ISSN: 2230-9926

Available online at http://www.journalijdr.com

IJDR

International Journal of Development Research
Vol. 13, Issue, 11, pp. 64225-64229, November, 2023
https://doi.org/10.37118/ijdr.27449.11.2023

RESEARCH ARTICLE OPEN ACCESS

COMPARATIVE STUDY OF MIDDLE TIER CACHING SOLUTION

Sourabh Sethi*[1], Sarah Panda[2] and Ravi Kamuru[3]

[1]Infosys, USA; [2]Microsoft, USA; [3]New York Technology Partners, USA

ARTICLE INFO

Article History:
Received 06th August, 2023
Received in revised form
08th September, 2023
Accepted 11th October, 2023
Published online 27th November, 2023

Key Words:

Caching, Web Caching, CDNs, Spring Cache, Jcache, DXPs Caching Layers.

****Corresponding author: Sourabh Sethi,***

ABSTRACT

Caching is the practice of storing data in proximity or within a faster storage system for rapid retrieval. This occurs at various locations, and in this paper, we will delve into the different areas where data caching is possible. We will closely examine the challenges associated with caching and explore ways to address them through a comparative study of middle-tier caching solutions. Additionally, we will explore the range of caching solutions available in the computer science ecosystem.

Citation*: *Sourabh Sethi, Sarah Panda and Ravi Kamuru. 2023. "Comparative study of middle tier caching solution". *International Journal of Development Research*, 13, (11), 64225-64229.

INTRODUCTION

This article guides a stepwise walkthrough about how to use cache in middle tier applications. What is caching? Caching can be described as the process of storing data in close proximity to the client or within a faster storage system to enable rapid retrieval. Caching are meant to be used in Read heavy systems. Caching occurs at various levels, with the initial level being the browser. The browser stores frequently accessed data to expedite subsequent access. Let's explore the different tiers of caching. In the case of in-browser caching, specific IP addresses can be cached, reducing the need for the browser to repeatedly query the DNS server for the same IP address. This form of caching involves smaller, less frequently changing entries and is known as in-browser caching. The browser caches DNS information and static content, including images, videos, and JavaScript files. This is why a website may take some time to load on the first visit but subsequently loads quickly due to the browser's stored cache. CDN Caching involves scenarios where your browser is located in one region, such as India, and you need to retrieve files stored on servers situated in another region, like the United States. When you attempt to access these files from your browser, a request is sent to a load balancer, which subsequently communicates with the application server to fetch the required files from the file storage. It's well-known that transferring data between machines within the same region is typically faster, but when machines are geographically distant, data transfer can be time-consuming.

The solution to this challenge lies in the use of CDNs, or Content Delivery Networks. Examples of CDNs include well-known companies like Akamai, Cloudflare, and Amazon's CloudFront.The core function of these companies is to establish a global presence by deploying machines in various geographic regions. They take on the responsibility of storing your data, distributing it to all these regions, and offering distinct CDN links for accessing data in specific areas. Imagine you are requesting data from the US region; naturally, you'll receive the HTML or code portion swiftly, as it is considerably smaller compared to multimedia files. When it comes to multimedia content, you are provided with CDN links to files located in your nearest region. Accessing these files from the closest region significantly enhances the speed of retrieval. It's important to note that the usage of these CDN services typically incurs charges based on your actual usage. To prevent the cache from becoming outdated, stale or inconsistent with database, one proposed solution is to implement a Time-to-Live (TTL). Additionally, maintaining synchronization between the cache and the database can be accomplished using strategies such as Write-through cache, Write-back cache, or Write-around cache.

TTL (Time to Live)

In this approach, this strategy can be employed when a short-lived cache is acceptable, allowing for periodic refresh. Cached entries remain valid for a limited duration, after which they must be fetched anew. For instance, if you cache an entry t0 at timestamp T with a time-to-live (TTL) of 60 seconds, any requests for entry t0 made within 60 seconds of t0 will be served directly from the cache. However, if a request for entry t0 is made at timestamp T+61, the cached entry t0 expires, and you must retrieve it again.

Write through cache

When writing data to the database, the process involves initially passing through the cache (which may involve multiple cache machines). The data is stored in the cache (updating it), and then it's subsequently updated in the database before returning a success response. In case of a failure, the changes are rolled back in the cache. While this approach might slow down write operations, it significantly accelerates read operations. For systems with a high volume of read requests, this method can prove to be quite effective. Additional caching types include Local Caching, which optimizes performance by caching data at the application server level, reducing the necessity for repeated database queries. Conversely, Global Caching, also known as In-memory caching, relies on systems like Redis and Memcache to accelerate the retrieval of both raw and processed data.In this article, we will evaluate various middle-tier caching solutions available in the market while also examining their integration with the abstraction layers such Spring Cache and Jcache Specifications.

Issue related to caching

Caching comes with a set of challenges, including its size limitations. Caches merely contain duplicates of the actual data, which resides elsewhere, often referred to as the 'source of truth.' As time passes, cached data can become outdated and diverge from the accurate data in the database, as changes in the source of truth are not automatically reflected in the cache. Additionally, the cache can reach its storage capacity, causing it to become full. This raises the questions: What measures can we implement to ensure data consistency across Cache & Database? And how can we accommodate new entries when the cache is already at its maximum capacity?

In this article, we have examined the diverse challenges linked to caching and delved into potential solutions by implementing Cache Invalidation Strategies and Cache Eviction Policies.

Cacheinvalidation strategy

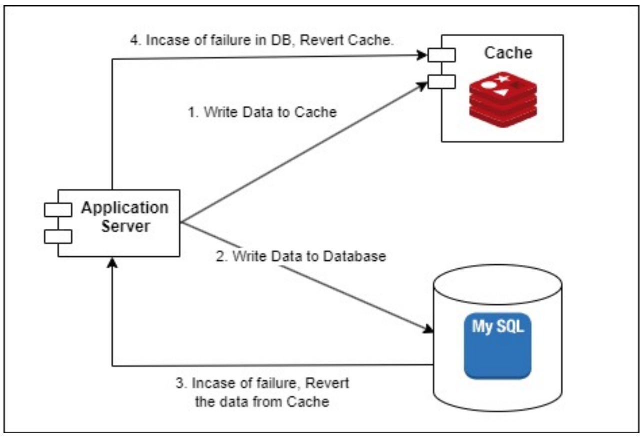
4. Incase of failure in DB, Revert Cache.
Cache
1. Write Data to Cache
Application Server
2. Write Data to Database
My SQL
3. Incase of failure, Revert the data from Cache

Figure 1. Write through Cache

Write back cache

Initially, the write operation is recorded in the cache. Once the cache write is successful, a success response is promptly returned to the client. Subsequently, the data is synchronized with the database asynchronously, without hindering ongoing requests. This method is favored in scenarios where immediate data loss is not a critical concern, such as in analytical systems where the precise data in the database holds little significance. In this context, the occasional data loss is tolerable, as it doesn't significantly impact analytical trend analysis. While it may introduce some inconsistency, it offers the advantage of achieving exceptionally high throughput and minimal latency.

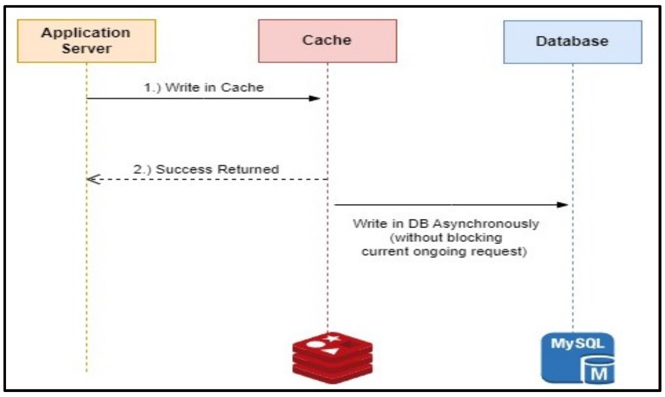

Figure 2. Write Back Cache

Write around cache

In this case, write operations are executed directly in the database, and there may be a discrepancy between the cache and the database. As a remedy, TTL or a similar mechanism can be employed to periodically retrieve data from the database and synchronize it with the cache.

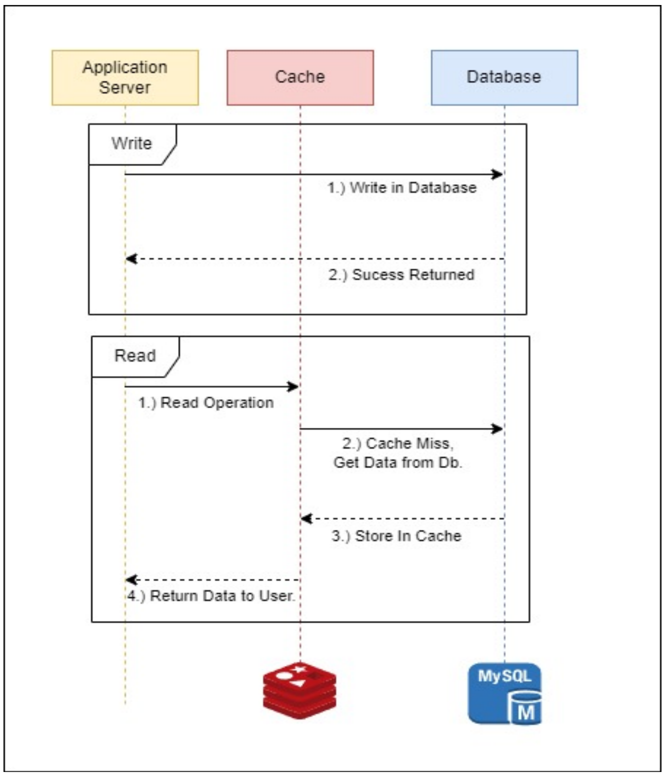

Figure 3. Write Around Cache

Cache Eviction Policy

The selection of an eviction strategy should be determined by the data most frequently accessed. The caching strategy should be carefully crafted to maximize cache hits while minimizing cache misses. There exist several eviction strategies aimed at freeing up space in the cache for new data writes. A few of these strategies include:

FIFO (First In, First Out)

FIFO, an acronym for "First In, First Out," is a principle or method in which the items that are first added or acquired are the first ones to be utilized or processed, while the most recent items are addressed or processed later in the sequence. In the realm of computing and data structures, FIFO is commonly employed in scenarios like queue management, where the order of arrival dictates the order of processing or retrieval.

LRU (Least Recently Used)

An LRU (Least Recently Used) Cache is a caching approach in which the items accessed or used least recently are the first to be removed or replaced when the cache reaches its capacity or encounters an eviction policy trigger. The likelihood of retaining data in the cache is determined by the order in which items are accessed or utilized in an LRU Cache. This strategy prioritizes keeping the most recently used items in the cache, ensuring that frequently accessed data is readily available for quicker retrieval. LRU Caching finds common use in scenarios where optimizing for recent usage patterns holds significance.

LIFO (Last In, First Out)

A Last In, First Out (LIFO) Cache operates by removing or processing the most recently added or inserted item first when the cache reaches its capacity or when an eviction policy is enacted. In this caching mechanism, the order of insertion dictates the order of removal, with the latest entry being the first to be evicted. Often referred to as a stack-based approach, LIFO Caching treats items in a last-in, first-out manner, akin to stacking objects. This strategy is utilized in scenarios where prioritizing the order of recent additions is crucial.

when an eviction policy is enacted. In an MRU Cache, the likelihood of retaining items is determined by the order in which they are accessed or utilized. This strategy focuses on maintaining the availability of the most recently used items for quicker retrieval, placing emphasis on recent access patterns. MRU Caching is frequently utilized in situations where optimizing for recent usage is paramount. All cache eviction policy solutions are implemented using the strategy design pattern by using the basic principle of polymorphism & Interfaces i.e. Dependency Inversion Principle (DIP). The concrete implementation of the algorithm is contingent upon the interface, and our clients, who utilize the cache mechanism, rely on the CacheStrategy Interface. Clients employing our caching mechanism are unaware of the specific concrete implementation details, Caches abstraction as shown in the above figure. Various caching solutions implement JCache, an interface provided by the Java community under the JSR 107: JCACHE - Java Temporary Caching API or Spring Framework has also provided abstraction interface which is implemented by various caching solutions available in the market. This specification standardizes the in-process caching of Java objects, offering an efficient implementation and relieving programmers from the responsibility of handling cache expiration, mutual exclusion, spooling, and cache consistency. It supports caching objects with types unknown until runtime, but only those implementing the serializable interface can be spooled.

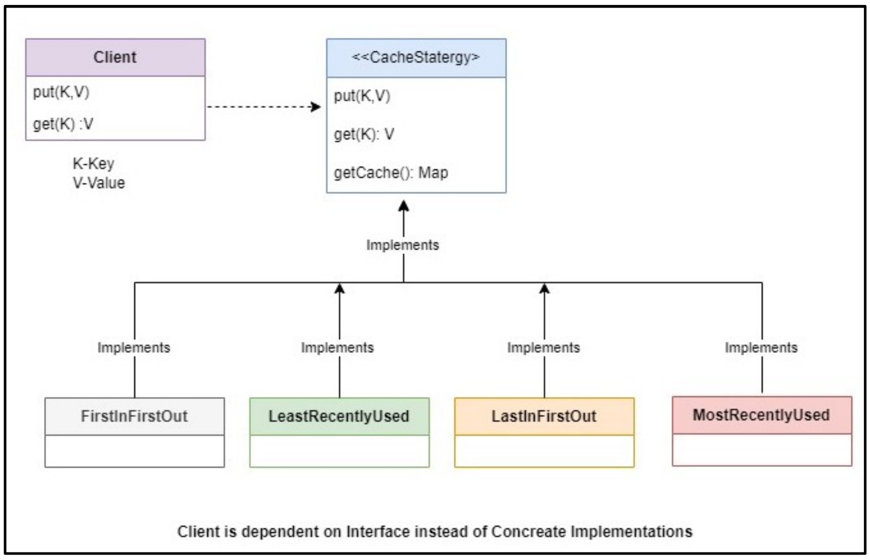

Figure 4. Cache Strategy

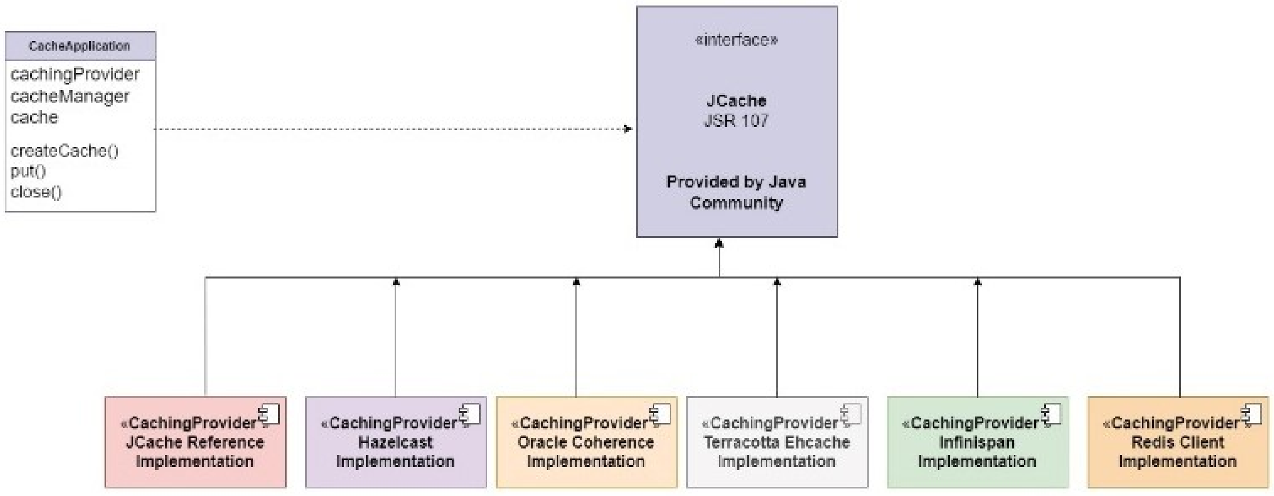
CacheApplication
cachingProvider
cacheManager
cache
createCache()
put()
close()
«interface»
JCache
JSR 107
Provided by Java Community
«CachingProvider» JCache Reference Implementation
«CachingProvider» Hazelcast Implementation
«CachingProvider» Oracle Coherence Implementation
«CachingProvider» Terracotta Ehcache Implementation
«CachingProvider» Infinispan Implementation
«CachingProvider» Redis Client Implementation

Figure 5. Jcache JSR 107

MRU (Most Recently Used)

A Most Recently Used (MRU) Cache follows a caching strategy where the most recently accessed or utilized items are given priority and are the last to be removed when the cache reaches capacity or

Spring's caching abstraction seamlessly integrates with the JCache standard (JSR-107) annotations, including @CacheResult, @CachePut, @CacheRemove, and @CacheRemoveAll, as well as the companions @CacheDefaults, @CacheKey, and @CacheValue.

These annotations can be utilized even without migrating the cache store to JSR-107. The internal implementation leverages Spring's caching abstraction, providing default CacheResolver and KeyGenerator implementations compliant with the specification. In essence, if you are already using Spring's caching abstraction, transitioning to these standard annotations is possible without altering your cache storage or configuration. Various caching solutions, including the JCache Reference Implementation (Concurrent HashMap), Hazelcast, Oracle Coherence, Terracotta Ehcache, Infinispan, and Redis, have implemented the JCache standard. In the figure depicted below, our application relies on the JCache API, adhering to the dependency inversion principle. Here, other Cache Providers have implemented the JCache API provided by the Java community. This approach simplifies the task for application developers, allowing them to use any cache without delving into the implementation details. As developers of the application, our responsibility lies in providing the configuration to the CacheManager for Cache Invalidation Strategy, Cache Eviction Policy, Time to live, and creating the cache object based on that configuration so that we can use it throughout application.

Cache Solutions

In this section, we will discuss different cache solutions available in the market around Java ecosystem & spring framework. Caffeine is an exceptionally high-performance caching library designed for near-optimal performance. Operating as a local caching solution, it is non-blocking in nature and is supported by Jcache and SpringCache. Caffeine also features Time-to-Live (TTL) support and offers two eviction strategies: FIFO and Weight-based eviction. The latter involves assigning a weight to each entry, adhering to a custom algorithm, and setting a weight limit for the cache. If a new entry exceeds the maximum weight, the 'heaviest' entries are removed until the sum of the weights falls below the defined threshold. Ehcache, an open-source, standards-based cache, enhances performance, reduces database load, and simplifies scalability. Widely used in Java environments, Ehcache is robust, proven, and integrates seamlessly with popular libraries and frameworks. Functioning as a local caching solution, it is single-threaded and blocking, with support for JCACHE and SPRING CACHE, along with TTL. Ehcache provides three eviction strategies: LRU, LFU, and FIFO. Moreover, Ehcache allows users to implement custom eviction policies. Terracota is the enterprise version of Ehcache, providing distributed capabilities. Infinispan, an open-source in-memory data grid, offers versatile deployment options and robust capabilities for storing, managing, and processing data. Supporting local/global distributed caching, it is configurable and offers LFU as an eviction strategy. Coherence is a scalable, fault-tolerant, cloud-ready distributed platform for constructing grid-based applications and securely storing data. With non-blocking behavior, it supports both local and distributed caching mechanisms. Coherence supports Jcache, TTL, and implements LRU and LFU as eviction strategies. Ignite, a distributed database designed for high-performance computing with in-memory speed, features its own asynchronous primitives. It is non-blocking, supports Jcache and Spring Cache, and provides eviction policies such as TTL, LRU, and FIFO. Apache Geode is a data management platform offering real-time, consistent access to data-intensive applications across widely distributed cloud architectures. Operating as an in-memory data management system, Apache Geode supports Spring Cache and TTL, with LRU as its sole eviction policy.

Hazelcast is a streaming and memory-first application platform for fast, stateful, data-intensive workloads on-premises, at the edge or as a fully managed cloud service. It is non-blocking in nature which supports Jcache& Spring Cache. Redis provides a method for storing key-value pairs in various data types like Lists, Sets, and Hashes. The data is kept in memory, ensuring rapid retrieval when requested. This characteristic makes Redis an ideal choice as a cache for applications where quick data retrieval is crucial. Redis operates in a non-blocking, single-threaded manner. Similar to Redis, Memcached is an open-source solution for storing key-value pairs in memory, resulting in swift data retrieval. This makes Memcached another effective option for applications where speed is a priority. Memcached is also multithreaded, potentially offering performance improvements by utilizing multiple cores. While Redis functions as an in-memory (mostly) data store and is non-volatile, Memcached operates as an in-memory cache and is volatile. Additionally, Memcached is constrained to the LRU (Least Recently Used) eviction policy, whereas Redis supports six different eviction policies.No eviction, resulting in an error when the memory limit is reached. All keys LRU, removing keys based on the least recently used criterion. Volatile LRU, removing keys with an expiration time set, based on the least recently used criterion. All keys random, removing keys in a random manner. Volatile random, removing keys with an expiration time set randomly. Volatile TTL, removing keys with an expiration time set based on the shortest time to live criterion. Redis supports persistence, earning its classification as a data store, through two distinct methods. The first is the RDB snapshot, which captures a point-in-time snapshot of the entire dataset. This snapshot is stored in a file on the disk and is taken at specified intervals, allowing the dataset to be restored upon startup. Another method is the AOF (Append Only File) log, which records all write commands executed in the Redis server. Similar to the RDB snapshot, this log is stored on disk, and the dataset can be reconstructed by re-executing the commands in their chronological order during startup. The AOF log is preferable when zero data loss is imperative, as it updates with every command and avoids corruption issues due to its append-only nature. However, it may lead to larger file sizes compared to an RDB snapshot.

In this paper, we have evident the LRU policy evicts the least recently used keys first using redis.

```
127.0.0.1:6379> CONFIG SET maxmemory-policy allkeys-lru
OK127.0.0.1:6379> CONFIG SET maxmemory 1mb
OK
```

Let us also set the maxmemory to 1MB.

```
127.0.0.1:6379> CONFIG SET maxmemory 1mb
OK
```

To test the LRU policy, the redis-cli's capabilities are limited. Hence, we will switch to a python script using the redis-py library.
Let us first create an instance of the redis client.

```
fromredisimport Redis

instance = Redis(host=DEFAULT_HOST, port=DEFAULT_PORT)
```

Flush the database to remove any existing data.

```
instance.flushall()
```

Let's test out creating a key-value pair.

```
instance.set('key', 'value')
```

Getting the value of the key.

```
instance.get('key')

>b'value'
```

Let us check the memory usage of the redis instance.

```
print(f"Used memory: {node.info()['used_memory_human']}")

> Used memory: 1.49M
```

To simulate the LRU policy, we will create some key-value pairs first and then access a subset of them. Then, we will flood the redis instance with a lot of key-value pairs. This will cause the redis instance to reach the memory limit and start evicting keys. According to the LRU policy, the least recently used keys will be evicted first. Hence, the keys that we accessed earlier will not be evicted. Let us first create a utility function to create a key-value pair and access it.

```
defsave_data(end: int, start: int = 0) -> None:
fori in range(start, end):
```

```
instance.set(f"key-{i}",
f"value-{i}")

defread_data(end: int, start: int = 0) -> None:
fori in range(start, end):
instance.get(f"key-{i}")
```

Let us create the initial key-value pairs.

```
store_data(redis, end=5000)
```

Read a subset of the keys.

```
read_data(redis, end=1000)
```

Now, let us flood the redis instance with a lot of key-value pairs.

```
store_data(redis, end=10000, start=5000)
```

Let us see if the keys that we accessed earlier are still present.

```
instance.get('key-1')

>b'value-1'

instance.get('key-1000')

>b'value-1000'
```

You could also use the mget command to get multiple keys at once.

```
instance.mget([f"key-{i}" for i in range(1, 1001)])
```

Now, plotting the distribution of the keys that were evicted affirms the LRU policy using Redis Cluster.

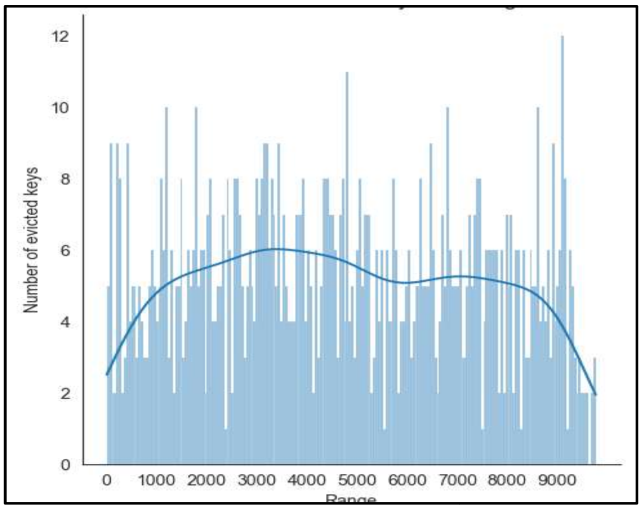

Figure 6. LRU Policy

It is evident here in above Figure 6: Keys accessed earlier experience fewer evictions, while the remaining keys in the initial set undergo a higher rate of evictions. The keys in the second set are evicted in the order of their creation, ensuring that the most recent keys are retained.

Use Case

In a banking application, when users log in to their online banking portals via a web app or a mobile app, they anticipate finding a comprehensive dashboard displaying all their account data. This includes information about their checking and savings accounts, mortgage loans, investment accounts, and/or retirement accounts. The alternative, logging in separately to distinct online portals for each account, is inconvenient and may drive users to competitors with more integrated banking systems. Typically, users maintain multiple accounts within a single bank, and this information is distributed across different systems in the bank's network. Therefore, it is crucial for banks to implement caching to consolidate diverse data belonging to a single user and promptly present all account information upon login. Moreover, caching ensures swift access for customers to their account information, providing a positive user experience and enabling the bank to handle substantial online traffic without compromising responsiveness. Connecting user information across various banking channels, coupled with enhanced responsiveness, not only elevates user satisfaction and loyalty but also streamlines the bank's ability to assess each user's assets and financial history conveniently.

CONCLUSION

As Digital Experience Platforms (DXPs) rapidly progress, relying solely on a basic cache, like storing data in a hashmap, may prove insufficient as the volume of data coursing through your company's systems expands. The demands of processing such extensive data will eventually outstrip the modest benefits offered by a simple cache's data storage mechanism. It becomes apparent that evolving data requirements necessitate a more advanced caching solution, such as a Redis cluster.We have gone through the eviction policy & cache invalidation strategies which is supported by various caching solutions such as Redis, EhCache and many more using interface JCache in Java.

ACKNOWLEDGMENT

We extend our appreciation to all the developers we've partnered with. Their valuable contributions have played a crucial role in identifying caching solutions in the market and implementing diverse caching solutions across various enterprise applications over the years.

REFERENCES

Altınel, Mehmet, Christof Bornhövd, Sailesh Krishnamurthy, Chandrasekaran Mohan, Hamid Pirahesh, and Berthold Reinwald. "Cache tables: Paving the way for an adaptive database cache." In Proceedings 2003 VLDB Conference, pp. 718-729. Morgan Kaufmann, 2003.
Bornhövd, Christof, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. "Adaptive Database Caching with DBCache." IEEE Data Eng. Bull. 27, no. 2 (2004): 11-18.
Carlson J. Redis in action. Simon and Schuster; 2013 Jun 17.
Li, Songhuan, Hong Jiang, and Mingkang Shi. "Redis-based web server cluster session maintaining technology." In 2017 13th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD), pp. 3065-3069. IEEE, 2017.
Liu, Qian, and Haolin Yuan. "A High Performance Memory Key-Value Database Based on Redis." J. Comput. 14, no. 3 (2019): 170-183.
Website : https://redis.io/docs/reference/eviction/

****