Technical University Of Crete

School of Electrical and Computer Engineering

Diploma Thesis

A Caching Platform for Large Scale Data-Intensive Distributed Applications

A thesis submitted in fulfillment of the requirements for the degree of Diploma in Electrical and Computer Engineering

Author:
Nikolaos Kafritsas

Thesis Committee:
Professor Minos Garofalakis (Supervisor)
Associate Professor Vasilis Samoladas
Associate Professor Antonios Deligiannakis

Chania, August 2019

"Artificial Intelligence is the new electricity"

Andrew Ng

Abstract

In the last decade, data processing systems started using main memory as much as possible, in order to speed up computations and boost performance. Towards this direction, many breakthroughs were created in the stream processing systems, which must meet rigorous demands and achieve sub-second latency along with high throughput. These advancements were feasible due to the availability of large amounts of DRAM at a plummeting cost and the rapid evolution of in-memory databases. However, in the Big Data era, maintaining such a huge amount of data in memory is impossible. On the other hand, the use of disk-based databases to remedy the situation is prohibitively expensive in terms of disk latencies. The ideal scenario would be to have the high access speed of memory, with the large capacity and low price of disk. This hinges on the ability to effectively utilize both the main memory and disk. Consequently, developing a solution which somehow combines the benefits of both worlds is highly desirable.

This diploma thesis tackles the aforementioned problem by proposing an alternative architecture. More specifically, hot data are stored in memory, while cold data are moved to disk in a transactionally-safe manner as the database grows in size. Because data initially reside in memory, this architecture reverses the traditional storage hierarchy of disk-based systems. The disk is treated as an extended storage for evicted elements/cold data, not the primary host for the whole data. Based on this architecture, a multi-layered platform is presented which is highly scalable and can work in a distributed manner. The memory layer acts as a cache with configurable capacity and provides several eviction policies, the most important being a variation of the traditional LFU eviction policy. In particular, data regarded as cold could return back to memory if it becomes hot again, a case that occurs when the distribution of data changes in online processing. Thanks to this feature and the sub-second latency that is achieved, the platform can also perform efficiently in a streaming environment and can be used as a stateful memory component in a real-time architecture. The disk layer is flexible and elastic, meaning that users can use the database of their choice as a disk-based storage for cold data. Finally, the platform is tested in different scenarios under heavy load, and the benchmarks showed that it can perform extremely well and achieve throughput in the order of thousands of elements per second.

Acknowledgements

First of all, I would like to express my sincere gratitude to my advisor, professor Minos Garofalakis, for initiating me to the world of large databases, motivating and supervising me. I am also thankful to Antonios Deligiannakis and Vasilis Samoladas for their fruitful comments and the knowledge that I acquired from their courses during my studies.

Next, I am deeply grateful and I acknowledge the time that doctor Odysseas Papapetrou spent, in order to guide me throughout this whole process, as well as for his thoughtful and detailed comments.

Furthermore, I am grateful to my family, Dimitris, Vasso and Marialena, for constantly supporting me in an unconditional way, and not holding grudges when I was abrupt towards them over the phone in times of stress.

Last but not least, I would like to thank my dear Maria, for always being there for me, constantly providing inspiration, support and reassurance during the time I was writing this thesis.

Contents

Abstract ii

Acknowledgements iii

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Thesis Motivation . . . 1
- 1.2 Thesis Contribution . . . 4
- 1.3 Thesis Outline . . . 5

2 Problem Statement 6
- 2.1 Technical Challenges . . . 6
- 2.2 Problem Formulation . . . 7
- 2.3 Alternative approaches . . . 8
  - 2.3.1 Backpressure . . . 8
  - 2.3.2 Micro-Batching . . . 9
  - 2.3.3 Approximation . . . 9
  - 2.3.4 Faster Storage . . . 10
- 2.4 Alternative Approach . . . 11

3 Background and Related Work 13
- 3.1 The Concept of Caching . . . 13
  - 3.1.1 How Caching Helps . . . 14
    - 3.1.1.1 Speeding up CPU-Bound Applications . . . 14
    - 3.1.1.2 Speeding up I/O-Bound Applications . . . 15
    - 3.1.1.3 Characterization of this Platform . . . 16
  - 3.1.2 Expected Speedup . . . 16
  - 3.1.3 Eviction Strategies . . . 17
  - 3.1.4 Caches and Machine Learning . . . 20
  - 3.1.5 Writing Strategies . . . 20
    - 3.1.5.1 Write-Through Cache . . . 20
    - 3.1.5.2 Write-Around Cache . . . 21
    - 3.1.5.3 Write-Back Cache . . . 21
- 3.2 Caching Challenges . . . 21

3.3 Databases and Key-Value Stores . . . 22
3.3.1 Redis . . . 23
3.3.2 Apache Ignite . . . 23
3.3.3 Ehcache . . . 24
3.3.4 MapDB . . . 25
3.3.5 Conclusion . . . 26

4 Implementation **27**
4.1 Layers of the Platform . . . 27
4.1.1 First Layer . . . 27
4.1.2 Second Layer . . . 27
4.1.3 Third Layer . . . 29
4.2 Additional Specifications . . . 29
4.2.1 Modularity and Flexibility . . . 30
4.2.2 Scalability . . . 31
4.2.3 Generic Values . . . 32
4.3 Differences with a Traditional Caching System . . . 33
4.4 Platform Architecture . . . 35
4.4.1 Bloom Filter . . . 35
4.4.1.1 Overview . . . 35
4.4.1.2 Implementation Details . . . 36
4.4.1.3 Technical Details . . . 36
4.4.1.4 How the Bloom Filter is Parameterized . . . 37
4.4.2 Memory Layer . . . 38
4.4.2.1 LRU . . . 39
4.4.2.2 MRU . . . 41
4.4.2.3 LFU . . . 43
4.4.3 Persistence Layer . . . 49
4.4.3.1 DAO Pattern . . . 50
4.4.3.2 Abstract Factory Pattern . . . 52
4.4.3.3 Connection Pooling . . . 55

5 Performance Evaluation **57**
5.1 Experimental Setup . . . 57
5.2 The Naive Approach . . . 59
5.3 Second Version . . . 59
5.4 Cache Hit Ratio . . . 61
5.5 Comparison with Naive Approach . . . 63
5.6 Large Scale Experiments . . . 64
5.7 Memory Management . . . 65

6 Conclusion **68**
6.1 Closing Remarks . . . 68
6.2 Future Work . . . 70

Bibliography **71**

List of Figures

1.1 The concept of Structured Streaming in Apache Spark 3
1.2 Example of handling late data in a stream, where event time is different from ingestion timevery important for handling time-sensitive data e.g. sensor data 4

4.1 The overall architecture of the system. Each layer is depicted with a different color 28
4.2 The stream is partitioned and each itemset is assigned to a particular instance 31
4.3 The system can be used inside each bolt, where Apache Storm's processing logic takes place 32
4.4 Left: A traditional caching system, Right: The implementation of this thesis 34
4.5 The UML class depicting the DAO Pattern 50
4.6 UML class about Abstract Factory. Note that the specific details of how the database is managed (e.g. connection credentials) is implemented in the Factory class of each database independently 54
4.7 The Connection Pool Module 55

5.1 Cache hit ratios with (1) get requests, (2) update requests, (3) random requests 62
5.2 Total time for each version when 75% of elements are in memory 63
5.3 Cache hit ratios with (1) one million distinct elements, (2) two million distinct elements 65
5.4 VisualVM monitors CPU and memory usage, among other things 66

List of Tables

5.1 System configuration for all experiments 57
5.2 Experimental results of the first version 59
5.3 Experimental results of the second version using zipfian distribution with s=1 60
5.4 Experimental results of the second version using zipfian distribution with s=2 60
5.5 Experimental results of the second version using uniform distribution . . . 61
5.6 Experimental results on a larger scale 64

List of Algorithms

1 LRU – Put operation 39
2 LRU – Get operation 40
3 LRU – Eviction 41
4 MRU – Put operation 41
5 MRU – Get operation 42
6 MRU – Eviction 42
7 LFU – Put operation 45
8 LFU – Get operation 48

Chapter 1

Introduction

1.1 Thesis Motivation

With the advent of stream-processing frameworks, a lot of software architectures shifted towards real-time processing. Initially, most of ETL operations were re-implemented to accommodate for a real or near-real time processing architecture. Many algorithms required additional memory to store summary statistics, temporary results and so on. These are called stateful architectures. At a high level, state in stream processing can be considered as memory in operators or data structures that remembers information about past input and can be used to influence the processing of future input. Some frameworks like *Samza* and *Apache Flink* have APIs for providing stateful functionalities since their inception, while *Apache Storm* introduced its stateful functionality subsequently in version 1.0.0. In the past few years, stateful deployments have surpassed stateless ones in streaming processors [1]

As the time was passing by, the demand for implementing more complex algorithms necessitated the deployment of more sophisticated systems. More complex streaming pipelines generally need to cache in memory some sort of operator state in order to execute the application logic. Examples include keeping some aggregation or summary of the received elements. There are also more complex states such as keeping a state-machine for detecting patterns for fraudulent financial transactions or holding a model for some machine learning application. While in all mentioned cases some kind of summary of the input history is kept, the concrete requirements vary greatly from one stateful application to another.

Another common practice is to decouple the role of the state management from the stream processor. The most obvious way to do this is to use an external database

system or a caching platform where the state will be stored. This can be something as simple as a key-value store or more demanding like a NoSQL database. However, the performance of disk-based databases, slowed down by unpredictable and high access latency of disks, is no longer acceptable in meeting the rigorous low-latency, real time demands of Big Data. The performance issue is further exacerbated by the overhead (e.g., system calls, buffer manager) hidden by the I/O flow. To meet the strict real time requirements for analyzing a massive amount of data and servicing requests within milliseconds, an in-memory database that keeps data in the main memory all the time is a promising alternative. In the past few years, the availability of large amounts of DRAM at a lower cost than before helped to create new breakthroughs, making it possible to deploy in-memory databases where a significant part, if not the entirety, of data fits in main memory.

Over time, the database community realized that the most efficient way to boost performance and speed is to store data in memory. If data reside in memory, all computations would be faster, since the flow and the transformation of data is faster. This signified the transition of the *Hadoop-based* era to the *Spark-based* era and the introduction of *Lamda architecture* in 2011 by Nathan Marz. However, all big data processing frameworks show the tendency to integrate their batch and stream processing APIs in the same execution engine. Two of the most state-of-the art open source frameworks, *Apache Flink* and *Apache Spark* started abandoning the notion of *Tuple* and *RDD* respectively and focus on the more advanced, *DataStream* and *DataFrame* APIs. In this paradigm, the stream is not regarded as a collection of tuples, but as a Streaming Database, which changes over time (Figure 1.1). In order to get at this point, the data processing platforms achieved 2 goals: Exactly once processing in true stream processing and handling of late data in the stream (also called Structured Streaming, Figure 1.2). Spark announced on March 2018 that its new version 2.4 will introduce the new *Streaming API V2*, a new execution engine that can execute streaming queries with sub-millisecond end-to-end latency. It is clear that in the future batch and stream processing will unify under a single execution engine. This would also mark the evolution of *Lamda* architecture, which wanted the batch and speed layers in separate configurations executed by different APIs. The downside of this technique is that big data systems are increasingly relying on having large amounts of memory available in order to achieve fast processing speeds and accommodate a real time processing architecture. This entails moving, shuffling and transforming data in memory where all of the above actions can be performed extremely fast at DRAM processing speeds.

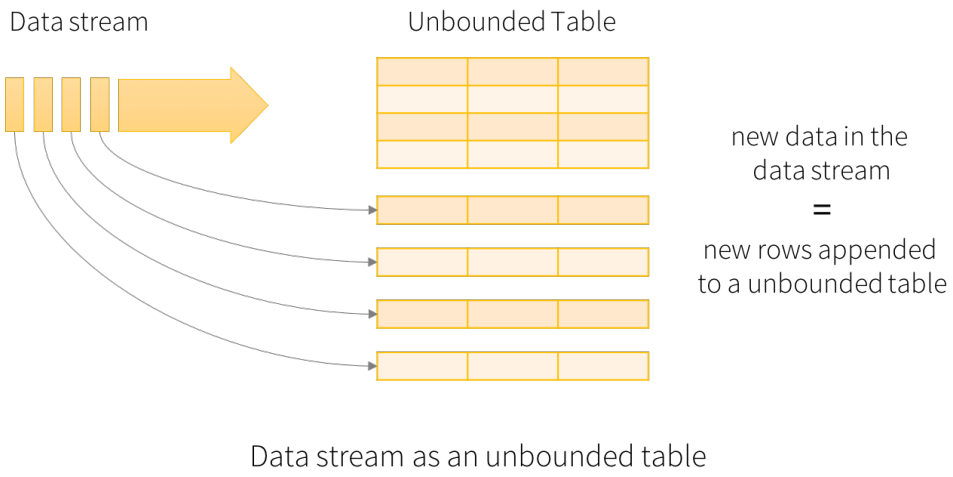

FIGURE 1.1: *The concept of Structured Streaming in Apache Spark*

A completely in-memory data management system presents many challenges [2], [3] and it is not only prohibitively expensive, but also in some cases impossible. Memory is not unlimited, and therefore expecting all of the data to fit is counter-intuitive. Inevitably, a disk-based database would be used as well to store additional data which do not fit in memory. However, in the context of a streaming application which requires a rigorous threshold in terms of latency and throughput, this is not always possible. Stream processors are usually assumed to consume unbounded streams which contain data with unknown distributions and they are expected to stay fully functional even after random spikes in traffic. The parallel use of a disk-based database would likely introduce unexpected disk latencies which cannot be ignored. Therefore, it is difficult to estimate how fast the data will be processed and how much pressure will be put on the streaming application. The next chapter discusses a hypothetical situation and demonstrates how this use-case could be a problem.

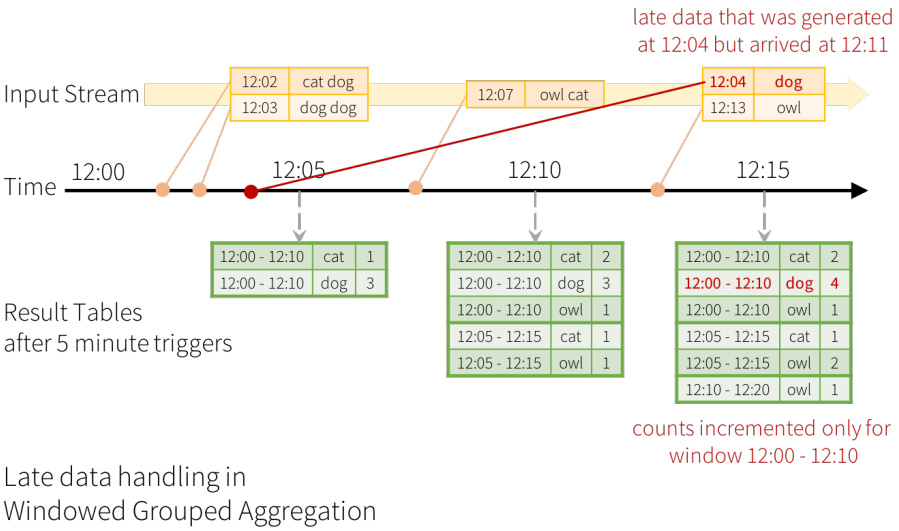

FIGURE 1.2: *Example of handling late data in a stream, where event time is different from ingestion timevery important for handling time-sensitive data e.g. sensor data*

1.2 Thesis Contribution

This thesis tackles the problem of limited memory by proposing a multi-layered hybrid platform which acts as a caching system and is accompanied with secondary data storage. The main functionality of the caching component is to store hot elements and evict them to disk according to an eviction policy. Therefore, the disk acts as a cold storage for evicted data. Apart form that, cold data can return from disk back to memory if they become hot again. This mechanism is referred to throughout this thesis as *restoration*. Therefore, eviction and *restoration* work interchangeably and are responsible for keeping hot elements in memory at all times, in an application-agnostic way. To achieve this functionality, this thesis proposes three novel eviction strategies which are variations of the LRU, MRU and LFU policies. They are similar to the traditional policies, however, they have been configured to support both eviction and *restoration*. Using a small amount of memory to keep only the hot elements cached at all times is very beneficial for streaming applications, which require extremely low latency and thus suffer from expensive I/Os to the disk. To our knowledge, there is no similar system which provides this functionality. There are either a) in-memory databases which provide optional backup to disk and b) hybrid systems which evict cold elements to the disk, but they don't examine the possibility of returning them back to memory if they become hot again. This platform can work in a standalone mode or a distributed one, and can also be used as an embedded stateful component in popular streaming frameworks. Finally,

the thesis discusses numerous optimizations and additional features such as database interchangeability (the users can use the database of their choice as the default data storage) and the use of bloom filter, which enhance performance and versatility.

1.3 Thesis Outline

This thesis is divided into multiple Chapters and there are 6 in total. Chapter 2 provides a detailed formulation of the problem that is addressed in this thesis, along with the relevant technical challenges. Next, Chapter 3 provides useful background related to concepts used throughout this thesis. Chapter 4 presents the main contribution in detail and describes in depth the platform that was implemented in order to solve the problems that were posed in Chapter 2. Chapter 5 presents a detailed performance evaluation. Finally, Chapter 6 discusses the thesis in general and outlines its possible expansions through future work.

Chapter 2

Problem Statement

This Chapter presents the problem that was addressed in this thesis, and argues why it cannot be solved by off-the-shelf solutions.

2.1 Technical Challenges

Suppose there is a distributed streaming processor which is 'assigned' to an unbounded stream and continually ingests new data. The data must be processed in real time, which means that the system is working on a single data point at a time and the goal is to achieve sub-second-level latency between the data being created and the results being available (this assumption excludes the micro-batching use case). In other words, the system must have a throughput in the order of more than thousands of datapoints per second and latency of a few milliseconds. Furthermore, assume that the particular system computes an algorithm which saves or reads a complex state as an intermediate result in an external storage system before fully processing and delivering the desired results. Unfortunately, in this case the process of reading and writing the complex state from storage is very costly. This is a logical assumption because it is not possible to expect the enormous throughput of datapoints which are flowing throughout the various stages of the streaming processor to match the speed of I/O operations. On top of that, things become worse if the algorithm requires the I/Os to take place one-at-a time, and not in batches, which they are much faster.

As a consequence, this use case looks like an impossible scenario, because it is absurd to compare the speed of in-memory operations with I/Os. After the algorithm is deployed on the stream processor, the system would not work properly, because the intermediate datapoints would not have been fully processed and ready to be delivered at the output

by the time the new datapoints start arriving. The latency of the I/Os would surpass all other costs throughout the system. Eventually, the connection between the stream processor and the external storage system becomes the bottleneck. The internal buffers of the stream processor which hold the processed datapoints at various nodes across the cluster would overflow because they would receive more tuples than they are able to send (in other words, their holding capacity would be exceeded). After a while, the stream would start lagging behind. The system would either crash or start dropping datapoints/tuples in order to flush its buffers, an action that is very likely to break the algorithm.It is worth mentioning that this problem does not affect stateless architectures, which don't require to store intermediate or temporary results based on previous computations. It only concerns stateful proccessing, where the state cannot fit in main memory. The following sections discuss some common workarounds to this problem and how these affect the behavior of the system in terms of validity, robustness and easiness of implementing them.

2.2 Problem Formulation

The problem can be summarized as follows:

How can a stateful stream processor, with a limited amount of memory, store a large state (which doesn't fit in memory) and simultaneously avoid the disk storage because the real-time processing will suffer from expensive writes?

Looking more closely, there are three main challenges:

- All streaming frameworks which perform real-time processing (sub-second latency) should be able to process large amounts of data extremely fast - at DRAM processing speeds.
- Memory is finite, and therefore in some cases it would be impossible to store all data in memory.
- However, if the disk is used to store at least some portion of it, the expensive disk lookups will hurt the performance and negate the purpose of real-time proccessing.

2.3 Alternative approaches

This section discusses some common workarounds to this problem and how these affect the behavior of the system in terms of validity, robustness and easiness of implementing them.

2.3.1 Backpressure

The most obvious solution is somehow to throttle the throughput of the input stream in order to match the slower speed of the I/Os. While this technique can be done manually, many stream frameworks expose this functionality as, what it is called, *automatic backpressure*. By definition, automatic backpressure enables the stream processor to consume new data depending on its capacity. If it is full and cannot accommodate for new datapoints, then the consumers 'choke' the input stream, and therefore the throughput is lowered. Some systems like Twitter Heron and Apache Flink use more advanced and sophisticated mechanisms to implement automatic backpressure, while other systems use simpler and more naïve techniques to provide that functionality. Either way, backpressure is a fundamental part of all stream processing systems, and that is why it is present in some form in most of them. Unfortunately, this workaround is not a viable solution. First of all, even if the stream will not probably be lagging anymore, the throughput would have been lowered, leading to a non-optimal architecture. To put it differently, if a stream can support a much higher value of throughput, and for whatever reason it is lowered in order to accommodate the speed of the slower components, then this architecture defeats the purpose of high scalability. The overall end-to-end latency of the system is defined by the latency of the slowest part, no matter how fast and scalable the other components are. This is analogous to having a cluster with 3 nodes, where one of them is slower than the others.In this case, the latency of the slowest node dominates the overall latency, no matter how fast the other nodes are. Finally, even if backpressure is used to cope with this problem, it is widely known that is not supposed to be used in this way. Backpressure was introduced as a temporal solution when the stream processor was experiencing random and short spikes of traffic. Also, it is used when a node in the cluster had failed, and for a limited amount of time, some data had to redirected to another node, thus putting some temporary pressure on the system, until the faulty one is fixed and comes back online. It is not meant to be used forever, as a way to throttle down the stream indefinitely and prevent it from crashing. That's why it is impossible to use backpressure as a fix to the situation that was described above.

2.3.2 Micro-Batching

Another solution would be to use micro-batching instead of pure stream processing. Micro-batching does not impose so much strict requirements regarding time, because this architecture can accept a latency of a few seconds or more (depends on the batch size and the window time). While micro-batching can also be considered a subcategory of stream processing, there are some notable differences. First, they are unsuitable for working on single-data-point-at-a-time kinds of problems. Secondly, this approach implies giving up the associated super-low latency in processing one data point at a time. If a user can surmount these limitations, then this is an acceptable solution. Nevertheless, this thesis discusses real time processing and focuses on extremely high throughput and sub-second-level latency, and therefore this scenario as well does not provide a viable solution.

2.3.3 Approximation

Moreover, a possible workaround for the above problem would be to use a probabilistic data structure which somehow approximates the results accurately enough with a small margin of error. The data processing literature has some very popular and efficient algorithms that can do this job such as *Space-Saving* and *Lossy counting* [4] [5], as well as *sketches* [6], which cover a vast variety of use cases. One of the most significant advantages of those algorithms is that they require sublinear space, at most poly-logarithmic in n to work. Therefore, it is not necessary to use an external data storage system because the main memory will probably be adequate. Nonetheless, their disadvantage is that they don't provide an exact answer. While this drawback may not be problem in some use cases, they cannot be used when the system handles sensitive data, such as financial data. Furthermore, if the output of the stream processor is fed into a software which builds a machine learning model using those data, then it is counter-intuitive to use approximate results because that would hurt the model during the training phase. In other words, the system would hold an approximation of the data in memory (e.g. summary statistics, sketches, histograms) in order to conserve space, which means that the results at the output would be within a margin of error. This thesis however does not completely reject the idea of using a probabilistic data structure. The next Chapter will discuss the possibility of using one that on the one hand it does not hurt the validity and the correctness of the results, and on the other hand it boosts performance.

2.3.4 Faster Storage

Finally, since the external storage system is essentially the bottleneck, there is a way to minimize the cost of the I/Os and reduce latency by using SSDs instead of HDDs. SSDs are much faster than a traditional hard disk, but slower than traditional DRAM nevertheless. A typical DRAM has a transfer rate of approximately 2-25GB/s, whereas typical SSDs have a transfer rate of 500MB/s. At the time of writing this thesis, the maximum threshold that the fastest SSDs can reach caps at 560 MB/s. Taking this into account, there is a huge gap in speed between DRAM and SSDs (one to two orders of magnitude). The same is also true for flash memory, because modern SSDs are flash-based. Consequently, using an SSD as a secondary external storage does not guarantee that the system will work smoothly or the stream would not lag behind because still SSDs cannot match the latency of memory operations. The next scenario is to consider if the substitution of HDDs with SSDs will bridge the gap between memory and disk latencies and hence solve the problem of the lagging stream. Despite the advantages of SSDs over HDDs in terms of speed, but there are still some issues regarding durability and capacity. The latter is not an overwhelming issue, because the maximum threshold of 1 TB offered by modern SSDs is adequate for most applications. As far as durability is concerned, SSDs are more likely to fail quicker than an HDD when they are put under heavy load. To understand why durability is an issue, it is important to understand how SSDs work. In particular, they have to erase an entire block at a time before writing to it. Because SSDs write data to pages but erase data in blocks, the amount of data being written to the drive is always larger than the actual update. If a single change is made to a 4KB file, for example, the entire block that 4K file sits within must be updated and rewritten. Depending on the number of pages per block and the size of the pages, the SSD might end up writing 4MB worth of data to update a 4KB file. The only way for an SSD to update an existing page is to copy the contents of the entire block into memory, erase the block, and then write the contents of the old block along with the updated page. If the drive is full and there are no empty pages available, the SSD must first scan for blocks that are marked for deletion but that haven't been deleted yet, erase them, and then write the data to the now-erased page. This is why SSDs can become slower as they age. A mostly-empty drive is full of blocks that can be written immediately, while a mostly-full drive is more likely to be forced through the entire program/erase sequence. Therefore, they wear out at a much higher rate than the HDD and they are extremely susceptible to wear if they are put under an infinite stream of writes and updates. Of course, modern SSDs have more advanced controllers which try to minimize the damage and increase life-span. If the system entailed mostly reads and a small of amount of writes and update operations, then the SSDs would be a good solution as a secondary storage. There are also other issues like power consumption, cost of buying,

and so on, that do not concern this thesis and will not be further discussed. On the other hand, there are a few promising technologies under development that may change the memory/disk storage landscape. Both magnetic RAM and phase change memory are upcoming technologies that have presented themselves as candidates. Nonetheless, both technologies are still in early stages and must overcome significant challenges to actually compete as a replacement to the current state of an SSD drive. Taking all the above into consideration, although SSDs seem a good alternative in comparison to the above workarounds, it is still unclear if they solve the problem at the time of writing this thesis or in the near future. Their lower latency compared to HDDs comes at a cost of less reliability and fault-tolerance. They could be used with some additional modifications, (like for example mirroring the contents of an SSD to another disk and so on) which may increase the complexity of the system. Either way, they seem to be a good approach, but this thesis will concentrate on finding an algorithmic solution without the drawbacks of the SSDs.

2.4 Alternative Approach

Taking all the above into account, it is clear that in order to provide a viable solution to the problem and address all the challenges, an out-of-the box approach should be used. This thesis proposes that the system which will address the problem efficiently should satisfy the following requirements:

- The proposed system should act as a cache with configurable element capacity, which will store the hot elements.
- The cold data would be evicted to disk according to an eviction policy. The disk should not act as backup for the data in memory. If that would be case, the mirroring of the updates to the disk would make the whole process too slow. Therefore, the disk and the caching components will have different data. Also the data updates to the disk should be consistent.
- Having the hot elements available in memory at all times ensures that the expensive trips to disk would be minimized, thus ensuring low latencies. This makes the proposed architecture able to be used as a stateful memory component for a stream processor. It could also be used as a standalone system or a distributed one in a real-time architecture.
- Since the system would be used in a real time architecture and the distribution of streaming data may change (online processing), cold elements should be able

to automatically return to memory again if they start getting referenced. The proposed eviction strategy that would perform well is the LFU policy because it can advantage of the skew in the data - however the system should also be able to work with little or no skew at all.

Chapter 3

Background and Related Work

This chapter summarizes required background knowledge about specific subjects that are mentioned throughout the thesis. It also presents and analyzes available databases, and checks whether they could contribute somehow to the solution of the problem that this thesis poses.

3.1 The Concept of Caching

Caching is one of the most fundamental topics of computer engineering. It is the technique of storing a copy of data temporarily in rapidly-accessible storage media (also known as memory) local to the central processing unit (CPU) and separate from bulk storage. In fact, it has been around for as long as computing itself.

From web browsers, to DNS systems and CPU caches, nowadays it is almost unimaginable to think that a modern system does not use any form of caching. Caching is not only limited to simple variables. It is also used for more complex objects like images, data access objects, security credentials, web pages, parts of a graph and many other things. The existence of cache is based on a mismatch between the performance characteristics of core components of computing architectures, namely that bulk storage cannot keep up with the performance requirements of the CPU and application processing.

The concept of caching is based on the "The Long Tail" term, which was studied by the mathematician Benoît Mandelbrot in the 1950s [7]. However, this term is not only limited to math and statistics. It is also found in financial systems, marketing models and social network mechanisms. For instance, in e-commerce, a small number of items may make up the bulk of sales, a small number of blogs might get the most hits and so on. While there is a small list of popular items, there is a long tail of less popular

ones. In statistics, the Long Tail is itself a vernacular term for a Power Law probability distribution. One form of a Power Law distribution is the Pareto distribution, commonly known as the 80:20 rule. This phenomenon is useful for caching. If 20% of objects are used 80% of the time and a way can be found to reduce the cost of obtaining that 20%, then the system performance will improve. This phenomenon also appears throughout the nature. An example is natural language processing. In the Brown Corpus of American English text, the word ”the” is the most frequently occurring word, and by itself accounts for nearly 7% of all word occurrences (69,971 out of slightly over 1 million). True to Zipf's Law, the second-place word ”of” accounts for slightly over 3.5% of words (36,411 occurrences), followed by ”and” (28,852). Only 135 vocabulary items are needed to account for half the Brown Corpus [8]. The next section describes how an application can be benefited from caching.

3.1.1 How Caching Helps

In order to thoroughly understand how caching enhances a system in terms of speed and robustness, it is important to understand what is the problem that caches solve in the first place.

There are 2 types of bottlenecks in an application: In particular, the applications can be either CPU-bound or I/O-bound. If an application is I/O-bound then then the time taken to complete a computation depends primarily on the rate at which data can be obtained. If it is CPU-bound, then the time taken primarily depends on the speed of the CPU and main memory.

While the focus for caching is on improving performance, it is also worth realizing that it reduces load. The time it takes something to complete is usually related to the expense of it. So, caching often reduces load on scarce resources.

3.1.1.1 Speeding up CPU-Bound Applications

CPU bound applications are often sped up by:

- Improving algorithm performance.
- Parallelizing the computations across multiple CPUs (SMP) or multiple machines (Clusters).
- Upgrading the CPU speed.

The role of caching, in this case is to temporarily store computations and data that may be reused again. For example:

- Large web pages that have a high rendering cost.
- Caching of authentication status, where authentication requires cryptographic transforms or computing complex cryptographic hashes.

3.1.1.2 Speeding up I/O-Bound Applications

Many applications are I/O-bound, by either disk or network operations. In the case of databases, they can be limited by both.

There is no Moore's law for hard disks. A 10000 RPM disk was fast 10 years ago and is still fast. Hard disks are speeding up by using their own caching of blocks into memory. Network operations can be bound by a number of factors:

- Time to set up and tear down connections.
- Latency, or the minimum round trip time.
- Throughput limits.
- Marshaling and unmarshaling overhead.

The caching of data can often help a lot with I/O-bound applications. Some examples of this case are:

- Data Access Object caching for a database or an ORM.
- Web page caching, for pages generated from databases.
- Blob caching, for large files such as multimedia objects.

In this case, caching may be able to reduce the workload required. If caching can cause 90 of that 100 to be cache hits and not even get to the database, then the database can scale 10 times higher than otherwise.

3.1.1.3 Characterization of this Platform

Obviously, the situation that the particular thesis describes is characterized as I/O-bound. Of course, this depends on the data that are handled by the system. If it was assumed that there are data which require complex CPU-intensive computations, then the application could be CPU-bound as well. However, this thesis focuses only on data-intensive applications and therefore it is assumed that the system is I/O-bound. Another fundamental issue that needs to be discussed is how faster the system will become if caching is used. This is considered next.

3.1.2 Expected Speedup

One of the most fundamental questions regarding this topic is how much an application will speed up with caching. There is not a definite answer, because that depends on a multitude of factors being:

- How many times a cached piece of data can and is reused by the application. This factor is crucial because it largely determines which eviction strategy is most suitable.
- The proportion of the response time that is alleviated by caching.

In applications that are I/O-bound, which is most business applications, most of the response time is spent on getting data from a database. Consequently, the speed up mostly depends on how much reuse a piece of data gets. In a system where each piece of data is used just once, it is almost zero. In a system where data is reused a lot, the speed up is large.

The speedup of the application can be estimated numerically using *Amdahl's Law* [9] which is used to find the system speed up from a speed up in part of the system. It can be formulated in the following way:

$$Speedup = \frac{1}{(1-p) + \frac{p}{s}} \tag{3.1}$$

where

- *Speedup* is the theoretical speedup of the execution of the whole task;

- s is the speedup of the part of the task that benefits from improved system resources;
- p is the proportion of execution time that the part benefiting from improved resources originally occupied.

At the time of writing this thesis, a random read/write in DRAM is in the order of nanoseconds, while a disk I/O is in the order of milliseconds. Therefore, DRAM is 1000000 times faster than a disk I/O. If a system spends 70% of the execution time retrieving or storing an object from the database and the rest doing calculations in main memory, then the expected speedup using *Amdahl's Law* is calculated as follows:

$$Speedup = \frac{1}{(1 - 0.7) + \frac{0.7}{1000000}} \tag{3.2}$$

This example assumes a monolithic system and it is merely estimation. Also *Amdahl's Law* assumes that each operation has a fixed cost. In the context of this example, it assumes that the cost of each 'trip' to the database is constant throughout the whole execution. Chapter 4 will argue that this is not always the case because some disk accesses will cause additional lookups if there is an eviction. However, this is a simple case which demonstrates numerically the usefulness of a cache.

3.1.3 Eviction Strategies

One of the most important factors that characterizes a cache is its eviction strategy. In fact, the use of a particular eviction strategy largely depends on the nature of the problem that should be solved. For example, if the data follows a Pareto distribution, then a frequency-based cache is more useful than a recency-based one. Also, some caches are more complex and require more resources than others. There are also caches which are not widely used, but they are optimal than their counterparts at very specific use-cases. Generally speaking, all eviction strategies are partitioned in 2 subcategories:

1. **Timed-based Eviction or Time bound cache**
   As the name suggests, data are subjected to an expiry policy. When an object is inserted in the cache, the user specifies a time interval for that particular object. The user can also specify a time interval that applies to all elements at once. Therefore, the entries are held for a predefined period. The most usual eviction strategies in this sub-category are:

- **Time to Idle Seconds (or TTI)** – The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache.
- **Time to Live Seconds (or TTL)** – The maximum number of seconds an element can exist in the cache regardless of use. Same as before, the element expires at this limit and will no longer be returned from the cache.

The greatest advantage of this type of eviction policy is that it is a simple and very easy to use. For instance, in Java it can be implemented using *ExpringMap*, or by using an off-the-self library like *Guava*.

2. **Size-based Eviction**
   This type of caches holds entries until they are invalidated by subsequent updates. A threshold is specified, after which the entries are invalidated and hence replaced according to an eviction strategy. The threshold can be a memory constraint or a maximum number of entries allowed inside the cache. The latter is more popular, because it can be used with all eviction policies and is the easiest to implement.

   Least Recently Used (LRU): This is the most popular eviction strategy. It works pretty well because of the locality of reference phenomenon and is the default strategy in most caches. In this scenario, the element with the oldest timestamp is evicted. The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call. In fact, this is its simplest form. However, this algorithm has evolved the past years, and has become more efficient. For example, Adaptive Replacement Cache (ARC) [10] which is based on LRU has better performance. That's because it combines the use of the locality of reference with frequency.

   Least Frequently Used (LFU): One of the most efficient eviction policies. It works well when data popularity is skewed and follows a Pareto distribution. For each get or update call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached) the element with least number of hits, the Least Frequently Used element, is evicted. One variation of this cache is the Least Frequent Recently Used (LFRU) cache, [11] which combines both LRU and LFU. In LFRU, the cache is divided into two partitions called privileged and unprivileged partitions. The privileged partition can be defined as a protected partition. If content is highly popular, it is pushed into privileged partition. If it is required to replace content from the privileged partition, the replacement is done as follows: LFRU evicts content from unprivileged partition, pushes content from privileged partition to unprivileged

partition, and finally inserts new content in privileged partition. The privileged partition acts an LRU, while the unprivileged acts an approximated LFU. This type of cache is most suitable for in-network applications, such as Content Delivery Networks.

LFU with Dynamic Aging (LFUDA): This variant of LFU uses dynamic aging to accommodate shifts in the set of popular objects [12]. In the dynamic aging policy, a cache age factor is added to the reference count when an object is added to the cache or an existing object is referenced. This prevents previously popular documents from polluting the cache. Instead of adjusting all key values in the cache, as some aging mechanisms require, the dynamic aging policy simply increments the cache age when evicting objects from the cache, setting it to the key value of the evicted object. This has the property that the cache age is always less than or equal to the minimum key value in the cache. This also prevents the need for parameterization of the policy, which LFU-Aging requires.

First In First Out (FIFO): Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).This algorithm is used if the use of an element makes it less likely to be used in the future. An example of this case would be an authentication cache.

Most Recently used (MRU): A less popular algorithm which discards, in contrast to LRU, the most recently used items first. Despite the fact that this policy has not many uses, in some cases it outperforms other policies. For example, when a file is being repeatedly scanned in a Looping Sequential reference pattern, MRU is the best replacement algorithm [13]. Moreover, for random access patterns and repeated scans over large datasets (sometimes known as cyclic access patterns) MRU cache algorithms have more hits than LRU due to their tendency to retain older data [14]. MRU algorithms are most useful in situations where the older an item is, the more likely it is to be accessed.

Randomized eviction: This policy randomly chooses entries to evict. It is the simplest in terms of implementation and the least efficient of all strategies. This eviction policy is mainly used for debugging and testing purposes.

Randomized LRU: It is a page-based eviction policy that combines LRU and randomization. Once a memory region defined by a memory policy is configured, an array is allocated to track the 'last usage' timestamp for every individual data page. When a data page is accessed, its timestamp gets updated in the tracking array. When it is time to evict a page, the algorithm randomly chooses a configurable number of indexes from the tracking array and evicts the page with the

oldest timestamp. If some of the indexes point to non-data pages (index or system pages), then the algorithm picks another page. This implementation has many variations though. Another version is to evict 2 pages. More specifically, the two most recent access timestamps are stored for every data page. At the time of eviction, the algorithm randomly chooses 2 indexes from the tracking array and the minimum between two latest timestamps is taken for further comparison with corresponding minimums of four other pages that are chosen as eviction candidates. This method outperforms LRU by resolving the *one-hit wonder* problem: if a data page is accessed rarely but accidentally accessed once, it's protected from eviction for a long time.

3.1.4 Caches and Machine Learning

In the past few years, with the explosive growth of machine learning/deep learning and its many applications, a lot of research has been made in the caching area too. In fact, it is not the first time that a machine learning-powered approach is made to a problem which traditionally lies in the field of data structures and databases. For example, Tim Kraska et al. in [15] use neural networks to train indexes which can learn the sort order or structure of lookup keys and use this signal to effectively predict the position or existence of records. Similarly, in caching the idea is to treat an eviction policy as a statistical model and try to predict the elements which should be evicted in order to optimally increase cache-hit ratio [16] [17]. Although these approaches surpass the performance of traditional eviction strategies in certain scenarios, there is still room for improvement in the future.

3.1.5 Writing Strategies

There are three main caching techniques that can be deployed, each with its own pros and cons.

3.1.5.1 Write-Through Cache

Write-through cache directs write I/O onto cache and through to underlying permanent storage before confirming I/O completion to the host. This ensures data updates are safely stored on, for example, a shared storage array, but has the disadvantage that I/O still experiences latency based on writing to that storage. Write-through cache is good for applications that write and then re-read data frequently as data is stored in cache and results in low read latency.

3.1.5.2 Write-Around Cache

Write-around cache is a similar technique to write-through cache, but write I/O is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write I/O that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a "cache miss" and have to be read from slower bulk storage and experience higher latency.

3.1.5.3 Write-Back Cache

Write-back cache is where write I/O is directed to cache and completion is immediately confirmed to the host. This results in low latency and high throughput for write-intensive applications, but there is data availability exposure risk because the only copy of the written data is in cache. Nevertheless, the past decade things have improved and suppliers have added resiliency with products that duplicate writes. Users need to consider whether write-back cache offers enough protection as data is exposed until it is staged to external storage. Write-back cache is the best performer for mixed workloads as both read and write I/O have similar response time levels.

3.2 Caching Challenges

Generally speaking, the most important parameterization for a cache is finding the right eviction strategy and adjusting it according to the needs of the application. Additionally, caching can encounter challenges that include, for example, the problem of cache warm-up, where cache needs to be loaded with enough active data to reduce cache misses and allow it to start improving I/O response times. There is a technique to cope with this issue which will be shown in the next chapter.

The next step is to evaluate which eviction policy is the most suitable for solving the problem that this thesis posed. To recap, the problem are the random I/Os at the external storage. If the goal was to perform batch-processing, then the additional latency of the I/O cost would not hurt the performance. Stream processing however requires low latency, in order to sustain high throughput. Consequently, solving the problem comes down to 2 sub-problems, which can be tackled separately:

1. **Limit the number of I/Os as much as possible**
   The truth is, it is impossible to get rid of disk I/Os, but what can be done is to limit them as much as possible. The idea is to deploy a cache in main memory, where

the most accessed entries would be kept in. The other entries which are deemed as less frequent would be kept in a hard disk, which is assumed to be much larger than the main memory. The eviction strategy which is more appropriate for the problem described above is the LFU. Apart from that, LFU also takes advantage of skew in the data. Naturally, most data distributions have skewed elements. By keeping the most frequent entries in the cache, the number of accesses to the disk is limited, because the algorithm interacts with much higher probability with the "hot" entries. Of course, that does not mean that the system should only work with a heavily-skewed distribution. The goal is to have a satisfactory performance even with the worst-case scenario (uniform case), and simultaneously prove that the system performs better the more skew the distribution has. Finally, since the distribution of the elements which the system consumes is supposed to be unknown, there is a chance that a cold element which is not referenced frequently, to become frequent in the future. Therefore, the application should not simply evict elements and apply the LFU policy only in the cache. In this case, cold elements on the disk which may start becoming hot again, must be transferred back to the cache.

2. **Lower the latency of I/Os as much as possible**
   Unfortunately, the latency of the I/Os cannot be limited below a certain threshold, because the cost of moving the head assembly must also be accounted for. According to database theory, the cost of random I/Os is much higher than sequential ones. For example, main memory is only about 6 times faster when the disk is performing sequential access (350 Mb/sec for memory compared with 60 Mb/sec for disk). Nonetheless, memory is about 100000 times faster when the disk performs random access. Therefore, the problem can somewhat be alleviated by not doing completely random accesses at the disk. The best strategy in this scenario is to use a write buffer. The buffer will hold some entries until it becomes full. After that, all changes will be committed and synched to disk in a single access, thus limiting the overall cost.

3.3 Databases and Key-Value Stores

This section presents and analyzes the in-memory database ecosystem. The goal is to perform a search and try to find an eligible system which satisfies all the aforementioned requirements, thus avoid creating one from scratch. Even if a database is rejected, the reasons of its rejection will also be presented. It is worth mentioning that the successful database should also have the option for persistence. This means that pure in-memory systems such as *Memcached* and *Apache Arrow* will not be discussed at all. Also, open

source systems will be considered more seriously. If a commercial product is found to do the job similarly to an open source one, then the latter will be used instead.

3.3.1 Redis

Redis [18] is one of the most popular in-memory key-value store [19], implementing a distributed, in-memory key-value store. First introduced in 2009, it is a reliable and battle-tested system because it is widely used in both industry and academia. It is open-source, with a great number of contributors, while it is also supported by popular companies like Facebook and Twitter. As far as its memory-processing capabilities are concerned, there is no match, because it excels in high throughput, fault-tolerance and robustness. Redis is a state-of-the-art project that offers advanced features, such as partitioning, replication, publish-subscribe functionality and so on. Redis however does not have full ACID properties, since it does not support transactions. Also, it supports eventual consistency instead of immediate consistency, which is less robust and reliable. This is not a critical drawback because Redis is supposed to work that way: It favors high availability over consistency. There are some workarounds that can improve consistency at the application level. Furthermore, it also offers the choice for data persistence. If data persistence has been activated, then the contents of the in-memory data structures stored in Redis are mirrored to the disk, thus guaranteeing fault-tolerance and no single point of failure. Of course, this feature induces additional costs because data should be updated at the disk as well. The problem though is that the purpose of this thesis requires the hot data to be in stored in main memory and the cold ones in disk, which means that main memory and disk storage must have different data. Therefore, although Redis offers a plethora of advantages regarding the in-memory processing functionalities, it cannot be used because of its persistence limitations.

3.3.2 Apache Ignite

Apache Ignite [20] is one of the newest additions in the in-memory platform ecosystem and very similar to Redis, which makes it a direct competitor. Introduced in 2014 and incubated in 2015 at the Apache foundation, it is a promising project open-source with a plethora of in-memory processing capabilities and options for persistence. It is mostly implemented in Java. Data in Ignite is stored in the form of key-value pairs in the off-heap memory by default, in order to avoid long pauses from garbage collection. There is also the option of storing data in the Java heap as well. The database component scales horizontally, distributing key-value pairs across the cluster in such a way that every node owns a portion of the overall data set. Therefore, its clustering component is based on

the shared nothing architecture. Contrary to Redis, Ignite supports the ANSI-99 SQL standard with joins, ACID transactions, immediate consistency as well as MapReduce like computations and off the shelf machine learning APIs. As a downside, it does not support master-slave replication or multi-master replication that Redis does. Finally, Ignite also offers native persistence, similar to Redis, but with some extra features. In particular, persistence is optional and can be turned on and off. When turned off, Ignite becomes a pure in-memory store. With the native persistence enabled, Ignite always stores a superset of data on disk, and as much as possible in RAM. For example, if there are 100 entries and RAM has the capacity to store only 20, then all 100 will be stored on disk and only 20 will be cached in RAM for better performance. Every time the data is updated in memory, the update will be appended to the tail of the write-ahead log (WAL). The purpose of the WAL is to propagate updates to disk in the fastest way possible and provide a consistent recovery mechanism that supports full cluster failures. In other words, Ignite simply mirrors data stored in main memory at the disk, which is something that also Redis does (without the transactional part), and as it was explained above, this is not the desirable use case. Apart from that, Ignite also offers the option of eviction from memory and persistence to disk, when the memory designated by the user is filled up. The user has a variety of eviction strategies to choose from such as LRU, FIFO, eviction using TTL (time-to-live requests) and completely random eviction. While LRU is a very good option, according to the problem that the thesis formulated above, an LFU eviction strategy would be more suitable. Nevertheless, Apache Ignite is a very popular project with many contributors and supporters, and it would be no surprise if in the future it will move towards this direction.

3.3.3 Ehcache

Ehcache [21] is probably the most widely used and the most popular Java cache platform. It is mostly open source and has a lot of contributors. Unfortunately, some of its features require a paid license and are only available in commercial products. These include the Enterprise Ehcache and BigMemory, which offer major functionalities such as distributed caching, replication and Fast Restartability Consistency (option to store a fully consistent copy of the in-memory data on the local disk). This is a big downside, because all the above features have many applications in modern deployments. Apart from that, Ehcache is general purpose caching system and it can be found preinstalled in a plethora of systems, such as Hibernate, Google App Engine, JTA (a component of JEE ecosystem) and so on. Moreover, it has a powerful set of APIs such as SOAP and RESTful services, a gzip caching servlet filter. The latter provides a set of general purpose web caching filters in the ehcache-web module. Using these can make a

significant difference to web application performance. With built-in gzipping, storage and network transmission are highly efficient. Cache pages and fragments can be compressed and stored in memory or in the disk. As far as the caching functionalities are concerned, the memory obviously can be used as a cache and its content can be flushed to the disk according to some eviction strategy. Unfortunately, one major drawback of Ehcache is that since version 2.0 it cannot be used exclusively as a disk database. The memory component on top of the disk storage must be activated. This excludes the usage of Ehcache solely as a datastore like *LevelDB*. Of all the aforementioned systems, Ehcache has the richest set of expiry policies. More specifically, it supports LRU, random eviction, FIFO, TTL, TTI (timeToIdleSeconds, which is the maximum number of seconds an element can exist in the cache without being accessed) and LFU. LRU is implemented in both a probabilistic and a deterministic way. On the other hand, LFU is implemented exclusively in a probabilistic way. According to the official documentation, it takes a random sample of the elements and evicts the smallest. Obviously, this technique increases speed but lowers accuracy. This poses a problem for the use case that was stated in the beginning of this section. However, even if a probabilistic way could be accepted, Ehcache does not return an element stored at the disk back to the cache, if it becomes hot again after a while. Taking all the above into account, it does not meet the required expectations. The fact that it is not open-source and some vital features are not available also has a negative effect and does not constitute Ehcache a successful candidate.

3.3.4 MapDB

MapDB [22] is an open-source project which can be used both as an in-memory data store and a database. Although it was released in 2014, the creation of MapDB is the culmination of years of research and development to get the project to that point. According to the author, the concept of MapDB is largely based on JDBM, a project which the author started developing in 1999. The first version was written purely in Java, but since then development has started moving towards Kotlin. In fact, the latest stable version of MapDB (version 3.0) has more lines of code in Kotlin than in Java. Of all the aforementioned systems, it is the least popular (both in industry and academia) and has the least number of contributors. The greatest advantage of MapDB is that it leverages the already existing Java Collections API, which is so familiar to Java developers that most of them literally use it daily in their work. More specifically, MapDB provides Java maps, concurrent maps, sets, lists, queues and so on, which are stored in the JVM off-heap memory. Therefore, it does not only function as a simple key value store. Moreover, the same data structures can be used for disk storage as well. This is one of

the biggest strengths of MapDB that distinguishes it from the aforementioned systems. For example, any HashMap object can be handled similarly both in memory and in disk, by using similar APIs, thus reducing complexity in the code. Consequently, concepts like serialization/deserialization, transformations, writes and so on are more flexible in terms of dealing with objects. Furthermore, given the capabilities of MapDB, it can be used as a caching system, with optional disk persistence as a secondary storage. MapDB supports eviction mechanisms, but there is not a wide range of options to choose from, as opposed to other systems described in this section. In particular, it uses LRU, Random, and TTL eviction. Unfortunately, the desired case of the LFU eviction is absent, and therefore it cannot be used for its caching mechanisms. Finally, MapDB is not a distributed system and therefore it does not support features such as sharding, replication etc. It is meant to be used as an embedded database. However, it is not difficult to add an extra layer above and partition it at the application layer.

3.3.5 Conclusion

Taking all the above into consideration, there is no suitable system that meets all the requirements in order to solve the problem that this thesis poses. Some desired features such as partitioning, fault-tolerance and a rich set of in-memory data structures are present in almost all of the databases above. But in order to give a conclusive solution, the eviction policy that is most suitable for the current situation is LFU. The next chapter discusses the requirements and the details of building such as a system.

Chapter 4

Implementation

This chapter describes the architecture of the platform in depth and explains all the implementation-specific design choices that have been made along the way in order to increase robustness and speed.

4.1 Layers of the Platform

The subsections below present the layers in depth. Each layer serves a different functionality. Apart from that, the multi-layered organization makes the platform more flexible and elastic. The overall architecture is visually presented in Figure 4.1. The implemented algorithms are also explained here.

4.1.1 First Layer

The first layer on top of the whole a system contains a *Bloom filter*. Newly arrived data will access the bloom filter first, before hitting the memory layer. The deployment of the bloom filter is an optional technique since there is a trade-off between memory occupancy and throughput. The user can choose whether to deploy it or not, according to his needs.

4.1.2 Second Layer

The second and the most fundamental layer of the system contains the caching logic. Since there is no off-the-shelf in-memory database that covers all the aforementioned requirements, the implementation of caching logic is custom and does not depend on external in-memory databases. More specifically, this layer contains:

- The data structures which manage the entries in main memory-processing.
- The write buffer.
- The APIs which communicate with the other layers.
- The implementation of various eviction policies.

The eviction policies include the adapted versions of MRU, LRU and LFU. This thesis however focuses mostly on the LFU policy, which is the most difficult to implement and its proposed adaptation cannot be found in other popular systems. Apart from that, LFU is the most useful policy in the context of this project, because it is the most efficient for solving the problem of the expensive I/Os. The rest of the policies were built in order to present a fully-functional caching system.

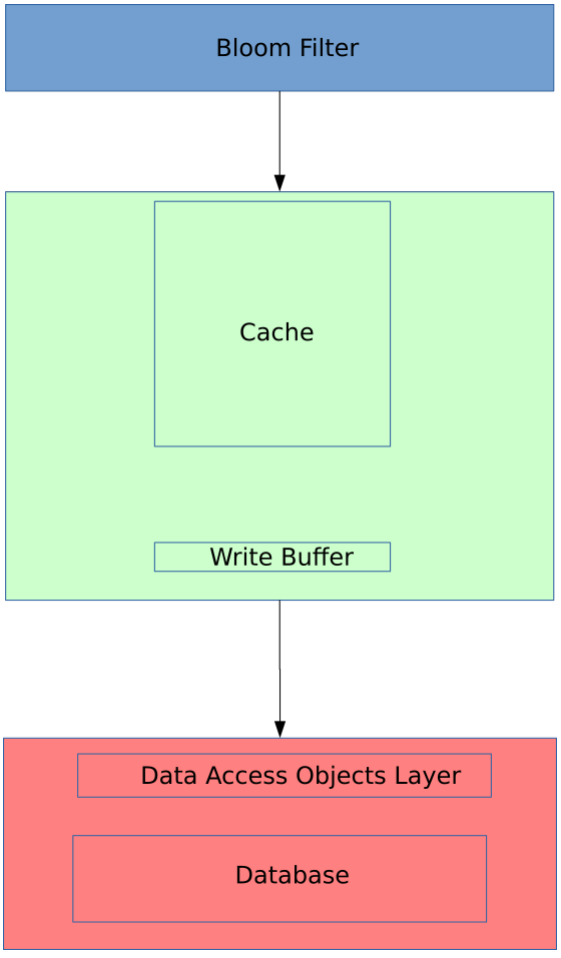

FIGURE 4.1: *The overall architecture of the system. Each layer is depicted with a different color*

4.1.3 Third Layer

Finally, the persistence layer contains the database for storing the less frequent elements. Contrary to the previous layer, there is no need to create a custom database system, since any existing system could do the job. This layer contains the APIs that read, save and update the elements on the external system. The user can choose any database he wants, as long as it is compatible with the APIs that explicitly define the correct structure of the data that are used in the platform. In other words, the database is not hard-coded in the system. All functions on top of the database that handle data operations are completely unaware of the database. For the current thesis, the database that was chosen is MapDB. It is worth mentioning that MapDB is an embedded key-value store and not a pure database. MapDB was chosen for 2 main reasons. First and foremost, it is a light storage system which can be configured and deployed easily. Secondly, the platform is implemented in Java, same as MapDB. This in turn has many advantages. In particular, it is possible to store complex Java objects as values in MapDB without additional overhead. For example, a complex object in Java could be a *JSON* object. Other key-value stores that are written in different languages, such as *LevelDB* [23] which is written in C++, require additional wrappers for Java types and do not support a rich set of values such as MapDB. For instance, LevelDB cannot store Java Objects, Hash Maps etc as values. Consequently, using MapDB has the advantage of providing a variety of options to the user and simultaneously keeps the architecture simple and robust. Last but not least, this layer includes a *static Connection Pooling module*. This component gives the ability to open many parallel connections to the database from an application. When the connections are closed, they are returned to the pool, without being destroyed. Therefore, they are ready to be used by other requests, and the cost of creating connection objects anew (which is expensive) is avoided. This mechanism boosts performance and adheres to the modern way of deploying complex big data systems in production.

4.2 Additional Specifications

The system, which can be characterized as a hybrid caching system with additional persistence capabilities, also has the following specifications:

4.2.1 Modularity and Flexibility

Modularization is a big issue in today's programming. Programmers all over the world are trying to avoid the idea of adding code to existing classes in order to make them support encapsulating more general information. Take the case of an information manager which manages phone number. Phone numbers have a particular rule on which they get generated depending on areas and countries. If at some point the application should be changed in order to support adding numbers form a new country, the code of the application would have to be changed and it would become more and more complicated.

Contemporary software engineering architectures aim to increase modularity by allowing the separation of cross-cutting concerns [24]. They do so by adding additional behavior to existing code (an *advice*) without modifying the code itself. Instead, they separately specify which code is modified via a "pointcut" specification, such as "log all function calls when the function's name begins with '*set*'". This allows behaviors that are not central to the business logic (such as logging) to be added to a program without cluttering the code, core to the functionality.

This entails breaking down program logic into distinct parts (so-called *concerns*, cohesive areas of functionality). Nearly all programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (e.g., functions, procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns. Some concerns "cut across" multiple abstractions in a program, and defy these forms of implementation. These concerns are called *cross-cutting concerns* or horizontal concerns.

For example, in the current thesis the cache is considered a 'concern', and the eviction strategy is an 'advice'. The cache is the main entity, and each eviction policy gives a different behavior to the cache. The user can add eviction policies/behaviors to the cache seamlessly by encapsulation. The cache is decoupled from the eviction policy because its only job is to abstract the get and the put methods, without taking into consideration how each eviction policy is implemented. This plays a significant role in the flexibility and the robustness of the system because the user can add different eviction policies without breaking the code. The same technique is also used in the persistence layer as explained above. That's why it is modular and the user can plug his own system which handles the persistence functionality. This principle in software engineering is also known as aspect-oriented programming [25]. More details about how these features are implemented are analyzed in the next sections.

4.2.2 Scalability

Generally speaking, all big data systems nowadays are distributed in order to be able to handle a large amount of data with efficiency. Apart from that, a plethora of streaming algorithms is expected to work in parallel by taking advantage of many nodes.

Consequently, an additional characteristic of the system is its ability to work in a distributed manner. In order to achieve this, the user can create more than one instances of the system. The stream then can be partitioned among these instances (by using for example hash partitioning). This can be implemented relatively easily, since each key can be assigned to a particular instance, according to a partitioning function. Moreover, the stream itself can be divided into one or more sub-streams. A top-level view of this feature is shown in Figure 4.2

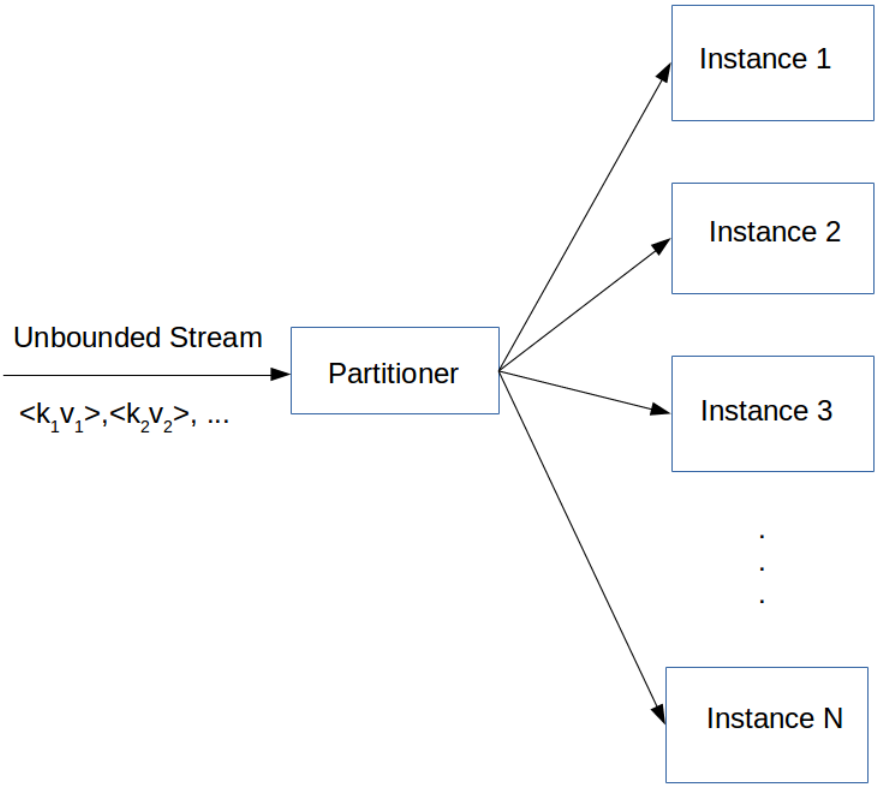

FIGURE 4.2: *The stream is partitioned and each itemset is assigned to a particular instance*

In addition to this, the platform is also built to work in tandem with a stream processor. More specifically, the user can instantiate an instance of the system (as a single instance)

inside a worker/node of a stream processor. The system, using a limited amount of memory, can help the sub-partition of the stream processor store and handle large amount of data at speeds which require sub-second latencies. A useful case would be to use this platform embedded inside the popular real-time processing framework Apache Storm. This is shown in Figure Figure 4.3. Spouts and bolts are the two basic primitives that the user has at his disposal in order to perform the needed computations. A spout can be thought as a ”tap” that produces tuples at a constant, variable or event driven rate. Bolts are responsible for all the transformations and processing that happens within each of the currently running topologies in the cluster.

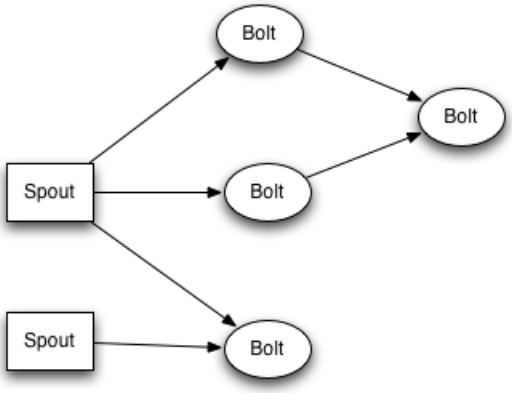

FIGURE 4.3: *The system can be used inside each bolt, where Apache Storm's processing logic takes place*

In the above example, the system can be embedded as a single instance inside each bolt where Apache Storm's processing logic takes place. Although this example shows only Apache Storm, the platform is configured to work with other stream processors too. This is because the platform can work independently using the AOP pattern and support real-time processing speeds.

4.2.3 Generic Values

One the most fundamental features of a caching system or a database is the datatype of the values that is compatible with. In a traditional implementation, the choice of the variable would largely depend on the type of the database that is used in the underlying persistence layer. In the case of this thesis, the system which is presented is able to use any possible variable as a value or even a Java object. This is mainly achieved because MapDB (which is the default database in this architecture) can handle directly any Java object. If the user wishes to use a different database instead, like for example

a relational database, all he has to do is to is implement the *Data Access Object APIs* and handle the object in a way that is compatible to the database. For example, if the user had used PostgreSQL instead, he would have to create a table with 2 variables (key and value) and then use the JDBC driver to handle the data by calling the Data Access Object APIs of the platform.

However, since MapDB gives the freedom to use any Java object directly as a value, the value variable in the cache interface is defined as a *Java Generic type Object*. The user explicitly specifies what that type wishes to be during the instantiation of the whole system. Some examples of values which could be used are:

- Data structures like arrays, hash maps and lists. For example, the user can store a random variable as a key and an one-dimensional histogram or a pdf as hash map (the value).
- Complex Java objects, which can be a combination of any data type. For example, the key could be a node in the graph and the value could be a Java object which consists of:
  - An integer number containing the weight of the particular node
  - A hash map containing its neighbors and their respective values
  - A hash map or a list containing the edges and their respective values
- Custom Java objects which are created using an external Java library. For instance:
  - A *JSON* object (by using e.g. the Jackson Library)
  - An *XML* object (by using e.g. the XMLBeans Library)
  - A *DoubleMatrix2D* (by using the Colt Library)
  - etc

Taking all the above into consideration, using a Java-based database like MapDB gives a variety of choices to the user by default. Moreover, the user has also the freedom to plug the database of his choice. It is obvious that every database has its pros and cos, and a single database cannot cover all the challenges that are presented.

4.3 Differences with a Traditional Caching System

As it was stated above, the architecture of the particular platform uses an alternative approach to the traditional storage hierarchy of disk-based systems. There are some notable differences:

- First and foremost, in the traditional caching systems, disk storage acts as a backup for the whole dataset, while in the particular implementation the disk acts as a cold storage for evicted data. This also means that data in disk are not copied back to the cache.
- The fundamental assumptions about the memory size are different. In the traditional paradigm, the memory size is much smaller than the total data size. In the proposed architecture, the memory size is larger and can hold a relatively bigger amount of data. That is why throughout this thesis, the layer which is responsible for memory management, is referred to as memory layer instead of caching layer. This definition helps distinguishing the two cases.
- In the caching domain, in case of a cache miss, the application retrieves data from the disk storage and stores them in the cache. Evicted data do not return back to disk from memory since they already exist in the disk. On the other hand, in this platform, the application does not have privileges to directly access the disk storage. Instead, only the memory layer can access the disk and dynamically evict or restore elements to the memory, as long as it complies with the necessary eviction policy. By assigning this responsibility to the memory layer, the computation of which elements are hot can be done in an algorithmic and application-agnostic way. This process is reflected in Figure 4.4

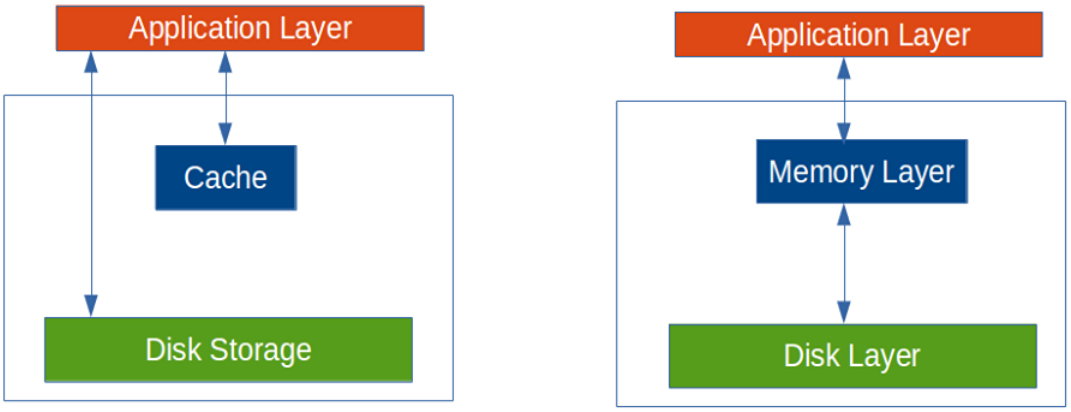

FIGURE 4.4: *Left: A traditional caching system, Right: The implementation of this thesis*

- Finally, the primary function of all caches is the eviction mechanism. However, in the particular implementation there are two fundamental operations which work in tandem, eviction and restoration. These two operations are responsible for optimally managing the dynamic exchange of elements between the memory and disk layer with the purpose of constantly keeping the hot elements in memory at all times.

4.4 Platform Architecture

This section presents in depth the algorithms and all the components which are used in each layer.

4.4.1 Bloom Filter

4.4.1.1 Overview

The first layer of the system is a bloom filter [26]. First of all, a bloom filter is essentially a data structure designed to designate, rapidly and memory-efficiently, whether an element is present in a set. It's like a set but it does not contain the elements. The basic bloom filter supports two operations: test and add.

Test is used to check whether a given element is in the set or not. If it returns:

- False then the element is definitely not in the set.
- True then the element is probably in the set. The false positive rate is a function of the bloom filter's size and the number and independence of the hash functions used.

Add simply adds an element to the set.

All the above operations have *O(K)* complexity, where K is the number of hash functions that are used. Practically, the cost is constant.

It is worth mentioning that removal is impossible without introducing false negatives, but extensions to the bloom filter are possible that allow removal like counting filters [27]. Also, counting bloom filters are less space-efficient. Hence, only the basic bloom filter is used in this thesis.

The price paid for this efficiency is that a Bloom filter is a probabilistic data structure. If the element is not present in the bloom filter, then it is known for sure that there is no need to perform the expensive lookup. On the other hand, if it is present in the bloom filter, then the lookup is performed, and it is expected to fail some proportion of the time (the false positive rate). Consequently, false positive matches are possible, but false negatives are not.

The classic example is using bloom filters to reduce expensive disk (or network) lookups for non-existent keys. For instance, bloom filters are used in web browsers, databases

authentication modules and so on. Moreover, they are embedded in other database systems, such as *Google BigQuery*[28]. Another interesting usage is using a bloom filter to optimize an SQL query (as many popular databases do).

4.4.1.2 Implementation Details

The role of the bloom filter in this system is to decrease expensive cache misses. If the bloom filter returns false, the algorithm will not search in the cache or disk for that itemset, and hence the system would immediately return that the itemset does not exist. This is particularly helpful at the initial phase when the cache is warming up and is more susceptible to cold misses. This prevents a potential heavy concentration of expensive lookups at the start that could have a negative impact on the throughput of the system. If the bloom filter returns true, then the algorithm will search for the item first in the cache and then in the disk. In case of a false positive, the system would return that the particular itemset does not exist. It is worth mentioning that the system is built to withstand random data as well, which negate the benefits of a cache. In this case, the use of the bloom filter is essentially important.

4.4.1.3 Technical Details

First and foremost, the most integral part of the bloom filter is the choice and the number of hash functions. The number of the hash functions that were used is considered in the next sub-section. The hash function that was chosen for this current implementation is *MurmurHash3* [29]. It is a popular non-cryptographic function with many applications. It basically consists of 2 operations, which are applied in this order: (mu)ltiply, (r)otate, (mu)ltiply, (r)otate along with some XOR operations. The reasons for using murmur hashing are the following:

- **Simple and fast:** It uses as few instructions as possible, while being as fast as possible and remaining statistically strong.
- **Distribution:** It passes the Chi-Square distribution tests for practically all keysets and bucket sizes to ensure there is no correlation whatsoever and is similar to pure randomness. The hash space is be filled randomly.
- **Avalanche Effect:** When one bit in the key changes, at least half the bits should changes in the hash. It is to ensure the function has a good randomization and no forecast is possible (or hardly).

- **Collision Resistance:** A good hash function should almost never have collisions. In the 128-bit variant, the hash space is quite huge: 3.4028237e+38: it should be nearly impossible to have a collision. Moreover, two different keys should have only a random chance to collision, no more.

These features make murmur hashing ideal for many applications such as UID generators, checksums, Hash tables and so on. It is also widely used in FM sketches, because it ensures a good randomization of 1's over the bitmap

There are two versions of this hash function: The 32-bit variant which produces integers, and the 128-bit variant which produces or 1 or 2 longs. As noted above, the 128-bit variant has a huge hash space and hence the number of collisions is minimized. However, the 32-bit variant is much more space efficient, and that's why this variant is used in the current thesis.

The implementation of bloom filter uses the *Funnel* class of the *Guava* library [30]. More specifically, the Funnel class helps storing and serializing the data of the bitmap.

4.4.1.4 How the Bloom Filter is Parameterized

Let m be the number of bits in the bitmap, n the expected number of elements to insert, and f the false positive probability. There must also be k different hash functions defined each of which maps or hashes some set element to one of the m array positions, generating a uniform random distribution. In general $k << m$. It is proved that:

$$k = \frac{m}{n} ln2 \tag{4.1}$$

$$m = -\frac{nlnf}{(ln2)^2} \tag{4.2}$$

Consequently, the bloom filter is simply parameterized by the expected number of elements and the false positive probability[31]. The second formula is asymptotically approximated and holds as $m \to \infty$. The implementation of the bloom in this thesis is based on the corresponding implementation of other popular systems like *Guava*[30], *Apache Cassandra*[32] and *RocksDB*[33]. Using the same formulas, they estimate k and m. For example, in RocksDB, when a user requests a bloom filter, RocksDB uses these

formulas to estimate the size of the bitmap, and it is only deployed when it can safely fit in main memory.

Having multiple hashes is necessary to avoid too many false positives: one hash function implies to check only one bit. With two hash functions, it needs to check two bits, therefore, because there is less chance to have both set. The other extreme is not good either. Having tons of hash functions means the bitmap is going to be filled quickly with 1's, therefore the rate of false positives is going to grow. It's a delicate equilibrium. In the current implementation, when a new instance of the system is created, the user should specify the false positive rate and the expected number of elements.

If the user underestimates the number of expected number elements, the bloom filter will start filling up. If the bloom filter becomes full, then the system assumes that every element has been seen by the bloom filter. In this case, the LFU algorithm will stop benefiting from the advantages of the bloom filter and the user will pay a memory cost from having a full bloom filter. However, even this case the system is designed to function normally and the algorithm of the eviction policy won't break.

If the user expects a large number of unique elements in the stream, but cannot afford to allocate enough memory in order to build the bloom filter, the solution is to use one or more instances. For example, if the number of expected elements is d, then the user can instantiate 2 instances and partition the stream among them. By doing so, each instance will receive $d/2$ elements. Certainly, the user can allocate as many partitions as he wants, according to the available resources or the nature of his application/algorithm. This feature demonstrates the flexibility and the scalability of the system.

Finally, the user may choose to not use the bloom filter at all. The first layer is deactivated and all the incoming elements will hit the memory layer.

4.4.2 Memory Layer

This is the most important layer in the system. The memory layer implements all the eviction policies and is responsible for the management of all elements in memory. Under the hood, this layer essentially implements the *Factory* design pattern [34]. This pattern is used when there is a super class with multiple sub-classes and a particular sub-class is returned based on the input. This pattern takes out the responsibility of instantiation of a class from the program to the factory class. Using this pattern provides 2 advantages:

- The class instantiation is deferred to the sub-classes. In other words, each eviction policy is instantiated independently without exposing the caching logic.

- It provides flexibility because more eviction policies can be added seamlessly.

The next section describes the eviction policies. While the LRU and MRU policies will be described in detail, the focus of this thesis is the LFU policy.

4.4.2.1 LRU

The LRU eviction policy is consisted of 3 procedures: *put*, *get* and *eviction*. The data structure which is used in the cache is ListOrderedMap, a map decorated as a list. Although the algorithm manages the elements in main memory, it also interacts with the database (when needed) using the Data Accessing Object APIs. The cache is oblivious to the type and the kind of the database and interacts with the persistence layer by using these APIs.

Algorithm 1 LRU – Put operation

```
1:  procedure PUTLRU(k, v)
2:      if (k ∈ cache) then
3:          cache.remove(k)
4:          cache.put(0, k, v)
5:      else
6:          cache.put(0, k, v)
7:          if (k ∈ bloom filter ) then
8:              LruDAO.remove(k)
9:      if (cache.size == cache.maxCapacity ) then
10:         call evictionLRU()
11:     bloomfilter.put(k)
```

Each time an element is accessed (whether it previously existed in the cache or not), it is moved to the first position. Obviously, the last element will be the candidate for eviction. If an element does not exist in the cache, then it is either in the disk or a completely new element. In this case, the element is added in the first position with its updated value, but it must also be checked if the element exists in the disk (it is not possible to have the same element both in cache in the disk). The algorithm then asks the bloom filter. If the bloom filter returns false, then the elements with 100% probability does not exist in the disk. If the bloom filter returns true, then the algorithm searches for that element in the disk and removes it. If this case was a false positive, then the element won't be found in the disk. The alternative way would be to search for the element in the disk every time there was a cache update. By taking advantage of the bloom

filter, the algorithm knows if an element does not exist in the disk (it is completely new so far), and therefore these I/Os are avoided. Furthermore, the algorithm checks the cache capacity and if it is exceeded, then the candidate element for eviction is removed. Finally, the element is set on the bloom filter. It is worth mentioning that the algorithm accesses the disk via the LruDAO function which implements the DAO pattern. This will be explained better in 4.4.3.1

Algorithm 2 LRU – Get operation

```
1:  procedure GETLRU(k)
2:      if (k ∈ bloom filter) then
3:          return null
4:      else
5:          if (k ∈ cache ) then
6:              v = cache.remove(k)
7:              cache.put(0, k, v)
8:              return v
9:          else
10:             v = LruDAO.remove(k)
11:             if (v == null) then
12:                 return null
13:             else
14:                 cache.put(0, k, v)
15:                 call evictionLRU()
16:                 return v
```

Get is slightly more complex than put. The algorithm firstly asks the bloom filter to check if the element exists in the system. If the bloom filter returns false, then with 100% percent accuracy the element does not exist anywhere in the platform and therefore the expensive lookup on the disk is spared. If it returns true, the algorithm first checks the cache and then the disk. If the element is found in the cache, then it is returned and its temporal reference is updated. If the element is found on the disk, then it is removed from the disk, its value is returned, and is put back on the cache in order to update its temporal reference. If the element is not found in the cache or in disk, that means the decision of the bloom filter was a false positive, and hence the algorithm returns null.

Algorithm 3 LRU – Eviction

```
1: procedure EVICTIONLRU
2:     k = cache.get(cache.size-1)
3:     v = cache.remove(k)
4:     LruDAO.put(k, v)
```

The eviction algorithm is simple: The algorithm removes the element in the last position from the cache, and stores it in disk via the *LruDAO* API.

4.4.2.2 MRU

The MRU eviction policy is very similar to LRU. It uses the same techniques and the procedure is based on the same methodology

Algorithm 4 MRU – Put operation

```
1: procedure PUTMRU(k, v)
2:     if (k ∈ cache ) then
3:         cache.remove(k)
4:         cache.put(cache.size,k, v)
5:     else
6:         cache.put(cache.size(),k, v)
7:         if (k ∈ bloom filter ) then
8:             LruDAO.remove(k)
9:     if (cache.size == cache.maxCapacity ) then
10:        call evictionMRU()
11:    bloomfilter.put(k)
```

Algorithm 5 MRU – Get operation

```
1:  procedure getMRU(k)
2:      if (k ∈ bloom filter ) then
3:          return null
4:      else
5:          if (k ∈ cache ) then
6:              v = cache.remove(k)
7:              cache.put((cache.size,k, v)
8:              return v
9:          else
10:             v = LruDAO.remove(k)
11:             if (v == null) then
12:                 return null
13:             else
14:                 cache.put((cache.size(),k, v)
15:                 call evictionMRU()
16:                 return v
```

Algorithm 6 MRU – Eviction

```
1:  procedure evictionMRU
2:      k = cache.get(cache.size-1)
3:      v = cache.remove(k)
4:      LruDAO.put(k, v)
```

The eviction algorithm is the same as the LRU policy. The algorithm removes the element in the last position from the cache, and stores it in disk via the *LruDAO* API.

Taking all the above into consideration, by using the same methodology and by taking advantage of the flexibility of the system architecture, it is possible to seamlessly create 2 fundamental eviction policies which covers a plethora of use cases. Moreover, it is obvious that the usefulness of bloom filter can play a catalytic role in reducing disk I/Os. In many cases, the algorithms accessed the first layer in order to decide whether to take some actions or not. Finally, using the same template, more eviction strategies can be implemented, like for example FIFO which is based on the same meth odology. However, the thesis will focus on the LFU strategy, which is more complex and more likely to solve the 'lagging stream' problem. This is considered in the next section.

4.4.2.3 LFU

The functionality of the LFU follows a dual-purpose strategy:

- Store the frequent elements in main memory and the less frequent ones on disk. Each element has the form of a key-value entry. Furthermore, each key must be unique throughout the whole system. In other words, a particular key will either exist in the cache or in disk at any given time.
- When the system consumes the stream of itemsets, the hot and the cold elements must always be balanced. As it was stated in the first chapter, because it is assumed that the distribution of elements is unknown, a cold element which is stored in disk may become hot again in the future. This means that the particular element will be moved back to the cache, and take the position of the least frequent element (the potential candidate for eviction at that time). This is a complex process and requires additional checks to ensure that the above functionality is valid. The main data structure which holds the elements is a hash map. There is also a smaller structure which plays the role of the inverted index. Specifically, it indexes the elements by frequency and helps finding immediately the potential candidate for eviction. It is worth mentioning that the LFU algorithm does not only manage the elements in the cache. In many cases, the LFU procedures access the disk via the appropriate APIs as well as the bloom filter.

Since the LFU eviction policy must also take into account the frequency of each element somehow, an additional parameter should be added. To solve this problem, the value of the LFU cache is not a single data type. Instead, a composite Java object is used, which is comprised of 2 variables: the value itself of the itemset (which is a Java Generic type as it was previously specified) and the frequency of the itemset. Each time an itemset is accessed/updated, this change is reflected on the frequency variable. The frequency is a simple primitive integer variable (4 bytes). Consequently, the value of the LFU eviction strategy, contrary to the previous eviction policies is a Java object. The class is also implemented as serializable (to correctly approach the definition of the *POJO* class), because these objects must be stored to the database (or retrieved from it) without any serialization/deserialization errors. The *serialVersionUID* variable is defined in order to avoid errors during deserialization.

```
public class LFUCacheEntry<N> implements Serializable{

private static final long serialVersionUID = 42L;
```

```
private N data;
private int frequency;

// default constructor
public LFUCacheEntry(N data,int frequency){
this.data=data;
this.frequency=frequency;
}
```

LISTING 4.1: *The LFUCacheEntry object*

Furthermore, the data structure which holds the elements in main memory is the *Int2ObjectOpenHashMap*. As the official javadoc says, it's a "type-specific hash map with a fast, small-footprint implementation". The table is filled up to a specified load factor, and then doubled in size to accommodate new entries. If the table is emptied below one fourth of the load factor, it is halved in size; however, the table is never reduced to a size smaller than that at creation time: this approach makes it possible to create maps with a large capacity in which insertions and deletions do not cause immediately rehashing. Moreover, halving is not performed when deleting entries from an iterator, as it would interfere with the iteration process. This is particularly helpful, because deletions and re-insertions are very frequent in the context of the LFU policy. This hypothesis also agrees with the experimental results.

The main procedures in the LFU policy are the put and get operations. As far as the put operation is concerned, the input is the key-value tuple $(k, (v, f))$, where (v, f) is the *LFUCacheEntry* object consisted of the value v and the frequency f. From now on the potential candidate for eviction is referred as the *LFUelement*.

Algorithm 7 LFU – Put operation

```
1:  procedure PUTLFU((k, (v,f))
2:      if (cache.size == cache.maxCapacity) then
3:          if (k ∈ cache ) then
4:              f++
5:              cache.put(k, (v, f))
6:          else
7:              if (k ∉ bloom filter ) then
8:                  LfuDAO.put(k, (v, 1))
9:              else (v, f) = LfuDAO.remove(k)
10:                 if ((v, f) == null) then
11:                     LfuDAO.put(k, (v, 1))
12:                 else
13:                     if (f >= cache.getLFUelement.frequency) then
14:                         LfuDAO.put(LFUelement.k,(LFUelement.v,LFUelement.frequency)
15:                         f++
16:                         cache.put(k, (v, f))
17:                     else
18:                         f++
19:                         LfuDAO.put(k, (v, f))
20:     else if (cache.size<cache.maxCapacity) then
21:         if (k ∈ cache ) then
22:             f++
23:             cache.put(k, (v, f))
24:         else
25:             cache.put(k, (v, 1))
```

- **Line 3-5:** The algorithm checks if the element already exists in the cache. If so, the element gets updated with the new value and its frequency is incremented by 1.
- **Line 6-8**: The algorithm asks the bloom filter if the element exists in the system. If it doesn't, then algorithm returns null without doing a disk lookup because in that case the bloom filter is 100% correct. In other words the element is completely new, and therefore it is placed directly in the database with frequency 1 (the cache is full, and there is no way the element will surpass the frequency of the *LFUelement* - even in the most extreme case the *LFUelement* will have a frequency of 1).

- **Line 10-11:** Since the element does not exist in the disk, it means the bloom filter gave a false positive. As before, the element is completely new, and hence it is treated as described above.
- **Line 13-16:** The incoming element has equal or bigger frequency than the frequency of the LFUelement. Consequently, the *LFUelement* is moved the database and the incoming element takes its place with its frequency incremented.
- **Line 18-19:** The incoming element has less frequency than the frequency of the *LFUelement*. Therefore, the element is simply updated in the disk with its new value and its frequency incremented by 1.
- **Line 20:** In this case the cache is not full, and therefore no eviction will have taken place so far. To put it differently, the database will be empty.
- **Line 21-23:** The algorithm checks if the element already exists in the cache. If so, the element gets updated with the new value and its frequency is incremented by 1.The first step in this case is similar to the previous case.
- **Line 25:** The element does not exist in the cache at all because it is completely new. Therefore it is added in the cache with frequency 1.

COST ANALYSIS for Put operation

This subsection presents an analysis for costs that each operation induces. The main factor is the I/O costs (since all operations in memory have constant access and therefore their costs are negligible). The putLFU procedure is partitioned in many sub-cases, and each one has to be studied differently. This analysis will also help to understand the logic of the algorithm better. Moreover, all I/Os in the disk are assumed to have $O(1)$ complexity, since the underlying data structure in the database is a hash map (it functions like an index).

There are 4 different use cases:

Case A: The element is completely new in the system, since the bloom filter returns false with 100% accuracy. This process is consisted of 1 I/O

- In line 8, the element is simply added in the database with frequency 1. (1 write[1])

Case B: This is almost the same case as before. The decision of the bloom filter however is designated as a false positive. This process is consisted of 1 I/O

[1]Technically speaking, the writes don't immediately hit the disk because of the write buffer. However, for the sake of algorithmic complexity, it is said that this operation induces one write

- In line 11, the element is simply added in the database with frequency 1 (1 write)

Case C: The entry is located in the disk and it has to be updated with a new value. This process is consisted of 2 I/Os:

- In line 9, the entry is probed in order to get its value (1 read)
- In line 14, there is need for eviction. The *LFUelement* is moved to the disk (1 write)

Case D: The entry is located in the disk and it has to be updated with a new value. This process is consisted of 2 I/Os:

- In line 9 (same as before), the entry is probed in order to get its value (1 read)
- In line 19, the entry has its frequency updated in the disk (1 write)

The last 2 cases are the most expensive. In both of these cases, the algorithm has to locate the element in the disk in order to make a decision. On the other hand, if the cache is not full, then there is no need to access the disk since there is no chance of eviction.

Algorithm 8 LFU – Get operation

```
1:  procedure getLFU(k)
2:      if (k ∉ bloom filter ) then
3:          return null
4:      else
5:          if (k ∈ cache ) then
6:              f++
7:              cache.put(k, (v, f))
8:              return v
9:          else
10:             (v, f) = LfuDAO.remove(k)
11:             if ((v, f) == null) then
12:                 return null
13:             else
14:                 if (f >= cache.getLFUelement.frequency) then
15:                     LfuDAO.put(LFUelement.k,(LFUelement.v,LFUelement.frequency)
16:                     f++
17:                     cache.put(k, (v, f))
18:                     return v
19:                 else
20:                     f++
21:                     LfuDAO.put(k, (v, f))
22:                     return v
```

- **Line 2-3:** The algorithm probes the bloom filter in order to check if the element exists in the system. If not, the algorithm returns null without performing a disk access.
- **Line 5-8:** The algorithm checks if the element already exists in the cache. If so, the value of the element is returned and its frequency is incremented by 1.
- **Line 10:** The element is not present in the cache, which means either it is stored in the disk or the bloom filter gave a false positive.
- **Line 11-12:** In this case the output of the bloom filter was a false positive, and therefore the procedure returns null.
- **Line 14-18:** The requested element has equal or bigger frequency than the frequency of the *LFUelement*. Consequently, the *LFUelement* is moved the database and the incoming element takes its place with its frequency incremented. Finally,

its value is returned. This case shows that even a get operation can cause an eviction.

- **Line 20-22:** The incoming element has less frequency than the frequency of the *LFUelement*. Firstly, its value is returned. Then the element has its frequency incremented by 1.

COST ANALYSIS for Get operation

The cost is prominent in 2 cases:

Case A: The entry is located in the disk and the procedure has to retrieve it. This process is consisted of 2 I/Os:

- In line 10, the entry is probed in order to get its value (1 read) (1 read)
- In line 15, there is an eviction. The *LFUelement* is moved to the disk (1 write)

Case B: Same as before, the entry is located in the disk and the procedure has to retrieve it. This process is consisted of 2 I/Os:

- In line 10, the entry is probed in order to get its value
- In line 21, the entry has its frequency updated in the disk (1 write)

Taking all the above account, the worst case in this scenario is having 2 I/Os. The use of bloom filter prevents a disk access in Line 3.

4.4.3 Persistence Layer

The last layer in the system is responsible for handling the objects in the database. The overall architecture of the persistence layer is comprised of 2 design patterns: The *Abstract Factory* and the *Data Access Object* (DAO) pattern. Generally speaking, the first one is used to add more databases seamlessly, and the latter is used to exchange objects between the caching and the persistence layer without refactoring the code of the eviction policies.

Firstly, the section discusses the above design paradigms in depth and how they are used to abstract the persistence logic. The section concludes by describing the role of Connection Pooling module and how it contributes to the overall efficiency and performance of the system.

4.4.3.1 DAO Pattern

Data Access Object Pattern or DAO [34] pattern is used to separate low level data accessing API or operations from high level computation services. Nowadays, it is the de facto software paradigm for implementing a connection between the application logic and a database. Many web and application frameworks have pre-implemented the DAO layers for popular databases without the user having to write its own.

Following are the participants in this design paradigm which are presented visually in the UML diagram

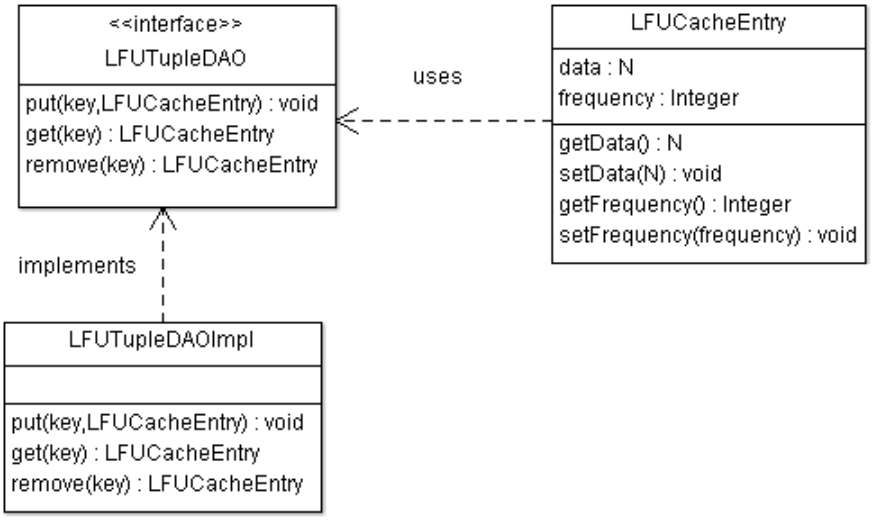

FIGURE 4.5: *The UML class depicting the DAO Pattern*

- **Data Access Object Interface** - This interface defines the standard operations to be performed on a model object. Sometimes, there is no need to use a model object if the data can be expressed by simpler variables. In the above figure, these classes are the *LFUTupleDAO* and *TupleDAO*. The first is used by the LFU cache, while the latter is used by the LRU and MRU caches. Their implementation is shown the following code snippets:

```
public interface TupleDAO<N> {

    public void put(Double id, N object);
    public N remove(Double id);
    public N get(Double key);

    public int getDBSize();
```

```
    public void printDB();
    public void printDBContentVerbose();

}
```

LISTING 4.2: *TupleDAO shows which operations a database should implement in order to be plugged in the caching layer.*

```
public interface LFUTupleDAO<N> {

    public void put(Double key, LFUCacheEntry<N> lfe);
    public LFUCacheEntry<N> remove(Double key);
    public LFUCacheEntry<N> get(Double key);

    public int getDBSize();
    public void printDB();
    public void printDBContentVerbose();

    public void close();

}
```

LISTING 4.3: *The only difference with the TupleDAO class is that this class uses the LFUCacheEntry as a model object to pass information between the caching and persistence layers.*

- **Data Access Object concrete class** - This class implements the above interface and is responsible to get data from a data source which can be database / XML or any other storage mechanism. In other words, it contains the database queries and the methods which are responsible for the database connection. The corresponding classes for this role are the *TupleDAOImpl* and *LFUTupleDAOImpl*
- **Model Object or Value Object** - This object is a simple *POJO* class containing get/set methods to store data retrieved using DAO class. The LFU cache uses the *LFUCacheEntry* which was described previously in the caching layer. Therefore, data between the LFU cache and the database are exchanged using this object. The LRU and MRU caches are simpler and don't require a new object to model their communication.

In general, the DAO paradigm is used to impose an API/interface upon the structure and the type of the data that the caching layer uses. The advantage of using data access objects is the relatively simple and rigorous separation between two important parts of an application that can but should not know anything of each other, and which can be expected to evolve frequently and independently. Changing business logic can rely on

the same DAO interface, while changes to persistence logic do not affect DAO clients as long as the interface remains correctly implemented. All details of storage are hidden from the rest of the application. Thus, possible changes to the persistence mechanism can be implemented by just modifying one DAO implementation while the rest of the application isn't affected. DAOs act as an intermediary between the application and the database. They move data back and forth between objects and database records. Based on this principle, the caching logic is separated from the persistence logic and does not depend on the query which characterizes a particular database. An example is shown the following pictures, where the cache removes an element from the database without knowing which underlying database or query is involved to perform this operation. As such, the database and the data access object can be chosen independently of the algorithm.

```
LFUCacheEntry<N> lce3= this.invalidateInDisk(key);

 if (lce3 == null) {
     lce3 = new LFUCacheEntry(object,1);
     prune(key, lce3);
     }
```

LISTING 4.4: *The algorithm calls the database implicitly without specifying the database or the model object which will transfer the data to the disk.*

```
public void prune(Double key,LFUCacheEntry<N>  lce) {
        lfutupledao.put(key, lce);
    }


public LFUCacheEntry<N> invalidateInDisk(Double key){
       LFUCacheEntry<N> lce = lfutupledao.remove(key);

       return lce;
    }
```

LISTING 4.5: *The access to the disk is specified independently according to which DAO is instantiated by the client.*

4.4.3.2 Abstract Factory Pattern

The Abstract Factory pattern [34] is in fact an extension and a more generic version of the Factory pattern that was previously used in the caching layer

Using this pattern, a framework is defined, which produces objects that follow a general pattern and at runtime this factory is paired with any concrete factory to produce

objects that follow the pattern of a certain class. In other words, the Abstract Factory is a super-factory which creates other factories (Factory of factories). In normal usage, the client software creates a concrete implementation of the abstract factory and then uses the generic interface of the factory to create the concrete objects that are part of the theme. The client doesn't know (or care) which concrete objects it gets from each of these internal factories, since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage and relies on object composition, as object creation is implemented in methods exposed in the factory interface. The advantages of using this design pattern are:

- Abstract Factory offers the interface for creating a family of related objects, without explicitly specifying their classes. The AbstractFactory class is the one that determines the actual type of the concrete object and creates it, but it returns an abstract pointer to the concrete object just created. This determines the behavior of the client that asks the factory to create an object of a certain abstract type and to return the abstract pointer to it, keeping the client from knowing anything about the actual creation of the object. The fact that the factory returns an abstract pointer to the created object means that the client doesn't have knowledge of the object's type. This implies that there is no need for including any class declarations relating to the concrete type, the client dealing at all times with the abstract type. The objects of the concrete type, created by the factory, are accessed by the client only through the abstract interface. This achieves the first objective of using a database without knowing its underlying mechanisms at the client code.

```
public abstract class DAOFactory<N> {

    public abstract TupleDAO<N>  getTupleDAO();
    public abstract LFUTupleDAO<N>  getLFUTupleDAO();

    public static DAOFactory getMapDBDAOFactoryUnPartitioned() {
        return new MapDBDAOFactory();
    }

}
```

LISTING 4.6: *If the user wants to use an alternative database other than MapDB, he must implement this class first and create his own factory. He must also take into account the DAOs which are defined here as abstract methods.*

- Use of this pattern makes it possible to interchange concrete implementations without changing the code that uses them, even at runtime. The second implication of

this way of creating objects is that when the adding new concrete types is needed, all the user has to do is modify the client code and make it use a different factory, which is far easier than instantiating a new type, which requires changing the code wherever a new object is created. Therefore, the user can switch to another database at the client level seamlessly without changing the implementation logic.

In this thesis, the factories are potential databases which can be plugged in the system. Each factory produces objects that handle a specific task. In this case, these objects are the DAOs that were mentioned previously. However, since the system expects different databases and cannot know beforehand how to embed them in the architecture, the solution is to encapsulate them as factories and abstract their underlying mechanisms. There is a super factory which is called DAOFactory (Figure 4.6). that imposes all specifications that a database/factory should have in order to be eligible to be used in the system. Moreover, the methods for creating new instances/factories are also present in the DAOFactory interface. Using the formal definition of the Abstract Factory pattern, these methods could have been defined in a different class. The reason for not doing this is because this variation gives the ability to the user to not only choose the database, but the DAO as well.

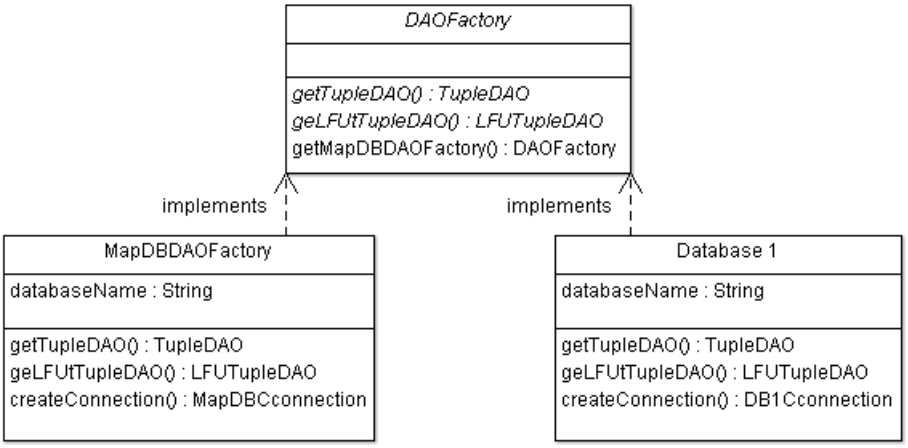

FIGURE 4.6: *UML class about Abstract Factory. Note that the specific details of how the database is managed (e.g. connection credentials) is implemented in the Factory class of each database independently*

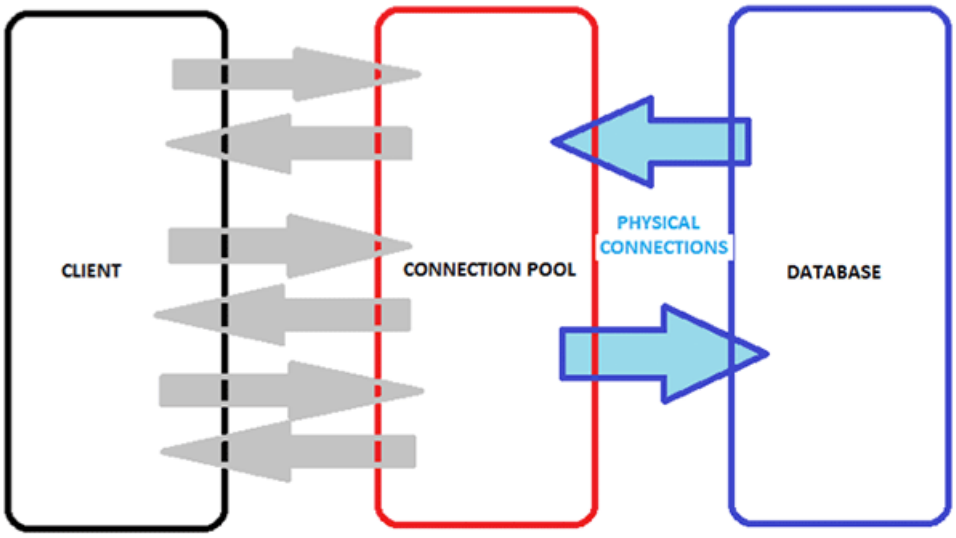

FIGURE 4.7: *The Connection Pool Module*

4.4.3.3 Connection Pooling

The final module in the database layer is the static Connection pool layer. A connection pool is a cache of database connections maintained in the database's memory so that the connections can be reused when the database receives future requests for data. To put it differently, it's like a thread pool, but the term applies to database connections. A simple depiction of this architecture is shown in Figure 4.7

Connection pools are used to enhance the performance of executing commands on a database. Opening and maintaining a database connection for each request, especially requests made to a dynamic database-driven application, is costly and wastes resources. In connection pooling, after a connection is created, it is placed in the pool and it is used over again so that a new connection does not have to be established. If all the connections are being used, a new connection is made and is added to the pool. Connection pooling also cuts down on the amount of time a request must wait to establish a connection to the database.

Apart from that, the connections in the pool are lazy initialized. Lazy Initialization is a performance optimization where the object creation is deferred until just before it is actually needed. One good example is to not create a database connection up front, but only just before it is necessary to get data from the database. The key reason for doing this is that it is possible to avoid creating the object completely if they will not be needed.

In the current implementation, the user simply specifies the number of the connections that will be available in the pool. This number should also take into consideration the

parameterization of the databases. Almost all databases have a variable which defines the maximum number of concurrent connections that are able to support. This variable should be in sync with the number of the available connections in the pool that the user has specified. Last but not least, these connections are lazily initialized as explained above and they are created only when they are requested.

Nevertheless, MapDB does not support Connection Pooling in its current version. Since it is intended to be used as an embedded database, it cannot accept more than one connection simultaneously. The main reason for that is because MapDB cannot accept concurrent write operations. However, MapDB's consistency assures that when an operation fails to complete, there will not be corrupted or incorrect data in the data storage. In the future, where MapDB will become a general purpose caching system (version 4), it will probably support Connection Pooling.

Regardless, the current system supports other databases as well, and therefore the Connection Pooling may seem useful. For example, all popular relational databases support connection pooling. Therefore, the current configuration which uses MapDB by default has the Connection Pooling component deactivated.

Chapter 5

Performance Evaluation

This chapter tests and evaluates the overall performance of the system. In order to do this, the system is put under heavy traffic with the purpose of monitoring its behavior. The experimental setup also involves tracking vital metrics such as time, cache hit ratio and so on. The data which will be consumed by the platform are created artificially by sampling from different distribution generators, each with its own parameters (they will be explained in depth). Both the best use case and the worst use case scenarios are considered. The Table 5.1 shows the hardware specifications of the hosting system where all the experiments took place.

5.1 Experimental Setup

BenchmarkConfiguration	
CPU	Intel® Core™ i7-4720HQ CPU @2.60GHz (8-cores with hyper-threading)
RAM	2x 8GB DDR3L @ 1600 MHz SDRAM
Storage	1TB HDD 7200 RPM
Network	Hypervisor Ethernet Adapter (at least Megabit)
Bare-metal OS	Ubuntu 15.04 Server

TABLE 5.1: *System configuration for all experiments*

The platform which is presented throughout the thesis is the final product of an implementing process which involved a lot experimentation and parameterization in order to achieve the desired result. The first version[2] of the system had less features and was less optimized. For example, it did not contain the bloom filter implementation and the write optimizations. Also, the LFU policy was much simpler: The keys did not return

[2]Actually, the very first version involved using MapDB for its caching properties as well. However, MapDB couldn't cope with the heavy load of the stream and 'crashed' after a few minutes of execution

back to the cache after their eviction. In other words, an element which becomes a frequent one subsequently (for a variety of reasons, such as the sudden change of the distribution in the stream), will not have the chance to return in memory. The only optimization that was used was the application of a fast compression technique- lzf [35] to further reduce the I/O traffic. The main idea behind the first version was to try a quick approach and check if the problem, which was formulated in the first chapter, can be solved without designing a complex architecture.

As far as the benchmark setup is concerned, it involved creating a stream of 1 million operations from 100000 distinct keys. The operations were partitioned in 3 categories i) get ii) update and iii) 50% get and 50% update. Moreover, each stream of elements was generated using a random generator (uniform distribution) and a zipfian generator with exponent s=1 and s=2. The candidate zipfian generator libraries that were considered to be used are from the Parallel Colt and the Apache Commons Maths package. Nevertheless, the latter was found to be much faster because it makes use of rejection-inversion sampling and also works for exponents less than 1. It does not require precalculating the CDF and keeping it in memory. Furthermore, the costs for generating one sample are constant and do not increase with the number of items. Each set of experiments was performed with i)75% keys in main memory and 25% on disk ii) 50% keys in main memory and 50% on disk and iii) 25% keys on main memory and 75% on disk. Moreover, the total time of execution as well as the time to complete the last 200001 requests (20%) is also measured. This was done in order to give the cache sufficient time to warm up. The time to generate the samples is deducted from the overall time metrics (although this cost is negligible and wouldn't hurt the results even if they were included). The same set of experiments with exactly the same parameters is applied to both the first and the second version of the platform and the experimental results are presented in Tables 5.2, 5.3, 5.4 and 5.5

5.2 The Naive Approach

#	Size (Mem/ disk)	Distribution	Action	Cache Hits	Cache Misses	Total Time (hh:mm)	Last 20% (minutes)
1	75/25	Uniform	Get	150330	49671	1:15	14
2			Update	150245	49756	1:18	15
3			Get/Update	150040	49961	1:16	15
4		Zipfian(s=1)	Get	169983	30018	0:13	2
5			Update	169499	30502	0:14	3
6			Get/Update	168830	31171	0:14	3
7	50/50	Uniform	Get	100168	99832	2:35	35
8			Update	100052	99948	2:37	34
9			Get/Update	100029	99972	2:34	33
10		Zipfian(s=1)	Get	167525	32476	0:35	7
11			Update	167411	32590	0:33	6
12			Get/Update	167504	32497	0:34	6
13	25/75	Uniform	Get	49926	150075	4:12	48
14			Update	50125	149876	4:00	45
15			Get/Update	49853	150148	4:06	44
16		Zipfian(s=1)	Get	164525	35476	0:59	11
17			Update	163501	36500	0:57	11
18			Get/Update	164627	35374	0:57	11

TABLE 5.2: *Experimental results of the first version*

In this approach, the initial results did not meet the expectations of a fast stream processor that achieves high throughput and low latency. In particular, the duration of the uniform benchmark is in the order of hours, and as the number of elements in the cache decreases, the performance deteriorates rapidly. Modern big data systems are able to process hundreds of thousands of elements in a few seconds. Obviously, the slowest benchmarks belong to the last category, where only the 25% of keys are in memory. However, there is still room for improvement, since the results that this version provided are not acceptable

5.3 Second Version

The second version of the system is the one which was presented throughout the second chapter. The experiments presented below are performed using all the aforementioned optimizations, as well as the improved version of the algorithm.

#	Size (Mem/ disk)	Distribution	Action	Cache Hits	Cache Misses	Total Time (seconds)	Last 20% (millisecs)
1	75/25	Zipfian(s=1)	Get	189290	10711	4	390
2			Update	188218	11783	4	410
3			Get/Update	189499	10502	4	395
4	50/50	Zipfian(s=1)	Get	181977	18024	5	542
5			Update	181411	18590	5	550
6			Get/Update	181504	18497	6	590
7	25/75	Zipfian(s=1)	Get	171381	28620	8	960
8			Update	171307	28694	8	910
9			Get/Update	171611	28390	8	922

TABLE 5.3: *Experimental results of the second version using zipfian distribution with s=1*

#	Size (Mem/ disk)	Distribution	Action	Cache Hits	Cache Misses	Total Time (seconds)	Last 20% (millisecs)
10	75/25	Zipfian(s=2)	Get	199833	168	2	117
11			Update	199828	173	2	102
12			Get/Update	199821	180	2	118
13	50/50	Zipfian(s=2)	Get	199796	205	2	130
14			Update	199787	214	2	135
15			Get/Update	199779	222	2	137
16	25/75	Zipfian(s=2)	Get	199640	361	2	140
17			Update	199641	360	2	189
18			Get/Update	199655	346	2	153

TABLE 5.4: *Experimental results of the second version using zipfian distribution with s=2*

The zipfian distribution is a favorable use case because the platform takes advantage of the skew in the data. In this situation, evictions (which are the costlier operations, as shown in the previous chapter) are minimized, and therefore the performance is very high. The highest average throughput of all experiments in the s=1 category is 250000 elements per second. On the other hand, the lowest throughput is observed when the system has 25% of all elements in memory (Experiments #7, #8, #9 in Table 3). In that case, the throughput, on average, is 125000 elements per second. In the s=2 category, where the skew is higher and the cache misses are low, the highest average throughput is 500000 elements per second. The fact that the most frequent elements are kept dynamically in main memory helps reducing the cache misses significantly. These results indicate that the platform can perform well in this scenario and can sustain a continuous stream without having its performance hurt.

#	Size (Mem/ disk)	Distribution	Action	Cache Hits	Cache Misses	Total Time (seconds)	Last 20% (millisecs)
19	75/25	Uniform	Get	149732	50269	8	1478
20			Update	150245	49756	9	1562
21			Get/Update	150040	49961	9	1742
22	50/50	Uniform	Get	99273	100208	14	3040
23			Update	100052	99948	14	2994
24			Get/Update	100029	99972	14	2957
25	25/75	Uniform	Get	49789	150212	21	4794
25			Update	49926	150075	23	4751
27			Get/Update	49853	150148	21	4614

TABLE 5.5: *Experimental results of the second version using uniform distribution*

The uniform case scenario is the most demanding one, because it negates the benefit of caching that the system provides. In other words, after the memory becomes full, evictions will happen at random and the cache hit ratio will be decreased. Nonetheless, the experimental results showed that performance did not decline significantly. The highest average throughput is 125000 elements per second (which is similar to the worst throughput of the zipfian case) and the lowest one is 43478 elements per second. Moreover, the total time and the last 20% metrics are not impacted significantly compared to the zipfian case. Even with only 25% elements in the cache, the stream is processed in less than half a minute.

5.4 Cache Hit Ratio

The primary objective of the LFU algorithm is to reduce cache misses as much as possible. This is mainly achieved by the balancing mechanism of the eviction policy which pushes the frequent elements in memory. A cache miss is expensive because in the worst case scenario, it may cause an eviction and a push back to the memory. This process may cost 2 disk lookups at worse. In other words, it is safe to emphasize that the cache hit ratio is a metric which is paramount for the cache efficiency. Figure 5.1 shows the cache hit ratio for different percentages of cached elements in memory. The cache hit ratio is measured during the last 20% of requests, where the cache is warmed up. All distributions are considered, including the Zipfian with different parameterizations. Zipfian1 and Zipfian2 are the cases where the exponent equals one and two respectively.

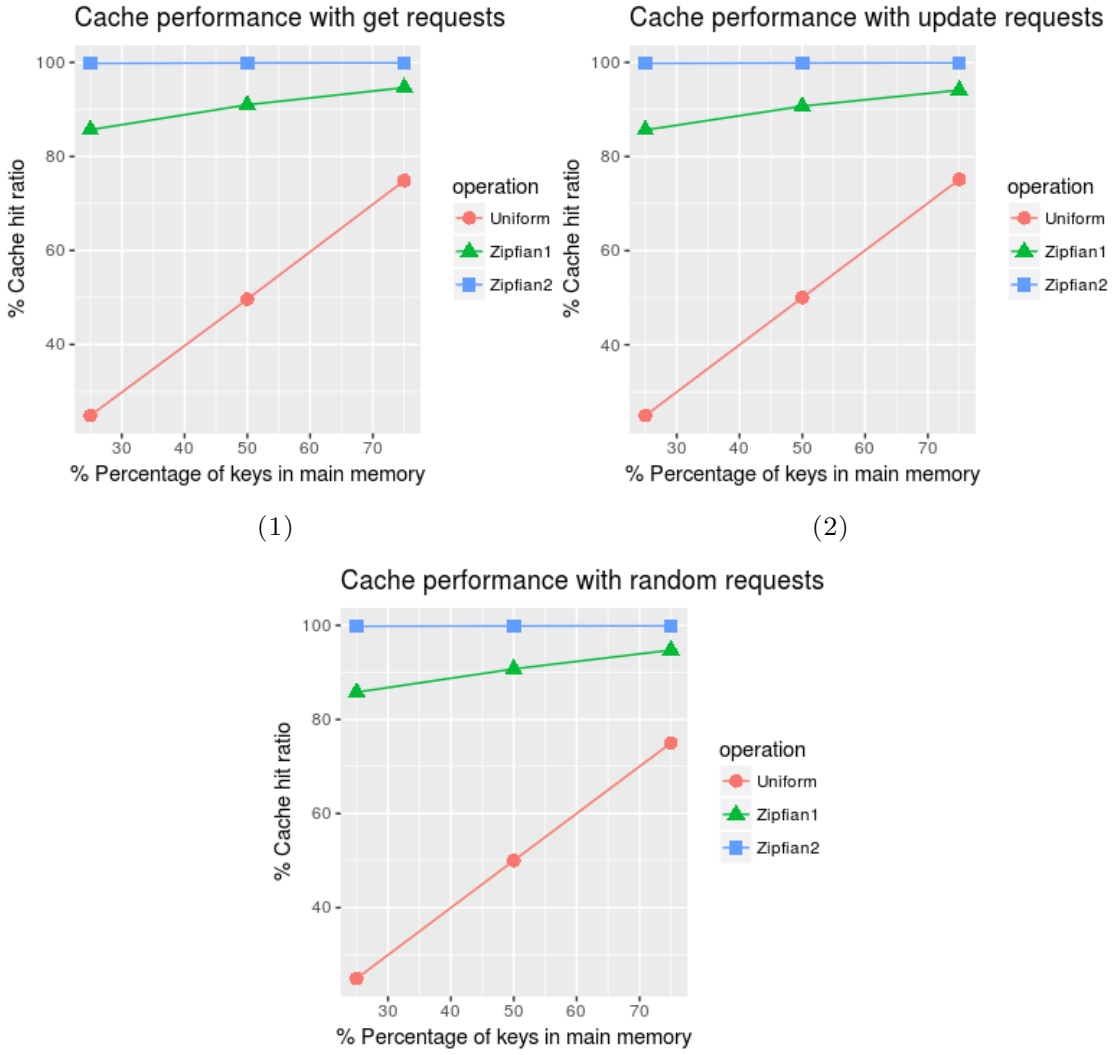

FIGURE 5.1: *Cache hit ratios with (1) get requests, (2) update requests, (3) random requests*

The figures are almost identical. These findings suggest that cache performance does not depend on the type of the request and stays almost the same whether they are get or update requests. This is very important because if there were differences between them the caching mechanism would not work properly. In the uniform case, the cache hit ratio is approximately linear. Intuitively, this makes sense because the cache hit ratio is equal to the number of elements in memory since the distribution is uniform. This also shows that the system scales smoothly as more memory is assigned to it. The zipfian cases are much better than the uniform, as expected. The case of zipfian with s=2 has extremely low cache misses, and in turn, less evictions. Consequently, the platform does not entail ingesting exclusively data from a highly skewed distribution (like the zipfian with s=2) to work well. A slightly skewed distribution can perform adequately as well. On the other hand, the uniform distribution can also provide satisfying results,

given enough memory (for example, the case where there are 50% percent of elements in main memory gives satisfactory results). However, the purpose of the platform is not to consume elements which are exclusively uniformly distributed, as this defeats the purpose of caching. It only shows that performance is not impacted significantly. If the consumed elements come from a varying distribution (which is a very frequent phenomenon in online learning), and that distribution later shifts towards the uniform case, then the system will not fall behind and the stream will not lag.

5.5 Comparison with Naive Approach

It is obvious that the second version has a significantly greater performance than the first one. All experiments are much faster, which means that the system can consume and process more elements during each unit of time (in other words, higher throughput). The most notable difference is in the experiments with the uniform distribution, which finish in a few seconds instead of hours. The zipfian experiments are also improved, because they finish in a few seconds instead of minutes (at least two orders of magnitude improvement in all zipfian cases). This observation is reflected visually in Figure 5.2

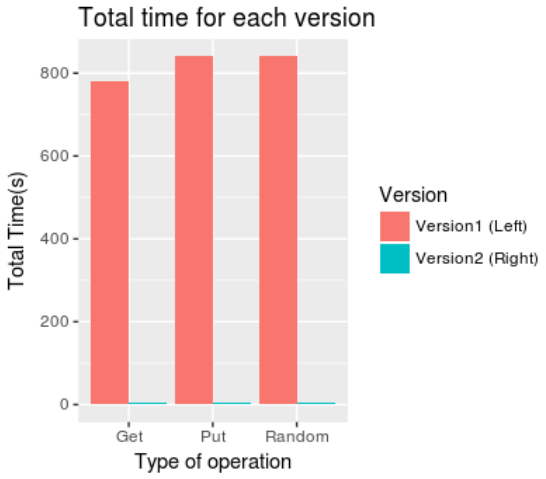

FIGURE 5.2: *Total time for each version when 75% of elements are in memory*

The figure demonstrates the total time to process 1 million requests for each version when 75% of keys are in memory. As expected, the difference is huge and the same goes for the rest of the experiments.

5.6 Large Scale Experiments

This section repeats the same type of experiments on the second version of the system, but on a larger scale. Instead of streaming 1 million elements, the new benchmarks stream 10, 50 and 100 million elements. The size of the disk storage in all cases is in the order of GBs. Similarly, the memory component consumes elements until it becomes full (16 GB approximately). To achieve this, the value of each element is defined as a hashmap, which is updated with new elements each time an update operation is required. In each category, the distinct elements that are used 1 and 2 million. Moreover, only the uniform and the zipfian distribution with s=1 are used to create samples. Finally, for the sake of brevity, only the cases where 25% and 75% of elements are in memory are considered. The types of requests are categorized into get and put operations, which occur randomly. The experimental results are shown in Table 5.6

#Updates	**#Distinct Keys**	**Size Mem/ disk**	**Distri-bution**	**Cache Hits**	**Cache Misses**	**Total Time (mm:ss)**	**Last 20% (millis)**
10000000	1000000	75/25	Uniform	1500846	499155	1:54	21218
			Zipfian	1895405	104596	00:36	4956
		25/75	Uniform	500806	1499195	4:04	45167
			Zipfian	1749262	250739	1:00	10218
	2000000	75/25	Uniform	1500711	499290	2:29	27696
			Zipfian	1839926	160075	00:37	5026
		25/75	Uniform	500220	1499781	5:19	60007
			Zipfian	1709702	290299	1:07	10707
50000000	1000000	75/25	Uniform	7502018	2497983	8:34	92554
			Zipfian	9757692	242309	2:36	20807
		25/75	Uniform	2499763	7500238	19:05	215810
			Zipfian	9004573	995428	4:31	46497
	2000000	75/25	Uniform	7500127	2499874	10:51	111315
			Zipfian	9705783	294218	3:08	26185
		25/75	Uniform	2501403	7498598	22:20	251051
			Zipfian	8988287	1011714	4:53	47174
100000000	1000000	75/25	Uniform	15000034	4999967	15:47	177793
			Zipfian	19549390	450611	5:03	45504
		25/75	Uniform	4999419	15000582	45:31	527995
			Zipfian	18044424	1955577	8:41	97259
	2000000	75/25	Uniform	14998029	5001972	19:32	209162
			Zipfian	19533832	466169	6:08	61407
		25/75	Uniform	5000542	14999459	45:12	511769
			Zipfian	18096228	1903773	10:36	116075

TABLE 5.6: *Experimental results on a larger scale*

Contrary to the previous benchmarks, in this case the total time has increased, especially in the uniform case. This occurs due to the fact that the number of distinct elements is bigger, and therefore the number of elements which exist on the disk is higher. This in turn means that the total amount of disk accesses and evictions are increased as well. Thus, the increase of total time was expected. In the zipfian experiments, the elements are consumed in less than five minutes, with the exception of benchmark #22 where the benchmark finished in approximately 6 minutes. The fastest one is benchmark #2, which consumes 10 million elements from a zipfian distribution and takes 36 seconds. The most significant result of these benchmarks is that the cache hit ratio stays the same in all cases and it is similar to the previous set of benchmarks (where the number of requests was 1 million). This is shown in Figure 5.3

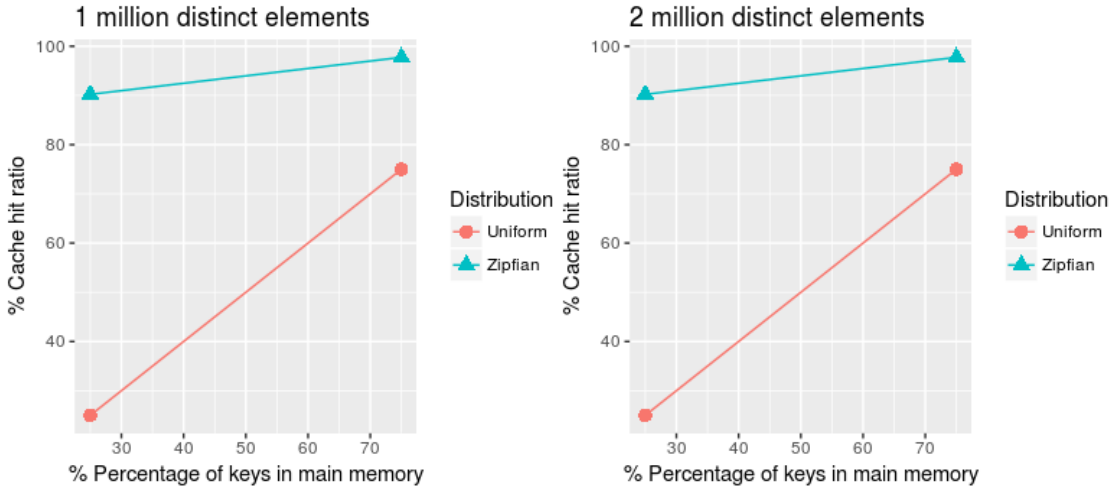

FIGURE 5.3: *Cache hit ratios with (1) one million distinct elements, (2) two million distinct elements*

These figures depict the case where the stream consists of 100 million requests. The other figures for the rest of the cases are identical, and therefore they are omitted for the sake of brevity. In general, the uniform case follows a linear pattern as before, and scales as more memory is allocated. The zipfian distribution performs well and achieves a high cache-hit ratio throughout the duration of the experiments

5.7 Memory Management

An integral part of a system which processes huge chunks of data continuously is memory management .These systems are more vulnerable to issues which can corrupt or

mishandle objects in memory, because they run indefinitely, and even a small miscalculation can crash the system after some time. For example, even a small memory leak can become an alarming issue in the long run. The JVM platform uses garbage collection, which automatically frees the storage for use by other processes and ensures that a program (or a process) using increasing amounts of pooled storage does not reach its quota. Nevertheless, Java is also susceptible to memory leaks and poor management of objects and classes. In most cases, the automation that the garbage collector provides makes the detection of memory issues more challenging. Besides, the garbage collection is the biggest advantage and disadvantage simultaneously when using the heap in JVM. Since the particular platform uses the default option of heap usage, additional checks should be made in order to make sure that all processes are stable and do not overuse memory. A good way to test a system for memory issues is to observe it for longer periods of time and check for anomalies in memory and the garbage collection calls. This is shown in Figure 5.4, which shows an example of memory management during the second set of experiments (Table 5.6) with uniform distribution.

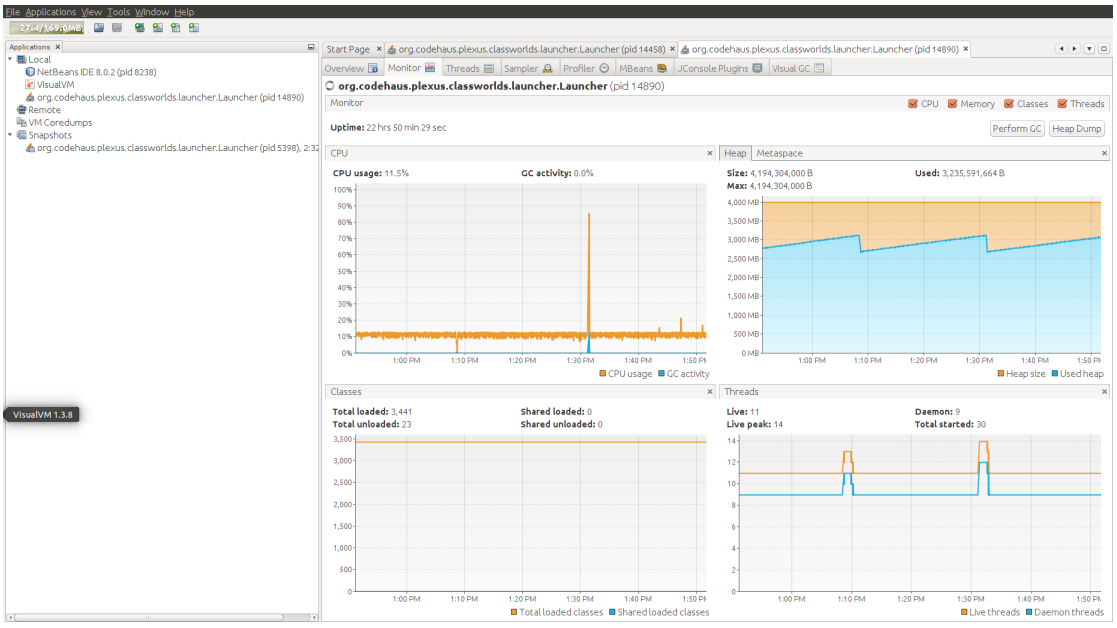

FIGURE 5.4: *VisualVM monitors CPU and memory usage, among other things*

The platform was tested by using the VisualVm [36] software, which facilitates the monitoring of JVM applications. The system was constantly receiving elements for approximately 24 hours and was being monitored during that time. As far as memory is concerned, the heap usage was stable over time and there was no sign of memory overuse or memory leaks. This was checked throughout the duration of all experiments

that were performed in this thesis. The garbage collector was functioning normally as well, without stressing the hardware resources. Last but not least, the CPU usage was also normal and rarely exceeded the 15% percentage. Taking all the above into consideration, the system did not face any issue in terms of memory management and CPU resources.

Chapter 6

Conclusion

6.1 Closing Remarks

This diploma thesis tackled the problem of limited memory that contemporary data processing systems need in order to increase their speed and achieve real or near-real time processing time. This is achieved with a proposed architecture which reverses the traditional storage hierarchy of disk-based systems. More specifically, the memory layer is used as the basic stateful storage and simultaneously functions as a cache with configurable element capacity by the user. The primary functionality of the memory layer is to keep hot elements and evict them to disk according to an eviction policy.

On the other hand, the disk layer acts as a cold storage for evicted data. In particular, data are moved from memory to disk in a transactionally-safe manner as the database grows in size. The opposite happens too: Cold data which become hot again are returned from disk back to memory. Consequently, there are two fundamental operations which work in tandem, eviction and restoration. These two operations are responsible for optimally managing the dynamic exchange of elements between the memory and disk layer with the purpose of constantly keeping the hot elements in memory at all times. The eviction strategies that are provided in the framework are variations of the LRU, MRU and LFU policies. They are similar to the traditional policies, however, they have been configured to accommodate the fact that elements may also return back to memory. Apart from that, the platform is equipped with additional optimizations, such as the bloom filter and the write buffer, which further reduce disk latencies as much as possible. All these components work in conjunction with the algorithms at the memory layer.

In general, the platform is highly scalable and it is supposed to work as a standalone service or a distributed one. Moreover, one of its biggest advantages is modularization

and elasticity. The persistence layer is not hardcoded and therefore users can plug the database of their choice into the system. The same is true for the memory layer, where users can integrate another eviction policy seamlessly without refactoring the core modules at the memory layer. The platform is configured to store both simple and complex data types, as was explained in Chapter 4.

Nevertheless, the primary purpose of the platform is to achieve sub-second latency and high throughput. This is feasible because hot elements are kept dynamically in memory at all times. Cold data are requested less frequently, thus minimizing the expensive disk lookups. This feature makes the platform suitable for real time processing, where rigorous demands should be met in order to achieve low latency. The most benefited applications from this system would be the stateful streaming applications that are required to store a complex state that is larger than the available memory. As a consequence, the platform can be utilized in conjunction with a stream processor, deployed in a real time configuration. Aside from real time processing, the platform can also be used in trivial cases which require some sort of fast Top-K implementation, like for example a web application which stores popular items.

Ultimately, of all the aforementioned eviction policies, the LFU strategy is the most versatile, because it takes advantage of skew in the data and it is not usually provided by most of the off-the-shelf in-memory systems. The experiments in Chapter 3 show that the system performed well even in the worst case scenario which involved consuming elements form a uniform distribution). This makes the system particularly useful in the following cases:

- The platform is not expected to work exclusively with skewed data, because as the experiments have shown, it can also handle the uniform distribution. The idea is that it leverages skewed data and increases its performance when they are present. If the consumed elements come from a varying distribution and that distribution later shifts towards the uniform case, then the system will not fall behind and the stream will not lag. In other words, this characteristic makes the platform particularly useful for online processing where the distribution of the incoming elements is unknown or changes over time. Should the uniformly distributed elements impacted significantly the overall performance, then the platform would not be eligible for real-time processing.

- If the platform consumes many streams simultaneously from different sources, then it could tolerate low skew or no skew at all in some of these streams. Obviously, the more skew in the data, the better the performance will be.

Generally speaking, the experiments showed more than satisfying results in every scenario. Performing this in a way that ensures the correctness of the algorithms without impacting performance is not a trivial task and requires significant effort.

6.2 Future Work

The past few years, software applications, especially those which run on a large scale, make use of more sophisticated deployments. Standalone installations or virtual machines have started becoming outdated since containers have emerged. In particular, containers facilitate deployments and provide more flexibility while simultaneously they require less hardware resources. By deploying the application platform and its dependencies in a container, differences in OS distributions and underlying infrastructure are abstracted away.

The platform which is presented in this thesis is also able reap the benefits of containerization. Doing so will facilitate its deployment as a distributed application. Also, the platform can be bundled more easily with other systems such as stream processors and application servers. An even better approach would be to configure the platform with a framework which automates deployment, scaling and management of containerized applications like *Kubernetes*.

Bibliography

[1] Data Artisans. *Apache Flink: user review in 2017.* "https://data-artisans.com/blog/apache-flink-user-survey-2017-recap".

[2] Hao Zhang et al. "Efficient In-memory Data Management: An Analysis". In: *Proceedings of the VLDB Endowment* 7.10 (June 2014), pp. 833–836.

[3] Kian-Lee Tan et al. "In-memory Databases: Challenges and Opportunities From Software and Hardware Perspectives". In: *ACM SIGMOD Record* 44.2 (Aug. 2015), pp. 35–40.

[4] Graham Cormode and Marios Hadjieleftheriou. "Finding Frequent Items in Data Streams". In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pp. 1530–1541.

[5] Gurmeet Singh Manku and Rajeev Motwani. "Approximate Frequency Counts over Data Streams". In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB '02. Hong Kong, China: VLDB Endowment, 2002, pp. 346–357.

[6] Graham Cormode and S. Muthukrishnan. "An Improved Data Stream Summary: The Count-min Sketch and Its Applications". In: *Journal of Algorithms* 55.1 (Apr. 2005), pp. 58–75.

[7] Hans Ulrich Obrist. *The Father of Long Tails. An interview with Benoît Mandelbrot.*

[8] Stephen Fagan and Ramazan GenÃ§ay. "An Introduction to Textual Econometrics". In: *Handbook of empirical economics and finance* (Dec. 2010), pp. 133–154.

[9] Gene M. Amdahl. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485.

[10] Nimrod Megiddo and Dharmendra Modha. "Proceedings of the 2Nd USENIX Conference on File and Storage Technologies". In: San Francisco, CA: USENIX Association, 2003, pp. 115–130.

[11] Muhammad Bilal and Shin-Gak Kang. *A Cache Management Scheme for Efficient Content Eviction and Replication in Cache Networks*. 2017.

[12] P. Jayarekha and T. R. Gopalakrishnan Nair. *An Adaptive Dynamic Replacement Approach for a Multicast based Popularity Aware Prefix Cache Memory System*. 2010.

[13] Hong-Tai Chou and David J. DeWitt. "An Evaluation of Buffer Management Strategies for Relational Database Systems". In: *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*. VLDB '85. Stockholm, Sweden: VLDB Endowment, 1985, pp. 127–141.

[14] Shaul Dar et al. "Semantic Data Caching and Replacement". In: *Proceedings of the 22th International Conference on Very Large Data Bases*. VLDB '96. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 330–341.

[15] Tim Kraska et al. *The Case for Learned Index Structures*. 2017.

[16] Chen Zhong, Mustafa Cenk Gursoy, and Senem Velipasalar. *A Deep Reinforcement Learning-Based Framework for Content Caching*. 2017.

[17] Thodoris Lykouris and Sergei Vassilvitskii. *Competitive caching with machine learned advice*. 2018.

[18] Redis. *An open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker*.

[19] DB-Engines. *Ranking of Key-value Stores*. "https://db-engines.com/en/ranking/key-value+store".

[20] Apache Ignite. *The Apache software foundation*.

[21] Ehcache. *An open source, standards-based cache that boosts performance and simplifies scalability*.

[22] MapDB. *A hybrid between java collection framework and embedded database engine*.

[23] LevelDB. *A light-weight, single-purpose library for persistence with bindings to many platforms*.

[24] Joshua Bloch. *Effective Java 3rd Edition*. 2018.

[25] Gregor Kiczales et al. "Aspect-Oriented Programming". In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)* (1997).

[26] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (July 1970), pp. 422–426.

[27] Deke Guo et al. "False Negative Problem of Counting Bloom Filter". In: *IEEE Transactions on Knowledge and Data Engineering* 22.5 (May 2010), pp. 651–664.

[28] GoogleBigQuery. *A fast, highly scalable, cost-effective and fully-managed enterprise data warehouse for analytics at any scale.*

[29] Adam Horvath. *MurMurHash3, an ultra-fast hash algorithm for C # / .NET.* 2012.

[30] Guava. *An open source, Java-based library developed by Google.*

[31] Fabio Grandi. "On the analysis of Bloom filters". In: 129 (Sept. 2017).

[32] Apache Cassandra. *A free and open-source distributed wide column store NoSQL database management system.*

[33] RocksDB. *An embeddable persistent key-value store for fast storage.*

[34] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1994.

[35] Marc A Lehmann. *LZF-compress is a Java library for encoding and decoding data in LZF format.*

[36] Visual VM. *A visual tool integrating commandline JDK tools and lightweight profiling capabilities.*