

The Australian National University
2600 ACT | Canberra | Australia



Australian
National
University

School of Computing

College of Engineering, Computing
and Cybernetics (CECC)

Optimizing Unsorted Log-Free Key-Value Stores for Small Items and Range Scans

— 12 pt research project (S1/S2 2024)

A report submitted for the course
COMP3770, Computing Research Project (R&D)

By:
Itay Yarom

Supervisor:
Dr. Shoaib Akram

November 2026

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the [University Academic Misconduct Rules](#);
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

November, Itay Yarom

Acknowledgements

I would like to acknowledge the assistance of my supervisor Shoaib Akram who has assisted me greatly across the entire research process. I would also like to acknowledge Jasper Quirk for supporting me throughout this project. Jasper has been a consistent help, and was always willing to lend a hand with his knowledge of very niche and interesting aspects of hardware.

Abstract

Key-value stores are a ubiquitous form of datastore used in many applications, including large commercial enterprises such as Meta, Google, and Amazon. The ability to serve a large number of requests in a timely manner is crucial to these services. The rise of modern Solid-State Drives (SSDs) have led to a change in paradigm of key-value store design. KVell by [Leapers et al. \(2019\)](#) is a model designed to take full advantage of SSD bandwidth. It uses a page-granularity cache to handle requests from the SSD, and persists items to their final locations on storage without sorting. This incurs significant wasted space for small items. The search index is also quite large compared to the size of small items. We propose incorporating a coalescing mechanism whereby hot items are collected onto the same page in a separate space. Furthermore, we institute dynamic movement of items on storage to maximise how many hot items can fit in memory. We observe that this significantly improves KVell’s capabilities for handling small items, especially in memory-restricted environments. In read-only workloads, we achieve up to $2.4\times$ better throughput than KVell with a small page cache, and up to $1.6\times$ compared to KVell with a large page cache. We also attain up to a 70% decrease in average latency and 58%/43% reduction in 95th and 99th percentile latency respectively. For write intensive workloads we get approximately $2\times$ improved average throughput, 38% decrease in average latency, and 43%/36% in write-intensive workloads. Our design suffers under continuous shifting of hotspots in the workload.

Table of Contents

1	Introduction	1
2	Background	5
2.1	NVMe SSDs	5
2.2	KVell	6
2.2.1	Motivations	6
2.2.2	Design	6
2.2.3	Benchmarking	7
2.3	Motivation	7
3	Related Work	9
3.1	Hot Item Handling	9
3.2	Small Items and Scans	10
4	Design and Implementation	11
4.1	Design	12
4.1.1	Coalescing Space	12
4.1.2	Evictions, Write Requests, and Item Movement	13
4.1.3	Profiler Design	13
4.2	Implementation	14
4.2.1	Modifications to B-tree Index	14
4.2.2	Profiling and Item Hotness	14
4.2.3	Coalescing Space	15
4.2.4	Item Movement Handling	17
4.2.5	Modified Request Paths	17
4.3	Alternative Choices	18
4.3.1	No Item Movement	18
4.3.2	Global Profiling and Coalescing	18
5	Evaluation	21
5.1	Benchmark Set	21
5.2	Evaluated Software	22

Table of Contents

5.3	Hardware Setup	22
5.4	Empirical Results	22
5.4.1	Throughput Results	22
5.4.2	Latency Results	26
5.4.3	Evictions	28
5.4.4	Requests Served by SSD	29
6	Concluding Remarks	31
6.1	Conclusion	31
6.1.1	Limitations	31
6.2	Future Work	32
	Bibliography	33

Introduction

Key-value stores are a type of datastore used across a large number of contexts in modern systems. They are an commonplace and used at many scales. In particular, there are several (implementations) designed and used by large enterprises such as Google’s LevelDB [Google \(2021\)](#), Meta’s RocksDB [Meta \(2022\)](#), or Amazon’s DynamoDB [DeCandia et al. \(2007\)](#). The former two are also open-source, meaning there are contributors not necessarily employed by the companies.

Key-value stores function on a very simple dictionary-like principle, we associate a key (the “word”) with some value (the “definitions”). Importantly, each key only has one value, so we can consider each item in the database as a key-value pair.

Key-value stores are desirable due to the simplicity of their interface and potential speed of request handling. They have an almost universal interface, with just about every key-value store supporting (at least) five main operations:

- Point Lookups: Requesting the value associated with one key
- Insertions: Add a new key-value pair to the database
- Updates: Update the value associated with a given key
- Deletions: Deletion of a key-value pair
- Scans: Requesting the value associated with a range of keys

While the interface is simple implementation details can be quite complex, and there is no one key-value store that is better than every other in all contexts. Implementations are designed to have advantages for certain workloads and hardware. For example, RocksDB is built on a concept known as a Log-Structured Merge Tree (LSM tree), which was initially tailored for write-heavy workloads with fast memory and slower storage devices like hard disks.

1 Introduction

One common feature of many commercially used and older key-value stores such as those based on LSM trees is that data is written and stored *sequentially*. This is because historically it is significantly faster to interact with storage by using sequential requests. LSM tree-based designs spend a lot of time making sure that this sequential ordering is maintained by writing items to storage in *sorted runs*. This process can lead to multiple copies of the same item being present on storage, so a background process known as *compaction* is used to get rid of dead copies of items and minimise space overhead. Regardless of how compaction is implemented, it is an expensive background process in terms of storage bandwidth utilisation and time spent not working on requests.

Older storage designs such as hard disks have a very large difference in sequential and random access speed. This is because it takes time to spin the disk to reach the required data. Modern NVMe SSDs still have a gap between random and sequential bandwidth, but it is significantly smaller. LSM trees were designed to make use of memory and force sequential accesses to account for the very large sequential-random bandwidth gap. However, [Lepers et al. \(2019\)](#) demonstrate with the *KVell* key-value store that with these advancements in hardware, the overhead spent enforcing sequential accesses to storage is no longer advantageous. They develop a model which minimises metadata overhead, movement of key-value pairs on storage, and thread synchronisation. To achieve this, [Lepers et al. \(2019\)](#) partition the store across different workers so that they require minimal synchronisation.

KVell uses a rudimentary page-granularity cache to handle item requests. This means that any time an item is requested, the entire page containing that item is brought into memory.

There are three universally considered metrics for analysing how effective a datastore is:

- Throughput — how many requests do we process in a given period of time
- Average Latency — how long the average request takes to process
- Tail Latency — the time in which most requests are processed

These three are the user-facing metrics, how users will experience interaction with the datastore. There are of course other metrics, such as space, read, and write amplification. These represent how much overhead there is in the space taken and the number of read/write requests made to the SSD for each item/request. For example, the dead items in LSM trees represent significant space overhead, and compaction induces a significant number of reads and writes on top of what the single request would imply.

Workloads in modern databases tend to be heavily skewed. As observed by [Chen et al. \(2020\)](#), it is not unreasonable for 50–90% of requests to go to just 1% of the items in the database. KVell’s page cache model means that a lot of memory can go wasted, as other items on the same page as these “hot” items are unlikely to be requested in the near future. Therefore, these items occupy space in memory that could be used to serve requests. This is exacerbated in the case of small items — the more items on each page,

the more space is wasted.

Furthermore, because KVell does not sort on storage, it does not significantly improve scan performance. The reasons for this are multifaceted:

- Sequential I/O is still faster than random I/O on modern storage devices, the gap is just smaller than before
- When sorted, scans rely less on the SSD as more items can be brought into memory at once, as they lie on the same pages

The aim of our research is to improve KVell’s throughput and item request latencies for skewed workloads, targeting small items and scans. We also significantly reduce the amount of memory required for KVell to run effectively by conserving space in the memory index and making significantly better use of caching in memory. This brings KVell closer to being applicable in restricted memory scenarios.

We propose adding a coalescing mechanism, whereby frequently-accessed items are brought onto the same pages in memory and on storage.

We make the following contributions:

- We halve the size of KVell’s in-memory index
- We improve the latency and throughput of requests in KVell, particularly for small items
- We significantly reduce the memory required for KVell to run efficiently under skewed workloads

Background

We discuss here important background information and motivations about Kvell that are important to understand how our design builds on the existing work.

2.1 NVMe SSDs

Solid-State drives (SSDs) have become increasingly popular in recent years, leading to a change in paradigm for storage hardware. Older hard disk drives (HDDs) have a high seek time, or the time it takes to spin the disk to arrive at the desired data. SSDs have much better internal parallelism than hard disk drives.

Previously, hard disk drives (HDDs) were the norm. These used a SATA bus and the Advanced Host Controller Interface (AHCI) to manage requests to storage. However, these are suboptimal on SSDs, which have much higher concurrent random access capabilities. A new bus, Peripheral Component Interconnect express (PCIe) is used to send data under the Non-Volatile Memory express (NVMe) host controller interface specification designed by Samsung. NVMe can take advantage of the SSD's design by managing $2^{16} - 1$ command queues of depth $2^{16} - 1$, compared to a single queue with depth 32 that AHCI assumes for use on HDDs (described in [nvm \(2024\)](#) and [ahc \(2024\)](#)).

This interface specification allows us to take advantage of the much larger number of request queues on Modern SSDs that were not exposed to AHCI (as it was designed for the serial HDDs).

Essentially this means that SSDs are not only faster, but due to the better internal parallelism, the gap between random and sequential access speed is significantly lower than on HDDs.

2.2 KVell

Our work builds directly off of the key-value store KVell, developed by [Lepers et al. \(2019\)](#). We describe here the motivations and design of KVell.

2.2.1 Motivations

[Lepers et al. \(2019\)](#) demonstrate that the lower random-sequential performance gap of modern NVMe SSDs nullifies the benefits of optimisations in many existing key-value stores. One example of this is Log-Structured Merge-Trees (LSM Trees), which dedicate a significant portion of CPU and I/O bandwidth to sorting and compaction: processes which are designed to enforce sequential I/O. [Lepers et al. \(2019\)](#) further demonstrate that the sheer increase in performance of NVMe SSDs over disk drives has made CPU the primary performance bottleneck in commonly used key-value stores. Thus, the value of trading increased CPU and SSD overhead to enforce sequential I/O is significantly diminished.

2.2.2 Design

Storage Structure

KVell eschews the principles of maintaining sorting of key-value pairs in storage, instead persisting items to their final location in files called “slabs.” These slabs are collections of items that are all within the same size range, and an item can be found by its index in the slab.

Worker Threads

The base design consists of multiple worker threads, each handling its own set of slabs — with one slab for each size range. Each worker has a unique id, from 0 to the number of workers, and items are assigned to each worker based on their 64-bit prefix hash modulo the number of workers, to distribute as evenly as possible. Worker threads only ever need to synchronise when handling scans. No item can be influenced by more than one worker, removing synchronisation overhead that normally comes with multithreading.

In-Memory Index

A per-worker in-memory B-tree index is used for handling lookups. When an item is requested, its location is found in the B-tree, which include the slab and index of the item on that slab. Lookups are done using the 64-bit prefix of the item.

Request Handling

A user-level page-granularity cache with an LRU policy is implemented by KVell. Whenever a request is made for an item not in the page cache, the entire page that the item is on is brought into the page cache from storage using the POSIX AIO interface.

Requests are handled internally by callbacks, which are updated at each part of the asynchronous request process to maintain the information required to fulfill the request.

When a write request is made, the item is updated in the cache and the page is marked as dirty. A write request for that full page is then sent to the SSD. Once the request has been completed, the page is marked as clean. If the page was already dirty, the update is made in memory and the callback is appended to a set of linked callbacks.

When a read request is made to an item in the cache, it is done synchronously. If the item is not already in the cache, a request is made to bring the page it is on from storage, and the callback is appended to a set of linked callbacks.

The set of linked callbacks is checked periodically and on processing of an I/O request to see if the requests in any of the callbacks can be executed. Those that can are executed, otherwise they are returned to the linked callbacks to be checked again later.

Requests to the SSD are batched in order to minimise the number of system calls and achieve the most effective bandwidth utilisation. This means that requests to the SSD are added to a queue which is flushed at the same time. The sizing of this queue manages a tradeoff between latency and throughput, with larger queues allowing for more SSD bandwidth utilisation but more delay before requests are sent to be processed.

2.2.3 Benchmarking

Benchmarking is conducted using a set of *Injector* threads which continuously generate and send requests to the database. In the original paper, both synthetic and production workloads were used for benchmarking purposes. We will only discuss and use the synthetic workloads.

The synthetic workloads used for this are created from the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al. \(2010\)](#) framework. This is a very commonly used benchmarking framework in the field. In particular we use YCSB workloads with a Zipfian distribution, which reflects skewed workloads where certain items are accessed very frequently, while most items are accessed quite infrequently.

Many parameters of the workloads are modifiable, of particular importance to our research are the following:

- Distribution of reads to updates
- Weighting of the Zipfian distribution
- Scan size

2.3 Motivation

While Kvell displays significant improvement compared to pre-existing key-value stores, there are a few key drawbacks;

2 Background

- Kvell is significantly less effective for small items
- Kvell shows much less improvement in scans
- Kvell requires a large amount of memory to run effectively

The reasoning here is relatively simple: suppose we have an item size of 128 bytes, with a page size of 4KB. Then we have 32 items per page. It is highly unlikely that the same page contains multiple frequently accessed items, so we can assume that no other requests are going to items on the same page. If this is the case, then only 1/32 of the page cache actually contains accessed data. This is especially important for scans, as with Kvell’s design, each item from the scan would be brought in its own page, but sorted models such as RocksDB bring in many items from the same scan with each page. In this scenario, Kvell incurs up to $32\times$ the number of SSD accesses for the same scan.

Small items also leads to less space-efficient memory indexing. Each entry in the B-tree index has a size of approximately 19B per item I.e. for 128B items, we would need memory equivalent to $19/128 \approx 14.8\%$ the size of the database. If this large proportion of memory cannot be provided, then Kvell’s performance dips dramatically, as noted in the original paper.

Moreover, Kvell does not make very effective use of hardware caches and the TLB. Even if there is enough memory to serve all frequently accessed items, they each lie on their own 4kb page in memory.

Kvell is also currently under-utilising CPU: without processes such as sorting and compaction, the main hardware bottleneck for Kvell is the storage medium. Their paper observes that CPU utilisation is below 50%, indicating there is room for additional CPU overhead.

Kvell is not effective in low-memory situations, as Kvell’s B-tree data structure incurs a high metadata overhead, particularly for small items. When the B-tree does not fit in memory, Kvell’s performance drops significantly, as outlined in the original paper and by [Conway et al. \(2020\)](#). It also requires a relatively large page cache to function at reasonable rates.

Related Work

Our research aims to improve the effectiveness of KVell for small items and scans. Our proposed method for handling this is by improving the locality of KVell under highly skewed workloads. Therefore, we break down related work into two categories:

- Works which attempt to improve key-value store performance by handling hot items
- Works which target improvements specifically for small items and/or scans

3.1 Hot Item Handling

Targeting of hot items in skewed workloads has been attempted before in a variety of designs. Many such designs also place emphasis on effective cache utilisation due to the improved locality.

HotRing by [Chen et al. \(2020\)](#) minimises average collision-handling duration in hash indexes by dynamically reordering buckets such that hot items are checked first. They use regular sampling to determine which items are hot. Unlike KVell, threads may interact with the same data structures (the ring) – a lockless implementation is instead used to minimise synchronisation overheads.

In contrast, [Wang et al. \(2020\)](#) explore methods of taking advantage of hardware cache through a variety of optimisations including hot item tracking, separation of key and value data, and cache-friendly hash indexing. In particular, this separation of key and value data targets cases where key and value size are disproportionate and wasting space that could be used for lookups in the cache.

FASTER by [Chandramouli et al. \(2018\)](#) is a key-value store centered around a *hybrid log* structure that places emphasis on fast handling of updates. It maintains hot items

3 Related Work

in memory, using a log for consistency. Similarly to these other models, it uses a cache-friendly concurrent hash index where buckets are aligned to cache lines.

Another design option for managing hot items is to physically migrate them into a space designated for hot items. One such example is HMCACHE, a design by [Jin et al. \(2020\)](#) which targets hybrid memory. Non-volatile memory (NVM) is an emergent hardware format, offering better performance than SSDs and more space than DRAM. [Jin et al. \(2020\)](#) target such a hybrid DRAM-NVM setup by maintaining separate DRAM and NVM zones, and migrating items between them once they cross a given threshold. They use a per-item access counter that is regularly decayed to record item hotness. This item counter is maintained as part of the item metadata in both memory and NVM. They also do item migration / reassignment from non-volatile memory (NVM) to DRAM to decide how many items from different slabs on NVM should be reassigned to DRAM for optimal benefits.

AC-key by [Wu et al. \(2020\)](#) is a design which dynamically resizes three differently organised caches to serve the current workload most effectively. It uses a key-value cache to cache hot KV pairs, a key-pointer cache for caching items with very large values (the pointer being a pointer to the location of the value), and a block cache which is efficient for scans, but not for serving point lookups. Essentially, dynamically resizing these three different caches can help serve different workloads more effectively, e.g. a larger KV cache is more beneficial for getting hot and small items.

3.2 Small Items and Scans

There are also many approaches specifically targeting small items and range scans with a variety of methods.

SplinterDB by [Conway et al. \(2020\)](#) is designed to minimise metadata size and reduce write amplification for small key-value pairs. It does so by minimising how often the SSD is accessed for lookups, and by implementing a concurrent clock-based page cache. Threads can work on requests for any items, and use reader-writer locks to avoid cache-line thrashing.

[Didona and Zwaenepoel \(2019\)](#) implement Minos, a datastore designed to decrease tail latencies in workloads with a mix of many small and few large items using a process called size-aware sharding. They assign small items and large items to be handled by separate cores so that small items aren't queued behind larger items.

Design and Implementation

KVell wastes a significant amount of memory when used with small items and workloads that are heavily skewed. Memory locality is a weakness of KVell's with smaller items, as the page cache is managed at a page granularity, incurring significant proportions of space in the page cache which are under-utilised.

We propose implementing a *Coalescing Space*, an item-granularity cache-like structure into which hot items are gathered to improve request locality in KVell. We further observe that we can maintain the benefits of this coalescing by persisting the new item locations to the SSD — thereby freeing up room in the Coalescing Space. See Figures 4.1 and 4.2 for a visual reference of the benefit of coalescing and item movement on SSD.

To make better use of memory, we cut down the size of the in-memory B-tree index by 8 bytes per item. In the modified schema, we also reduce the number of lookups guaranteed to miss in the Coalescing Space by co-opting one bit to identify where the

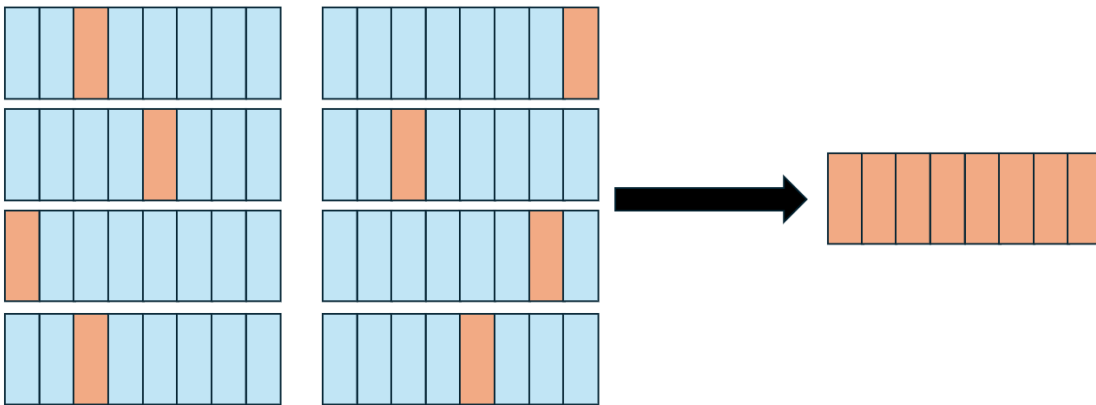


Figure 4.1: Visual reference for benefits of coalescing

4 Design and Implementation

item is currently in the space.

Memory-efficient item-access profiling is crucial due to the sub-linear relationship between memory and storage size. We thus use a *Count-Min Sketch* for approximating request counts for individual items. This scheme trades increased profiling time and minor inaccuracies for a reduced space utilisation.

4.1 Design

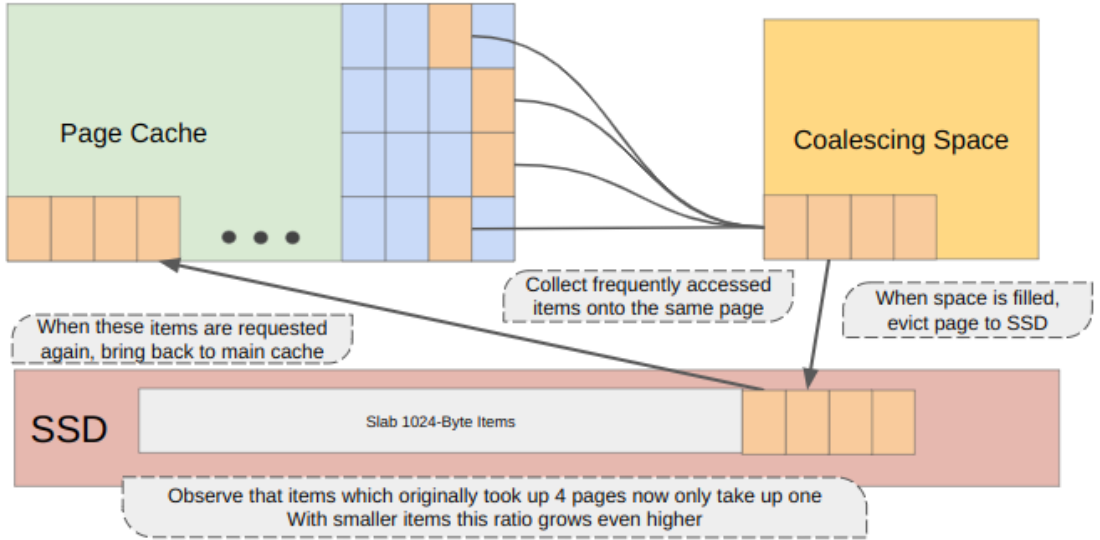


Figure 4.2: Coalescing Space with Item Movement Concept (1KB items depicted)

4.1.1 Coalescing Space

We use a Coalescing Space local to and controlled by each worker. This has two primary consequences:

- Workers spend time handling lookup, insertion, and eviction of items from the coalescing space
- Only items managed by the same worker are ever on the same page

The former is a drawback of this design, increasing the critical path of requests. We discuss in the implementation section the mechanisms by which we minimise this overhead. The latter implies that scans, will not benefit as much from coalescing as they would if all items were brought contiguously onto the same pages. In particular, small scans will not benefit as much from our design, especially under higher worker counts.

4.1.2 Evictions, Write Requests, and Item Movement

We diverge from KVell’s principle of persisting items to their final location on disk. Under heavily skewed workloads, we can observe that there is a trade-off between the overheads of managing item movement and the improvements to locality. Thus, we allow for coalesced items to be moved on the SSD, on writes and evictions from the Coalescing Space. In our current model, we also assume that once evicted, items never need to come back to the Coalescing Space, so we mark which pages are evicted and ensure items from those pages don’t return.

In particular, we can consider one page in the Coalescing Space as being worth N pages in the page cache, where N is the number of items on each page. The main motivation behind the item movement policy is this tradeoff, we can maintain this N to one tradeoff by returning the page to the page cache, freeing up N more slots in the coalescing page to coalesce a further N items. I.e. we take out the space of one item in the page cache to gain N spaces for items in the coalescing space.

Observe that write requests cannot be handled as trivially as reads, as they induce a write of the entire page to the SSD. However, items on the same page in the Coalescing Space are unlikely to be the same as on its standard page. Simply enacting the write request from the Coalescing Space would result in the overwriting of data on SSD.

The naive solution to this is to force writes to follow the original path in KVell. However this almost always forces an additional read from the SSD as it is unlikely the page the item came from is still in the page cache.

We instead maintain some of the benefits of coalescing for write requests by writing the entire page to the SSD contiguously at the end of the slab and updating the locations of all items on that page. Any subsequent writes to the same page are done to the same location on the SSD. Evictions can similarly be done to the same location the page was initially written to SSD if a write request had previously occurred.

4.1.3 Profiler Design

We perform profiling using a per-worker count-min sketch. We now briefly summarise the functionality of this algorithm, as it is an important consideration in the tie overhead resulting from our approach.

Count-min sketch is a profiling method for counting the occurrence of values in a given input stream. It is designed using a table where each row has an associated hash function. When an element in the stream is encountered, it is processed by each hash function, determining the index in each row to be modified. The value in each of these rows indexed by the hash function is incremented by one. The lookup for the occurrence count is done by finding the minimum count across all locations that the value hashes to. This provides some redundancy for when other values are hashed to the same index in some of the rows.

4 Design and Implementation

Observe that there is a time-space-accuracy trade-off here. The more rows and columns, the higher the accuracy, but the more space this table takes up in memory. Furthermore, the more rows we have, the more memory accesses must be done to update the count for each request. As we have implemented these per-worker, this adds these memory accesses to the critical path of the item requests.

While usually count-min sketch is used with a minheap or similar structure to keep track of the k most commonly accessed values (top- k flows), our model does not require this, as we are not interested in maintaining precisely the top- k most commonly accessed items in the coalescing space. We simply want hot items to be shifted to the coalescing space at some point

In our model, we use item prefixes as the values passed to the sketch, as these are assumed to identify unique items, and we simply count the raw number of requests that each item is receiving.

4.2 Implementation

4.2.1 Modifications to B-tree Index

Original KVell used memory index items that stored two pointers, for 16 bytes of payload. We cut this down to 8 by using the following schema (from most significant to least significant bit):

- 1 bit which states whether the item is currently in the coalescing space
- 4 bits indicating which slab the item belongs to
- 59 bits for determining index of the item within the slab

This does mean we are restricted to $2^{59} \approx 570$ quadrillion items per slab. We believe it is a reasonable assumption that no slab will need more than this number of items.

This very significantly reduces KVell's size

4.2.2 Profiling and Item Hotness

We define item hotness using the following steps whenever an item is accessed:

1. Whenever an item is accessed, increment our **request_count** only if it has not already been coalesced
2. If that item has not been accessed before (according to the approximation of the sketch), then also increment the number of items that have been accessed so far **approx_num_items**
3. Assign the threshold value $\min\{1, \text{request_count}/\text{approx_num_items}\}$

4. If on updating an item’s access count, it is greater than the threshold and has not already been coalesced, we coalesce that item immediately

We further observe that we wish to drive coalescing of new items throughout the bench. To support this, we avoid re-coalescing items, and decrement **request_count** by 1 whenever an item is evicted from the Coalescing Space. This reduction is a conservative estimate of the contribution of the old items to the request count: it takes a minimum of 2 accesses to the same item to move it to the Coalescing Space, and one of those accesses contributes to **approx_num_items**, so this access should be maintained.

4.2.3 Coalescing Space

Hashmap Index

Fast lookups to find item locations in the Coalescing Space are crucial, as we wish to minimise request time. Furthermore, the Coalescing Space contains only a small proportion of the database at any given time, but many items pass through, resulting in a lot of deletions from the lookup index. We thus use a hashmap index to perform lookups. We make effective use of hardware caches by using a linear probing insertion and lookup strategy. Probing length is reduced by using a robin-hood insertion policy. To effectively handle deletions, we use a backwards shifting mechanism instead of tombstoning due to the high thrashing of the index. This is a well-known combination of policies in modern hashmap design, as used e.g. by [Kelly et al. \(2018\)](#).

The robin-hood policy developed by [Celis \(1986\)](#) is designed to minimise the variance of probe length. There is some added overhead for item insertion and deletion required to implement it, but in exchange it achieves expected $O(\log n)$ probe length (for n the number of items in the hashmap). The algorithm is depicted in [algorithm 1](#). Under this policy, expected probe length is $O(\log n)$. In fact, an optimisation is noted that under these scenarios, the maximum probe length scales by $O(\log n)$, so we can simply do a search within the maximum probe length and terminate if the item is not found (this is essentially the search mechanism described by [Kelly et al. \(2018\)](#)).

Backwards shifting (or backshifting) is a process where we keep track of the probe length of each item in the hashmap (i.e. how many additional locations in the hashmap did we have to check to get this final location for this item). When an item is deleted, we can check the next item in the hashmap to see if its probe length is greater than 0. In this case, it must pass the location of the deleted item during its probe. Thus, we can shift that item back into the location of the deleted item. This process can be repeated for all subsequent items until an item is found that is already in its ideal location or we reach an empty entry in the array. With this model, we no longer need to keep tombstones for deleted items, as they are there to help satisfy probes, but with backward shifting we know that all probes will be in contiguous segments of present items.

Note that this specific implementation of the backwards shifting policy isn’t guaranteed to be optimal in all hashmap implementations, as technically items located past an

Algorithm 1: Robin-Hood Insertion Algorithm

```

1  $i \leftarrow h(v)$  ; /*  $h$  is our hash function to the index,  $v$  is the value to
   insert */
2  $c \leftarrow 0$  ; /*  $c$  is our current probe length */
3  $tempv \leftarrow 0$ ;
4  $tempc \leftarrow 0$ ;
   ; /*  $A$  is our array, each entry has a state (1 or 0), the current
   value  $v$ , and the probe length for that value  $c$  */
5 while  $A[i].state \neq 0$  do
6   if  $c > A[i].c$  then
7      $i \leftarrow i + 1$ ;
8      $c \leftarrow c + 1$ ;
9   else
10     $tempv \leftarrow v$ ;
11     $tempc \leftarrow c$ ;
12     $v \leftarrow A[i].v$ ;
13    goto 1;
14  $A[i].c \leftarrow c$ ;
15  $A[i].v \leftarrow v$ ;
16  $A[i].state \leftarrow 1$ ;

```

item in its ideal location may have been possible to insert before. However, with the robin-hood policy, this situation is averted.

To summarise, this setup provides us with the following advantages:

- Linear probing makes very effective use of the hardware cache
- Robin-hood policy minimises average probe length and hence the clustering issues that usually come with linear probing
- Backwards shifting gives us efficiency under many deletions

Other Metadata

We minimise item insertion time by maintaining two linked lists for page metadata, one for pages that are full and one for pages that still have free space. While this incurs some additional overhead from metadata management, this means that we only need to scan the free list to see if an item can be inserted. Note that we can't just insert at the head of the free list as insertion must be to a page whose item size matches that of the item to be inserted.

We manage writes and evictions by keeping track of the location on SSD associated with each given page (tracking if a write has already been done to this page). This includes

checking how many of the items on the page have already been written, so we only have to update the in-memory B-tree index for items that haven't already been updated (as the process of updating location metadata is relatively expensive due to the size of the index).

4.2.4 Item Movement Handling

Eviction and writes in the Coalescing Space move the location of items. This incurs many lookups in the main B-tree index to modify the metadata about item locations, an expensive operation. In the current implementation, eviction from the Coalescing Space with item movement is done synchronously. These aspects of the implementation lead to significant overheads that result from eviction.

4.2.5 Modified Request Paths

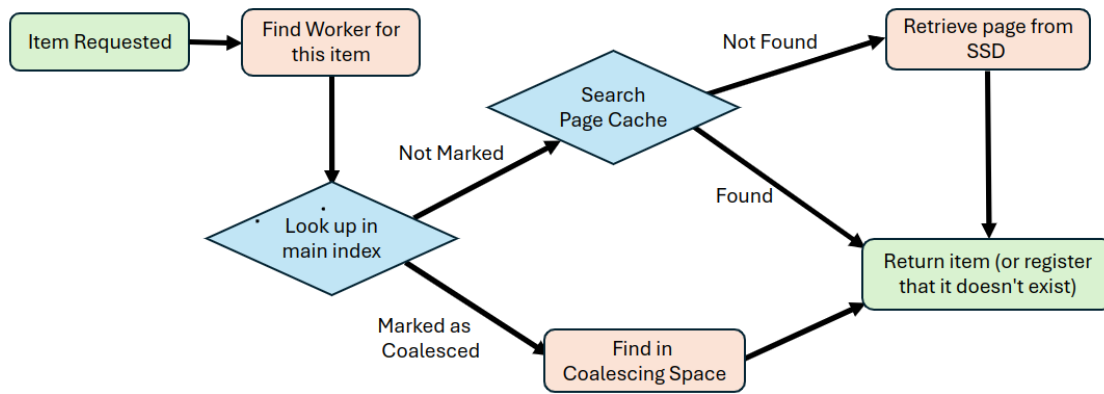


Figure 4.3: Modified read request path

We modify the request path to incorporate searching in the new coalescing space. The new path is depicted in Figure 4.3. Additionally, items are moved to the coalescing space after the request has been executed.

The write request path is essentially the same, except that once the item is found, it is updated and written back to the SSD.

Furthermore, the profiling is updated after the item has been read or updated in memory. In the case of updates, the update is done in memory before copying over to the Coalescing Space, and then written back to the SSD using the policy for the Coalescing Space described previously.

4.3 Alternative Choices

4.3.1 No Item Movement

Another question is whether incurring additional write requests to the SSD on eviction from the Coalescing Space is worth the benefit — the idea being that if we can coalesce all hot items into the coalescing space, then rearrangement on storage isn't as important. We explicitly test this case and demonstrate that our design achieves better performance across all metrics.

4.3.2 Global Profiling and Coalescing

There have been several choices made in our design in particular being whether to have thread-local or global coalescing and profiling. We can describe our model as a Local Coalescing and Profiling model, whereby everything is managed by the workers themselves. As we have noted previously this leads to overheads in the critical path of requests.

Under these models, we would implement a thread that does global profiling, recording access count for all items. As this is done on a separate thread, this can be done outside of the critical path of item requests. However, a few clear questions arise:

- Does the profiler manage the coalescing space or do the workers?
- If managed by the profiler, can items by different workers appear on the same pages?
- When can items that the profiler deems hot be moved to and evicted from the coalescing space?

Each choice for the first two questions represents a complex trade-off.

If the profiler manages the Coalescing Space, then there must be significant synchronisation overhead between the profiler and workers. This becomes even more complex due to the asynchronous nature of the requests, particularly for writes as evictions by the profiler can mess up active write requests.

Furthermore, having items from different workers on the same page may be quite beneficial for scans, but there needs to be some way to tell if items are in the Coalescing Space space, as otherwise separate lookups must be done by each thread regardless. Furthermore, this removes the simple eviction to SSD approach that we have implemented that significantly improves performance due to maintaining the benefits of coalescing. To do something similar in this case would require either a further thread or the profiler itself to handle requests, incurring even more overhead.

If the worker manages the Coalescing Space, then there must be some way for it to decide when to copy items over: note that this should only be done when the item is in memory, or else we incur additional SSD accesses. It must also keep track of which items should

4.3 Alternative Choices

be coalesced. This could be handled on the profiler by e.g. maintaining a minheap that the worker scans through and does as much coalescing as possible (which removes the guarantee that the items are in memory) or by receiving a signal every time the profiler determines an item to be hot and copying over (which requires synchronisation overhead).

Due to these complexities and additional overheads, we believe our local-local method to be an effective approach despite the additional memory access overhead on the critical path of item requests.

Evaluation

5.1 Benchmark Set

We use the Yahoo! Cloud Serving Benchmark (YCSB) framework [Cooper et al. \(2010\)](#) to benchmark our implementation, and a modified implementation to test the case where item hotness changes gradually over time. We specifically test with the benchmarks described in Table 5.1. Theta is the parameter controlling how skewed the requests are. At 0.9 approximately 0.5% of items contribute to 50% of the requests. The higher theta is, the more skewed the distribution. We evaluate with a lower theta value to observe precisely how effective eviction to SSD is in making effective use of memory.

We use a database of 500 million items with 128-byte item size and 4KB pages. For all benchmarks except YCSB E, we conduct 1.5 billion requests and six injectors. For YCSB E we use 30 million requests and eight injectors. We use 20 workers for all benchmarks.

Gradual Shift Workload

This benchmark emulates a gradual decrease in heat for already hot items, and new items jumping in popularity. We use the two following policy to achieve this:

- for every 10000 requests the injector makes, shift the value selected from the Zipfian

Table 5.1: Benchmark Set

Name	Distribution	Theta
YCSB A Zipfian	50% updates and 50% reads	0.9
YCSB C Zipfian	0% updates and 100% reads	0.9 and 0.75
YCSB E Zipfian	5% updates and 95% scans (scan size 500)	0.9
Shifting	0% updates and 100% reads	0.9

Table 5.2: Designs Tested

Name	Shortened Name	Description
Baseline-KVell	base	Original KVell
Baseline-KVell-30	base30	Original KVell using 30% page cache
Memory-Only	mem	Original KVell, all accesses from memory
Coalescing-No-Movement	coal-nim	Coalescing without persistent item movement
Coalescing	coal	Coalescing with persistent item movement

distribution down by 1

We use this to observe how our design reacts to continuous shifting of hot items.

5.2 Evaluated Software

We evaluate with the variations of KVell, depicted in 5.2. Excluding Baseline-KVell-30 and Memory-Only, these are run using enough memory to fit 2.5% of the database in memory at any given time. For Baseline-KVell, this means 2.5% page cache. For the coalescing designs, we split this 1.5% page cache to 1% Coalescing Space.

Memory-Only provides us with an upper bound on the effectiveness of our implementation, as essentially no matter how we implement this coalescing, it will not be better than with zero SSD accesses and no coalescing/profiling overhead.

We only evaluate Coalescing-No-Movement with YCSB C, as the write path for this model has been unimplemented. It is regardless quite evident that our model offers significantly improved performance with item movement than without.

cache

5.3 Hardware Setup

We use an Intel(R) Xeon(R) Gold 6330N CPU with dual-socket NUMA and 43 MB last-level cache for each core. The OS used is Linux, #129-Ubuntu SMP. We limit threads and memory used to a single NUMA node by using taskset and numactl. A SAMSUNG PM1735 3.2TB NVMe SSD is used for the datastore. Each NUMA node has 256 GB of RAM (or approx 251 GiB).

5.4 Empirical Results

5.4.1 Throughput Results

We observe that for all runs of Coalesced, there are periods during which throughput increases rapidly followed by relatively stable periods. We attribute this to the times at which the profiler send items to the Coalescing Space and induce slow evictions.

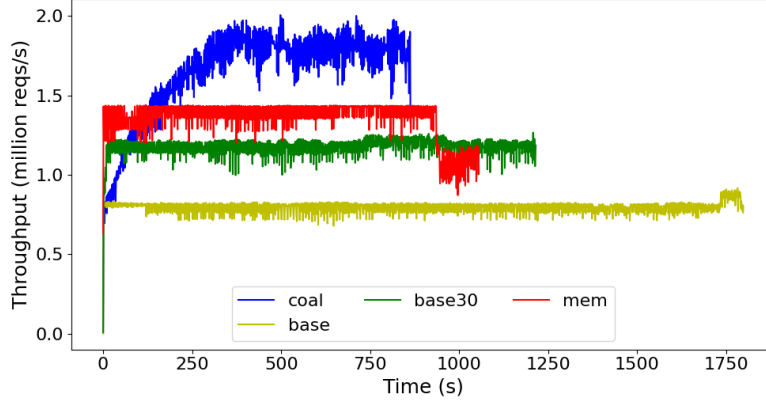


Figure 5.1: Throughput over time for YCSB A 0.9. Higher is better

Write-Intensive Workload (YCSB A)

For all of these workloads except Memory Only, all injectors run at the same speed. For the Memory Only case, there were two injectors running slower, hence the sudden dip in throughput around 900s that results from the other injectors terminating.

On write-intensive workloads, Coalescing achieves approximately $2\times$ improved average throughput during the bench compared to Baseline KVell. It also outperforms Baseline-KVell-30 by $1.4\times$, meaning that our implementation with combined page cache and Coalescing Space of 2.5% the size of database achieves significantly higher throughput than the original implementation which has $12.8\times$ the amount of available memory used for page cache.

Interestingly, Coalescing also achieves better throughput than Memory Only, despite Memory Only having no SSD overhead for the read portion of requests. There are a few contributing factors for this:

- The presence of fast and slow injectors may have skewed the access speeds.
- Rearrangement of hot items on the SSD means that there are more requests to the same pages in storage, which would be served faster due to increased locality

Read-Only Workloads (YCSB C)

On our 0.9-theta read-only workload, Coalescing achieves approximately $2.4\times$ improved average throughput during the bench compared to Baseline KVell. Similarly to the write-intensive workloads, our implementation also outperforms Baseline-KVell-30 on this workload. The improvement is slightly larger at $1.6\times$. This increased benefit to Read-Only workloads compared to write-intensive workloads is as expected, as the coalescing allows hot items to completely bypass SSD access for reads.

Coalescing-No-Movement shows an improvement of approximately $1.56\times$. This is sig-

5 Evaluation

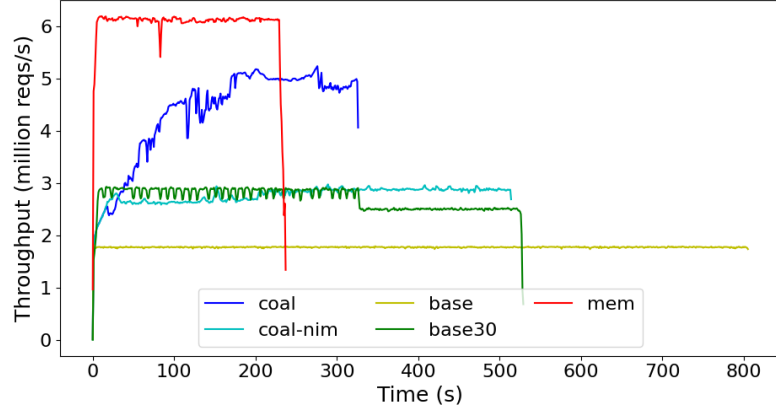


Figure 5.2: Throughput over time for YCSB C with Theta 0.9. Higher is better

nificantly lower than the benefits gained with coalescing.

Furthermore, Coalescing outperform the Memory Only case here. This is because our primary benefit in the case of read requests is reducing number of SSD accesses, and in the Memory Only case no accesses are made to the SSD later. Furthermore, as observed under latency analysis, a large proportion of requests in the Memory Only case were being handled in hardware cache as well.

Note for the Memory Only case, the data was being processed so quickly that our workers had significant wait times where they were idle (averaging approximately 10%. With more injectors, the observed value for Memory Only could be pushed even higher.

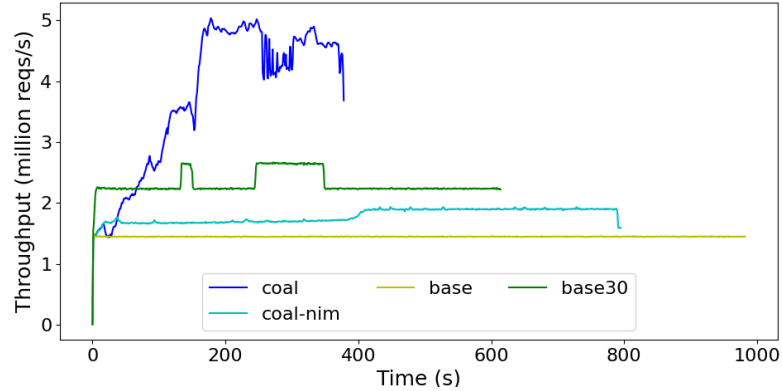


Figure 5.3: Throughput over time for YCSB C with Theta 0.75. Higher is better

We observe that with a Theta 0.75 workload (less skewed), Coalescing achieves a $2.6\times$ increase in average throughput compared to base Kvell. This is slightly better than under a Theta of 0.9.

However Coalescing-No-Movement shows significantly less benefits, with an improve-

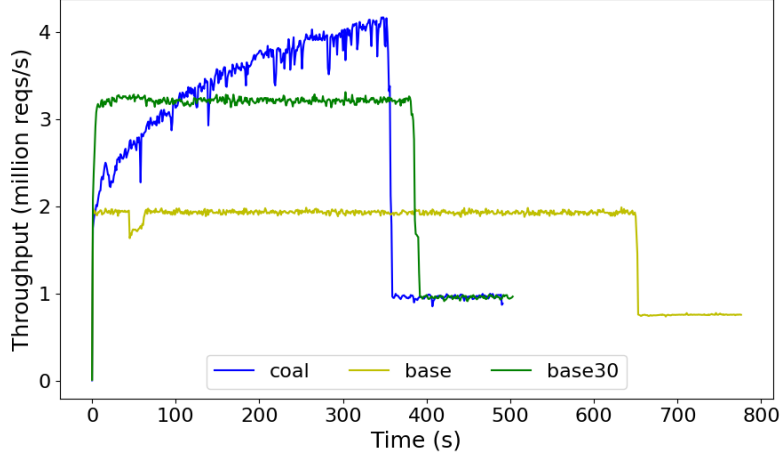


Figure 5.4: Throughput over time for YCSB E, with Theta 0.9. Higher is better.

ment of only approximately $1.24\times$. This is simply because the benefits of coalescing are not maintained on eviction, so the more evenly distributed the workload, the less effective coalescing will be as more accesses are handled outside of the Coalescing Space. Furthermore, more items are competing for space in the Coalescing space, leading to more evictions.

Scan-Intensive Workload (YCSB E)

Note that for all of these runs, there were two injectors that ran slowly (approximately $3/5$ the rate, speeding up after the other injectors had completed). This leads to the sudden drop in throughput present in all of the runs. This inconsistency may also skew the accuracy of these results.

Coalescing achieves almost $1.6\times$ baseline throughput on scan-intensive workloads. This is not as much as we expected, and is likely due to the added overhead of checking the Coalescing Space on every lookup of the scans. Coalescing also shows the same average throughput as Baseline-KVell-30. However, Coalescing does peak at a higher value, so over longer benchmark, Coalescing likely show more significant improvement.

Shifting Hotness

Our model is significantly less effective under gradual shifts. In fact, we observe that under a shifting workload, Coalescing-No-Movement offers significantly better throughput than Coalescing does. The reasons for this appear to be as follows:

- Our rudimentary profiler suffers in approximation of the total number of items accessed as the value increases, requiring items to be hotter over time to coalesce them
- Evictions from the coalescing space are continuously required. As these are done

5 Evaluation

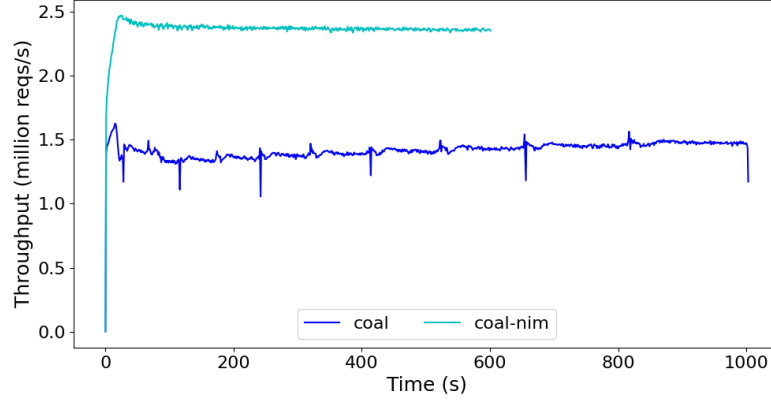


Figure 5.5: Throughput over time for the shifting workload, with Theta 0.9. Higher is better

Figure 5.6: Latencies for YCSB A with Theta 0.9. Lower is better

synchronously, and the benefits from them reduced, the tradeoff is much less advantageous

5.4.2 Latency Results

Average latency has dropped as expected, because we are doing fewer requests to storage. However, tail latency has also dropped significantly. This is surprising because over 5% of requests are still going to SSD in all benches.

We believe that this is due to locality of the accesses to the SSD, as the accesses that are going to the SSD are targeting a much smaller subset of pages in the SSD, and batching may reduce some of the overhead. It may also be due to the queue depth used for KVell being too small for the volume of requests that require SSD accesses, and our Coalescing reduces this volume.

Write-Intensive Workload

Interestingly, Coalescing achieves a better 95p latency than Memory-Only. This is odd as we are still doing read requests to the SSD in Coalescing. The difference likely comes from the locality of access to the SSD, as we do not reduce the volume of SSD requests compared to the Memory-Only.

Remaining Workloads

We do not discuss the remainder in detail as it is effectively what we expect. We give a brief summary here

- Coalescing has the lowest average and tail latencies except for Memory-Only in all

but YCSB E (scans). In particular it achieves a 70% decrease in average latency and a drop in 95p/99p latency of 58%/43% respectively for YCSB C with Theta 0.9

- In the write-intensive workload, Coalescing achieves similar average latency to Baseline-Kvell-30, and better tail latency
- For scanned workloads, Coalescing has worse tail latency than Baseline-Kvell-30 but still presents a significant improvement to Baselin-KVell

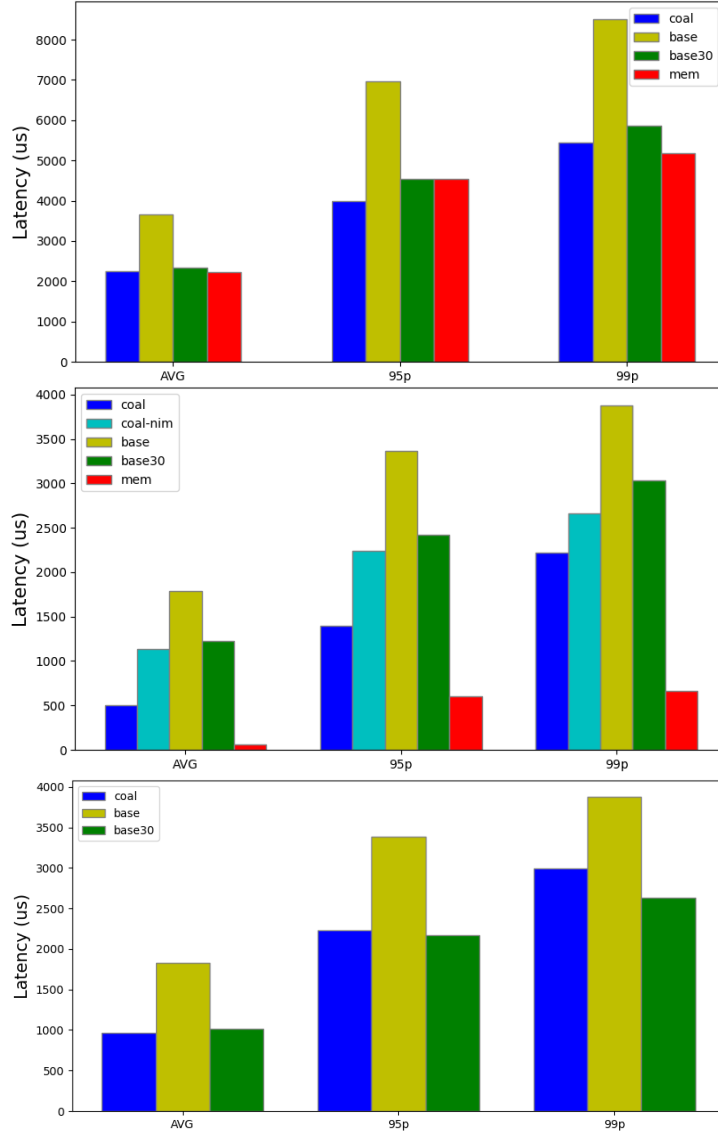


Figure 5.7: Latencies for YCSB A, (Upper) YCSB C (Middle) and YCSB E (Lower) with Theta 0.9. Lower is better

5.4.3 Evictions

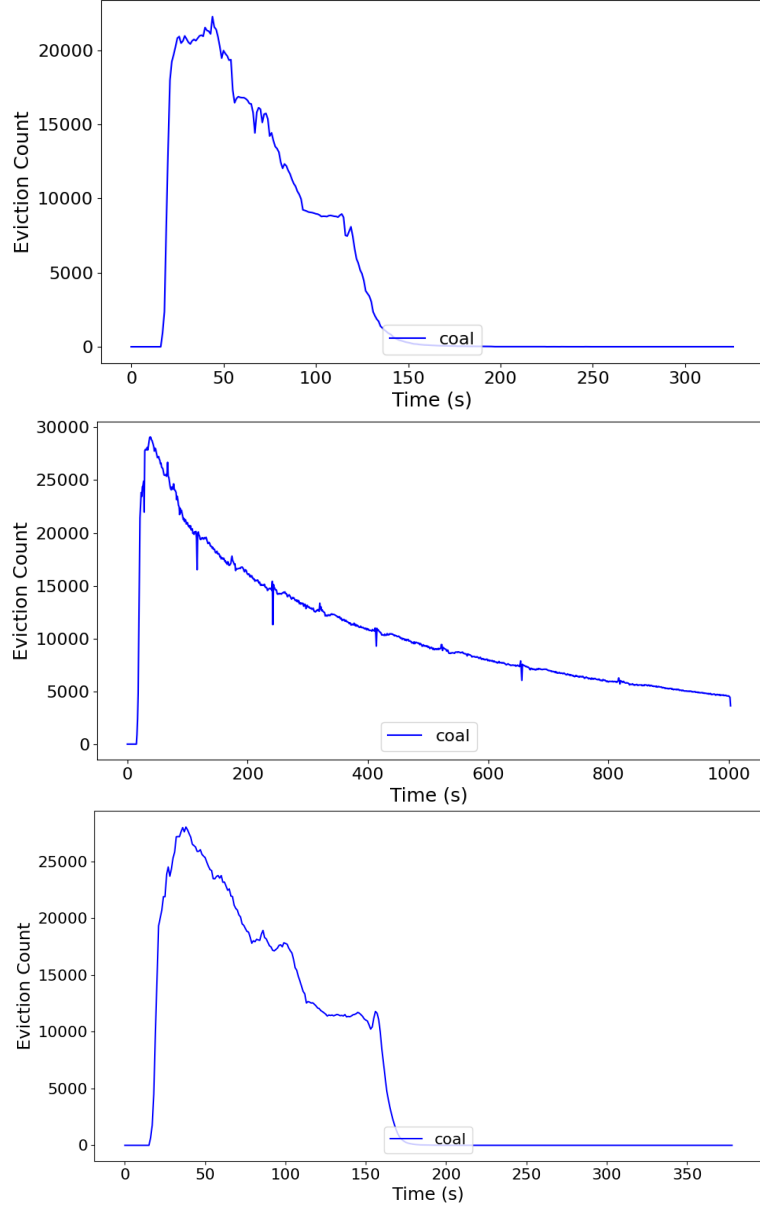


Figure 5.8: Eviction counts over time for YCSB C and Shifting workload with Theta 0.9 (Upper and Middle) and YCSB C with Theta 0.75 (lower)

We observe that evictions to storage cause significant delays. For example in YCSB C with Theta 0.75, we can see that there is a drop in eviction count at approximately 180s into the benchmark. This is precisely the time at which throughput spikes by its highest amount 5.3. The lack of evictions after this period is likely the major contributing factor to the effectiveness of Coalescing.

We can see from Figure 5.8 that the shifting workload has constant evictions, which degrades performance severely. We also observe that the decrease in eviction counts over time is likely due to a combination of the profiler registering fewer items as hot over time and more hits in semi-hot pages that had been coalesced earlier in the bench. However, the large quantity of remaining evictions is clearly causing performance degradation.

5.4.4 Requests Served by SSD

We track the number of requests that have been sent to the SSD over the benchmark. This includes any reads that were sent to support a later write.

It is clear that coalescing significantly reduces the number of requests that go to the SSD, in particular when we include item movement. However, in the case of a shifting workload, approximately 81% of requests went to the SSD. Coalescing-No-Movement has about 50% of requests going to SSD, likely because it always keeps the hottest set of items in the coalescing space instead of assuming they will be picked up in the page cache later.

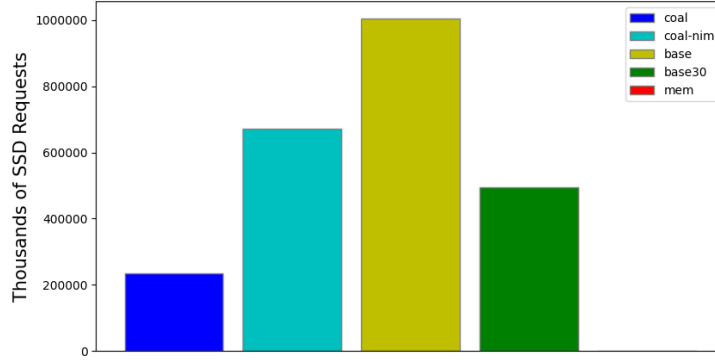


Figure 5.9: Number of requests sent to SSD for YCSB C with Theta 0.9

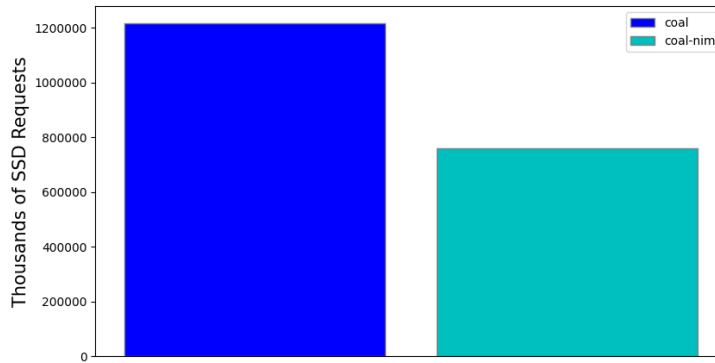


Figure 5.10: Number of requests sent to SSD for shifting workload with Theta 0.9

Concluding Remarks

6.1 Conclusion

We have implemented an item-granularity cache-like structure built to work with KVell’s pagecache to collect and reorganise hot items. This includes a policy of shifting item locations on SSDs for hot items to maintain the benefits of coalescing while freeing up more room in the coalescing space for more items.

Our evaluation has demonstrated that there is significant merit to tracking and coalescing hot items onto the same pages in KVell. Coalescing provides significant improvement in both read and write-intensive workloads, and has comparable or better performance to original KVell with a significantly larger pagecache. However, the improvement to scans, while still evident, seems to be limited in our implementation. This is likely due to the aforementioned policy of requiring a lookup in the coalescing space for every item in the scan to ensure consistency. We have also demonstrated that there are potential workloads and access patterns where we wish to suspend item movement on SSDs.

6.1.1 Limitations

There are several limitations to this work and the validity of the results. In both KVell and our design, sometimes when running workloads, the injectors run at significantly different speeds. When this happens, some injectors will finish before others, underloading the datastore for a large proportion of the run. This paints an inaccurate picture of the throughput of the database, as the relation between throughput and average latency isn’t maintained. Thus, for comparability purposes we have attempted to only include runs where the threads run at the same speed. This problem seems exacerbated for scan-based workloads.

The causes of this are yet to be found. We have attempted several mitigations and considerations, which have improved quality of the results but not fixed this issue

6 Concluding Remarks

- To minimize overhead from NUMA, we use taskset and numactl to ensure that all threads are on the same CPU and NUMA node.
- To ensure consistency in core speeds for different runs, we give time to cool down between runs.
- In case it is caused by having too many injectors attempting to submit requests at a time causing certain injectors to be referenced, we limit the number of injectors. This seems to minimise the issue, however it is infeasible for scans which require a larger number of injectors to saturate the workers.

Furthermore, our current implementation only functions when the number of items on each slab is divisible by the number of items per page on that slab. For example, in our workloads we use 500 million items with 32 items per page. Thus we use 20 workers, getting 25 million per worker. 25 million is divisible by 32, so this will work.

6.2 Future Work

There are several steps that can be taken to improve the design from here:

- Evicted pages should eventually be allowed to have its items undergo coalescing again, as over long periods of time, some of those items may become hot again while others remain cold
- We observed that workloads with continuous shifts in hot items suffer from our item movement policy. Thus, one potential avenue of future work would be to dynamically switch between coalescing with and without movement of items on storage depending on the characteristics of the current workload. to check at the end of serving a read that was awaiting data to arrive in the page cache whether there is a version in the Coalescing Space.
- Our current profiling is done for independent items. One could instead identify hot scans and do coalescing specifically with items in those scans
- Another consideration is that profiling should also be reset periodically to lower the threshold for coalescing items after approximation of new item counts becomes significantly inaccurate
- Garbage collection should be done on old copies of items on SSD to reduce space amplification

Bibliography

2024. *NVM Express Base Specification*. NVM Express. Revision 2.1. [Cited on page 5.]
2024. *Serial ATA Advanced Host Controller Interface (AHCI)*. Intel. Revision 1.3.1. [Cited on page 5.]
- CELIS, P., 1986. *Robin Hood Hashing*. Ph.D. thesis, University of Waterloo. [Cited on page 15.]
- CHANDRAMOULI, B.; PRASAAD, G.; KOSSMANN, D.; LEVANDOSKI, J.; HUNTER, J.; AND BARNETT, M., 2018. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18 (Houston, TX, USA, 2018), 275–290. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3183713.3196898. <https://doi.org/10.1145/3183713.3196898>. [Cited on page 9.]
- CHEN, J.; CHEN, L.; WANG, S.; ZHU, G.; SUN, Y.; LIU, H.; AND LI, F., 2020. HotRing: A Hotspot-Aware In-Memory Key-Value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 239–252. USENIX Association, Santa Clara, CA. <https://www.usenix.org/conference/fast20/presentation/chen-jiqiang>. [Cited on pages 2 and 9.]
- CONWAY, A.; GUPTA, A.; CHIDAMBARAM, V.; FARACH-COLTON, M.; SPILLANE, R.; TAI, A.; AND JOHNSON, R., 2020. SplinterDB: Closing the bandwidth gap for NVMe Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 49–63. USENIX Association. <https://www.usenix.org/conference/atc20/presentation/conway>. [Cited on pages 8 and 10.]
- COOPER, B. F.; SILBERSTEIN, A.; TAM, E.; RAMAKRISHNAN, R.; AND SEARS, R., 2010. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10 (Indianapolis, Indiana, USA, 2010), 143–154. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1807128.1807152. <https://doi.org/10.1145/1807128.1807152>. [Cited on pages 7 and 21.]

Bibliography

- DECANDIA, G.; HASTORUN, D.; JAMPANI, M.; KAKULAPATI, G.; LAKSHMAN, A.; PILCHIN, A.; SIVASUBRAMANIAN, S.; VOSSHALL, P.; AND VOGELS, W., 2007. Dynamo: amazon’s highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP ’07 (Stevenson, Washington, USA, 2007), 205–220. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1294261.1294281. <https://doi.org/10.1145/1294261.1294281>. [Cited on page 1.]
- DIDONA, D. AND ZWAENEPOEL, W., 2019. Size-aware sharding for improving tail latencies in in-memory key-value stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 79–94. USENIX Association, Boston, MA. <https://www.usenix.org/conference/nsdi19/presentation/didona>. [Cited on page 10.]
- GOOGLE, 2021. Leveldb. <https://github.com/google/leveldb>. [Cited on page 1.]
- JIN, H.; LI, Z.; LIU, H.; LIAO, X.; AND ZHANG, Y., 2020. Hotspot-aware hybrid memory management for in-memory key-value stores. *IEEE Transactions on Parallel and Distributed Systems*, 31, 4 (2020), 779–792. doi:10.1109/TPDS.2019.2945315. [Cited on page 10.]
- KELLY, R.; PEARLMUTTER, B. A.; AND MAGUIRE, P., 2018. Concurrent robin hood hashing. *CoRR*, abs/1809.04339 (2018). <http://arxiv.org/abs/1809.04339>. [Cited on page 15.]
- LEPERS, B.; BALMAU, O.; GUPTA, K.; AND ZWAENEPOEL, W., 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19 (Huntsville, Ontario, Canada, 2019), 447–461. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3341301.3359628. <https://doi.org/10.1145/3341301.3359628>. [Cited on pages v, 2, and 6.]
- META, 2022. Rocksdb. <https://rocksdb.org/>. [Cited on page 1.]
- WANG, K.; LIU, J.; AND CHEN, F., 2020. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. *Proc. VLDB Endow.*, 13, 9 (May 2020), 1540–1554. doi:10.14778/3397230.3397247. <https://doi.org/10.14778/3397230.3397247>. [Cited on page 9.]
- WU, F.; YANG, M.-H.; ZHANG, B.; AND DU, D. H., 2020. {AC-Key}: Adaptive caching for {LSM-based}{Key-Value} stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 603–615. [Cited on page 10.]