Intelligent Dynamic Caching Framework

K.L.T Gamage

2021

Intelligent Dynamic Caching Framework

A dissertation submitted for the Degree of Master of Computer Science

K.L.T Gamage
University of Colombo School of Computing
2021

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis. This thesis has also not been submitted for any degree in any university previously.

Student Name: K.L.T. Gamage

Registration Number: 2016/MCS/032

Index Number: 16440327

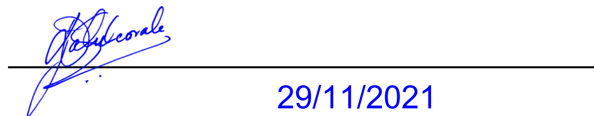
29/11/2021

Signature of the Student & Date

This is to certify that this thesis is based on the work of Mr. K.L.T Gamage under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by,

Supervisor Name: Dr. D. A. S. Atukorale



29/11/2021

Signature of the Supervisor & Date

I would like to dedicate this thesis to my beloved parents.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to my supervisor, Dr. Ajantha Atukorale for his invaluable guidance throughout this study. He inspired and encouraged me all the time, to complete this study. I would also like to thank my parents and my wife for their understanding, support, and encouragement throughout this study.

ABSTRACT

Application-level caches can effectively improve the performance of any I/O bound applications. However, what needs to be cached in the application caches, should be decided by the developers. This holds true for enterprise level application cache implementations such as Redis and Memcached. Deciding what to be cached, would not be a straightforward task as it might require the knowledge of the system or the business domain, system specifications and possible workloads. In this thesis, a lightweight, simple to integrate and Intelligent application-level cache framework is proposed, and with the use of the framework, application logic can be decoupled from the cache logic. The proposed framework can be used in the general case, without limiting to any database centric caches as it mainly considers frequency and data size which are readily available in every application considered. The framework uses a Support Vector Machine (SVM) classifier model to predict what to cache, hence removing much burden from the developers. The framework sits between the application and the cache implementation, and handles the extra processing asynchronously and automatically, avoiding any overhead added by the extra processing used for the caching decision. Since the framework acts as a transparent layer between the application and the underline cache, for existing applications which already use a cache, this caching framework can be integrated seamlessly. The framework is written in Python and uses a Redis cache as underneath cache implementation but can be extended to support any type of cache implementation. The proposed framework was evaluated against uniform workloads and non-uniform workloads, with different cache eviction methods as well as with different time to live values. The experimental results show that the performance of the application can be improved up to 17% with the use of the proposed model when the specified cache size is limited compared to the total size of all the possible cacheable data.

LIST OF PUBLICATIONS

TABLE OF CONTENTS

ACKNOWLEDGMENTS iii

ABSTRACT iv

LIST OF PUBLICATIONS v

TABLE OF CONTENTS vi

LIST OF FIGURES vii

LIST OF TABLES viiii

CHAPTER 1: INTRODUCTION 1

1.1 Motivation 1

1.2 Statement of the problem 2

1.3 Research Aim and Objectives 2

1.3.1 Aim 2

1.3.2 Objectives 2

1.4 Scope 2

1.5 Structure of the thesis 3

CHAPTER 2: LITERATURE REVIEW 4

CHAPTER 3: METHODOLOGY 10

CHAPTER 4: EVALUTION AND RESULTS 13

CHAPTER 5: CONCLUSION AND FUTURE WORK 19

APPENDICES I

REFERENCES IV

LIST OF FIGURES

Figure 1 : High Level Design of the Solution 10
Figure 2 : High-level Architectural Design 11
Figure 3: Performance under Uniform workload, LFU eviction and high TTL..........14
Figure 4: Hit ratio under Uniform workload, LFU eviction and high TTL..........14
Figure 5: Performance under Uniform workload, LRU eviction and high TTL 15
Figure 6: Hit ratio under Uniform workload, LRU eviction and high TTL 15
Figure 7: Performance under Non-Uniform workload, LFU eviction and high TTL 16
Figure 8: Hit ratio Non-Uniform workload, LFU eviction and high TTL 16
Figure 9: Performance under Non-Uniform workload, LRU eviction and high TTL..........17
Figure 10: Hit ratio under Non-Uniform workload, LRU eviction and high TTL..........17
Figure 11 : Performance against Time to Live 18

LIST OF TABLES

Table 1: Accuracy of the models.....13

CHAPTER 1

INTRODUCTION

Application caches play a vital role in enterprise applications. Since most of the enterprise applications use database systems, third party systems, performance gain can be obtained by introducing in-memory caches. There are several cache implementations available such as Redis[1], Memcached[2], Hazelcast[3]. Caches can be categorized into two main cache types, namely static\pre-loading caches, and dynamic\lazy loading caches. Static caches populate the cache at the application initialization. Dynamic caches populate the cache on demand.

When cache size is limited, cache removal methods are also important. Popular cache removal algorithms can be identified as LRU [4] and LFU[5]. Although implemented caches support cache eviction policies, contrast to traditional low-level caches [6] [7] [8], however for all the cache implementations mentioned above, what needs to be cached, should be decided by the developers. Understanding what needs to be cached is not a straightforward task as developers are required to consider the application domain and the application workloads.

For this purpose, developers might use the following approaches.

- No decision making. Everything is added to the cache. Cache removal method decides what to be removed if cache size exceeds.
- Pre-defined rules.

First approach is a blind approach and for the second approach developers might need to consider the cache request distribution. However, this distribution might vary depending on the load of the application and the time. Both these approaches are not the ideal solutions in terms of performance and development aspects.

1.1 Motivation

Ideal solution is to come up with an intelligent caching framework which can automatically decide what needs to be cached. This would address the limitations of no decision making and pre-defined rules approach.

With this proposed framework, developers only need to focus on the application logic and the caching framework will take care of what needs to be cached. Also, with the workloads or

spikes of the load that the system might undergo from time to time, the framework would be able to adjust accordingly.

1.2 Statement of the problem

Although there are frameworks which are specific to certain type of domains, currently there is no general-purpose caching framework which can be used to automatically decide what needs to be cached. This framework should be intelligent enough to adapt to different situations such as workload changes.

1.3 Research Aims and Objectives

1.3.1 Aim

To come up with an general purpose, intelligent caching framework which could automatically decide what needs to be cached, so that the performance can be improved when there's a limitation in cache capacity which can be accommodated for application considered.

1.3.2 Objectives

1. To implement a caching framework which can be reusable across multiple caches. This would enable developers to only focus on application logic, not what needs to be cached.
2. Evaluate performance of the cache against the overhead added with the framework.

1.4 Scope

Scope includes a caching framework which can be reusable across multiple caches. Functionality will be available via an API (Application Programming Interface). Caching framework will take care of what needs to be cached and what needs not to be cached.

When we consider the difference between dynamic and static caches, static caches entries are populated at the initialization and dynamic cache entries are populated on demand. The proposed framework supports dynamic caching which is a harder problem.

Redis cache will be used as the underneath cache for the caching framework. Redis is accepted as one of the best caching solutions by the community and it's also open source.

Although scope is limited to dynamic cache, with minor changes, it should be possible to extend the same solution to static caches as well.

When there's no unlimited capacity (i.e., It's not possible to cache every possible entry), the decision of what to cache does matter. The proposed framework only automates the decision-making part while trying to optimize the hit ratio. However, the developer still needs to take care of generating\populating\creating a cache entry.

Also, it is assumed that total cache size is a constant so that it does not change due to the load of the system. Therefore, adding cache entry might evict existing cache entry. Cache eviction is handled by the underneath cache implementation. Therefore, the proposed framework will neither enhance the cache eviction nor determine the eviction method.

1.5 Structure of the Thesis

Chapter 2 describes the previous work done in the same research area and discusses and reviews those publications critically. It also highlights the research gap which is to be addressed by this research.

Chapter 3 describes the problem in detail and how to solve the problem. It also describes the deliverables and limitations of the solution.

Chapter 4 describes the findings and the evaluation of the research. Chapter 5 describes the conclusion and future work.

CHAPTER 2

LITERATURE REVIEW

This chapter describes the previous work done in the same research area.

In [9], a cache admission policy is discussed for the caches of search engines. The main objective of having an admission policy is to avoid adding queries which might not be hit again in the near future. Therefore, the basis of the admission policy is the frequency. To decide the frequency, some of the past properties such as past queries and some of the properties of the query itself such as no of characters in the query have been used.

Proposed approach is a simple approach which only targets a set of scenarios. The main problem to be solved by this approach is that when cache size is limited, adding an infrequent entry would evict one or more existing entries which might be more frequent than the newly added entry. Above problem could lead to reduction of overall cache hit ratio. Therefore, the admission policy is used to avoid adding such infrequent queries to the cache.

However, there could be scenarios where not only the frequency of the cache entry but also the size of the cache entry is also important. It would be more beneficial to add more frequent and small cache entries than more frequent and large cache entries as it would allow more entries to be cached, resulting in a higher overall hit ratio.

[10] introduces a framework to configure caches which can be used by the developers. This framework can be used for web applications which use hibernate. With the use of available web logs, workload to database access mappings are generated. With the use of colored Petri nets, cache configurations are optimized automatically.

Configuring caches would be a tedious task for most of the cases. Developers might need to change the cache configuration time to time based on the workloads as well. Also, there can be incorrect configuration done by the developers which might go unnoticed at the beginning. The proposed framework helps the developers to identify where to configure caches and adds them automatically reducing the manual effort required by the developers.

Although the proposed framework identifies where to configure caches in the application code, it does not optimize the caches by considering what needs to be cached for each configured caches when there’s a limited cache size available.

Web caching admission and replacement technique is introduced in [11]. Opposed to heavy machine learning techniques, a multinomial logistic regression (MLR) classifier is used as the model. The worthiness of the cache entry is classified using the proposed model. The MLR model is trained for classifying the web cache's object worthiness. The object worthiness is calculated with the use of object properties such as size of the object and the web traffic.

The output of the MLR model is a worthiness class. Worthiness of the cache entries are updated or calculated on demand. Based on the worthiness, cache admission and/or cache replacement policies are invoked.

This approach is applicable only when a system controls the underneath caching implementation so that admission and eviction policies are governed by the system itself. The usable scope is narrow when application-level caches are considered as most of the the enterprise systems only use existing implemented caches such as Redis.

[12] also proposes a framework which can be used by the developers to identify cacheable locations with a minimum effort. The cacheability patterns are based on the runtime monitoring of the web applications. The focus of the framework is to reduce the effort required by the developers to analyze the cacheable points in the system.

In this paper, APLCache Framework has been introduced by the authors. The framework is implemented in Java and can be integrated with either new applications or existing ones. With the use of aspect-oriented programming, system's method executions are intercepted. Collected data is analyzed and the framework is capable of producing recommendations to the developers.

For the recommendations following the criterion are used for the decision-making considerations.

- Staticity (ST) - Is the data static?
- Changeability(C) - Does the data constantly change?
- Frequency (F) - Is the data frequently requested?
- Shareability (SH) - Is the data user specific?
- Expensiveness (E) - Is the data expensive to compute?
- Large Data (LD) - Is the data size large?
- Large Cache (LC) - Is the cache size large?

In the above approach, the decision-making process is a somewhat rigid approach. Thus, making changes to the decision-making process would be a tricky task. Also adopting this approach for an existing system would be a challenging task as aspect-oriented monitoring needs to be established. Also, this could add additional overhead which could potentially decrease the overall system performance if not done properly.

[13] proposes a cache configuration automated approach named as SACC which stands for Smart Application-level Cache Configuration. In the proposed approach application logic is decoupled from the cache logic. Without any manual intervention, caching operations are executed automatically. LSTM (long short term memory) model is used to decide what needs to be cached.

This paper proposes a quantifiable method to evaluate cacheability of the objects, in contrast to the metrics such as recency and frequency. Cacheability of the data is used to determine whether data is suitable for caching. With the use of cacheable data, the LSTM neural network is trained so that the model can on the fly predict whether a particular data is suitable for caching. By adding a transparent layer between the application and the database, the SACC framework removes the requirement of manual cache configuration by the developers.

The given approach is only applicable to database centric applications. However, there can be scenarios where cacheable data are generated by complex calculations or hitting some third-party endpoints. Also, the given approach is a complex approach which might add an additional overhead to the system if not implemented properly. Also, there’s no indication of the computation overhead added by the proposed framework and might not be suitable for CPU bound applications.

[14] identifies two main approaches for application-level caching, reactive approach and proactive approach. In the reactive approach data is always cached after it has been requested. This would result in a reduced hit ratio of the cache for a certain period of time, due to the fact that the first requests of distinct cache keys always produce a cache miss. Proactive approach eliminates above limitation as it would prefetch data beforehand by predicting the cacheability with the use of a prediction model.

However, as per the authors, no system was found with an approach to cache data proactively. It is also identified that the current reactive approaches add a significant complexity in the design and implementation, hence the caching solutions with proactive approaches, would be

more complex than reactive approaches resulting in more effort and reasoning to be done in the design and implementation.

According to [15], there are three main benefits of using an application cache. Long-running computations can be avoided by using pre-calculated results on the same input, which can significantly improve the execution time of the application, allowing the application to process more requests per unit time. With the reduced per request time achieved with application caches, users will experience more responsiveness and in different workloads such as workload spikes, the application would still withstand the workload without disrupting the user experience. Application caches potentially reduce application servers and/or the database replicas. Due to the reduced resources, overall cost of the application infrastructure would be reduced.

2.1 Research Gap Identified

Research done in application caches, can be categorized in two areas mainly.

1. Frameworks that help developers to configure caches.
2. Frameworks when integrated, automatically handles what to cache.

In first area, application code is searched, or method calls are intercepted to find out where to implement caches.

In second area, the existing frameworks have one or more following characteristics.

1. The frameworks use complex algorithms and methods which might not be suitable for all the applications.
2. The frameworks act as a transparent layer between application and the cache so that the cache misses are also handled by the framework itself.
3. The frameworks are mostly integrated to database centric applications.

When general purpose cache framework is considered, the following properties are more suitable to have.

1. It should be simple enough to integrate to any application including existing applications.
2. The framework should not handle cache misses so that data retrieval is decoupled

from the cache framework. This would allow cache framework to be integrated to any kind of data retrieval not only to database retrievals.

3. Performance of the framework should be equal in every scenario not only in database centric applications.

By filling above three gaps, this research is expected to come up with a simple, easy to implement and integrate, general purpose, intelligent caching framework.

CHAPTER 3

PROBLEM ANALYSIS

Since currently caches are coupled with application logic, when a cache is integrated into a system, developers are required to implement what needs to be cached. However, when a cache and application logic are decoupled from each other, there needs to be transparency between both. Otherwise, there will be additional coupling added when a cache framework is integrated to a system.

Just adding another layer between the application and the cache would not be sufficient. As it would only add an overhead to the system. There needs to be a significant improvement when a caching framework is integrated. The best way to achieve that would be to have an increased hit ratio with/without a reduced cache size. As an example, when everything is cached, the hit ratio would be 20% and after integration with a framework, if the hit ratio can be increased to 30%, it would be beneficial for the overall application performance.

However, getting an increased hit ratio would not be an easy task as the system behavior and workloads can be changed overtime. One option would be to analyze history patterns (search patterns) of the system and configure the system accordingly. But this task would need to be done for each cache separately considering the context of cache usage. Also, when complex frameworks are integrated to the application, although it would increase the hit ratio of the cache, with the overhead added, there might not be a significant performance improvement.

Currently there is no general-purpose simple framework which can be used to automatically decide what needs to be cached and can be integrated to any application without putting in much effort.

METHODOLOGY

To make caches decoupled from the application logic, the proposed caching framework should be between the cache and the application.

High level diagram of how the proposed framework would work in a real-world application is shown below.

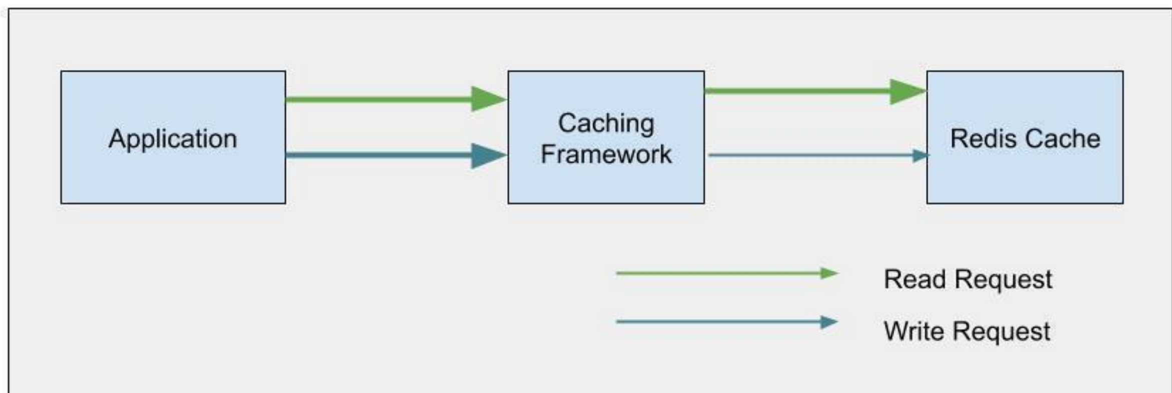


Figure 1 : High Level Design of the Solution

As shown above, if the framework is integrated, the application needs to go through the caching framework for any cache operations such as read cache and write cache. When the caching framework receives a read request, it will forward that request to the actual cache and return the results to the application in a synchronous way. When a caching framework receives a write request, it will return a response to the application whether the write request is accepted and asynchronous writes to the actual cache if cache entry needs to be added. (Reduced weight of the arrow between caching framework and Redis cache indicates that only a portion of the write requests received by the framework are sent to the Redis cache.)

However, the caching framework does not handle the cache miss scenario. When a cache miss is found, the application needs to handle how to load data and the caching framework assumes that all externally loaded data will be fed back to the caching framework. This is to reduce the coupling of the caching framework to the application.

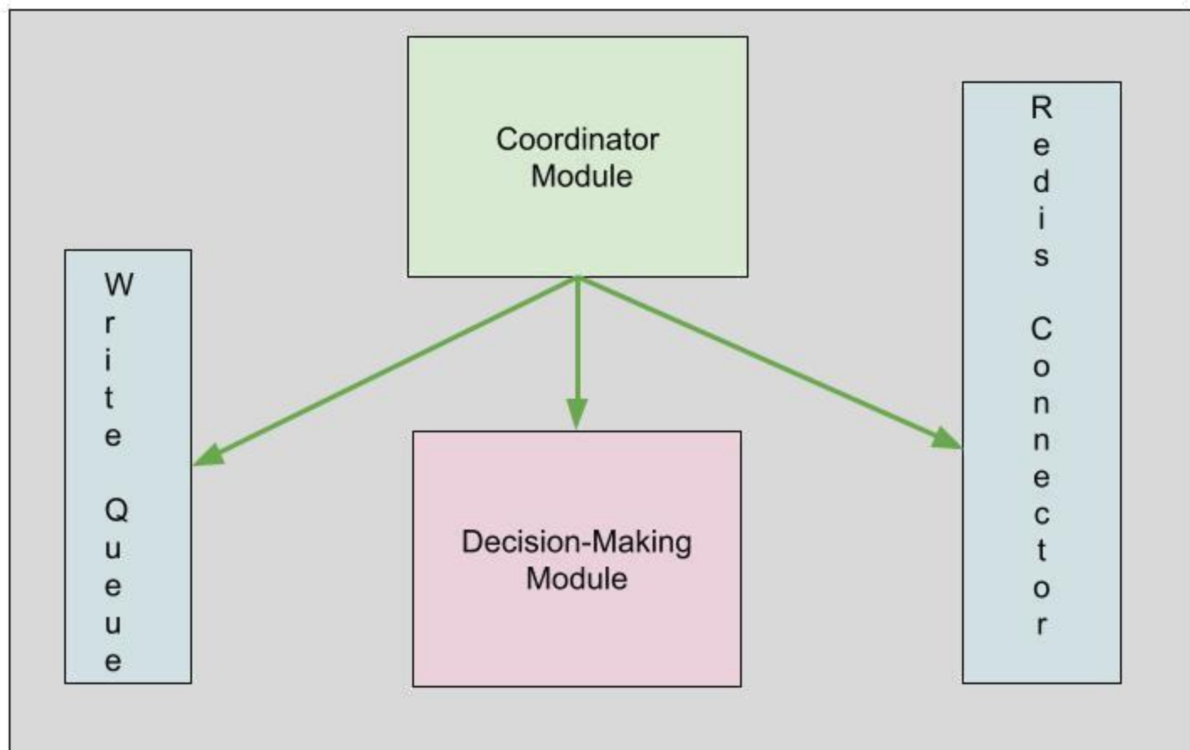


Figure 2 : High-level Architectural Design

As shown in the above diagram, the caching framework consists of four major components.

1. Write Queue

   This is used to asynchronously process write requests and reduce the overhead added to the application by the caching framework. The write requests from the application will be added to the write queue and will be processed by the Coordinator module.

2. Coordinator Module

   Coordinator module accepts any read cache requests and will retrieve the results via Redis connector module. This module is also responsible for processing write queue entries. When processing a queue entry, the decision-making module is invoked by the coordinator module to decide whether to cache or not.As the prediction model, Support Vector Machines (SVM) is used. If it needs to be cached, the coordinator module adds those via the Redis connector module.

3. Redis Connector

   This module handles actual communication to Redis cache.

4. Decision-making module

Since decision making of adding a cache entry is the main part of the framework, determining what should be cached is the key part of the project. Although a cache entry is the most recently added entry, if it is unlikely to retrieve that cache entry again in the future, there is no meaning of adding that cache entry. So there needs to be a worthiness limit which can be used to determine the decision. Therefore, the worthiness limit (boundary which needs to be exceeded by a cache entry to be eligible to be added to the cache) of adding a cache entry should be determined by the framework from time to time.

As a cacheability quantization method, frequency and size of the entry is used. Current cacheability is predicted with the use of previous 10 windows frequency and the data size. The objective is to maximize cache utilization. To achieve that, it is logical to cache high frequency and small size data. To decide on the optimal frequency boundary and data size boundary, the following three models are used.

A. 50-50 model:- frequency of the value >= 99th percentile of frequency or (frequency >= frequency mean & data size <= data size mean) , labeled as cacheable. More data is to be flagged as cacheable, hence high aggressive caching.
B. 75-25 model:- frequency of the value >= 99th percentile of frequency or (frequency >= frequency of 75th percentile & data size <= data size of 25th percentile) , labeled as cacheable. Less data is to be flagged as cacheable, hence moderate aggressive caching.
C. 90-10 model:- frequency of the value >= 99th percentile of frequency or (frequency >= frequency of 90th percentile & data size <= data size of 10th percentile) , labeled as cacheable. Lesser data is to be flagged as cacheable, hence low aggressive caching.

Three SVM models will be trained for the above three models and will be evaluated to determine the optimal model to be used.

CHAPTER 4

EVALUATION AND RESULTS

Evaluation is based on two main hypotheses which are,

1) Is the cacheability prediction model an accurate model to be used?
2) Can the proposed model improve performance of a system when integrated?

To measure the accuracy of the model, it is required to evaluate the training model attributes such as precision, recall and accuracy. To evaluate the performance improvement, it is required to run performance tests against a system with and without the framework. Since there are three model variations, both the evaluations (accuracy and performance) are done for each variation separately.

The dataset for the experiment was an online retail dataset[16]. This dataset is from UCI website which is a reputed Machine Learning Repository.70% of the dataset was used for training purposes and 30% of the data set was used for evaluation purposes. Following approaches were taken to test above-described hypotheses.

To evaluate the accuracy of the model tests were carried out for the test data set.

Table 1: Accuracy of the models

	Precision		Recall		Accuracy
Model	**0**	**1**	**0**	**1**	
50-50 Model	0.93	0.81	0.84	0.86	0.87
75-25 Model	0.91	0.77	0.89	0.80	0.86
90-10 Model	0.91	0.76	0.98	0.42	**0.9**

As per the above table it can be seen that all three variations converge quite well and the 90-10 model has the highest overall accuracy. Also, performance of the framework was evaluated under following configurations:

1) No cache
2) Caching each data
3) with the proposed model.

Evaluation is based on the throughput of the different cache configurations and the baseline is considered as the throughput performance of the No cache configuration.

For the experiment, JMeter tool [17], was used to carry out the performance tests. All the cache configurations including proposed three SVM models were evaluated against uniform and non-uniform workloads, LFU and LRU cache eviction methods and different time to live values for cache entries. All the tests were run on the same computer which has an Intel i7-6500U CPU @2.50GHz and 8.00 GB RAM.

Figure 3 and Figure 4 show the performance and the hit ratio results obtained for the models under the uniform workload with least frequently used (LFU) eviction for high TTL values (TTL > sample test run) and Figure 5 and Figure 6 show the performance and the ratio results obtained for the models under the uniform workload with least recently used (LRU) eviction for high TTL.

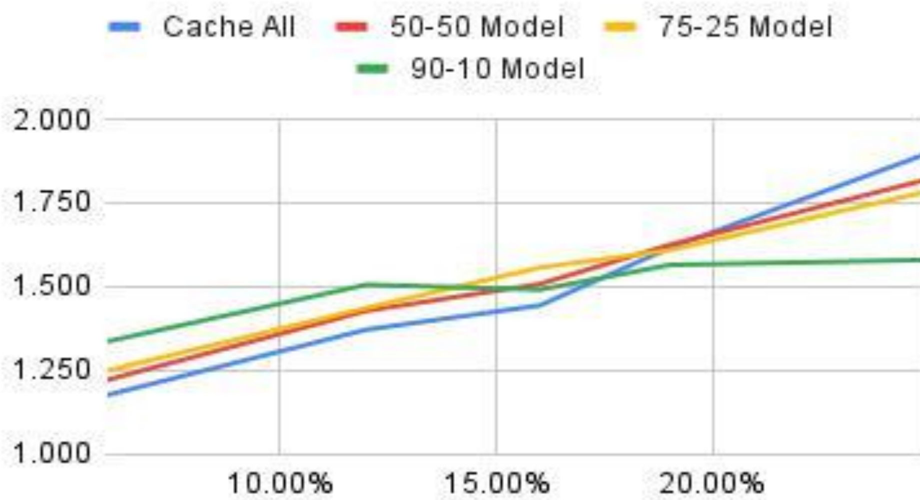


Figure 3: Performance under Uniform workload, LFU eviction and high TTL

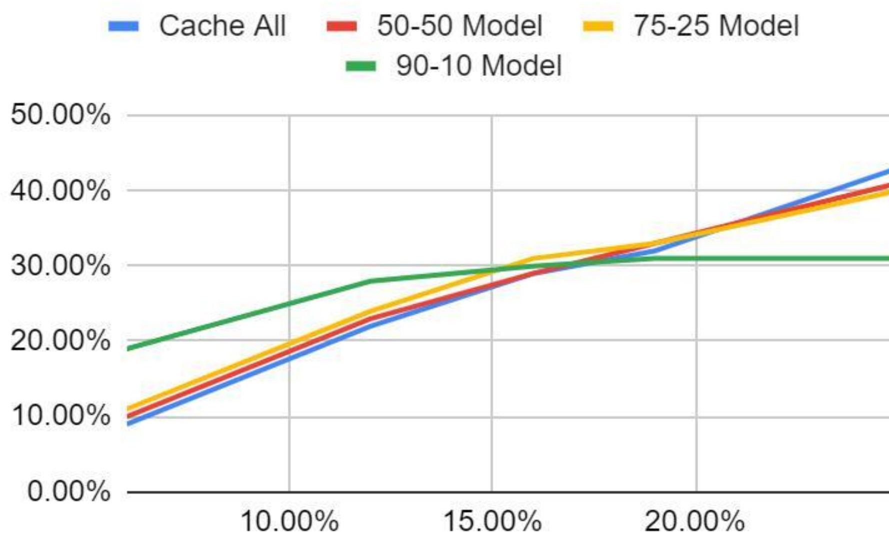


Figure 4: Hit ratio under Uniform workload, LFU eviction and high TTL

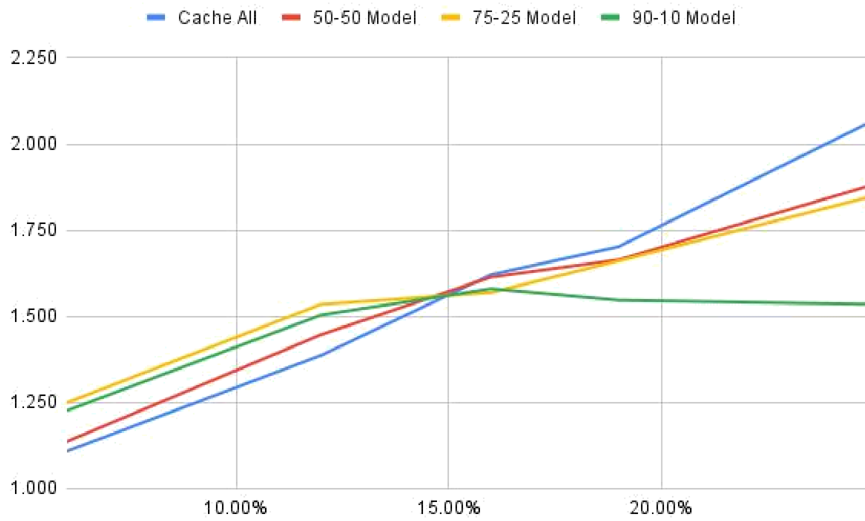


Figure 5: Performance under Uniform workload, LRU eviction and high TTL

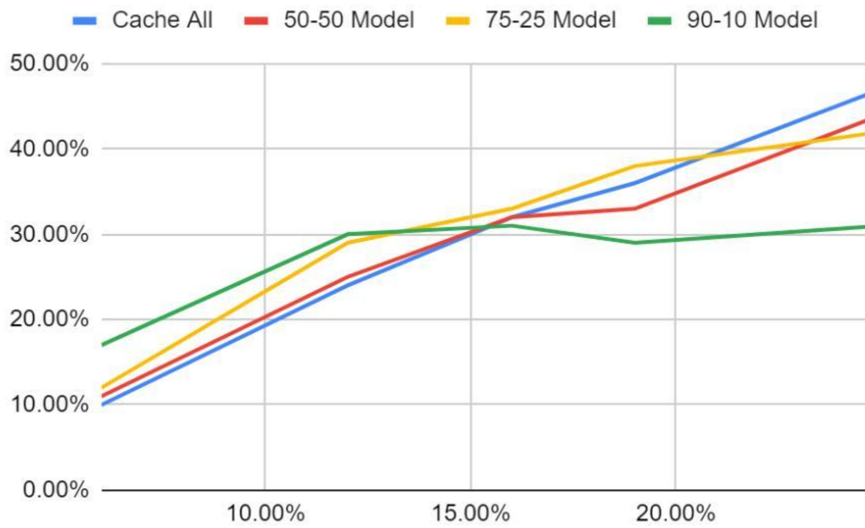


Figure 6: Hit ratio under Uniform workload, LRU eviction and high TTL

As per the above figures, irrespective of the eviction method, the performance and the hit ratio of the models are higher than cache all configuration, under uniform workload, when the cache capacity is below 15%. Also, the performance of the models is higher when the LFU method is used as the eviction method than the LRU method.

Figure 7 and Figure 8 show the performance and the hit ratio results obtained for the models under the non-uniform workload with least frequently used (LFU) eviction for high TTL values and Figure 9 and Figure 10 show the performance and the hit ratio results obtained for the models under the non-uniform workload with least recently used (LRU) eviction for high TTL.

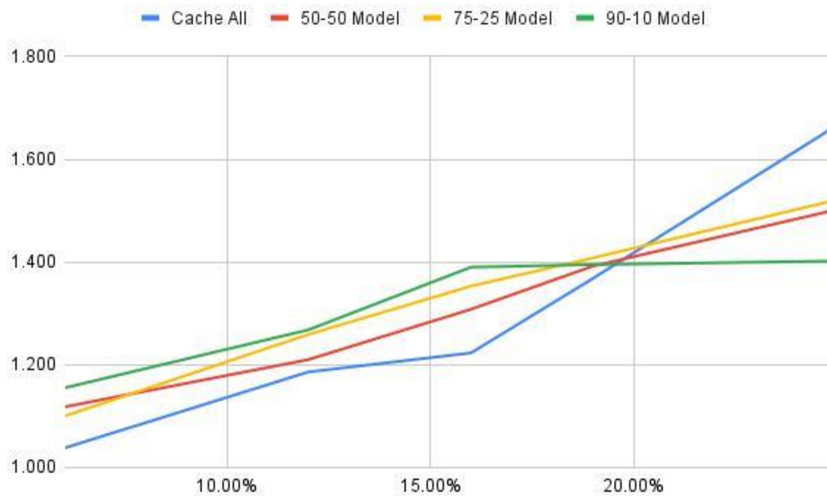


Figure 7: Performance under Non-Uniform workload, LFU eviction and high TTL

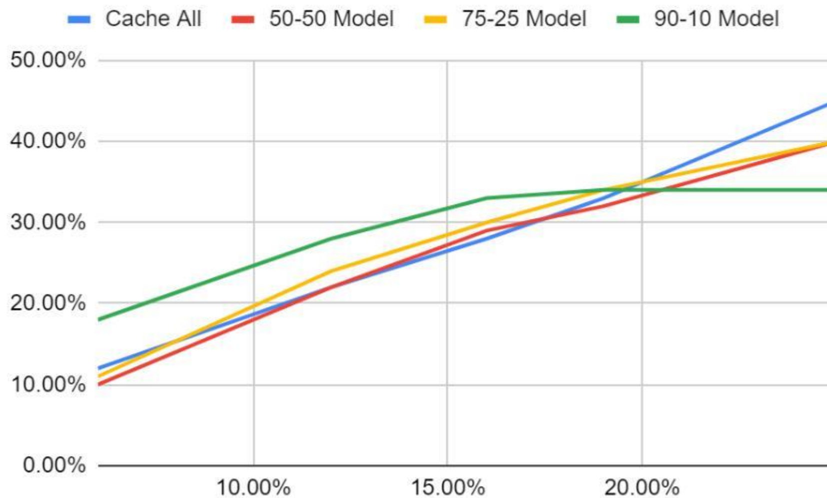


Figure 8: Hit ratio Non-Uniform workload, LFU eviction and high TTL

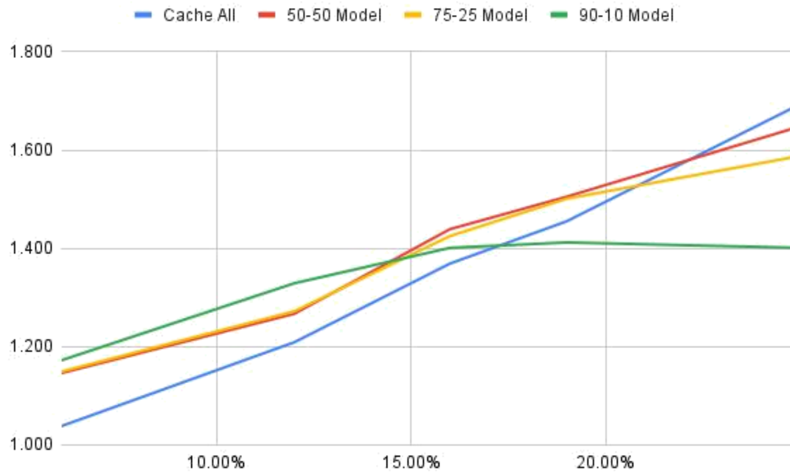


Figure 9: Performance under Non-Uniform workload, LRU eviction and high TTL

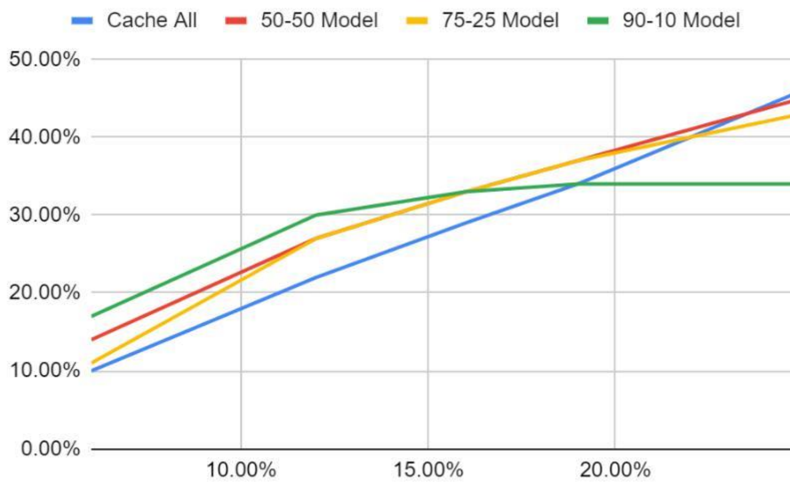


Figure 10: Hit ratio under Non-Uniform workload, LRU eviction and high TTL

As per the above figures, irrespective of the eviction method, the performance and the hit ratio of the models are higher than cache all configuration, under non-uniform workload as well, when the cache capacity is below 17%. Also, the performance of the models is almost the same for both eviction methods.

Figure 11 shows the performance of each model with different cache capacity, against the time to live value of the cache entries. The higher TTL values would facilitate higher performance of all three models.

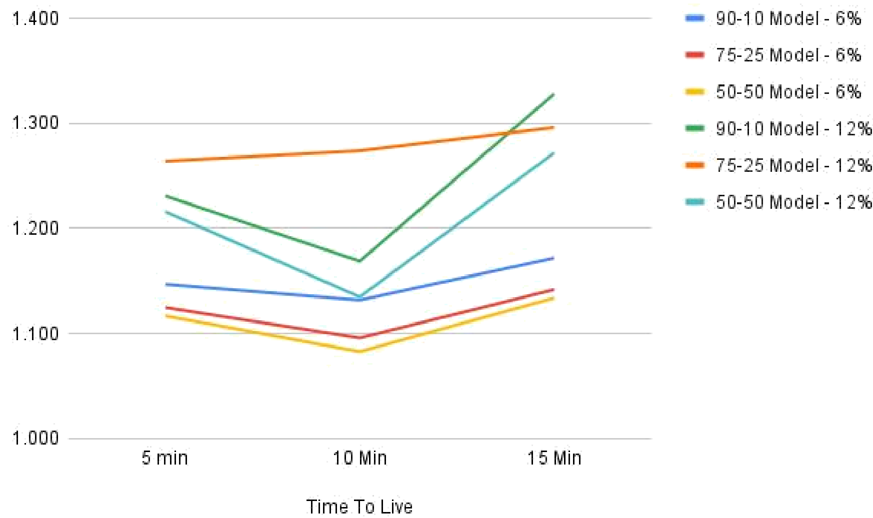


Figure 11 : Performance against Time to Live

Overall, the performance of the models is high due to the increased hit ratio of the models when compared to cache all scenarios. However, when hit ratios are almost same, cache all configuration works better due to the additional overhead added by the proposed framework. Additionally, when 90-10 model is considered, the performance and the hit ratio get flatten. This is due to the underutilization of the cache capacity as 90-10 model decides to cache very a smaller number of entries.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Application caches play a vital role in today's enterprise software industry as it could be beneficial to integrate more and more caches to obtain a higher throughput, low latency, and low infrastructure cost. However, when configuring the caches, developers are required to have in-depth knowledge of the domain and possible workloads. Also, when there's a limitation on the cache size which can be accommodated for a particular cache, cache utilization should be at the optimal level to have the maximum effect of introducing a cache. There are already available for general purpose cache eviction methods such as least recently used and least frequently used, however there's no general-purpose method for determining the validity of cache admission.

This thesis proposes a general purpose, intelligent caching framework which is based on the most recent frequency and data size. Since these metrics are readily available for any system and with the simplicity of the overall design, the proposed model would be a good candidate for a smooth integration for any existing system. The main idea behind the proposed framework is that the cache utilization can be improved by only adding entries which would potentially have a higher frequency and lower cache entry size. By adding only potentially high frequency cache entries, it would remove cache pollution as much as possible and improve the overall hit ratio. By only adding lower sized cache entries, increased number of cache entries would be accommodated in the cache for a particular time hence there would be maximum utilization of the allowed cache capacity.

The proposed framework uses a support vector machine classifier (SVM classifier) to automatically decide what to be cached. To minimize the overhead added with the framework, the cache retrievals are blindly done, and cache additions are done asynchronously. However, all the additions are evaluated for the cacheability decision hence some of the entries would not be added to the cache which would limit the cache pollution as much as possible.

There are three major parts in the proposed approach, a simple cacheability identification technique, a SVM model to automatically decide what to cache and a general-purpose caching framework which can be integrated to new or existing systems without much effort. The cacheability identification method introduces a general-purpose quantitative approach for simply determining what to be cached by looking at an already existing dataset. Since the cacheability identification method uses frequency and data size, cacheability for a given dataset can be easily determined. Based on the cacheability flagged dataset, a support vector machine is trained for cacheability decision. Support vector machine model is such that it only

uses previous ten windows frequency and cache entry size. By only considering recent windows only, it would allow the model to indirectly consider recency as well.

Three variations of the cacheability identification techniques (90-10, 75-25, 50-50 models) were evaluated for its accuracy of learning and performance against uniform and non-uniform workloads as well as for different time to live values. The overall accuracy of all three models is more than 85% hence models converge well in all three variations. Experiments were carried out for testing the performance of the caching framework for three different variations above, keeping the no cache configuration as the baseline. The results show that the proposed framework would improve the throughput up to 17% compared to traditional cache all configuration, when the cache capacity (the average number of cache entries can be accommodated out of all possible cache entries, for a given time) is under 15%.

Out of three variations, 90-10 model which labeled cacheable data for training when the frequency of the value >= 99th percentile of frequency or frequency >= frequency of 90th percentile and data size <= data size of 10th percentile, shows higher throughput in most cases compared to other two variations. Also, that same model has the highest overall accuracy when compared to other two models. But, the 90-10 model does not perform well when cache capacity is beyond 15% because the model is underutilizing the cache capacity. However, beyond this capacity 'cache all' configuration performs better in almost all the cases. Considering all above, it can be concluded that when cache capacity is less than 15%, 90-10 model can be selected among three variations and it would improve the overall throughput up to 17%, compared to traditional cache all configurations.

With the proposed work, there are some unanswered questions like whether it is possible to improve the accuracy of the model so that the precision and recall can be improved as SVM model used for the framework is only trained for default parameters. Also, would more lightweight classifiers with lees accuracy improve the overall performance by reducing the overhead added when deciding what to be cached? To address above questions, as future works, it would be worth investigating whether SVM model can be further improved to have a higher accuracy. In addition to that it would be worth investigating other lightweight classifiers such as Logistic Regression or Random Forest Regression, to determine whether using those classifiers overall throughput is improved or not.

APPENDICES

Appendix A: Results

Eviction Method	Load Type	Expire Time	Performance				Hit Ratio			
			Cache All	50-50 Model	75-25 Model	90-10 Model	Cache All	50-50 Model	75-25 Model	90-10 Model
LFU	Uniform	No Expiry	1.176	1.221	1.249	1.336	9.00%	10.00%	11.00%	19.00%
LFU	Uniform	No Expiry	1.372	1.428	1.437	1.506	22.00%	23.00%	24.00%	28.00%
LFU	Uniform	No Expiry	1.444	1.508	1.557	1.491	29.00%	29.00%	31.00%	30.00%
LFU	Uniform	No Expiry	1.616	1.625	1.609	1.565	32.00%	33.00%	33.00%	31.00%
LFU	Uniform	No Expiry	1.900	1.822	1.786	1.580	43.00%	41.00%	40.00%	31.00%
LRU	Uniform	No Expiry	1.108	1.136	1.249	1.226	10.00%	11.00%	12.00%	17.00%
LRU	Uniform	No Expiry	1.387	1.447	1.535	1.504	24.00%	25.00%	29.00%	30.00%
LRU	Uniform	No Expiry	1.621	1.614	1.569	1.580	32.00%	32.00%	33.00%	31.00%
LRU	Uniform	No Expiry	1.702	1.664	1.661	1.547	36.00%	33.00%	38.00%	29.00%
LRU	Uniform	No Expiry	2.068	1.882	1.849	1.535	47.00%	44.00%	42.00%	31.00%
LFU	Non-Uniform	No Expiry	1.038	1.118	1.100	1.155	12.00%	10.00%	11.00%	18.00%
LFU	Non-Uniform	No Expiry	1.186	1.210	1.259	1.268	22.00%	22.00%	24.00%	28.00%
LFU	Non-Uniform	No Expiry	1.223	1.308	1.353	1.390	28.00%	29.00%	30.00%	33.00%
LFU	Non-Uniform	No Expiry	1.368	1.392	1.408	1.395	33.00%	32.00%	34.00%	34.00%
LFU	Non-Uniform	No Expiry	1.666	1.502	1.521	1.402	45.00%	40.00%	40.00%	34.00%

LRU	Non-Uniform	No Expiry	1.038	1.146	1.149	1.172	10.00%	14.00%	11.00%	17.00%
LRU	Non-Uniform	No Expiry	1.209	1.267	1.272	1.329	22.00%	27.00%	27.00%	30.00%
LRU	Non-Uniform	No Expiry	1.369	1.439	1.425	1.401	29.00%	33.00%	33.00%	33.00%
LRU	Non-Uniform	No Expiry	1.455	1.505	1.501	1.412	34.00%	37.00%	37.00%	34.00%
LRU	Non-Uniform	No Expiry	1.693	1.648	1.588	1.401	46.00%	45.00%	43.00%	34.00%
LFU	Non-Uniform	5 min	1.063	1.117	1.125	1.147	11.00%	13.00%	13.00%	18.00%
LFU	Non-Uniform	5 min	1.170	1.216	1.264	1.231	24.00%	24.00%	26.00%	23.00%
LFU	Non-Uniform	5 min	1.338	1.342	1.371	1.218	30.00%	31.00%	32.00%	23.00%
LFU	Non-Uniform	5 min	1.407	1.437	1.417	1.215	32.00%	35.00%	35.00%	23.00%
LFU	Non-Uniform	5 min	1.582	1.508	1.456	1.238	45.00%	41.00%	37.00%	23.00%
LRU	Non-Uniform	5 min	1.036	1.158	1.153	1.164	11.00%	14.00%	15.00%	19.00%
LRU	Non-Uniform	5 min	1.141	1.335	1.308	1.218	25.00%	28.00%	28.00%	23.00%
LRU	Non-Uniform	5 min	1.348	1.442	1.415	1.185	32.00%	34.00%	33.00%	22.00%
LRU	Non-Uniform	5 min	1.502	1.495	1.456	1.178	37.00%	37.00%	36.00%	22.00%
LRU	Non-Uniform	5 min	1.759	1.617	1.494	1.171	48.00%	43.00%	37.00%	22.00%
LFU	Non-Uniform	10 Min	1.046	1.083	1.096	1.132	10.00%	12.00%	14.00%	19.00%
LFU	Non-Uniform	10 Min	1.215	1.135	1.274	1.169	24.00%	25.00%	26.00%	25.00%
LFU	Non-Uniform	10 Min	1.344	1.384	1.378	1.283	31.00%	32.00%	32.00%	28.00%

LFU	Non-Uniform	10 Min	1.487	1.285	1.449	1.253	35.00%	34.00%	35.00%	27.00%
LFU	Non-Uniform	10 Min	1.641	1.376	1.511	1.240	45.00%	40.00%	40.00%	26.00%
LRU	Non-Uniform	10 Min	1.084	1.039	1.112	1.138	11.00%	13.00%	14.00%	15.00%
LRU	Non-Uniform	10 Min	1.282	1.329	1.266	1.327	24.00%	29.00%	29.00%	28.00%
LRU	Non-Uniform	10 Min	1.376	1.421	1.459	1.451	32.00%	34.00%	35.00%	34.00%
LRU	Non-Uniform	10 Min	1.490	1.472	1.489	1.518	37.00%	38.00%	38.00%	38.00%
LRU	Non-Uniform	10 Min	1.781	1.634	1.556	1.649	48.00%	44.00%	41.00%	45.00%
LFU	Non-Uniform	15 Min	1.053	1.134	1.142	1.172	9.00%	13.00%	14.00%	19.00%
LFU	Non-Uniform	15 Min	1.226	1.272	1.296	1.328	24.00%	25.00%	26.00%	29.00%
LFU	Non-Uniform	15 Min	1.380	1.393	1.401	1.347	31.00%	31.00%	32.00%	31.00%
LFU	Non-Uniform	15 Min	1.470	1.442	1.467	1.348	35.00%	34.00%	35.00%	31.00%
LFU	Non-Uniform	15 Min	1.681	1.559	1.570	1.311	44.00%	40.00%	42.00%	30.00%
LRU	Non-Uniform	15 Min	1.030	1.109	1.132	1.168	11.00%	13.00%	15.00%	21.00%
LRU	Non-Uniform	15 Min	1.208	1.270	1.311	1.296	24.00%	27.00%	30.00%	29.00%
LRU	Non-Uniform	15 Min	1.378	1.405	1.438	1.281	32.00%	34.00%	35.00%	30.00%
LRU	Non-Uniform	15 Min	1.531	1.483	1.494	1.304	37.00%	38.00%	38.00%	30.00%
LRU	Non-Uniform	15 Min	1.712	1.584	1.568	1.301	48.00%	45.00%	42.00%	30.00%

REFERENCES

[1] Redis.io. 2020. *Redis*. [online] Available at: https://redis.io [Accessed 5 September 2020]

[2] Memcached.org. 2020. *Memcached - A Distributed Memory Object Caching System.* [online] Available at: https://memcached.org/ [Accessed 5 September 2020].

[3] Hazelcast. 2020. *Hazelcast | The Leading In-Memory Computing Platform.* [online] Available at: https://hazelcast.com/ [Accessed 5 September 2020].

[4] En.wikipedia.org. 2020. *Cache Replacement Policies.* [online] Available at: https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU) [Accessed 5 September 2020].

[5] En.wikipedia.org. 2020. *Cache Replacement Policies.* [online] Available at: https://en.wikipedia.org/wiki/Cache_replacement_policies#Least-frequently_used_(LFU) [Accessed 5 September 2020].

[6] Altinel et al, (2003) 'Cache tables: Paving the way for an adaptive database cache', *29th international conference on Very large data bases,* Volume 29. VLDB Endowment, pp. 718–729.

[7] Candan et al, (2001) 'Enabling dynamic content caching for database-driven web sites', *ACM SIGMOD Record*, vol. 30, no. 2. ACM, pp. 532–543.

[8] Choi et al, (2000) 'A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references', *4th conference on Symposium on Operating System Design & Implementation*, Volume 4. USENIX Association, p. 9.

[9] Junqueira et al, (2007) 'Admission policies for caches of search engine results', *International Symposium on String Processing and Information Retrieval. Springer*, pp. 74–85

[10] Chen et al, (2016) 'Cacheoptimizer: Helping developers configure caching frameworks for hibernate-based database-centric web applications', *2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM*, pp. 666–677.

[11] Sajeev, G. and Sebastian, M., 2013. 'Building semi-intelligent web cache systems with lightweight machine learning techniques', *Computers & Electrical Engineering*, 39(4), pp.1174-1191.

[12] Mertz, J., and Nunes, I. (2018). 'Understanding and Automating Application-level Caching'

[13] Fei et al, (2019) 'SACC: Configuring Application-Level Cache Intelligently for In-Memory Database Based on Long Short-Term Memory', *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 1350-1357.

[14] Mertz, J. and Nunes, I., (2017). 'A Qualitative Study of Application-Level Caching', *IEEE Transactions on Software Engineering,* 43, 798-816.

[15] Mertz et al, (2021). 'Satisfying Increasing Performance Requirements With Caching at the Application Level', *IEEE Software*, 38, 87-95.

[16] Archive.ics.uci.edu. 2021. *UCI Machine Learning Repository: Online Retail Data Set.* [online] Available at: https://archive.ics.uci.edu/ml/datasets/online+retail [Accessed 26 June 2021].

[17]Jmeter.apache.org. 2021. *Apache JMeter - Apache JMeter™.* [online] Available at: https://jmeter.apache.org/ [Accessed 30 August 2021].