



**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Martin Koreček

**A Comparison of Strategies
for Database Caching**

Department of Distributed and Dependable Systems

Advisor of the bachelor thesis: William Brown

Local supervisor: prof. Ing. Petr Tůma, Dr.

Study programme: Computer Science

Study branch: General Computer Science

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

Author's signature

I would like to thank my advisor, William Brown, not just for sharing his knowledge of the research topic and for the time he devoted to it, but also for the insight about working for an international software company.

I would also like to thank professor Petr Tůma for taking on the role of my formal supervisor and for his helpful advice and patience.

Title: A Comparison of Strategies for Database Caching

Author: Martin Koreček

Department: Department of Distributed and Dependable Systems

Advisor: William Brown, SUSE Labs

Local supervisor: prof. Ing. Petr Tůma, Dr., Department of Distributed and Dependable Systems

Abstract: Database caching is the practice of keeping an amount of data in memory, to reduce the cost of accesses to the main storage, and thus improve the performance of a database machine. We particularly focus on two properties of database caches. Firstly, how different cache replacement policies decide what data is kept in memory and secondly, what options exist to allow parallel accesses to the cached set by multiple threads. With a limited resource of access logs from real NoSQL databases in production, we will measure the performance of the discussed replacement policies. And we will measure the in-memory performance of the data structures that enable parallelism.

Keywords: cache, strategy, database, concurrency

Contents

Introduction	3
1 Cache replacement policies	5
1.1 LRU	5
1.2 2Q	5
1.3 ARC	6
1.4 LFU	7
1.5 2Q-LFU	7
1.6 LIRS	7
1.7 CLOCK	8
1.8 CLOCK-Pro	9
1.9 Random replacement	10
2 Evaluation of the replacement policies	11
2.1 More about our measurements	11
2.2 Results for Log 1	12
2.3 Results for Log 2	14
2.4 Discussion	15
3 Strategies for concurrency	17
3.1 Single lock	17
3.2 Associative caches	18
3.3 Small cache for each thread	19
3.4 Concurrently readable cache via MVCC	21
4 Our simulations of the concurrent strategies	22
4.1 LOCK	22
4.2 ASSOCIATIVE	22
4.3 PER THREAD	23
4.4 TRANSACTIONAL	23
5 Concurrency: Evaluation	26
5.1 The workload and measurement tools	26
5.2 Results for single thread execution	27
5.3 Results for parallel execution	28
5.4 Additional workloads	29
5.5 Interpretation of the results	30
Conclusion	31
6 Details	32
6.1 Parameters for the 2Q and 2Q-LFU replacement policies	32
6.2 The parameter for the LIRS replacement policy	39
6.3 Open source database engines' approaches to cache synchronization	42
6.3.1 389 Directory Server	42
6.3.2 LMDB (part of the OpenLDAP project)	42

6.3.3	Postgres	42
6.4	Analysis of the <i>ASSOCIATIVE</i> strategy	44
6.5	Transactional hash map implementation	46
6.5.1	COW-friendly B+ tree	46
6.5.2	Hash map made with the B+ tree	47
6.5.3	Trie-based transactional hash map	47
6.5.4	Performance evaluation	47
Bibliography		49
List of Abbreviations		50
A Attachments		51
A.1	Source code	51

Introduction

Caching (sometimes called *buffering* or *paging*) is a well known technique in computing, which addresses the tradeoff between storage capacity and access speed by storing a portion of data in smaller but faster storage. It has been studied broadly both theoretically (as an algorithmic problem) and in practical use.

In this thesis, we would like to focus on *uniform caching for databases*, where the data resides on a disk drive, and is cached inside memory. *Uniform* caching means that we have a known limited number of elements that can be cached together. Once this limit would be exceeded, we need to evict records. One of our efforts is to minimize the number of times that a desired element is not present in the cache, also called the number of “*cache misses*”. Doing this optimally is, of course impossible, since we can’t predict what future queries will be given to our database.

It is worth noting that there are other versions of this problem. Some work with cached elements of non-uniform sizes, where the limit is given by a portion of memory. This is a significantly harder problem that has been proven to be NP-hard in the offline case, where all accesses are known in advance (Folwarczny and Sgall [2016]). As discovered by László Bélády (Belady [1966]), in uniform caching, it is optimal to always evict the element that will be accessed the furthest in the future, and that is easily implemented in polynomial time.

Focusing on databases further brings to the table

- the specific access patterns observed in database queries,
- production databases being highly concurrent systems often dealing with very high workloads,
- multiple queries being grouped into transactions, yielding further requirements for the caches.

Of course, different access patterns will be observed in different databases in production, and even at different times.

Optimizing the efficiency of a real database engine is a complex task. The performance depends on many different parameters and workload properties. The only way to really do it is to experiment with a real database system and a real workload, and we don’t have that option. We will give an overview of strategies that may be used for database caching. Firstly, we will look at what cache replacement policies are commonly used to try minimize the number of cache misses. Secondly, we will look at data structures that enable parallel access to the cached records by multiple threads.

We will also try to measure the properties we can measure. Thanks to my advisor, we can use some access logs from LDAP-based databases in production. This allows us to see how well the replacement policies perform with this type of workload, how important the choice of the replacement policy is, and perhaps whether our results are any different from those obtained in available publications. Then we will evaluate the in-memory performance of our simulations of the parallelization methods, written in the Rust programming language.

Both of these are highly specific measurements. We use a very limited amount of data from NoSQL databases, and also measure the overhead of data structures, which may be minuscule compared to other factors in a practical database, depending on many other implementation parameters. Nevertheless, they will give us additional insight about the discussed strategies, and accompany our discussion.

To keep the main text as fluent as possible, we will cover some topics separately in the Details chapter, and only reference them where relevant.

1. Cache replacement policies

In deciding which records we evict when it is necessary, the fact that we have no way of predicting the exact future accesses makes us rely on mere heuristics. In this chapter, we will briefly introduce some of the most popular ones, also adding a new experimental policy. The next chapter will provide their evaluation with access logs from actual databases in production.

As stated in the introduction, we only discuss policies for uniform caching. This means that the number of total records which may be cached together is fixed (under the presumption that stored values each occupy the same amount of memory, or at least reasonably similar amounts). The eviction strategy comes into play once this number is reached and another record is to be inserted into the cache.

The following sections aim to provide a simplified outline of the replacement policies. They won't explain the more complicated strategies thoroughly, but we offer citations of the papers where they were originally introduced and where their explanation is given in depth, for anyone interested.

At the start of each section, we also state where in the attached directory these policies are implemented.

1.1 LRU

Implemented in `src/lru.rs`

This policy always discards the least recently used (*LRU*) record in our cache. It may be implemented with a queue where records return to the queue's back on reaccess. We evict from the queue's front. To enable constant-time searching for the records, a hash map that stores pointers to the nodes of our queue can be utilized. In fact, we use a hash map for this purpose in all our implementations of cache replacement data structures.

It is not by chance that we discuss this policy as the first one, as its basic principle has worked as a basis for many later designed, more advanced, policies.

1.2 2Q

Implemented in `src/qq.rs`

The *2Q* strategy is especially relevant for databases, as it prevents large one-time scans of a subset of our DB from filling the cache with too many records that will not be accessed again any time soon.

It considers records “warm”, and therefore worth caching long-term, when they get reaccessed, but not just soon after their insertion. To do this, it uses two smaller queues (let's call them *Q1* and *Q2*) and a large main queue (“*M*”), each of these have their own capacities, which are hardcoded in the data structure, meaning that there is not just one *2Q*. We will discuss the choice of these parameters later (Section 6.1), and see how much of a difference there may be in two *2Q*s with differently set capacities.

When a new record is inserted into our cache, we push it into Q1. In practice, this will make the queue's capacity overflow, and so we pop from it into Q2. If also Q2 were to overflow, we just remove the queue's front element. Reaccesses to records inside Q1 have no effect. Reaccessing those inside Q2 makes them move into M. M itself acts just like the LRU. It returns reaccessed element to the back of the queue and once it would overflow (with a new record coming from Q2), we evict the record at the queue's front. This approach makes 2Q atypical, considering the introduction to this chapter, as it may evict records even long before reaching its full capacity.

The paper where this was originally introduced (Johnson and Shasha [1994]) explained it with the first two queues being one. Having 2 queues is where this policy got its name.

1.3 ARC

Implemented in `src/arc.rs`

The “Adaptive Replacement Cache” (established in Megiddo and Modha [2003]) uses another scan-resistant strategy. The structure utilizes two queues, L1 and L2, both further divided into top and bottom parts (extending our naming convention to T1, B1, T2 and B2). L1 contains records that have been accessed recently, but only once. L2 holds records which have been accessed at least twice recently. “Recently” meaning since the last time they have been uncached.

L1 and L2 can each hold as many elements as is the selected total capacity of the cache, but only the top parts (T1 and T2) contain resident records, the bottom ones are composed of non-resident records, meaning they only hold the record's metadata, but not the cached value itself. When the structure is full, it holds metadata for twice the given capacity of records, but only half of that is resident.

As we said, the combined size of T1 and T2 is the given total capacity, once the cache is full, but what portion of that is in one and what in the other, is given by a variable that starts at an even distribution and changes adaptively based on the observed access patterns. Reaccessed records always go to the back of the L2 queue (to its top, that is). Newly inserted, uncached, elements of course enter the top of L1 (we don't know about their past accesses). According to the numbers of records in each segments, the front (bottom) elements of T1 turn into non-resident B1 elements and the bottom elements of B1 are evicted altogether. T2 and B2 operate likewise.

Our implementation really stores T1, B1, T2 and B2 as four separate queues, and their respective capacities are maintained independently. Not having to hard-code any parameters to the structure, as well as the fact that implementing this is fairly easy, are attractive benefits of this strategy.

IBM holds a patent for ARC, which can make its use in databases distributed under many imaginable terms impossible¹.

¹More in Elein Mustain's blog post “The Saga of the ARC Algorithm and Patent”: <http://www.varlena.com/GeneralBits/96.php>

1.4 LFU

Implemented in `src/lfu.rs`

The abbreviation stands for “least frequently used”. We will need to store how many times each record has been accessed. The least accessed record will be evicted.

This can actually be implemented with a minimum binary heap (inside an array) to avoid linear-time iterations through all records. Each record in the heap holds its number of reaccesses. Before the cache’s capacity is reached, new records can be inserted via standard heap bottom-up insertions. Once the heap is full, we can make use of the fact that the removed element is at the heap’s “top” and the newly inserted record is guaranteed to have the minimum number of reaccesses, 0. I.e., we can just exchange the old, least used, element for the new one.

Our implementation will do one more thing though, and that is always placing elements the furthest down in the heap (towards more frequently used ones) that their access count allows, on both insertion and reaccess. This is to additionally approximate the LRU behavior and make an effort to not evict freshly accessed records before others that share the same access count.

Some use-cases may require regular decreases of access counts of records, to avoid having records accessed many times for a limited period of time stay in the cache for unnecessarily long, or even running into the case of reaching the limit number of accesses for too many records, degrading the whole policy. We don’t do this in our implementation.

1.5 2Q-LFU

Implemented in `src/qq_lfu.rs`

This is just a simple experimental modification to the $2Q$, where the large main “queue” behaves as the *LFU*, rather than the *LRU*. So we still have the same **Q1** and **Q2**, only the main queue, where records move from **Q2**, is now the *LFU* priority queue. The frequency counting for a record only starts once it enters this main (priority) queue.

1.6 LIRS

Implemented in `src/lirs.rs`

The “low inter-reference recency set” eviction policy attempts to tackle the shortcomings of LRU by not just considering the most recent access to an element, but two consecutive most recent ones (their inter-reference recency - “IRR”). Its thorough explanation is given by the authors in Jiang and Zhang [2002]. Here, we will try to explain the information most important to us.

To avoid the time-inefficient process of storing the exact delay between the two most recent reaccesses as a number for all records, we divide them into two sets, the *LIRs* and the *HIRs*, meaning low *IRR* and high *IRR*.

The *LIRs* are those records for which we are aware of low *IRR*, the *HIRs* are those for which we don't have any information about the recency of the second most recent access, and can't therefore say anything about their *IRR*. The principal part of our replacement structure will be a queue, similar to that of the LRU strategy, keeping note of a portion of the most recent accesses to records. Some *HIR* elements will be non-resident, meaning that they will only have an entry in this queue, keeping track of their most recent access, but their actual value won't be present in memory, similarly to *ARC*'s non-resident records.

For evictions, we always choose the least recently used *resident HIR* record, if it also still has a record in the recency queue, this makes it turn into a non-resident *HIR*. The *LIRs* keep their *LIR* status by getting reaccessed, a reaccessed *HIR* record becomes a *LIR* one, turning the least recently accessed *LIR* into a (resident) *HIR*.

It is good to note that the number of non-resident records is not explicitly limited by this strategy, and that in fact, when only inserting not-yet-seen elements for example, non-resident records never get removed, making their amount potentially unlimited. Cases where the working set is so much larger than the cache capacity, that this leads to significant amounts of thus occupied memory, may be unlikely for most real use-cases, but the lack of control over this is certainly a disadvantage, especially in cases where the *keys* by which we cache are of similar size as the actual cached *values*.

The total capacity of our cache is the sum of the maximum amount of the *LIRs*, and of the resident *HIRs*. Those two numbers are fixed for the data structure. This, like in the case of $2Q$, adds the requirement to define these constants. The cited article about this policy concludes, based on certain estimates, that making the *HIR* capacity as little as 1% of the total capacity is a good choice. We will look more into this later (Section 6.2), seeing what effect the parameter has on performance, and whether setting the *HIR* capacity this low is a good choice.

The cited article includes quite a thorough evaluation of this strategy, that also uses Postgres access logs, relevant to our subject matter and worth looking into.

1.7 CLOCK

Implemented in `src/clock.rs`

CLOCK is another very basic policy (described as early as 1968, Corbato [1968]). This one keeps records in a circular list called "clock" (potentially implemented with a modular array). Each record has a reference bit attached, unset on insertion. We keep a pointer, called "hand" by convention, pointing to one node of our list. New records are inserted behind this **hand**. Once evictions are necessary, we always check if the record pointed to by **hand** has its reference bit unset. If so, we evict this very element and move our hand forward in the list. If not, we unset the bit, move **hand** forward and repeat this process until a record has been evicted.

Reaccesses are handled simply by setting the record's reference bit, making them quite cheap operations.

We can see that in the presumably unusual case when all cached records have been reaccessed since the last insertion, **hand** needs to travel through all records to find a victim for eviction. However, an observation can be made about insertion, along with reaccess, having amortised constant time complexities. This is because our **hand** only needs to travel through a record if it has been reaccessed. So by noting that a reaccess leads to (up to) two operations instead of just one, all steps of an eviction, except for the final one, may be assigned to their respective reaccesses.

1.8 CLOCK-Pro

Implemented in `src/clock_pro.rs`

The *CLOCK-Pro* replacement policy (established in Jiang et al. [2005]) combines *LIRS*' emphasis on reuse distance with the compactness of *CLOCK*. Like *CLOCK*, it stores its records in the nodes of a circular linked list. This time, we call records either *hot*, or *cold* and cold records may be non-resident. Besides the reference bit, cold records will also hold a test bit, which tells us if they are in their "test period". Reaccesses are, again, handled just by setting the reference bit, but choosing a victim record for eviction will become quite a bit more complicated.

Instead of just one "**hand**", we will have three: **hand-hot**, **hand-cold** and **hand-test**. New records are inserted as resident cold ones behind **hand-hot**, with the test bit set, meaning they are in their "test period". Evictions are performed by **hand-cold**, when this happens, the hand travels through the clock and only affects resident cold nodes. It

- turns referenced cold records in their test period into hot records
- removes cached values of unreferenced records in their test period (making them non-resident)
- removes unreferenced records with their test bit unset altogether
- unsets the reference bit of the rest

By iterating this process, it always removes one cached value.

There may only be up to as many non-resident records as the total capacity of resident ones (so that there is twice the cache's capacity of nodes in our clock, at the very most). Once the number of non-resident records would be exceeded, **hand-test** is triggered, which removes the first non-resident record it finds, ending cold records' test periods along the way.

If turning a cold record into a hot one were to exceed the limit number of hot records, **hand-hot** is called, which unsets reference bits of hot records and when it finds one that has the bit unset already, it turns it into a cold record. It also acts just as **hand-test** towards cold records it passes, no matter how many non-resident records there currently are.

Reinserting a record that is stored as a non-resident (cold) record turns it straight into a hot one.

The *CLOCK-Pro* structure internally holds the maximum number of hot and resident cold elements it can hold (those two numbers always sum up to the total capacity of our cache) and it adapts these capacities by itself, based on the access patterns observed. Only setting the capacity of our cache, and no other parameter, is another advantage of *CLOCK-Pro* over *LIRS*.

Of course, if hot records get reaccessed often enough, **hand-hot** may have to travel through the whole clock when triggered. Unfortunately, unlike in the *CLOCK* policy, no amortization argument can be made about this, it is in fact possible to design access patterns that force our cache operations to take time linear in the number of records, on average. However, it is not unreasonable to say that such behavior is so specific that we may disregard it for practical use.

1.9 Random replacement

Implemented in `src/rr.rs`

Whenever a record needs to be evicted, this policy chooses the victim record uniformly at random out of all cached records. Originally, this policy got included in our evaluation to compare this blindly selecting policy with ones that build on the conscious knowledge of the caching problem, rather than because of it being popular for in-memory caching. But it didn't turn out quite as a lower bound for the quality of replacement policies, and considering the exact structure of our access logs, one can see why.

Furthermore, its simplicity (under the assumption of a good enough pseudo-random generator being available) could turn out very convenient for our next topic. Not having to take into account the access history might allow for a concurrent implementation of this policy that other policies can't have. However, this is an advantage which we don't elaborate on in this thesis.

2. Evaluation of the replacement policies

In this chapter, we will see how the policies listed above perform with accesses coming from real-life access logs.

Many similar evaluations have been made, and the interested reader is encouraged to look at those published by the authors of caching policies themselves in the publications cited in the previous chapter (i.e. Johnson and Shasha [1994], Megiddo and Modha [2003], Jiang and Zhang [2002], Jiang et al. [2005]). Nevertheless, many of those measurements use very artificial access patterns, often unrelated to databases, and so we take the time to check how our results may differ.

Thanks to my advisor, I was able to work with several (sanitized) access logs from 389 Directory Servers¹ (LDAP) in production. Two of them were high enough quality to obtain meaningful results. We will simply call them *Log 1* and *Log 2*. Of course, this is a very limited resource that cannot represent databases in general, but we will see if any interesting information can be obtained with them.

2.1 More about our measurements

The way we evaluate our policies is by counting how many times they fail to have a desired record cached, i.e. how many “cache misses” happen when we use them. We always start with an empty cache of a given total capacity, and then iterate through the accesses in our log, each time seeing if the current element is cached. If it’s not, we increment the miss count, and use our replacement policy to add the element into the cache. The very first access to each element inevitably leads to a cache miss, and so we don’t count these.

Two remarks need to be made about the parameterized policies. The *2Q*, *2Q-LFU* and *LIRS* policies each rely on additional hardcoded parameters. The way we choose them is described in the **Details** chapter (sections 6.1 and 6.2). On top of that, the *2Q* (and *2Q-LFU* as well) policy works in a way that may lead to elements being evicted even before the cache’s capacity is reached. This disadvantages the policy. If we wanted to evaluate it with the reasonable presumption that the replacement policy only works the best it could once the cache is full, we would need to start counting the misses only from that point. We will not adapt all our measurements because of this, however. So albeit this is a real weakness of the policy, our results must be taken for what they are. We cannot just say that *2Q* performs worse than another policy because of these miss rates, since we start the measurements with an empty cache.

To say more about the logs themselves, *Log 1* contains a total of 260 998 accesses to 1 001 distinct *keys*. The most accessed key is accessed a full 105 530 times alone (far more than the second most accessed, which is 2 337 times), so about 40 % of all accesses are to a single record. The median number of accesses

¹More information at <https://directory.fedoraproject.org/>

to a record is 209. Several keys get accessed the least number of times in the log, which is 14.

Providing more variety, *Log 2* contains a total of 213 551 accesses to 7 069 distinct *keys*. The largest number of accesses to a single key is also significantly larger than the second largest, but not quite as significantly (58 174 versus 6 368). The median number of accesses to a record is 6. Twenty-one records only get accessed once.

Although we don't have the necessary additional information to be sure, we should note that the very frequently accessed elements are likely to be the roots of the tree-based database structure specific to LDAP. It is unfortunate that we don't have access logs of a comparable quality from other types of databases.

As already mentioned, we can also count the actual optimal miss counts for each cache capacity for comparison, meaning how many we would get if we really evicted records the very best way possible. The best way is to always evict the record that will be accessed the furthest in the future, which has been showed by László Bélády in his article Belady [1966]. For this reason, the results are called the "Bélády optima".

A real database would cache pages, rather than database entries themselves. We have no information about the size and locality of the entries. Our access logs only provide their UIDs, which we would need to further map to the pages in a practical implementation. Therefore, we pretend that it is in fact entries that are cached directly, as an approximation of the real behavior.

Another remark relating to practical database implementation that we shall make before we move on, is that although our access logs aren't exactly small, this number of records could in fact be expected to fit into memory completely. We are also working with cache capacities well below what we would expect in practice, to see the replacement policies' behavior. The evaluation in the *ARC* article for example (Megiddo and Modha [2003]) uses access logs much larger than ours, which we can't compete with. On the other hand, the scale of the *LIRS*' evaluation (Jiang and Zhang [2002]) is quite comparable to ours.

2.2 Results for Log 1

Table 2.1 provides the exact results by policy and cache capacity (of course, the result for the *Random replacement* policy is one possible outcome that we happened to measure). Perhaps a plot serves better to visualize them. For a good sense of the relative quality of the replacement policies, the standard way of visualizing this is not by plotting the miss counts, but rather the hit rates. That is, what portion of the accesses lead to a cache hit (the element being present in the cache). Figure 2.1 gives us a plot like this. We can see, that the relative hit rates turned out to be so similar for the different policies, that they appear barely distinguishable on the plot.

	80	150	250	400	600	800	1006
LRU	125211	106919	81079	46068	13268	10697	0
2Q	116849	99278	75972	42167	12370	4952	3110
2Q-LFU	123086	105648	80670	43861	15492	5964	3246
ARC	120326	104938	77831	46240	12727	10239	0
LFU	124112	105171	79868	41596	5311	2722	0
LIRS	122045	102056	77288	41116	5884	4542	0
CLOCK	124675	106177	80748	45947	12972	7606	0
CLOCK-Pro	121691	102465	78321	43052	5457	5359	0
Random rep.	126758	108024	82806	48321	16611	7264	0
OPT	80767	58301	37545	17377	5212	2612	0

Table 2.1: Absolute miss counts for *Log 1*: The horizontal parameters determine the caches' capacities in each measurement.

The best result for each capacity is written in gray. The last line provides the *Bélády optima*.

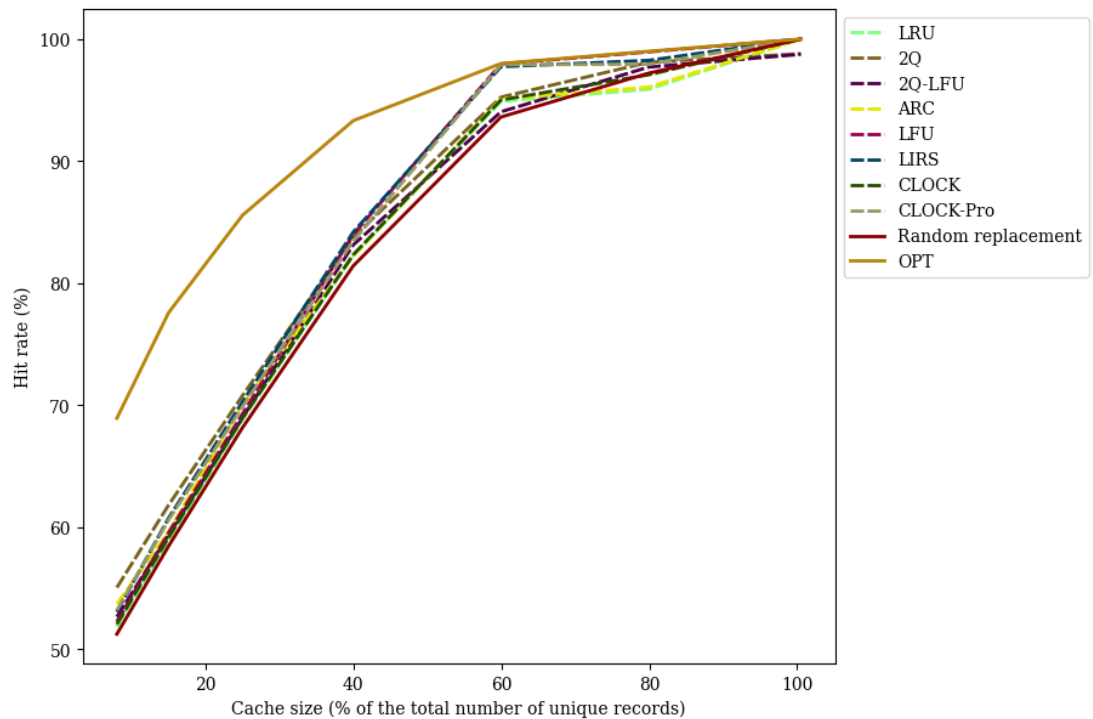


Figure 2.1: The hit rates for different replacement policies and cache capacities, measured with *Log 1*.

2.3 Results for Log 2

The results for *Log 2* are, again, described in Table 2.2 and a hit rate plot in Figure 2.2.

	80	141	232	353	494	650
LRU	30061	27344	23879	22676	21617	20307
2Q	26355	24462	22810	21257	20092	19371
2Q-LFU	31913	28386	26188	23219	21653	20499
ARC	28836	24785	23476	21736	20654	19703
LFU	58131	54734	49335	43715	37378	33559
LIRS	29237	24658	23359	21755	20716	19740
CLOCK	30041	26873	23906	22592	21491	20387
CLOCK-Pro	29611	25552	23556	22327	21060	19914
Random rep.	37977	32247	28449	25853	23849	22414
OPT	21049	18622	16529	14540	12843	11380
	850	1107	1768	3537	5660	7075
LRU	19054	17815	14447	8021	2166	0
2Q	18524	17685	15751	12038	8514	6149
2Q-LFU	19296	18201	15732	10314	4913	1986
ARC	18499	17024	14624	7865	2256	0
LFU	30186	25706	20662	11414	3754	0
LIRS	18492	16967	14303	8384	2465	0
CLOCK	19015	17721	14694	8343	2442	0
CLOCK-Pro	18770	17419	14448	8396	2535	0
Random rep.	20884	19116	15650	8673	2279	0
OPT	9908	8405	5640	1764	0	0

Table 2.2: Absolute miss counts for *Log 2*: The horizontal parameters determine the caches' capacities in each measurement.

The best result for each capacity is written in gray. The last lines provides the *Bélády optima*.

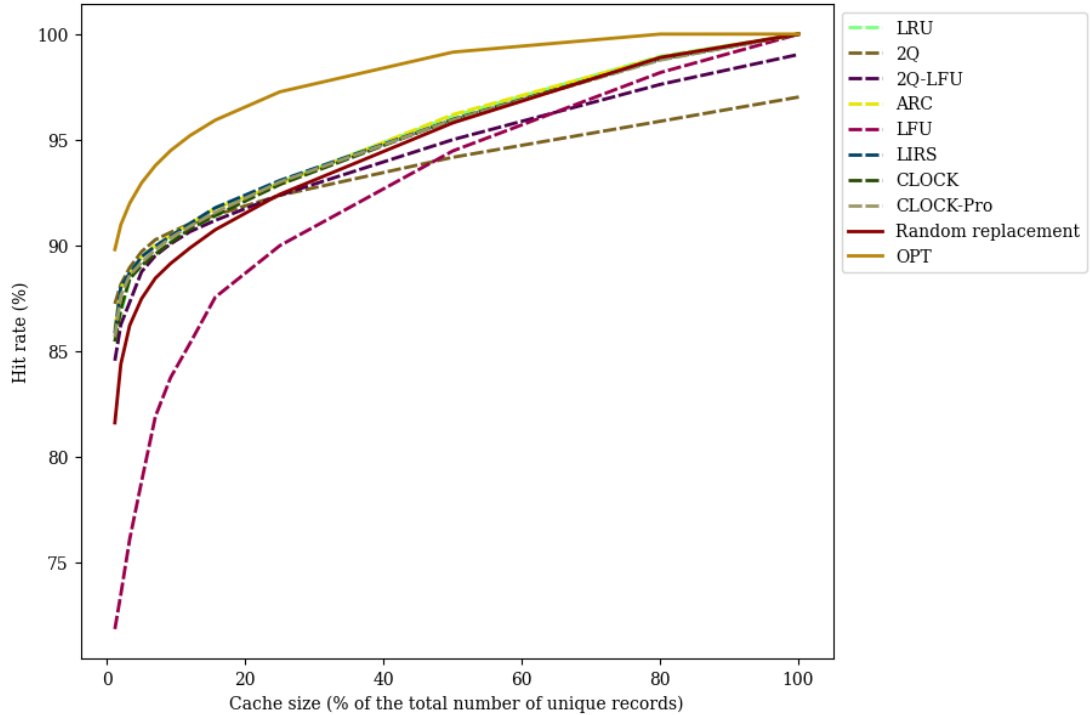


Figure 2.2: A hit rate plot for *Log 2*

2.4 Discussion

In the case of *Log 1*, we have seen differences between the absolute miss counts by the different policies, but we can conclude that they aren't too significant in relative terms. This is especially apparent when looking at the hit rate plot.

With *Log 2*, we get three outliers, *2Q*, *2Q-LFU* and *LFU*. In the case of our experimental *2Q-LFU* policy, we could simply say that it didn't prove itself as a good modification of *2Q*. We have already discussed the disadvantage *2Q* has in these measurements. It actually performed very well with the lower capacities, but its shortcomings showed with the higher ones. And we have seen how non-trivial it is to set its parameters optimally (Section 6.1). And for the *LFU*, it is no surprise that it didn't perform well here, based on the measurements available in the articles. In fact, its rather great performance with *Log 1* comes as a bigger surprise than this. Therefore, even here we may conclude that the policy choice doesn't in fact make such a difference, in the case of the policies where we would expect it.

Perhaps the real surprising result, in the light of the *ARC* article (Megiddo and Modha [2003]), is how little improvement (and even decline at times, actually) the *ARC* policy provided over *LRU*. We should repeat that the article evaluates on a much larger scale of workloads, however.

If we want to go into detail with the miss counts, an interesting comparison can be made between the *LFU*, which performed quite surprisingly well on *Log 1*, but not *Log 2*, *ARC*, which was conversely well suited for *Log 2*, but *Log 1* not that much, and *LIRS*, which was probably the most consistent policy overall (if we can call any performance consistent with just two samples). The comparison is visualized in Figure 2.3.

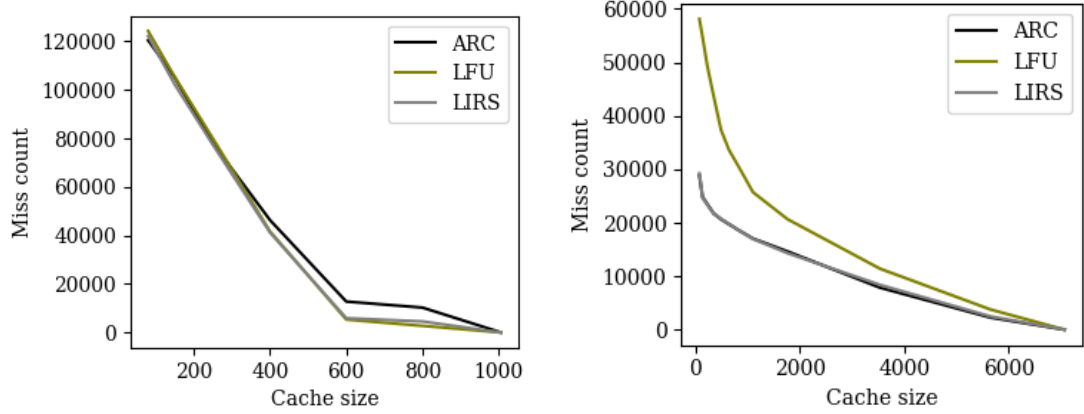


Figure 2.3: Comparison of the miss counts by *LFU*, *ARC* and *LIRS*. *Log 1* results on the left, *Log 2* on the right.

Seeing how well *LIRS* performed, we may also appreciate *CLOCK-Pro*, as it served as a really good approximation of *LIRS*, with a controlled number of non-resident records and, perhaps more importantly, no need for a hardcoded parameter.

3. Strategies for concurrency

Moving on to our second major focus, let's discuss how an in-memory cache can make use of the multiple cores of a typical database system.

Of course, this is not just about making the caching data structures accessible to multiple threads. The caches are used to perform database transactions, which need to maintain the ACID properties at all times. All four strategies that we discuss here allow for multiple ways of implementing them, so here we only give an outline of their principles. The next chapter will be about their specific implementations, or rather how we simulate them in our benchmark, which is the subject of the chapter after.

A brief overview of how some open-source database implementations handle this is given in section 6.3 of the Details chapter.

3.1 Single lock

The simplest approach is having a single lock, protecting the data structure and ensuring exclusive accesses to it. Figure 3.1 shows a model diagram of how this might behave in a concurrent system.

Of course, from the viewpoint of concurrent data structures, this is even worse than only operating sequentially in a single thread, but using this makes sense if manipulating the cache is just a fraction of what a thread needs to do to handle a transaction.

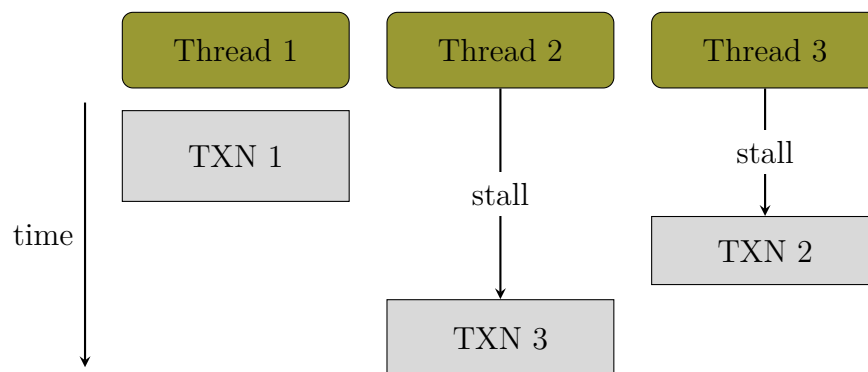


Figure 3.1: Time diagram of threads obtaining exclusive accesses to a cache to handle queried transactions.

3.2 Associative caches

A little more sophisticated approach is dividing the cache into multiple caches, “slots”, of lower capacities, where each one is used to store a specific subset of records. Naturally, we make an effort to divide records into slots as evenly as possible, for example by hashing the records’ keys and dividing the hash set uniformly.

This way, a thread doesn’t need exclusive access to the whole cache, but “only” to the slots where records affected by (or affecting) the processed transaction reside, introducing overhead to grant this. An incorrect implementation of this may easily cause deadlocks.

Again, the diagram in Figure 3.2 attempts to visualize how this may function in a concurrent system.

We analyze this approach a little more in Section 6.4 of the Details chapter.

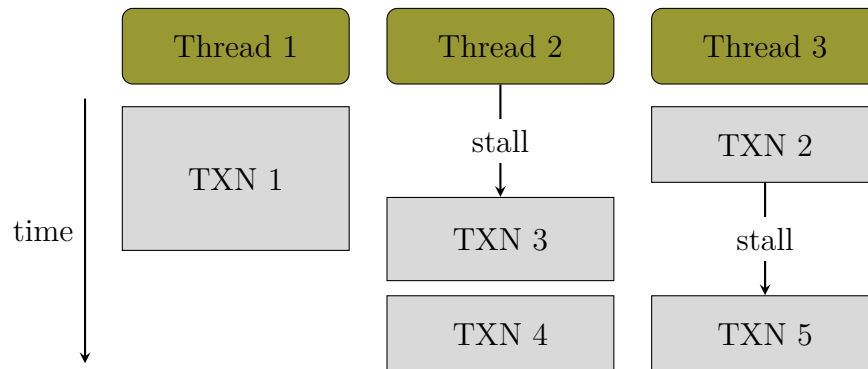


Figure 3.2: Time diagram of threads accessing associative “slots” of our cache exclusively. Transactions may proceed in parallel, as long as they access different slots.

For example, we assume that TXN 3 needs to access a slot that TXN 2 needs as well, but doesn’t share slots with TXN 1.

Fine-grained locking is a related approach, which instead of “slots” only directly locks the very records themselves. Or in practice, their pages, more probably.

3.3 Small cache for each thread

Imitating the CPU L1 caches, we might also give each thread its own small cache that it can access at any time. Two major problems come with this approach.

The first is that now we need to distinguish between the different types of transactions. If we had just read-only transactions, we wouldn't need to worry about anything more. A transaction that modifies records in our database, however, both requires additional management to avoid conflicting modifications by two threads, and a way to warn the other threads that some data has changed globally (which may, of course, mean multiple things for different implementations) and that they may need to update, or invalidate some of their cached data.

The second problem is that although our evaluation of the replacement policies earlier didn't show too much of a difference in hit rates amongst different policies, we can clearly see a difference when providing the cache with a fraction of its capacity. Depending on how costly a cache miss is, how much we insist on minimizing the latency of operations over the total throughput, and what kind of workloads we expect (and especially how often modifications are expected), a developer would need to think over if this approach is worth considering at all for a given database system. It is certainly not a good universal choice for a cache implementation.

Another problem in addition is that implementing this correctly, to really guarantee compliance with the desired properties of transactions, can become very difficult, leaving a lot of room for potential error.

Figure 3.3 shows a really simple time diagram for this strategy. It doesn't consider the problem of avoiding conflicting writes, as that can be implemented in different ways.

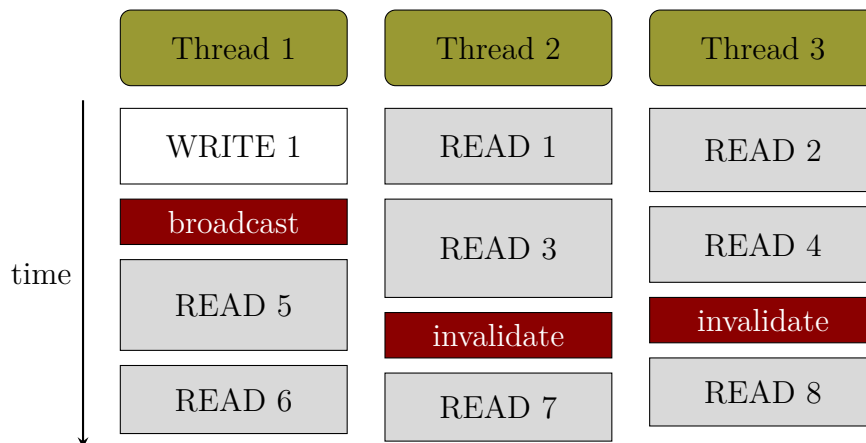


Figure 3.3: An example diagram for this strategy. After a write transaction, Thread 1 broadcasts to other threads, that they may need to invalidate certain records they hold.

Notice that we are expecting Read 3 and Read 4 to terminate successfully even after the broadcast, and their respective threads performing the cache invalidation only after they are finished. This may not always be the case, depending on the particular implementation.

Surely, this strategy leads to a whole variety of approaches. We could combine

it with the locking approach and have pairs or larger groups of threads share a cache, allowing for a less significant decrease in capacity for each cache (in imitation of the CPU L2 caches, we could say).

3.4 Concurrently readable cache via MVCC

Data structures that allow being accessed by multiple readers in parallel without a risk of deadlock, while also allowing (up to) one thread to perform modifications, are called concurrently readable.

Making our cache concurrently readable is an approach tailored specifically for database transactions. It can be implemented via the Multiversion Concurrency Control (*MVCC*) method. Each thread may potentially be accessing a different version of our cache, but each version provides a valid set of cached records. Only the thread with modification privilege can modify our data structure and create a new version of its data.

Again, we provide a time diagram of how this may work in Figure 3.4. Data structures like this are also commonly called “transactional”. As mentioned, we will discuss how this may be implemented in the next chapter.

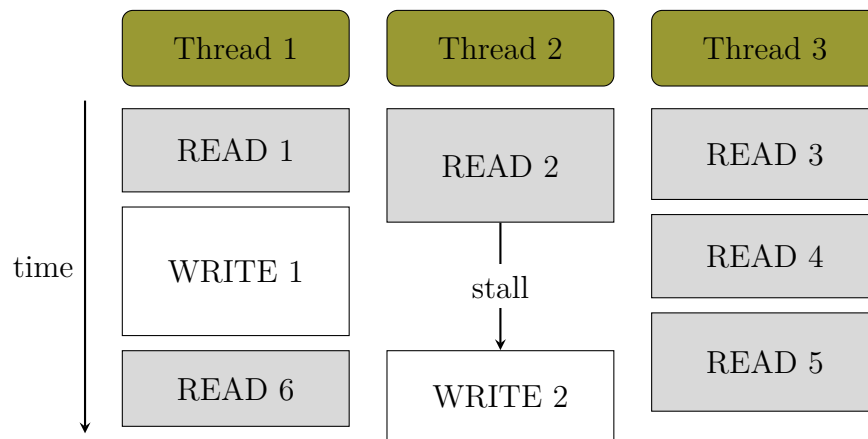


Figure 3.4: A time diagram for the concurrently readable approach to database caching. **Thread 2** needs to stall to gain the unique modification privilege for a write transaction, but read-only transactions may access the cache at any time, always gaining access to a valid “snapshot” of the cached data.

4. Our simulations of the concurrent strategies

As we said, there are multiple ways of implementing the strategies. Let's talk about how we simulate them for the sake of our benchmark. From this point, we will call the “*Single lock*” approach simply “*LOCK*”, the one that uses associative caches “*ASSOCIATIVE*”, the one with small caches per each thread “*PER THREAD*”, and the one with a concurrently readable cache “*TRANSACTIONAL*”.

Again, we must keep in mind that our cache (or caches) needs to be used in ways that maintain the ACID properties of the transactions we process. Durability doesn't concern us, as that isn't the task of the cache, and neither does consistency, which relates to the logical model of our database (the cache doesn't need to provide an additional mechanism for transaction rollbacks). However, we need to ensure the atomicity and isolation of transactions.

4.1 LOCK

This one is very straight-forward to simulate. Our cache will be protected by the standard `Mutex` and threads will need to lock it for each transaction they perform.

4.2 ASSOCIATIVE

A thread must be able to lock all the slots that contain the records accessed by a transaction, before it performs the transaction. A wrongly implemented ordering of the locking can easily lead to a deadlock, and so we use the following cautious approach. We have one “main” `Mutex` and one for each slot, slots themselves are just separate caches of smaller capacities, and each stores a specific fraction of the records based on a uniform division of the records' keys' hash set. When we acquire unique access to slots for a certain set of records, we

1. Lock the main `Mutex`, meaning that we stall until we are the only thread locking slots.
2. One by one, lock the slots we will need.
3. Unlock the main `Mutex`, i.e. allow other threads to start locking slots we don't need.

After the transaction is finished, we may release the locks to slots in any order.

A possible extension of this could be making threads attempt to lock the slots necessary for their transaction without locking the main lock, in case that they only need unique access to a “small” number of slots, because then there is a high probability of them being able to lock all (and they would only need to lock the main lock if they weren't). If a thread fails at this, it can simply unlock the slots it managed to take, and proceed to acquire access in the standard way.

If we were to implement the mentioned fine-grained locking strategy, we would need to solve the issue of updating the replacement policy’s logic structure. The structure is generally near impossible to parallelize. An example of what we could do is having a concurrent queue of demands on the logic structure. The threads would access exactly the records (or pages) they need, and send the cache hit information through the queue, perhaps as well as requirements for insertions into the cache. The logic structure would independently update its state based on the contents of the queue.

4.3 PER THREAD

Here, each thread has its own cache, the capacities of these caches are uniform.

We have a single “modification `Mutex`” that threads need to lock when they simulate modifying transactions, as a cautious means against conflicting modifications by multiple threads. When a thread simulates the modification of records, it keeps track of keys to all records that it modifies. We imagine that at one point, the modifications take effect globally, which other threads need to be warned about. There are multiple ways to broadcast such message from one thread to the others. The way we do it is really just a simulation that one wouldn’t replicate when implementing a real database. We dedicate a special thread that just listens to an `mpsc` channel from the worker threads, which may send it invalidation requests containing the recorded keys of elements that have been modified. This dedicated thread distributes the requests to all threads except the ones that requested them. Worker threads carry out these requests before each new transaction they perform, by evicting all the invalidated records.

When a thread processes a search-only transaction and stumbles upon a cache miss, it needs to process invalidation requests as well and potentially restart the transaction over. This way, we always make sure that each transaction is processed with a valid set of records. The modifying threads do so by being the only ones that may change the global state, and the reader threads by always carrying out their whole transaction with only the content in their cache, which is valid at all times. Note that if this was used in practice, large transactions that need to access more records than a thread’s cache can hold, would make this method inapplicable, even more so if we used the $2Q$ replacement logic for our caches, which could keep evicting records that we need for our transaction.

Of course, the process could be quite different in a real database, which would likely postpone a transaction that needs a missing record until the record is retrieved from the disk. But again, we will be trying to evaluate just the in-memory performance of these strategies.

4.4 TRANSACTIONAL

We must first point out that this approach isn’t based on any article or scheme used in existing databases. It originates in the assumption, that transactional data structures are practical for database implementation, and the observation that a transactional cache can in fact be implemented, for example in the following way. This approach hasn’t proven itself in practice (or doesn’t seem to be used

anywhere that I'm aware of). So how can we make a cache concurrently readable? For our implementation, we will need to have a concurrently readable hash map. How that may be implemented is discussed in Section 6.5 of the Details chapter.

Once we have a concurrently readable hash map, which allows for obtaining read-only snapshots of its data at any point, and also for up to one thread privileged to modify it, this is how we can use it to implement a transactional cache.

We separate the cache replacement logic and the data that is really cached, by using the concurrently readable hash map for storing all our records. To provide a read-only snapshot of our cache, we may simply provide a snapshot of the hash map. Only the (up to) one modifying thread will have access to the cache replacement logic of our cache, which allows us to implement that as a sequential data structure.

When a modifying thread accesses records in the cache, it obtains them from the concurrently readable hash map, and performs a cache hit on the separate replacement logic data structure. When it inserts a new record to our cache, and the cache has reached its capacity, the logic structure tells it what record to evict from the hash map, after that, the thread inserts the new record (of course, also inserting its entry to the logic structure).

Modifying records is trickier, because that forces us to reinsert the records, rather than modify them in place, as the reader snapshots of the hashmap could access these modified records. So what we do in the case of such modification is that we reinsert the record into our hash map in its modified form, and perform a cache hit in our logic structure.

To use this transactional cache in our simulation, we must notice that threads with its read-only snapshots can't affect the cache's replacement logic, which could lead to our cache evicting records that are in fact "warm". Therefore, we will have all threads keep track of what records they have hit with just the non-modifying snapshots, and they will perform their cache hits in the logic structure once they acquire the modification privilege. We will also set a hardcoded maximum of how many elements the threads may thus record. If too many cache hits by a "reader thread" don't propagate globally, we make the thread acquire modification privilege even if otherwise not necessary, and perform the hits, to avoid warm records being evicted too soon.

If a thread performing a search-only transaction with a read-only snapshot to the cache stumbles upon a cache miss, it has no option but to acquire the modification privilege and insert the missing record. Once more, a real database implementation could do many things differently, and would most probably need to adjust these processes to handle disk I/O asynchronously, but our focus now will be the in-memory performance.

Since the threads need unique modification privilege to perform modifying transactions, we may also propose the following way of using the transactional cache (we don't simulate this one in the next chapter).

Only one thread would have the modification privilege, and it would have it at all times. Only this thread would perform modifying transactions. Other worker threads, which would serve to perform read-only transactions exclusively, would communicate with the modifying thread using a concurrent queue. They would send the cache hit information through it, and also requests for insertions into the

cache. Besides performing all the modifying transactions, the modifying thread would also have to listen to these requests and update the cache accordingly, occasionally committing the new versions for the reader threads to access.

We would probably also have to provide a way for the modifying thread to inform reader threads that it has inserted a record that they requested to be inserted.

5. Concurrency: Evaluation

We will now attempt to evaluate the in-memory performance of our cache data structures and the approaches to concurrency. This means that we will neglect many practical aspects of database implementation, like disk I/O (and its asynchronous performance tuning) and even the real operations transactions perform on cached data, and will only measure the overhead the bare data structures cause.

This is a very specific focus. In practice, the concurrent strategies most importantly serve to allow threads to process transactions in parallel. Such transactions may take long to execute compared to the time needed to just allow them to access records in our cache. And in the likely case that we can't fit all records we need to work with inside our cache, the latency may be mostly due to disk I/O, which may make approaches like *PER THREAD* completely unusable (as having smaller caches leads to significantly more cache misses, which we have seen). However, we will see how our data structures contribute to the overall throughput of the whole process.

What we will do exactly, is spawn several worker threads and distribute a workload to them using a concurrent queue of references to transaction objects. The worker threads will pop the references out of the queue until the simulated execution of all the transactions is finished. The transaction objects only hold a list of keys of records to access and information about which ones get modified by the transaction. The way workers simulate the execution is described in the previous chapter. As stated, we don't perform any actual operations on the cached values, and in fact, the values will be empty. On top of that, on a cache miss, a thread will simply insert the missing record instantly. This way, we really only measure the overhead induced by our data structures, as declared.

The caches will be given the same capacity, no matter the particular concurrent approach and its practical memory overhead (in *PER THREAD*, this means that the caches split this capacity evenly). Of course, the transactional cache needs to keep records specific to the active read snapshots inside memory, which would make us, in practice, assign this strategy a lower capacity, but that doesn't concern us in the evaluation of the measured properties.

5.1 The workload and measurement tools

Unfortunately, *Log 1* only states the UIDs of records that get accessed, but *Log 2* also informs about what transactions they are accessed by. Therefore, we will work with the workload from *Log 2*. Some information about the log is given in Section 2.1. We can add that it is composed of 72 555 transactions, 71 969 of which access just 3 or less records. The median number of accesses to records by a transaction is 3, with the mean just below that at about 2.94. Nevertheless, there are very "long" transactions too, 16 perform over seven hundred accesses (735 being the very most). **Only 7 of the transactions modify any records** (all seven happen to access three records and modify exactly one of them). We set the capacity of our caches to about half of the number of unique records accessed

by the log.

We perform the measurements on a machine with 40 *Intel Xeon Gold 6230* cores and over 60 *GiB* of memory, which has proven to be significantly more than the measurements utilize in CPU and memory usage.

To measure the execution times with an amount of statistical confidence, we use the **Criterion**¹ micro-benchmarking library. Criterion uses linear regression to produce estimates of the “typical” times the measurements take. Besides determining the one “typical” value, it gives us the lower and upper bound of a confidence interval for that value, based on a configurable confidence level. We use the default confidence level of 95 %. In the results that we will state here, we naturally use the typical execution times, and we also inform about the wideness of the confidence interval to get an idea about the observational error.

5.2 Results for single thread execution

Before evaluating the concurrent strategies, let us see how the cache replacement data structures behave in a single thread.

We perform the same exact setup as we described for our concurrent measurements, except that we only spawn one worker thread, which works with a sequential cache.

The measured results are given in Table 5.1. These are results for the different replacement policies, and for increasing cache capacities. The bounds of the confidence intervals always fit in under 4 % away from the typical value. We can see that the execution times generally tend to decrease slightly with the increasing capacity of the caches, showing that the costs of multiple inserts outweigh those of operating with a larger caching data structure, at this scale. Even though we are caching empty values here (and 16 bit keys of the records, that we use).

We also perform a special “NULL” measurement, which performs all operations the other measurements need to perform in addition to the ones specific to their data structure (i.e. things like receiving the workload from the concurrent queue and iterating through the list of keys to accessed records), so that it effectively measures the overhead of the measurement itself. This measured overhead appeared to be almost exactly 1 millisecond (0.96 *ms* more specifically, with the confidence interval bounds within 1 % away from this value).

¹<https://docs.rs/criterion/0.3.4/criterion/>

	200	1000	2000	3456
Random rep.	6.5	5.4	5.0	4.8
LRU	8.1	7.4	7.6	7.2
LFU	10.9	11.1	10.4	10.5
CLOCK	7.4	6.6	6.9	6.5
CLOCK-Pro	9.4	9.1	8.3	8.1
LIRS	19.8	19.1	18.9	18.4
2Q	9.5	8.5	8.7	8.1
2Q-LFU	12.4	11.3	11.6	11.3
ARC	13.3	13.0	12.3	11.9

Table 5.1: Execution time estimates in milliseconds for the single-threaded measurement.

The horizontal parameters specify the set cache capacity (in the number of elements). The vertical ones are the evaluated caching data structures.

The highest capacity we measured here, 3456, is the capacity we use for the concurrent measurements (to clarify, this number was chosen for its convenient prime factorization that will allow us to split the caches uniformly in the *PER THREAD* and *ASSOCIATIVE* measurements, while also actually being just about half the number of unique records in *Log 2*).

5.3 Results for parallel execution

Now we finally measure the execution times for the concurrent data structures. First, we should note that all four measured concurrency strategies are implemented with the *LRU* replacement logic, which we choose for its simplicity and the solid hit rate we obtained with it in its evaluation in Chapter 2. We have now seen that also its execution performance in a single thread is very solid, but both these properties aren't too significant in relative terms compared to the other replacement policies, as we see in our results. In fact, any replacement policy that we discussed could be used to implement our concurrent strategies, so the choice of *LRU* is more or less arbitrary.

The results are written in Table 5.2. We measure all the described strategies with increasing worker thread counts. It is worth remembering that the sequential cache managed to go through the whole workload in just 7.2ms , which not even the fastest concurrent approach, *PER THREAD*, matches. Once more, the concurrent approaches allow us to perform transactions in parallel, for the cost of such performance overhead. The results here let us learn more about the data structures, and show the overhead they bring in exchange for their benefits, which we discussed in Chapter 3.

	2	4	8	12
LOCK	47.4	44.0	47.7	55.5
ASSOCIATIVE	174.2	173.0	191.7	203.9
PER THREAD	13.9	12.5	15.4	17.0
TRANSACTIONAL	74.7	83.7	88.0	93.0

Table 5.2: Execution time estimates in milliseconds for the concurrent measurement.

The horizontal parameters state the thread counts. The vertical ones are the evaluated concurrent strategies.

The confidence interval bounds were in under 3% away from the stated values.

5.4 Additional workloads

To help with the interpretation of these results, we generate two supplementary workloads with very specific properties.

We will call the first one “*sequential*”. It will feature transactions that each only access one record. The accessed keys will repeat the sequential iteration of numbers between 0 and 499, 500 times around (so there will be 250 000 “single access” transactions in total). Right about 1% of the accesses will randomly be marked as modifying, which is significantly more modifications than just the 7 amongst 213 551 total accesses in *Log 2*.

The increased number of modifications doesn’t affect the *LOCK* and *ASSOCIATIVE* approaches, but we can expect a drop in the performance of *PER THREAD*, which will be forced to lock the modification *Mutex* more often and perform a higher number of invalidation requests. It could also affect the *TRANSACTIONAL* approach slightly, as only one modifying transaction can be processed at a time.

We will leave the total capacity at 3456, which will make cache evictions appear only in the second two measurements of *PER THREAD*. Based on the analysis in Section 6.4, the *ASSOCIATIVE* approach should be well suited for this type of workload.

The results for the workload are given in Table 5.3, the sequential (single threaded) *LRU* managed to finish it in 9.1 *ms* (with the confidence interval being well under 1% wide from this).

	2	4	8	12
LOCK	85.9	94.8	114.4	133.0
ASSOCIATIVE	99.4	212.1	253.4	260.9
PER THREAD	29.5	39.1	52.8	60.3
TRANSACTIONAL	67.0	74.8	82.4	90.9

Table 5.3: Execution time estimates in milliseconds for the additional concurrent measurement with the “*sequential*” workload.

The horizontal parameters state the thread counts. The vertical ones are the evaluated concurrent strategies.

The confidence interval bounds were in under 5% away from the stated values.

We'll call the second artificial workload “*random*”. It will feature 40 000 search-only transactions. Each will access exactly five records. The accessed keys will be generated (uniformly) at random out of a range of exactly 3 456 elements, i.e. the total cache capacity, which we keep. The results for this workload are written in Table 5.4. The sequential *LRU* finished its simulation in 6.7 *ms* (the confidence interval bounds, again, well under 1 % of that wide). Notice that even without the need of broadcasting invalidation requests, the *PER THREAD* approach still couldn't outrun the sequential cache with the 12 threads.

	2	4	8	12
LOCK	46.4	37.0	35.4	38.0
ASSOCIATIVE	158.2	162.7	162.4	154.1
PER THREAD	19.2	16.6	9.1	7.8
TRANSACTIONAL	25.0	27.7	29.7	31.9

Table 5.4: Execution time estimates in milliseconds for the additional concurrent measurement with the “*random*” workload.

The horizontal parameters state the thread counts. The vertical ones are the evaluated concurrent strategies.

The confidence interval bounds were in under 2 % away from the stated values.

We should remember, that the artificial workloads serve purely to help us understand the in-memory behavior of our data structures. Only *Log 2* is based on the workload of a real database system.

5.5 Interpretation of the results

What can we conclude about our data structures? The cost of locking is apparent. The *LOCK* results show the simplest effect of this. Besides the disadvantages discussed in section 6.4, the *ASSOCIATIVE* approach is slowed down by all the locks required. Even *PER THREAD*, which only requires locking for modifications, showed noticeable overhead of the necessary inter-thread messaging, and we have seen in the sequential measurement that the higher miss rates caused by splitting the cache capacity may not just affect latency unfavorably, but even the in-memory throughput because of the costs of insertions.

We must not forget that even the transactional cache needs a lock at the instant of an access to its hash map, as described in section 6.5. It is safe to say that in comparison to the other cache, the *TRANSACTIONAL* one performed almost surprisingly well in these measurements for all its benefits. And we may note that there is still room for its optimization, as stated in Section 6.5.

Although these are relevant observations, let's not forget that we are only working with specific simulations of the strategies, and that also our workload is limited.

Conclusion

We have explored several strategies for database caching.

Firstly, we discussed replacement policies for *uniform* caching. In practice, database caching is in fact non-uniform. That's because what gets cached are pages. And the database entries themselves, which are stored inside them, have variable sizes. Nevertheless, as developers, we are forced to work with uniform caching schemes.

We extended the existing collection of miss counting measurements with two modest-scale access logs from LDAP-based databases. Those didn't show too significant relative performance differences between replacement policies. However, even though they came from the same type of database, they yielded quite distinct behavior when comparing their results between each other. This suggests how challenging it is to design a universal policy, which is even more clearly visible in the differences we observe with the parameterized policies in Sections 6.1 and 6.2.

Secondly, we examined the methods for safe accesses to the cached records by multiple threads in parallel, and tried to evaluate the in-memory performance overhead these methods cause.

Perhaps most interestingly, we studied the possibility of using a *transactional* cache, which we believe is promising for the use in databases specifically. We have therefore discussed a selection of its implementation aspects and provided additional experiments with two versions of a transactional hash map in Section 6.5. An issue particular to the transactional approach is the difficulty to control the exact number of records cached at a time, as we must preserve all records of its active read snapshots, whose size depends on the workload and the exact way we use the cache. This approach could be expanded on in future research.

6. Details

This chapter is dedicated to the curious reader. It contains information about details that aren't crucial for our main text, but play a role in our discussion.

6.1 Parameters for the 2Q and 2Q-LFU replacement policies

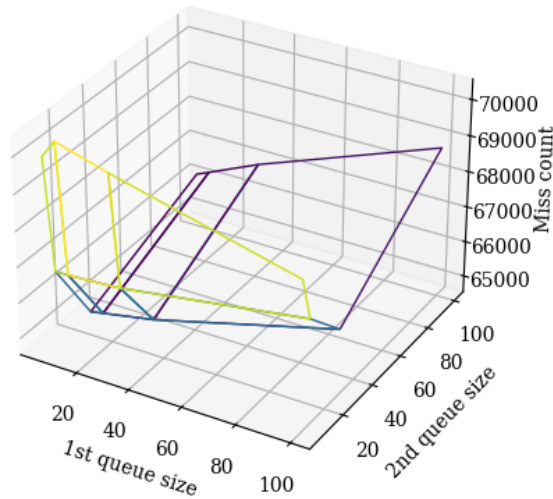
To determine our choices for Q1 and Q2 sizes in our main measurement, we make separate measurements just for these two policies. This also serves as a reference for how different the results could be with a different choice.

The observed behavior is visualized in Figures 6.1 through 6.12. In each of the tables, Q1 sizes rise with the vertical parameters and Q2 with the horizontal ones. The better the results for a pair of parameters, the lighter the number in the table.

We can see that finding the right choices for the parameters isn't easy. The behavior changes with the increasing cache capacity, and is quite different for the two access logs (that are coming from the same **type** of database). Visibly, a wrong choice can lead to miss counts several times worse than possible. Our heuristic choices for the measurements in Chapter 2 are the following:

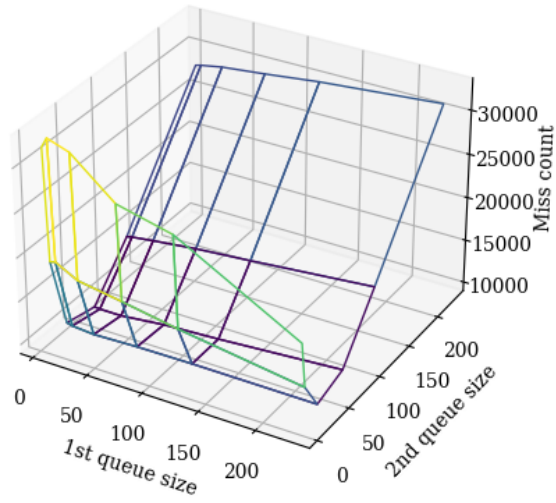
2Q's Q1 size is set to 10, the size of Q2 to a sixth of the total capacity.

For *2Q-LFU*, Q1 and Q2 sizes are both set to a fifth of the total capacity.



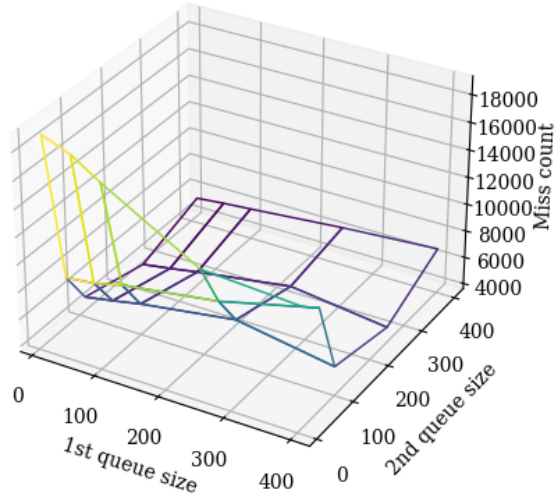
	5	10	30	100
5	69933	66603	64661	66083
10	70441	66605	64732	66238
30	70009	66695	65005	66878
100	68695	67450	66359	68767

Figure 6.1: **2Q** parameters: Total capacity: 300; Obtained with *Log 1*.



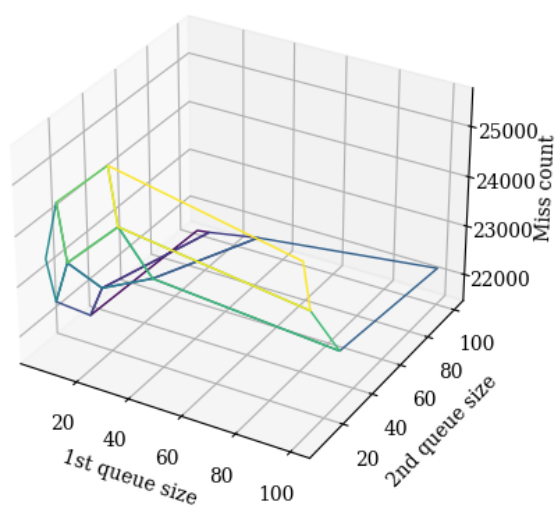
	5	10	30	70	120	230
5	31942	18839	10397	9517	14587	28122
10	33111	19012	10221	9460	14710	28281
30	32252	17654	10017	9458	15029	28717
70	28077	16810	10186	9982	15585	29226
120	26549	15673	10246	10308	16086	29951
230	18874	13583	10180	11241	17301	31120

Figure 6.2: **2Q** parameters: Total capacity: 600; Obtained with *Log 1*.



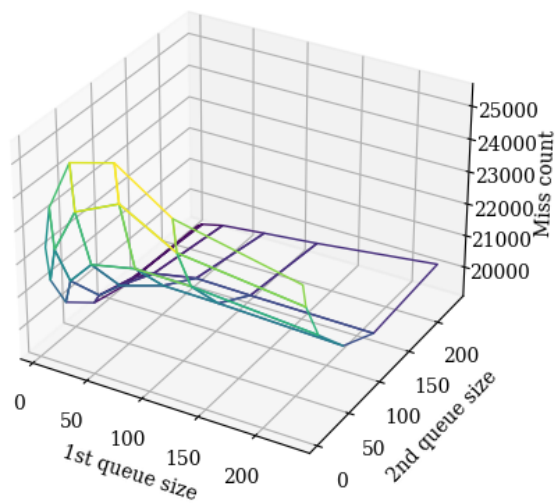
	10	55	100	250	400
10	19012	7711	5132	3923	5524
55	18197	7934	5506	4245	5726
100	16850	8435	5917	4607	5889
250	12830	9430	6937	5653	6479
400	12370	11140	5745	4756	6939

Figure 6.3: **2Q** parameters: Total capacity: 1000; Obtained with *Log 1*.



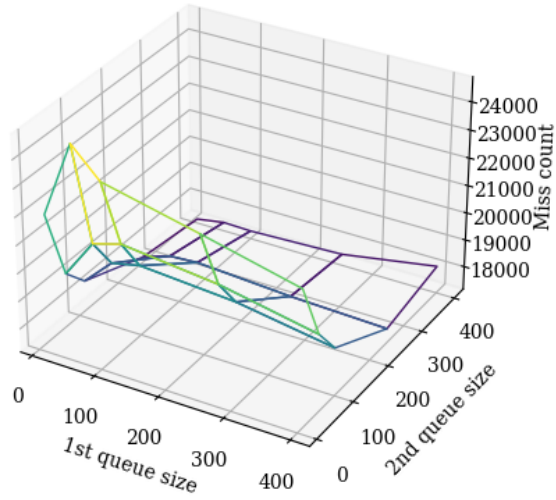
	5	10	30	100
5	23491	22494	21651	21538
10	24673	23354	22295	21574
30	25667	24368	22808	21764
100	24926	23858	22500	22209

Figure 6.4: **2Q** parameters: Total capacity: 300; Obtained with *Log 2*.



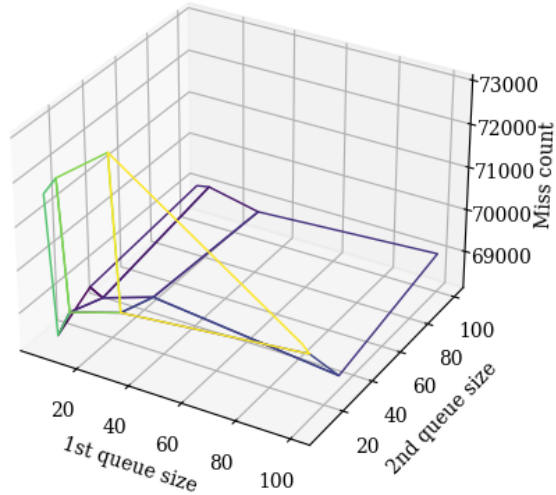
	5	10	30	70	120	230
5	22407	21320	20270	19524	19239	19201
10	23655	22305	20989	19816	19480	19308
30	25173	23639	21696	20330	19823	19467
70	25541	24265	21982	20747	20083	19607
120	24405	23321	21958	20719	20055	19775
230	23595	22861	21658	20582	20031	20214

Figure 6.5: **2Q** parameters: Total capacity: 600; Obtained with *Log 2*.



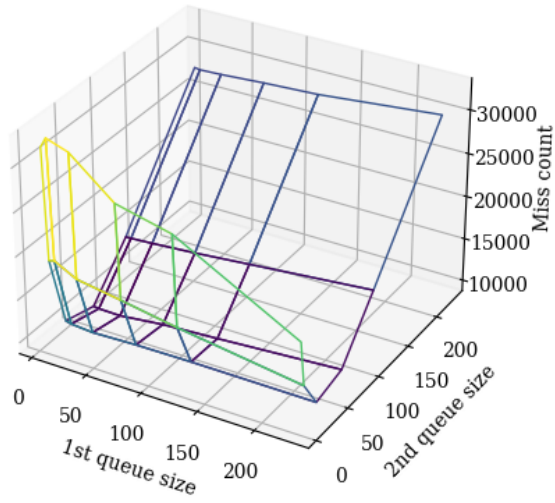
	10	55	100	250	400
10	21978	19331	18515	17609	17312
55	24727	20722	19488	17956	17480
100	23710	21013	19710	18021	17453
250	22919	20629	19451	17802	17576
400	22085	19949	18891	17681	18156

Figure 6.6: **2Q** parameters: Total capacity: 1000; Obtained with *Log 2*.



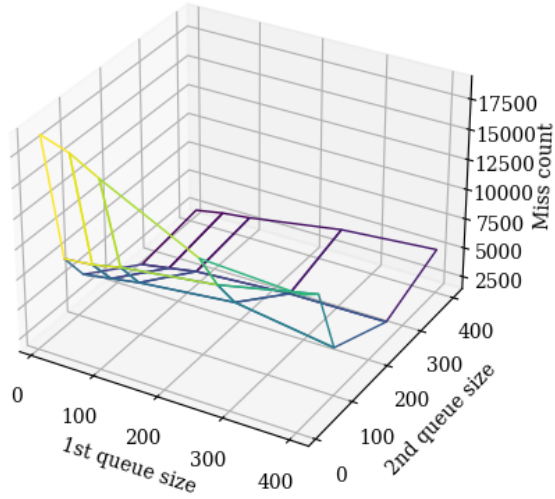
	5	10	30	100
5	71673	68217	68716	69014
10	72115	68864	68551	69068
30	73019	69238	68945	68802
100	70051	69650	68457	69033

Figure 6.7: **2Q-LFU** parameters: Total capacity: 300; Obtained with *Log 1*.



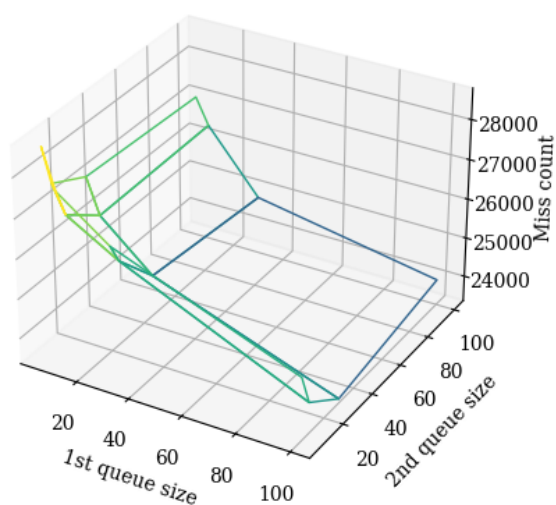
	5	10	30	70	120	230
5	31942	18839	10397	9361	14356	27808
10	33111	19012	10221	9273	14382	27702
30	32252	17654	10017	9223	14535	27806
70	28077	16810	10186	9781	14909	28140
120	26549	15673	10246	10144	15492	28466
230	18874	13583	10180	11045	16761	29845

Figure 6.8: **2Q-LFU parameters**: Total capacity: 600; Obtained with *Log 1*.



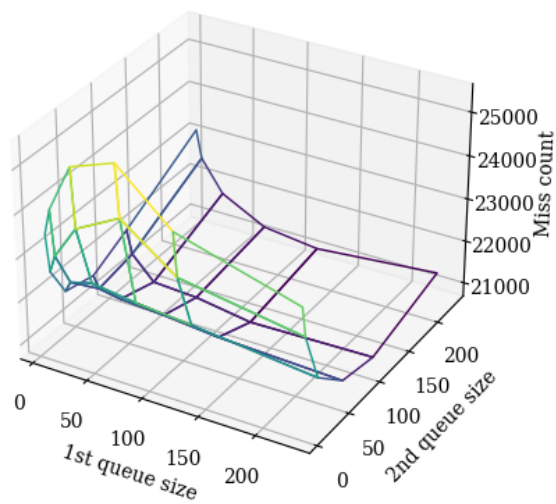
	10	55	100	250	400
10	19012	7711	5132	1808	2559
55	18197	7934	5506	2206	2988
100	16850	8435	5893	2631	3264
250	12830	9430	6701	3146	4545
400	12370	11140	5465	3220	5177

Figure 6.9: **2Q-LFU parameters**: Total capacity: 1000; Obtained with *Log 1*.



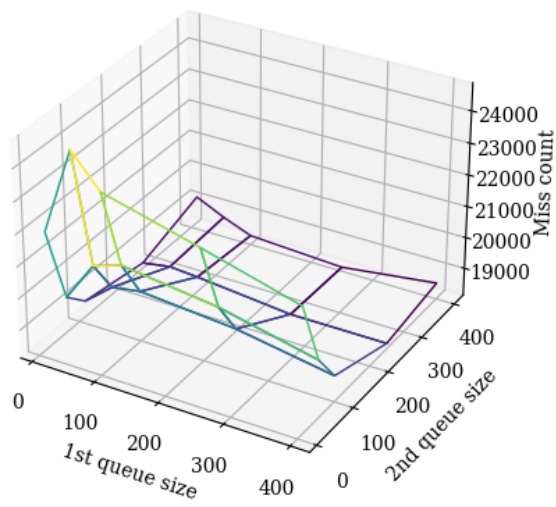
	5	10	30	100
5	28669	27617	27146	27011
10	27676	26948	26271	26360
30	26726	26196	25137	24826
100	24947	24134	23464	24004

Figure 6.10: **2Q-LFU parameters**: Total capacity: 300; Obtained with *Log 2*.



	5	10	30	70	120	230
5	23490	22564	21838	21670	22096	23136
10	24082	22961	22090	21395	21561	22488
30	25173	23730	22222	21304	21037	21767
70	25541	24265	22079	21238	20977	21239
120	24405	23321	21959	21118	20778	21079
230	23595	22861	21657	21007	20836	21328

Figure 6.11: **2Q-LFU parameters**: Total capacity: 600; Obtained with *Log 2*.



	10	55	100	250	400
10	21978	19456	18851	18522	19205
55	24727	20722	19575	18665	18780
100	23710	21013	19718	18597	18425
250	22919	20629	19462	18295	18257
400	22085	19949	18949	18297	18641

Figure 6.12: **2Q-LFU parameters**: Total capacity: 1000; Obtained with *Log 2*.

6.2 The parameter for the LIRS replacement policy

We also need to choose the *HIR capacity* for the *LIRS* policy. The article where it is established (Jiang and Zhang [2002]) suggests making it just 1% of the total capacity, albeit admitting that that could manifest negatively on hit rates. Let's look at the plots of miss counts observed with different *HIR capacities*. Again, those are obtained by counting the misses throughout the whole logs. Figures 6.13 and 6.14 are the plots for *Log 1*. Results for *Log 2* are visualized in Figures 6.15 and 6.16.

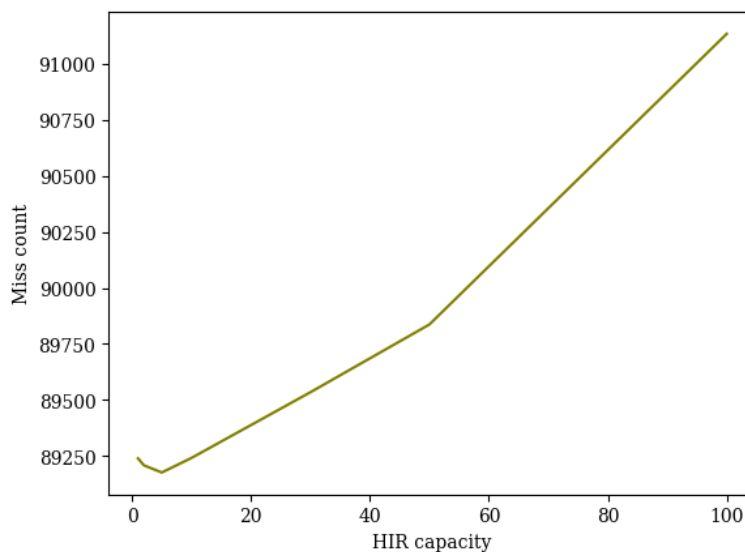


Figure 6.13: The absolute miss counts for different HIR capacities. The accesses come from *Log 1* with the total cache capacity set to 200.

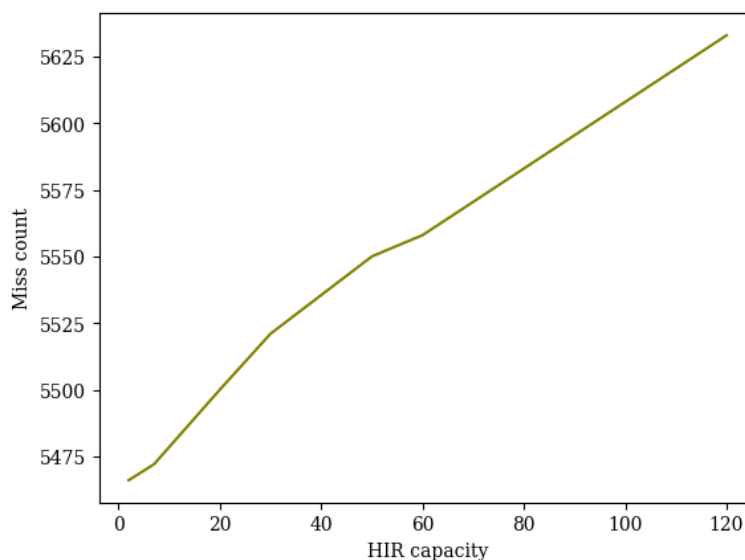


Figure 6.14: The absolute miss counts for different HIR capacities. The accesses come from *Log 1* with the total cache capacity set to 700.

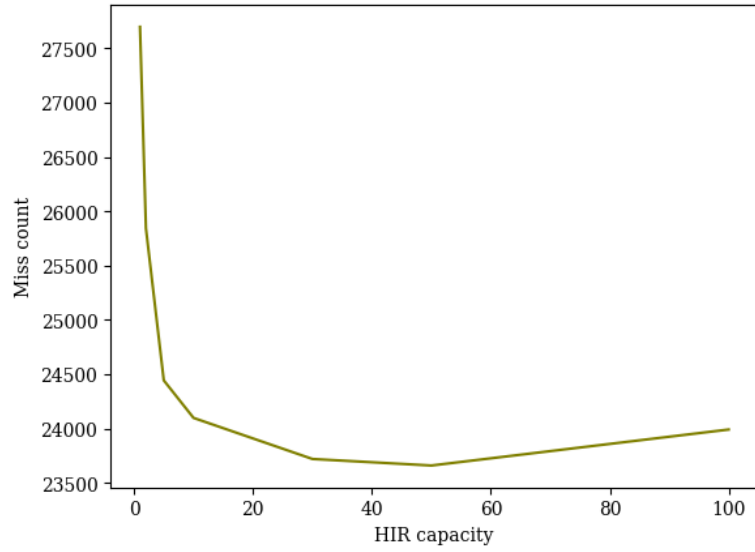


Figure 6.15: The absolute miss counts for different HIR capacities. The accesses come from *Log 2* with the total cache capacity set to 200.

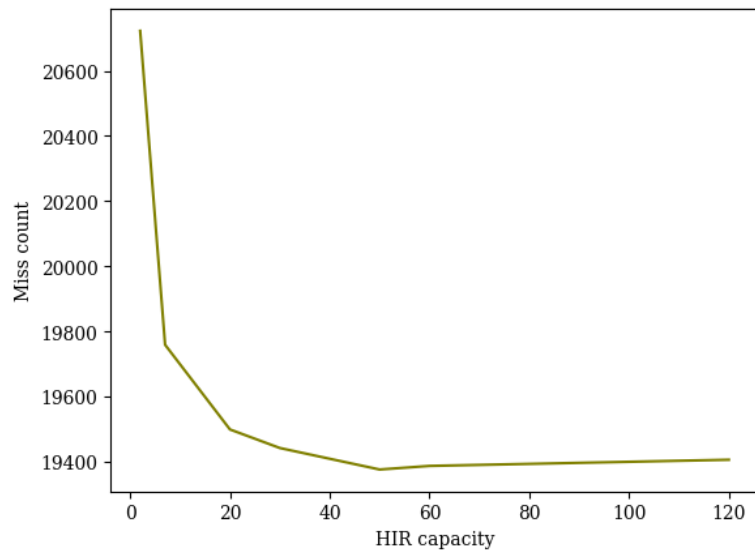


Figure 6.16: The absolute miss counts for different HIR capacities. The accesses come from *Log 2* with the total cache capacity set to 700.

We can see that the performance gets affected in quite different ways for the two access logs, once more showing the practicality of self-adapting policies, especially for purposes where the access patterns change over time. The differences aren't negligible either. As may be seen in Figure 6.15, setting the *HIR capacity* too low may increase the miss count to about 115% of the counts obtained for a range of better choices.

All measurements in Chapter 2 set the *HIR capacity* to exactly 30 as a compromise between the higher optimum for *Log 2* and the increasing miss counts in the case of *Log 1*, making it perform quite well compared to the other policies. But this section should help see how different that could be with a wrong choice of the parameter (including the one suggested in the LIRS article).

6.3 Open source database engines' approaches to cache synchronization

We will try to find out how existing databases approach caching. This does not *exactly* translate to what we perform in our measurements. The practical caches store pages rather than database entries. But of course, our access logs only hold UIDs of records, to which the page content would need to be further mapped, so our measurements pretend that it is in fact entries that are stored directly.

6.3.1 389 Directory Server

The engine that our two logs come from, **389-DS**, uses an *LRU* with a *single lock*, which is locked for the necessary timespan during the execution of transactions¹. This is quite like our *LOCK* simulation.

6.3.2 LMDB (part of the OpenLDAP project)

The **LMDB** doesn't perform application-level caching². It relies on the operating system's buffer cache, which of course could mean different things. For example a write-back *LRU* cache³ with page-wise locking, which is written through at regular intervals.

Besides reducing the implementation complexity, this of course helps use just the available amount of memory, and is a clever way of exploiting what the operating system offers. On the other hand, this provides no control over how much memory is made available for the DB cache.

6.3.3 Postgres

The article at ⁴ states that **PostgreSQL** uses a light-weight lock over the whole buffer table with a shared and an exclusive mode. The exclusive mode is used for modifications. However, the article is certainly deprecated, as it states **clock sweep** (a version of *CLOCK* that stores usage counts rather than reference bits as metadata, and whose *hand* decreases this number and evicts the first record found with a *usage count of zero*) as the utilized replacement policy. We know for a fact that the current version of **Postgres** uses *2Q* as the replacement policy⁵, before which it also had used *ARC*.

Nevertheless, from the available buffer manager source file⁶, it seems like the

¹There doesn't seem to be a blog post or article about this to cite, but we may look directly into the source code The cache is located in the following file. This is the most recent version at the time of writing, and the line where there's the locking method: <https://github.com/389ds/389-ds-base/blob/abb93243b427ee122a5a80ed095c0118d7b6616e/ldap/servers/slapd/back-ldbm/cache.c#L1628>

²See "No application-level caching" at <https://symas.com/lmdb/>

³<https://tldp.org/LDP/sag/html/buffer-cache.html>

⁴http://www.interdb.jp/pg/pgsql08.html#_8.3.

⁵<http://www.varlena.com/GeneralBits/96.php>

⁶The version current at the time of writing: <https://github.com/postgres/postgres/blob/f10f0ae420ee62400876ab34dca2c09c20dcd030/src/backend/storage/buffer/bufmgr.c>

locking scheme stayed the same. There is a buffer protected with a shared lock, which may be “upgraded” to a unique-access one.

Postgres may run in multiple processes and the mentioned buffer appears to be shared by all backends. Apparently, there is another type of cache that is process-local. We can find this source file ⁷ where shared memory is used to implement an invalidation scheme to synchronize between processes. This would suggest that there are more approaches used. One is a shared cache with a special two-mode lock. The other is an extended *PER THREAD* strategy with process-local caches.

⁷Current version at the time of writing: <https://github.com/postgres/postgres/blob/d68a00391214be2020e49be4b55f761d47a5c229/src/backend/utils/cache/inval.c>

6.4 Analysis of the *ASSOCIATIVE* strategy

Section 3.2 only gives an intuitive explanation of the benefits that the *ASSOCIATIVE* strategy offers. Our cache has multiple slots, which may allow multiple threads to gain simultaneous unique access to different records. In this section, we shall provide a more exact way of estimating the probability of this happening.

Let's assume that we really managed to divide the cached set into the slots evenly. How likely is it then that two transactions which access disjoint sets of records can be processed in parallel? I.e. that the records of the two transactions belong to disjoint slots?

Let's say we have n cache slots, and that the first transaction accesses m_1 records, the second one m_2 . The total number of ways those records may be divided into the slots is $n^{m_1+m_2}$ and our assumption says that the slot each record falls into is de facto uniformly random. We may imagine that the first transaction locks slots first, and leaves u slots unlocked for the second one. Then there are u^{m_2} possibilities of dividing the second transaction's records into the free slots.

However, we haven't yet evaluated how many options there are for the first transaction to leave exactly u slots unlocked. We will be forced to count separately for each possible number of slots locked by transaction 1, that is between 1 and $\min(m_1, n)$ slots. How many options are there for each such number " x " of slots (taken by transaction 1)? There are $\binom{n}{x}$ possibilities for the exact slots taken. The number of ways the records can fit into x ordered slots is given by the Stirling partition number⁸ $S(m_1, x)$. And we need to further multiply that by $x!$ to get the unordered version.

Overall, we need to consider all possible numbers of slots locked by the first transaction, count all possible ways the records could fall into that number of slots, and then multiply by the ways the second transaction's records could fall into the slots left. All this can be summed up into the probability of two transactions that access m_1 and m_2 disjoint records, respectively, accessing disjoint slots in an associative cache with n slots total being

$$\frac{\sum_{x=1}^{\min(m_1, n)} \binom{n}{x} S(m_1, x) x! (n-x)^{m_2}}{n^{m_1+m_2}}$$

with our assumption that each record belongs to a uniformly random slot.

To visualize how this confusing calculation translates into practical parallelization, we acquire Table 6.17. This shows how the probability of two transactions needing disjoint sets of slots increases with the total number of slots. With the important assumption that the keys themselves are disjoint. Since the median number of accesses by a transaction in *Log 2* was 3 (See section 5.1), we calculate these probabilities with m_1 and m_2 both set to 3.

As we would probably expect, it takes a considerable number of slots to approach certainty. Setting the number too high might on the other hand increase the number of cache misses, since we work with many caches with no global information about the access patterns. But most importantly, let's not forget about

⁸Also called Stirling number of the second kind. More information at https://www.whitman.edu/mathematics/cgt_online/book/section01.08.html.

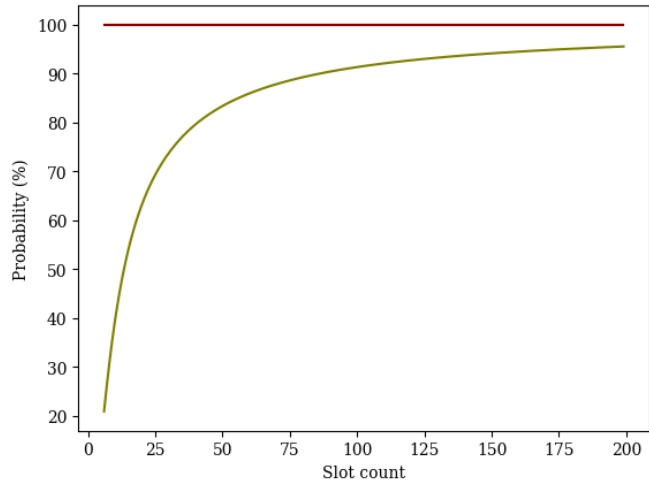


Figure 6.17: The increasing probability of two transactions accessing disjoint sets of 3 records also accessing different slots in the associative cache, in relation to the growing number of slots.

our assumption that the accesses by the two transactions are disjoint. If any key access is common to both transactions, they will also need this key's common slot. And in fact, in *Log 2*, **over 70% of transactions share at least one key access with the one transaction recorded right before them**. Even if we implemented fine-grained locking in some way, where we would only lock the exact records accessed, this would make it impossible for the transactions to proceed in parallel. And let's remember *Log 1*, where 40% of all accesses were to just one common key.

One must conclude that the overhead of this approach is rather large for the little benefit it brings, at least in the case of the access log we measure with. This may not be too significant in the case of our execution time measurements, but it's worth saying that all the concurrent measurements in Chapter 5 set the slot count to thrice the number of threads.

6.5 Transactional hash map implementation

Here, we will explain how a transactional hash map can be implemented, and what implementation our transactional cache uses.

Let's first recapitulate what we expect from the hash map. Any number of threads may acquire read-only snapshots to the hash map, each snapshot provides a valid state of the hash map, and the state doesn't change in any way for the whole existence of the snapshot. A thread can only retrieve values from the snapshot, but cannot modify it. We will start calling the read-only snapshots "*read transactions*" of our hash maps.

We will also make the map provide "*write transactions*" to up to one thread at a time. Write transactions also offer an isolated state of the hashmap, and the possibility of modifying this state, *locally*. Only once a write transaction is *committed*, do its modifications made to the hash map propagate to the map's globally accessible state, and new read (or write for that matter) transactions will then provide the committed state of the hash map. We will not be using this in the cache, but for the sake of completeness, a write transaction may also be rolled back, meaning that it is dropped without its modifications changing the state of the hash map globally.

In each subsection, we will state where in the attached directory these data structures are implemented.

6.5.1 COW-friendly B+ tree

Implemented in `src/bpt.transactional.rs`

Our implementation will be inspired by filesystems, specifically the copy-on-write (COW) B+ tree of the BTRFS, described in section 3.1 of Rodeh et al. [2013], we will not explain it thoroughly here. The basis is the standard B+ tree. To create both read and write transactions (of the B+ tree, keeping the terminology from the previous paragraph), the tree provides the transactions with a reference to the root node. In the case of the read transaction, this is the only thing needed, the root lets us search for any elements present in the tree at the time of the transaction's creation.

When a write transaction modifies the tree in any way, it must not directly change any of the tree's nodes, as they may be getting accessed by read transactions. Instead, it creates the necessary paths anew, referencing original, unchanged, nodes where possible. This means that the global tree still holds the original root, and the original state of the tree, that new read transactions will access, up until the write transaction is committed by exchanging the original root with the one the write transaction created.

Implementation-wise, we protect the root node with a `Mutex`, so that write transaction commits are safe. This way, threads will need to lock it just for the instant of obtaining the tree's transactions. To free the nodes from memory only once that no transaction may access them (i.e. they are neither a part of the current global tree state, nor of any active transaction's tree), we use atomic reference counting⁹.

⁹See <https://doc.rust-lang.org/std/sync/struct.Arc.html>

6.5.2 Hash map made with the B+ tree

Implemented in `src/bptree_hash_map.rs`

Turning our B+ tree into a transactional hash map is rather straight forward. We can simply hash the keys and store our records in the B+ tree, ordered by the hashes. We only need to add a mechanism for storing multiple records sharing a hash, in case of a hash collision.

6.5.3 Trie-based transactional hash map

Implemented in `src/trie_hashmap.rs`

We may notice that the B+ tree is too general for us, and try to implement a faster data structure for this purpose. The principle discussed in the previous paragraphs may be used for any tree-based structure. To better imitate the approach of the more standard, array-based, hash map implementations, we may imagine splitting the array into tree leaves. More specifically, we will build an N-ary trie (N-ary for the purpose of still utilizing the CPU caches) using the hash as the “word” the trie is ordered by. This way, branch nodes don’t need to hold any keys, and only store references to their child nodes. Leaf nodes contain just the key-value pairs stored for the specific hash. Our implementation uses a constant depth of the leaves. The context of using this for implementing a cache allows for further optimization by setting the depth just right based on our cache’s capacity (but more general approaches can, of course, be invented as well, which would use a variable depth of leaves), and naturally, if we were to really make an effort to optimize the performance, we would have to fine-tune the number of children of each branch, for our hardware (CPU caches, namely).

It is worth noting that while even our unoptimized “prototype” performs better than the B+ tree, as we will see, it is also considerably easier to implement.

6.5.4 Performance evaluation

To make sure we have really improved the performance with our trie-based map, we make a little experiment and make our two implementations perform randomly generated operations. We may only compare the sequential performance of write transactions, as both implementations use the same concurrency primitives, lock-protected acquisitions of root references, and atomic reference counting.

We generate three workloads for the write transactions, first with 2 000 operations on 100 unique keys, second with 35 000 on 3 000 keys, and a third large one with 20 million operations on 100 000 unique keys. Besides running with our two hash maps, we will run the workload with Rust’s standard library (STD) hash map for comparison, and also offer a special “NULL” benchmark, that won’t perform any hash map operations, but will only iterate through the workload in the same way the other measurements do, to see the overhead of the measurement itself. Some of the generated operations are write transaction commits (after which, a new write transaction handle is created), the STD hash map omits them of course.

We measure this with the same hardware and tools described in section 5.1. The results are available in Table 6.1. The confidence interval bounds fit in under

1 % away from these values for all measurements.

The trie-based map really performed better, as expected, and we use it to implement the transactional cache. Besides the already stated optimization options, there is even more room for improvement. The Master’s thesis Hart [2005] gives a good overview of alternatives to atomic reference counting for memory reclamation, that could further boost the data structure.

	2 K	35 K	20 M
NULL	0.0	0.3	171.1
STD	0.1	0.9	697.1
Trie	0.2	7.0	7 199.1
B+ Tree	0.2	8.7	12 597.5

Table 6.1: Execution times in milliseconds for the comparison of hash maps. The horizontal parameters are the numbers of operations in each measurement.

Bibliography

- L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2), 1966. ISSN 0018-8670. doi: 10.1147/sj.52.0078. URL <https://doi.org/10.1147/sj.52.0078>.
- Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.
- Lukas Folwarczny and J. Sgall. General caching is hard: Even with small pages. *Algorithmica*, 79:319–339, 2016.
- Thomas Edward Hart. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures, 2005.
- Song Jiang and Xiaodong Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, June 2002. ISSN 0163-5999. doi: 10.1145/511399.511340. URL <https://doi.org/10.1145/511399.511340>.
- Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association. URL <https://www.usenix.org/conference/2005-usenix-annual-technical-conference/clock-pro-effective-improvement-clock-replacement>.
- T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, 1994.
- Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association. URL <https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>.
- Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Trans. Storage*, 9(3), August 2013. ISSN 1553-3077. doi: 10.1145/2501620.2501623. URL <https://doi.org/10.1145/2501620.2501623>.

List of Abbreviations

- ACID: Atomicity, Consistency, Isolation, Durability (the standardized set of properties database transactions must satisfy)
- ARC: Adaptive Replacement Cache (also atomic reference counting in the context of section 6.5)
- BTRFS: B-Tree Filesystem
- COW: Copy-on-write
- CPU: Central processing unit
- DB: Database
- GiB: Gibibytes
- HIR: High inter-reference recency
- I/O: Input/Output
- IRR: Inter-reference recency
- LDAP: Lightweight Directory Access Protocol
- LFU: Least Frequently Used
- LIR: Low inter-reference recency
- LIRS: Low inter-reference recency set
- LRU: Least Recently Used
- MVCC: Multiversion concurrency control
- STD: Standard (in the sense of The Rust Standard Library)
- TXN: Transaction

A. Attachments

A.1 Source code

The attached zipped directory contains all the code used for the evaluations in this thesis. More information on its contents is in the directory's `README`.