International Journal of Management, IT & Engineering
Vol.14 Issue 10, October 2024,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gate as well as in Cabell's Directories of Publishing Opportunities, U.S.A

Enhancing Fintech Microservices Performance with GemFire: A Comprehensive Analysis of Caching Strategies

Sumit Bhatnagar*
Roshan Mahant**

Abstract

In the competitive and dynamic world of financial technology (fintech), high performance and low latency are crucial for real-time transactions and data processing. Microservices architecture, combined with efficient caching mechanisms, significantly enhances scalability and responsiveness. VMware GemFire, an in-memory data grid, provides robust caching solutions that, when paired with Java 17's Z Garbage Collector (ZGC) and heap Least Recently Used (LRU) eviction strategies, can maximize cache performance. This paper proposes a system integrating GemFire with ZGC to optimize memory management, reduce latency, and maintain high throughput. The heap LRU eviction strategy ensures that the least recently used data is evicted first when memory limits are reached, maintaining high cache hit rates and reducing the risk of the JVM running out of memory. By adjusting ZGC and using heap LRU eviction, this combined method makes it easier for microservices to handle changing workloads, grow horizontally, and speed up data access, which improves the overall performance and reliability of the system. Different settings for SoftMaxHeapSize have different effects on heap usage, operation throughput, and garbage collection performance. The results of the experiments show how to tune GemFire cache performance to work best in a microservices environment.

Keywords:

GemFire; Microservices Architecture; Caching Strategies; Fintech;

Author correspondence:

Sumit Bhatnagar
Vice President of Software Engineering, New Jersey, USA
Sumit.bhatnagar@outlook.com

Roshan Mahant
Senior Software Consultant, Texas, USA
roshanmahant@gmail.com

1. Introduction

In the ever-evolving landscape of financial technology (fintech), where speed, reliability, and scalability are paramount, the performance of applications is critical. Fintech solutions often deal with real-time financial transactions, complex data analytics, and user interactions that demand instantaneous responses. The microservices architecture has become the cornerstone of modern fintech applications due to its ability to decompose large systems into smaller, manageable, and independently deployable services.

However, this architectural approach, while providing flexibility and scalability, also introduces new challenges, particularly in maintaining high performance across

distributed systems. To address these challenges, effective caching strategies are essential. Caching reduces the latency of data retrieval operations by storing frequently accessed data closer to the application, thus minimizing access times and reducing load on backend systems. One of the leading technologies in this domain is GemFire, an in-memory data grid that provides powerful caching capabilities.

Fintech applications, with their demanding requirements, find GemFire ideal due to its high scalability and low latency design. GemFire's architecture supports a variety of caching strategies, such as read-through and write-through caching, write-behind caching, near caching, and distributed caching, each serving different purposes and scenarios. Read-through and write-through caching ensure data consistency by synchronously interacting with the underlying data sources, making them suitable for transaction processing. Write-behind caching, on the other hand, decouples write operations from the immediate database writes, thus improving write performance by performing these operations asynchronously. Near caching reduces latency further by keeping a local cache close to the application layer, which is particularly beneficial for real-time analytics and reporting. Distributed caching leverages the power of horizontal scaling by partitioning data across multiple nodes, thereby balancing load and enhancing fault tolerance. The integration of GemFire's caching strategies into fintech microservices can revolutionize application performance. Transactional caching, for example, can ensure reliable and consistent transaction processing in real-time using read-through and write-through strategies.

Implementing near caching significantly accelerates analytics and reporting, enabling faster data retrieval and more responsive dashboards. Distributed caching efficiently manages user sessions, ensuring scalability and high availability even under peak loads. (Supriyanto & Ismawati, 2019). Additionally, by asynchronously managing log entries, write-behind caching can optimize audit logging, meeting compliance requirements without compromising performance.

2. Theoretical Study

These days, technology is an important part of almost everything people do. The rapid growth of information and communication technology has brought about significant changes in various areas, including the economic, social, and other domains discussed by Bernardus RedikaWestama Putra and EvangsMailoa. So fast (Ngafifi, 2014), microservices are being added to fintech apps using the Express JS 560 framework. In this era of technology, the financial sector is also evolving in a more practical and modern manner (Rahadiyan& Sari, 2019). It is critical right now to provide technological innovation and use it in business (Supriyanto & Ismawati, 2019).

Businesses are starting to transform their operations through the use of technology. Businesses require these changes to maintain their competitiveness. We must transform the challenge of technology growth into an opportunity, as it offers numerous advantages (Darman, 2019). As technology changes quickly, new financial apps have come out that combine technology with financial systems. These are called financial technologies. FinTech, a type of digital technology, assists in resolving public money issues. Currently, Fintech possesses a wide range of capabilities and is experiencing rapid growth. Fintech can now take care of many things, like e-money, loans, fundraisers, payments, and more (Muhamad Rizal, Erna Maulina, 2018).

Due to businesses transitioning to digital platforms, a plethora of websites and mobile apps have emerged, enabling seamless business transactions and payments, regardless of time or location, provided that the computers and phones remain connected. The scale of the application will increase as the business expands. According to Asfifah and Setiaji (2019), Rest API (Representational State Transfer) is a software development

method that follows specific guidelines for creating services. When you send or receive data in the form of JSON, you always use the HTTP protocol (Rajagukguk, 2018). We often build an application using this method. The microservice-based application we are currently building is one example. Using Rest provides numerous benefits. For instance, it becomes simpler to modify the system, enabling faster and more efficient data sharing and transmission.

A web service is a piece of software that lets two different programs talk to each other over the internet. HTTP is the network that most web services use. Web services, which are public applications, also enable clients to receive or use data (Perwira & Santosa, 2017). Web services also provide mechanisms for inter-web service communication. On the web service, the URL, also known as an endpoint, contains the necessary data and instructions, such as "Get" and "Post." Web services enable clients to share data regardless of the type of database or system they utilize. These advantages have led to a surge in the use of computer services in recent times.

Developers have increasingly used microservice design in the past few years, developers have been using microservice design more and more, which has grown along with software architecture . Microservices are a type of architecture that breaks up big systems into smaller functional parts to make them more modular (Karabey Aksakalli, Çelik, Can, &Tekinerdoğan, 2021). Microservices enable developers to quickly and easily create software due to their freedom. Monolithic architecture's inability to effectively manage system failures contributes to the emergence of microservices. This is because a monolithic architecture application will have only one point of failure if one service fails or an error happens.

3. Proposed Systems

To maximize GemFire cache performance with ZGC and heap LRU eviction in the context of microservices, the integration focuses on enhancing scalability, responsiveness, and resource efficiency. Microservices architectures benefit significantly from efficient caching mechanisms like GemFire's distributed caching, which stores frequently accessed data in memory across multiple microservices instances. By leveraging Java 17's Z Garbage Collector (ZGC), microservices can manage memory more effectively, minimizing pauses and optimizing garbage collection cycles to maintain consistent performance.

The heap LRU eviction strategy further ensures that the cache maintains high hit rates by evicting the least recently used data when memory limits are reached, thereby reducing latency and enhancing overall system throughput. This integrated approach improves microservices' ability to handle variable workloads and scale horizontally, but it also enhances reliability and responsiveness by reducing database load and improving data access times across distributed environments.

```
import org.apache.geode.cache.*;
import org.apache.geode.cache.eviction.*;

public class GemFireConfig {
    public static void main(String[] args) {
        CacheFactory cacheFactory = new CacheFactory();
        cacheFactory.set("name", "GemFireCache");
        Cache cache = cacheFactory.create();

        RegionFactory<String, String> regionFactory = cache.createRegionFactory(RegionShor
        regionFactory.setEvictionAttributes(
            EvictionAttributes.createLRUHeapAttributes(
                null, EvictionAction.LOCAL_DESTROY)
        );
        Region<String, String> region = regionFactory.create("exampleRegion");

        // Populate the region with data
        for (int i = 0; i < 1000; i++) {
            region.put("key" + i, "value" + i);
```

Fig.1 Configuration of System

Algorithm

Initialize System: Configure GemFire for distributed caching.
Set Java 17 JVM options for ZGC:
-Xmx H_{max} (Maximum heap size)
-XX:SoftMaxHeapSize S_{max}
Define GemFire eviction threshold (EthE_{th}Eth).
Pre-populate Cache:
Load initial data into the GemFire cache.
Ensure long-lived heap usage is approximately
Hmax, where α is a constant, e.g., 0.4 (40%).
Start Monitoring:
Continuously monitor heap usage and eviction metrics.
Eviction Headroom Calculation:
Calculate Eviction Headroom:
Eviction Headroom=Eth−Smax
Heap Usage Calculation:
Calculate Heap Usage (Husage
Husage=Lset+G
Where:
Lset is the Live Set Size. –
G is the amount of Garbage.
Garbage Collection and Memory Management:
If Husage>Smax
Trigger ZGC garbage collection.
Eviction Management:

If Husage>Eth
Evict least recently used (LRU) entries from the cache.
Performance Tuning Loop:
Adjust Smax and Eth based on observed performance:
Increase Smax if frequent evictions occur. –
Decrease Smax if garbage collection is too frequent
Finalize settings when optimal performance is achieved.

LRU Eviction Algorithm

Description: When the cache reaches its capacity, LRU eviction operates on the principle of evicting the least recently used items first. It maintains a record of usage for each item and removes the item that hasn't been accessed for the longest time when space is needed for new items.

Tracking Usage: A timestamp or counter identifies the last access time for each item in the cache.

Let Ti represent the timestamp or counter value for item i.

Eviction Decision: When the cache reaches its capacity and a new item j needs to be added: Calculate T_{LRU}, the minimum T_i among all items currently in the cache.Evict the item i
where
$\mathbf{T_i=T_{LRU}}$.
Example Scenario:
Suppose the cache has capacity C and is currently holding items i_1,i_2,...,in} with their respective timestamps {T_{i1},T_{i2},...,T_{in}}
When a new item j is requested to be added:
If n <C < add j directly.
If n=Cn identify I
Where Ti=min(T$_{i1}$,T$_{i2}$,...,Ti$_n$) and replace i with j.

Heap LRU Eviction Works

Heap LRU eviction is an algorithm for maintaining cache performance while protecting against the risk of the JVM running out of memory. In VMware GemFire, heap LRU eviction works like this: GemFire continually monitors heap usage. When heap usage exceeds a user-configured threshold, GemFire evicts eligible entries from memory until heap usage falls back below the threshold. Every entry evicted from memory increases the chance of a cache miss, which can reduce cache performance. To maintain cache performance, GemFire tries to evict the entries that are the least likely to be used in the near future. LRU eviction selects entries to evict based on the assumption that the least recently used entries are the least likely to reappear in the near future. When the workload satisfies this assumption, evicting the least recently used entries minimizes the chance of a cache miss. GemFire's heap LRU eviction algorithm relies on the JVM's garbage collector to very quickly collect the memory used by evicted entries. Evicting an entry does not, all by itself, make the entry's memory available for allocation. It merely makes the object and its memory "unreachable." This unreachable memory becomes available for allocation

only when the garbage collector collects it. Heap usage remains high until the garbage collector collects the memory from evicted entries.

ZGC Decides When to Collect Garbage

ZGC Goals. ZGC works to ensure that any thread that requests memory can get it with minimal delay. If an application thread attempts to allocate more memory than is currently available, ZGC pauses that thread until a garbage collection completes. This pause is called an *allocation stall.* ZGC works very hard to avoid allocation stalls, and to do this with minimal impact on application performance.

ZGC Decision Rules. Ten times per second, ZGC samples the application's heap usage and memory allocation rate, then applies seven rules to decide whether to initiate garbage collection. One rule, the High Usage rule, checks whether heap usage is above ZGC's target maximum heap usage or is close enough to the target to cause concern. Another, the Allocation Rate rule, predicts whether the application is likely to run out of available heap memory if ZGC does not intervene immediately.

Tuning ZGC for Use with Heap LRU Eviction

When tuned for this purpose, ZGC is well suited for use with heap LRU eviction. To tune ZGC well, you will need to know some key characteristics of your workload and the key tuning knobs at disposal.

Workload heap usage. To tune ZGC well, you will need to know several key characteristics of your workload's heap usage:

- **Long-lived heap usage:** The amount of heap that GemFire requires in order to hold cached data in memory. This includes the memory used for the data's keys and values, plus the data structures that GemFire uses to maintain the data, plus other long-lived data structures that GemFire uses in order to present its services. Long-lived heap usage does not include the short-lived objects that GemFire uses to perform a particular operation.

- **Live set size:** The amount of heap used by all live objects. This includes long-lived objects and any short-lived objects currently in use. Over time, GemFire'sZHeap Collection Used Memory statistic gives an approximation of live set size.

ZGC tuning knobs. Java offers two key JVM options to tune ZGC for use with heap LRU eviction:

- **Xmx:** The JVM's maximum heap size. For a given workload, a larger heap size reduces the chance of allocation stalls, and allows ZGC to work efficiently with fewer worker threads.

- **XX:Soft Max Heap Size:** ZGC's "soft" limit for maximum heap usage. ZGC will strive to keep heap usage below this limit, but may allow heap usage to exceed it when necessary. As I will show, setting SoftMaxHeapSize lower reduces the risk of eviction, but makes garbage collections more frequent and less CPU-efficient.

Setting it higher reduces ZGC's CPU consumption, but increases the risk of eviction.

GemFire tuning knobs. GemFire's primary tuning knob for governing heap LRU eviction is:

- **Eviction-heap-percentage:** GemFire's target heap usage threshold, expressed as a percentage of max heap size. Whenever heap usage exceeds this threshold, GemFire evicts entries to bring heap usage down.

Experimenting with SoftMaxHeapSize

To understand how SoftMaxHeapSize affects heap usage, operation throughput, and garbage collection performance, I ran a series of scenarios on a GCP instance with 16 CPUs. Each scenario:

- Starts a GemFire server with max heap size (-Xmx) set to 32g and with GemFire's eviction threshold set to 60%.
- Pre-populates a set of heap LRU regions with enough total data to bring long-lived heap usage to about 40% of max heap size. The data consisted of 1,205,264 total entries, each holding a 10000 byte array. (Actual measured long-lived heap usage was 40.5%.)
- Runs 16 threads to perform as many updates as possible for 2 minutes. Each update replaces a randomly selected value in the cache with a new value of the same size (a 10000 byte array). This sustained updates phase generates a great deal of garbage (about 2g per second) while keeping long-lived heap usage essentially constant.varied SoftMaxHeapSize from 40%, just below long-lived heap usage, to 70%, well above the eviction threshold.

Run these scenarios as experiments, not as benchmarks. Each scenario uses 16 client threads running in a separate JVM but on the same GCP instance as the GemFire server. Additionally, several other minor processes coordinate the experiments. We should not take the results as absolute measures of performance, but rather as general effects and trends. These scenarios generate an unusually uniform workload. In a production environment, the workload will be far more variable.

Garbage production rate. In these scenarios, the sustained update phase allocates memory at a rate of about 2000 MB/s. Given the nature of the scenarios, every allocation results in corresponding garbage. Some allocations are for new values that will live in the cache but replace existing values, making the old values unreachable. The remaining allocations are for short-lived objects that will become unreachable as soon as they complete their role in the operation. This means that the measured allocation rate is the same as the garbage production rate. Every 16 seconds or so, each scenario generates a full heap worth of garbage (32 g).

4. Performance Matrix

Eviction headroom is the difference between the eviction threshold and the SoftMaxHeapSize.

Eviction Headroom=Eviction Threshold−SoftMaxHeapSize

SoftmaxHeapSize and Heap Usage

Heap usage is managed by setting an appropriate SoftMaxHeapSize. The heap usage can be modeled as follows:

Heap Usage=Live Set Size+Garbage
Where:
Live Set Size is the memory used by all live objects.
Garbage is the memory allocated by objects that will be collected by the garbage collector.

Patterns of Heap Usage

Insufficient Collection Headroom: If SoftMaxHeapSize is too low, ZGC will collect garbage continuously:

Collection Headroom=SoftMaxHeapSize−Live Set Size

If Collection Headroom is negative, ZGC will struggle to keep up with garbage collection.

Insufficient Eviction Headroom: If SoftMaxHeapSize is above the eviction threshold:

Eviction Headroom=Eviction Threshold−SoftMaxHeapSize

If Eviction Headroom is negative, frequent evictions will occur, leading to cache misses.

Sufficient Collection and Eviction Headroom: Both collection and eviction headroom should be positive:

Collection Headroom>0
Eviction Headroom>0

Throughput Analysis: Throughput can be affected by the SoftMaxHeapSize. As SoftMaxHeapSize increases, throughput improves due to reduced garbage collection overhead.

Throughput:

$$\text{Throughput} = \frac{Total\ Operations}{Time}$$

CPU Availability:

Available CPUs=Total CPUs−CPUs Used for GC

SoftMax HeapSize Affects Heap Usage

The Heap Usage graph shows the minimum and maximum heap usage during the sustained updates phase of each scenario, as measured by ZGC’s memory manager:

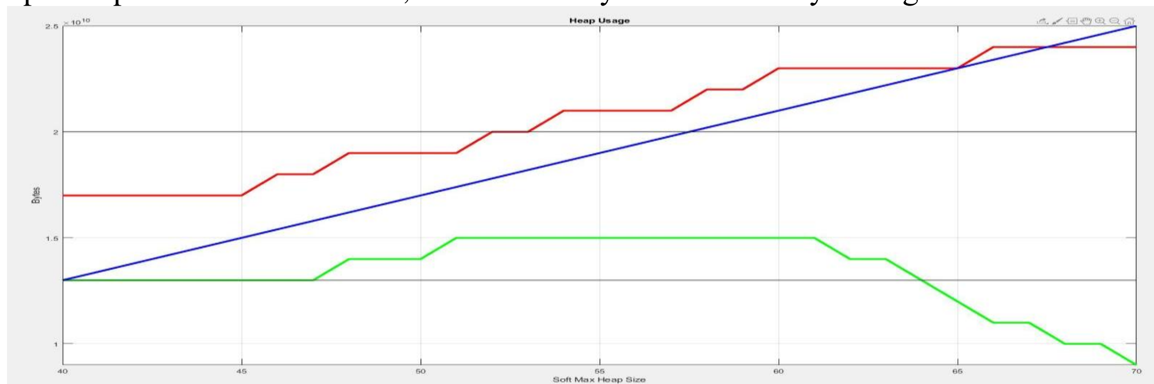

Fig.2 Heap usage

The colored zones highlight different patterns of performance. In the green zone, ZGC is able to keep heap usage below or near its target SoftMaxHeapSize, and the system is able to avoid eviction. Despite constantly collecting garbage and consuming every CPU cycle available to it, ZGC is unable to meet its SoftMaxHeapSize target in the orange zone. In the red zone, despite ZGC keeping heap usage below or near SoftMaxHeapSize,

GemFire evicts many entries from memory. Two key factors govern which performance pattern the system exhibits:

- **Collection headroom:** The difference between SoftMaxHeapSize and the live-set size.
- **Eviction headroom:** The difference between the eviction threshold and SoftMaxHeapSize.

Even in the green zone, ZGC may need to perform many expensive garbage collections to keep heap usage at or below SoftMax HeapSize. Later, we will see that as SoftMax HeapSize increases from the left side of the green zone to the right, ZGC's performance improves steadily, consuming fewer and fewer CPU cycles.

Collection Headroom

ZGC tries to keep heap usage below SoftMaxHeapSize. ZGC can meet this goal when it has enough collection headroom, as shown in the green and red zones on the graph. In the orange zone, collection headroom is too low. Given the rate of garbage production and the live objects maintaining the cache and performing operations, ZGC cannot collect garbage fast enough to keep heap usage below SoftMax HeapSize. The first scenario sets SoftMaxHeapSize to 40% of the maximum heap size. This is not only below the live-set size (approximated by the green line), but also below long-lived heap usage (40.5% of the maximum heap size). Having negative collection headroom clearly makes it impossible for ZGC to keep heap usage below SoftMaxHeapSize. Even in this impossible scenario, heap usage exceeded SoftMaxHeapSize by at most about 8% of the maximum heap size.

Eviction Headroom

GemFire's heap LRU eviction algorithm tries to keep heap usage below the eviction threshold. To meet this goal, it will evict eligible entries from memory when necessary. To avoid evictions, the system needs eviction headroom. Each scenario generates garbage at a rapid rate, relentlessly pushing heap usage up. Given enough eviction headroom, as in the orange and green zones, ZGC responds to the upward pressure in plenty of time to avoid evictions. In the red zone, the eviction headroom is too low. As heap usage rises, it crosses the eviction threshold before it hits SoftMaxHeapSize. By the time the rising heap usage triggers a collection, GemFire has already evicted numerous entries. The number of entries evicted depends on how often heap usage rises above the eviction threshold and how long it stays there. With SoftMaxHeapSize set to 70% and the eviction threshold set to 60%, GemFire evicted nearly 600,000 entries (about half of the entries in the cache) in two minutes.

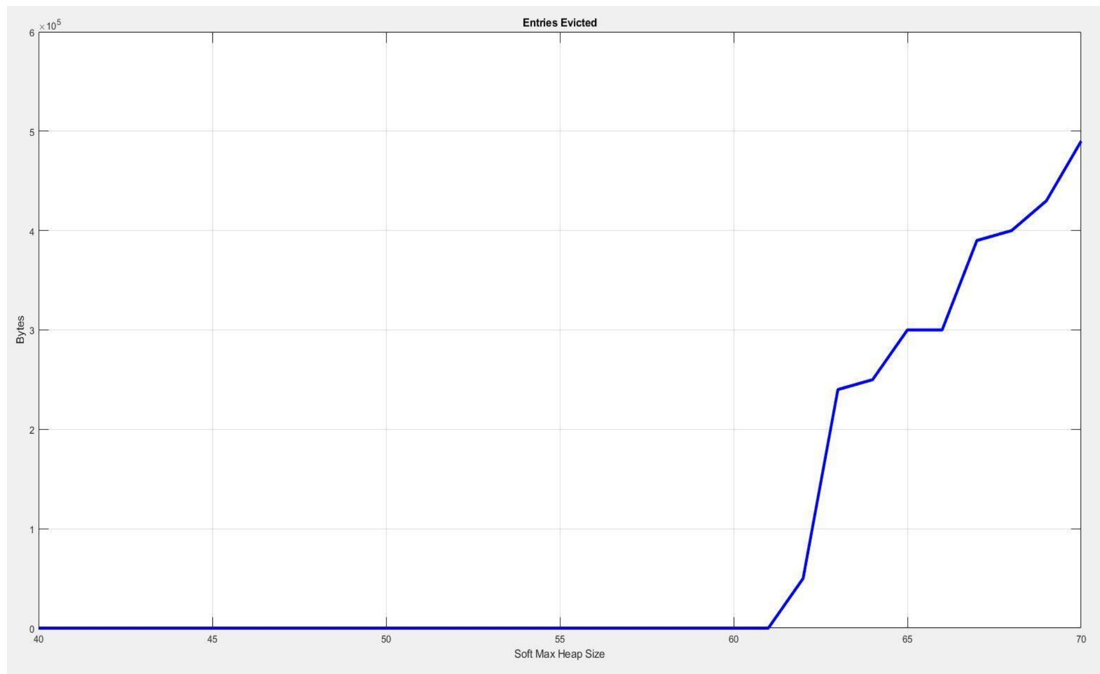

Fig. 3 Evicted rate

Patterns of Heap Usage

Once per second, GemFire samples many statistics, including ZHeap Current Used Memory (self-explanatory) and ZHeap Collection Used Memory, the amount of memory in use at the end of the most recent garbage collection cycle. ZHeap Collection Used Memory gives a reasonable approximation of live-set size, though it will also include any garbage that was generated during the most recent garbage collection cycle. Different SoftMaxHeapSize settings impact heap usage in relation to the eviction threshold and the live-set size.

Insufficient Collection Headroom

Here's what heap usage looks like when ZGC is not given enough collection headroom:

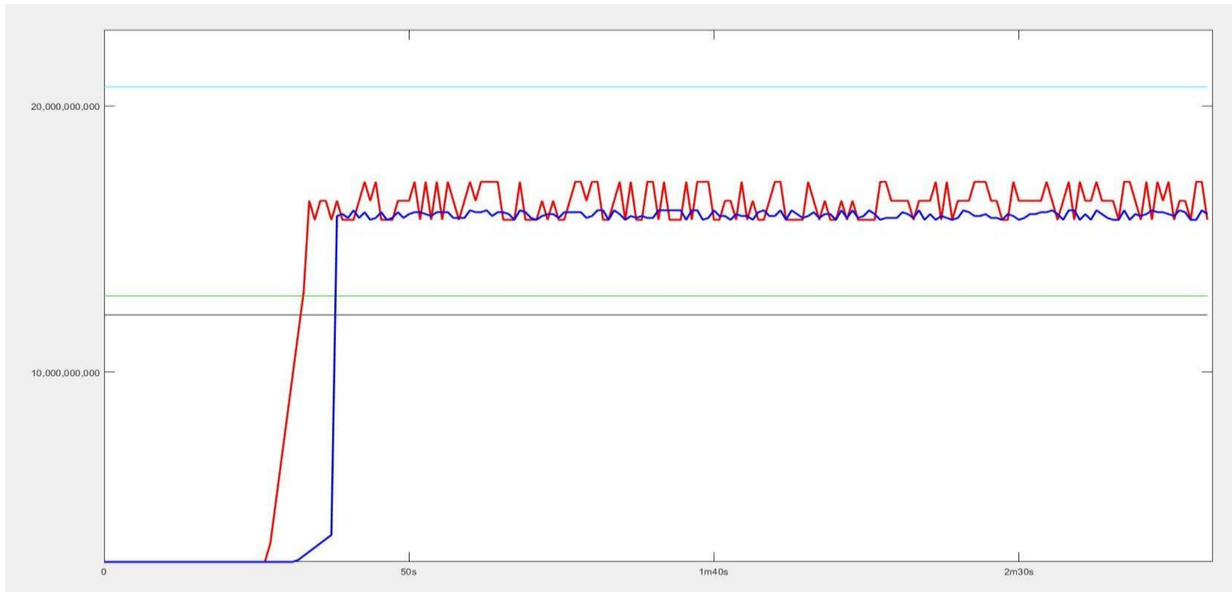

Fig. 4 Heap current used memory status

Current usage never strays far from collection usage. This is a sign of insufficient collection headroom. Because memory usage remains above SoftMaxHeapSize during the entire sustained updates phase, ZGC collects garbage continuously.

Insufficient Eviction Headroom: The heap usage pattern is very different when SoftMaxHeapSize is set above the eviction threshold:

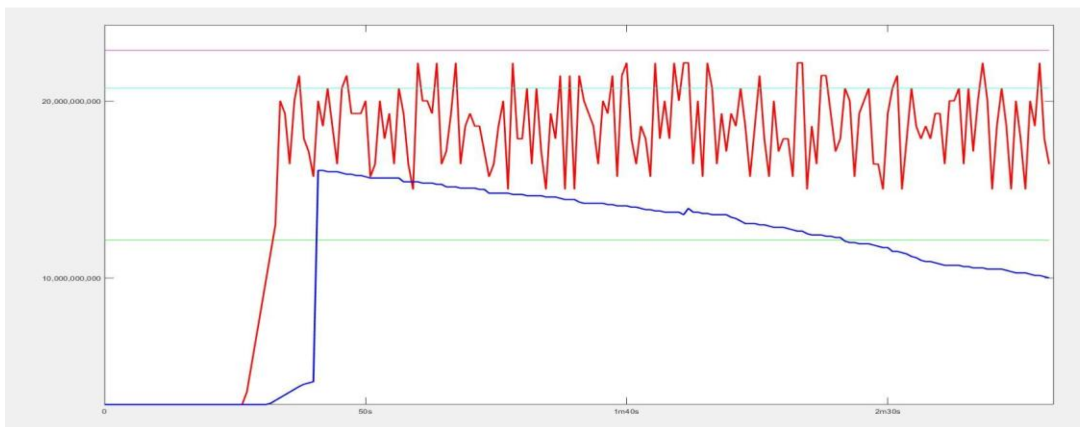

fig.5 heap usage pattern is very different when SoftMax HeapSize is set above the eviction threshold

In this scenario, ZHeap Current Used Memory shows frequent excursions above the eviction threshold. The result is that GemFire evicts entries from memory. As entries are evicted and collected, the live-set size decreases, as reflected in the declining ZHeap Collection Used Memory. As we will see, collections are far less frequent in this scenario compared to the "insufficient collection headroom" scenario described above. But the cost is that many entries are evicted, increasing the likelihood of cache misses.

Sufficient Collection Headroom and Eviction Headroom-In this scenario, SoftMax HeapSize is set well below the eviction threshold and well above the live-set size:

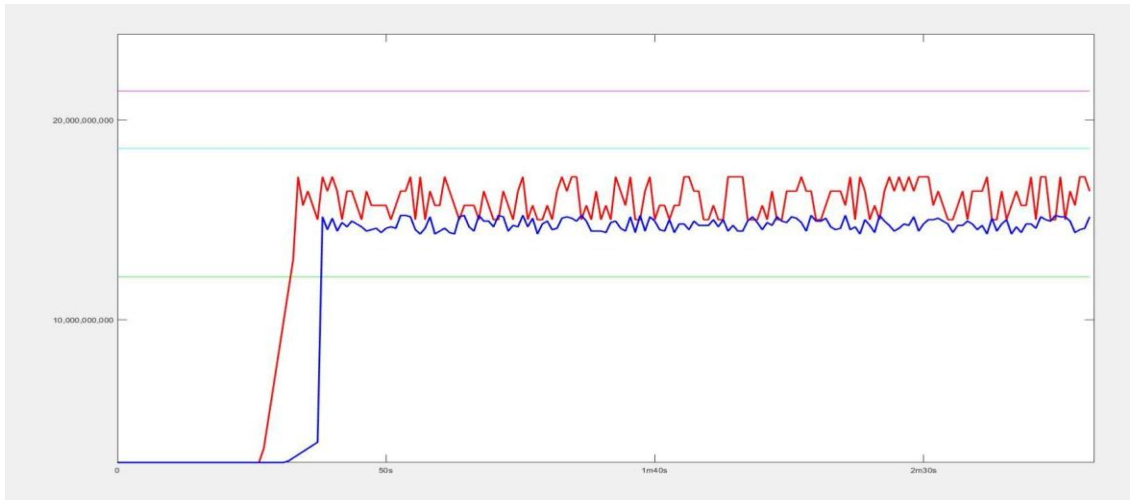

Fig.6 eviction threshold and well above the live-set size

This gives the system sufficient collection headroom below SoftMaxHeapSize and sufficient eviction headroom above. Tuned in this way, ZGC keeps heap usage well away from the eviction threshold, allowing the system to avoid eviction. And it is able to do this with relatively infrequent garbage collections.

How SoftMaxHeapSize Affects Throughput: As SoftMaxHeapSize rises, application throughput rises (puts per second):

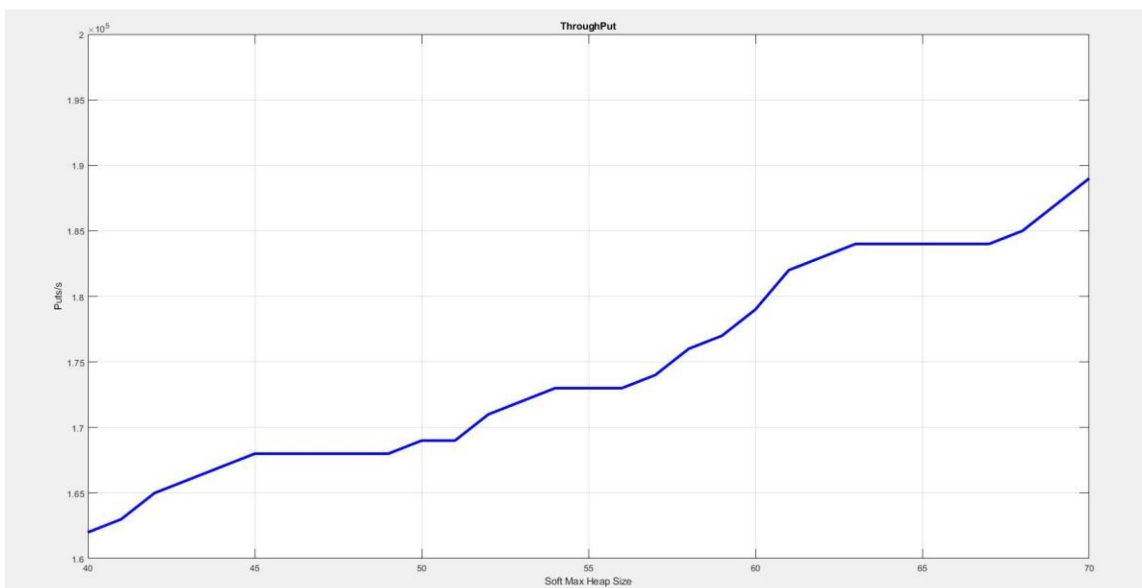

Fig.7 throughout

In each scenario, 16 client threads performed fixed-size puts as fast as possible for 2 minutes. Note that though the client and server executed in separate JVMs, they both ran on the same GCP instance. For the instance's 16 CPUs, the client threads competed with the server's operation threads and ZGC threads. The throughput curve is slightly S-shaped, with a slightly higher slope in the middle of the graph than at either end. Setting SoftMaxHeapSize too low interferes with performance, resulting in slightly flatter throughput on the left side of the graph. On the right side of the graph, the slope decreases

slightly as collection CPU utilization asymptotically approaches 0. When ZGC has insufficient collection headroom, it collects garbage more often and assigns more threads to the garbage collection tasks. In the scenarios where collection headroom was lowest, ZGC kept 4 CPUs busy at nearly all times. This is the maximum number of CPUs that ZGC will assign by default on a 16-core host.

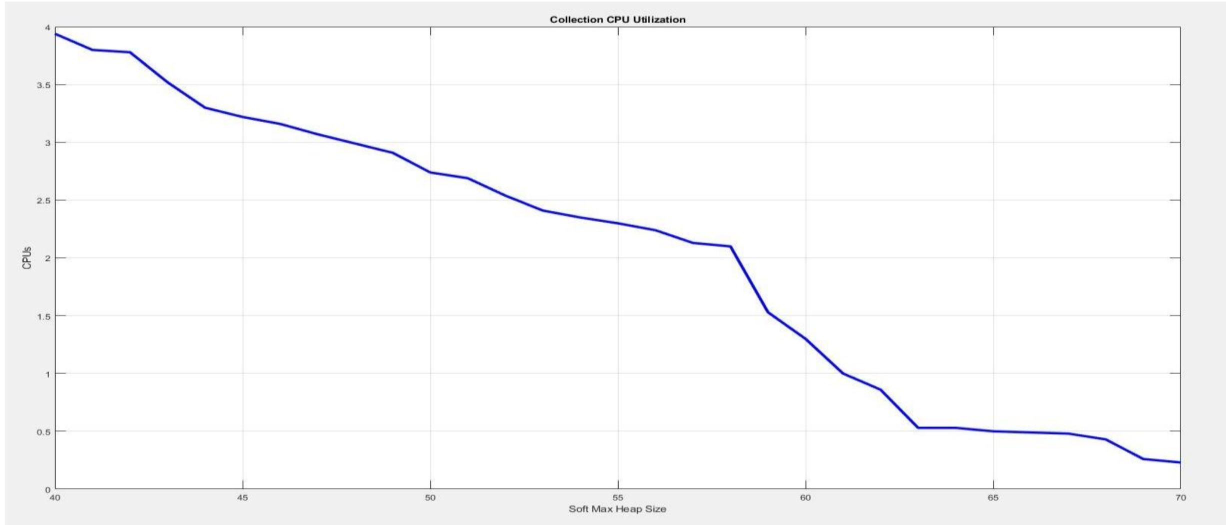

Fig.8 CPU utilization

If subtract the number of CPUs busy doing garbage collection from the total number of CPUs (16), we get the number of CPUs available for other tasks:

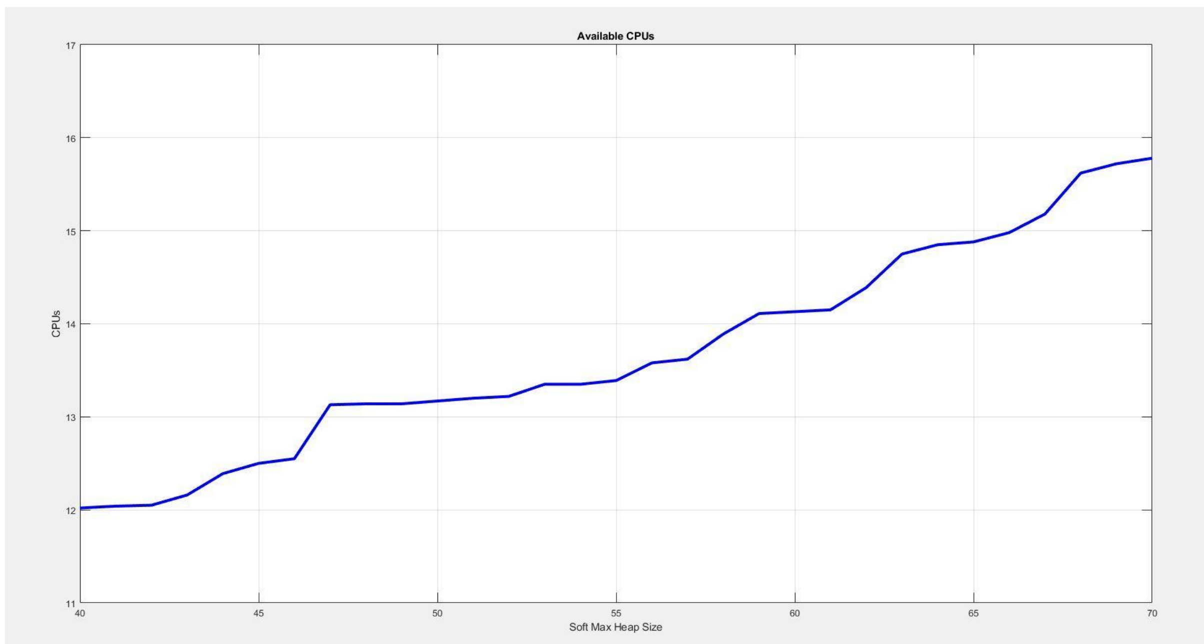

fig.9 available CPU

This graph has essentially the same shape as the throughput graph. The lesson is clear: To maximize throughput, give ZGC plenty of collection headroom.

How SoftMaxHeapSize Affects Garbage Collection Performance

As collection headroom increases, the frequency of garbage collections drops:

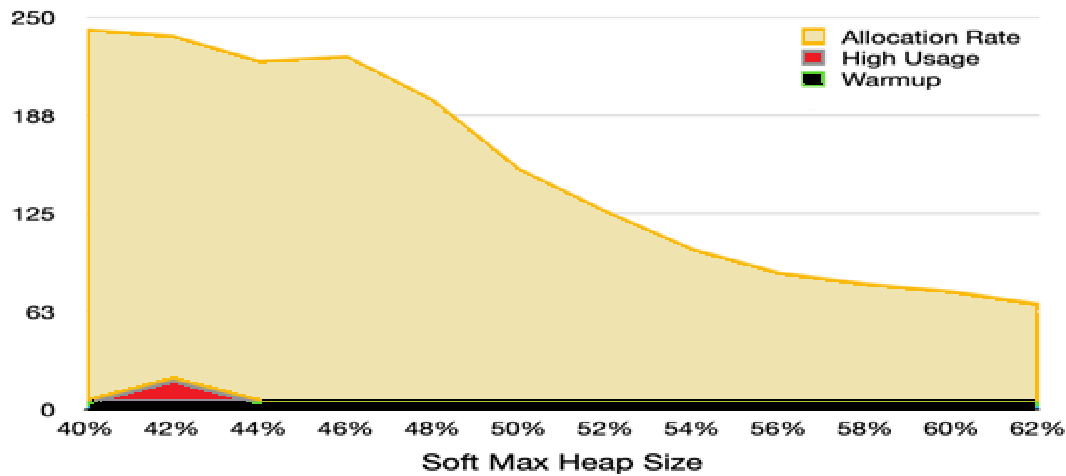

Fig.10 frequencies of garbage collections drops

In the scenarios where collection headroom is too low, ZGC performs garbage collections at the rate of 2 per second. At the same time, the mean CPU consumption of each collection also drops, even as the garbage production rate rises:

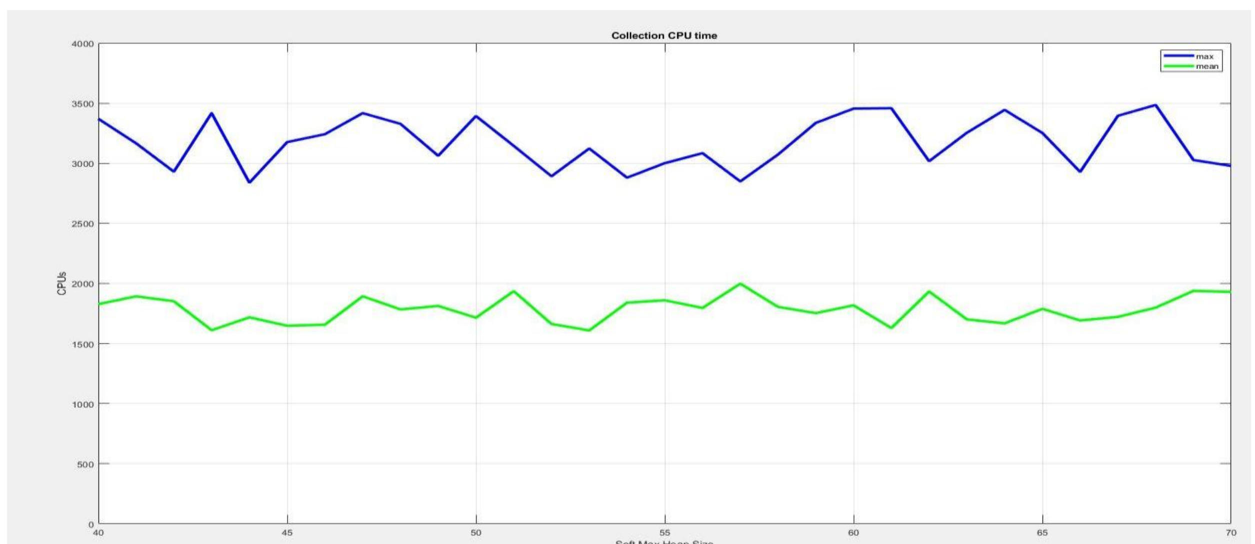

Fig.11 CPU collection time

5. Conclusion

Integrating VMware GemFire with Java 17's Z Garbage Collector (ZGC) and utilizing the heap LRU eviction strategy represents a transformative approach for optimizing fintech microservices. This combination addresses critical performance challenges by enhancing both caching efficiency and memory management. GemFire's

distributed caching enables the storage of frequently accessed data in memory across multiple microservice instances, reducing access times and improving data retrieval speed. With its focus on minimizing garbage collection pause times, ZGC complements this by managing memory more effectively, ensuring smooth and uninterrupted service.

When memory limits are approaching, the heap LRU eviction algorithm systematically removes the least recently used data from the cache, thereby maintaining high cache hit rates and reducing latency. Together, these technologies facilitate a highly scalable and responsive microservice architecture that can handle varying workloads with ease. This integration not only improves throughput and system reliability, but also reduces the load on databases by minimizing cache misses and optimizing data access patterns. As a result, fintech applications benefit from enhanced performance, better resource utilization, and improved user experiences, making this approach a compelling solution for modern financial systems that demand both high availability and rapid processing capabilities. By continuously monitoring and adjusting parameters based on real-time metrics, organizations can achieve a finely tuned infrastructure that meets the dynamic needs of today's financial services landscape.

References

1. Ngafifi, Muhamad. (2014). KemajuanTeknologi Dan Pola Hidup Manusia Dalam PerspektifSosialBudaya. Jurnal Pembangunan Pendidikan: Fondasi Dan Aplikasi, 2(1).
2. Rahadiyan, Inda, & Sari, AlfhicaRezita. (2019). Peluang Dan TantanganImplementasi Fintech Peer To Peer Lending Sebagai Salah Satu Upaya PeningkatanKesejahteraan Masyarakat Indonesia. Defendonesia, 4(1), 18–28.
3. Supriyanto, Edi, & Ismawati, Nur. (2019). SistemInformasi Fintech Pinjaman Online Berbasis Web. JurnalSistemInformasi, TeknologiInformasi Dan Komputer, 9(2), 100–107
4. Darman. (2019). Financial Technology (Fintech): Karakteristik Dan KualitasPinjaman Pada Peer To Peer Lending Di Indonesia. JurnalManajemenTeknologi, 18(2), 130– 137. Https://Doi.Org/10.12695/Jmt.2019.18.2.4
5. Rajagukguk, Rio Chandra. (2018). PenggunaanKriptografi Pada Jwt (Json Web Token) Dalam ImplementasiKeamanan Api
6. Afifah, Khairina, &Setiaji, Hari. (2019). Pengembangan Rest Api SebagaiTeknologiInteroperabilitas Pada AplikasiUii Training Center.
7. Bachri, HendroFebrian, Priyambadha, Bayu, &Rusdianto, Denny Sagita. (2018).
8. PengembanganAplikasiManajemen Event Berbasis Web (Studi Kasus: FakultasIlmuAdministrasi Universitas Brawijaya Malang). JurnalPengembanganTeknologiInformasi Dan IlmuKomputer E-Issn, 2548, 964x.
9. Edy, Edy, Ferdiansyah, Ferdiansyah, Pramusinto, Wahyu, & Waluyo, Sejati. (2019). Pengamanan Restful Api MenggunakanJwtUntukAplikasi Sales Order. JurnalResti (RekayasaSistem Dan TeknologiInformasi), 3(2), 106–112. Https://Doi.Org/10.29207/Resti.V3i2.860
10. Ekasmara, Alif Sani, & Santoso, Nurudin. (2020). Pengembangan Web Portal Landing Page E-Commerce Dengan Pola Single Page Application. 4(8), 2713–2721.
11. Karabey Aksakalli, I., Çelik, T., Can, A. B., &TekiNerdoğan, B. (2021). Deployment And Communication Patterns In Microservice Architectures: A Systematic Literature Review. Journal Of Systems And Software, 180.
12. Muhamad Rizal, Erna Maulina, NendenKostini. (2018). Fintech As One Of The Financing Solutions For Smes. 3(61), 89–100.
13. Ngafifi, Muhamad. (2014). KemajuanTeknologi Dan Pola Hidup Manusia Dalam PerspektifSosialBudaya. Jurnal Pembangunan Pendidikan: Fondasi Dan Aplikasi, 2(1).
14. Perwira, Rifki, & Santosa, Budi. (2017). Implementasi Web Service Pada Integrasi Data Akademik DenganReplikaPangkalan Data Dikti. Telematika, 14(1), 1–11.
15. Putra, Rahmad Ade. (2018). Analisa ImplementasiArsitekturMicroservocesBerbasisKontainer Pada KomunitasPengembangPerangkat Lunak Sumber Terbuka (OpendaylightDevops Community). JurnalSistemInfomasiTeknologiInformasi Dan
16. Komputer (Just It) Universitas Bina Nusantara Magister ManajemenSistemInformasi Jakarta, 150–162.
17. Eduvest – Journal of Universal Studies Volume 1 Number 7, July 2021 567 http://eduvest.greenvest.co.id

18. Rahadiyan, Inda, & Sari, AlfhicaRezita. (2019). Peluang Dan TantanganImplementasi Fintech Peer To Peer Lending Sebagai Salah Satu Upaya PeningkatanKesejahteraan Masyarakat Indonesia. Defendonesia, 4(1), 18–28.
19. Rahmanda, Rama. (2018). RancangBangunAplikasiBerbasis Microservice UntukKlasifikasiSentimen. Studi Kasus: Pt. Yesboss Group Indonesia (Kata. Ai). InstitutTeknologiSepuluhNopember.
20. Rahmatulloh, Alam, Sulastri, Heni, & Nugroho, Rizal. (2018). Keamanan Restful Web ServiceMenggunakan Json Web Token (Jwt) Hmac Sha-512. Jurnal Nasional Teknik Elektro Dan TeknologiInformasi (Jnteti), 7(2). Https://Doi.Org/10.22146/Jnteti.V7i2.417
21. Supriyanto, Edi, & Ismawati, Nur. (2019). SistemInformasi Fintech PinjamanOnline Berbasis Web. JurnalSistemInformasi, TeknologiInformasi Dan Komputer, 9(2), 100–107.