NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Håvard Langdal

Cache replacement policies in NoSQL databases and full-text search engines

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

NTNU
Kunnskap for en bedre verden

Håvard Langdal

Cache replacement policies in NoSQL databases and full-text search engines

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The problem of replacing cache content in the most performant way to optimize system performance has been a popular research topic, originating from paged virtual memory systems. In most cases, the previous work focuses on algorithmic improvements without considering the environments and use cases of the systems. NoSQL databases and full-text search engines are trying to keep up with the ever-increasing amount of data they store and process, which requires active use of caching. Their performance is highly dependent on the cache replacement policy, which dictates which items should reside in the cache, and how it should replace items, once its capacity is reached. Building on existing work that spans from page replacement algorithms in paged virtual memory systems to modern adaptive cache replacement policies, it tries to approach some of the more overlooked issues in cache replacement policies, such as utilizing historical access patterns when adapting to workload changes and utilizing size or retrieval cost when replacing items. The size and retrieval cost implementations are simulated towards a variety of datasets in two different runtime scenarios, weighted and cost-based caching simulations, while for historical access patterns, a simulation with a loopy workload is performed to display the effects of a human expert preparing the system. The results suggest that utilizing historical access patterns may contribute to cache performance, but it is highly dependent on identifying trends in data access patterns, and also that the simplest extensions of state-of-the-art cache replacement policies contribute to higher performance.

Sammendrag

Problemet, eller utfordringen bak optimale måter å erstatte innhold i en cache har lenge vært populære tema innen forskning, som stammer fra sidet virtuelt minne. I det meste har foregående arbeid konsentrert seg på algoritmiske forbedringer uten å ta i bruk hvilke miljø cachen befinner seg i og hvilke bruksområder det miljøet har. NoSQL databaser og dokument søkemotorer jobber med å holde tritt med den stadig økende mengden data de skal lagre og bearbeide, som krever aktiv bruk av caching. Deres ytelse er avhengig av hvilken algoritme eller policy som blir brukt for erstatning av cachen når dens kapasitet er nådd. Gjennom tidligere arbeid som baserer seg på alt fra algoritmer for utskifting av sider i sidet virtuelt minne til moderne adaptive cache erstatnings policyer, prøver denne masteroppgaven å tilnærme seg probleme som ofte blir oversett innen cache erstatnings policyer, som for eksempel bruk av historiske data aksess mønstre for å tilpasse seg endringer i arbeidsmengden, samt å bruke størrelse av objektet eller kostnaden involvert i å hente objektet på nytt fra sitt primære lagringsmedium til utskiftning av cachen. Implementasjonene av cache erstatnings policyene som tar for seg størrelse og kostnad er simulert mot en mengde med datasett i to forskjellige kontekster, som er vektet caching og kostnadsbasert caching, mens for historiske data aksess mønstre, en simulering med vanskelig løkke lignende data viser effekten av at en menneskelig ekspert går aktivt inn for å forberede systemet. Resultatene antyder at å bruke historiske data aksess mønstre kan hjelpe i å opprettholde høy ytelse selv når utfordrene endringer skjer med arbeidslasten, og de enkleste endringene av topp moderne cache erstatnings policyer fører til høyere ytelse.

Preface

This master's thesis is the pinnacle of my work at the Norwegian University of Science and Technology. This work has been supervised by Svein Erik Bratsberg, during the spring semester of 2020. The thesis builds upon work already done in a pre-project, where cache replacement policies used in NoSQL databases and full-text search engines were analyzed with regards to their performance, design and use cases. The thesis is a summary of the implementation and results of enhancing cache replacement policies to better suit the needs of NoSQL databases and full-text search engines. Cache replacement algorithms originate from page replacement algorithms used in paged virtual memory systems. The thesis was inspired by the desire to understand how NoSQL databases and full-text search engines approach performance. Caching is often overlooked and it is common to find popular NoSQL databases and full-text search engines not utilizing state-of-the-art cache replacement policies capable of enhancing performance. The thesis assumes the reader has some prior knowledge of NoSQL databases, full-text search engines and what separates traditional relational databases from NoSQL databases. Furthermore, the work assumes the reader is familiar with concepts such as computational complexity and data structures.

I would like to thank the supervisor of this project, Svein Erik Bratsberg for insightful discussions and for having a genuine interest in the project and its results. I would also like to thank my girlfriend Malene for her patience and my puppy Todd for reminding me of the joys of spending time outside.

Håvard Langdal
Trondheim, June 9, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Caching has always been a critical part of computers and computer design. A cache dramatically reduces the access cost of any object by keeping it in a faster medium of access. With the rapid increase in popularity for NoSQL databases and full-text search engines, there has also been an increased focus on how these systems can perform better and cope with the ever-increasing amount of data they process. NoSQL databases and full-text search engines are deployed in a large variety of systems which process large amounts of data.

Based on the already performed literature review [1], presented in chapter 3, we are going to enhance cache replacement policies based on its findings. Traditionally a cache replacement policy focuses solely on the eviction, while recent advances in the field suggest that cache admission policies alongside with the cache eviction policies result in highly performant combinations. More specifically, we are going to experiment with historical access patterns, weighted caching and cost-based caching to enhance the performance of cache replacement policies which are sometimes found in NoSQL databases and full-text search engines. Many policies adapt their internals based on the recent past, which in many cases ensures high performance, but no publicly known policy or caching system utilizes historical access pattern to adapt to workloads as well as withstanding difficult workloads. Items in today's storage and search systems have different retrieval cost and size, meaning that some items are more expensive to retrieve than others. NoSQL databases and full-text search engines could utilize this to enhance performance.

1.1 Background and motivation

NoSQL databases and full-text search engines are complex data storage, search and processing systems that are designed to deliver high performance for a large variety of use cases. They do not provide the same data integrity guarantees as relational databases do, by sacrificing data integrity they are able to increase the performance. Nonetheless, most of the systems are still limited by the speed of auxiliary storage, with caching, they could overcome some of the performance limitations. NoSQL databases and full-text search engines also operate on results with different retrieval cost, which means that a simple query to fetch a small subset of results from a table or a key lookup has low retrieval cost compared to an aggregation query over a large table or a collection of documents. NoSQL databases and full-text search engines are extremely versatile and deployed in many different environments, which means that there is not a single cache replacement policy considered optimal for every system and environment. NoSQL key-value stores such as Redis and Memcached often used as caches. They need to be extremely performant, making exact implementations of cache replacement policies undesirable due to the algorithmic complexity it involves, nonetheless, sampled cache replacement policies are still capable of providing high performance. Cassandra, a NoSQL column store uses caching in several ways to optimize its read operations. One of them is keeping track of where partitions start, while another usage is to cache large parts of the data it stores. Both Apache Solr and ElasticSearch uses caching to store query results, which often comes with a variable computational cost, which none of the full-text search engines utilizes in their replacement policy. Choosing the right cache replacement policy is crucial to ensure high performance.

The optimal cache replacement algorithm may be different from deployment to deployment, or even from a minute to the next minute. This is why there is an increased focus on building a "silver bullet" cache replacement policy; a policy that dynamically adapts to the workload of the system to provide high performance.

As caching both in terms of hardware and software caches are for many systems a crucial part of obtaining high performance, optimizing them is necessary. The motivation is to increase the performance of NoSQL databases and full-text search engines by optimizing cache replacement policies. More specifically to enhance policies that suit better suit the needs of NoSQL databases and full-text search engines. Optimizing cache replacement has enormous potential to reduce operational costs, latency, energy consumption, bandwidth consumption and overall computation power. Cache replacement algorithms are present in every cache, and even though this work is focused around the use cases of NoSQL databases and full-text search engines, the policies apply to every software cache.

1.2 Goals and research questions

The goal of this thesis is to find ways to optimize cache replacement algorithms even further, which again will contribute to NoSQL databases and full-text search engine performance. The goal and research questions are based on findings in the literature review done as a pre-project for this thesis [1].

Goals *Increase the performance of cache replacement policies by utilizing item size, retrieval cost and historical access patterns to better suit the needs and performance requirements for NoSQL databases and full-text search engines.*

Reasearch question 1 (RQ1) *How could weighted hit rate be enhanced without sacrificing hit rate in cache replacement policies?*

Research question 2 (RQ2) *How could historical access patterns be utilized in adaptive cache replacement policies to enhance cache hit rate?*

For RQ1, based on the literature review presented in chapter 3 we will attempt to take already well-functioning models and architectures for cache replacement policies and enhance them before comparing them towards other policies. While for RQ2, based on the literature review, we have learned that no publicly available policy or data is available that utilizes historical access patterns, which makes the task of utilizing it even more complex.

1.3 Thesis Structure

- Introduction - The introduction of the project, this includes background, motivation, goals and research questions and thesis structure.
- Background - This chapter introduces enough background theory for the reader to understand choices made throughout the project. NoSQL, full-text search engines, virtual memory, caching and counting. This chapter is partly based on project work [1].
- Related work - The chapter introduces well-established cache replacement policies, policies considered state-of-the-art and policies in the NoSQL databases and full-text search engines. This chapter is partly based on project work [1].
- Design and architecture - Describes the implementation done in other to answer the research questions and perform experiments.
- Experiments - Presents test cases, setup and the results.
- Results and analysis - Contains the evaluation and discussion of the results.
- Conclusion - Concludes the discussion and research questions.
- Future work - Describes how to take this work further, both specifically for this project, caching in general and caching for NoSQL databases and full-text search engines.

Chapter 2

Background

In this chapter, relevant background theory is presented. More specifically, NoSQL databases, full-text search engines, virtual memory, caching and counting are explained. The purpose is to provide enough background information to understand how the different systems have different performance requirements as well as all the necessary background knowledge when it comes to caching.

2.1 NoSQL

The term "NoSQL" or "not only SQL" refers to databases or data stores created to store large amounts of unstructured data. Since 2009 the term has been prevalent throughout the software industry. Big technology companies, such as Amazon and Google were the early adopters of NoSQL, but eventually, more and more companies started to adopt it as well. NoSQL databases have different key features or characteristics than traditional relational databases [2].

- Horizontal scaling - can easily run in clusters of computers
- Ability to replicate and partition - high availability for data and increased throughput with horizontal partitioning
- Flexible schema - schema can be changed to support future requirements without schema migration process
- High availability - systems have high availability, meaning they operate on a high level of operational performance and availability

Traditional relational databases like PostgreSQL and MySQL are designed after the ACID properties, meaning they provide the user with guarantees of Atomicity, Consistency, Isolation and Durability. Some databases within the NoSQL category are sometimes referred to as BASE, which was popularized by Brewer [3] is an abbreviation of Basically Available, Soft state and Eventual consistency.

- Basically Available - the system guarantees availability in the terms of the CAP theorem [3].
- Soft state - the state will change unless maintained by the user.

- Eventual consistency - changes propagate the system after the request is terminated.

NoSQL databases are meant to provide efficient storage and indexing for systems that can sacrifice ACID properties for a simpler, but more performant interface.

2.1.1 Key-value stores

Key-value stores are the NoSQL databases that have the simplest interface provided to the user. As a client it is possible to get a value for the key, put a value for a key and delete the key from the key-value store. Some popular key-value stores like Redis [4] uses hashing to accomplish constant lookup time on key, while others like Foundation DB [5] have their keys ordered, meaning they need to search for the keys, which increases the complexity of point lookups.

Due to the simplicity of the API provided, key-value stores are usually found in different parts of larger systems. Even though key-value stores have a limited API, they are still being used to solve a large variety of problems, some of which are listed below [2].

- Session data
- Caching
- User profiles
- Preferences
- Shopping cart data

Redis

Redis is a remote in-memory key-value store capable of storing more advanced data types than similar key-value stores. Redis is found in a large variety of systems often performing tasks that require low latency. Of the use cases mentioned in chapter 2.1.1, session management and caching are common. Due to Redis supporting a large variety of data types, it is often referred to as a data structures server, which, compared to simpler key-value stores makes Redis the tool of choice when selecting a key-value store, especially if the use cases are more advanced than values as strings [6].

Memcached

Memcached, originally implemented by Fitzpatrick is a high-performance distributed in-memory key-value store which is intended to reduce latency in applications with a shared distributed cache architecture. Furthermore, Memcached is generic, thus it is applicable to many different use cases. Memcached design philosophy is simplicity, the outcome of this design philosophy is listed below.

- Unlike Redis, Memcached does not support advanced data types as values, which means values passed to Memcached needs to be pre-serialized.
- Memcached relies on the client knowing which server to read from or write to.

- Memcached nodes are disconnected from each other, which means there is no support for replication.
- Cache invalidation is done on a per-node basis, each node invalidates the data it owns.

Memcached uses a variant of the segmented Lru with maintainer threads running in the background. The core functionality of the Segmented LRU is explained through a state machine which can be seen in figure 3.5 [8].

2.2 Full-text search engines

Full-text distributed RESTful search engines such as Elasticsearch [9] and Apache Solr [10] inhibit many of the same system characteristics as NoSQL databases. Instead of exact match queries, their purpose is to serve clients with results based on how relevant it is to a query. Furthermore, both Elasticsearch and Apache Solr are distributed search engines often found to be running in clusters, with sharding to replicate and partition data for high performance and resiliency. A full-text search engine has an index structure, which is used to map terms to documents. The collection may either reside on disk or in memory, dependent on its size.

2.3 Virtual memory

In the early days of electronic computing, fast-access storage was extremely expensive, which led to a hierarchical organization of memory. This organization usually has at least two levels; "main memory" and "auxiliary memory". A program's instructions and data need to reside in main memory to be referenced. This introduced the challenge of keeping both instructions and data with the immediate likelihood of being referenced in main memory, which is often called the *storage allocation problem.* The problem was solved by dividing the program into a sequence of segments, which would be swapped from main memory to auxiliary memory.

Manually swapping segments was manageable for a while, but with the increased complexity and high(er) level programming languages, a better way of handling swaps, or overlays was needed. This is where virtual memory management comes in. Virtual memory abstracts away the need for manual memory management by having the memory management unit (MMU) map virtual addresses into physical addresses. Virtual memory enables programs to exceed the size of available physical memory [11].

2.3.1 Paged virtual memory

Paged virtual memory is an implementation of virtual memory which divides the virtual address space into equal size blocks of addresses called pages. In paged virtual memory, page tables are used by the MMU to translate virtual addresses to physical addresses. A page fault occurs when a requested page is not in RAM. The operating system, which has now gained control over the program, must then find the requested data, obtain an empty page frame in RAM, load it into

RAM, update the page table and return control to the program [11]. A page replacement algorithm decides which pages should be written to disk when a page needs to be allocated when a page fault occurs, and if there are no available free pages or some threshold is limiting [12].

Page replacement algorithms

Page replacement algorithms make replacement decisions based on the information about page accesses from the hardware. Page replacement algorithms are evaluated on how they contribute to lowering the runtime of a program. Most page replacement algorithms share the same characteristics as cache replacement algorithms, which is explained in chapter 2.4 and more implementation details in chapter 3.1. Page replacement algorithms typically have very high-performance requirements, both in terms of the algorithmic complexity of the policy itself and how it contributes to lowering the runtime of a program. Of the many page replacement algorithms, some of the most fundamental ones are explained below [11].

FIFO

In FIFO the pages are maintained in a queue-like or linked list data structure. New pages are placed in the back, and pages in the front are replaced due to them being oldest.

Clock

Clock enhances FIFO with a second-chance approach by not altering the internal data structure. It accomplishes this by maintaining a clock of reference bits, which is associated with the reference of items in the cache. If a reference bit is set for an item, the item is skipped and the reference bit is reset before it moves on to the next item without having to add the page to the queue again. [13] Clock is further explained in chapter 3.1.4.

LRU

In LRU shares similar design as FIFO, but if a page is referenced while already in the queue, it is moved to the back of the queue. The cost of altering the queue at each request is too costly for high-performance systems, which is what Clock tries to do by approximating LRU without the need of excessive updates to the queue. Further explained in chapter 3.1.1.

Not frequently used

In not frequently used, counters are introduced. These counters are incremented if the page is referenced, and when a replacement is needed, the page with the lowest counter is paged out. Pages that are heavily used for a short period before the program changes access patterns may occupy space needed for other pages due to their high counters. The issue with high counters may be solved with ageing the counters.

2.4 Caching

Caching refers to data having its access cost reduced to obtain it again faster. Traditionally, it does so by storing it in a faster medium of access. A cache could be both hardware and software, it could be imported as a library in an application, or it could be a separate server. Caches attempt to capture data access patterns by prioritizing to keep the assumed most relevant items in the cache, and if the cache becomes full, replace the assumed less relevant items. Inserting items that are not relevant could lead to *cache pollution*. Cache pollution occurs when new irrelevant items are inserted into the cache, forcing more relevant items to be replaced [14].

2.4.1 Introduction

Traditionally, page replacement in virtual memory systems, described in chapter 2.3.1 was the predecessor of caching. Page replacement and hardware caching, and more specifically CPU caches have been the primary target of research on caching and cache mechanisms. CPU caches are one of the most vital parts of a computer in terms of obtaining high performance. On the other hand, software caches are not an equally established field of research. However, since the mainstream adoption of the internet in the 2000's there has been an explosion in terms of performance-sensitive applications and data. Which again results in an increased focus on how to make these applications as performant as possible.

Complex NoSQL databases such as Apache Cassandra and Apache HBase are examples of performance-sensitive applications that depend on caching to obtain desirable performance. Today they are using a software cache that resides in memory to reduce bottlenecks. The NoSQL databases mentioned above are excellent examples of systems with varying workloads, meaning they are often found in read-heavy systems, as well as write-heavy systems. In most cases, it is a combination of both, which again, increases the complexity and the performance requirements for components such as a software cache.

Traditionally, application servers have also applied some form of in-memory cache to avoid unnecessary queries to the primary database, but with the emergence of the cloud, modern web applications are often deployed in container environments, to scale dynamically based on the load. Which again leads to several containers, each containing an instance of a web application having their in-memory cache, and possibly their version of cache contents. This is why distributed caches such as Redis and Memcached are used. They are NoSQL key-value stores often deployed next to an application back end server. Keeping the cache and the back end servers separated also increases the systems ability to scale its components individually.

Even the simplest application could benefit from caching just as much as a complex NoSQL database could. This means that introducing caching into a system is a well-known approach to increase the overall performance of the system [15]. Because if applied smartly, it could prevent the system from doing unnecessary work and reduce bottlenecks. The need for caching varies from use case to use case, a database cache will usually not have the same requirements as an application cache. Even though cache design varies from use case to use case, the fundamental goal does not, which is to offer a faster way of accessing hot items to reduce latency [16].

2.4.2 Cache types

As mentioned, a cache can both be a hardware cache or a software cache. Hardware caches are often divided into CPU, GPU and buffer caches. Software caches are often divided into disk, web, distributed, database, memoization and data caches. Disk caches are used to cache disk blocks to reduce the number of disk operations. A disk cache is usually implemented using a portion of the systems main memory for caching disk pages [17]. Web caches were introduced as a way to reduce bottlenecks by caching webserver content such as web pages and images. Traditionally these are either stored with the client, for instance in its web browser, or in front of the webserver [18] [16]. Distributed caches are often used to cache database- and web session data (database caching). As the name suggests, it is a distributed cache, meaning that it is accessed over network connections. Distributed caches have become viable due to the decrease in main memory costs and increased network speeds. As mentioned in the previous chapter, keeping the cache distributed helps with separation of concerns. Similar to data caches, distributed caches are often considered application-level caches, meaning that they are often referenced directly from within the application code [19].

Database caches are quite similar to disk caches, but they could cache data on a finer level of granularity than blocks, to help reduce the number of disk I/O operations. [20] Memoization techniques are simply storing the result of an expensive operation to prevent it from being unnecessarily computed. Memoization and general-purpose in-memory caching share goals, but memoization is often found to be more of an optimization technique than caching [21]. General-purpose data caches or application-level caches is all about storing results or data that is likely going to be needed again, directly from within the application's or system's code. Even though most of the caching is in-memory, it is useful to label software libraries, embedded directly from application code as a type of cache. Caffeine is an example of such a library, which is discussed in chapter 3.4.5 [22] [23].

2.4.3 Principle of locality

Caches are designed after the principle of locality, which is fundamental in the world of computer science. More specifically, the principle is fundamental to obtain high performance, especially in CPU caches, but it is also applied in memory pre-fetching and branch predictions in execution pipelining. The principle of locality refers to processing units are more likely to access the same memory locations over a short period. More intuitively explained, keeping items that are recently used and will most likely be used again shortly in as fast as possible storage reduces latency and resource usage. The principle is often divided into two types, temporal and spatial locality. Temporal locality is based on that if a memory address is used, it is most likely going to be used again in a short period, thus keeping it in faster storage is more performant. Spatial locality is based on that if a memory address is used, the neighbouring addresses are most likely going to be accessed shortly, thus loading them into faster storage is more performant [24] [25].

2.4.4 Cache replacement policies

Cache replacement policies have adopted the work on page replacement and adapted it into caching. This means that several cache replacement policies are page replacement policies. The only difference between a page replacement policy and a cache replacement policy is its use case and environment.

When a cache is full, it needs to make room for new items, it does so by replacing content based on some policy, where the policy tries to replace items that are less likely to be referenced shortly. For replacement, it is common to use eviction policies, but new approaches to cache replacement with very convincing performance suggest that the most optimal approach is to use an admission policy next to the eviction policy. The theoretically optimal cache replacement algorithm is called Bélády's optimal algorithm/clairvoyant algorithm, which predicts future access patterns to reduce cache misses. The ultimate goal of cache replacement policies is to keep relevant items in the cache while evicting items that are not considered relevant anymore. To capture relevancy of the items, it is common to use the recency and frequency of the items [15]. State of the art cache replacement policies, which are presented in chapter 3, are able to capture both recency and frequency by combining well-established methods for cache replacement.

In addition to capturing both recency and frequency, cache replacement policies strive to withstand frequent bursts of references to non-temporal data (called scans). Such scans are also known to cause cache pollution, which is when a cache is inserting useless data, forcing more useful data out of the cache [26].

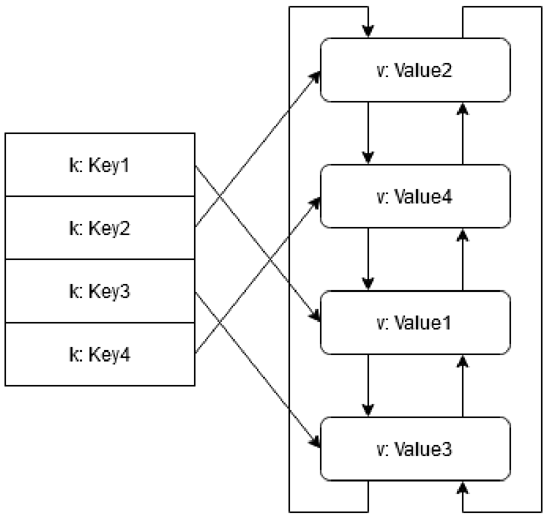

Figure 2.1: A simple software cache of 4 items, where each key in the lookup table points to a node in the doubly-linked list containing the data. Value2 is the head of the eviction list, while Value3 is the tail. When a new item is to be inserted into this cache, it needs to make room by evicting an item. The item is chosen based on the cache replacement policy.

2.4.5 Cache eviction policies

A cache eviction policy is a policy that determines which item to evict from the cache when its capacity is reached. A cache eviction policy is often implemented in such a way that the algorithm determines the internal ordering of the cache data structure, such as a min-heap for evicting the item with the lowest frequency. The internal data structures of the cache should complement the cache eviction policy to keep the algorithmic complexity low. In figure 2.1, if we employ the FIFO eviction policy, the cache item with key Key3 and value Value3 will be removed.

2.4.6 Cache admission policies

A cache admission policy usually co-exists with a cache eviction policy. Its purpose is to evaluate a candidate towards one or more victims to decide whether to admit the item into the cache, or not. One benefit of employing an admission policy is that it could contribute to withstand scan-like workload changes which usually causes some kind of cache pollution. Cache admission policies implementations span from simple thresholds to more advanced reference frequency estimators [27].

2.4.7 Capturing data orthogonality

Modern caches attempt to capture both the recency and frequency of data access patterns. This is usually done by modelling the cache into separate parts or by using a combination of admission and eviction policy. The combination of policies should be orthogonal to each other to capture both recency and frequency. An example of this is the Caffeine caching library, which uses W-TinyLfu for admission and S4LRU for eviction [23] [28].

2.4.8 Cache internals

Keeping admission and eviction separate from each other follows the separation of concerns principle. A cache is often a combination of several data structures, in figure 2.1, the FIFO cache consists of a map with pointers to elements that are a part of some linked data structure such as a doubly linked list. Data structures used in caches are carefully selected based on the algorithmic complexity of doing lookup, insertion and deletion.

2.4.9 Cost-based cache replacement

Policies such as LFU relies on a data structure that sorts its items based on an item attribute or metadata associated with the item. In LFU, this attribute is the frequency and the data structure is often a min-heap. Items admitted should be stored as long as their total cost is contributing more to high performance than admitting another item or several items instead, which is a difficult and relatively unexplored issue for NoSQL databases and full-text search engines.

2.4.10 Caching with key-value stores

A popular use case for NoSQL key-value stores is caching, more specifically application caching, where the purpose is to utilize the key-value store's fast access methods to retrieve content in a highly efficient matter from the application to both improve end user experience and relieve the primary database load. Redis and Memcached are ranked most popular pure key-value stores intended for caching on db-engines.com [29], they are also open source, which makes it easier to dig into the underlying mechanics. Redis and Memcached are further discussed and explained in chapter 3.4.

The principle of locality is prevalent in key-value stores at a higher level of abstraction than hardware caches, meaning that the same access optimization principles are applied to key-value caches, on key-value items. Of the two underlying types of the principle of locality, the temporal locality is prevalent in key-value stores. Items read from the primary database are loaded into the cache for faster access, because they are most likely going to be referenced again.

2.4.11 Caching in full-text search engines

Caching in full-text search engines is done heavily with the help of the file system cache, but also some specific cache strategies are also required to speed up the processing of results. In Elasticsearch, one utilization of software caching is done to cache shard requests [30]. The shard request cache comes to play when a query requires every involved shard to compute the results locally before responding to the coordinating node. Each shard then caches its results locally, possibly preventing the query from being processed again.

2.4.12 Caching libraries

While having a shared distributed cache as a standalone server is very useful for applications that need to scale dynamically, it is not suitable for every system. For instance, a horizontally partitioned NoSQL database, such as Cassandra does not rely on a shared, distributed cache. Instead, it uses Caffeine to solve in-memory caching. Caffeine is further explained in chapter 3.4.5.

2.5 Counting

Counting is essential in many high-performance systems, such as big data streaming applications. Counting precisely is costly, to avoid this cost, approximate counting has established itself as the way to approach this issue. Some popular ways of counting approximately are by using counting bloom filters or count-min sketches. These data structures are called probabilistic because they may be inaccurate due to hash collisions [31].

2.5.1 Count-min sketch

Count-min sketch (CM sketch), created by Cormode and Muthukrishnan is an probabilistic data structure that offers sub-linear space consumption [32]. CM Sketch is a frequency table for items/events in a stream or continuous flow of data. It applies several hash functions on incoming items to organize the hits in a sketch. A sketch is a matrix, where the number of rows is equal to the number of hash functions chosen, and the row width is equal to the maximum output of the hash functions. For a new item, CM Sketch applies all the hash functions, and then for each result of the hash operation, the cell is incremented. CM sketch approximates because of hash collisions. Point queries for an item towards CM sketch is answered by hashing the item and then extracting all the frequencies before applying the min function.

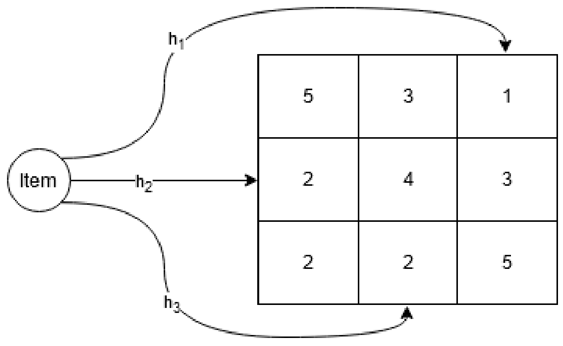

Figure 2.2: A small Count-in sketch illustration. The new item is going to be inserted, and it is hashed into sketch index 2 from h_1, index 0 from h_2 and index 1 from h_3. Point lookup for the item returns 1.

2.6 Motivation

Cache replacement policies are prevalent in countless both software and hardware components. Researching them, and understanding why some policy may outperform another policy and for which environments this is true is crucial to create high performance NoSQL databases and full-text search engines. Research conducted on cache replacement policies are usually focusing on algorithmic improvement, usually striving to improve theoretical cache hit rate, while few focus on working with the specific environments that NoSQL databases and full-text search engines often find themselves in. The environment of NoSQL databases and full-text search engines are filled with different types of data such as query results, I/O blocks and partition metadata, to mention a few examples. NoSQL databases and full-text search engines are usually parts of larger deployments, organized as clusters, which opens up the opportunity for doing the background processing required for capturing historical access patterns and utilize them in optimizing cache performance.

- Improving cache hit ratios is in many cases free performance
- Even for local caches, there is often a high cost related to a cache miss and retrieving the item from its storage medium
- Only a subset of item metadata is used when replacing cache content
- Improving weighted/cost-based cache hit rate could lead to better overall system performance compared to focusing entirely on cache hit rate

Chapter 3

Related work

This chapter presents the majority of the literature review performed as a pre-project performed during the autumn semester 2019 [1]. The review itself focused on both basic policies and state-of-the-art policies that are often used in NoSQL databases and full-text search engines.

3.1 Well-established cache replacement policies

Although they are not state of the art, well-established cache replacement policies such as LRU and LFU are still widely used in NoSQL databases and software artefacts throughout the industry. They are also commonly used as building blocks for more advanced policies, making it essential to understand their design and use cases.

3.1.1 Least/Most Recently Used

The Least Recently Used cache replacement policy evicts the items that have been prevalent in the cache longest without being referenced. LRU caches can be thought of as a singly- or doubly linked list, where if an element gets referenced, it gets re-linked to the head and if not, it will eventually be unlinked off the tail of the list. The LRU policy does not require a lot of computation or maintaining of metadata, it does, however, require altering of the internal data structure maintaining the order of access. Since every access results in the data structure being altered, lock contention is an issue, which is why LRU was considered not performant enough for page replacement. The time complexity for LRU is O(1) for insertion and deletion. The LRU cache is not scan resistant, making it unsuitable for workloads with a lot of large sequential accesses. However, the LRU is widely adopted in many types of caches, it is also often used as a baseline measure when evaluating new eviction policies.

The Most Recently Used cache replacement policy evicts the most recently accessed items when necessary. MRU is usually found in scenarios where old items are more likely to be accessed than new items. Because of its use case in file caches and RDBMS caches, it is not found in many key-value stores or general-purpose caches [20].

3.1.2 Least/Most Frequently Used

The Least Frequently Used cache replacement policy evicts items that have the lowest reference count. A cache using LFU needs to keep track of how many times the different items are referenced, which compared to the LRU, makes a pure LFU implementation more resource-demanding to maintain. Maintaining this information is often used by using a min-heap, which has an O(log(n)) time complexity both for inserting and deleting items. An article by Shah *et al.* describes an O(1) implementation of the LFU algorithm, where each key in the lookup table points to an element in a linked list, where every element accessed by the lookup table has a parent, which they are naming frequency node, which also is part of another linked list. The paper fails to mention concurrency support of the proposed data structure scheme, which for many real-world applications is both critical and necessary to obtain the required performance [33]. LFU implementations often have issues with adding new items into the cache because of already existing items having a high reference count, making the cache evict recently added items. Bulk accessing items are also known to cause issues with items occupying unnecessary space in the cache. Real-world implementations of the LFU policy usually have some sort of mechanism to make sure that no item has life-time tickets for a spot in the cache. For Redis, this mechanism is to halve the reference counter for the item from time to time. Also inserting items with a low reference count immediately makes them a candidate for eviction. Redis solves this by having an initial frequency count larger than zero, to give the item a fair chance to survive and build up hits. A consequence of having items set with a start frequency other than 0, or the currently lowest frequency in the cache, is that it could be less scan resistant, which for other NoSQL databases and full-text search engines could become a bottleneck [34].

The Mfu works similarly to Lfu, but instead it evicts the most frequently used items, which will only be performant in very specific deployments.

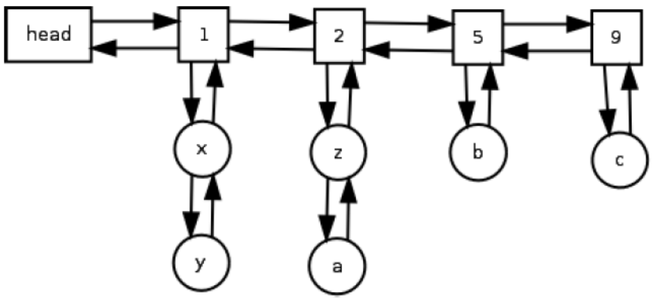

Figure 3.1: The O(1) LFU layout proposed by Shah *et al.* [33, p.4]. Each key in the lookup table points to a linked list item, where the parent of that item is a so called frequency node. In this example, x and y has frequency of 1, z and a has frequency 2, b has frequency 5 and c has frequency 9.

3.1.3 Segmented LRU

The segmented LRU or SLRU captures both recency and frequency by distinguishing between items recently accessed more than once, and items recently accessed only once. The SLRU manages this by utilizing two fixed LRU segments, where one acts as a probation segment and one acts as a protected segment. New items enter the probation segment, and if referenced when in the probation segment, it is moved to the protected segment. If either of the segments become full, their items are evicted LRU style, but items evicted from the protected segment end up in the probation segment again [35].

3.1.4 Clock

The Clock page replacement algorithm is highly suitable for caching due to its design for high performance and a minimum amount of updates to the cache data structure. Clock evicts items based in insertion order, like FIFO. Different from FIFO, it has a second chance mechanism that does not evict items recently accessed. This is maintained by a Clock like structure, which essentially is an array that maintains a bit for each item in the cache. If an item already existing in the cache gets updated, its bit in the Clock gets set, so whenever the item is to be evaluated for eviction, it may survive, and the bit will be reset, and the next item will be evaluated instead. The Clock cache replacement algorithm is an LRU approximation that eliminates some of the lock contention issues [13].

3.1.5 Low Inter-reference Recency Set

The LIRS improves on the fundamentals laid by the LRU by ranking its replacement candidates based on reuse distance, which captures more than reference history. In order to accomplish that, LIRS employs Inter-Reference Recency (IRR) as the history for each item. IRR of an item tracks how many other unique items are referenced between a reference of the item until the item is referenced again. LIRS dynamically keeps track of and adjusts the low IRR items (LIR) from the high IRR items (HIR). The capacity is often divided into 99% for LIR and 1% for HIR. LIRS assumes that if the IRR of an item is relatively high at some point, it is also most likely high at some other point. Based on this assumption, LIRS evicts items with high IRR (items in the HIR set), even though they may be recently referenced. items in LIR and HIR are often switched, based on the inter-reference recency [36].

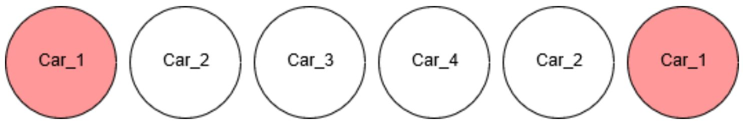

Figure 3.2: Access sequence from left to right for items in a cache. Car_1 is accessed twice, and in between, 3 unique items are accessed, making the Inter-Reference Recency for Car_1 3, adapted from [37].

3.1.6 Clock-Pro

The Clock-Pro page replacement algorithm is an enhancement of the Clock page replacement algorithm. Even though it is intended for page replacement, it is still used to solve caching. Clock-Pro attempts to utilize successful parts of the I/O buffer cache replacement algorithm LIRS, to enhance the performance of Clock. Clock-Pro uses inter-reference recency to label pages as cold(large IRR) or hot(low IRR). There are three hands or clocks, one for hot pages, one resident cold pages and one non-resident cold pages. Both hot and cold pages are kept in the same list ordered by their accesses, and cold pages are given a test period, to make sure that they are not causing cache pollution. If a cold page is re-referenced in the test period, it is labelled as a hot page, and if it is not re-referenced, it will leave the list. Cold pages in the test period may be replaced, but the metadata is kept throughout the test period in case they get re-referenced [38].

3.1.7 Clock-Pro+

Clock-Pro+ or CP+ extends the idea of Clock-Pro by dynamically adapting based on hits if either a non-resident cold page is accessed, or if the reference bit set on a resident cold page demoted from hot [39]. If the former occurs, the resident cold page size is grown by the maximum of 1 and $P_{\bar{n}}/P_{\bar{d}}$. Where $P_{\bar{n}} =$ 1/current number of non-resident pages and $P_{\bar{d}} =$ 1/current number of resident cold pages demoted from hot pages. If the latter occurs, the hot page size is grown by the maximum of 1 and $P_{\bar{d}}/P_{\bar{n}}$. Clock-Pro+ takes the Clock-Pro policy and applies the adaptive algorithms found in ARC/CAR.

3.1.8 S4LRU

S4LRU or quadruply-segmented LRU by Huang *et al.* is an eviction policy that maintains four queues, from levels 0 to 3 [40]. When a cache miss occurs, the item is inserted into the head of level 0. On cache hit, the item is moved to the head of the next queue, and items in level 3 are bumped to the head of level 3 when referenced again. Being evicted from a level means going from a higher level to a lower level. This means that items evicted from level 0 are evicted from the cache.

3.2 Sampled cache replacement policies

The idea of having sampled policies was introduced by Psounis and Prabhakar for web-caches [41]. Sampled policies do not require an expensive data structure maintaining items in the background, making it highly applicable for high-performance scenarios such as in the NoSQL key-value store Redis.

3.2.1 Hyperbolic

The Hyperbolic eviction policy, introduced by Blankstein *et al.* [42] evicts the item with the minimum result of the function how frequently they are used divided by their age.

3.3 Adaptive cache replacement policies

State-of-the-art cache replacement policies try to adapt to access patterns of users, usually combining the established and battle-tested policies to capture several useful features in the data and especially the data access patterns.

3.3.1 Adaptive Replacement Cache

The adaptive replacement cache, often referred to as ARC was introduced by Megiddo *et al.* [43]. The ARC consists of two LRU lists, where the first one, L1 keeps track of items recently referenced once, while L2 keeps track of items recently referenced at least twice. Thus, the primary responsibility of L1 is recentness of the items, while L2 has the frequency. The lists L1 and L2 consists of the LRU's T1 and T2 but also the ghost lists B1 and B2. The ghost lists tracks metadata about evicted items, which the ARC utilizes when it needs to adapt to change in access patterns and resource usage. New entries enter T1, eventually being pushed off to B1. If an item in T1 is being referenced one more time, it is added to T2, which again evicts in LRU style to B2. Entries that are re-referenced in T2 are re-linked to the head of T2. The ARC dynamically adapts to load by having L1 and L2 increase and decrease in size relative to each other within a fixed cache size as the workload changes. If a cache hit occurs in either of the ghost list, the related LRU cache will get its size increased by having the other LRU evict a candidate to its ghost list and forfeiting space for the other LRU. The ARC is visualized in figure 3.3.

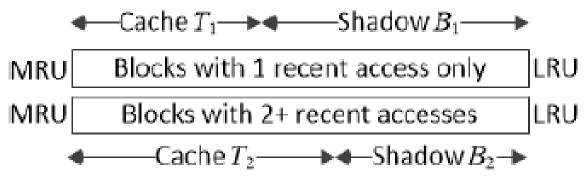
Cache T_1
Shadow B_1
MRU
Blocks with 1 recent access only
LRU
MRU
Blocks with 2+ recent accesses
LRU
Cache T_2
Shadow B_2

Figure 3.3: The adaptive replacement cache, from [44][p.61]. It shows the caches T_1 and T_2 with their ghost lists B_1 and B_2. Items flow from caches to ghost list, left to right, eventually being kicked out of the ghost list. The T_2 cache is in this case larger than T_1, which implies cache hits in B_2.

3.3.2 Clock with Adaptive Replacement

Clock with Adaptive Replacement or CAR is a cache replacement policy that follows the similar model as ARC. This means two caches with the same cache eviction policy where each cache has a ghost list, where recently evicted items or keys are kept for the adaptive steps. Differently from ARC is the usage of Clock instead of LRU for the two caches [45].

3.3.3 LeCaR

LeCaR is presented as a framework that is quite similar to ARC. The LeCaR cache framework is based on machine learning. More specifically, LeCaR uses reinforcement online learning with regret minimization. LeCaR, as implemented in the article, uses the two orthogonal policies, LRU and LFU for their two sub-caches as well as ghost entries for both caches. Authors of LeCaR experimented with using many different variants, for example, ARC (3.3.1) instead of an LRU cache as well as LFU with and without ageing, and surprisingly LRU came out ahead of ARC in this use case and pure LFU came out ahead of LFU with ageing. LeCaR dynamically adapts weights, which represent the probability distribution of items let into each cache based on regret. When a cache miss occurs, one of the policies are chosen (random based on regret values or weights due to misses they have caused) and they are eventually added to the ghost list of that policy. A decision is poor, or regrettable if it results in a cache miss and if the item is found in history because then we can regret evicting that entry to the ghost list, and update weights accordingly [46].

3.3.4 TinyLFU

Einziger, Friedman, and Manes approached cache replacement by creating an effective admission policy rather than an eviction policy. TinyLFU is a cache admission policy, which means it co-exists with eviction policies, which is often an eviction policy that is orthogonal to the admission policy. As the name suggests, TinyLFU is based on LFU. When TinyLFU is to admit a new item, it needs a candidate to evict before it decides whether the new item is expected to increase the hit ratio or not. In order to decide whether a new item will increase hit ratio or not, TinyLFU keeps track of frequencies of the items by approximations with the use of a Count-min sketch or some other probabilistic counting technique, rather than maintaining a min-heap or combining actual counting and sorting (described in 2.5.1). To keep the sketch up to date, TinyLFU uses a freshness mechanism, which is a periodic reset, or halving of the frequencies.

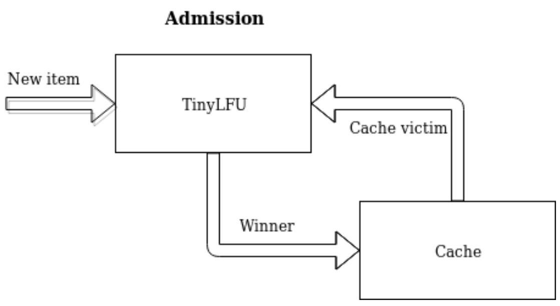

Figure 3.4: A cache with TinyLFU working as the admission policy, from [28]. The admission policy requests a cache victim, and decides whether replacing that item will enhance most likely increase the hit ratio, and admits based on that decision.

In the article, TinyLFU performed well on most of the benchmarks it was run against, but in some simulations, TinyLFU came out worse than other state-of-the-art cache replacement policies. The workloads that caused TinyLFU to fall behind was sparse bursts to the same items. More specifically, when bursts occurred, other items did not manage to build up enough hits before being evicted, which is a common issue. Redis approaches this issue by setting the logarithmic frequency counters of newly admitted items to 5. For TinyLfu to overcome this issue, W-TinyLFU was implemented, which is described in chapter 3.3.5 [28].

3.3.5 W-TinyLfu

W-TinyLFU, from Einziger *et al.* [47] was created as an optimization of the admission policy TinyLFU. W-TinyLFU adds a new cache area, called a window, hence the name W-TinyLFU. The window uses LRU as the cache replacement policy. As of the TinyLFU article writing, the default window to main cache ratio was 1/99. Instead of having the constant threshold, it is instead continuously optimized by using hill climbing optimization. Efforts to optimize the sketch has also been worked on, but results suggest that optimizing window/main size ratio is more effective. When an item arrives it is admitted into the window, and the victim of the window is then passed to the TinyLFU admission process, where it is given a chance to enter the main cache. Ghost entries are not maintained by TinyLFU or W-TinyLFU, which reduces the overall complexity of the implementations.

3.4 Cache replacement in the NoSQL landscape

In this chapter, some cache replacement policies in popular NoSQL databases and full-text search engines are presented.

3.4.1 Redis

Redis supports a variety of cache eviction policies, divided into evicting all keys, volatile keys (keys with expiration set) or no keys at all. Most of the eviction policies by Redis are approximations rather than an exact implementation because of the memory cost of maintaining enough data to perform precise evictions.

- No eviction - return an error when the cache is full
- All keys LRU - of all the keys, evict the least recently used
- All keys random - evict keys randomly
- All keys LFU - evict least frequently used keys
- Volatile LRU - of keys with expire set, evict least recently used keys
- Volatile random - of keys with expire set, evict randomly
- Volatile LFU - of keys with expiration set, evict least frequently used
- Volatile TTL - of keys with expiration set, try to evict the keys with the shortest TTL of candidates

Volatile eviction policies in Redis behave like the no eviction option if they do not have candidates to evict. Redis approximates by keeping a pool of eviction candidates, and based on the policy applied items will be evicted from the pool. Due to the high-performance design of Redis, the Lfu implementation stores both a logarithmic counter and last decremented time in 24-bits in each item, Redis considers the issue where newly added items do not survive for a long time to be worse than the issue of scan-like workloads. Having the initial counter of items set to 5 makes the item most likely to survive for a while, but if a sequential scan of some kind occurs, a large portion of the cache may get evicted, which is not the case in standard Lfu [34].

3.4.2 Memcached

Memcached uses a variant of the segmented LRU, described in chapter 3.1.3. Memcached implementation of the SLRU differs from the research article introducing it. Memcached SLRU implementation is able to both capture recency and frequency of the cached items by prioritizing to retain items in some of the LRU states/segmented LRU's (Warm in figure 3.5).

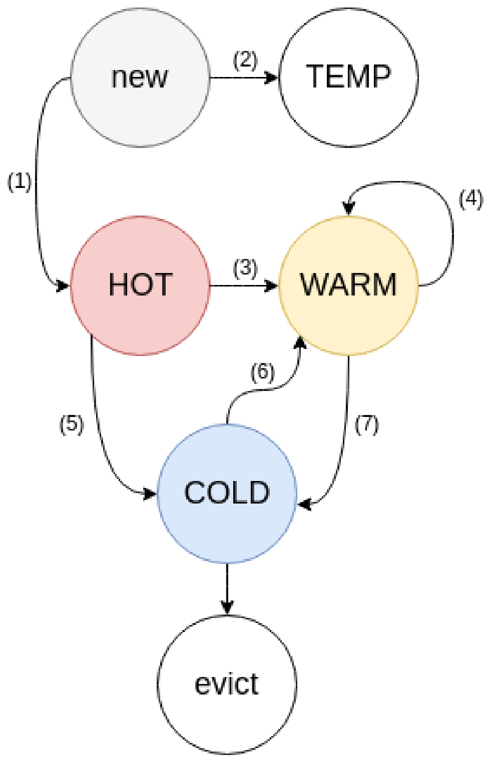

Figure 3.5: The Memcached variant of Segmented LRU state machine [48]. The LRU is split into four sub LRU's; Hot, Warm, Cold and Temp which are all carefully designed to serve clients with the highest performance in mind.

The Segmented LRU splits the LRU into four sub-LRU's, all of which have a maintainer background thread. Items are labelled with a two-bit activity flag, either set to fetched (if it has been requested) or active (if it is been accessed a second time), the active flag is reset when bumped or moved, bumped refers to an item being re-linked into the head. Items within the HOT LRU are expected to have strong temporal locality (described in chapter 2.4.3) or short TTL. In Hot, items are not bumped because of the overhead, thus when an item reaches the tail of the LRU, it is moved to Warm if the item is active (3) or Cold if it is not(5). Items in Warm are prioritized to be kept when eviction occurs. In Warm, if a tail item is active, it is bumped(4), otherwise, the item is moved to

Cold(7). Cold holds the eviction candidates, if an item becomes active again it is moved asynchronously to Warm(6), and if the Cold gets full, it starts to evict items at its tail. The Temp state is meant for items with very short TTL(2), thus items are not moved to other states from Temp, which is done to save CPU and lock contention [48].

3.4.3 Elasticsearch

Elasticsearch has implemented a concurrent read optimized cache. The cache is divided into 256 segments, backed by hash maps. Each segment holds a reentrant read/write lock. Each segment also maintains a doubly linked list for evictions. This cache implementation differs from traditional LRU in that sense that it supports both time-based and weight-based eviction. The weight-based eviction is simply a threshold that gets compared to the summarized weight of the entry, which defaults to 1, meaning that each segment by default can hold up to a thousand entries [9].

3.4.4 Cache2k

Cache2k has implementations for both Clock and ARC in their codebase and documentation. However, they are utilizing the Clock-Pro+ or CP+ cache replacement policy [49].

3.4.5 Caffeine

Caffeine is a high performance caching library for Java 8. it is maintained and developed by one of the authors of TinyLFU, Caffeine is used by NoSQL graph database neo4j, NoSQL wide column store Cassandra, NoSQL wide column store Apache HBase and several other popular projects. Different types of NoSQL databases have different needs in terms of caching, some aim to cache expensive queries, while others focus more on more traditional approaches, with caching storage blocks in order to minimize the number of trips to relatively slow storage. Internally, Caffeine uses W-TinyLFU with hill climbing optimization for dynamically optimizing the size of the window and main cache. Initially, this defaulted to having the window 1% and the main cache 99% of the cache size. For the main cache, Caffeine uses segmented Lru, which is described in chapter 3.1.3. Cache replacement that maintains ghost entries are known to have a higher memory overhead as well, where the W-TinyLFU implementation in the Caffeine library has an overall space overhead of 8 bytes per item. [23]

Chapter 4

Design and architecture

In this chapter, the implementation of the cache replacement policies is explained as well as some background theory on weighted and cost-based caching. For the historical access patterns, the approach to enhance an adaptive cache replacement policy is also explained.

4.1 Introduction

In order to answer the research questions, thorough problem analysis is required. To successfully capture historical access patterns, high-quality data is needed. Introducing and capturing more features when making evictions has a large potential for NoSQL databases and full-text search engines because some items are expensive to retrieve, while others are cheap. The cost could both be utilized when making admissions and evictions, all dependent on the goal of the cost-based addition. The implementation is data dependant, which is often mentioned in the chapters explaining the algorithms. In addition to being data dependant, policies may have a different design based on whether the caching context is weighted or not.

4.1.1 Caffeine simulation project

Caffeine comes with a well-designed simulation project for comparing cache replacement policies. There is only a limited amount of policies that support weighted caching and are eligible for the experiments that hope to answer RQ1. In Caffeine, the cache eviction policies are divided into Adaptive, Linked, Sampled, Irr based and sketch-based policies, while all the admission policies that originate from Caffeine are Count-min sketch based.

4.2 Weighted and cost-based cache replacement

Items cached by NoSQL databases and full-text search engines have different retrieval costs affiliated with them. In some cases, the cost is equal to the size, while in other cases, the cost represents some other item attribute or metadata associated with the item, such as retrieval cost.

NoSQL databases and full-text search engines support storing items, documents and query results of different size and type. Making a replacement solely on the access metadata of that item means that one item, which is expensive to retrieve, could be replaced before an item which is cheap to retrieve. In content delivery networks, engineers have approached this issue, which can be seen in AdaptSize [50]. AdaptSize is intended for another level of abstraction than the needs of NoSQL databases and full-text search engines.

Extending the feature space with a cost related to each item is not that difficult, but figuring out how the cost should weigh and contribute to item relevancy compared to other access metadata is a difficult challenge. Policies that have a different internal ordering than access time and sampled policies are suitable for a feature extension.

4.2.1 Weighted and cost based

Weighted cache replacement is when an item comes with a weight. When caches work with weighted items, the maximum number of items depend on the sum of weights of items in the cache. In cost-based caching, each item is assumed to occupy one slot in the cache, while their cost represents their retrieval cost. For weighted caching, the weight is often set to the item byte size. For cost-based caching, the cost of an item is set to the byte size as well because most of the available traces are I/O traces.

The problem with weighted caching

The issue in weighted caching, illustrated in figure 4.1 is that when the admission policy presents a candidate, it needs to account for the total cost of admitting a new item in terms of what does the cache lose if admitted. In the case of figure 4.1, it has a maximum size of 3072, so the candidate for admission is marked in yellow with key Key5, value Value5, estimated frequency 10 and weight 3000. Admitting this item into the cache would force not just the victim marked in red, but all the other items to be evicted. State-of-the-art policies do not evaluate this today.

The problem with cost-based caching

In cost-based caching, each item has a retrieval cost or computation cost associated with itself. Cost-based caching is a model for how query cost could be an indicator of item relevancy along with other item attributes and access metadata such as frequency and recency. In the cost-based caching, the goal is to keep relevant items of high cost in the cache, while not admitting or evicting items considered not relevant and with low cost. The problem or the challenge is to balance the item attributes and access metadata because it is difficult to determine whether an item with low access frequency and high cost are more relevant than an item with high access frequency and low cost.

4.2.2 Admission

One way to include a cost or weight in a cache replacement policy is to have the admission policy evaluate it. The sketch-based admission policies, such as

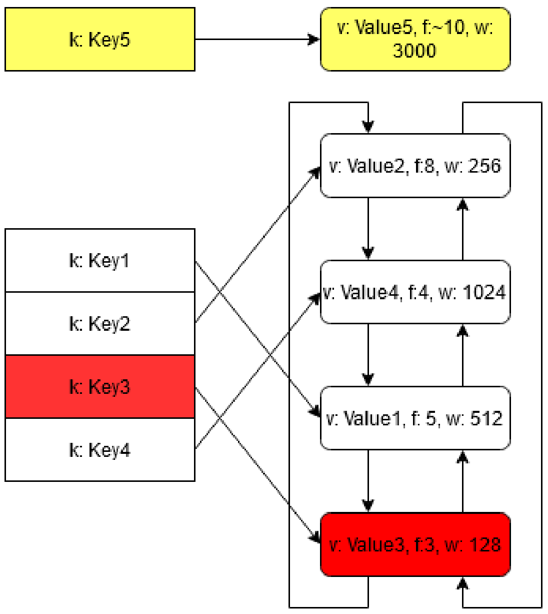

Figure 4.1: The problem of weighted caching illustrated with this imaginary cache that has a maximum size of 3000. The yellow key and value is the candidate presented by the admission policy, while the red is the victim from the eviction policy.

TinyLFU, evaluate whether the candidate is a better option to admit into the cache compared to a victim. To extend the admission policy to evaluate cost as well could be as simple as determining how to evaluate size vs frequency. The AdaptSize article explains a few different cost- or size-based admission strategies. [50]

- Threshold-based admission policy, meaning that items are admitted based on whether they are above/below a certain threshold in terms of an item attribute such as frequency, weight or cost.
- Frequency-based admission policy, comparing (estimated) frequency and weight of candidate versus victim and choosing the best one to be admitted.
- Probability-based admission policy, where each item has a certain probability of being admitted, often based on item attributes.

The arguments made in the AdaptSize paper about probabilistic admission could also be translated into the use case for NoSQL databases and full-text search engines, but it needs to be adjusted. The probability function used to determine whether to admit an item or not is exponentially decreasing with size, making large items less likely to be inserted. Preventing an item of entering the cache in NoSQL databases without considering the total item lifetime cost could lead to poor performance.

4.2.3 Eviction

In order to evict based on item cost or weight, it is necessary to order items based on them, but only evicting based on cost will lead to poor performance. Merging the cost or weight with other important item attributes is a viable strategy. Some policies already operate on a form of cost-based internal ordering, such as Lfu, which is often implemented with a min-heap. The min-heap could easily evaluate cost or weight in addition to other item attributes. In policies which do not have an already existing cost, keeping a pool of eviction candidates, such as the sampled policies could be done.

- LFU and LFU-like policies could make a combined internal score for frequency and weight/cost.
- LFU and LFU-like policies could make ageing of the items be based on their weight/cost.
- LRU and LRU-like policies, since LRU does not keep any ordering than the reference time, no other option than selecting a small sample, and choosing the item with the lowest cost to evict. Successful examples of random sampled eviction can be seen in Redis [6], where due to the large cost associated with exact implementations was considered too high compared to always evicting the correct item.
- LIRS with IRR and LIRS-like policies have the potential to add more types of items, for instance, items with large weight and low IRR. do a similar enhancement of the internal ordering measurement to include size. Similar to what a sampled policy could do in this case, LIRS could evaluate size when evicting from the HIR set.

4.3 Replacement policy design

Due to state-of-the-art policies often have divided the cache into admission and eviction, the implementation will follow this scheme. To distinguish between approaches which work well when cost equals the size of the item and where it is some other metadata such as item retrieval cost, most of the implementations of the admission process 4.3.1 is done considering both use cases.

4.3.1 Admission

The idea behind enhancing the admission algorithm for the replacement policies is to maintain a high hit rate as possible while increasing the weighted hit rate. As mentioned, the implementation will differ based on the goals of introducing cost. If the goal is to keep items of high cost in the cache, then comparing the cost could be viable. To enhance the admittor role, several approaches are investigated, from what is expected to perform poorly, to smarter approaches.

- Admit based on comparing weights/costs
- Threshold-based admission

- Evaluate weight/cost as a secondary measure when frequencies are equal in TinyLFU
- Multiplying frequency and weight/cost to form a combined score
- TinyLFU with increments based on how big the weight or cost of the item is.

Comparing weights/costs

Comparing weights/costs has no obvious way of evaluating the total cost of an item within the cache, meaning admitting items of larger or smaller weight/cost is no guarantee of them having a high reference count and short re-reference interval. Nonetheless, the approach is included to showcase the fact that evaluating only weight or cost is no viable approach.

Threshold based admission

Threshold-based admission admits items under or over a certain weight threshold, based on whether it is weighted or cost-based caching. It will not work unless the threshold is dynamic, as it could easily get stuck evaluating a low-/high-value victim. To solve this, a change parameter to the threshold is introduced. Its purpose is to scale up/down the threshold to avoid lifetime tickets of high weighted/cost items. As with comparing weights/costs, this approach will showcase the issue of threshold-based admission. For NoSQL databases, both items of high and low weight/cost need to be admitted, and prioritizing items of either high or low weight/cost does not translate into overall cache performance.

$$T_d(c) = \frac{T_{d-1} + \frac{(\sum W)*c}{cache.size}}{2} \tag{4.1}$$

To ensure that no item could block the admission process forever, we scale up or down the accumulated weights by having change variable c in the equation increased or decreased by x for every iteration where the victim prevails.

Weight/Cost as a secondary attribute for admission

Using sketches to evaluate frequencies of the candidate and victim in the admittor role is considered state-of-the-art. If a candidate and a victim have the same frequency, the victim is kept. The extension to this decision logic is simple, if their estimated frequency is equal, keep the item with the highest or lowest cost, which depends on the use case and desirable cache data. For weighted caching, we keep the item of the lowest weight/cost, while for cost-based caching, we keep the item with the highest cost. The TinyLfu defaults to using a 4-bit sketch, meaning that each cell in the sketch has a maximum frequency of 15, making this policy very likely to have an impact.

Combining frequency with cost

Combining frequency and cost or weight by multiplying them comes down to finding a suitable scale for the data, and determining which item metadata contributes the most to a high cache hit ratio. For weighted caching, multiplying yields undesirable results due to items of lower weight are contributing to better performance. Hence for weighted caching, the frequency is divided by the weight. Scaling weight or cost and frequency of the same magnitude is very hard without domain knowledge of the data that is being cached. Nonetheless, this approach to caching is expected to perform quite similar to comparing weights or costs due to the sketch reset intervals being quite short, making weight or cost solely what determines admission.

Sketch increment

TinyLfuBoost is based on incrementing items by a given number that represents the weight or cost of the accessed item. This again requires a delicate way of scaling down the weights/costs. Doing this wrong will lead to many cells in the 4-bit Count-Min sketch having the max value 15, even though it is possible to utilize a low reset interval to keep the frequencies low, but this is not optimal.

4.3.2 Eviction

All cache replacement policies have some sort of eviction policy that dictates which item to evict when necessary. Policies applicable for cost-based eviction are those that rely on some internal sorted data structures, such as LFU, which sort based on item access frequency. Even though some policies are more applicable than others, the approaches are simple and would fit most cache eviction policies.

Lfu cost boost

The same approach as sketch incrementing based on weight/cost applied to Lfu with the frequencies. The implementation requires a similar scale down strategy to limit the O(1) LFU traversal. The algorithm is extended by instead of bumping a node by incrementing it by one (adding it to the next frequency node, or creating one), we traverse the frequency nodes until we find our target node or we create a new one at the right link [33, p.4]. The function that determines how much we should increment is data dependant, and in this project, with I/O traces, the max of 1 and the natural logarithm of the cost is used.

Lfu ageing based on cost

Some cache eviction policies, such as LFU, sometimes employs ageing to fight items with sparse bursts access patterns, which make them unnecessary occupy space in the cache without being accessed. Ageing simply means that in some specified interval for each item, or for specific items, decrement the access frequency of that item. The idea is that the number we decrement the frequency with should represent what properties we prioritize to keep in the cache. For cost-based caching, we would age items of low cost more every ageing step than items of large cost.

One Every given interval the frequency of all the nodes gets decremented by one.

Skip Every given interval the frequency is decremented by one if the cost of the item is considered small. The threshold is set through cache settings.

Boost Every given interval the frequency is decremented by a function that attempts to scale down the cost, or weight of an item. The current approach is taking the max of 1 and the boost part seen in equation 4.2, which divides the natural logarithm of the weight with the Euler's constant to the power of a commonly used block size(in the Caffeine experimental project).

$$Boost(w) = \frac{ln(w)}{e^{-(\frac{w}{512})}} \tag{4.2}$$

4.4 Historical access patterns

Since the mainstream adoption of the cloud, there are currently several tools that are doing predictive scaling and self-tuning systems, such as Facebook spiral [51] and Amazon predictive scaling [52]. However, neither the source code or the data foundation of the tools are available to the public, which makes it hard to review their applicability when it comes to caching. Thus, a proof of concept experiment is conducted, which aims to help to gain insight into RQ2.

Making a system capturing historical access patterns is difficult because it requires a way to detect and isolate trends in a high paced environment. The idea behind capturing historical access patterns and utilizing it to reach a desirable adaptive state lies in it being an addition to the original algorithms deciding and adapting the internal cache state. So if the historical access pattern detects a trend at time x, while the cache itself disagrees with that decision, one would design the algorithm to override the adaptive steps based on recent past to prevent system performance regression.

When the system decides to change the internal state, it needs to be sure that it makes the right decision. Making the wrong one could cause a denial of service towards itself, leading to drastically reduced system performance. In the cases where both the adaptive algorithms agree to adapt in one direction the process of adapting to a desirable state could be done quicker.

4.4.1 Enhancing LeCaR

Adaptive replacement policies often adapt based on hits in ghost lists or some other strategy to adapt to changing data access patterns. Both LeCaR and ARC use the adaptive strategies to determine the distribution or size of the sub-caches, while Caffeine with W-TinyLfu uses it to determine the size of the window relative to the main cache. LeCaR is a suitable policy to answer and gain insight in RQ2 due to its design with a probability of having items added to either sub-cache. In the LeCaR article, Lru and Lfu were used for their sub-caches. Having the most scan resistant policy "active" when a difficult workload occurs could be beneficial to reduce cache pollution. Three variants of LeCaR is to be experimented with, as seen below.

- LeCaR with no change in the internal state.
- LeCaR with setting learning rate very low.
- LeCaR with setting learning rate very low and manually changing the weights to be more scan resistant.

4.5 Development

The algorithms were all developed using the experimental project in the Caffeine caching library, which is open-source and available on Github [23], the cache implementations in this thesis can be found at the forked Caffeine GitHub repository [53].

4.5.1 Tools and frameworks

In this chapter, the primary tools and frameworks used in development, experimentation and post-processing are presented.

Java

Java was chosen as the language for implementation due to Caffeine being a Java library. Java is also a common language and platform to use when developing NoSQL databases due to its feature-rich ecosystem. The Apache Software Foundation relies heavily on the Java Virtual Machine for their projects, which again has made Java common in NoSQL and search engine implementations. For full-text search engines, the Apache Lucene project is used as the underlying library that handles search and indexing for both ElasticSearch [9] and Apache Solr [10]. For NoSQL databases, HBase, Cassandra and Accumulo are well-known projects, which all are Java projects that utilize the Caffeine caching library [54] [55].

Caffeine simulator

Caffeine simulator is used throughout the project to ensure fair comparisons of cache replacement policies.

Akka

In order to give each simulation a fair chance to perform, they are all run in parallel as actors using the Akka framework [56]. This ensures that the policies are run in isolation. A router routes requests in batches to every running actor with its policy.

fastutil

fastutil: Fast & compact type-specific collections for Java [57].

Python

Python was used in the initial experimentation on using historical access patterns to provide better adaptive decisions. Another usage of python and python libraries is listed below.

- Numpy, some numerical analysis of historical access patterns and data processing
- Pandas, data processing and visualization
- Matplotlib and seaborn for visualizations

4.5.2 Code structure

In the Caffeine simulation project, code is separated into simulation code(settings, generators), eviction policies, admission policies, trace parsing, membership and reporting. The simulation code contains wrappers for the settings files, the main simulation code (with actors) and a synthetic trace generator. Eviction policies, or the package *policy* contains eviction policies, interfaces and tools for the design of the policies. Policies either implement the interface *Policy* or *KeyOnlyPolicy*. Admission policies, or the package *admission* contains admission policies, interfaces and tools. Admission policies implement the interface *Admittor*. Each simulation contains at least one, but possibly several traces. Traces have parsers related to them, which determines whether the simulation can be weighted or not. If a simulation is weighted, only the policies that support weighted cache replacement are executed, of every policy listed in the configuration file. The membership code contains algorithms and data structures for checking whether an element is a part of a set or not, typically bloom filters. The reporting code, in the package *report* contains code to report cache performance metrics and output it according to what the configuration specifies.

The configuration file controls every setting in the cache, for example, maximum size, batch size of trace message passing, traces, which eviction policies to use, which admission policies to use and so forth.

4.5.3 Limitations

The limitations of this work relate to the limited amount of public production cache traces. This problem applies to both adapting based on historical access patterns and weighted and cost-based cache replacement. Most of the traces are I/O related, which makes them decent at testing the implementations, but for weighted and cost-based caching the size which is the weight in weighted caching and cost in cost based caching, of the items is usually high, which makes it hard to balance with other item attributes or metadata.

4.5.4 Testing

Since this is an experimental phase, some manual testing to validate cache behaviour is done. This includes investigating and validating metrics and checking counters. For adaptive policies such as ARC, some runtime assertions are done to validate that the internal distribution of different cache types is correct.

Chapter 5

Experiments

In this chapter, expected results and goals of the experiments, the experimental plan, setup and results are presented.

5.1 Introduction

The following experiments are designed to show the potential behind the approaches that were implemented in chapter 4, to answer and gain insight in RQ1. Furthermore, a proof of concept experiment is also performed to gain insight into RQ2, which shows the potential that lies within using historical access patterns in adaptive cache replacement policies. The experiments are evaluated by typical cache performance evaluation metrics, presented in table 5.2. The proof of concept experiment for historical access patterns also presents a figure showing hit rates over time.

5.2 Goals and expected results

The goals and expected results are related to research questions, experiment design, traces and algorithms.

Goals

The goals of the experiments are to show how the simple extensions of the cache replacement policies are performing when simulated on different traces. Furthermore, the insight and performance metrics will contribute to answering the research questions and have a thorough discussion.

- For historical access patterns, the goal is to show the potential behind actively making a caching system that utilizes an adaptive cache replacement policy more scan resistant when expecting a scan-like workload. This means showing that preparing a system for known workloads could keep performance as high as possible.
- Provide insight towards enhancing a state-of-the-art cache replacement policy which evaluates the weight or cost of items.

- Perform a performance evaluation of different policies on different traces and compare their results.
- Provide insight towards the correlation between hit rate and weighted hit rate for both weighted cache replacement and cost-based cache replacement.

Expected results

- For historical access patterns, LeCaR with learning rate and weights altered is expected to perform better when the scan like access pattern occurs. Also LeCaR with learning rate adaption will perform very similarly to no adaption by human experts due, based on how many hits the ghost lists receive during the scan like access pattern.
- For performance simulation, the expected results will depend on whether the simulation is run with cost equals the size of the objects, often referred to as weighted caching or with cost representing other metadata.
- Expect specialized eviction policies such as MRU and MFU to be worse than other policies when simulated on a web/SQL trace.
- Expect admission policy Comparison to have varying performance due to it only evaluates weight or cost which does not translate to total item cost.
- Expect admission policies TinyLfuMulti and Threshold to have varying performance since they require specific knowledge of the data in order to be performant.
- The Secondary admission policy will have varying performance based on whether the simulation is weighted or cost-based.
- The performance of TinyLfuBoost will usually be equal or slightly better than TinyLfu for the same combination of policies.

5.3 Test cases

The experimental plan for this project is to experiment with the cache replacement policies in an isolated environment. The Caffeine caching library comes with a simulation project that is a suitable foundation for experimentation. The project has implemented the majority of the state-of-the-art policies as well as a comparison test suite where we can supply specific traces.

5.3.1 Historical access patterns

As mentioned in chapter 4.4 capturing trends in a dynamic caching environment is no easy task, which is why the experiment shows the potential of having a human expert intervene when a scan-like workload is expected.

Human expert

In this experiment, a human expert knows that eventually, the caching system will find itself in a point in time where the hit rate is expected to drop. The drop in hit rate could be explained with a scan like workload triggered by nightly reports. When the hit rate is expected to drop, the system prepares itself to withstand the workload change to reduce cache pollution. When the hit rate is expected to regress, the human expert triggers the changes to the policies in the experiment.

In order to show that preparing the system to handle difficult workload changes, a combination of datasets was used in the experiment, and this particular combination was *multi1*, *cs*, *multi2* and *cs* traces from Lirs [44]. The LeCaR replacement policy with the suggested LRU and LFU combination was run with an initial learning rate of 0.3, forced low learning rate 0.0001, discount rate as in the LeCaR paper and cache size 500.

5.3.2 Cost based cache replacement

Extending the feature space with a cost related to each item is not that difficult, but figuring out how important the cost is compared to other access metadata is a difficult challenge, as it will likely vary from use case to use case. If the engineer controlling the cache has specific domain knowledge such as how large the items usually are, they can optimize adaptive policies to obtain the desired balance between cost and other access metadata. Both the cases where the cost of the item is equal or proportional to its size and where the cost represents some other access metadata is covered and they are often referred to as weighted caching and cost-based caching.

As mentioned in chapter 4.2.3 cache replacement policies that have an additional internal ordering than an access point in time are candidates for introducing a cost function in addition to frequency or similar access metadata. In most cases, and the Caffeine simulation project, weight is the size in bytes of the item. In addition to cache hit rate, the weighted hit rate is an interesting metric that is the rate of weights of hits to the weight of the requests. In the cost-based experiments, algorithms are run on traces with a cost similar to the previous ones, but we imagine the cost represents something other than size or weight, for instance, database query cost.

Ruling out policies

The purpose of this experiment is to show how specialized eviction policies and some simple admission policies performs versus more adaptive policies run on a web/SQL trace (web_0 from MSR Cambridge). All policies that support weighted/cost-based eviction are run combined with every admission policy. The purpose is to trim down the number of both eviction and admission policies for further experiments.

Performance simulation

The purpose of these experiments is to provide insights into how the combination of policies perform on different kinds of traces, but also how they perform against each other. Metrics of interest are both hit rate and weighted hit rate. For cost-based caching, the metric weighted hit rate means the same as with weighted caching, but the maximum size in cost based caching is not restricted by the accumulated weights of the items there.

5.4 Setup

The setup required to fully understand the results. The focus of the experimentation is not algorithm execution time, but the simulation environment is still listed.

5.4.1 Cache parameters

- Maximum size is usually set to around 10% of request count when running simulations where size equals weight and around 5% of request count when running simulations where size is 1. This does vary because, in some traces, weights/size of items may be very high, while in other traces there are not that many unique keys. A successful outcome of a simulation is when the metrics are both big and small enough to see the difference between the admission and eviction policies.

- LeCaR learning rate is 0.3 and forced low learning rate of 0.0001.

- Historical access patterns human expert intervened at discrete time 5000 and reset it at discrete time 16000.

- TinyLFU uses a Count-min sketch 4-bit implementation with periodic resets. The periodic resets by default halves all counters. The maximum value of each cell in the sketch is 15.

- The 4-bit Count-Min sketch has a periodic halving interval 16, meaning every at every 16 items evaluated, the sketch is reset, or aged.

- Sampled policies are configured to have a random sample size of 8, which is in the range of what Redis discusses in its documentation. Redis defaults to 5 but explains that keeping a pool of 10 results in a more precise approximation of the actual policy [58]. The policies uses a random guessing algorithm to obtain samples for eviction.

- Threshold-based admission 4.3.1 is an adaptive policy that it uses a strategy to scale down the threshold if an item gets stuck in the victim role of the admission policy. It updates threshold according to equation 4.1, where the c parameter is initially 1.0, and reset to 1.0 if an item is admitted. For weighted caching, we increment by 0.01, while for cost-based caching we decrement by 0.01. This ensures the algorithm not getting stuck for long while maintaining the performance benefits of using an admission policy.

- Cost as a secondary attribute for admission 4.3.1 depends on the data and context. For weighted caching, an item with smaller weight is admitted, while for cost-based caching, higher cost results in being admitted. This observation and implementation choice corresponds with the probabilistic admission policy in AdaptSize [50].

- Combining frequency and cost 4.3.1 is implemented by division or multiplication of the frequency and weight or cost into a combined total cost to evaluate.
- Sketch increment 4.3.1 works by incrementing the sketch with a maximum of 1 and the natural logarithm of the cost divided by 512.
- Lfu cost boost, or frequency boosting 4.3.2 is a similar approach as Sketch incrementing but it is altering the main data structure holding the items and not a sketch for admission. Instead of incrementing the frequency by one, we increment by the natural logarithm of the block size.
- Ageing based on cost 4.3.2 has primarily two approaches, and two baseline approaches since this is a min-heap implementation of the LFU, and not the O(1) variant. [33] Skip ageing skips ageing for items with small size/cost, for these experiments, the threshold was set to 10000. For the age boost approach, the frequency of the item is decremented by dividing the natural logarithm of the weight with the Euler's constant to the power of the number of blocks (equation 4.2).

5.4.2 Hardware

- Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz 2.71
- 16GiB System Memory

5.4.3 Software

- Java version 11.0.5
- Java virtual machine heap size: 4GiB
- Caffeine 2.8.1 [23]

5.4.4 Datasets

Datasets used in the experiments can be seen in table 5.1. For reproduction purposes, traces from the experimental phases of the articles are included if available and if relevant for the weighted/cost-based caching. The Lirs paper labelled traces into four categories, looping, probabilistic, temporally-clustered and mixed traces. Cs trace from Lirs is categorized as a looping type, while multi1 and multi2 are categorized as a mixed type, meaning that they are a combination of looping, probabilistic and temporally-clustered type [36]. MSR Cambridge traces are from enterprise servers at Microsoft, in the caffeine project they are one of the few traces and parsing algorithms that support weights. The traces of interest are web/SQL and proxy traces [59]. The traces from the University of Massachusetts Amherst, often referred to as UMass trace repository are common to find in articles that run cache simulations. Financial OLTP I/O and Websearch I/O are used from UMass [60].

Table 5.1: Datasets

Source	Trace Name	Description
UMass	Financial	I/O traces from OLTP applications running at two large financial institutions.
UMass	WebSearch	I/O traces from a popular search engine.
MSR Cambridge	web	I/O traces from Web/SQL server
MSR Cambridge	prxy	I/O traces from two volumes of firewall/web proxy
Lirs	cs	An interactive C source program examination tool trace
Lirs	cpp	GNU C compiler pre-processor trace
Lirs	postgres	Join queries among four relations in a relational database system from the University of California at Berkeley.
Lirs	multi1	A combination of executing cs and cpp trace together
Lirs	multi2	A combination of executing cs, cpp and postgres together

5.4.5 Metrics

All of the metrics emitted from the Caffeine simulation project can be seen in table 5.2. The metrics of particular interest are cache hit rate and weighted hit rate. These metrics represent important characteristics of cache replacement policies and will be a good indicator of how it would perform in a production scenario in either a NoSQL database or full-text search engine.

Table 5.2: A table of cache metrics emitted from the Caffeine simulation project. Metrics such as Average Penalty, Average Miss Penalty and Steps are only implemented by a few policies.

Metric name	Description
Hit rate	Rate of the number of hits to the number of requests
Hits	Number of hits
Misses	Number of misses
Requests	Number of requests
Evictions	Number of evictions
Admit rate	Rate of admitted items when using admission to the number of requests
Requests Weight	The accumulated weight of the requests
Weighted Hit Rate	The rate of weights/costs of hits to the requests weights/costs
Steps	Number of steps performed by the different policies
Time	Simulation time

5.5 Results

The results of the experiments. For the historical access patterns, a figure and a table are presented to provide context for the discussion. For the cost-based replacement experiments, the ruling out policies experiment is presented with graphs and top-10 tables, while the rest of the performance simulation results is only presented with top-10 tables due to a large number of policies and results.

5.5.1 Historical access patterns, human expert

The results are shown both in table 5.3 and figure 5.1. The figure shows continously hit rate throughout the experiment, which makes it easier to see the actual impact of the human expert role in the experiment.

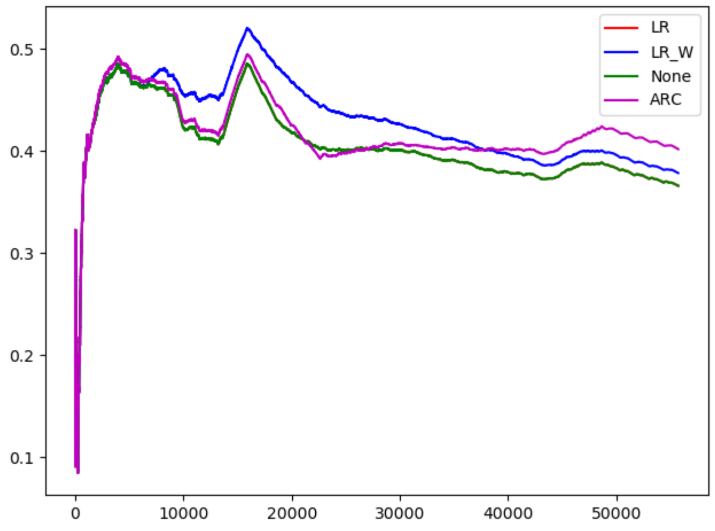

Figure 5.1: Hit rates of an expert knowing how the system changes over time. LeCaR and ARC are run on a combination of traces to form a loopy like workload. LR (learning rate), LR_W (learning rate and weights) and None (nothing) is the LeCaR cache replacement policy with different historical adaptation settings, and ARC is displayed for reference. When the discrete time hits 5000, the expert prepares the caching system, and when the time hits 16000, the changes are reset. The red line, which is barely visible only changes the learning rate, the blue changes learning rate and weights, and the green one changes nothing.

5.5.2 Weighted and cost based cache replacement

Here the results of cost based cache replacement will be presented. Initially in the Ruling out policies experiments, the primary evaluation metric is the cache hit rate, while in the performance simulation, the weighted cache hit ratio will also be presented.

Table 5.3: End results of simulation run with a combination of Lirs traces to create a workload that gets loopy, and how it is possible to prevent these performance drops with preparing the system. While not apparent, directly after the scan like workload occurred, lecar.LR_W outperformed the other ones, even though ARC came out with the higher hit rate in the end.

Policy	Hit rate	Hits	Misses	Evictions	Time
adaptive.Arc	40,16%	22383	33348	32848	424
lecar.LR(0,300000)	36,58%	20388	35343	34843	456
lecar.LR_W(0,300000)	37,83%	21083	34648	34148	560
lecar.None(0,300000)	36,58%	20388	35343	34843	613

Ruling out policies

The results of ruling out both admission and eviction policies based on simulations. For ruling out policies, the web_0 trace from MSR Cambridge is used.

Weight based replacement

The results of ruling out policies with size equals weight/cost.

Table 5.4: Average percentage hit rate and weighted hit rate change compared to using no admission in weighted caching simulated on the web_0 trace from MSR Cambridge. [59]

Admittor	**Percentage hit rate change**	**Percentage weighted hit rate change**
Comparison	-26.12%	-35.57%
Secondary	-8.56%	-30.73%
Threshold	-24.04%	-40.34%
TinyLfu	+36.20%	+15.77%
TinyLfuBoost	+27.44%	+6.73%
TinyLfuMulti	-12.60%	-41.80%

Table 5.5: Average percentage hit rate and weighted hit rate change compared to using no admission in cost based caching towards the web_0 trace by MSR Cambridge. [59]

Admittor	**Percentage hit rate change**	**Percentage weighted hit rate change**
Comparison	-39.92%	-19.96%
Secondary	+29.19%	+17.96%
Threshold	-38.69%	-21.86%
TinyLfu	+32.43%	+16.43%
TinyLfuBoost	+29.54%	+19.80%
TinyLfuMulti	-10.61%	+9.97%

Performance simulation

The performance of running the simulations on different traces with some policies filtered out. In order to present the performance of the cache replacement policies we need to filter out and present algorithms that both have high hit rate and weighted hit rate.

Ruling out policies with weighted caching simulation on the web_0 trace from MSR Cambridge

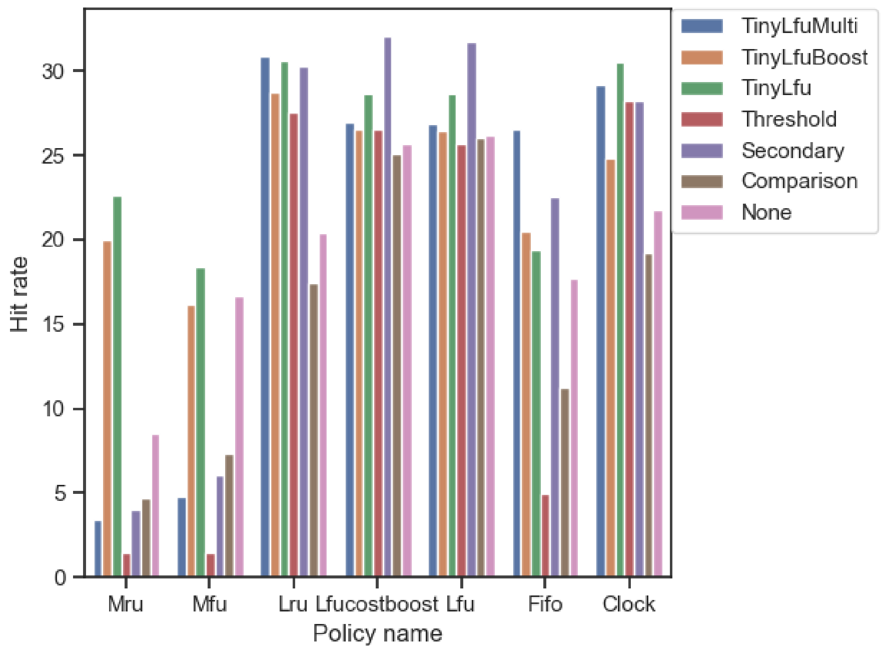

Figure 5.2: The linked policies of the Caffeine simulation project with different admission algorithms which is described in 4.3.1. Simulated with the web_0 trace from MSR Cambridge [59] with a maximum weight of 100000.

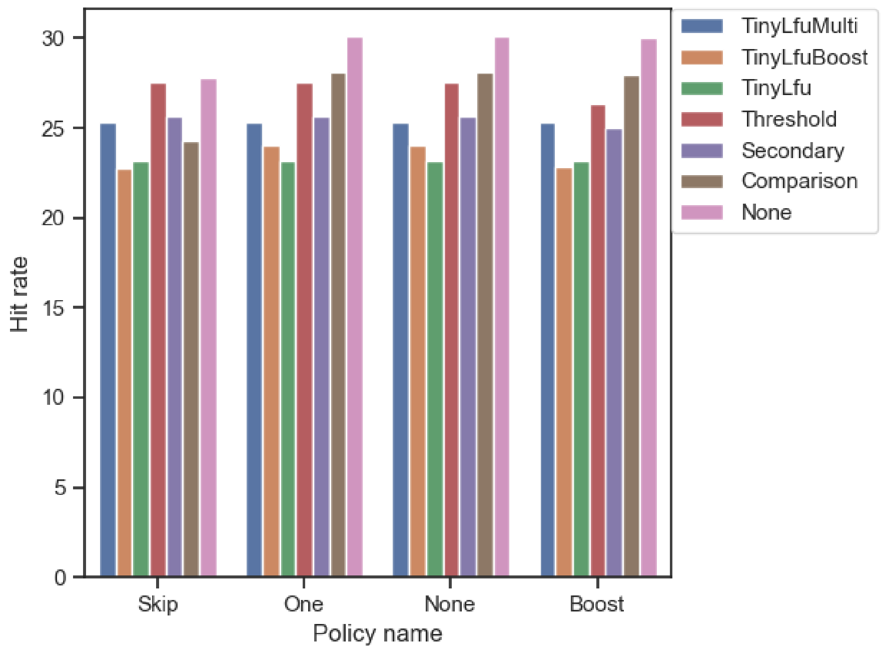

Figure 5.3: Min-heap LFU eviction policies described in chapter 4.3.2, with admission policies described in chapter 4.3.1. The trace is web_0 from MSR Cambridge [59] with maximum weight 100000.

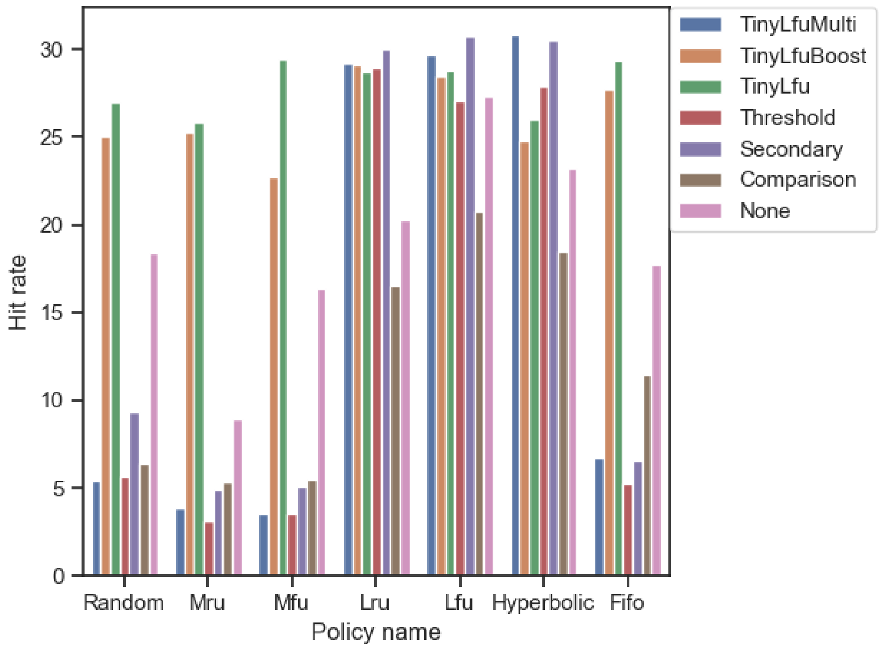

Figure 5.4: The sampled policies of the Caffeine simulation project with different admission policies, described in chapter 4.3.1. The trace is web_0 from MSR Cambridge [59] with maximum weight 100000.

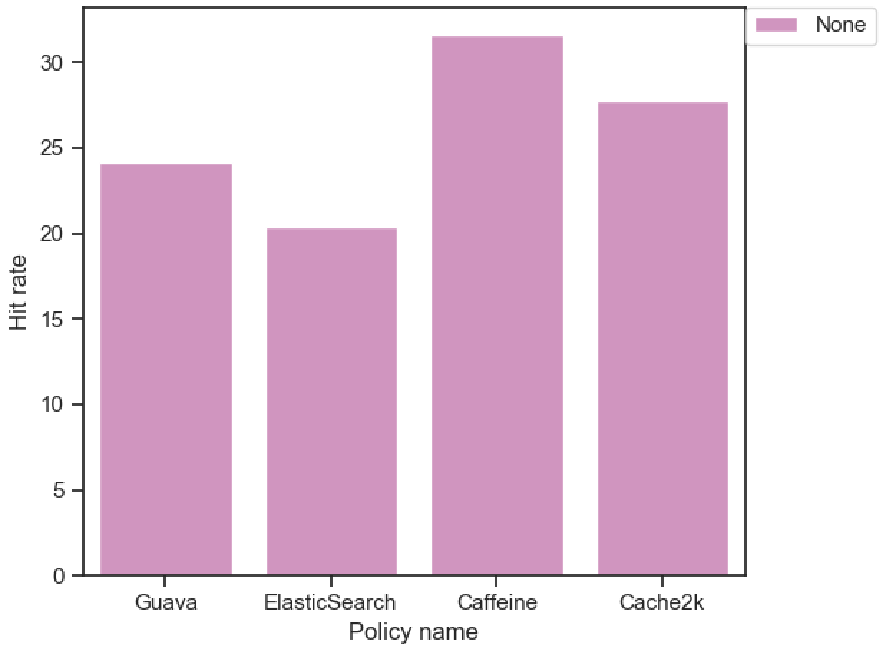

Figure 5.5: The products that support weight based eviction run on the web_0 trace from MSR Cambridge [59] with maximum weight 100000

Ruling out policies with cost based caching on the web_0 trace from MSR Cambridge

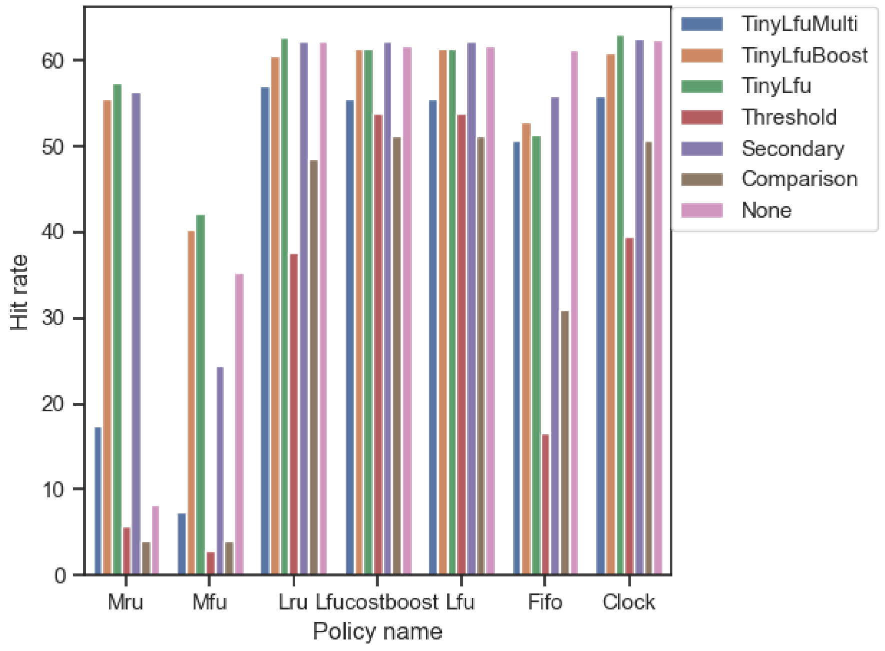

Figure 5.6: Linked policies run on the web_0 trace with maximum size of 10000 where item size is equal to one and cost representing other access meta data.

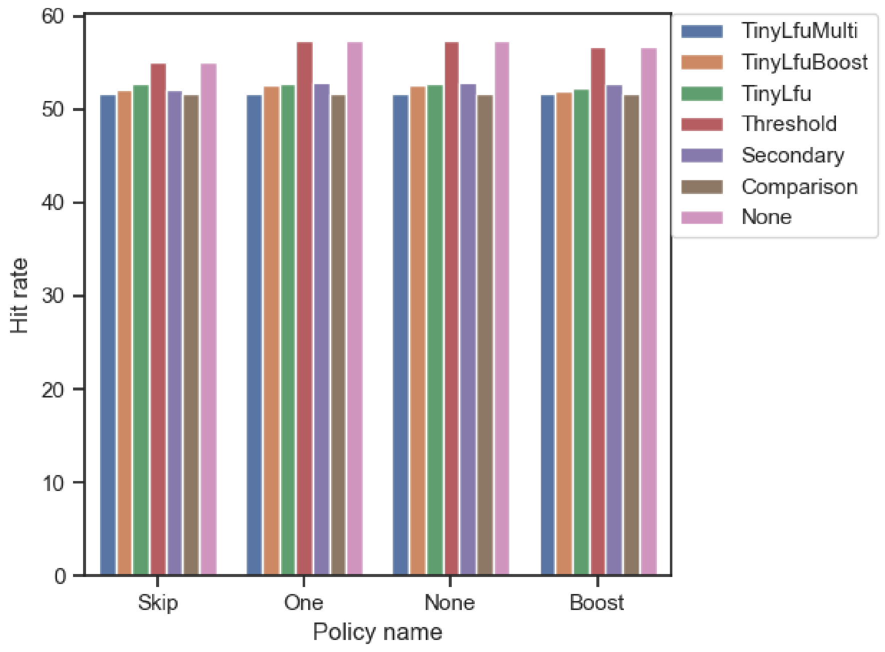

Figure 5.7: Min-heap LFU policies run on the web_0 trace with maximum size of 10000 where item size is equal to one and cost representing other access meta data.

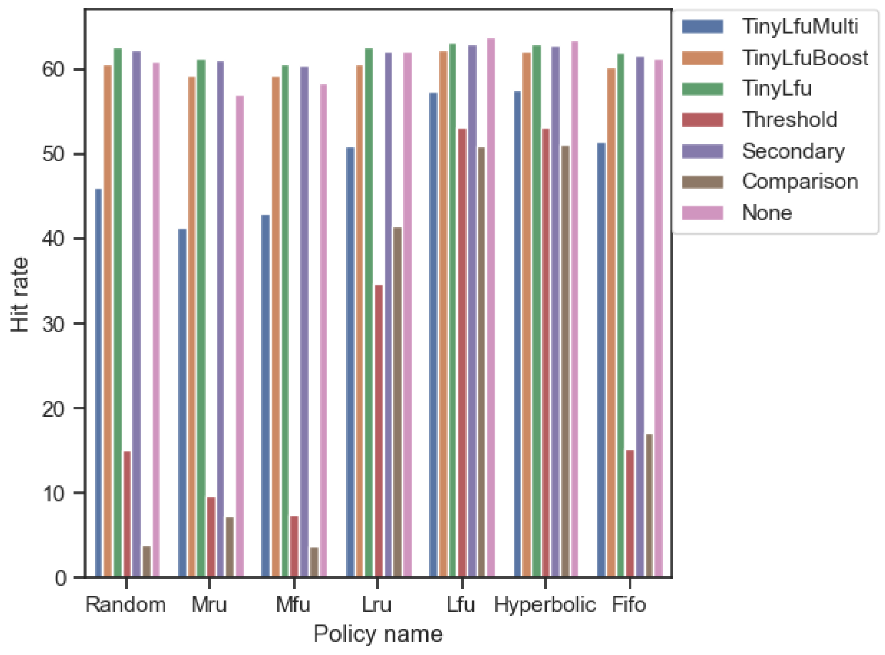

Figure 5.8: Sampled policies run on web_0 trace with maximum size of 10000 where item size is equal to one and cost representing other access meta data.

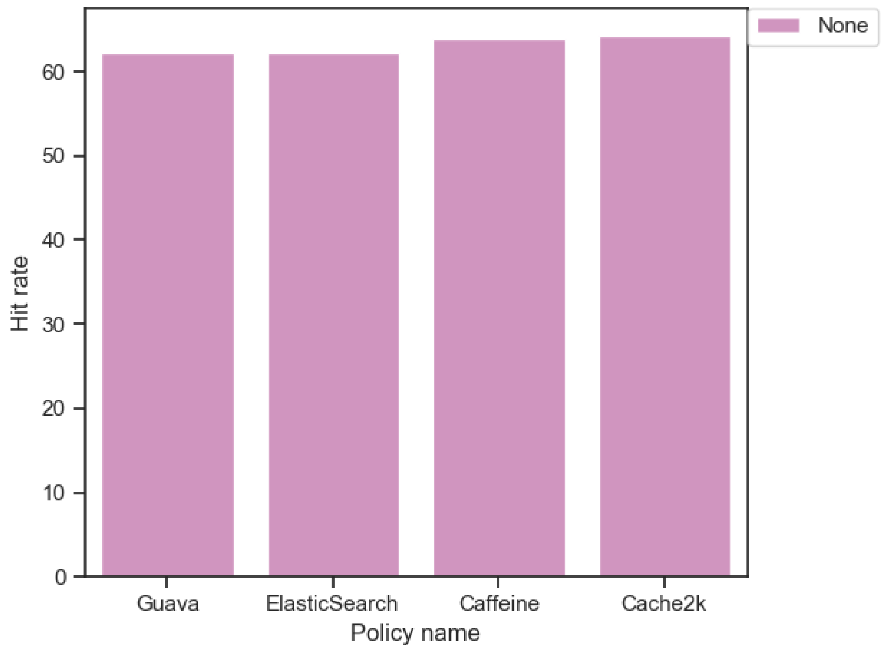

Figure 5.9: Caching algorithms from products and libraries run on web_0 trace with maximum size of 10000 where item size is equal to one and cost representing other access meta data.

Table 5.6: Top 10 policies weighted caching on the Web trace 0 from MSR Cambridge with a maximum size/weight of 100000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
linked.Lfucostboost	Secondary	32.08%	9.91%
linked.Lfu	Secondary	31.77%	9.54%
product.Caffeine	W-TinyLfu	31.61%	9.67%
linked.Lru	TinyLfuMulti	30.88%	7.99%
sampled.Hyperbolic	TinyLfuMulti	30.85%	7.74%
sampled.Lfu	Secondary	30.76%	9.31%
linked.Lru	TinyLfu	30.66%	9.84%
linked.Clock	TinyLfu	30.52%	9.8%
sampled.Hyperbolic	Secondary	30.51%	9.25%
linked.Lru	Secondary	30.27%	9.25%

Table 5.7: Top 10 policies with cost based caching on the Web trace 0 from MSR Cambridge with a maximum size of 10000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
product.Cache2k	None	64.29%	27.51%
product.Caffeine	W-TinyLfu	63.89%	27.4%
sampled.Lfu	None	63.89%	27.4%
sampled.Hyperbolic	None	63.59%	27.1%
sampled.Lfu	TinyLfu	63.14%	26.85%
sampled.Hyperbolic	TinyLfu	63.1%	26.84%
linked.Clock	TinyLfu	63.08%	26.83%
sampled.Lfu	Secondary	62.99%	27.6%
sampled.Hyperbolic	Secondary	62.93%	27.52%
linked.Lru	TinyLfu	62.71%	26.48%

Table 5.8: Top 10 policies in weighted caching run on Web Search 1 trace from the UMass trace repository with maximum size/weight of 1000000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
sampled.Random	Secondary	9.59%	5.51%
sampled.Hyperbolic	Secondary	9.48%	5.53%
sampled.Lfu	Secondary	9.46%	5.52%
sampled.Fifo	Secondary	9.05%	5.23%
sampled.Lru	Secondary	8.77%	5.23%
sampled.Lru	TinyLfuBoost	8.14%	5.24%
sampled.Lru	TinyLfu	8.14%	5.24%
sampled.Fifo	TinyLfu	8.11%	5.1%
sampled.Fifo	TinyLfuBoost	8.11%	5.1%
sampled.Random	TinyLfuBoost	8.07%	5.2%

Table 5.9: Top 10 policies in cost based caching run on Web Search 1 trace from the UMass trace repository with maximum size of 100000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
sampled.Lru	TinyLfuBoost	15.13%	11.51%
sampled.Lru	TinyLfu	15.13%	11.51%
linked.Lfucostboost	TinyLfuBoost	14.97%	11.53%
linked.Lfucostboost	TinyLfu	14.97%	11.53%
linked.Lfu	TinyLfu	14.97%	11.53%
linked.Lfu	TinyLfuBoost	14.97%	11.53%
sampled.Lfu	TinyLfu	14.96%	11.44%
sampled.Hyperbolic	TinyLfu	14.96%	11.43%
sampled.Lfu	TinyLfuBoost	14.96%	11.44%
sampled.Hyperbolic	TinyLfuBoost	14.96%	11.43%

Table 5.10: Top 10 policies in weighted caching run on Web Search 2 trace from the UMass trace repository with maximum size/weight of 1000000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
sampled.Random	TinyLfuBoost	13.93%	8.34%
sampled.Random	TinyLfu	13.93%	8.34%
sampled.Lru	TinyLfuBoost	13.76%	8.29%
sampled.Lru	TinyLfu	13.76%	8.29%
sampled.Hyperbolic	TinyLfu	13.64%	8.52%
sampled.Hyperbolic	TinyLfuBoost	13.64%	8.52%
sampled.Lfu	TinyLfu	13.56%	8.46%
sampled.Lfu	TinyLfuBoost	13.56%	8.46%
sampled.Fifo	TinyLfu	13.29%	7.6%
sampled.Fifo	TinyLfuBoost	13.29%	7.6%

Table 5.11: Top 10 policies in cost based caching run on Web Search 2 trace from the UMass trace repository with maximum size of 500000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
sampled.Lru	TinyLfuBoost	56.99%	50.66%
sampled.Lru	TinyLfu	56.99%	50.66%
sampled.Hyperbolic	TinyLfuBoost	56.87%	50.53%
sampled.Hyperbolic	TinyLfu	56.87%	50.53%
sampled.Lfu	TinyLfu	56.85%	50.49%
sampled.Lfu	TinyLfuBoost	56.85%	50.49%
sampled.Hyperbolic	Secondary	56.51%	51.87%
sampled.Lfu	Secondary	56.49%	51.84%
sampled.Lru	Secondary	56.36%	51.81%
product.Cache2k	None	56.35%	49.88%

Table 5.12: Top 10 policies in weighted caching run on Web Search 3 trace from the UMass trace repository with maximum size/weight of 1000000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
sampled.Random	Secondary	14.21%	8.23%
sampled.Lfu	Secondary	13.99%	8.35%
sampled.Hyperbolic	Secondary	13.9%	8.35%
sampled.Lru	Secondary	13.65%	8.0%
sampled.Random	TinyLfuBoost	13.52%	8.17%
sampled.Random	TinyLfu	13.52%	8.17%
sampled.Fifo	Secondary	13.4%	7.41%
sampled.Lru	TinyLfuBoost	13.36%	8.12%
sampled.Lru	TinyLfu	13.36%	8.12%
sampled.Hyperbolic	TinyLfuBoost	13.23%	8.35%

Table 5.13: Top 10 policies in cost based caching run on Web Search 3 trace from the UMass trace repository with maximum size of 500000.

Policy	**Admittor**	**Hit rate**	**Weighted hit rate**
sampled.Lru	TinyLfuBoost	57.63%	52.15%
sampled.Lru	TinyLfu	57.63%	52.15%
sampled.Hyperbolic	TinyLfu	57.5%	52.01%
sampled.Hyperbolic	TinyLfuBoost	57.5%	52.01%
sampled.Lfu	TinyLfuBoost	57.47%	51.99%
sampled.Lfu	TinyLfu	57.47%	51.99%
sampled.Hyperbolic	Secondary	57.13%	53.29%
sampled.Lfu	Secondary	57.12%	53.26%
product.Cache2k	None	57.04%	51.42%
sampled.Lru	Secondary	56.99%	53.2%

Table 5.14: Top 10 policies in weighted caching run on the Financial 1 trace from the UMass trace repository with maximum weight of 50000.

Policy	Admittor	Hit rate	Weighted hit rate
linked.Clock	None	46.06%	37.83%
product.Guava	None	45.63%	37.28%
linked.Lru	None	45.58%	37.26%
product.ElasticSearch	None	45.58%	37.26%
sampled.Lru	None	45.45%	37.18%
product.Cache2k	None	45.41%	38.38%
sampled.Hyperbolic	None	45.4%	38.1%
sampled.Lfu	None	44.04%	36.84%
product.Caffeine	W-TinyLfu	43.98%	34.5%
linked.Fifo	None	43.16%	35.48%

Table 5.15: Top 10 policies in cost based caching run on the Financial 1 trace from the UMass trace repository with maximum size of 5000.

Policy	Admittor	Hit rate	Weighted hit rate
product.Cache2k	None	43.11%	36.73%
sampled.Hyperbolic	None	42.76%	36.17%
sampled.Lfu	None	42.27%	35.38%
linked.Clock	None	41.95%	34.62%
product.Caffeine	W-TinyLfu	41.61%	34.8%
linked.Lru	None	41.37%	34.02%
product.ElasticSearch	None	41.37%	34.02%
product.Guava	None	41.35%	34.0%
sampled.Lru	None	41.21%	33.91%
sampled.Fifo	None	38.37%	32.19%

Table 5.16: Top 10 policies in weighted caching run on the Financial 2 trace from the UMass trace repository with maximum weight of 50000.

Policy	Admittor	Hit rate	Weighted hit rate
product.Caffeine	W-TinyLfu	75.2%	70.68%
product.Cache2k	None	74.54%	70.42%
sampled.Lru	Secondary	72.61%	63.82%
sampled.Lfu	None	72.37%	67.79%
sampled.Hyperbolic	None	72.13%	67.56%
sampled.Random	Secondary	72.1%	59.42%
linked.Clock	None	72.0%	68.0%
sampled.Hyperbolic	Secondary	71.91%	64.67%
sampled.Lfu	Secondary	71.72%	64.71%
linked.Lru	None	71.4%	67.41%

Table 5.17: Top 10 policies in cost based caching run on the Financial 2 trace from the UMass trace repository with maximum size of 5000.

Policy	Admittor	Hit rate	Weighted hit rate
product.Cache2k	None	65.53%	60.64%
product.Caffeine	W-TinyLfu	64.58%	59.94%
sampled.Lru	Secondary	62.91%	59.86%
sampled.Lru	TinyLfu	62.73%	58.93%
sampled.Lru	TinyLfuBoost	62.73%	58.93%
linked.Clock	None	62.73%	57.54%
sampled.Hyperbolic	None	62.47%	56.74%
sampled.Hyperbolic	Secondary	62.15%	58.75%
sampled.Lfu	Secondary	62.08%	58.76%
sampled.Random	Secondary	62.04%	58.83%

Table 5.18: Top 10 policies in weighted caching run on the proxy_0 trace from the MSR Cambridge with maximum size/weight of 1000000.

Policy	Admittor	Hit rate	Weighted hit rate
linked.Fifo	Secondary	35.34%	13.13%
sampled.Fifo	TinyLfu	32.71%	20.91%
linked.Lru	Secondary	32.15%	22.78%
linked.Clock	Secondary	31.02%	22.22%
sampled.Random	TinyLfu	27.05%	21.39%
sampled.Lfu	TinyLfu	25.9%	21.64%
linked.Clock	TinyLfu	25.68%	21.59%
sampled.Hyperbolic	TinyLfu	25.63%	21.58%
sampled.Lru	TinyLfu	25.63%	21.58%
sampled.Fifo	TinyLfuBoost	25.52%	20.74%

Table 5.19: Top 10 policies in cost based caching run on the proxy_0 trace from the MSR Cambridge with maximum size of 10000.

Policy	Admittor	Hit rate	Weighted hit rate
product.Cache2k	None	91.06%	54.87%
sampled.Lru	TinyLfu	90.84%	55.49%
sampled.Lru	Secondary	90.84%	56.74%
sampled.Hyperbolic	Secondary	90.79%	56.19%
sampled.Hyperbolic	TinyLfu	90.77%	55.01%
product.Caffeine	W-TinyLfu	90.76%	54.77%
sampled.Lfu	Secondary	90.7%	53.48%
sampled.Lfu	TinyLfu	90.69%	52.35%
sampled.Random	Secondary	90.69%	56.09%
sampled.Random	TinyLfu	90.67%	54.7%

Table 5.20: Top 10 policies in Weighted based caching run on the proxy_1 trace from the MSR Cambridge with maximum size/weight of 2000000.

Policy	Admittor	Hit rate	Weighted hit rate
linked.Clock	None	29.13%	34.48%
linked.Lru	None	29.09%	34.46%
product.ElasticSearch	None	29.09%	34.46%
product.Guava	None	29.09%	34.46%
sampled.Lru	Secondary	29.09%	34.46%
sampled.Lru	None	29.09%	34.46%
sampled.Fifo	None	28.83%	34.37%
linked.Fifo	None	28.83%	34.37%
sampled.Fifo	Secondary	28.83%	34.37%
sampled.Random	Secondary	28.06%	33.53%

Table 5.21: Top 10 policies in cost based caching run on the proxy_1 trace from the MSR Cambridge with maximum size of 10000.

Policy	Admittor	Hit rate	Weighted hit rate
linked.Clock	TinyLfu	89.57%	91.7%
sampled.Fifo	TinyLfu	89.27%	90.59%
sampled.Random	TinyLfu	89.21%	90.63%
sampled.Fifo	Secondary	88.97%	92.86%
linked.Clock	Secondary	88.61%	91.95%
linked.Fifo	Secondary	88.25%	92.57%
sampled.Hyperbolic	TinyLfu	88.23%	89.95%
linked.Lru	TinyLfu	88.19%	90.0%
sampled.Fifo	TinyLfuBoost	88.08%	92.76%
sampled.Lfu	TinyLfu	87.99%	89.88%

Chapter 6

Results and analysis

The results will be evaluated and explained before being discussed. The evaluation and discussion for the historical access patterns experiment include some design concerns about adaptive cache architectures. Throughout the simulations, there is usually a very clear discrepancy between combinations of policies that use admission and those that do not, especially for weighted caching. Nonetheless, there are still several simulations where using no admission policy comes out with a strong performance.

6.1 Results

The results of each of the experiments.

6.1.1 Historical access patterns

As visible both by the table 5.3 and the figure 5.1, when a loopy workload is expected, preparing the system helps minimize the scan pollution.

Even though LeCaR with weight and learning rate reset (lecar.LR_W) handled the loopy workload best, it still ended up with a hit rate of 37.83%, while Arc, without any system preparation, ended up at 40.16% hit rate. Both LeCaR with learning rate reset (lecar.LR) and plain LeCaR (lecar.None) came in at 36.58% hit rate. This is due to ARC/LeCaR design, when a loopy workload occurs, no internal adaptive steps will occur in the cache since it is based on ghost hit design, which is further discussed in chapter 6.2.1.

6.1.2 Ruling out policies

State-of-the-art cache replacement policies are both a combination of admission and eviction policies. However, this means that it is vital to select the optimal combination to capture data orthogonality in the best manner possible, similar to what state-of-the-art policies already do.

Ruling out some policies is quite simple. Based on the charts presented in chapter 5.5.2 it becomes clear that several policies perform poorly compared to other policies, especially for the weighted simulations. Some policies which are ruled out for further experimentation and simulation will still have the potential to be performant policies for some use cases, but not for general-purpose cache performance.

Table 6.1: The percentage difference of using the sampled policies compared to the exact implementations run on the web 0 trace with maximum size 100000.

Policy name	**Difference of using sampled vs exact**
Fifo	+0.06%
Fifo_Secondary	-71.1%
Fifo_TinyLfuBoost	+35.32%
Fifo_TinyLfu	+51.49%
Lru	-0.54%
Lru_Secondary	-0.69%
Lru_TinyLfuBoost	+1.29%
Lru_TinyLfu	-6.46%
Lfu	+4.08%
Lfu_Secondary	-3.18%
Lfu_TinyLfuBoost	+7.63%
Lfu_TinyLfu	+0.52%

Admission policies

Table 5.4 and 5.5 presents the average performance difference of using the different admission policies, where Comparison for weighted came in at -26.12% and cost based at -39.92%. Secondary came in with a performance regression on weighted caching at -8.56%, while for cost-based it came in at +29.19%. The Threshold admission came in at -24.04% for weighted caching and -38.69% for cost-based caching. TinyLfu, considered state-of-the-art comes in at +36.20% for weighted and +32.43% for cost-based. TinyLfuBoost, slightly behind TinyLfu at +27.44% for weighted caching and +29.54% for cost-based caching. TinyLfuMulti came in at -12.60% for weighted caching and -10.61% for cost based caching. The average performance change compared to using no admission policy may have some extremes, due to Mru and Mfu still being a part of the simulation.

As visible in figure 5.2 and the rest of the web_0 simulation, TinyLfuMulti does have some cases where the performance is adequate, but it varies. The Threshold admission policy paired with Lru came in surprisingly at 27.56% hit rate, while if paired with Lfu it ended up with 25.70% hit rate, while both of these numbers are not particularly bad, the Threshold admission policy performs poorly when paired other policies such as Fifo. The Comparison admission policy paired with Lru comes in at 17.48% and with Lfu 26.04%. Looking at the Comparison performance for the two policies it is easy to observe that it is much worse when paired with Lru than for Lfu.

Eviction policies

Figure 5.2 shows both Mru and Mfu lagging behind the other linked policies, especially if we consider the variants with no admission policy. Mfu without any admission policy came in with a hit rate at 16.71%, while Mru without admission came in at 8.48%. Mfu has a surprisingly high cache hit rate, being quite close to Fifo at 17.72%. In the cost-based experiment, Mfu performs worse compared to the other policies, which can be seen in figure 5.6.

Mru with TinyLfu at 22.63% outperforms any Fifo policy combination, with the best being Fifo with Secondary at 22.56%. Comparing Fifo and Lru we can see the effects Lru obtains by using either TinyLfu, TinyLfuBoost or Secondary. The same trend is not present for Fifo.

6.1.3 Performance simulation

Simulations are intended to show different performance characteristics of different combinations when faced with different datasets. Performance and performance differences are expected to vary from dataset to dataset, which is why each simulation of a group of traces is explained and analyzed individually.

Both cache hit rate and weighted hit rate are important metrics. When an experiment results in a policy being superior in terms of hit rate, while not in weighted hit rate, it is difficult to evaluate which one was more performant without testing it further in a system.

Web trace

The web_0 trace has figures for all categories of eviction and admission policy combinations, seen in figures 5.2, 5.3, 5.4 and 5.5 for weighted caching and figures 5.6, 5.7, 5.8 and 5.9.

In the web_0 simulation, Lfu with Secondary admission came in at 31.77% cache hit rate, while LfuCostBoost with Secondary admission came in slightly better at a hit rate of 32.08%, which is promising results for LfuCostBoost. The difference of 0.31% may not seem big, but it quickly adds up, especially if we consider the weighted hit rate as well, where the difference is 0.37%. Since both LfuCostBoost and Lfu, the top-2 policies both use the Secondary admission policy the differences between the two lies entirely in the eviction policy. However, it is often the interaction between admission and eviction and their combined ability to capture a large variety of access pattern that determines performance. This is why capturing data orthogonality has been a popular design reason for modern state-of-the-art cache libraries such as Caffeine. The same trend, with LfuCostBoost performing slightly better than Lfu is also the case when they are paired with TinyLfuBoost where LfuCostBoost came in at 26.54% hit rate and Lfu at 26.46%.

Nonetheless, LfuCostBoost with Secondary admission is only ranking at 7th place when it comes to weighted hit rate. The Lfu heap Boost policy with no admission policy is the top policy when evaluating the weighted hit rate, which came in at 10.06%. The difference from heap Boost to LfuCostBoost in terms of weighted hit rate is 0.15%, while the difference of the two combinations of policies in terms of hit rate is 2.01%. Even though the weighted hit rate is quite similar between the two, a difference of about 2% in cache hit rate is significant.

For cost-based caching, many of the top-10 are sampled variants. Cache2k, which uses a Clock-Pro variant came out with 64.29% hit rate, which is surprisingly high considering Cache2k does not employ an admission policy. Caffeine with W-TinyLfu, comes in 2nd at 63.89%. When considering the weighted hit rate, sampled Lfu with TinyLfuBoost comes in at 27.89%, sampled Hyperbolic with TinyLfuBoost at 27.68% and sampled Lfu with Secondary at 27.6%. In terms of weighted hit rate, the top-6 combinations all consists of TinyLfuMulti, which is surprising. At the top is the sampled Hyperbolic with TinyLfuMulti at 30.16% weighted hit rate. Even though the TinyLfuMulti policy is performing adequately in terms of weighted hit rate, it is no guarantee for cache hit rate, which for the sampled Hyperbolic with TinyLfuMulti comes in at 57.54%.

Web Search traces

The top-10 for the WebSearch1 simulation for weighted caching reveals only sampled eviction policies amongst the top-10, with the next one not being sampled is the linked Clock policy with TinyLfuBoost admission at 7.72% hit rate. Sampled random came out on top with 9.59% hit rate, with sampled Hyperbolic coming in second at 9.48%. Of the top-10 admission policies, only TinyLfu and TinyLfu based admission policies are prevalent. Looking at sampled Lru with Secondary, TinyLfuBoost and TinyLfu admission the differences between them is that Secondary ended up with 8.77% hit rate, while both TinyLfuBoost and TinyLfu came in at 8.14% hit rate. The combination of sampled Lru with TinyLfuBoost and TinyLfu both ended up with a weighted hit rate of 5.24% and sampled Lru Secondary at 5.23%. For cost-based caching of the WebSearch1 trace, the top-2 policies sampled Lru with TinyLfuBoost and TinyLfu came out with an equal performance at 15.13% hit rate and 11.51% weighted hit rate. The rest of the top-10 are different variations of TinyLfu and TinyLfuBoost for admission and LfuCostBoost, Lfu, sampled Lfu and sampled Hyperbolic as the eviction policy. When evaluating the weighted hit rate, sampled Lfu with Secondary is coming out strong with 11.91% weighted hit rate. It becomes apparent that in this simulation, the TinyLfuBoost will not increase performance compared to TinyLfu. Even though it is not ideal, encountering weights or costs which does not play well with the scale down function makes TinyLfuBoost equivalent to TinyLfu is to be expected.

The results from weighted caching on the WebSearch2 trace presents an interesting trend where all the top-10 eviction policies are sampled ones and they are paired with either TinyLfuBoost and TinyLfu. As with the cost-based WebSearch1 simulation, each of the sampled policies paired with either TinyLfu or TinyLfuBoost have identical performance. This result indicates clearly displays that for some datasets the TinyLfuBoost enhancement for TinyLfu will not increase hit rates, but it will not decrease the hit rate either.

When it comes to cost-based caching for the WebSearch2 simulation, as with the weighted simulation, there are combinations of an eviction policy with TinyLfuBoost and TinyLfu that have identical performance. When looking into weighted hit rate, sampled Hyperbolic with Secondary admission comes out with the best performance at 56.51% hit rate and 51.87% weighted hit rate which is 1.21% better than sampled Lfu with TinyLfuBoost, which is the top-performing combination in terms of hit rate.

For weighted caching in the WebSearch3 simulation, the top-10 consists solely of sampled eviction policies. The top one being sampled Random with Secondary admission at 14.21% hit rate and 8.23% weighted hit rate. When evaluating the weighted hit rate, the sampled Lfu with Secondary along with many other policies came in at 8.35% weighted hit rate and 13.99% hit rate.

For cost-based, the majority of the eviction policies amongst the top-10 are sampled ones, while all the admission policies are TinyLfu or TinyLfu based. Cache2k is the best non-sampled policy combination with a hit rate of 57.04%. Sampled Lru with TinyLfuBoost and TinyLfu have exactly the same performance, while sampled Lru with Secondary performed slightly worse than the two, regressing 0.12% for hit rate, and 1.62% for weighted hit rate. Sampled Hyperbolic with Secondary admission is the top-performing policy when it comes to weighted hit rate at 53.29% and 57.13% hit rate.

Financial traces

For the weighted Financial 1 simulation, there is no combination of policies that use admission policies except Caffeine. This can indicate that the complexity and variety of data access patterns within the dataset may not be that challenging for policies to capture. Surprisingly, linked Clock came in with the best performance at 46.06% hit rate before Guava at 2nd place with 45.63% and Lru came in third with 45.58%. Cache2k has the highest weighted cache hit rate at 38.38% and 45.41% hit rate. For cost-based caching, Cache2k comes in at 43.11% hit rate, and 36.73% weighted hit rate, performing best both in terms of hit rate and weighted hit rate. Sampled Hyperbolic comes in at 2nd place with 42.76% and sampled Lfu third with 42.27% hit rate. For both weighted and cost-based caching the only policy combination amongst the top-10 with an admission policy is the Caffeine library with W-TinyLfu.

For the Financial 2 trace, the weighted simulation reveals Caffeine and Cache2k as the top-2 policies with 75.2% and 74.54% hit rate. Coming in third is sampled Lru with Secondary admission with 72.61% hit rate and then sampled Lfu without any admission policy at 72.37%. The majority of the top-10 policies are sampled with sampled Lru, Lfu and Hyperbolic. Sampled policies without admission policies are very dependent on sample selection, which assuming from the performance numbers seem to be very on point in this simulation. Of the top-10 policies, only Caffeine with W-TinyLfu and sampled policies paired with the Secondary admission policy are prevalent.

In the cost-based simulation, Cache2K and Caffeine are the top-2 performing policies, with 65.53% and 64.58% hit rate. Sampled policies are dominating the top-10 with sampled Lru, sampled Hyperbolic and sampled Lfu combinations. Sampled Lru with Secondary comes in at 62.91% hit rate, while sampled Lru with both TinyLfu and TinyLfuBoost comes in at 62.73% hit rate. The difference between Secondary and TinyLfu/TinyLfuBoost in terms of hit rate is not big, but the performance difference in the weighted hit rate of using the Secondary admission policy is 0.93%. The sampled Hyperbolic policy is present without an admission policy at 62.47% hit rate and with Secondary admission at 62.15% hit rate. The difference in terms of weighted hit rate is 2.01% in favour of the combination with the Secondary admission.

Proxy traces

From the simulations on the proxy traces, it reveals that in the cost-based simulations, even keeping the cache maximum size at 10000 results in a surprisingly high hit rate, which implies a limited number of unique keys. For the weighted proxy_0 simulation, Fifo with the Secondary admission policy performed the best in terms of hit rate. However, it ends up with a low weighted hit rate, at 13.13%, which is surprisingly low because of the high hit rate of 35.34%. The best policy in terms of weighted hit rate was Caffeine with a hit rate of 24.59% and weighted hit rate of 23.03%. The unexpected high hit rate and low weighted hit rate of Fifo with Secondary admission led to a new simulation where the operator of the Secondary admission policy flipped as done in the cost-based simulations. It led to a distinct regression for every policy combination that the Secondary admission policy was a part of, which means that the result is valid. Sampled Fifo with TinyLfu came in second with 32.71% hit rate and 20.91% weighted hit rate while Lru with Secondary came in third at 32.15% hit rate and 22.78% weighted hit rate.

For cost-based caching, the Cache2k policy comes out the strongest with a hit rate of 91.06%. The rest of the top-10, except Caffeine are sampled policies with TinyLfu based admission policies. However, when evaluating the weighted hit ratio, we find that sampled Lfu with TinyLfuBoost came in at 90.49% hit rate and 61.23% weighted hit rate. In this case, the 0.57% difference is nothing compared to the 6.36% difference in terms of weighted hit rate.

The weighted proxy_1 simulation is one of the simulations where the majority of the top-10 combinations does not contain admission policies. It could be related to the fact that the dataset is expected to have a limited amount of unique keys, even though they all come with a weight of their own. Linked Clock came out on top with 29.13% hit rate and 34.48% weighted hit rate, while linked Lru, ElasticSearch, Guava, sampled Lru with Secondary and sampled Lru came in with 29.09% hit rate and 34.46% weighted hit rate.

In the cost-based proxy_1 simulation, Clock with TinyLfu admission came out with the highest hit rate at 89.57% and 91.7% weighted hit rate. However, the top-performing policy when it comes to weighted hit rate is sampled Lfu with TinyLfuBoost at 86.21% hit rate and 93.73% weighted hit rate. Sampled Fifo with TinyLfu came in second at 89.27% hit rate and 90.59% weighted hit rate, while sampled Fifo with Secondary admission came in at 88.97% hit rate and 92.86% weighted hit rate.

6.2 Discussion

In this chapter, the results of the project are discussed and analyzed with the research questions in mind.

6.2.1 Historical access patterns

By looking at the figure 5.1 we can clearly see the effects of preparing an adaptive policy which internally uses a combination of policy to choose the most scan resistant one. This shows the potential behind utilizing historical access pattern to withstand scans. The experiment does not show the harm of doing a wrong decision, which is an important aspect of adaptive cache replacement. However, algorithms could be specifically designed to handle situations where adapting based on the recent and historical past disagrees.

If a smart system is capable of finding and isolating trends in data access patterns, it would enable adaptive cache replacement policies to account for historical patterns as well as the recent past when adapting to the workload. If an adaptive policy finds itself in a situation where both adapting based on recent past and historical past agrees to adapt to a different state, it could reach the optimal state quicker by setting larger step size in hill climbing used by W-TinyLfu, setting learning rate higher in LeCaR and change the size quicker in ARC.

For the experiment, the change in learning rate will have little to no effects in the design of LeCaR due to it only adapting its internals when a hit occurs in either of the ghost lists which does not happen many times during a scan, making the change in weights necessary to withstand loopy workloads for LeCaR. Around discrete-time of 7000, we can see in figure 5.1 that the LR_W line plot is avoiding the drop in hit rate due to loopy access patterns. Comparing LR_W to LR, which only minimizes learning rate, it is clearly experiencing cache pollution similar to LeCaR with no manual adaption. LeCaR design is resistant towards scans if the Lfu weight is substantially larger than the Lru weight. Even though the system prepared itself for loopy workloads, ARC still came out with higher cache hit rate than all LeCaR variants, but as seen in figure 5.1, at the end of the intervention period of the human expert, LeCaR with LR_W has higher hit rate.

Work on utilizing historical access patterns in caching is limited by the lack of high quality publicly available datasets, but also by cache replacement policy design. Due to the lack of datasets, it was either necessary to try to create a dataset using combinations of traces or proceed with a synthetic trace. Very few combinations of traces ended up being suitable for the experiment, which limited the creativity of the experiments and was eventually the reason for the experiment ended up being quite simple. The implementation of the experiment is that it requires manual intervention of some sorts, it is not a smart system which is able to self-adapt based on historical access patterns. It requires a human expert, or a decision support system with knowledge of when the system is expected to enter a scan-like workload.

Nonetheless, LeCaR with Lru and Lfu is excellent at demonstrating the potential behind utilizing historical access patterns because of its design. Even though LeCaR manages to get through the scan-like workload when adjusting both weights and learning rate, ARC still comes out as the superior policy. The performance of LeCaR, as described in the LeCaR article versus ARC has not been successfully reproduced in the experiments conducted in this project, no matter what the settings were set to.

Adaptive cache replacement policies, such as LeCaR have the potential to utilize historical access pattern to enhance adaptation by having the internal state optimized for the expected workload change. This could also be applied to other policies which adapt to workload changes, such as W-TinyLFU.

Comparing the state-of-the-art W-TinyLFU with LeCaR, the issues of LeCaR becomes apparent. In a scan like a workload, W-TinyLFU will adapt due to its design to use hill climb optimization after the hit rate. LeCaR on the other hand and ARC for that matter will not adapt because it relies on adaptive steps after hits to either ghost list. In a scan-like workload, not many hits can be expected, which means that LeCaR will be stuck with its current configuration, which can be seen in figure 5.1, where the red line(set learning rate low) and green line(do nothing) is equal.

ARC is scan resistant in the sense that if a scan-like workload occurs, items will be added to the L_1 cache without being added to the L_2 cache. For LeCaR, based on the current state, new items will be added at random between the two caches. The worst-case for scan resistance for LeCaR is that every item gets added to the LRU cache, which is similar to what ARC accomplishes. If we assume an even distribution amongst the two caches, LeCaR will have superior performance under a scan-like workload because of LFU.

6.2.2 Ruling out policies

The discussion for ruling out policies. The varying performance and little to no benefit over well-established policies was the number one reason for exclusion for both admission and eviction policies.

Admission policies

As seen in tables 5.4 and 5.5 the performance of the Comparison, Threshold and TinyLfuMulti admission policies are on average worse than using no admission policy at all. They lack the overall ability to adapt to workload changes and evaluate total item lifetime cost when admitting or evicting. TinyLfuMulti and Threshold lack a proper scaling function that makes the weight/cost and frequency of the same magnitude to make it possible to determine how it should weigh the item's attributes against each other.

Looking at weighted caching with the web_0 trace, TinyLfuMulti has decent performance throughout the experiments but it often results in low weighted hit rate. For cost-based, the performance is also adequate. It is even able to increase the weighted hit rate by almost 10% on average. For both the weighted simulation and the cost-based simulation, a simple division or multiplication without scaling the weight or cost to the same magnitude as the frequency results in very unbalanced total cost which again leads to unstable performance. Furthermore, the limitation of TinyLfuMulti is scaling the weight or cost with the frequency, but it is also limited by the 4-bit sketch and low reset interval. This is because if most cells in the sketch have a low frequency, we will have quickly have issues with weight/cost being multiplied by a frequency estimate of 0 or a very low number before being compared. Having the sketch operate with a low reset interval makes the TinyLfuMulti also value recency more than the frequency in the sense that items with a steady inter-reference interval may build up their access frequency, but items with short bursts of access may still be prioritized by the admission policy.

TinyLfuMulti was first implemented equally for both weighted caching and cost-based caching, but due to the characteristics of weighted caching and the weighted caching problem, it was changed to division by the weight of the item. This was done because ideally, a cache with weighted items would prefer to admit items of high frequency and low weight in order to fit more items in the cache. However, the approach of using division shares the same performance characteristics as using multiplication, it still lacks the stability and performance required to compete with the TinyLfu based policies.

The Threshold admission policy was created as a simple admission policy that had one of the simplest forms of admission, a threshold. A static threshold will not be very helpful as it could easily get stuck evaluating the same victim over and over again if the threshold is high at this point. The Threshold admission policy has a 1% threshold change based on whether it is weighted or cost-based admission. It is a relatively small change, and if unlucky with a high or low threshold, it could take many iterations admitting new items again. The change parameter could be tuned to converge quicker into a reasonable threshold to avoid getting stuck, however, this could increase the admission rate, which again could affect performance. The major difficulties with approaches to threshold admission are trying to come up with a reasonable threshold that is able to fill the cache with desirable data and at the same time reject items. The Threshold admission policy has the same issues as the Comparison admission policy. It does not evaluate total item cost when it evaluates items to set a new threshold. If the Threshold was changed from only having a weight/cost threshold to a total item cost threshold, then the performance could potentially be more stable. However, as with TinyLfuMulti and other policies, this is dependent on determining how the scale between weight/cost, frequency and other item attributes should be calculated.

The performance of the Comparison admission policy is unstable and usually worse than using no admission policy at all. The Comparison admission policy as with the Threshold was included to display the effects of having simple and unstable admission policies in a caching system. The Comparison admission policy only evaluates weight or cost when it is to make an admission decision. It does not evaluate any other item attribute, which makes the cache admit items without evaluating how much it will contribute to cache performance.

In conclusion, the admission policies TinyLfuMulti, Threshold and Comparison are excluded for further experimentation. They lack the ability to adapt to different datasets and in some cases, they end up performing worse than using no admission policy. Fine-tuning and further enhancing them could be worth it but without knowledge of the dataset, or programmatic setting cost for cost-based caching, the increased computational complexity involved with improving them make them inapplicable for many high-performance scenarios. Thus, in their current state, without expert knowledge of the data, they are not applicable for NoSQL databases and full-text search engines. If they have insight into the data and domain, TinyLfuMulti has the ability to become viable because of the design similarities with TinyLfuBoost without the sketch over-estimation issues.

Eviction policies

For eviction policies, both Mru and Mfu were expected to perform poorly, because for general purpose caching they do not keep relevant and highly referenced items within the cache.

Mfu may have performed well on weighted caching due to items of high weight populating the cache. Evicting the wrong item is not as costly as with cost-based caching due to a smaller eviction pool. Looking at sampled Mru and sampled Mfu performance, we can see that the reduced eviction pool usually helps specialized policies. Mru and Mfu do not benefit equally from using an admission policy, as Mru with TinyLfu and TinyLfuBoost outperforms both Mfu and Fifo with the same admission policies. Pairing Mfu with TinyLfu, and

TinyLfuBoost admission policies results in a combined caching system where the admission policy and eviction policy capture the same item attribute, frequency. But the admission policy prioritizes items with larger frequency, while the eviction policy does the opposite. Mru on the other hand will not find itself in capturing the same item attribute. Thus it will perform significantly better than Mfu when paired with TinyLfu and TinyLfuBoost. Both Mru and Mfu perform poorly because they are specialized eviction policies, and for a general-purpose web I/O trace, they will lag behind the other policies, which is why they are not included in further experimentation.

Sampled policies minimize the issues of using specialized policies because the eviction pool is smaller. Nonetheless, sampled policies will find themselves in situations where the pool consists of a few good candidates for eviction, and it will not select any of them. Nonetheless, sampled policies will usually be performant, and for many systems, it will be worth choosing over exact implementations considering its low algorithmic complexity. As can be seen in both figure 5.4 and figure 5.7, the performance of sampled policies can compete with the exact implementations, which is presented in table 6.1.

The min-heap based Lfu eviction policies all perform adequate, but compared to the O(1) implementation of the Lfu, the cost of using a min-heap with worse algorithmic complexity for common operations makes them candidates to be ruled out as well. However, the reason for including them is to observe how approaching the weighted/cost-based caching problem with ageing, they are included for further experiments. The O(1) implementation of the Lfu could age its frequency nodes all together if the ageing is uniform no matter what the weight/cost of the item is. If the goal is to age the individual items based on weight/cost, the O(1) model of the Lfu suddenly becomes equally hard to keep performant as the min-heap based Lfu due to the need for iteration.

In conclusion, Mru and Mfu, both normal and sampled variants are excluded for further evaluation. They may very well be applicable for a specialized component in NoSQL databases and full-text search engines, but not for general-purpose caching.

6.2.3 Weighted and cost based caching performance simulation

For research question 1, while hit rate and weighted hit rate are correlated, small improvements in weighted hit rate without sacrificing any, or very little hit rate could be ensuring even higher performance for a NoSQL database or a full-text search engine. This is prevalent in several results, but it is impossible to define a reasonable threshold for what constitutes as a little sacrifice in terms of cache hit rate.

Throughout the simulations, results vary a lot from simulation to simulation. However, for weighted caching, there were very few policies amongst the top-10 results that did not have an admission. The financial traces from the UMass trace repository ended up with very few combinations that used admission policies amongst the top-10, which could indicate that the traces had a simple access pattern since both Clock and Lru had an adequate performance.

For weighted caching, using a frequency-based admission policy has been successful due to frequency being a very good attribute to evaluate when deciding whether to admit it or not. Considering the state-of-the-art, frequency is currently the best estimator for total lifetime cost throughout its lifetime within the cache. The Secondary and TinyLfuBoost extensions of TinyLfu, they are sometimes capable of outperforming their originating policy. The Secondary policy is often amongst the top-10 policies, and there are several examples of it increasing performance when comparing it to TinyLfu in the same combination. In cost-based caching, the results ignore the actual size of the items cached will not be as precise as if both retrieval cost and actual byte size were included, but it is still applicable due to it showing the potential behind using a specific combination of admission and eviction policies where there is only a restriction of the maximum number of items. Not using an admission policy will in most cases, for most deployment scenarios lead to reduced performance due to more items passing the cache. The scan resistance of eviction policies may be sufficient to withstand scans, but having an admission policy could help prevent items from entering the cache in the first place.

Using TinyLfu and TinyLfu based admission policies usually results in enhanced cache hit rate. In weighted caching, the frequency estimates from the Count-Min sketch provide a good estimation for total item cost. However, admitting based on frequency, or any item attribute solely is no guarantee for high performance because of the weighed caching problem, explained in chapter 4.2.1. In this thesis, TinyLfu used the 4-bit sketch combined with a low reset interval, which means that the Count-Min sketch is able to keep track of inter-reference recency as well as frequency. In other words, unless an item gets updated regularly, its frequency will be kept low by the reset intervals and if there is a sudden burst of access to an item, we know that its frequency will not surpass 15 (4-bit sketch). These sudden bursts of access to one or a few items could lead to artificially good victims presented from the eviction policy, which could lead to the cache being stuck evaluating the same item for a while. Until the reset interval is able to halve its estimated frequency.

When comparing TinyLfuBoost and TinyLfu it shows that they are often performing equally. This could reveal that the cost of the dataset is not that big, making the scale down of the cost to not surpass one, making the TinyLfuBoost equal to TinyLfu. Having TinyLfuBoost act as TinyLfu when weight/costs are low makes its applicability better due to the already proven performance of TinyLfu. All the TinyLfu based policies use a 4-bit sketch to keep track of frequencies, meaning that some items with large weight/cost will very quickly get a high/max frequency, but with a periodic reset interval of 16, they will quickly come down again. Nonetheless, this sudden spike in frequency quickly adds up and may contribute to the over-estimation issues of the Count-Min sketch. Over-estimation issues could in the worst case lead to a wrong admission decision. TinyLfuBoost with a 64-bit sketch could have possibly been better at ensuring high weight/cost items stay in the cache due to the frequency can surpass 15. Even with a 64-bit sketch, an approach of scaling down the weight/cost to ensure that the sketch stays somewhat balanced and relative to the actual frequencies and cost is necessary and perhaps even more important due to the Count-Min sketch 64-bit variant is not "protected" by low reset intervals and low-frequency limits. The average performance change of combinations of the TinyLfuBoost admission policy, presented in tables 5.4 and 5.5. When comparing TinyLfu and TinyLfuBoost we can see that for cost-based caching, TinyLfuBoost came out worse than TinyLfu in terms of hit rate, but managed to perform better in terms of weighted hit rate. This trend is also present in the cost-based proxy simulations, where TinyLfuBoost usually lags a bit behind TinyLfu, but it is able to increase the weighted cache hit rate.

TinyLfuBoost is a step in the right direction for RQ1, but it is necessary to consider sketch over-estimation combined with approaches to scale down weight/cost in order for it to be a more performant approach. The over-estimation issue could propagate and affect the performance of the cache for both the 4-bit and 64-bit sketch. TinyLfuMulti has the same potential as TinyLfuBoost but without making the over-estimation of the sketch worse. As already mentioned in chapter 6.2.2, its success is solely dependent on a function that is able to scale down the weight/cost.

The Secondary admission policy, which is a very simple extension of TinyLfu has had a surprisingly good performance. In the simulations, a 4-bit sketch with a low reset interval is being used, which means that many frequency estimates will be equal. Weighted caching will have a benefit of admitting items of lower weight due to total accumulated weight is the constraint on how many items can be present in the cache at the same time. Having the Secondary admission policy prioritize items of lower weight has for the majority of the weighted simulations both led to increased hit rate and weighted hit rate compared to TinyLfu. One could argue that evicting an item of high weight for an item of lower weight could be undesirable due to the possibility that the item of large weight has a large cost in retrieving the item from its primary storage medium. But since weight is not always directly linked to retrieval cost, it is easier to support the effects of admitting items of lower weight due to the cache being able to fit more items.

For cost-based caching, the operator is flipped, because we have no weight constraint and we want to keep all the items of high cost since it could potentially increase the weighted hit rate of the caching system. This approach works well in cost based caching and it usually enhances the weighted hit rate, but it might not be able to enhance the hit rate. The cost based web 0, WebSearch2, WebSearch2, Financial2, Proxy0 and Proxy1 are examples of this trend. In other words, it is able to optimize weighted hit rate, but in order to do so, the hit rate regressed a tiny bit. This leaves us with determining how weighted hit rate should be evaluated compared to hit rate.

As already mentioned, the performance of the Secondary admission policy is highly dependent on the sketch size and reset interval. Having the 64-bit sketch in play instead of the 4-bit one could result in fewer equal estimates which again would limit the effect of the admission policy.

The min-heap based eviction policies do share the potential of LfuCostBoost, but it was not able to perform adequate throughout the experiments. The min-heap policies with different ageing approaches usually perform worse than using no ageing at all, which might be due to the ageing interval of every hundred items the cache evaluates after a warm-up time of 10000 evaluated items. Ageing is considered necessary for many Lfu implementations, to ensure no item has lifetime tickets in the cache. However, the computational costs of doing ageing quickly add up since it requires to iterate through the entire heap and access each element, which is an industry implementation would require a lock. Comparing the heap-based policies to the O(1) Lfu, we can see that the O(1) Lfu is capable of ageing by iterating the frequency nodes, which reduces the number of iterations needed to age if we can age simply by altering the frequency nodes, and thus the performance. The ageing approaches did not enhance hit rate or weighted hit rate for the majority of the experiments, with some exceptions. But in theory, the same approach as LfuCostBoost uses for incrementing the frequency of the items based on weight/cost should apply to ageing the items as well. One fundamental difference between the two approaches, in general, is that LfuCostBoost only increments on hits, while the ageing approach ages every eligible item every age interval. The performance of the heap-based ageing approaches did not perform up to par and was usually lagging behind LfuCostBoost, which is a less complex algorithm.

LfuCostBoost really performed well in the web_0 simulation where the combination without an admission policy is able to increase weighted hit rate by 1.65%, while hit rate regressed by 1.87% compared to Lfu. But for many of the other simulations, Lfu and LfuCostBoost roughly equal, which indicates that there is a need for some fine-tuning of the scale down the function of the cost. When paired with admission policies LfuCostBoost may outperform the same combination with Lfu, which the weighted web_0 simulation is an example of. The performance of the eviction policy LfuCostBoost was not convincing throughout the experiments, which may be due to many combinations of policies operating and making decisions based on completely different frequency data. Nonetheless, for the simulations that end up with different performance, the LfuCostBoost usually has a higher weighted hit rate, but slightly lower hit rate. When paired with the most high-performance admission policies, LfuCostBoost finds itself in capturing the same attributes in data access patterns as the admission policy, which should preferably be two orthogonal policies.

Even though the sampled policies was not the focus of the experiments, their performance can not be ignored. In most simulations, they came out very strong for both weighted caching and cost-based caching when paired with various admission policies. Even though the sampled are highly dependent on getting a good sample, we have seen that their approximations make for very performant policies. Factors that play a role when the sampled policy is doing caching is a good rollover strategy from sample to sample and a good sample selection algorithm. The sampled Hyperbolic policy, with its combined use of both frequency and age, provide a suitable and well-fitting model for total item lifetime cost. The impressive performance of the sampled policies, and especially the Hyperbolic eviction policy makes it a suitable foundation for further enhancement of sampled policies in weighted and cost-based caching. Furthermore, algorithms considered too expensive to work on a full cache could implement a sampled variant, which will reduce the overhead of using it.

Sampled policies combined with the TinyLfu based admission policies delivered high performance throughout the experiments. As with capturing data orthogonality, it might be best for either the admission policy or the eviction policy to evaluate and handle weight/cost. Or at least evaluate different aspects of the items in order to admit and evict the correct items to ensure both high hit rate and weighted hit rate.

For some NoSQL databases, having room for more items is more crucial than the most optimal cache replacement algorithm. In Redis, the Lfu implementation is done by having a frequency count and the last decrement time for each item, in 24 bits total. The 24 bits is not important, but knowing the performance implications of storing size or item retrieval costs to the items is why Redis in its Lfu implementation uses logarithmic counters. For Redis, Memcached and other high-performance key-value stores, storing item size or cost may be too costly. However, having an admission policy evaluate an estimated size upon inspection frees up the cache from storing anything extra, but it would require a victim lookup when deciding whether to admit an item or not. If Redis and Memcached were to extend the item to store size or cost as well they would have a very good foundation in utilizing and extending the Hyperbolic eviction policy. In the cases where key-value stores are solely used as caches, having an admission policy makes sense, as the clients are not aware of which are in the cache in a given time or not. This would be a massive performance boost for NoSQL key-value stores in general.

Both Apache Solr and ElasticSearch utilize caching to cache query results, which in their current form does not evaluate either size or query retrieval cost. Having an admission policy do frequency estimations of a newly computed query might not make sense, but for obvious performance reasons, the query result needs to be admitted into the cache. Apache Solr already uses Caffeine, which makes it no issue to use TinyLfuBoost or Secondary to control how expensive query results are handled. ElasticSearch, with their read, optimized Lru implementation, which is very close to being a Clock implementation would benefit from a performance standpoint to utilize an admission policy in their cache. It is hard to argue against an eviction policy specifically designed for ElasticSearch without knowing all the different use cases for the cache. Nonetheless, switching the Lru implementation to LfuCostBoost, or Lfu with cost specific ageing could help control how the cache replaces complex query results such as aggregate query results.

6.3 Contributions

The main contributions of this work are showing that simple enhancements to state-of-the-art cache replacement policies to utilize size and cost can lead to increased performance. The contributions are not that significant, but a step in the right direction for weighted and cost-based caching in the NoSQL database and full-text search engine landscape. Another contribution of this work is a preview of how historical access patterns could help enhance performance for adaptive cache replacement policies.

Chapter 7

Conclusion

7.1 Research questions

In this master's thesis, we have analyzed, implemented and experimented with cache replacement policies to improve performance of these policies in the context of NoSQL databases and full-text search engines by looking into the following research questions:

RQ1 How could weighted hit rate be enhanced without sacrificing hit rate in cache replacement policies?

RQ2 How could historical access patterns be utilized in adaptive cache replacement policies to enhance cache hit rate?

To address RQ1 firstly, based on the already performed literature review, we have chosen to enhance both well-established policies, as well as state-of-the-art policies which support weighted caching in the Caffeine simulation project. We have created replacement policies that were expected to perform poorly to display the effects of using a poorly designed replacement policy. We have implemented small adjustments to policies, that include incrementing frequency counts based on size/cost, ageing based on size/cost, secondary comparison of equal frequency estimates and combining frequency and size/cost into a total cost. The implementations have been simulated towards a large variety of datasets in both weighted caching, where the maximum size of the cache is limited by item size/weight, and cost-based caching, where the size of the items is ignored and rather evaluated as a retrieval cost of this item. The results show that small and simple modifications of the algorithms have the potential to improve performance. Some require sufficient knowledge about the data in order to enhance performance, while others do not. The TinyLfuBoost admission policy performing almost always equal to TinyLfu, which may be due to the short reset interval and small sketch (4-bit). We have also seen surprisingly good results from TinyLfu with the Secondary admission policy, which speaks for the limitation of TinyLfuBoost, where frequencies are not being built up due to the 4-bit sketch being used. We have learned when using the Secondary admission policy for weighted caching, it usually results in better cache performance if we prioritize items of smaller weight due to the weighted caching problem. For cost-based caching, flipping this operator leads to the cache being filled with items of higher cost. Nonetheless, we have seen the performance of the Secondary admission policy to surpass TinyLfu in many simulations. For the 4-bit sketch, it is no doubt that there will be many frequency estimations from the sketch that will end up with the same frequency. Thus the effect of the Secondary admission policy is significant and it will in many cases enhance both hit rate and weighted hit rate, especially in weighted caching. We have also seen the simple extension to Lfu, LfuCostBoost has one case of improved weighted hit rate, but in order to achieve that, the hit rate regressed. We have seen the ageing approaches have little to no impact on the performance of the cache replacement policy, which as for LfuCostBoost, means that in order to achieve higher performance, some fine-tuning of the weight/cost scaling is needed.

For RQ2, the experiment of utilizing historical access patterns revealed the potential and use cases in caching. We have learned that some adaptive policies, such as LeCaR with LRU and LFU, could purposely use the policy which is more scan resistant when a loopy workload occurs and it will help contribute to keeping the loss in performance to a minimum due to preventing cache pollution. The caching system would either need to have the capabilities of doing background processing itself or having another smart system contribute with adaptive suggestions. The system would complement the adaptive steps based on the recent past by either increasing the step length of the adaption if they suggest adapting in the same direction. If they disagree in the direction of the adaption, we must determine whether to trust the adaption based on recent past or based on the historical access patterns. Getting this wrong could be very costly and possibly lead to reduced performance, which is why we need to be sure about the data access pattern being consistent.

We were not able to show the true potential behind utilizing historical access patterns due to not having an automated smart system perform this internal adaption. We have also learned that until there are more high-quality datasets available, that do contain different access patterns, capturing patterns is a difficult task. Nonetheless, a caching system, which already adapts its internals based on the recent past can have its cache hit rate increased by evaluating historical access patterns. Adaptive policies are applicable for such an extension, and they could utilize historical access pattern to converge to an optimal internal distribution of sizes or probabilities faster. They could also, as shown in the experiment prevent cache pollution by optimizing the internal state to be more scan resistant.

7.2 Summary

For high-performance NoSQL key-value stores such as Redis and Memcached, having an admission policy would not make sense in some use cases, making the eviction policy their best bet.

The performance of the different combinations of policies and products for weighted and cost-based caching shows large variations in performance from policy to policy, especially when it comes to weighted caching. The performance of TinyLfu is impressive throughout the experiments, which makes it a suitable model and foundation for other admission policies that attempt to capture more features. This can both be seen in TinyLfuBoost and Secondary, which utilizes the highly performant base with their small enhancements. The TinyLfuBoost admission policy has shown its potential in the experiments, usually prevalent amongst the top-10 results of the simulations. However, it could lead to the Count-Min sketch providing even worse over-estimations, all based on what magnitude the weight/cost is, and how the scale down approach is. The over-estimation problem should be analyzed further, which is described in chapter 8.

The different ageing mechanisms in the min-heap based policies does theoretically have the same potential as ageing or periodic resets in the Count-Min sketch. However, having the ageing in the admission policy frees up the system to perform background tasks in the data structure holding the cache items. In other words, while the admission policy performs a periodic reset, the cache internal data structure could also do some processing of items or similar.

In this thesis, we have seen that small and very simple changes in state-of-the-art cache replacement policies lead to increased cache performance, both in terms of hit rate and weighted hit rate. The improvement of the weighted cache hit rate often comes with a small regression of the cache hit rate, which is most likely due to the strong correlation between the two. Nonetheless, some experiments were successful at showing that it is possible to keep the cache hit rate high as possible while increasing the weighted hit rate. We have also seen the potential behind making use of historical access patterns to optimize caching, even though for a fully automated solution, there is still work to be done.

Chapter 8

Future work

Caching for NoSQL databases and full-text search engines will become even more important for maintaining high performance in the future. Having fully functioning caches for a variety of use cases will help the systems maintain high performance while being able to control the content of their caches in terms of variable size and retrieval cost.

Data

The most crucial step for further enhancing cache replacement policies is to get access to high-quality traces of different systems. Most traces available and used in research today are I/O traces, which works for simulation purposes, but it is not optimal for every NoSQL database and full-text search engine because they operate in very different environments where the data, data access patterns and weight/cost will vary. This issue is related to both weighted/cost-based caching and capturing historical access patterns.

Count-min sketch bias

The Count-min sketch bias could quickly become an issue for the TinyLfuBoost admission policy, which is why solutions to the issue need to be investigated. Both Deng *et al.* [61] and Goyal *et al.* [62] evaluates different sketch algorithms and comes with recommendations for lowering the over-estimation error in the Count-Min sketch algorithms, which should be investigated if TinyLfuBoost is to be further experimented upon.

Admission

For admission policies in general, for weighted caching the future work consists of creating admission policies that evaluate the set of items needed for eviction if a candidate is to be admitted.

Eviction

The Hyperbolic sampled eviction policy has shown itself very performant throughout the experiments which makes it a suitable foundation for extending it with item size or item retrieval cost.

LeCaR and its model of the cache replacement problem could also be tweaked in order to adapt better when faced with weighted items. LeCaR already uses graded regret based on how long the item has been prevalent in the ghost list. This idea could be extended to grade the regret based on weight or retrieval cost.

Caching in general

In a discussion with authors of state-of-the-art cache replacement policies, predicting workload changes came up repeatedly. More specifically, as with RQ2, using historical access patterns to predict workload changes could be an approach worth investigating, especially for larger deployments where the background processing of continuously identifying trends is computationally feasible. Other issues within the caching landscape that needs further attention in the future.

- Prediction of workload changes.
- Multi-tenant capacity allocation [63].
- Optimal cache size prediction [64].

Bibliography

[1] H. Langdal, "Cache replacement policies in NoSQL databases and full-text search engines", Norwegian University of Science and Technology, Trondheim, Norway, Project report, 2019.

[2] P. J. Sadalage and M. Fowler, *NoSQL distilled: a brief guide to the emerging world of polyglot persistence.* Upper Saddle River, NJ: Addison-Wesley, 2013, 164 pp., ISBN: 978-0-321-82662-6.

[3] D. E. A. Brewer, "Towards robust distributed systems", 2000.

[4] S. Sanfilippo. (2010). Redis. https://github.com/antirez/redis-io, [Online]. Available: `https://redis.io/documentation` (visited on 10/11/2019).

[5] Apple, N. Lavezzo, D. Rosenthal, and D. Scherer. (2013). FoundationDB 6.2 — FoundationDB 6.2, [Online]. Available: `https://apple.github.io/foundationdb/` (visited on 11/05/2019).

[6] S. Sanfilippo, *Antirez/redis*, original-date: 2009-03-21T22:32:25Z, Nov. 7, 2019. [Online]. Available: `https://github.com/antirez/redis` (visited on 11/07/2019).

[7] B. Fitzpatrick, "Distributed caching with memcached", *Linux Journal*, vol. 2004, no. 124, p. 5, Aug. 1, 2004, ISSN: 1075-3583.

[8] memcached and D. Interactive, *Memcached/memcached*, original-date: 2009-04-24T23:34:25Z, 2009. [Online]. Available: `https://github.com/memcached/memcached` (visited on 11/07/2019).

[9] E. NV, *Elastic/elasticsearch*, original-date: 2010-02-08T13:20:56Z, 2010. [Online]. Available: `https://github.com/elastic/elasticsearch` (visited on 11/12/2019).

[10] A. S. Foundation, *Apache/lucene-solr*, original-date: 2016-01-23T08:00:06Z, Nov. 12, 2019. [Online]. Available: `https://github.com/apache/lucene-solr` (visited on 11/12/2019).

[11] P. J. Denning, "Virtual memory", *ACM Computing Surveys (CSUR)*, vol. 2, no. 3, pp. 153–189, 1970, Publisher: ACM New York, NY, USA.

[12] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement", *Journal of the ACM (JACM)*, vol. 18, no. 1, pp. 80–93, 1971, Publisher: ACM New York, NY, USA.

[13] F. J. Corbato, "A paging experiment with the multics system", MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[14] S. Mittal, "A survey of cache bypassing techniques", *Journal of Low Power Electronics and Applications*, vol. 6, no. 2, p. 5, 2016.

[15] A. J. Smith, "Cache memories", *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[16] M. Rabinovich and O. Spatscheck, *Web caching and replication.* Addison-Wesley Boston, USA, 2002, vol. 67.

[17] A. J. Smith, "Disk cache-miss ratio analysis and design considerations", *ACM Transactions on Computer Systems*, vol. 3, no. 3, p. 43, 1985.

[18] J. Erman, A. Gerber, M. T. Hajiaghayi, D. Pei, and O. Spatscheck, "Network-aware forward caching", in *Proceedings of the 18th international conference on World wide web - WWW '09*, Madrid, Spain: ACM Press, 2009, p. 291, ISBN: 978-1-60558-487-4. DOI: `10.1145/1526709.1526749`. [Online]. Available: `http://portal.acm.org/citation.cfm?doid=1526709.1526749` (visited on 11/06/2019).

[19] S. Paul and Z. Fei, "Distributed caching with centralized control", *Computer Communications*, vol. 24, no. 2, pp. 256–268, Feb. 2001, ISSN: 01403664. DOI: `10.1016/S0140-3664(00)00322-4`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0140366400003224` (visited on 11/06/2019).

[20] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems", *Algorithmica*, vol. 1, no. 1, pp. 311–336, Nov. 1986, ISSN: 0178-4617, 1432-0541. DOI: `10.1007/BF01840450`. [Online]. Available: `http://link.springer.com/10.1007/BF01840450` (visited on 10/11/2019).

[21] P. Norvig, "Technical correspondence techniques for automatic memoization with applications to context-free parsing", *Computational Linguistics*, vol. 17, no. 1, p. 9, 1991.

[22] J. Mertz and I. Nunes, "Understanding application-level caching in web applications: A comprehensive introduction and survey of state-of-the-art approaches", *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, p. 98, 2018.

[23] B. Manes, *Ben-manes/caffeine*, original-date: 2014-12-13T08:45:11Z, 2014. [Online]. Available: `https://github.com/ben-manes/caffeine` (visited on 10/21/2019).

[24] P. J. Denning, "The profession of IT the locality principle", *COMMUNICATIONS OF THE ACM*, vol. 48, no. 7, p. 7, 2006.

[25] P. J. Denning and S. C. Schwartz, "Properties of the working-set model", vol. 15, no. 3, p. 8, 1972.

[26] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)", *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, p. 60, Jun. 19, 2010, ISSN: 01635964. DOI: `10.1145/1816038.1815971`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=1816038.1815971` (visited on 11/05/2019).

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching", *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, p. 381, Jun. 9, 2007, ISSN: 01635964. DOI: `10.1145/1273440.1250709`. [Online]. Available: `http://portal.acm.org/citation.cfm?doid=1273440.1250709` (visited on 11/05/2019).

[28] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A highly efficient cache admission policy", *arXiv:1512.00727 [cs]*, Dec. 2, 2015. arXiv: `1512.00727`. [Online]. Available: `http://arxiv.org/abs/1512.00727` (visited on 09/25/2019).

[29] (). DB-engines ranking, DB-Engines, [Online]. Available: `https://db-engines.com/en/ranking/key-value+store` (visited on 10/08/2019).

[30] E. NV. (2019). Shard request cache | elasticsearch reference [7.4] | elastic, [Online]. Available: `https://www.elastic.co/guide/en/elasticsearch/reference/current/shard-request-cache.html` (visited on 11/30/2019).

[31] P. Flajolet, "Approximate counting: A detailed analysis", *BIT*, vol. 25, no. 1, pp. 113–134, Mar. 1985, ISSN: 0006-3835, 1572-9125. DOI: `10.1007/BF01934993`. [Online]. Available: `http://link.springer.com/10.1007/BF01934993` (visited on 10/17/2019).

[32] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications", *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, Apr. 2005, ISSN: 01966774. DOI: `10.1016/j.jalgor.2003.12.001`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0196677403001913` (visited on 10/17/2019).

[33] K. Shah, A. Mitra, and D. Matani, "An o(1) algorithm for implementing the LFU cache eviction scheme", p. 8, 2010.

[34] S. Sanfilippo. (Aug. 28, 2016). Random notes on improving the redis LRU algorithm - <antirez>, [Online]. Available: `http://antirez.com/news/109` (visited on 10/11/2019).

[35] R. Karedla, J. Love, and B. Wherry, "Caching strategies to improve disk system performance", *Computer*, vol. 27, no. 3, pp. 38–46, Mar. 1994, ISSN: 0018-9162, 1558-0814. DOI: `10.1109/2.268884`.

[36] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance", *ACM SIGMETRICS Performance Evaluation Review*, vol. 30, no. 1, pp. 31–42, 2002.

[37] G. Zamarreño, "Keeping infinispan in shape: Highly-precise, scalable data eviction", Technology, [Online]. Available: `https://www.slideshare.net/galderz/pse-jud-con` (visited on 11/30/2019).

[38] S. Jiang, F. Chen, and X. Zhang, "CLOCK-pro: An effective improvement of the CLOCK replacement", p. 14, 2005.

[39] C. Li, "CLOCK-pro+: Improving CLOCK-pro cache replacement with utility-driven adaptation", in *Proceedings of the 12th ACM International Conference on Systems and Storage*, Haifa Israel: ACM, May 22, 2019, pp. 1–7, ISBN: 978-1-4503-6749-3. DOI: `10.1145/3319647.3325838`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3319647.3325838` (visited on 05/04/2020).

[40] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of facebook photo caching", in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*, Farminton, Pennsylvania: ACM Press, 2013, pp. 167–181, ISBN: 978-1-4503-2388-8. DOI: `10.1145/2517349.2522722`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=2517349.2522722` (visited on 11/06/2019).

[41] K. Psounis and B. Prabhakar, “Efficient randomized web-cache replacement schemes using samples from past eviction times”, *IEEE/ACM transactions on networking*, vol. 10, no. 4, pp. 441–454, 2002, Publisher: IEEE.

[42] A. Blankstein, S. Sen, and M. J. Freedman, “Hyperbolic caching: Flexible caching for web applications”, p. 15, 2017.

[43] N. Megiddo and D. S. Modha, “Outperforming LRU with an adaptive replacement cache algorithm”, *Computer*, vol. 37, no. 4, pp. 58–65, Apr. 2004. DOI: `10.1109/MC.2004.1297303`.

[44] C. Li, “DLIRS: Improving low inter-reference recency set cache replacement policy with dynamics”, in *Proceedings of the 11th ACM International Systems and Storage Conference on - SYSTOR ’18*, Haifa, Israel: ACM Press, 2018, pp. 59–64, ISBN: 978-1-4503-5849-1. DOI: `10.1145/3211890.3211891`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=3211890.3211891` (visited on 11/29/2019).

[45] S. Bansal and D. S. Modha, “CAR: Clock with adaptive replacement.”, in *FAST*, vol. 4, 2004, pp. 187–200.

[46] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, “Driving cache replacement with ML-based LeCaR”, p. 6, 2018.

[47] G. Einziger, O. Eytan, R. Friedman, and B. Manes, “Adaptive software cache management”, in *Proceedings of the 19th International Middleware Conference*, ACM, 2018, pp. 94–106.

[48] A. Kasindorf. (Oct. 15, 2018). Memcached - a distributed memory object caching system, Replacing the cache replacement algorithm in memcached - Dormando (October 15, 2018), [Online]. Available: `https://memcached.org/blog/modern-lru/` (visited on 10/03/2019).

[49] *Cache2k/cache2k*, original-date: 2013-12-18T17:25:04Z, Mar. 2, 2020. [Online]. Available: `https://github.com/cache2k/cache2k` (visited on 03/05/2020).

[50] D. S. Berger, R. Sitaraman, and M. Harchol-Balter, “AdaptSize: Orchestrating the hot object memory cache in a content delivery network”, p. 17, 2017.

[51] V. Bychkovsky, J. Cipar, A. Wen, L. Hu, and S. Mohapatra. (Jun. 28, 2018). Spiral: Self-tuning services via real-time machine learning, Facebook Engineering. Library Catalog: engineering.fb.com Section: AI Research, [Online]. Available: `https://engineering.fb.com/data-infrastructure/spiral-self-tuning-services-via-real-time-machine-learning/` (visited on 03/03/2020).

[52] J. Barr. (Nov. 20, 2018). New – predictive scaling for EC2, powered by machine learning, Amazon Web Services. Library Catalog: aws.amazon.com Section: Amazon EC2, [Online]. Available: `https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/` (visited on 03/03/2020).

[53] H. Langdal, *Havlan/caffeine*, original-date: 2020-01-14T11:40:07Z, Jun. 9, 2020. [Online]. Available: `https://github.com/havlan/caffeine` (visited on 06/09/2020).

[54] A. S. Foundation. (). Apache HBase – apache HBase™ home, Apache HBase, [Online]. Available: `https://hbase.apache.org/` (visited on 11/05/2019).

[55] ——, *Apache/accumulo*, original-date: 2011-10-06T07:00:09Z, Mar. 20, 2020. [Online]. Available: `https://github.com/apache/accumulo` (visited on 03/20/2020).

[56] Akka. (2020). Akka: Build concurrent, distributed, and resilient message-driven applications for java and scala | akka, [Online]. Available: `https://akka.io/` (visited on 03/31/2020).

[57] S. Vigna, *Vigna/fastutil*, original-date: 2015-05-11T16:59:13Z, Mar. 20, 2020. [Online]. Available: `https://github.com/vigna/fastutil` (visited on 03/20/2020).

[58] S. Sanfilippo. (2019). Using redis as an LRU cache – redis, [Online]. Available: `https://redis.io/topics/lru-cache` (visited on 10/11/2019).

[59] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage", *ACM Transactions on Storage (TOS)*, vol. 4, no. 3, pp. 1–23, 2008, Publisher: ACM New York, NY, USA.

[60] U. of Massachusetts Amherst. (). Storage - UMass trace repository, [Online]. Available: `http://traces.cs.umass.edu/index.php/Storage/Storage.` (visited on 10/23/2019).

[61] F. Deng and D. Rafiei, "New estimation algorithms for streaming data: Count-min can do more", p. 12, 2007.

[62] A. Goyal, H. D. Iii, and G. Cormode, "Sketch algorithms for estimating point queries in NLP", p. 11, 2012.

[63] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers", in *Proceedings of the Sixth ACM Symposium on Cloud Computing - SoCC '15*, Kohala Coast, Hawaii: ACM Press, 2015, pp. 174–181, ISBN: 978-1-4503-3651-2. DOI: `10.1145/2806777.2806933`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=2806777.2806933` (visited on 05/29/2020).

[64] C. A. Waldspurger, T. Saemundson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations", p. 13, 2017.