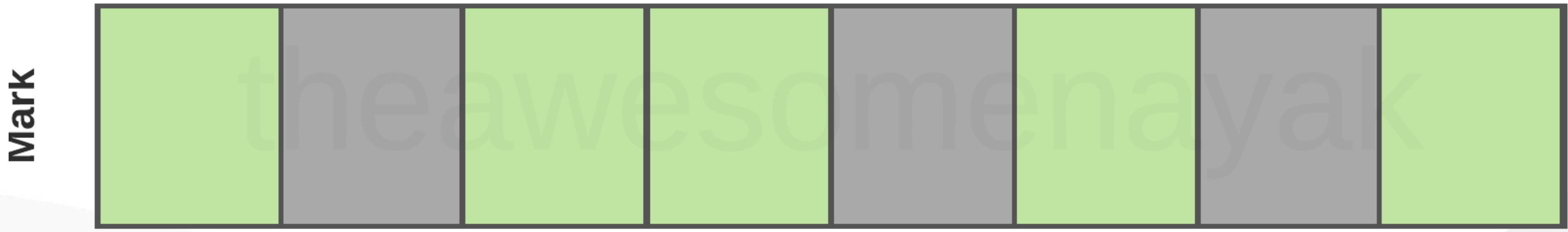
- Garbage Collection (GC) is a cornerstone of Java’s automatic memory management system. It eliminates the need for developers to manually free memory, which reduces the risk of :
    - **Memory Leaks:** This happens when a programmer allocates memory but forgets to free it. Over time, these unfreed chunks of memory accumulate, leading to a gradual reduction in the available memory, which could eventually crash the application or system.
    - **Dangling Pointers:** Sometimes, memory that’s still in use or will be used later gets deallocated prematurely. Accessing such memory spaces can result in undefined behaviors, including application crashes or unpredictable results.
    - **Double Freeing:** It’s an issue where a programmer tries to deallocate an already freed space. This can corrupt the memory and lead to erratic behavior.
    - Other low-level memory errors.
- With the challenges of manual memory management evident, the concept of automated garbage collection was introduced.
- In this system, programmers are only responsible for allocating memory.
- The responsibility of determining when and what memory to free is entrusted to the garbage collector.

Garbage Collection Roots

- Garbage collectors work on the concept of Garbage Collection Roots (GC Roots) to identify live and dead objects.
- Examples of such Garbage Collection roots are:
    - Classes loaded by system class loader (not custom class loaders)
    - Live threads
    - Local variables and parameters of the currently executing methods
    - Local variables and parameters of JNI methods
    - Global JNI reference
    - Objects used as a monitor for synchronization
    - Objects held from garbage collection by JVM for its purposes
- The garbage collector traverses the whole object graph in memory, starting from those Garbage Collection Roots and following references from the roots to other objects
- A standard Garbage Collection implementation involves three phases:
    - Mark
    - Sweep
    - Compact

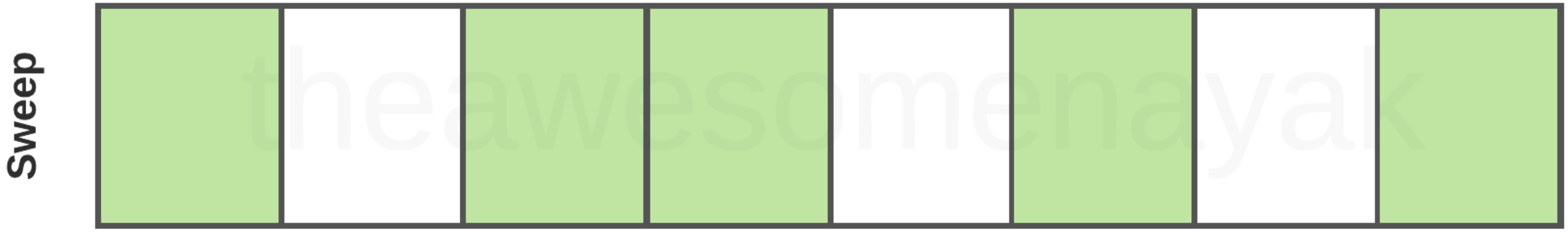
Mark Phase

- Mark objects as alive
- In this step, the GC identifies all the live objects in memory by traversing the object graph.
- When GC visits an object, it marks it as accessible and thus alive. Every object the garbage collector visits is marked as alive. All the objects which are not reachable from GC Roots are garbage and considered as candidates for garbage collection.



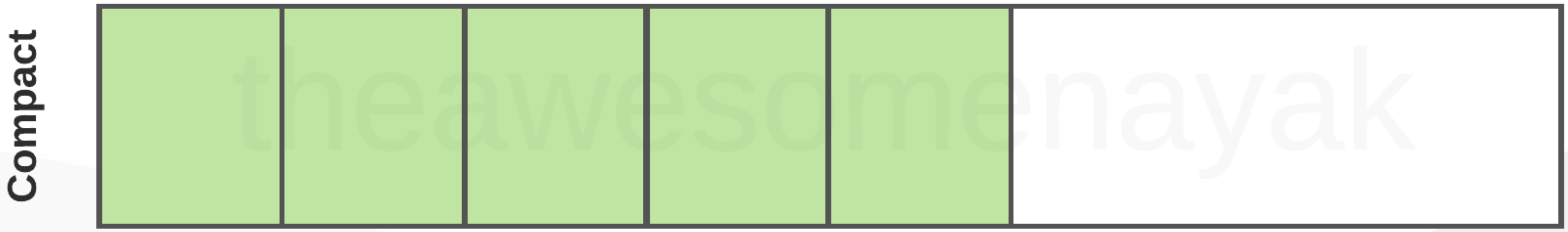
Sweep Phase

- After marking phase, we have the memory space which is occupied by live (visited) and dead (unvisited) objects. The sweep phase releases the memory fragments which contain these dead objects.



Compact Phase

- The dead objects that were removed during the sweep phase may not necessarily be next to each other. Thus, you can end up having fragmented memory space.
- Memory can be compacted after the garbage collector deletes the dead objects, so that the remaining objects are in a contiguous block at the start of the heap.
- The compaction process makes it easier to allocate memory to new objects sequentially.



Generational Garbage Collection

- Java Garbage Collectors implement a generational garbage collection strategy that categorizes objects by age.
- Having to mark and compact all the objects in a JVM is inefficient. As more and more objects are allocated, the list of objects grows, leading to longer garbage collection times. Empirical analysis of applications has shown that most objects in Java are short lived.
- Take the following sequence as an example:
    - Eden has all objects (live and dead)
    - Minor GC occurs - all dead objects are removed from Eden. All live objects are moved to S1 (FromSpace). Eden and S2 are now empty.
    - New objects are created and added to Eden. Some objects in Eden and S1 become dead.
    - Minor GC occurs - all dead objects are removed from Eden and S1. All live objects are moved to S2 (ToSpace). Eden and S1 are now empty.
    - So, at any time, one of the survivor spaces is always empty. When the surviving objects reach a certain threshold of moving around the survivor spaces, they are moved to the Old Generation.

Garbage Collection After

- Once an object becomes unreachable, meaning it is no longer accessible from any GC root, it becomes eligible for garbage collection.
    - Frees up memory occupied by unreachable objects
    - Prevents memory leaks and heap exhaustion
    - Ensures long-running applications continue to operate without manual intervention
- System.gc()
- Example GarbageCollection Project (Then set -Xms300m , VM arg)

Java 8 vs. Java 11: Garbage Collection Differences

- In Java 11, the JVM introduced optimizations to return free memory back to the operating system. While this helps conserve system resources, it can lead to performance degradation, especially if too much memory is released too quickly.
- To prevent this, we can use JVM flags like -Xms to ensure the JVM retains a certain amount of memory:
- This flag keeps the heap from shrinking too aggressively, providing a balance between memory usage and performance.

Garbage Collection Algorithms in the JVM

- JVM has introduced several garbage collection algorithms to meet the diverse needs of Java applications, from simple single-threaded programs to large-scale, low-latency systems.
- Types:
    - Serial Garbage Collector
    - Parallel Garbage Collector (Throughput Collector)
    - Concurrent Mark-Sweep (CMS) Deprecated
    - G1 GC (Garbage First)
    - ZGC (Z Garbage Collector)
    - Shenandoah

Serial GC

- The Serial Garbage Collector uses a single thread to perform all garbage collection tasks, pausing the application during each collection.
- Its simplicity and low overhead make it ideal for single-threaded or small applications where pause times are acceptable.
- Serial GC uses the simple mark-sweep-compact approach for young and old generations garbage collection i.e. Minor and Major GC.
- This collector is best suited for small heaps, stand-alone applications , embedded systems, machines with smaller CPU and development environments where predictable behavior matters more than performance or for small applications with low memory footprint.
- However, it does not scale well to multi-core CPUs or large memory heaps due to its long pause times.
- Enable with: -XX:+UseSerialGC

Parallel GC

- Parallel GC is same as Serial GC except that it spawns N threads for young generation garbage collection where N is the number of CPU cores in the system.
- We can control the number of threads using -XX:ParallelGCThreads=n JVM option
- The Parallel GC, also known as the Throughput Collector, because it uses multiple CPUs to speed up the GC performance. Parallel GC uses a single thread for Old Generation garbage collection.
- It is designed for applications where maximum CPU efficiency and throughput are more important than pause time, such as batch processing systems or compute-heavy services. However, since it still pauses the application during collection, it may not be ideal for latency-sensitive applications.
- Can be used for applications with medium-sized to large-sized data sets that are run on multiprocessor or multithreaded machines.
- Enable with: -XX:+UseParallelGC

- **Parallel Old GC (-XX:+UseParallelOldGC)**: This is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation garbage collection.

G1 GC

- G1 GC is a region-based, low-pause collector designed to balance throughput and pause time predictability.
- G1 is a server-style collector designed for multiprocessor machines with a large amount of memory. The collector tries to achieve high throughput along with short pause times, while requiring very little tuning.
- It divides the heap space into multiple equal-sized heap regions. When a garbage collection is invoked, it first collects region with lesser live data, hence “Garbage First”
- G1 became the default collector starting in Java 9 and is well-suited for server applications, microservices, and systems with moderate to large heaps. It provides a good balance between performance and manageability, although tuning can be more complex than older collectors.
- The G1 collector can achieve high throughput and low latency for applications that meet one or more of the following criteria:
    - Large heap size: Specifically, more than 6GB where more than 50% is occupied by live objects.
    - Rates of allocation and promotion between garbage collection generations that may vary significantly during the application's run.
    - A large amount of fragmentation in the heap.
    - The need to limit pauses to a few hundred milliseconds.

Enable with: -XX:+UseG1GC (default in Java 9+)

G1 String Deduplication

- Starting with JDK 8 update 20, the G1 collector provides another optimization through string deduplication, which could decrease the application's heap use by about 10%.
- The -XX:+UseStringDeduplication compiler option causes the G1 collector to find duplicate strings and keep a single active reference to one string while performing garbage collection on the duplicates.
- No other Java garbage collector currently supports string deduplication.
- Additional G1 compiler options
    - -XX:+UseG1GC enables the G1 garbage collector.
    - -XX:+UseStringDeduplication enables string deduplication.
    - -XX:+PrintStringDeduplicationStatistics prints detailed duplication statistics, if run with the previous option.
    - -XX:StringDeduplicationAgeThreshold=n causes string objects reaching the age of n garbage collection cycles to be considered candidates for deduplication. The default value is 3.

ZGC

- ZGC is a low-latency GC introduced in Java 11, designed to keep pause times consistently below 10 milliseconds, even with heap sizes in the multi-terabyte range.
- It achieves this by performing all heavy GC phases concurrently with the application.
- ZGC is ideal for latency-sensitive, memory-intensive workloads such as real-time analytics, financial systems, and large-scale services.
- It does not perform compaction in the traditional sense, but uses colored pointers and load barriers to relocate objects with minimal impact.
- Platform support includes Linux, Windows, and macOS on x86_64 and AArch64.
- Setting a maximum heap size is very important when using ZGC, because the collector's behavior depends on allocation rate variance and how much of the data set is live. ZGC works better with a larger heap, but wasting unnecessary memory is also inefficient, so you need to tune your balance between memory usage and the resources available for garbage collection.
- You can set the number of concurrent GC through the XX:ConcGCThreads=n compiler option. This parameter determines how much CPU time is given to the garbage collector. By default, ZGC automatically selects how many threads to run, which works for some applications but needs to be tuned for others. Specifying too many threads ends up using a lot of CPU, whereas specifying too few threads causes garbage to be created faster than it can be collected.
- Enable with: -XX:+UseZGC (stable since Java 15)

Shenandaoh

- Shenandoah is another modern low-pause collector, developed by Red Hat, which performs concurrent compaction, a feature ZGC lacks.
- This makes it suitable for systems that require both low pause times and heap defragmentation.
- It reduces pause times by performing more garbage collection work concurrently with the application, including concurrent compaction. Shenandoah's pause time is independent of the heap size. Garbage collecting a 2GB heap or a 200GB heap should have a similar short pause behavior.
- Shenandoah is best suited for low-latency services such as databases, real-time systems, or interactive applications with strict response time requirements. It can introduce slightly higher CPU overhead due to more frequent write barriers and increased GC activity.
- Platform support is currently limited to Linux and Windows on x86_64 and AArch64. Enable with: -XX:+UseShenandoahGC (production-ready in Java 15+)
- It’s mainly supported in: Red Hat builds of OpenJDK (since Shenandoah originated at Red Hat)

Collector	Pause Time	Concurrency	Compaction	Best Use Cases	Platform & JDK Support
Serial GC	High (stop-the-world)	None	Yes	Small apps, single-threaded or embedded systems	All platforms, all JDKs - XX:+UseSerialGC
Parallel GC	Moderate–High	Parallel stop-the-world	Yes	Batch jobs, compute-heavy services, large heaps with loose latency requirements	All platforms, all JDKs - XX:+UseParallelGC
CMS (Deprecated)	Low (some phases concurrent)	Mark & sweep are concurrent	No (leads to fragmentation)	Legacy low-latency systems (Java 8 only)	Removed in Java 14 - XX:+UseConcMarkSweepGC
G1 GC	Low–Moderate	Concurrent marking, mixed mode	Partial (region-based)	General-purpose apps, moderate-latency SLAs	Default in Java 9+ - XX:+UseG1GC
ZGC	Very low (<10ms)	Fully concurrent	No (uses colored pointers)	Real-time, large-heap, low-latency systems	Linux, Windows, macOS (x86_64, AArch64), Java 15+ -XX:+UseZGC
Shenandoah	Very low (heap size independent)	Fully concurrent	Yes (concurrent compaction)	Interactive, real-time systems needing low pause and compaction	Linux, Windows (x86_64, AArch64), Java 15+ - XX:+UseShenandoahGC

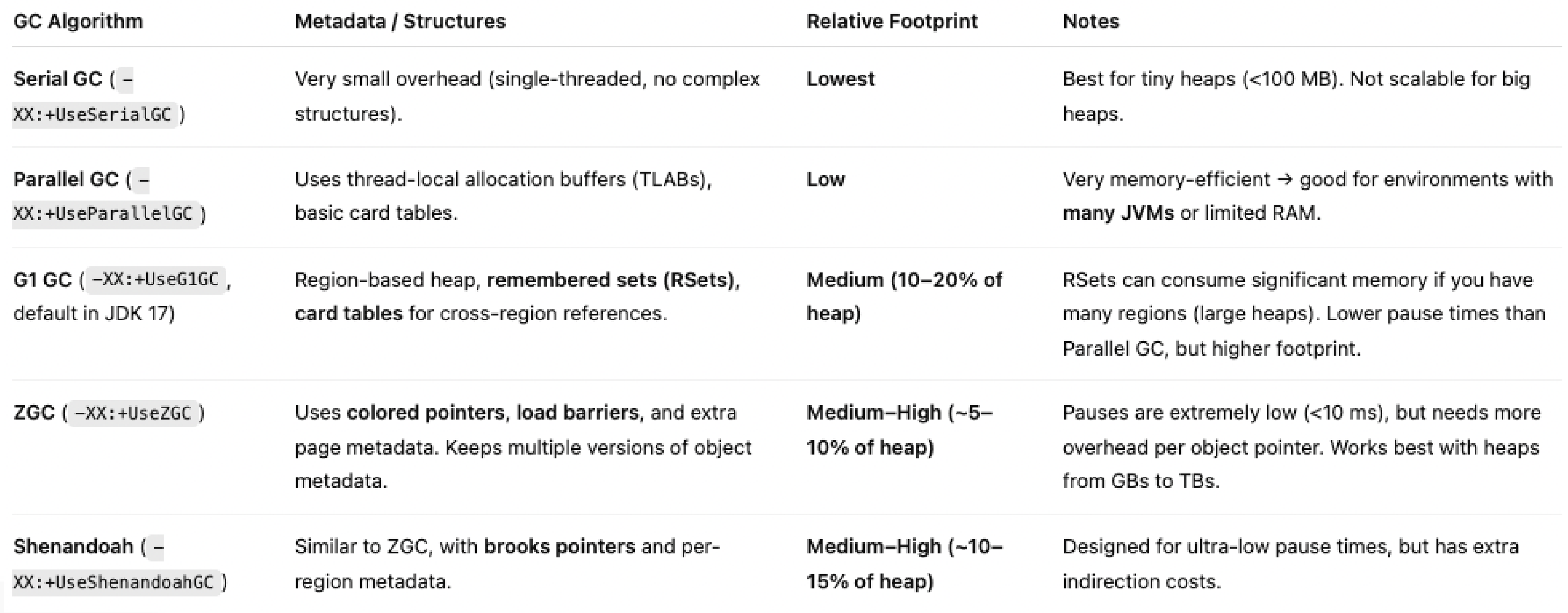
Low Pause vs Low Latency

- **Low Pause Time:**
- This refers specifically to minimizing the duration of "stop-the-world" (STW) pauses during which the application threads are completely halted for garbage collection. Historically, these pauses were a major concern, as they directly impacted application responsiveness and throughput. Modern GC algorithms like ZGC and Shenandoah aim for sub-millisecond or very short pause times, even with large heaps.
- **Low Latency:**
- This is a broader concept encompassing the entire responsiveness of an application, including but not limited to GC pauses. While short GC pauses are a crucial component of achieving low latency, other factors also contribute significantly. These can include:Application-level bottlenecks: Inefficient code, excessive locking, or blocking I/O operations can introduce latency independent of GC.
- JIT compilation overhead: Initial compilation or re-compilation of hot code paths can cause temporary performance dips.
- Operating system scheduling: The way the OS schedules threads and manages resources can impact overall latency.
- Memory allocation patterns: Frequent allocation of short-lived objects can increase GC activity, even with low pause times, potentially impacting overall latency.

Choosing A Garbage Collector

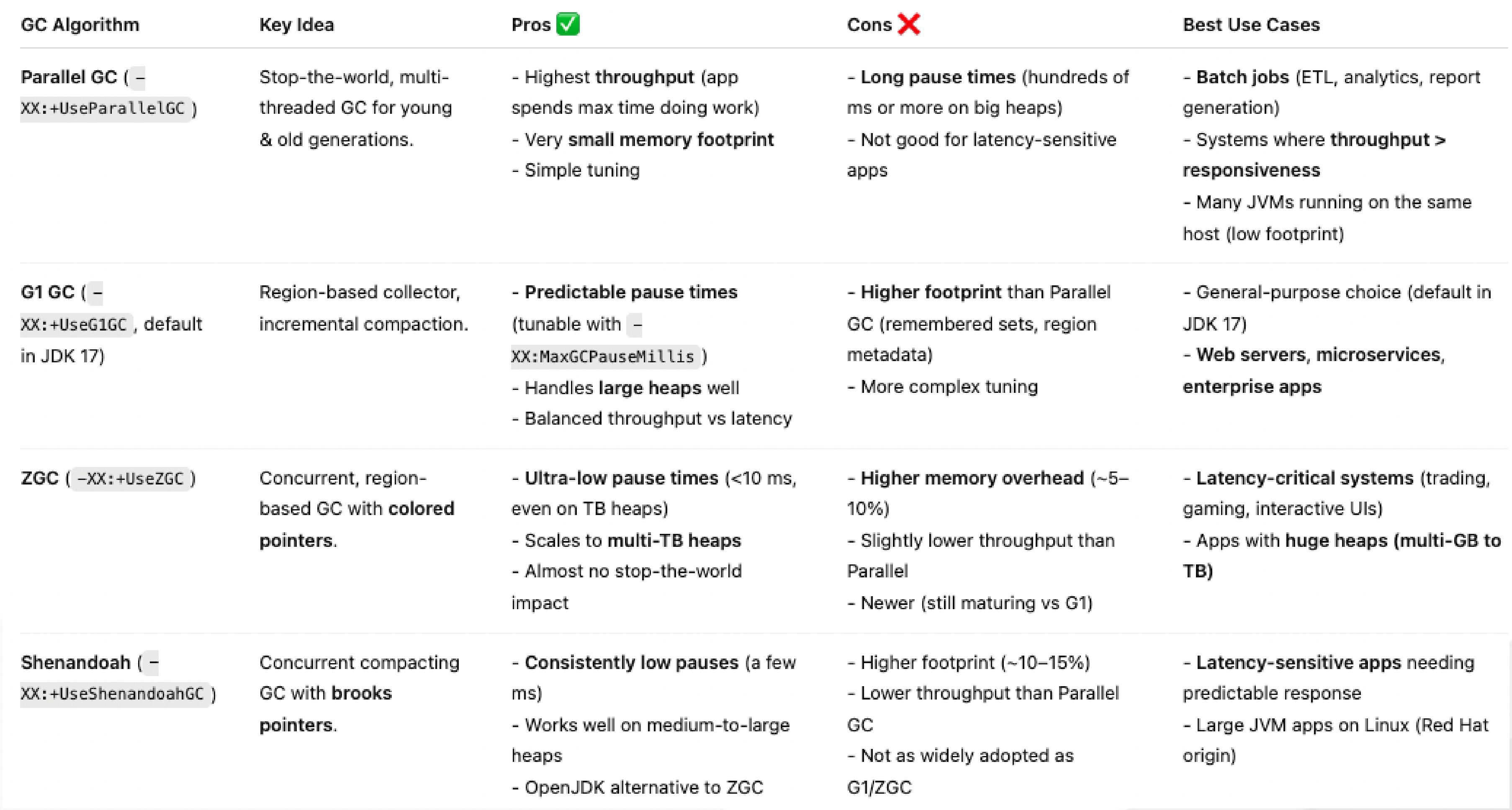
- The essential criteria are:
- **Throughput:**
  The percentage of total time spent in useful application activity versus memory allocation and garbage collection.
  For example, if your throughput is 95%, that means the application code is running 95% of the time and garbage collection is running 5% of the time. You want higher throughput for any high-load business application.
- **Latency:**
  Application responsiveness, which is affected by garbage collection pauses. In any application interacting with a human or some active process (such as a valve in a factory), you want the lowest possible latency.
- **Footprint:**
  The working set of a process, measured in pages and cache lines.
- Different users and applications have different requirements.
  - Some want higher throughput and can bear longer latencies in exchange, whereas others need low latency because even very short pause times would negatively impact their user experience.
  - On systems with limited physical memory or many processes, the footprint might dictate scalability

FootPrint Comparison



GC Algorithm	Metadata / Structures	Relative Footprint	Notes
Serial GC (`–XX:+UseSerialGC`)	Very small overhead (single-threaded, no complex structures).	**Lowest**	Best for tiny heaps (<100 MB). Not scalable for big heaps.
Parallel GC (`–XX:+UseParallelGC`)	Uses thread-local allocation buffers (TLABs), basic card tables.	**Low**	Very memory-efficient → good for environments with **many JVMs** or limited RAM.
G1 GC (`–XX:+UseG1GC` , default in JDK 17)	Region-based heap, **remembered sets (RSets)**, **card tables** for cross-region references.	**Medium (10–20% of heap)**	RSets can consume significant memory if you have many regions (large heaps). Lower pause times than Parallel GC, but higher footprint.
ZGC (`–XX:+UseZGC`)	Uses **colored pointers**, **load barriers**, and extra page metadata. Keeps multiple versions of object metadata.	**Medium–High (~5–10% of heap)**	Pauses are extremely low (<10 ms), but needs more overhead per object pointer. Works best with heaps from GBs to TBs.
Shenandoah (`–XX:+UseShenandoahGC`)	Similar to ZGC, with **brooks pointers** and per-region metadata.	**Medium–High (~10–15% of heap)**	Designed for ultra-low pause times, but has extra indirection costs.

GC Java 17 Comparison



GC Algorithm	Key Idea	Pros ✅	Cons ❌	Best Use Cases
Parallel GC (`–XX:+UseParallelGC`)	Stop-the-world, multi-threaded GC for young & old generations.	- Highest **throughput** (app spends max time doing work) - Very **small memory footprint** - Simple tuning	- **Long pause times** (hundreds of ms or more on big heaps) - Not good for latency-sensitive apps	- **Batch jobs** (ETL, analytics, report generation) - Systems where **throughput > responsiveness** - Many JVMs running on the same host (low footprint)
G1 GC (`–XX:+UseG1GC` , default in JDK 17)	Region-based collector, incremental compaction.	- **Predictable pause times** (tunable with `–XX:MaxGCPauseMillis`) - Handles **large heaps** well - Balanced throughput vs latency	- **Higher footprint** than Parallel GC (remembered sets, region metadata) - More complex tuning	- General-purpose choice (default in JDK 17) - **Web servers, microservices, enterprise apps**
ZGC (`–XX:+UseZGC`)	Concurrent, region-based GC with **colored pointers**.	- **Ultra-low pause times** (<10 ms, even on TB heaps) - Scales to **multi-TB heaps** - Almost no stop-the-world impact	- **Higher memory overhead** (~5–10%) - Slightly lower throughput than Parallel - Newer (still maturing vs G1)	- **Latency-critical systems** (trading, gaming, interactive UIs) - Apps with **huge heaps (multi-GB to TB)**
Shenandoah (`–XX:+UseShenandoahGC`)	Concurrent compacting GC with **brooks pointers**.	- **Consistently low pauses** (a few ms) - Works well on medium-to-large heaps - OpenJDK alternative to ZGC	- Higher footprint (~10–15%) - Lower throughput than Parallel GC - Not as widely adopted as G1/ZGC	- **Latency-sensitive apps** needing predictable response - Large JVM apps on Linux (Red Hat origin)

Garbage Collection Logs

- Enable detailed GC logging and direct the output to a specified file Java 8:
  -XX:+PrintGCDetails -Xloggc:<file-path>
- -XX:+PrintGCDetails: Enables detailed GC information to be printed.
- -Xloggc:<file-path>: Specifies the path to the file where the GC logs will be written. If no file path is provided, the logs are printed to standard error (STDERR).
- Java 9 introduced a unified logging system, and the -Xlog option is now used for GC logging:
  -Xlog:gc*:file=<file-path>
- -Xlog:gc*: Configures logging for all GC-related messages.
- file=<file-path>: Specifies the path to the file where the GC logs will be written.
- Java 8:
  java -cp $CLASSPATH -XX:+PrintGCDetails -Xloggc:/tmp/gc.log MyMainClass
- Java 11:
  java -cp $CLASSPATH -Xlog:gc*:file=/tmp/gc.log, filecount=10,filesize=10m main.Main

Garbage Collection Logs Example

- Create a Main file in SoftLeak Project
  Run Configurations: -Xmx10m -Xlog:gc*
  [time][level][tags] EventID Phase (Cause) (Action) Before→After(Total) Duration
- To change GC type diable the default an dspecify the one to be used:
  //-Xmx5m -Xlog:gc* -XX:-UseG1GC -XX:+UseConcMarkSweepGC -XX:UseParallelGC

[Detail with different versions of java](https://sematext.com/blog/java-garbage-collection-logs/)
https://sematext.com/blog/java-garbage-collection-logs/

Garbage Collection Logs Detail

```
[207.610s][info][gc,start ] GC(0) Pause Young (Normal) (G1 Evacuation Pause)
```

Log Part	Example	Meaning
Timestamp	[0.024s]	Time since JVM started (uptime).
Level	[info]	Log level (debug, info, warning).
Tag(s)	[gc], [gc,heap], [gc,start]	Category of the log line: - gc → summary - gc,start → GC started - gc,heap → heap usage info - gc,marking → concurrent marking phase
Event ID	GC(0)	GC event sequence number. Increments each GC.
Phase / Action	Pause Young (Normal) (G1 Evacuation Pause)	What kind of GC happened: - Pause Young = Young gen GC - Pause Full = Full GC (old + young) - (Normal) = reason/cause - (G1 Evacuation Pause) = specific action by G1
Heap Before → After (Total)	20M->5M(50M)	Heap usage: - 20M = used before GC - 5M = used after GC - (50M) = total committed heap
Pause Time	4.123ms	Time application threads were stopped for GC.

Garbage Collection Logs Example

- **Example 1: Minor GC**

  ```
  [0.024s][info][gc] GC(0) Pause Young (Normal) (G1 Evacuation Pause) 20M->5M(50M) 4.123ms
  ```

  - At **0.024s uptime**, a **young generation GC** happened.
  - Freed memory: 20M → 5M (total heap 50M).
  - Pause duration: 4.1 ms.
- **Example 2: Full GC**

  ```
  [2.543s][info][gc] GC(4) Pause Full (G1 Compaction Pause) 150M->75M(256M) 35.678ms
  ```

  - At **2.5s uptime**, a **full GC** happened.
  - Freed memory: 150M → 75M (heap 256M).
  - Pause: 35.6 ms (longer because it's full GC).
- **Example 3: Concurrent Phase**

  ```
  [2.000s][info][gc,marking] GC(3) Concurrent Mark Cycle [2.050s][info][gc,marking] GC(3) Concurrent Mark Cycle 50.00ms
  ```

  - G1 started a **concurrent marking cycle** at 2.0s.
  - Finished in 50 ms, mostly in background (app keeps running).

Additional Flags

- Additional Flags (Applicable to both versions with minor syntax differences in Java 9+):
- -XX:+PrintGCDateStamps: Adds a timestamp with date and time to each log entry.
- -XX:+PrintGCTimeStamps: Adds a timestamp indicating the time elapsed since JVM startup (in seconds).
- -XX:+UseGCLogFileRotation: Enables rotation of GC log files.
- -XX:NumberOfGCLogFiles=<number>: Specifies the number of GC log files to retain during rotation.
- -XX:GCLogFileSize=<size>: Sets the maximum size of a single GC log file before rotation (e.g., 10m for 10 megabytes).
- -XX:NewRatio
- -XX:SurviorRatio
- -XX:MaxTenuringThreshold

New Ratio

- -XX:NewRatio=n
- Set the ratio of Old Generation size to Young Generation size" in the heap.
- The formula is, OldGenSize : YoungGenSize = n : 1
- -XX:NewRatio=2, means old Gen is 2 times bigger than Young Gen.

Option	Old Gen %	Young Gen %
-XX:NewRatio=1	50%	50%
-XX:NewRatio=2	66%	33%
-XX:NewRatio=4	80%	20%
-XX:NewRatio=8	88.9%	11.1%

```
jps
jinfo –flag NewRatio <pid>
```

Survivor Ratio

- -XX:SurvivorRatio=n
- It set the ratio of Eden space size to a single Survivor space size in the Young Generation.
- -XX:SurvivorRatio=8 → means, eden is 8 times bigger than a single Survivor space.

SurvivorRatio	Eden % of Young Gen	Each Survivor %
4	66.7%	16.7%
6	75%	12.5%
8 (default)	80%	10%
10	83.3%	8.3%

```
jps
jinfo –flag SurvivorRatio <pid>
```

Max Tenuring Threshold

- -XX:MaxTenuringThreshold=n
- It set the maximum number of minor GCs an object can survive in the Young Generation before being promoted to the Old Generation.
- How it works
    - When objects are created, they go into Eden.
    - If they survive a Minor GC, they move into a Survivor space and get an age counter (tenuring age) incremented.
    - Each Minor GC that they survive increases their age by 1.
    - Once their age ≥ MaxTenuringThreshold, they are promoted (moved) to the Old Generation.

```
jps
jinfo –flag MaxTenuringThreshold <pid>
```

Use Adaptive Size Policy

- -XX:-UseAdaptiveSizePolicy
- It turn OFF the adaptive sizing policy for heap memory regions.
- Heap space sizes will not be adjusted automatically.
- By default, most modern JVMs dynamically adjust the sizes of the Young Generation, Old Generation, and Survivor spaces during runtime to optimize throughput and pause times.
- Sizes will remain fixed as per your startup parameters (like - Xmn, -XX:NewRatio, -XX:SurvivorRatio).
- Why might you disable it?
  - ❑ You want predictable heap space division for performance testing or benchmarking.
  - ❑ You have manually tuned the heap sizes and don’t want the JVM to change them.
  - ❑ You are diagnosing GC issues and want a stable configuration without runtime adjustments.
  - ❑ For example,
    java -Xms512m -Xmx512m -Xmn256m -XX:-UseAdaptiveSizePolicy CustomerApplication
  - ❑ Young Gen = fixed 256 MB • Old Gen = fixed 256 MB (since total heap is 512 MB)

GC Tuning JVM

- java -Xmx120m -Xms30m -Xmn10m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar demo.jar
- https://bell-sw.com/blog/guide-to-jvm-memory-configuration-options/#mcetoc_1gv0shv8715v
- *-Xms* sets the initial heap size for the JVM.
- *-Xmx* sets the maximum heap size for the JVM.
- *-Xss* sets the size of the thread stack for each thread's internal use.
- -XX:+UseCompressedOops enables the use of compressed object pointers to reduce memory usage.
- *-XX:+UseThreadPriorities* instructs the JVM to use native thread priorities.
- *-XX:PermSize* sets the initial size of the garbage collector's Permanent Generation space.
- *-XX:MaxPermSize* sets the maximum size of the garbage collector's permanent generation space.
- *-XX:NewSize* sets the initial size of the Young Generation space.
- *-XX:MaxNewSize* sets the maximum size of the Young Generation space.
- *-XX:SurvivorRatio* sets the ratio of Eden space to Survivor space.
- *-XX:MaxTenuringThreshold* sets the maximum age for objects in the Survivor space.

GC Tuning JVM

- *-XX:+UseParNewGC* instructs the JVM to use the new parallel generation garbage collector.
- *-XX:+UseSerialGC* instructs the JVM to use the serial garbage collector.
- *-XX:+UseG1GC* instructs the JVM to use the Garbage First (G1) garbage collector.
- *-XX:+UseZGC* instructs the JVM to use the ZGC garbage collector.
- *-XX:+HeapDumpOnOutOfMemoryError* tells the JVM to create a heap dump file when an OutOfMemoryError occurs.
- *-XX:HeapDumpPath* provides a custom path for the JVM to write the contents of the heap during a heap dump.
- *-Djava.library.path* enables you to specifiy paths to native libraries needed at runtime.
- *-Duser.timezone* enables you to set a custom timezone for the JVM.
- *-XX:+PrintGCDetails* instructs the JVM to print out detailed garbage collection logs to help you with GC optimization.
- *-XX:+PrintFlagsFinal -version* prints out all of the currently configured flags and options set on your JVM

GC Tuning Takeaway

- GC tuning is not one-size-fits-all. The ideal configuration depends on:
    - Application type (batch job vs real-time service)
    - Heap size
    - Allocation rate
    - Pause time sensitivity
    - Number of CPU cores
- For example:
- A microservice handling short-lived HTTP requests might prioritize low pause times and benefit from G1 or ZGC.
- A batch processing application may favor throughput and perform well with Parallel GC.
- An interactive desktop GUI might suffer from GC pauses and require careful heap sizing and collector selection.

ThankYou

shalini06mittal@gmail.com

7738460004