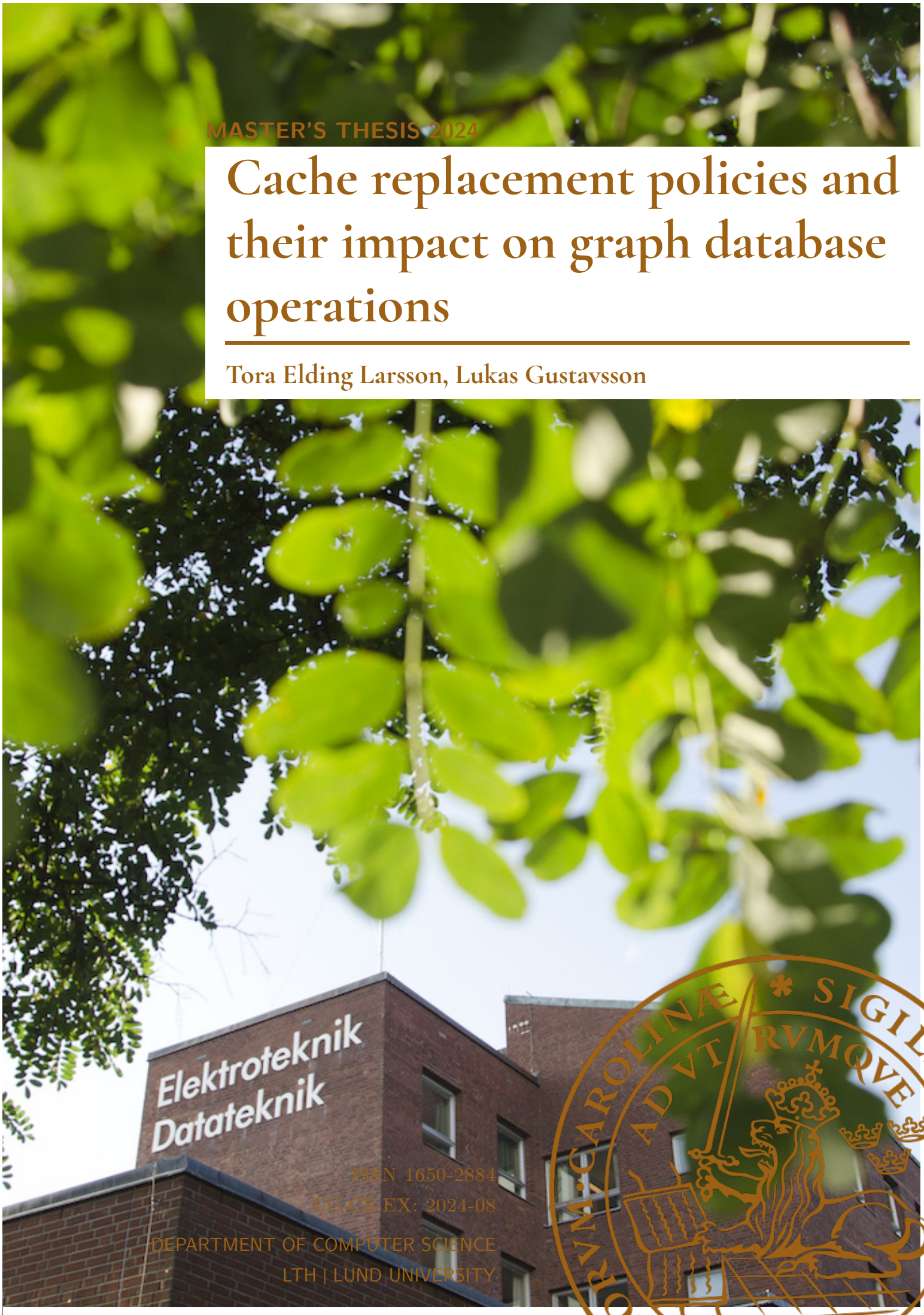

MASTER'S THESIS 2024

Cache replacement policies and their impact on graph database operations

Tora Elding Larsson, Lukas Gustavsson

ISSN 1650-2884
LU-CS-EX: 2024-08

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-08

Cache replacement policies and their impact on graph database operations

Cacheutbytespolicyer och deras inverkan på grafdatabasoperationer

Tora Elding Larsson, Lukas Gustavsson

Cache replacement policies and their impact on graph database operations

(Performance considerations of the Neo4j database with regards to caching)

Tora Elding Larsson
tora.eldinglarsson@gmail.com

Lukas Gustavsson
l.gustavsson9801@gmail.com

February 8, 2024

Master's thesis work carried out at Neo4j, Inc.

Supervisors: Anton Klarén, anton.klaren@neo4j.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

Examiner: Michael Doggett, michael.doggett@cs.lth.se

Abstract

In this master thesis project, the page caching strategy of the Neo4j database is researched and attempted to be improved. Focusing on the eviction protocol of the page cache, several different algorithms are evaluated in both experimental prototyping using Python, and in the Neo4j database kernel. Using the measurements of the prototypes and the results of the Neo4j benchmarks conclude that the current page replacement policy is hard to beat with a different strategy. However, modifying the current page replacement policy by using a global instead of thread-local data structure and tuning parameters increased the hit rate and throughput. Furthermore, the measurements on the different implementations showed that the hit rate can be increased at the cost of some overhead, but implementing a complicated algorithm quickly increases the overhead and might decrease the throughput enough to make the algorithm ineffective.

Keywords: software cache, replacement policies, graph database, performance, hit rate, LRU, CLOCK

Acknowledgements

We would like to thank Jonas Skeppstedt, our supervisor at LTH, for answering all our questions and having helpful discussions with us whenever we needed to. Thank you for all the knowledge we have received from your courses at LTH.

We would also like to thank Anton Klarén and the rest of the kernel team at Neo4j for answering all our questions with patience and eagerness. Thank you for inviting us to your weekly games, you have both helped us with our thesis and made us feel like part of the team at Neo4j.

A special thank you to Simon Priisalu at Neo4j for helping us with benchmarking measurements and answering an endless amount of questions.

Contents

Chapter 1
Introduction

Graph databases are a field that has proved to be very efficient for certain operations [15], and Neo4j's distribution is one of the commonly used graph databases [15]. The database has several demands, such as maintaining a consistent state and being capable of handling multiple transactions in parallel, and everything whilst maintaining a good performance. One of the things that slows down the performance in a Neo4j database is when a page has to be accessed from the disk and not from the cache [11]. Therefore, minimizing the amount of faults, whilst still maintaining a small overhead in memory and minimizing locking, is desirable.

Cache replacement policies are a field that has been researched a lot since it could minimize the amount of page faults. However, most research presented in this thesis is on hardware caches with only one thread and not on multi-threaded software caches. Neo4j has solved the problem with the replacement policy by using a simple implementation with a minimized amount of locking and overhead, realized with an algorithm that takes very simple decisions on what to evict [9].

This thesis investigates the current cache and compares the performance to implementations with more overhead and more locking, but with smarter decisions on what to evict. It also compares the current cache to a random cache replacement policy to see how much the algorithm improves the cache hit ratio.

The thesis presents the background and describes the research questions in chapter 2, gives an overview of the method used in chapter 3, presents the results from the investigation in chapter 4 and discusses the results in relation to the research questions in chapter 5. It ends with chapter 6, giving a conclusion of what have been found during the work of the thesis and presenting some possible future work that can be done.

Chapter 2
Background

2.1 Problem statement

The thesis investigates the cache replacement policy used by Neo4j to see if it could be improved. The problem was to increase the hit rate without affecting the latency of the program negatively.

2.1.1 Research questions

To limit the extent of the thesis work, three main research questions were formulated and investigated:

RQ1. What improvements can be made to the current cache replacement policy?

RQ2. How much is there to gain by using a more effective cache replacement policy?

RQ3. How big can the implementation overhead be before the implementation costs more than it gives?

2.2 Distribution of work

For the most part, everything has been worked on by both students. Tora is the only contributor to graphics in the background chapter, and Lukas is the only contributor to the charts in the result chapter. Tora has contributed to the majority of the background chapter and Lukas has contributed to the majority of the results chapter. The rest of the work can be assumed to have been equally divided.

2.3 Cache memories

A cache is a small volatile memory that is fast to access but limited in size. Since the latency of writing and reading memory is much higher than the latency of executing instructions, cache memories are of high importance [4]. Computers have a hierarchy of hardware memories, going from large but slow memories to fast but small ones [10]. Figure 2.1 shows the memory hierarchy in computers. Caches that are high up in the hierarchy are smaller and faster than the ones further down.

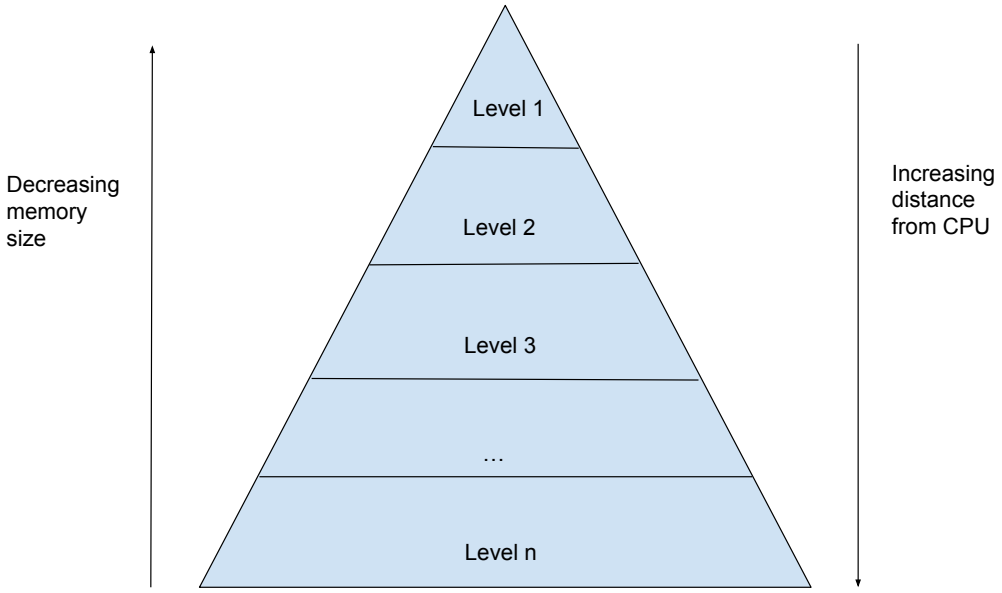

Figure 2.1: Figure showing how the memory in a computer is hierarchically structured, drawn from image in [10]

When discussing caches and their performance, there are some terminology worth knowing:

- Hit: When a wanted page is in the cache.
- Miss: When a wanted page is not in the cache.
- Fault: When a page that is not in the cache is loaded into the cache from memory.
- Evict: When a page that is in the cache is thrown out of the cache - often to make room for something else.

Most often when talking about cache memories, the discussion goes around hardware caches, as has been done so far. However, in this thesis, a software implemented cache is investigated.

2.3.1 Software implemented cache

Neo4j uses a software implemented cache to decrease the latency of operations for the database. What this means is that during program execution, pages are loaded from hard drive into memory allocated by the program. This makes it much faster to access the data in the pages. The entire implementation, how to handle misses, evictions, faults, etc. are all implemented in Java classes and interfaces. This means that it is easier to configure and adapt compared to a hardware cache, but it still has some limitations. For program execution to run smoothly and effectively, the cache needs to be limited in size and the operations on it can not have too high latency, since this leads to a slower execution time.

The software cache implemented by Neo4j is shared among multiple threads and therefore needs to be thread safe. This places further demands on the implementation, which needs to make sure that the cache is in a coherent state while still not introducing too much overhead in the form of locking.

2.3.2 Cache optimizations

Caches can be implemented in different ways to optimize their performance. One way to optimize the cache is to make sure that the data that the program needs is almost always in the cache, since if it is not, it must be collected from memory with much higher latency [10]. To measure how well a cache performs on this matter, the *hit rate* is usually investigated. The hit rate is calculated according to equation 2.1, where s is the number of memory requests resulting in a hit and r is the total number of requests made to the cache [4].

$$\frac{s}{r} \tag{2.1}$$

One way to improve the hit rate would be to increase the size of the cache [4]. However, for hardware caches, the faster memories are often more expensive to implement than the larger ones, and caches therefore have a limit to how large they can be before the cost of manufacturing them is unfeasible [10]. In software caches it is not the hardware implementation of a cache that is the expensive part, but since there is a limited amount of memory available to allocate memory for a program execution, the size of the cache still must be limited. It is therefore possible to assume that all data might not fit in the cache and that during execution, old data may have to be evicted from the cache to make room for newer.

This leads to another way of improving the hit rate; to optimize the algorithm which decides what to evict from the cache when it is full [4]. If something is thrown out that needs to be accessed again soon, it has to be faulted in from memory which is a timely operation. Therefore, the algorithm should be optimized so that it evicts data that is not going to be accessed shortly. This way of increasing the hit rate is what this thesis investigates.

The problem with optimizing this part of the cache is that it is impossible to tell what the program will need in the future [4]. Several different *cache replacement policies*, which use

the current behavior of the cache to predict what might happen exist, and some of these are presented in the next section.

2.4 Cache replacement policies

As mentioned in the previous section, cache replacement policies try to predict the future to make good decisions on what to evict from the cache when it is full. There are several different approaches to how these algorithms can be implemented, but all of them look at the history or the current situation of the cache to try and predict how it will behave in the future. Some policies prioritize pages when they are faulted into cache. They do the prioritizing based on the access history of the pages. These policies are called *fine-grained policies* [4]. Another group of policies prioritize pages during their time in the cache based on what happens to them once they are faulted into cache. These policies are called *course-grained policies* [4]. The following sections present some different cache replacement policies important for this thesis work.

2.4.1 Béladý

Béladý's algorithm for cache replacement is also known as Béladý's optimal algorithm since it gives the optimal replacement for a cache [14]. The algorithm decides what to evict based on what page is accessed farthest in the future [4][14]. Figure 2.2 shows an example of a full cache containing three pages, and a trace of pages that are about to be accessed. Béladý's algorithm looks through the trace and decides to throw out page 1 since it is the page that is accessed farthest into the future.

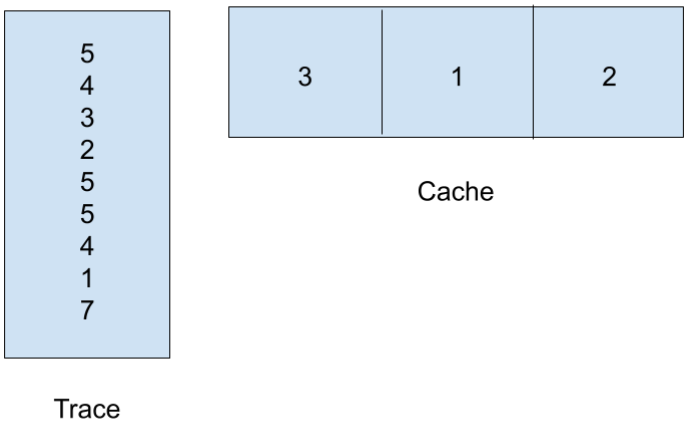

Figure 2.2: Figure showing a full cache containing three pages, and a trace of pages that are about to be accessed.

The problem with Béladý's algorithm is that it is impossible to implement since it decides based on actions that will happen in the future [4][14].

2.4.2 LRU

LRU stands for Least Recently Used and is an algorithm that is *recency-based*. This means that it decides what to evict from the cache based on how recently it was accessed [4]. It is a simple algorithm and one of the most commonly used recency-based algorithms. It keeps track of how recently pages were accessed and evicts the least recently used page [4]. For it to work effectively, the access pattern in the cache has to make use of temporal locality, i.e. if a page has been accessed it will be accessed soon again [4]. However, for other patterns, LRU performs badly.

One of these problematic examples is scans, where many pages are accessed once, possibly evicting important residing pages in cache [5]. Since LRU only evicts the least recently used, the pages accessed during the scan will not be evicted even though they might not get any more hits during program execution.

Another example of an access pattern where LRU performs badly is in loops, where data is accessed several times, but the recency distance is too large for it to remain in the cache between each iteration [5].

2.4.3 LFU

LFU stands for Least Frequently Used and is an algorithm which is *frequency-based*. Instead of deciding what to evict from the cache based on when pages were last accessed, as with LRU, it bases the decision on how many times a page has been accessed [4]. In the LFU algorithm, this means that the page that has been accessed the fewest times is evicted. The simplest way of implementing such an algorithm is to simply have a counter for each page and evict the page with the lowest count [4].

This algorithm performs well on patterns where pages are accessed several times but with longer intervals. It also solves the problem the LRU algorithm has with scans [4] since the pages faulted in will not be prioritized over pages residing in the cache that have been frequently accessed.

However, as with LRU, this algorithm also has some disadvantages. One example is that the algorithm prioritizes pages that have been accessed a lot previously over pages that will be accessed a lot in the future. When a new page that will be frequently accessed is faulted in for the first time it will be chosen to be evicted over pages that might not be accessed in the future simply because they have a history of being accessed a lot [4].

2.4.4 CLOCK

CLOCK is an algorithm that is a very good approximation of LRU [5]. The cache structure is maintained as a circular list, and each page has a reference bit that is set when it is accessed. A clock arm is used to go through the pages in the cache. In case of an eviction, the page the clock arm points to is evicted if the reference bit is set to 0. If the bit is set to 1 it is set to 0 and the arm goes to the next page and does the same procedure. This continues until a page is evicted [5].

Figure 2.3 shows the clock structure. The arm points to a page with the reference bit set to 1. If an eviction is happening the bit is set to 0 and the arm is moved to the next page which is evicted since its reference bit is set to 0.

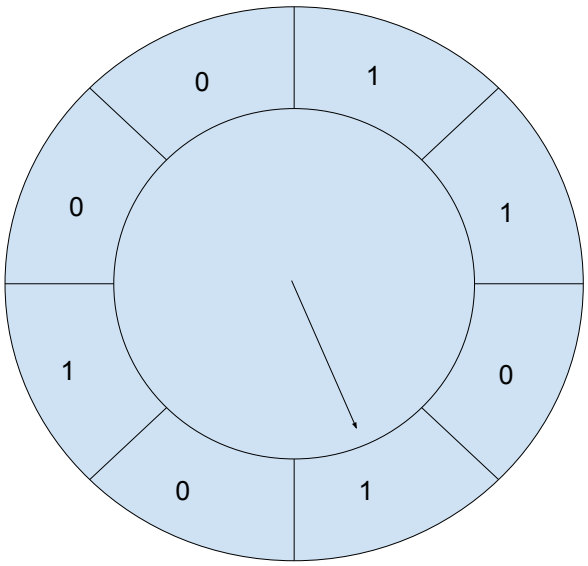

Figure 2.3: The structure maintained by the CLOCK algorithm. The cache is seen as a circular structure, and every page has a reference bit. The clock arm points to the current page.

2.4.5 LIRS

LIRS stands for Low Inter-reference Recency Set and is an algorithm that was created to try and remove the disadvantages with LRU [6]. The difference between LIRS compared to LRU is that it evicts based on reuse distance rather than recency. To be able to do this it keeps track of two sets, LIR (Low Inter-reference Recency) and HIR (High Inter-reference Recency). LIR contains cache pages that currently reside in the cache and have a low reuse distance, indicating that they are accessed more frequently than pages with a high reuse distance. HIR contains pages that both reside in the cache and previously have resided in the cache but have been evicted. These pages have a high reuse distance. When a page has to be evicted, a page that is in the HIR set and resides in the cache is evicted [6]. This means that a page with higher recency can be evicted simply because it is not accessed particularly often.

Since a page might go from being accessed infrequently to being accessed frequently, or the opposite way around, there must be a way for a page to switch between the different sets. Therefore the algorithm makes it possible for a HIR page that suddenly gets a very small reuse distance to switch with a LIR page with less reuse distance [6].

2.4.6 CLOCK-Pro

CLOCK-Pro is an algorithm that aims to bring the LIRS functionality into CLOCK [5]. The implementation is based on CLOCK but additionally keeps track of hot- and cold pages and some information about recently evicted pages.

Instead of only having one clock arm as in the CLOCK algorithm, CLOCK-Pro has three arms; a cold-, hot- and test arm. The residing pages in the cache are divided into hot pages, which are accessed frequently, and cold pages, which are accessed infrequently. In addition to this, the algorithm keeps track of non-resident pages, i.e. pages that have been recently evicted. The set with non-resident pages is never larger than the size of the cache [5].

As can be seen in figure 2.4, the three arms represent the hot-, cold- and test arms. The slots filled with red numbers contain hot pages, and the numbers represent their reference bit. The empty slots contain non-resident pages and the slots filled with black numbers contain cold pages.

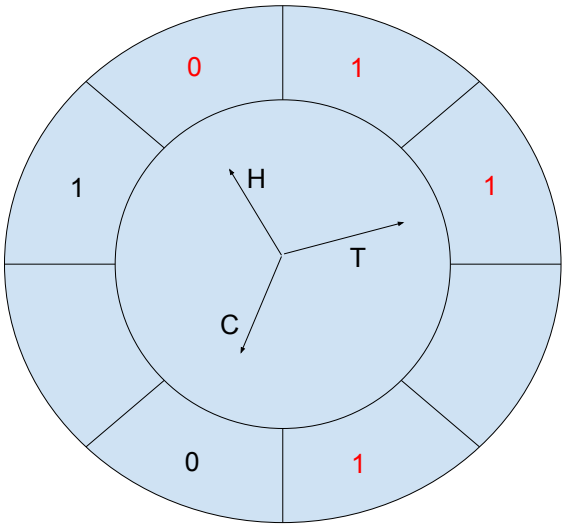

Figure 2.4: Structure of the CLOCK-Pro algorithm. The slots containing red numbers are hot pages, the empty slots are non-resident pages, and the slots containing black numbers are cold pages. The clock arms represent the hot-, cold- and test arm.

When a page is faulted into cache, it starts as a cold page and begins its test period. If it is accessed during its test period, the page turns into a hot page. If it is evicted during its test period, it is saved as a non-resident page. A non-resident page can either be faulted in again, in which case it is turned into a hot page, or it is removed because it has not been faulted in again during its test period. If a cold page is evicted outside of its test period it is not saved as a non-resident page. When the cache contains too many hot pages, a hot page that has its reference bit set to 0 is turned into a cold page.

Song Jiang, Feng Chen and Xiadong Zhang add another layer to their CLOCK-Pro algorithm by making it adaptive. The algorithm will then adapt to the pattern of the cache, and change how many hot pages can reside in the cache depending on how many hot pages are needed [5]. They propose that if there is a hit on a cold page during its test period, the capacity of cold pages should be increased by one, and if a cold page terminates its test period

without being accessed, the capacity of cold pages should be decremented by 1. This means that when the access pattern is LRU friendly and there is not a set of hot pages which are accessed very frequently, the set of hot pages will be small and the algorithm will behave like a CLOCK algorithm. However, if there are some pages that have a much higher frequency in their accesses, the cache will make room for these pages.

CLOCK-Pro has some limitations which occur when there is a large number of hits in non-resident pages simultaneously as a large number of cold pages are terminated [8].

2.4.7 CLOCK-Pro+

Li Cong suggests an algorithm based on CLOCK-Pro, with an improved way of adapting the cache [8] which should help with the problem mentioned above. Li Cong reasons that if a non-resident page was accessed, the miss could have been prevented if the capacity of cold pages were increased by 1. This means that the probability of getting a miss would decrease by $\frac{1}{C_n}$, where C_n is the number of non-resident pages. The same goes for when a cold page that previously was a hot page is referenced. In this case, the cold page could have stayed a hot page had the capacity of hot pages been increased by 1. The probability of it not turning into a cold page would decrease by $\frac{1}{C_d}$ where C_d is the number of resident cold pages that have previously been hot pages [8].

With this in mind, Li Cong suggests that if a non-resident page is accessed the capacity of cold pages should be increased by $min\{1, \frac{C_d}{C_n}\}$ and if a cold page which has previously been a hot page is accessed, the capacity of hot pages should be increased by $min\{1, \frac{C_d}{C_n}\}$ [8].

2.5 The Muninn Page Cache

The page cache implementation currently used by Neo4j is implemented in Java and is called the Muninn Page Cache [9]. It makes sure the database is not only available in memory but also on disk, and solves the issue of the database running out of working memory by freeing some of that memory by writing it back to disk.

2.5.1 Structure

The Muninn Page Cache consists of an array with slots for pages, the size of which is configurable by the user at database setup. The currently implemented cache replacement policy is a CLOCK algorithm [9], with some tweaks compared to the one described in 2.4.4. As was written in 2.4.4, this means that it is an approximation of the LRU algorithm, described in 2.4.2. One big difference is that the algorithm does not set only one reference bit, but counts references up to 4 [9]. There are however bits enough to count the references to 7, but the limit is currently set to 4. Another difference is that it sets the reference bit to 1 instead of 0 when a page is faulted into cache, and evicts a page with reference bit set to 0 or 1 and not only 0.

2.5.2 Eviction

Eviction happens in several ways in the Muninn Page Cache. Different types of eviction run in the Neo4j kernel to cater to different states of the cache. The following sections describe the two main eviction processes in the page cache. There are some other forms of eviction in the Muninn Page Cache that have to do with files closing or other external forces forcing eviction, but the focus in this work is on the most usual eviction procedures.

Background Eviction

If the page cache reaches a threshold of page occupancy, the background evictor thread starts. The background evictor thread is the one simulating a CLOCK algorithm. The thread loops through all the pages in the cache and decrements the reference by 1. If a page has its reference bit set to 1 or 0 before it is decremented, that page is chosen to be evicted [9]. However, this page might still be spared because the cache is shared between multiple threads, as was mentioned in 2.3.1. This means another thread might have an exclusive lock to the page, making it impossible for us to evict it. In that case, the reference count is decremented and the arm moves on. The reason eviction is beneficial before the cache is full is that a pool of free pages is kept. This means there is no compounding cost to the database query consisting of making an eviction decision and the cost of flushing the dirty data to disk.

Cooperative Eviction

If there are no free pages available produced by the background evictor thread, a state called cooperative eviction is entered. Here the thread that needs a slot in the cache for it to fault in a page makes room for it itself by running the same algorithm as the background thread, starting at a random page and looping through the list of pages [9].

2.6 Benchmarking

To study the performance, some way of measuring it is needed. At Neo4j, the benchmarks are either in-house developed data sets, or based on the Linked Data Benchmark Council. Neo4j has an extensive framework to benchmark the graph database. The benchmarks that are presented in the thesis run remotely in a controlled environment, with warm-up runs and several iterations to try and eliminate measurement faults.

2.6.1 Linked Data Benchmark Council

Linked Data Benchmark Council, or LDBC, is a widely used benchmark framework for graph applications. The workload that was used in this study mimicked a social network [13] [3].

2.7 Related work

There are other approaches to cache replacement that were not implemented for this thesis work. Some that seemed promising for performance gain, if implemented with minimal

overhead, are brought up in this chapter.

2.7.1 Multigen LRU

Linux has for a long time used something akin to the 2Q algorithm [7], which utilizes two queues that have different priorities based on the frequency of access of the pages within them [2].

Recently a new patch set that leverages different access patterns to find out what pages are better to keep in memory than others has been developed and is performing very well. It is in consideration to be the default for the Kernel, but needs to be proven to be an improvement "always" for it to fully replace the old algorithm. The strategy is called the Multi-generational LRU in which generations of pages are brought and kept in memory and the oldest generations are those considered for replacement by the strategy [2].

How the algorithm works and is implemented is available in open source. However the implementation and description of the Multigen LRU are very strongly tied to the Kernel, thus it is a formidable challenge to apply said strategy to another system, even though the principles might be of great use to all shared memory caches [16].

2.7.2 Detox

There has been research discussing cache hits as a metric of performance, and it not being a one-to-one ratio in gaining hits meaning gaining throughput. Instead, it has been proven that scoring hits differently depending on whether they helped speed up a transaction or not, and replacing the cache based on that metric gained a big performance increase. This was implemented for a Redis server and showed great potential for performance gains, which is a similar use case to Neo4j [1]. Perhaps this kind of cache replacement policy can be implemented in the Neo4j kernel to improve overall throughput. However, in the current form of the database, all queries are served single-threaded, so it might be better to associate pages that are accessed shortly after one another to be in a group, rather than in parallel. The transactional hit rate as described in the Detox paper would become of great interest if queries get parallelized in the future.

Chapter 3
Method

3.1 Analyzing the current policy

To come up with an improvement plan for the current policy, the first thing done was to analyze the potential weaknesses it could have, and how well it currently performs. This is to get some answers to both RQ1 and RQ2, described in 2.1.1. This was mostly done by deep diving into the current implementation of the cache policy and comparing it to information gathered from research papers. The study revealed some possible weaknesses in the current policy, and the findings were used to decide which improvements to implement.

3.1.1 Algorithms to implement

The following changes were decided to be implemented in the cache to measure if it somehow affected the hit rate. It was decided that the improvements would be close to the current implementation since that would mean the entire cache would not have to be rebuilt. Therefore, all improvements are close to the CLOCK algorithm.

- Introducing a global clock arm shared between all threads.
- Measuring different values of the max reference count, i.e. counting references from values 1-7.
- Implementing a pure CLOCK policy which sets the reference bit to 0 when a page is faulted into the cache, only counts to 1, and evicts it when it is 0.
- Implementing a new adaptive cache replacement policy, based on the CLOCK policy to make it easily implemented in the current implementation. CLOCK-Pro and CLOCK-Pro+ were decided to be implemented.

- Adding a data structure to the current implementation to remember recently evicted pages so that they can be put on a high frequency when they are faulted into the cache.

- A random replacement algorithm to see how much improvements the current policy gives.

3.2 Testing

To test the functionality of the code, some existing integrated tests for the page cache were run. These mostly checked that the cache managed to evict things and did not take too much time. To test the functionality of the different improvements, some small integration tests were added for some of the implementations. These were added for the CLOCK-Pro implementation and were mostly added to see if the implemented algorithm behaved as expected in terms of adding and removing hot pages and non-resident pages.

The integration tests did not use any user data but only tested specific functionality of the cache. To get an approximate measurement of how the implementations would work on real data, a local LDBC benchmark was run. In this run the cache could be specified to various sizes, meaning every implementation could be checked if they worked when many evictions occurred. Furthermore, by changing the cache size to a very small value, the algorithms could be checked for livelocks, which occurred when the cache was too small and the eviction algorithm too slow so that no thread could find anything to evict for a set amount of time. If an implemented improvement livelocked at a bigger cache size than the original algorithm, it gave a small hint that the implementation might be too slow to be effective.

3.3 Benchmarking

The Neo4j benchmarks were used to measure the performance and hit rate of the different implementations. The workload that was used to assess the performance of the page cache is based on LDBC, which was described in 2.6.

Several different settings of the size of the cache was benchmarked to find a size that would give enough cache misses without taking too much time. To find a good setting of the cache size, several benchmarks were run with the initial cache code with different cache sizes.

One problem with the performance measuring was that even though page faults occurred, most of the data was probably saved in the OS cache of the machine running the benchmark. Because of that, the time of a page fault would be less than if the page had to be collected from disk. Therefore, the throughput should be interpreted as how costly an algorithm is, and the hit rate as how well the algorithm performs.

3.4 Implementation

The following sections describe some of the steps taken to implement the changes listed in 3.1.1.

3.4.1 Random replacement

To make a random replacement algorithm, the clock arm was changed. Both the background evictor thread and each thread that ended up in cooperatively eviction got their indices from a random integer instead of a set integer. Instead of increasing it by 1 each time it did not succeed in evicting a page, it was given a new random integer. A page was always evicted if it was not currently used by another thread, no matter its reference bit.

3.4.2 Tuning of usage count

Since updating the reference was already written for a threaded cache, there was no need to think about the concurrency. The only thing needed to implement these changes was to change the max frequency, the reference bit in a fault and at what number a page would be evicted. Various combinations of these changes were benchmarked.

3.4.3 Introducing a global arm

Introducing a global arm meant that it had to be shared between several threads. The current implementation of the arm is an integer, specific for each thread, which is increased and wrapped around whenever it reaches the end of the cache. To implement a shared arm, an atomic long was used and shared between each arm. It was increased atomically and instead of resetting it to 0 whenever it had to wrap around, the index was calculated by floormoding the index with the size of the page cache.

3.4.4 CLOCK

The CLOCK implementation was done by combining the two implementations described above. An atomic arm was added, and the max usage count was changed to 1. Every page was faulted in with a reference bit of 0 and eviction could only happen when the reference bit of a page was 0.

3.4.5 CLOCK-Pro

The CLOCK-Pro implementation is based on the CLOCK implementation, but with two added arms; one hot- and one test arm. These two arms were implemented in the same way as described in 3.4.3. The arm in the CLOCK algorithm was called the cold arm. To keep track of hot pages, non-resident pages, and test periods, some data structures had to be added. Two concurrent sets were added, one containing hot pages and one containing pages in their test period. To keep track of the non-resident pages, a synchronized biMap was added. The biMap mapped indexes in the cache to paths of pages and vice versa. To make sure that the number of hot pages was limited, an atomic long storing the capacity of hot pages was added.

Adding adaptiveness

To make the CLOCK-Pro algorithm adaptive, the capacity of hot pages had to be changed in various places following the algorithm described in 2.4.6. To make sure that the hot pages did not increase too much, since this could cause a high latency to find a page to evict, a max capacity of hot pages was added. The max capacity was set to be 5 percent of the cache size.

CLOCK-Pro+

As was described in 2.4.7 CLOCK-Pro+ has a different way of changing the capacity of hot pages. To be able to do this, cold pages which used to be hot pages needed to be saved. This was done by adding a set containing all pages which had been demoted.

3.4.6 Added history

An attempted improvement made was to add the possibility of remembering recently evicted pages and then prioritizing them by setting their usage count high, to the algorithm with the atomic arm. The counter was first set to 7 when a recently evicted page was faulted in, whilst the max count in other references was set to 4. However, experiments where the max usage counter was 7 were also benchmarked. Initially, this was implemented using a concurrent set. When this set got over a certain capacity the oldest entry was removed, thus only keeping recent history in memory.

Optimizing the data structures

To implement an approximation of remembering history with minimal blocking data structures, an ordered queue and a bitset were used. The queue remembered the order of evicted pages and a specialized thread worked on this queue to remove the oldest entry from the bitset when over the target number of entries (where the target number was a tunable parameter). Then, as a second scaling precaution an atomic bitset was implemented to remove missed writes when writing to the same word concurrently in the bitset.

The approximation was much more lightweight than its original counterpart, but since hashing and a bitset was used, collisions occurred in the set, meaning some pages got accidentally boosted. This was remedied by making the set bigger, but the risk of collision and the implication of collisions were still present, only lessened.

There was also some benign data racing for the sake of performance meaning some decisions might be made on stale data. The hope was that if the algorithm would act on somewhat stale data but not ancient, it would not matter.

3.5 Prototyping

To measure theoretically how well the current policy performs, and how it differs from the new implementations, prototypes of the algorithms were implemented in Python. These prototypes were single-threaded, meaning that the potential overhead and benign races in the real implementation were excluded. To have a trace to benchmark the algorithms on, a local LDBC benchmark was run and the access order of files was recorded to a text file. The

names of the files recorded were used as the unique pages for the Python experiments, and the order was used as a trace.

Beyond the implementations described in 3.1.1, an implementation of the Bélady algorithm was also measured. As is described in 2.4.1, Bélady gives the optimal trace and was used to measure how close to an optimal trace the rest of the algorithms were. Furthermore, a completely random replacement policy was also implemented. This was to get some measurement of how much the current algorithm improved the hit rate.

The Python code for the prototypes can be found here.

Chapter 4

Results

In this chapter, the results from the experiments are presented.

4.1 Weaknesses in the current policy

This section presents the result of the study done to decide what type of improvements could be added to the cache.

As described in 2.5, the currently implemented policy is a CLOCK algorithm, an approximation of an LRU algorithm, with the added possibility for more than one second chance due to the usage counter going to 4 instead of 1. This means that the algorithm can be summarized as an LRU with some LFU functionality. However, the algorithm evicts not only when the reference bit is set to 0, but when it is 0 or 1, meaning the frequency part of the algorithm is small. Another difference from the CLOCK algorithm is that a page's reference bit is set to 1 instead of 0 when it is faulted into the cache.

As was described in 2.4.2 the LRU algorithm does not perform well for all access patterns. One such example is scans. Fortunately, when the algorithm was examined it was discovered that whenever there is a scan pattern, the reference is not counted up. This means that the cache should not be as contaminated by scans as a regular LRU approximation. However, access patterns where there are pages which are not accessed frequently enough, for example, the loop problem described in 2.4.2, should still perform badly with the current policy.

Another problem found when examining the current policy occurs whenever the cooperative eviction is started. When this happens, each thread starts its own clock arm at a random placement in the cache. This means that the CLOCK approximation is worsened, since one thread can start an arm just behind the regular clock arm, causing it to evict pages too early.

4.2 Benchmarking

To find a suitable way to benchmark the cache, several runs were made to measure how small the cache had to be before evictions were seen. The benchmark run was an LDBC read benchmark, and the results can be seen in table 4.1.

Cache size (M)	Hit rate (%)
300 000	99.84
75 000	97.15
37 500	88.84

Table 4.1: Hit rate for different cache sizes

Since the standard setting, 300,000 Mebibyte, only faulted in pages at the beginning of the run and never evicted any pages, it could not be used to benchmark the cache. Using a cache with size 37 500 was very slow to run, and therefore a cache size of 75 000 was decided to be used in later experiments.

4.3 Implementations

The sections below contain the results from the different implementations presented in 3.4.

4.3.1 Random and baseline

Table 4.2 shows the hit rate for the current cache policy and a random replacement. As can be seen, the hit rate is better for the current policy.

Algorithm	Hit rate (%)
Current	97.15
Random	94.95

Table 4.2: Hit rate for random and current

4.3.2 Tuning the usage count

Figure 4.1 shows the results for different max usage counts on the current page cache. As can be seen in the figure, the hit rate increases with an increasing max usage count.

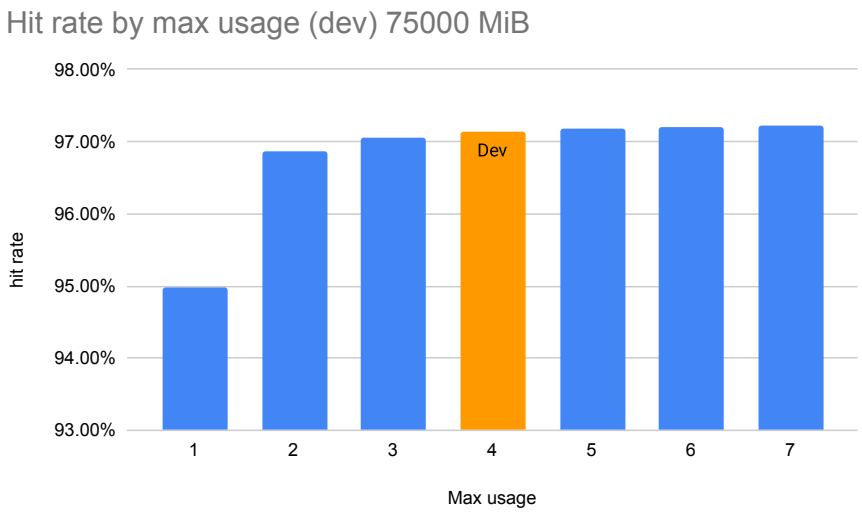

Figure 4.1: Hit rate for different max usage counts for the Muninn Page Cache, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000MiB cache. Using the dev branch of Neo4j as a base. The bar marked orange and labeled dev is the unchanged development branch of Neo4j.

Figure 4.2 contains the throughput for different max usage counts on the current implementation. As can be seen, it varies between different usage counts, but the best throughput is given when the maximum usage count is 7.

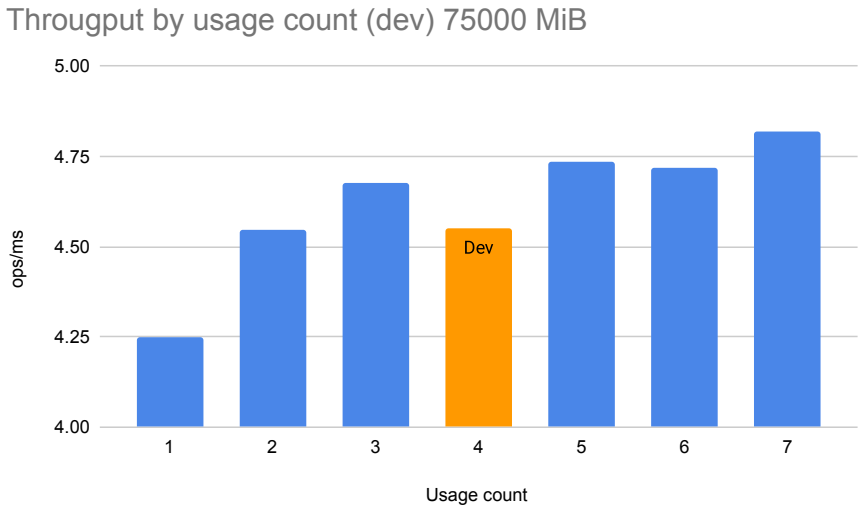

Figure 4.2: Throughput for different max usage counts, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000 MiB cache. Using the dev branch of Neo4j as a base. The bar marked orange and labeled dev is the unchanged development branch of Neo4j.

Figure 4.3 and 4.4 show results for the same experimental setup as 4.1 and 4.2 but with a halved cache. This results in a more stressed cache, observable in both throughput and hit rate.

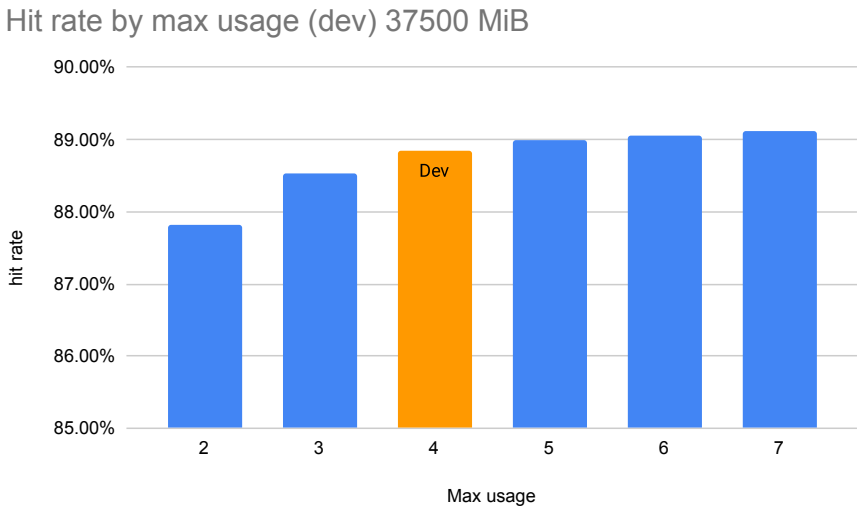

Figure 4.3: Hit rate for different max usage counts for the Muninn Page Cache, measured on a scale factor 100 read benchmark from the LDBC social network workload 37 500 MiB cache. Using the dev branch of Neo4j as a base. The bar marked orange and labeled dev is the unchanged development branch of Neo4j.

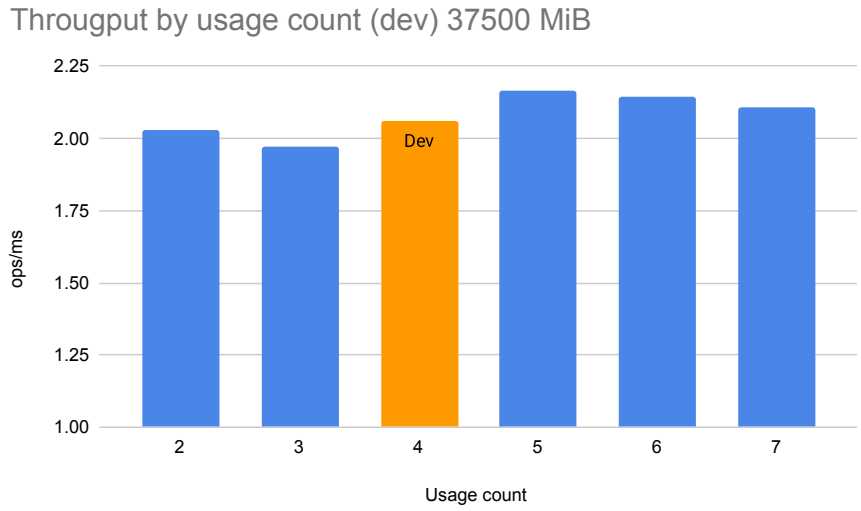

Figure 4.4: Throughput for different max usage counts, measured on a scale factor 100 read benchmark from the LDBC social network workload 37 500 MiB cache. Using the dev branch of Neo4j as a base. The bar marked orange and labeled dev is the unchanged development branch of Neo4j.

4.3.3 Using a global arm

Figure 4.5 shows the hit rate for different max usage counts when using an atomic arm. As can be seen, the hit rate increases when the maximum usage count increases.

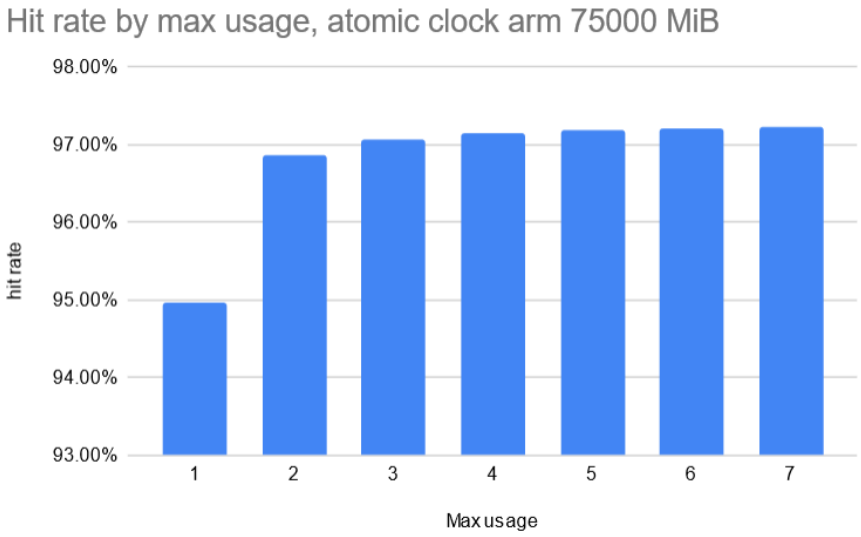

Figure 4.5: Hit rate for different max usage count using an atomic clock arm, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000 MiB cache

Figure 4.6 contains the throughput for different max usage counts when using an atomic clock arm. As can be seen, it varies a bit for different numbers, but the best throughput is given when the maximum usage count is seven.

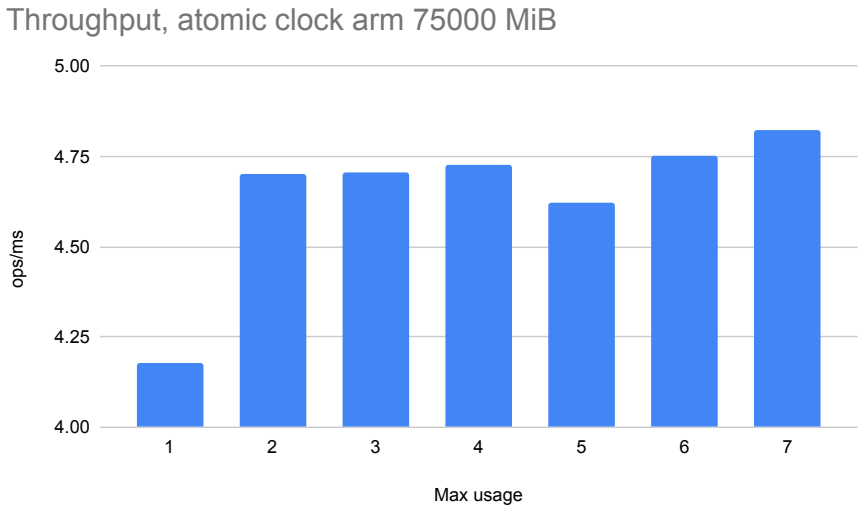

Figure 4.6: Throughput for different max usage count using an atomic clock arm, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000 MiB cache

Figures 4.7 and 4.8 show results for the same experimental setup as 4.5 and 4.6 but with a halved cache. This results in a more stressed cache, observable in both throughput and hit rate. The results differ from the larger cache version of the same experiment in that the throughput penalty is minimal when the maximum usage counter is set to 1. The hit rate is still best for maximum usage 7.

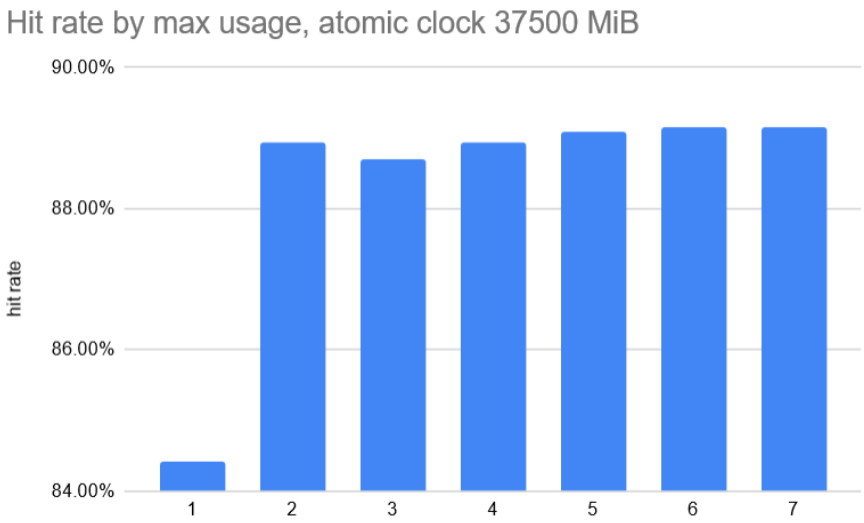

Figure 4.7: Hit rate for different max usage count using an atomic clock arm, measured on a scale factor 100 read benchmark from the LDBC social network workload 37 500 MiB cache

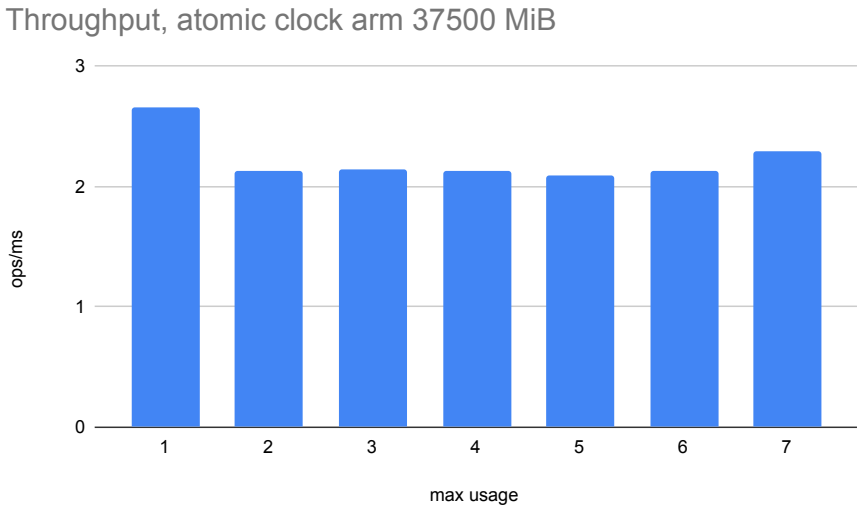

Figure 4.8: Throughput for different max usage count using an atomic clock arm, measured on a scale factor 100 read benchmark from the LDBC social network workload 37 500 MiB cache

4.3.4 CLOCK implementations

Figure 4.9 shows the throughput for the different CLOCK implementations. As can be seen, the clock algorithm has a much higher throughput than the rest of the algorithms.

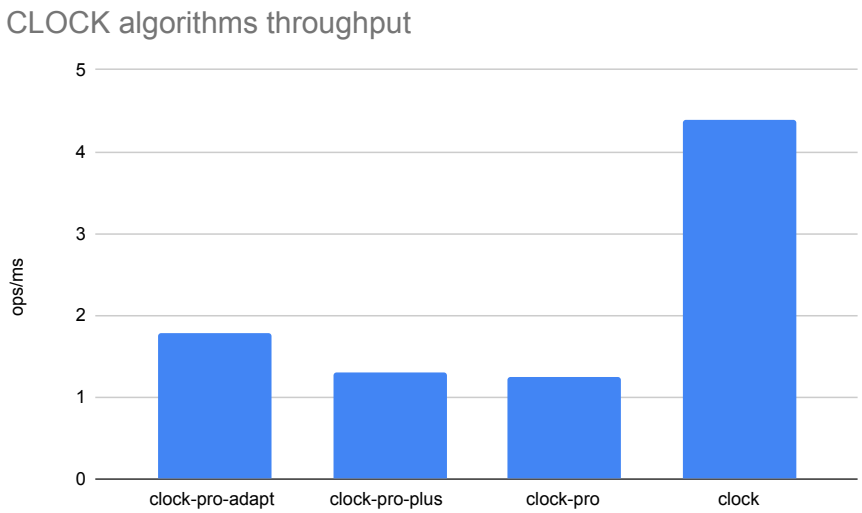

Figure 4.9: Throughput for different CLOCK-based algorithms, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000 MiB cache

Figure 4.10 shows the hit rate for the different CLOCK implementations. As can be seen, CLOCK-Pro with some adaptiveness has the best hit rate.

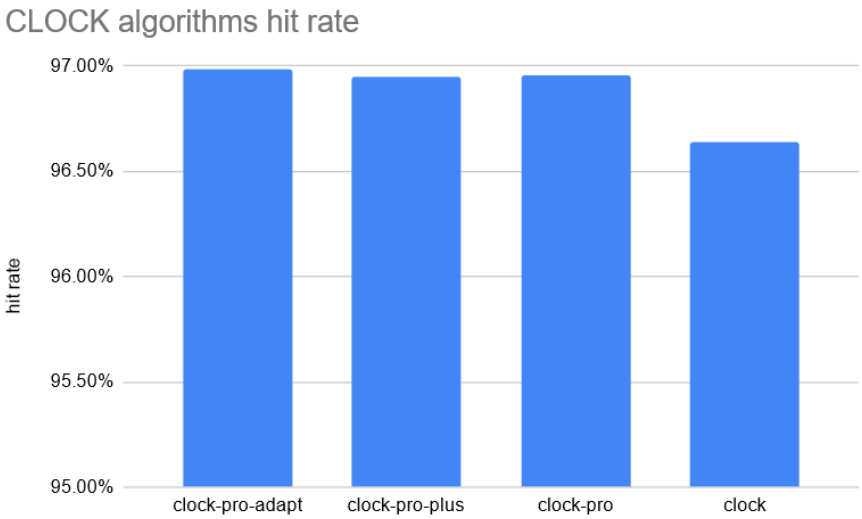

Figure 4.10: Hit rate for different CLOCK-based algorithms, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000 MiB cache

4.3.5 Added history

Figure 4.11 shows the throughput for different implementations of the algorithm that remembered recently evicted pages. The implementation used an atomic arm and had a max count of 4, unless it faulted in a recently evicted page in which case its reference count was set to 7. As can be seen, the concurrent hashset has a very low throughput compared to the other two implementations.

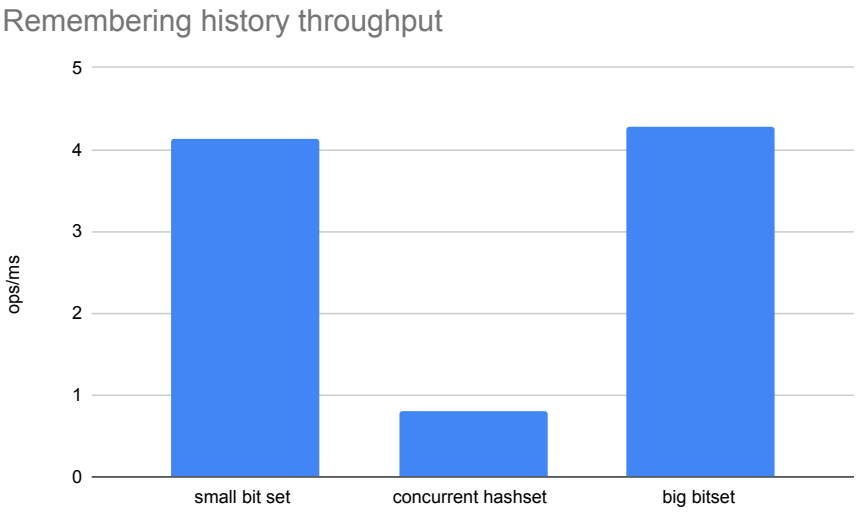

Figure 4.11: Throughput for history-dependent implementations, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000 MiB cache

Figure 4.12 shows the hit rate for the different implementations. As can be seen, the concurrent hashmap had a higher hit rate than the other two implementations.

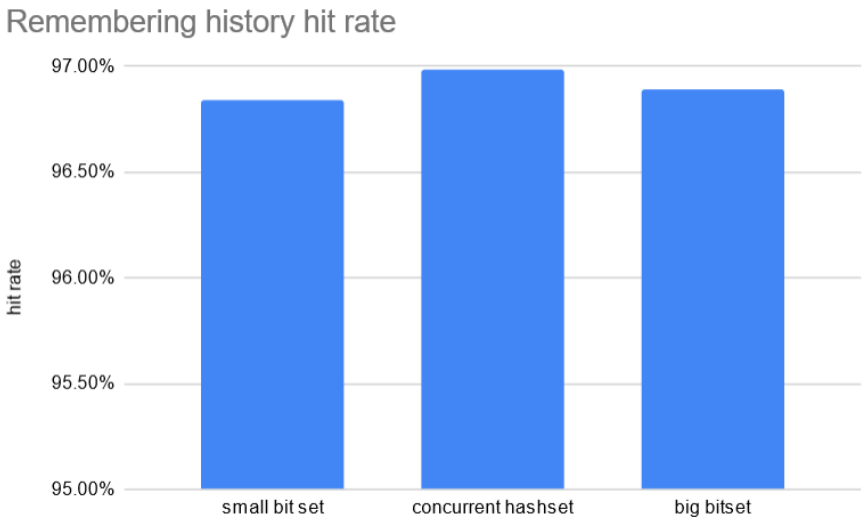

Figure 4.12: Hit rate for history-dependent implementations, measured on a scale factor 100 read benchmark from the LDBC social network workload 75 000 MiB cache

The experiment where the max usage counter was set to 7 gave an even better hit rate for a cache of size 75 000 MiB, namely 97.11%.

4.3.6 Summary

Table 4.3 summarizes each section's best hit rate result so that it can be easily compared to random and current.

Algorithm	Hit rate (%)
Current	97.15
Random	94.95
Count to 7	97.22
Count to 7 with global arm	97.22
CLOCK-Pro adaptive	96.98
History with concurrent set count to 7	97.11

Table 4.3: Best hit rates from each section compared to random and baseline

4.4 Prototyping

The results from the Python prototypes are summarized in table 4.4. The table shows each algorithm with their hit count, hit rate and the hit rate in percentage of the optimal hit rate produced by the Bélady algorithm.

Algorithm	Total number of hits	Hit rate (%)	Hit rate compared to optimal (%)
Bélady	66 969	47.73	100
Random	65 438	46.64	97.72
Current	66 681	47.53	99.58
Current count to 7	66 690	47.53	99.58
Current count to 6	66 687	47.53	99.58
Current count to 5	66 685	47.53	99.58
Current count to 3	66 671	47.52	99.56
Current count to 2	66 653	47.51	99.54
Current count to 1	65 982	47.03	98.53
Current with history	66 771	47.59	99.71
CLOCK	66 653	47.51	99.54
CLOCK-Pro	66 647	47.50	99.52
CLOCK-Pro+	66 645	47.50	99.52
CLOCK-Pro adaptive	66 647	47.50	99.52

Table 4.4: Python results

As can be seen in the table, the implementation that had best result was the one which remembered recently evicted pages. This implementation counted to 4, but every time a recently evicted page was faulted, its reference bit was bumped up to 7. Furthermore, it is clear when looking at the result that a random replacement is significantly worse than the current implementation. It is also clear that the higher the reference, the better the hit rate.

Since counting to 7 and adding history seemed to give the best result, an experiment was executed combining the two solutions. This means that the references could be counted to 7, and if a recently evicted page was faulted into cache its reference count was set to 7. This yielded the best result as can be seen in table 4.5.

Algorithm	Number of hits	Hits compared to optimal (%)
Bélady	66 969	100
Current	66 681	99.57
Current count to 7	66 690	99.58
Current with history	66 771	99.70
History and count to 7	66 782	99.72

Table 4.5: Combining the two best solutions

Chapter 5
Discussion

5.1 RQ1 - What improvements can be made in the current policy?

As the results showed, some experiments did manage to improve the hit rate in the cache. The sections below summarize the findings that gave an improved hit rate, and some that did not. Although some findings only improved the trace in Python, others improved the hit rate in the benchmarks.

5.1.1 Changing the usage counter

Both the Python implementations and the actual benchmarked implementations showed that an increased usage counter gave an improved hit rate. It was only measured up to a reference count of 7 since this was the number of bits available, explained in 2.5. These results indicate that letting some short-term frequency behavior be captured more than it is in the current state of the cache is of value to Neo4j, i.e. mixing some LFU into the LRU algorithm. Even though it could be assumed that this would negatively impact the time to evict, since fewer pages would be ready for eviction, this was not observed in the throughput metrics of the experiments. However, it is important to consider that the trace used in the Python experiments and the traces run on the benchmark might be more suited for some added frequency. If more traces and access patterns were tested then maybe the frequency boost would not be as visible as it was for these experiments.

5.1.2 Atomic clock arm

As was seen in the results for the experiments using one global, concurrently accessible clock arm for all eviction processes in the page cache 4.3.3, this appeared to increase both through-

put and hit rate.

That the hit rate increased was not very surprising since it gave a more consistent eviction and less of a random eviction. Since the clock arm during the cooperative eviction process previously started at a random index, this meant that a page that had just had its reference count decremented to 0 from 1 could be demoted too fast if the random arm ended up on that position. As was seen in the results, the current LRU approximation performed better than a random algorithm 4.3.1, which indicates that removing further randomness from the eviction process would yield a better hit rate.

The improvement in throughput was more surprising since making a global arm comes with some increased sequential behavior in the threading structure. However, having a global arm also decreased the probability of collisions, i.e. that all threads would try to evict the same page. Since atomic operations are less costly than for example locks, the improvements of using a global clock arm seemed to outweigh the time added by using some costly operations.

Another interesting thing that could be seen in the results was that the improvement from using a global arm increased when the size of the cache decreased. This further strengthens the theory that it is the randomness of the cooperative eviction process that negatively affects the eviction process. This since if the cache is smaller more pages need to be evicted, which increases the probability of ending up in cooperative eviction.

5.1.3 Added history

Even though the implementations with added history did not provide an increased hit rate for the Neo4j implementation, it showed great promise in the Python prototype. This means that the trace used in the Python experiments had access patterns that were beneficial for this implementation. The question is whether these access patterns are common enough for this to be an actual improvement. Since no improvement was visible for the LDBC trace, either it is not common enough, or more testing on different traces needs to be executed.

That it would be an improvement to prioritize some recently evicted pages goes in line with the literature study on LRU 2.4.2. This since the problems the LRU algorithm has with patterns where pages are accessed with longer reuse distances, for example in loops, would decrease since some reuse distance is taken into account.

The biggest problem with this implementation was how to tackle the large overhead introduced by the implementation. The algorithm implemented without benign races and a unique identification for each page, i.e. the concurrent hashset, decreased the throughput by almost 90%. The implementations that allowed some benign racing and some collisions on page identification did not experience such a large decrease in throughput but at the cost of a worse hit rate.

5.1.4 CLOCK algorithms

Neither of the implemented CLOCK algorithms improved the hit rate. That the regular CLOCK algorithm did not improve the hit rate was not surprising since previous results showed that an increased reference count gave a better result. However, the hope was that CLOCK-Pro and CLOCK-Pro+ would increase the hit rate since they are adaptive and take into account both frequency and recency. However, since none of them increased the hit rate for either the prototype implementation or the real one, it seems that Neo4j does not benefit

from these algorithms. One possible explanation for this is that the access pattern has few short periods of patterns which corresponds to creating hot pages. Whenever this happens the cache is changed to contain more hot pages, and it takes a while for it to rearrange itself to work like a CLOCK algorithm again. This contaminates the cache and lowers the hit rate. Another explanation could be that the implementations are faulty or not optimally implemented. The implementations have been made to try and follow the articles as closely as possible, but it is difficult to eliminate all human errors.

5.2 RQ2 - How much is there to gain by using a more effective cache replacement policy?

Several examples show that when the hit rate goes up, it comes with an increased throughput, and since the OS cache is doing some heavy lifting for us when our page cache is full the real-world implications are even bigger than the results we have observed in the benchmarks. However, we saw an upwards of 0.07% increase in hit rate for no additional overhead cost. Given that a page fault roughly takes 0.1 ms in a real system, the performance implication of just a tiny change in hit rate can be massive [12]. This decrease in page faults in the benchmark was approximately 12 500 000 fewer. Without the OS cache stepping in this would result in a 1 250 seconds faster run, or approximately 20 minutes.

The prototype implementations revealed that the current implementation has a hit rate that is 99.58% of an optimal replacement. This means that the margins of improvement are very small, and any small increase could be beneficial.

5.3 RQ3 - How big can the implementation overhead be before the implementation costs more than it gives?

As was seen in the results, adding blocking data structures has massive performance costs. This could especially be seen in the results of the algorithm with added history which counted to 7 and was implemented with a concurrent hashset. The hit rate for this implementation was 97.11% compared to the current implementation which had a hit rate of 97.15%. Even though the hit rate is very close to that of the current implementation, the throughput decreased by almost 90%. Even if the algorithm had led to a small increase in hit rate, the overhead was too much for it to be efficient.

To attempt to answer more precisely how big the overhead can be, a thought experiment can be done. If a page fault is assumed to take 0.1 ms - a very generous estimate -, then reducing 1 page fault can take a maximum of 0.1 ms [12]. For a run with a 10% fault rate, that means that there are 9 page accesses with hits and 1 with a fault. Those ten accesses can together have an overhead of 0.1 ms, so every decision can have an overhead of 0.01 ms.

If the numbers presented here are applied to the example with the concurrent hashset, it

can be easily seen that the cost of the implementation is too expensive. The hit rate decreases by 0.04% for the concurrent hashset, corresponding to an increase in runtime of 80 ms. This would lead to a decrease in throughput corresponding to approximately 0.02%. Instead, the algorithm yields an increase in throughput of almost 90%, which is way too high. If the OS cache did not exist, the runtime would decrease by approximately 10 seconds. This would be a decrease of throughput by approximately 2%, which still is much less than the actual increase.

Chapter 6
Conclusion

As was seen, the page cache could be improved with a fairly small and not costly implementation. Increasing the usage count and using a global arm increased the hit rate by 0.07%, corresponding to approximately a 20 minutes faster run. However, for more accurate conclusions it should be tested on a wider range of benchmarks. Other algorithms had the potential to increase the hit rate had they not increased the throughput as much as they did.

The Python implementation of Belády's algorithm suggests that the cache replacement policy has a maximum limit of improvement of 0.42%. The best real improvement that was seen was an improvement of 0.07%. This improvement means the hit rate is now 17% closer to a hypothetical optimal hit rate, so even though it seems small, it is in reality a big improvement.

Since the cost of a page fault is very expensive compared to other costs in the database, an algorithm could add some overhead without affecting the throughput if the hit rate is increased. However, adding very expensive blocking structures seems to add too much overhead for it to be an efficient improvement. It seems like the most important thing for throughput is to keep the cache as parallel as possible, whilst still being somewhat smart in its decision. However, it is hard to draw any concrete conclusions regarding this since none of the experiments had an increased hit rate and a higher throughput at the same time.

6.1 Future work

The following sections describe what could be worked on for future investigations.

6.1.1 Other implementations

As evident by the results from the experimentation with the usage counter, it seems that capturing frequency is of value. However, using purely LFU to make eviction decisions was tested in a small Python implementation and did not yield an improved result.

Furthermore, as was mentioned in 2.4, there are two main types of replacement policies, course-grained and fine-grained. This thesis only investigated how different course-grained policies could improve the cache, but a future investigation could look into whether a fine-grained implementation would be a better improvement.

6.1.2 Accurate performance benchmarks

The cost of actually going to storage is offset in the current Neo4j benchmarks by instead going to the operating system cache in many cases. To gain an accurate estimate of how much time is lost and saved dependent on the caching and eviction, the OS cache would need to be disabled to force access to storage instead.

6.1.3 Transactional hit rate

As discussed in the Detox paper it might be of interest to create metrics and measure how many blocking faults we are waiting on that make some hits useless. This might become more relevant in the future if Neo4j starts parallelizing the queries to a greater degree.

References

[1] Audrey Cheng, David Chu, Terrance Li, Jason Chan, Natacha Crooks, Joseph M. Hellerstein, Ion Stoica, and Xiangyao Yu. Take out the TraChe: Maximizing (tra)nsactional ca(che) hit rate. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 419–439, Boston, MA, July 2023. USENIX Association.

[2] Jonathan Corbet. The multi-generational lru. `https://lwn.net/Articles/851184/`, April 2021.

[3] Linked Data Benchmark Council. `https://ldbcouncil.org/`.

[4] A. Jain and C.Lin. Cache replacement policies: synthesis lectures on xyz #13. `https://par.nsf.gov/servlets/purl/10113803https://par.nsf.gov/servlets/purl/10113803`. Accessed Nov. 08. 2023.

[5] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 35, USA, 2005. USENIX Association.

[6] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, page 31–42, New York, NY, USA, 2002. Association for Computing Machinery.

[7] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Very Large Data Bases Conference*, 1994.

[8] Cong Li. Clock-pro+: Improving clock-pro cache replacement with utility-driven adaptation. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Neo4j. Muninn page cache architecture and internals. `https://github.com/neo4j/neo4j/blob/5.10/community/io/src/main/java/org/`

neo4j/io/pagecache/impl/muninn/page-cache-internals.adoc. Accessed Nov. 08, 2023.

[10] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, Burlington, MA, USA, 2014.

[11] José Rocha. Understanding memory consumption. https://neo4j.com/developer/kb/understanding-memory-consumption/. Accessed Nov. 27. 2023.

[12] PassMark Software. Top disk - random seek read write (iops 32kqd20) chart. https://www.harddrivebenchmark.net/random-read-write.html. Accessed Nov. 29. 2023.

[13] Gábor Szárnyas, Brad Bebee, Altan Birler, Alin Deutsch, George Fletcher, Henry A. Gabb, Denise Gosnell, Alistair Green, Zhihui Guo, Keith W. Hare, Jan Hidders, Alexandru Iosup, Atanas Kiryakov, Tomas Kovatchev, Xinsheng Li, Leonid Libkin, Heng Lin, Xiaojian Luo, Arnau Prat-Pérez, David Püroja, Shipeng Qi, Oskar van Rest, Benjamin A. Steer, Dávid Szakállas, Bing Tong, Jack Waudby, Mingxi Wu, Bin Yang, Wenyuan Yu, Chen Zhang, Jason Zhang, Yan Zhou, and Peter Boncz. The linked data benchmark council (ldbc): Driving competition and collaboration in the graph data management space. In *Proceedings of the Fifteenth TPC Technology Conference on Performance Evaluation & Benchmarking*, July 2023. Fifteenth TPC Technology Conference on Performance Evaluation; Benchmarking (TPCTC 2023), TPCTC 2023 ; Conference date: 28-08-2023.

[14] Wikipedia. Cache replacement policies. https://en.wikipedia.org/wiki/Cache_replacement_policies. Accessed Nov. 07. 2023.

[15] Po Wu, Jiangnan Zhang, Yiming Ruan, Guanghui Chang, Yi Wang, and Yanlou Song. Construction of knowledge graph for substation automation equipment ledger based on neo4j graph database. In *2023 3rd Power System and Green Energy Conference (PSGEC)*, pages 1080–1086, 2023.

[16] Yu Zhao. [patch v1 00/14] multigenerational lru. https://lwn.net/ml/linux-kernel/20210313075747.3781593-1-yuzhao@google.com/, March 2021.

Appendices

Appendix A
Java pseudo code

Listing A.1: Current implementation

```
1  private static final MAX_USAGE_COUNT long = 4;
2
3
4  private long cooperativelyEvict() {
5      int iterations = 0;
6      boolean evicted = false;
7      long pageRef;
8      long armPos = random(0,PageCache.size);
9      do {
10         if (getFreelistHead() != null) {
11             /* Eviction is no longer needed since there is
                  something in the list of free pages */
12             return 0;
13         }
14
15         if (armPos == pageCount) {
16                 armPos = 0;
17         }
18
19         pageRef = pages.deref(armPos);
20         if (PageList.isLoaded(pageRef) && PageList.
               decrementUsage(pageRef)) {
21             /* The page was in the cache, and had low enough
                  usage counter to be eligable for eviciton */
22             evicted = pages.tryEvict(pageRef, faultEvent);
23         }
24         armPos++,
25     } while (!evicted);
```

```
    return pageRef;
}


static boolean decrementUsage(long pageRef) {
    /* get usage count from our meta data */
    usage = getUsage(pageRef);
    if (usage > 0) {
        long update = value - 1;
        /* Somewhat safely update usage counter
            of the meta data */
    }
    /* ok to evict if counter was 1
      or zero before decrementing */
    return usage <= 1;
}


/* This function is called when
 a page in the cache gets a hit */

static void incrementUsage(long pageRef) {
    /* get usage count from our meta data */
    usage = getUsage(pageRef);
    if (usage < MAX_USAGE_COUNT)
    {
        long update = value + 1;
        /* Somewhat safely update usage counter
            of the meta data */
    }
}
```

Listing A.2: Atomic usage 7

```
private static final AtomicLong clockArm = new AtomicLong();

private static final MAX_USAGE_COUNT long = 7;

private long cooperativelyEvict() {
    boolean evicted = false;
    long pageRef;
    do {
        if (getFreelistHead() != null) {
            /* Eviction is no longer needed since there is
               something in the list of free pages */
            return 0;
        }
        long armPos = atomicallyIncrementClockArmAndReturn();

        pageRef = pages.deref(armPos);
```

```
        if (PageList.isLoaded(pageRef) && PageList.
           decrementUsage(pageRef)) {
            /* The page was in the cache, and had low enough
               usage counter to be eligable for eviciton */
            evicted = pages.tryEvict(pageRef, faultEvent);
        }
    } while (!evicted);
    return pageRef;
}

/* For the movement of the clock arm we let an atomic long
   counter represent the clock arm position, where we use modulo
    to get the actual position. We let the atomic long counter
   overflow and wrap around, but using floorMod allows this to
   still have correct clock arm movement */

long atomicallyIncrementClockArmAndReturn() {
    long arm;
    arm = clockArm.getAndIncrement();
    return Math.floorMod(arm, pages.getPageCount());
}
```

Listing A.3: CLOCK-Pro

```
private static AtomicLong capacityHotPages = PageCache.size /
   100;
private static final MAX_USAGE_COUNT long = 1;
private final AtomicLong testArm = new AtomicLong();
private final AtomicLong coldArm = new AtomicLong();
private final AtomicLong hotArm = new AtomicLong();

MutableBiMap<Object, Object> nonResidentColdPages = new
   HashBiMap<>().asSynchronized();
public Set<Long> hotPages = ConcurrentHashMap.newKeySet();
public Set<Long> testPages = ConcurrentHashMap.newKeySet();

private long cooperativelyEvict() {
    boolean evicted = false;
    long pageRef;
    long idx;
    do {
        if (getFreelistHead() != null) {
            /* Eviction is no longer needed since there is
               something in the list of free pages */
            return 0;
        }

        /* The cold arm returns us a index of the list of pages
           to apply the eviction logic on */
        idx = increaseColdArm();
```

```
23          pageRef = pages.fetch(idx);
24          boolean hot = hotPages.contains(pageRef);
25          if (hot) {
26              continue;
27          }
28
29          /* If the page is recently referenced (used) the
               reference counter will be 1, here returned as true if
                so is the case */
30          boolean ref = PageList.getAndDecrementUsage(pageRef);
31          boolean inTestPeriod = testPages.contains(pageRef);
32          String evictedId = obtainUniqueFileMapping(pageRef);
33
34          if(ref && inTestPeriod) {
35              hotPages.add(pageRef);
36              if (hotPages.size() >= capacityHotPages.get()) {
37                  moveHotHand();
38              }
39          }
40
41          if (!ref) {
42              evicted = pages.tryEvict(pageRef, faultEvent);
43              if (evicted) {
44                  if (inTestPeriod) {
45                      addNonRes(evictedId, idx);
46                  }
47                  if (getNbrOfNonRes() > PageCache.size) {
48                      moveTestHand();
49                  }
50                  testPages.remove(pageRef);
51              }
52          }
53      } while (!evicted);
54      return pageRef;
55  }
56
57  /* Seek a nonRes page and remove it.
58   On the way terminate test periods */
59  void moveTestHand() {
60      while (!closed && nonResidentColdPages.notEmpty()) {
61          int idx = increaseTestArm();
62          long pageRef = pages.fetch(idx);
63          if (testPages.remove(pageRef) && !hotPages.contains(
               pageRef) && PageList.getUsage(pageRef) == 0) {
64              increaseCapacityHotPages();
65          }
66
67          if (null != removeNonResIdx(idx)) {
68              return;
```

```
69          }
70      }
71  }
72
73  void moveHotHand() {
74      while (!closed && !hotPages.isEmpty()) {
75          long idx = increaseHotArm();
76          long pageRef = pages.fetch(idx);
77
78          if (removeNonResIdx(idx) != null) {
79              increaseCapacityHotPages();
80          }
81
82          boolean wasTest = testPages.remove(pageRef);
83
84          if (!hotPages.contains(pageRef)) {
85              if (wasTest && PageList.getUsage(pageRef) == 0) {
86                  increaseCapacityHotPages();
87              }
88
89              continue;
90          }
91
92          if (!PageList.getAndDecrementUsage(pageRef)) {
93              if (hotPages.remove(pageRef)) {
94                  return;
95              }
96          }
97      }
98  }
99
100 // Called when a page is faulted in
101 void init(long pageRef) {
102     String id = obtainUniqueFileMapping(pageRef)
103     Object wasNonRes = removeNonResId(id);
104     if (wasNonRes != null) {
105         hotPages.add(pageRef);
106         if (hotPages.size() >= capacityHotPages.get()) {
107             moveHotHand();
108         }
109     }
110     testPages.add(pageRef);
111 }
112
113 void increaseCapacityHotPages() {
114    PageCache.capacityHotPages.set(Math.max(value + 1,
       capacityMaxValue));
115 }
116
```

```
117 void decreaseCapacityHotPages() {
118     PageCache.capacityHotPages.set(Math.max(value - 1, 0));
119 }
```

Listing A.4: CLOCK-Pro+

```
1  public Set<Long> demotedPages = ConcurrentHashMap.newKeySet();
2
3  /* called when the reference bit is observed
4   as set on a demoted page */
5  void increaseCapacityHotPages(long maxPages) {
6      /* Change is calculated from the utility of increasing the
          hot pages after resident cold page is seen with a ref and
           has been demoted from hot */
7      int nbrOfColdPages = PageCache.size - hotPages.size();
8      int change = Math.min(1, nbrOfColdPages / demotedPages.size
          ());
9      long value = PageCache.capacityHotPages.get();
10     PageCache.capacityHotPages.set( Math.min(value + change,
          maxPages / 20));
11 }
12
13 /* called when a page fault is on a non-resident cold page */
14 void decreaseCapacityHotPages() {
15     /* Change is calcuated from the utiltiy of increasing number
           of cold pages with hypothetical access to a non resident
           cold page.*/
16     int nbrOfColdPages = PageCache.size - hotPages.size();
17     int change = Math.min(1, demotedPages.size() /
          nbrOfColdPages);
18     long value = PageCache.capacityHotPages.get();
19     PageCache.capacityHotPages.set(Math.max(value - change, 0));
20 }
```

Appendix B
Python pseudo code

Listing B.1: Simulation

```
def simulate(self, evict):
    self.cache_refs = {}
    while self.trace:
        page = self.trace.pop(0)
        if page in self.cache_refs:
            self.hits += 1
            self.cache_refs[page] = min(self.cache_refs[page] +
               1, self.max_ref)
            #decrease capacity hot pages if
            #clock pro adaptive and cold page in test period
        else:
            self.faults += 1

            #only for clock-pro implementations
            self.test_pages.add(page)

            if len(self.cache_refs) == self.cache_size:
                self.eviction_policies[evict]()
                self.cache_refs.pop(self.cache[self.free_index])
                self.cache[self.free_index] = page
                self.cache_refs[page] = self.fault_ref

                #only for clock-pro and added history
                #implementations
                if page in self.non_res_pages:
                    self.non_res_pages.popitem(page)
                    self.non_res_cache[self.free_index] = None

```

```
                    #decrease hot pages capacity if
                    #clock pro plus or adaptive

                    #if clock-pro
                        self.hot_pages.add(page)
                        if len(self.hot_pages) > self.
                           capacity_hot:
                            self.move_hot_hand()
                    #if added history, set ref to max
                        self.cache_refs[page] = self.max_ref
            else:
                self.cache[self.free_index] = page
                self.free_index += 1
                self.cache_refs[page] = self.fault_ref
```

Listing B.2: Random

```
def evict(self):
    self.free_index = random.randint(0, self.cache_size-1)
```

Listing B.3: Bélady

```
def evict(self):
    max_distance = -1
    self.free_index = 0
    non_known = set()
    for i in range(self.cache_size):
        non_known.add(self.cache[i])

    for i in range(self.cache_size):
        dist = self.page_distance[i]
        if dist > 0:
            non_known.remove(self.cache[i])

    for j in range(len(self.trace)):
        page = self.trace[j]
        if page in non_known:
            for i in range(self.cache_size):
                if self.cache[i] == page:
                    self.page_distance[i] = j
            non_known.remove(page)

    if len(non_known) != 0:
        for i in range(self.cache_size):
            page = self.cache[i]
            if page in non_known:
                self.free_index = i
                self.page_distance[i] = -1
                self.decrement_distance()
                return
```

```
29
30      max_index = 0
31      for i in range(self.cache_size):
32          if self.page_distance[i] > max_distance:
33              max_distance = self.page_distance[i]
34              max_index = i
35
36      self.free_index = max_index
37      self.decrement_distance()
38      self.page_distance[self.free_index] = -1
```

Listing B.4: CLOCK variants (including current)

```
1
2   def clock_evict(self):
3       while True:
4           self.free_index += 1
5           if self.free_index >= cache_size:
6               self.free_index = 0
7           page = self.cache[self.free_index]
8           refs = self.cache_refs[page]
9           self.cache_refs[page] = max(self.cache_refs[page] - 1,
                0)
10          if refs == self.evict_ref:
11              return
```

Listing B.5: CLOCK-Pro

```
1   def move_hot_hand(self):
2       removed = False
3       while not removed:
4           index = self.hot_hand % self.cache_size
5           self.hot_hand += 1
6           page = self.cache[index]
7           non_res_page = self.non_res_cache[index]
8           if page in self.hot_pages:
9               if self.cache_refs[page] == self.evict_ref:
10                  self.hot_pages.discard(page)
11                  removed = True
12              self.cache_refs[page] = 0
13          if page in self.test_pages:
14              self.test_pages.discard(page)
15              #increase hot pages capacity if
16              #clock pro adaptive and not referenced cold page
17          if non_res_page:
18              self.non_res_pages.popitem(non_res_page)
19              self.non_res_cache[index] = None
20              #increase hot pages capacity if clock pro adaptive
21
22  def move_test_hand(self):
```

```
    removed = False
    while not removed:
        index = self.test_hand % self.cache_size
        self.test_hand += 1
        page = self.cache[index]
        non_res_page = self.non_res_cache[index]
        if non_res_page:
            self.non_res_pages.popitem(non_res_page)
            self.non_res_cache[index] = None
            removed = True
            #increase hot pages capacity if clock pro adaptive
        if page in self.test_pages:
            self.test_pages.discard(page)
            #increase hot pages capacity if
            #clock pro adaptive and not referenced cold page

def clock_pro_evict(self):
    evicted = False
    while not evicted:
        self.free_index += 1
        if self.free_index >= self.cache_size:
            self.free_index = 0
        page = self.cache[self.free_index]
        ref = self.cache_refs[page]

        if page in self.hot_pages:
            continue

        self.cache_refs[page] = 0

        #increase hot pages capacity if clock pro plus
        #and page is demoted with ref > 0

        if page in self.test_pages and ref > self.evict_ref:
            self.hot_pages.add(page)
            if len(self.hot_pages) > self.capacity_hot:
                self.move_hot_hand()
            continue

        if ref == self.evict_ref:
            page = self.cache[self.free_index]
            evicted = True
            if page in self.test_pages:
                self.non_res_pages[page] = 0
                self.non_res_cache[self.free_index] = page
                self.test_pages.remove(page)
                if len(self.non_res_pages) > self.cache_size:
                    self.move_test_hand()
```

Listing B.6: Added history

```
def add_non_res(self, page):
    if len(self.non_res_pages) == self.cache_size:
        self.non_res_pages.popitem(last = False)
    self.non_res_pages[page] = 0

def added_history_evict(self):
    while True:
        self.free_index += 1
        if self.free_index >= self.cache_size:
            self.free_index = 0
        page = self.cache[self.free_index]
        ref = self.cache_refs[page]
        self.cache_refs[page] = max(0, ref - 1)

        if ref <= self.evict_ref:
            self.add_non_res(page)
            return
```

DEPARTMENT OF COMPUTER SCIENCE | LUNDS TEKNISKA HÖGSKOLA | PRESENTED 2024-01-30

MASTER THESIS Cache replacement policies and their impact on graph database operations
STUDENTS Tora Elding Larsson, Lukas Gustavsson
SUPERVISOR Jonas Skeppstedt (LTH), Anton Klarén (Neo4j)
EXAMINER Michael Doggett (LTH)

What to remember and what to forget

POPULAR SCIENCE SUMMARY **Tora Elding Larsson, Lukas Gustavsson**

Imagine that you are working on an essay at your desk. Instead of having all your books on a bookshelf, you have the books you need for the essay at your desk so that you don't have to go back and forth all the time. Unfortunately, all books don't fit on your desk, so sometimes you need to replace one. The question is which one?

In this thesis, the bookshelf is a long-term memory and your desk is a cache memory. For Neo4j, all data must be stored long-term, but you still want some data in the cache memory so that you can find it more quickly. Whenever the cache is full, something needs to be replaced. Imagine the bookshelf again. If you replace a book that you need soon after it has been replaced, you would have to go back and forth to the bookshelf two times in a very short period. A better idea would be to replace something that you don't need in the future, or at least not for a very long time. In the cache, the decision of what to replace is made by an algorithm called the eviction policy. The focus of this thesis has been to investigate the policy currently used by Neo4j and to try to improve it.

The algorithm currently used decides what to replace based on when it was used. If we go back to the book example it would mean that the book that was used the longest time ago is replaced, since we assume that a book that was recently used will be used again shortly. In Neo4j the cache is also threaded, meaning several threads use the cache at the same time. Imagine that several people worked on the essay at the same time at the same desk. Then you wouldn't want a person to replace a book that you might need soon.

We tried many approaches to improve the algorithm, based on previous research. Some approaches were to tweak the current algorithm, and some was to implement entirely new algorithms. We discovered that the current algorithm was working very well. However, tweaking some parameters in it and changing the structure for how the different threads were organized, improved the cache.