Key-Value Stores on Flash Storage Devices: A Survey

Krijn Doekemeijer
Vrije Universiteit Amsterdam
Universiteit van Amsterdam

Abstract

Key-value stores (KV) have become one of the main components of the modern storage and data processing system stack. With the increasing need for timely data analysis, performance becomes more and more critical. In the past, these stores were frequently optimised to run on HDD and DRAM devices. However, the last decade saw an increased interest in the use of flash devices because of their attractive properties. Flash is cheaper than DRAM and yet has a lower latency and higher throughput than HDDs. This literature survey aims to highlight the changes proposed in the last decade to optimise key-value stores for flash devices and predict what role these devices might play for key-value stores in the future.

Keywords. Flash storage, SSD, NVMe, Key-value stores, NoSQL, LSM-tree, B-tree, Hash table

1 Introduction

It is estimated that we will generate over 175 zettabytes of data globally by the year 2025 [110]. This is mainly because of the ever-increasing interest in big data, the cloud and the internet of things [71, 110, 130]. As the size of the datasets keeps increasing, so do the demands of the systems that are used to store and process this data. This in turn has caused for an increased interest in optimising the data processing stack. A big part of this stack is used by key-value stores. It is therefore beneficial to look into how key-value stores can be optimised.

Key-value stores are a means of storing data and are radically different from the more traditional RDBs, also known as relational databases [59]. Key-value stores store data as a single collection, where each key is unique and leads to one value. Data can be accessed using these keys with basic operations such as: get, put, delete and scan. Key-value stores can be used for all sorts of applications and are not limited to a particular size or hardware. Some common applications include caching systems [50], messaging applications [21], games [42], web shops [119], SQL backends [48] and time series management [75].

Traditionally the main storage medium used to store key-value stores was the *Hard Disk Drive* (HDD) [35]. Most data structures and algorithms were thus optimised around the physical properties of these devices. These were among others high latencies, symmetric read and write speeds, slow random access and an infinite number of reads and writes for each block on the HDD. This caused certain HDD specific optimisations such as trying to always write and read sequentially [25, 51, 54, 101, 106].

However, as flash devices became cheaper, many data centres and consumers alike transitioned to flash devices [5, 85]. This made it important to ensure that applications can still be properly used with flash devices and are in addition also optimised for these devices. (Un)fortunately, most of the properties and assumptions that hold for HDDs, do not hold for flash devices. Flash devices have lower latencies, have asymmetric read and write speeds, do not allow for in-place updates, have an all-new erase operation, are indifferent to random reads on the cell level and individual cells have a finite life cycle. The finite life cycle, commonly known as *wear levelling* (WL), can in particular be problematic if unattended. If applications carelessly keep writing to the same cells, the cells will eventually stop working correctly. Lower latencies are also important as lower latencies are becoming more critical for applications [10]. Yet, at the same time lower latencies on flash result in the latency overhead moving to other parts of the key-value store, such as the software that is executed on the host, and therefore require different design considerations [10]. Because of such idiosyncrasies, properly and efficiently using these devices requires a transition [60].

This survey tries to highlight the changes proposed in the last decade for using key-value stores on flash. We will look into various optimisation strategies that can be used to use key-value stores more efficiently on flash. However, first we will take a look at flash and key-value stores themselves. We will then combine the two topics and take a look at the main design concerns that occur when combining them. After having defined the problem space, we will show how these problems

can be solved. We will start by covering the commonly used data structures for key-value stores. This will be followed by how key-value stores and flash communicate with each other and how this communication can be optimised. Then we will look at software optimisations, data related optimisations and the current role of flash. Lastly, we will identify forward-looking trends on the role of flash for key-value stores.

2 Related work

Key-value stores and flash are by no means new technologies, both already appearing in the 20th century [46, 125]. In this section we will look at other surveys on these topics and cover a few adjacent works. There have been a few surveys that cover flash storage [30, 53, 143] and there have been a few surveys on key-value stores [22, 36, 63]. Most of the related works on key-value stores focus on NoSQL [22, 36, 63]. It is important to note that key-value stores are considered to be a type of NoSQL, but that there also exist various other NoSQL stores that are by some considered key-value stores and by others not. Further on, some data stores satisfy the requirements of multiple types of NoSQL. This makes the exact definition and nomenclature of key-value stores blurry. To give an example, Cassandra can be considered as a wide-column store [57] but is at the same time also frequently referred to as a key-value store [101]. In this survey we will refrain from using the term NoSQL.

To give a few examples of NoSQL surveys: Idreos et al. cover various key-value storage engines [63], Cartell et al. cover scalable SQL and NoSQL data stores [22], and Chen et al. did a survey on NoSQL [36]. There have also been literature studies on various storage techniques for commonly used data structures in key-value stores, such as on LSM-trees [96].

To the best of our knowledge, there has not been a publically available academic survey that covers key-value store techniques or general optimisations for key-value stores driven by flash. There do exist a few works that aim to help with understanding key-value stores, such as a practical overview by Seegers et al [124] on a few key-value stores. Other works that we found only used key-value stores. Such as a survey on multimedia systems [111], data storage in the cloud [98] and a survey on big data systems [37].

For this survey, we are also interested in flash storage. There have been various surveys on parts of flash storage. Such as surveys on indexing in flash storage [53], the flash translation layer [30] or garbage collection and wear levelling [143]. We will provide a more comprehensive introduction to these topics in Section 4.

This survey aims to build the bridge between key-value stores and flash storage. It will thus combine topics from key-value store (NoSQL) surveys and flash storage surveys. In other words, the focus will not be on the technologies separately, but on the combination of the two.

Lastly, we shortly address some adjacent works on new storage technologies. Recently there have been hardware trends that introduced various new storage devices that could also be used as memory; such as Optane memory. A lot of effort has already been put into moving key-value stores to these devices [13, 67, 148]. A further study can look into these devices with regards to key-value stores. Similarly there have been new types of SSDs, made specifically for key-value stores, which are aptly named as KVSSDs. This could also be a valuable target for further investigations and has already garnered a healthy amount of interest [64–66, 83, 114, 139].

3 Study design

In this section we will describe the research goal and how this goal has been reached. In particular, we will go over the research questions, the scope of the research project and the methodology used to achieve the research goal.

3.1 Research Goal

This study aims to get an overview of how various key-value stores are optimised for flash storage. The main research question is: “What is the impact of flash storage on the design choices for key-value stores?". The following sub-questions were asked to aid in answering our research question:

- RQ1: What is the current role of flash for key-value stores?
- RQ2: How has flash influenced the design of key-value stores over the decade?
- RQ3: What are the main challenges involved in using key-value stores on flash and how can they be mitigated?
- RQ4: How will flash contribute to key-value stores in the future?

3.2 Scope

This study is about persistent key-value stores for flash devices that specifically use the block interface. Therefore, it does not cover caching or other non-persistent workloads. It also does not cover other promising novel non-flash solutions such as persistent memories. Key-value SSDs will also not be covered since they are object-addressable and are thus not a block-based technology. Further on, we will mainly focus on how the key-value stores are optimised for flash devices. We will for example not cover how key-value stores are optimised for the network, even if flash is involved. Lastly, we only cover relatively recent contributions. In this case from approximately 2010 up to early 2022, but exceptions can be made to create context. From these requirements, we set up the following inclusion & exclusion criteria:

- I.1: The work is novel.
- I.2: The work should be either about how flash impacts the design of key-value stores or how key-value stores can be optimised for flash.
- I.3: The key-value store must match the criteria of a persistent key-value store.
- E.1: The work is about ephemeral key-value stores such as caching.
- E.2: The work targets only non-flash solutions such as Optane memory or hard disk drives.
- E.3: The work merely uses flash or key-value stores but is not about them.
- E.4: The work is not relatively recent, main points should be from 2010 and up; papers from before 2010 can only be used for historical context.

Not every paper that satisfies these requirements is included. Papers are more likely to be included, in order of importance, if flash and key-value stores are the main topics, if they are cited frequently, if they are novel or if they were published recently.

3.3 Methodology

To accurately answer the research questions, it is important to select and evaluate a large selection of papers. Nevertheless, it is a Herculean effort to read every paper. Thus, only a selection of all papers related to the topic will be examined. A common approach is the *Snowball methodology*. With this methodology, you start with a few seed papers and recursively find related papers. This can be done by either looking at the references used in the paper or by verifying which papers have cited these papers themselves. It is also a good idea to start with at least a few recent papers, considering that they might bring up some novel ideas or terminology that did not exist before. The seed papers used in this research are:

Paper	Source	Year
NVMKV [99]	ATC	2015
LOCS [136]	EuroSys	2014
SILT [90]	SOSP	2011
SILK [9]	ATC	2019
WiscKey [94]	ACM TOS	2017
PebblesDB [116]	SOSP	2017
RocksDB space amplification [48]	CIDR	2017

Table 1: Seed papers. Paper titles are shortened to make space.

There are also a few conferences that typically reserve a few spots on storage topics. These were scraped from 2021 to 2012. This was done by iteratively checking the papers of the conferences and verifying if they might be relevant. Relevance is determined according to the rules defined in Section 3.2. The names of the conferences considered were: *USENIX ATC*, *FAST*, *NSDI*, *EuroSys*, *OSDI*, *SOSP*, *Systor*, *HotStorage*, *ASPLOS*, *SIGMOD*, *VLDB*, *SoCC*, *ICDCS* and *Middleware*. Lastly, a few papers were found by simply querying Google Scholar (sorted on relevance and limited up to 2022), Connected Papers(using Semantic Scholar), USENIX, ACM, DBLP and Semantic Scholar with keywords. The keywords were picked based on words that are considered important in the seed papers and conferences. For some keywords, we also tried various synonyms and closely related words. The used keywords are:

- Flash Key-Value
- Flash KV
- Flash NoSQL
- NVMe Key-value
- NVMe KV
- KVSSD
- SSD key-value
- SSD KV
- persistent key-value store flash
- LSM SSD
- LSM NVMe
- Btree SSD
- Btree NVMe
- Btree Key-value store SSD
- Key-value garbage collection
- Key-value wear levelling
- Key-value write amplification
- Key-value read amplification
- Key-value space amplification

At times the search results were irrelevant to the research because they failed to meet the requirements. Frequently they were about caching, NVM as memory and other devices such as SMR disks or Optane. These have to be filtered out by hand. Many papers reoccurred across different search tools, which should make it more reproducible to get a part of the papers. Nevertheless, for best results, it is advised to set the range in years to a maximum of 2022. The literature study

was conducted in January 2022 and can thus not contain any more recent papers. Also note, that the order of results can fluctuate between the years and different people might assess papers differently on the provided criteria. We aim to provide sufficient details about our methodology to aid the reproduction of work.

4 Background

Before delving into various optimisation techniques, we take a look at the problem space. We will first take a look at key-value stores and flash technologies separately. Then we look at the combination of the two and formulate the main performance concerns that arise when combining the two.

4.1 Persistent key-value stores

Key-value stores are ubiquitous [21, 42, 49, 63, 119]. They are a type of NoSQL store, but we will in general refrain from using that term in this survey since its exact definition can be blurry and there exist multiple NoSQL stores that have a KV-like design. Such as can be seen in a survey by Davoudian et al. [36] on NoSQL. Therefore, we will provide a different definition. Key-value stores revolve around a flat structure with key-value pairs. Each key is unique and points to data. This data can in general be anything and restrictions depend entirely on the specifications of the store at hand. Key-value stores are considered to be lightweight and to implement only a few standard actions. These actions are specified in Table 2.

Operation	Explanation	Required
put	Stores key-value pair	Yes
get	Retrieves key-value pair	Yes
delete	Deletes key-value pair	No
scan	Retrieves range of key-value pairs	No

Table 2: Key-value operations.

Key-value stores should at least implement a *put* and a *get* operation. Put associates a key with a value and set retrieves said value if it exists. The *delete* action is also regularly included, but can also be achieved with a put with an empty value. Lastly, most key-value stores define a *scan* operation to get a range of key-value pairs, but this is not required. An advantage of a scan operation is that if you already know that you need a range, you do not need to send multiple get requests. The key-value store can then also already optimise for such an operation. Lastly, not all key-value stores have to be persistent. They can for instance also be used for caching. We will only focus on persistent solutions. These in general follow BASE [112], but some also follow ACID [122]. This means that key-value stores are in general less strict on consistency. Some have even fewer restrictions than BASE. Some examples of popular key-value stores are RocksDB [49], LevelDB [54] and DynamoDB [126]. With such a small set of requirements, it can and is used in all sorts of different applications. It can for instance be used as a backend for relational databases [49], for gaming services [42], for web shops [119], for machine learning [21] and the social web [21].

That different workloads are used has performance implications. Not every application needs low latency or high throughput. For some it might be more beneficial to invest as little as possible in storage devices. This means that there is space to optimise key-value stores for different goals, which is also reflected in the literature that we will cover in this survey. Certain stores also optimise for a certain ordering of data. We will see a few of these optimisations later on.

Key-value stores have few constraints on the actual implementation and therefore contain a wide variety of implementations. They all make use of *data structures*. Data structures are structures that can be used to store, organise and access data. An example data structure would be an array. In key-value stores, the main data structure is used for *indexing*. Indexing means a method to get a reference to a value, think about getting a value based on a key. Indexing structures reserve a part of the memory. For example, it is very typical to keep the references in memory and store the actual values on storage. This can become problematic when the amount of memory required for each entry increases, as it hinders the maximum amount of key-value pairs that can be stored. It is therefore typical to also move a part of the index to storage. Moving a part of the index to storage does have a disadvantage. It requires more read and write operations to storage for each key-value store operation. That is because the index now also needs to be read and written. The amount of extra I/O operations needed for each logical operation is known as I/O amplification. Index structures have to make a trade-off between memory usage and I/O amplification. This trade-off is also highly dependent on the storage medium. For example, if the storage is slow, it becomes less advantageous to move more of the index to storage. Therefore, key-value stores that are optimised for flash have to consider the flash specifics for optimal performance, specifics which we will see in the next section. Commonly, additional data structures are used on top of the index to minimise the amount of I/O, such as buffers. Some common data structures that are used for indexing in key-value stores are LSM-trees, B-trees and hash tables. All three of these data structures will be explained in further detail in Section 5. For now, it is enough to know that each of these has its advantages and disadvantages.

4.2 Flash storage

There are quite a few characteristics that make flash memory stand out, and flash memory itself is not just limited to one design. In order to properly optimise key-value designs for flash, it is imperative to first define what types of flash are used

in key-value stores and what characteristics these types of flash poses. This should then allow us, to properly define the main performance concerns. For more in-depth explanations on flash, there exist various more in-depth explanations on flash, including explanations with more technical details [7, 14, 30, 85, 102, 143].

Flash is a non-volatile storage medium where data is stored as an electronic charge on a floating gate between a control gate and the channel of a CMOS transistor. We will not go to deep on what such technicalities entail. In fact, for this survey we only need to understand a couple of details. The number of bits the earlier mentioned gate can store depends on the internal cell technology. For example there are *single-level cells* (SLC) and *multi-level cells* (MLC) technologies. When the number of levels in a cell increases, so does the number of bits it can store. These technologies are as of now in order of their density: SLC, MLC, TLC, QLC and PLC [97]. Multi-level cell technologies allow us to store more data in a smaller area, but also have downsides. Multi-level cell technologies namely result in lower throughput and an increased wear levelling [2]. Wear levelling means that after each write, the cell slowly degrades. This results in a decreased performance and eventually leads to a dead cell. It is therefore considered harmful to do excessive writes on flash. Cell degradation is also not just limited to writes, reads can also degrade the performance of cells, albeit to a lesser degree. This is known as read disturbance [91].

NAND flash is a technique that can be used to pack the earlier mentioned flash cells together. NAND packs the cells together densely, which is a good fit for commercial mass storage but only allows addressing cells on the block level [34], which can be any size such as 4 KiB. NAND flash does not allow writing to an already written block. It first has to be erased, before it can be written. Therefore updating a single bit in a block, will result in writing an entirely new block and at some point in time erasing the old block. A NAND package is organised hierarchically in pages, that are stored in blocks, that are stored in *planes*, that are stored in *dies*. NAND packages can be processed in parallel. Planes can also process requests in parallel. Further on, NAND cells themselves can also be packed vertically, resulting in 3D NAND. This is a property that also should be addressed since the heat of one area also propagates vertically and can therefore increase the wear levelling of neighbouring cells [137].

NAND flash can then be used within a storage device, such as *Solid State Drives* (SSD) [34]. Note that SSDs can also use different types of storage other than flash. SSDs that use flash internally allow for parallelism through the number of independent NAND flash chips, which allows it to process multiple write, read and erase requests in parallel [136]. This parallelism can be exposed to the host in the form of flash *channels*. SSDs do not just include non-volatile storage, they come with a controller that allows managing the device and usually come with a small bit of DRAM. SSDs are known as solid-state disks because they contain no moving parts. This is in contrast to more traditional storage such as HDDs, which require moving a platter and an arm. This allows them to deliver both adequate sequential and random I/O, which would not be possible if parts had to move. There also exist various ways to connect SSDs; such as AHCI through SATA and NVMe through PCIe [82]. Depending on the connection, different protocols can be used and different levels of throughput, latency and concurrency can be achieved.

Lastly, SSDs frequently contain firmware. These are typically a *File Translation Layer* (FTL) and a *Garbage Collector* (GC) among others. An FTL maps virtual addresses to physical addresses [30]. This allows the SSD to determine what it considers an optimal location to store a block. It also removes the need for the host to issue erasure commands. For the host, overwrites are still allowed. The FTL determines how to manage such I/O internally. The Garbage collector (GC) is then used to erase dead blocks and move blocks around to more beneficial locations [143]. An optimised program should account for the internal logic of this firmware. There also exist a few alternatives that allow the host to drop the need of an internal FTL and GC (these must then be defined on the host) such as open-channel SSDs [19] and ZNS [18]. Open-channel SSDs are already used in a few key-value designs as we will see in Section 6.3. Note that we have intentionally left out some additional challenges for flash devices. *The Unwritten Contract of Solid State Drives* by He et al. is an interesting read on some additional challenges [60].

4.3 Performance challenges

In Section 4.1 and in Section 4.2 we respectively defined key-value stores and flash storage. We have already seen some challenges that are involved for flash and key-value stores separately, now we will describe the challenges involved in combining the two. We identify the following challenges that are commonly referred to in the literature: write amplification, read amplification, space amplification, garbage collection overhead, memory footprint and software overhead.

Write amplification: Write amplification (WA) means that the physical amount of data that is written on the device is more than the logical data that is written. For example writing 64 KiB of data to the device, when the application issued only 2 KiB. This can be because of the block size, garbage collection and internals of various data structures. Reducing write amplification seems to be an especially prevalent problem in studies. That is because write amplification can significantly reduce both throughput and increase latency. Another problem is wear levelling, which can degrade the flash devices and eventually make them unusable. Write amplification causes wear levelling and therefore inevitably also increases the monetary cost of using these devices. The effect of write amplification is also bigger in multi-cell technologies, such as MLC and TLC.

Read amplification: There are also a few works on reducing read amplification (RA). Read amplification means that the amount of device-internal data that is read is more than the user-visible data that is read. For example, issuing a read of 2 KiB and in the end reading a total of 64 KiB. This can for example be because the application needs to find the data, requiring more reads, or because of the block size of flash. This can be problematic in read-heavy stores that require low latency for reads. Nevertheless, write amplification is generally more problematic than read amplification because writes cause wear levelling, which is more harmful than read disturbance, and because of the asymmetric nature of I/O, which makes write operations more expensive than read operations.

Space amplification: Space amplification and space efficiency can also be a problem. Space amplification is an application problem and means that the key-value store stores more data on storage than there are key-value pairs. This is inevitable, as some metadata is always needed for persistency. Further on, some key-value stores also reserve space for background operations, further limiting the amount that can be used for key-value data. The main concern is to keep the amount of extra data needed to a minimum. It is also a possibility to resort to compression, to reduce the amount of data that needs to be stored, sometimes creating negative space amplification. Space amplification can be problematic because it results in a need to invest in more flash because there is not enough space.

Garbage collection: Garbage collection is also frequently stated as a problem. In this case, we refer to garbage collection in the device, as described in Section 4.2. Garbage collection causes erasure operations and moves data around in the background. Since the data is moved around, this can cause further write amplification. The background operations also introduce significant latency fluctuations. Works that focus on this problem aim to either reduce the latency fluctuations, use hints to help the garbage collection or properly separate hot and cold data, which helps the garbage collection by only moving hot data.

Memory footprint: Reducing the memory footprint is an issue that is mostly stated in situations where the usage of DRAM is more expansive or not a lot of DRAM is available in the first place. This is more prevalent in studies of the early 2010s. Generally, most of the data such as values are already stored on flash, but a part of the index and some caching is typically stored in memory, as described in Section 4.1. When memory is scarce, it is not feasible to keep full index data structures or big caches in memory, so these works try to reduce memory footprint with more lightweight index data structures and move more data to flash.

Software overhead: Lastly, some works aim to reduce software overhead on the host. These works mainly consider flash devices with lower latency and higher throughput, such as NVMe SSDs. In this case, the CPU on the host can become the performance bottleneck. Therefore, it can be beneficial to streamline the software stack and focus on performance techniques on the host side.

From these challenges we formulate the following main concerns:

- Q1: How to reduce write amplification?
- Q2: How to reduce read amplification?
- Q3: How to reduce space amplification?
- Q4: How to deal with garbage collection?
- Q5: How to reduce the memory footprint?
- Q6: How to reduce software overhead?
- Q7: How to integrate flash into an architecture and use each component efficiently?

The challenges and concerns introduced in this section answer research question RQ3: “What are the main challenges involved in using key-value stores on flash and how can they be mitigated?". In the next sections, we will cover various solutions that aim to find solutions to these concerns. We will find that there is no such thing as a free lunch. Indeed no solution is the solution to all our problems. On the contrary, most solutions we find will come with a trade-off or might be specific to only a few use cases.

We will give a short overview of the papers covered in the section and what concerns they address at the beginning of each section. Concerns will be marked with a corresponding Q number as given in the concern list we have just given. For example, concerns about write amplification will be referred to with *Q1*.

5 Data structures for flash

Various data structures can be used to implement persistent key-value stores. Together they represent the internals of the key-value store. Nevertheless, there have been three data structures in particular that have been most prevalent for key-value stores in the last decade. These are all index data structures as explained in Section 4.1. The data that they index to are the key-value pairs. Keys can also be used as the pointer. The three data structures are *Hash Tables*, *B-trees*, and *LSM-trees*. Many ideas have been proposed to either optimise these data structures for flash or use them as a part of their key-value store. Therefore, we will take a closer look at these structures and go into detail about how they can be optimised for flash. Of these three data structures, LSM-trees seem to have become the most commonly used and investigated. LSM-trees will therefore also be discussed most prominently. However, we will first take a short look at *approximate membership queries* (AMQ), which can be used along with all three data structures to increase performance.

Problem	Solution	Examples
Q2	AMQs	BloomStore [93]
Q1,Q2,Q3, Q5,Q6	Traditional hash solutions	BufferHash [3], ChunkStash [43], FlashStore [41], SkimpyStash [42], FAWN [4], uDepot [77]
Q1,Q2,Q4, Q5	Multi-stage hash solution	SILT [90]
Q1,Q2,Q4, Q5,Q6	B-tree solutions	FlashDB [103], TokuDB [131], WiredTiger [104], Tucana [107], ForestDB [1, 84], SplinterDB [32]
Q1,Q2	LSM-tree	LSM-tree [106]
Q1,Q2,Q4	LSM-tree: key-value separation	WiscKey [94], SardineDB [47], RocksDB [49], HashKV [24], DiffKV [88]
Q1,Q2,Q3, Q4,Q5,Q6	LSM-tree: reduce compaction overhead	PebblesDB [116], PTierDB [92], RocksDB [49], SifrDB [100], Accordion [20], SILK [9], LDC [23], TRIAD [8]
Q1,Q2,Q4, Q6	LSM-tree: Different index	Kreon [108]
Q3	LSM-tree: reduce space amplification	RocksDB [48, 49], FlashKV [147], LOCS [138]

Table 3: Overview of papers covered in Section 5.

5.1 AMQ

Approximate membership queries (AMQ), also known as filters, are a family of data structures that are used in close to all key-value solutions we will cover. They are data structures that are meant to prevent expensive I/O operations by approximating if a key is present in the key-value store or not [12]. An important characteristic is that they can lead to false positives, but not to false negatives. This characteristic can be used to prevent lookups. If the AMQ returns that it is not present, it is also definitely not present on flash. However, if the AMQ does return that it is present, it might be present on flash, necessitating a lookup.

One might already wonder about the use case for such queries, as they only seem to increase the amount of work necessary for each key-value operation. The reason is that AMQs are generally smaller than the data structure they approximate. The amount of space they require can be tuned, in this case space is traded for the accuracy of the AMQ. A smaller size allows it to be stored on a faster device that has less space and to be cached. If a device is fast enough such as DRAM, it can therefore eliminate the cost of accessing more expensive I/O such as flash. AMQs thus aim to solve issues with Q2. Some AMQs that we have seen in key-value stores optimised for flash include *bloom filters* [121], *cuckoo filters* [52] and *quotient filters* [12]. Bender et al. also proposed a *Cascading Filter* that combines multiple quotient filters and should also be efficient when parts of the filter are stored on flash [12], but AMQs should generally be kept completely in memory. Take note that many designs that we will see later on, can be improved by the use of such AMQs to prevent I/O operations. Lu et al. even propose BloomStore [93], a key-value design that revolves around using bloom filters as the main index structure.

5.2 Hash tables

Hash tables contain a wide array of different solutions, but they are all based on a common principle; they maintain a table of keys with a mapping to the address of their value, and can therefore also be referred to as *mapping tables*. This fits very closely to the definition of key-value stores, as key-value stores are also a collection of keys mapping to values. Each mapping in the mapping table occupies a location, which we will refer to as a *slot*. In all solutions we cover all values are maintained on flash, so only the offset to values is maintained in the mapping tables. Keys are stored along with the values but are regularly also a part of the mapping table itself. If keys are not part of the mapping table, either a hash of the key is used or a part of the key is used in the mapping table. The mapping table is cached at least partially in memory and can be constructed from the stored data. Hash table designs become problematic when a lot of key-value pairs are stored since the table increases linearly with the number of key-value pairs. Therefore not the entire hash table can fit in memory and a part needs to be repeatedly stored and loaded from persistent storage. This swapping between storage and memory can lead to read amplification when there is not a lot of memory. Further on, hash tables are also known to have problems with scan operations since keys and values are (typically) not sorted on flash. In this section we will see that a part of these problems can be mitigated. We will cover a few hash table designs, even some that consist out of multiple substructures.

5.2.1 Buffering

One of the first optimisations we will cover is *buffering*, which is deployed in many hash designs [3, 41–43, 90]. Buffering is

a technique that is not unique to hash designs, and a design we will also see in the other two index structures later on; see Section 5.3 and Section 5.4. The general idea is to keep items in memory until they collectively reach a certain size, which can be set as a threshold. To guarantee persistence, items should also be flushed to storage after a short amount of time has passed and the size threshold has not yet been reached. Buffering is done because most SSDs are block-based and force a minimum write size. Writing small strips of data would therefore require adding padding, which contains bytes that will not be used by the actual key-value store. This padding thus leads to write amplification. Buffering also removes the need for sending each operation to the device, essentially sending one big request. Sending each operation would incur a lower throughput, latency and more garbage collection (Q4) in the end. It would also incur a software overhead as each operation will need to be processed from beginning to end. Thus buffering can help by avoiding small I/O and batching operations (aiding Q6).

5.2.2 Reducing the memory footprint

Another problem is keeping the memory footprint to a minimum and at the same time keeping read amplification low (solving Q2 and Q5 issues). Key-value stores typically store all the values on flash in an append-only data structure, known as a *log*. Alternatives do exist but are not as common. SILT for example also stores values in a sorted log among others [90]. We will mainly focus on the log design, because of its frequency in usage [4, 41–43]. After the buffer is flushed, the data is appended to the log and the offset to said data is saved in the hash table. This forces sequential I/O on the storage device, which is considered better than random I/O. When a value is deleted or updated, the old entry in the log is marked as invalid. Eventually, such values *do* need to be erased and removed from the log, probably with a separate garbage collection process. Such a hash table design is visible in Figure 1.

A naive implementation would then simply conclude by making the hash table uniquely link the entire key to the offset of a value in the log, but we are not there yet. This implementation directly indexes full keys to value offsets. Unfortunately, full keys also take multiple bytes and this would not scale when billions of keys need to be saved. Instead, we need to find a way to only keep part of the keys or a part of each key in memory. For example what if we have 10 bytes of RAM and 5 keys of each 5 bytes. We would not be able to fit the entire hash table in memory. We either need to keep part of the key-value mappings on flash such as only 1 key, or we need to reduce the size of each mapping. We will cover a few examples of both approaches.

Our first solution is proposed by Andersen et al. in the form of *FAWN* [4]. FAWN reduces the memory footprint by only storing a part/fragment of each key in the hash table, reducing the memory footprint for each individual key-value pair. This allows more key-value mappings to remain in memory or to simply reduce the amount of DRAM used on the host. Such a design does have two problems. As only a fragment of each key is stored, it can lead to *hash collisions*, which means that two mappings end up in the same slot. Hash collisions can occur because two keys that are different can have exactly the same fragment. This would then force them to the same slot in the hash table, requiring multiple value offsets to occupy the same slot. Similarly, since you only know the fragment, you can not reliably tell if the key-value pair exists in the first place by only looking at the occupancy of the slot. FAWN solves this by storing the keys along with the values on flash, which allows verifying if it was indeed the correct key-value combination on lookup. Hash collisions are resolved in FAWN with the help of hash chaining. Slots in the hash table then contain chains of offsets instead of individual value offsets. Colliding keys are then added to the same slot in a chain in memory. Lookups can therefore issue multiple reads to find the correct pair because the entire chain might be read (creating Q2 problems). This is dependent on how big of a part is chosen from the key as a fragment, but Andersen et al. report numbers such as 1 in 1000 and 1 in 32,768 for having to do more than 1 lookup in their use case [4], indicating that extra lookups are rare. At the same time, this solution still requires some bytes in memory for each entry, even if it alleviates some memory issues by using fragments.

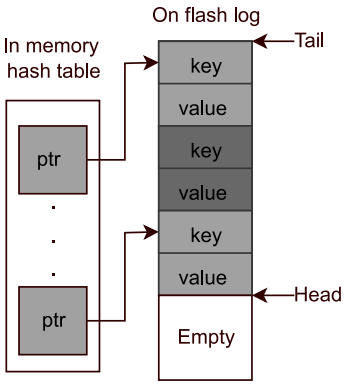

Figure 1: An example of a typical hash table. Pointers to key-value pairs are stored in RAM and values are stored in an append-only log. Darker key-value pairs can be reclaimed.

Debnath et al. take it one step further with the design of *SkimpyStash* [42], achieving on average less than 1 byte for each entry, while still using only one hash table. To achieve this they make use of a hash chain, similar to FAWN. However, instead of storing the chain in memory, they store the entire chain on flash. The entry in the hash table will point to the first entry of the chain. To get the next entry in the chain, some additional metadata needs to be stored on flash. Therefore on flash we will also write a pointer to the next entry in the chain along with each key-value pair. Such a design can however

lead to even more read amplification. It can also lead to read imbalance, especially when the chain is long. Long chains cause read imbalance because they can require significantly more reads than short chains. Lim et al. report an average of 5 lookups for each key-value pair [90] (increasing Q2). To mitigate such issues it does try to keep parts of the chain sequentially stored on flash to minimise the number of needed reads. It also adds a bloom filter as an AMQ for each chain, needing fewer reads for each chain on average.

BloomStore [93] takes it yet another step further, by completely moving the hash index structure to flash along with the key-value pairs. To achieve this, it uses a bloom filter to great effect. BloomStore keeps another bloomfilter based index in memory that can point to the hash indexes.

uDepot [77] tries a different route. It drops the idea of using just one hash table and uses a two-level hopscotch hash table instead. Hopscotch is a hashing method [61], that tries to move colliding keys into a *neighbourhood* close to the key, which has caching benefits among others. The key of a key-value pair is used for both hash tables. A part of the key is then used to access the location of a *directory*, the first level hash table. The other part of the key is used to access the index in this directory to retrieve an offset, the second level hash table. We thus have directories of offsets. Such a multi-structure design does incur more reads and more writes for each entry, but it does keep both memory overhead and read amplification relatively low. It is essentially a trade-off.

Small Index Large Table (SILT) [90] also uses multiple data structures. In fact it uses a total of three and they are not all hash tables, which we will explain soon. SILT also keeps a low memory footprint but does not result in large unpredictable read amplification like some of the previous designs. Lim et al. recognised that other data structures might be beneficial as well, instead of only using a hash design [90]. Different data structures have different advantages and disadvantages, combining multiple data structures can therefore combine the best features of such structures. At the same time, it comes with its own set of challenges. Such as efficient translations between data structures.

This idea lead to SILT, which used a *multi-stage design* (MS) in the end, which is essentially a series of data structures that are chained in sequence like a funnel. An LSM-tree, which we will cover in Section 5.4 is also an example of a multi-stage design. Each successive data structure in SILT will keep fewer data in memory, but more data on flash. Such a design makes the average size of each key-vale pair in the indexing structure still small but also keeps performance up.

SILT uses in order a *LogStore*, a *HashStore* and a *SortedStore*, visible in Figure 2. The LogStore uses a hash table design that is close to the design we already discussed earlier. HashStores are simply LogStores stored to disk, indexes included. This requires minimal translation. SILT allows multiple HashStores on flash to enable some buffer space. The last store, the SortedStore stores all of its keys in order. There can only be one SortedStore present at the same time. SortedStore also adds compression, which allows SILT to significantly reduce space amplification (Q3). Entries slowly transition from structure to structure and each structure takes a little longer to read and write. This does incur write amplification, which is a major drawback of multi-stage designs. Further on, the transitions from HashStore to SortedStore can be quite expensive, which can cause some latency issues.

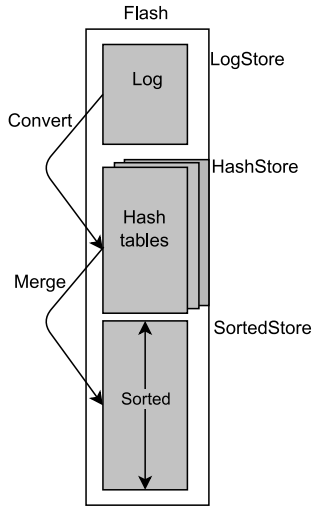

Figure 2: The multitier design of SILT. Reworked figure of the one made by Lim et al. [90]

In short, we have seen various hash-based designs and each has its own advantages and disadvantages. There is a trade-off to be made between the amount of the indexing structure that is maintained in memory and on flash. The more stored on flash, the bigger the read amplification and read latency (trading Q2 for Q5). At the same time multi-staged designs could be employed at the cost of more write amplification and wear levelling, but a possibly lower memory occupation and acceptable read amplification.

5.3 B-trees

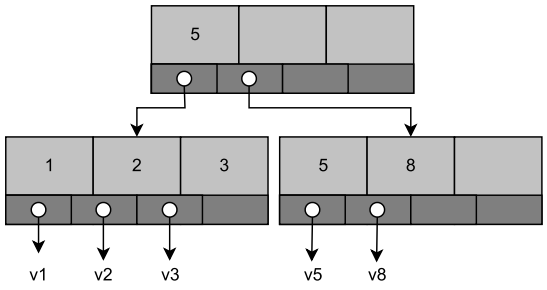

Figure 3: B+-tree with three nodes. Values are only stored in the leaves.

The *B-tree* is a tree data structure that is used widely in var-

ious databases. Especially in the field of traditional relational databases [31, 56, 79] and read-heavy key-value stores [104]. They are optimised for read performance and were used frequently in the early days of flash-based key-value stores, leading to among others *FlashDB* and *TokuDB* [103, 131]. However, they are still used in some modern designs such as *WiredTiger*, *ForestDB* or *Tucana* [1, 104, 107]. They are also used at times as substructures in for example LSM-trees [108]. They are known to have a good read performance because read operations have an upper bound in regards to the height of the tree, requiring only one read for each level in the tree. B-trees can thus be beneficial if a key-value store is read-heavy.

B-trees have a long history and have at least already been deployed to flash storage systems as least as early as 2003 [138] and were already considered ubiquitous in 1979 [31]. When a key-value store uses a B-tree as its main key-value pair indexing data structure, it typically actually refers to the *B+-tree*, which is also called a *B*-tree* [31]. B+-trees continue on the original design of B-trees and such a design can be seen in Figure 3.

Both the B-tree and the B+-tree are M-ary trees and both are designed to be self-balancing. This is accomplished by automatically merging and splitting nodes on insertions and deletions. All of the leaves are on the same height. This ensures upper bounds on the required number of reads to be equal to the height of the tree.

What differs between the two is what is stored in nodes and in leaves. In the B-tree design nodes and leaves both contain keys and values and keys are not duplicated. In B+-trees on the other hand, nodes can only contain keys in the form of pivots. Keys can be duplicated all the way from root to the leaf, also visible in Figure 3 for key *5*. Leaves store the actual value pointers. In addition, the leaves can be linked. This linking has a major advantage because sequential access can be improved. Scan operations for example can directly go on to the next linked leave and do not require traversing the entire tree again for each subsequent read. Yet another advantage of the B+-tree is that values are only stored in leaves and not on a random level, adding search efficiency. The key-value pairs are also sorted in the data structure, allowing the client to only read one node on each level of the tree. The earlier stated read performance is thanks to this sorted and sequential characteristic.

B+-trees store each node and leave separately on different blocks on the flash device. There is no guarantee that nodes and leaves will fill entire blocks, which can waste space and cause write amplification (Q1) as the entire block still needs to be written [80]. This can be reduced with compression as we will see in Section 8.4. Data is also not stored sequentially because inserts and updates change parts of the B-tree rather than appending their changes to the end. Reading, inserting and updates thus incur random I/O as data needs to be written to random locations. Random I/O causes a performance overhead, but not just because of the storage medium. We will show why this causes write amplification by looking at inserts. Parts of index data structures are typically cached, for example for B-trees a few blocks and leaves are cached. On an insert, data needs to be written to leaf blocks. If this block is in the cache, the write can simply be done in the cache. If the block is not in the cache, the block must first be loaded from I/O before being altered. The old block that was present in the cache is then generally written back to I/O as there is only a limited amount of memory. It is stated by Kuszmaul et al. that in the worst case, each insertion requires rewriting an entire block [80]. If the data is stored sequentially, multiple inserts can be written in one go. Lastly, B+-trees do not come with a lot of buffering by default, making it harder to amortise such operations.

There also exist various other B+-tree designs, which are tested on both flash and key-value stores, but we will only look at two B-tree designs: modified *HB+-tries* and *B$^\varepsilon$-trees*. However, first we will look at problems that occur when B-trees are directly stored on flash storage.

5.3.1 Implementing a B-tree directly on flash

It is possible to directly build B-trees on flash devices, without an extra layer between the B-tree and the storage. This could for example be an FTL (see Section 4) that uses a B-tree, but some devices, which we will discuss in more detail in Section 6.3, can also directly allow the key-value store to run on flash. Directly building a B-tree on flash comes with a few challenges, challenges which we do not have when there is some logic between the key-value store and the storage. We will explain these problems shortly.

B-trees store each node and leave separately on different blocks on the flash device. On an update this entire block needs to be rewritten as flash itself does not allow in-place updates (this is traditionally resolved with the help of an FTL). This inevitably brings us to a major crux of the B-tree when it is used directly on flash: if any node changes, its location changes. Therefore the parent node also needs to be updated because its pointers are no longer accurate, which recurses all the way back to the root. This is commonly referred to as the *wandering tree* [68]. Operations on B-trees can therefore cause a non-negligibly high write amplification (Q1).

A way to reduce the wandering tree problem is by packing multiple nodes and leaves into one page, allowing the cost of traversing writes to be reduced. Such as is done in the *μ-tree* [68], which packs nodes and leaves together in a manner optimised for flash. This has not been tested on key-value stores as far as we know and we have not seen a key-value store that itself directly uses a B-tree on flash storage. Nevertheless, it is still interesting as the underlying FTL might use this functionality, which has implications for the key-value store on top. Similarly, it leaves options open for future key-value stores that aim to build a B-tree directly on

flash storage.

5.3.2 HB+-trie

HB+-tries were originally implemented in ForestDB [1]. It uses a *trie*, which is a kind of data structure, where each node of the trie is a B+-tree. Leaves point either to another B+-tree node or to values. For more information on tries, we recommend looking at the ForestDB paper [1].

The HB+-trie comes with two optimisations that help with increasing the performance for flash. Updates to data instead of the data itself are stored in an append-only log, which avoids in-place updates. This helps flash because it reduces the need for random I/O. It also uses a small write buffer index. This index buffers references the log and is only flushed to the HB+-trie after a certain number of writes. This helps with amortising expensive I/O operations (addressing Q1).

Unfortunately, the append-only log requires regular cleaning, which necessitates a separate garbage collection process known as *compaction*. This requires copying old data over to new locations. This introduces more write amplification and temporarily reduces the latency and throughput for the client operations during the compaction. Lee et al. propose to make these compactions more optimised for SSDs by properly using the available parallelism [84]. It uses a parallel-fetch, which they call *p-fetch*. It submits multiple parallel reads, which can be linked to different flash channels. To make sure that the host does not have to wait for all operations, it makes use of asynchronous I/O (addressing Q6), which we will cover further in Section 7.3.1. This reduces the software overhead and the time compactions would take. We conclude by stating that such an approach could also be reused in other log designs.

5.3.3 B^ε-tree

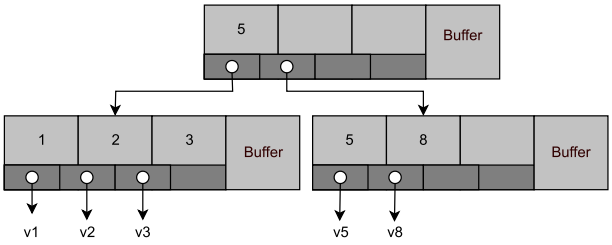

Figure 4: Example of the standard B^ε-tree. Each node also has some buffer space.

The *B^ε-tree* adds buffers to all intermediate nodes of the B+-tree. If we were to adapt our original B+-tree design, visible in Figure 3, with buffers it would look like Figure 4. These buffers can be used to store operations such as writes for new data or updates for older data and allow amortising expensive I/O operations (addressing Q1). When a buffer is filled, it *spills* its data and moves the data downwards until it reaches the leaves, which have no buffer. B^ε-trees allow for a write-heavy workload but do not suffer from compaction operations such as LSM-trees. At the same time, it does require more random I/O. Such a design is therefore only effective on devices that provide fast random I/O, such as NVMe SSDs (note that not every SSD is equal). It is also most effective when the entire index structure fits in memory (conflicting with Q5), which can be big. Such a design, albeit modified and further optimised, was implemented in *Tucana* by Papagiannis et al. [107] and is shown to be able to reduce the software overhead and increase throughput. Similar to WiscKey, which we will cover later, it is possible to separate the index structure from the values. This allows the index to be smaller and reduces the write amplification.

A remaining problem with B^ε-tree is that it can cause significant write amplification (Q1) in spite of the buffering. That is because every time a buffer needs to be altered the entire buffer needs to be rewritten. Conway et al. therefore proposed yet another data structure, The *STBe-tree* [32]. This covers ideas from both the LSM-tree and the B^ε-tree. Instead of using direct buffers in nodes, it uses multiple sub-indexes in the form of B-trees, replacing the log buffers of the B^ε-tree. Each of these B-trees can be updated independently, reducing write amplification. Similarly to the main B-tree, each sub B-tree should also come with an AMQ, such as a bloom filter, to reduce the number of reads necessary for these trees (addressing Q2).

Another novel idea proposed for the STBe-tree is *flush-then-compact* [32]. Flush-then-compact enables more I/O and CPU parallelism. This reduces Q6 and the immediate effects of Q1. Traditionally data is moved from parent to child with a process known as *flushing*. Think about data moving from node to node. Data in the target node then needs to be reorganised, which is known as a *compaction*. This value copying requires locks to be held while all values move and temporarily introduces expensive I/O operations that need to be completed in one go. With flush-then-compact, a *pointer-swing* is performed instead. References to the child's active branches are then copied from parent to child. This allows locks to be held very shortly, enabling better concurrency. Operations to copy the values can then be scheduled later on asynchronously without requiring locks. This also allows scheduling them in parallel trivially. Such a design, could in general also be used in other designs that make use of such a compaction operation.

5.4 LSM-trees

On flash storage writes are more expensive than reads, which makes it attractive to look for a data structure that is optimised for writes instead of reads. These types of data structures are commonly referred to as *Write Optimised Indexes* (WOI). An example of such a data structure is the *log-structured merge-tree* (LSM) [106]. A big part of the modern flash-optimised

key-value stores make use of LSM-trees [44, 48, 81, 94, 116]. Nevertheless, just putting a plain LSM-tree on flash storage does not immediately make it optimised. There are a few optimisations that can be done to increase its performance, most with trade-offs. A few of the more novel solutions will be discussed in more detail. We will first discuss the general LSM-tree structure, followed by various optimisations and trade-offs.

5.4.1 What is an LSM-tree

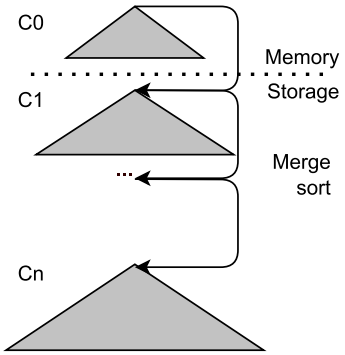

Figure 5: Original LSM-tree design in 1996, based on a figure by Lim et al. [106]

The LSM-tree was originally intended to be used for HDDs, all the way back in 1996 [106]. Back then, HDDs were commonplace and DRAM was becoming big enough to allow for buffering. Many of the properties of the LSM-tree were thus optimised for the characteristics of this system. Such a design is seen in Figure 5. This design starts with a small part of data in memory and then uses multiple layers on disk. Each incremental layer is bigger and merge sorts into the next when it becomes too big. This relatively simple model has since been heavily optimised/modified, and it it is this modified design that we actually see back in modern literature for flash key-value stores. This design is visible in Figure 6 and was popularised by Ghemawat et al. in LevelDB [54]. The data structure consists of four substructures, chained together in a *multi-stage* (MS) structure. A *sorted in-memory table*, a *sorted immutable in-memory table*, a *write-ahead log* (WAL) on storage and multiple *sorted string tables* (SSTables) that are stored on storage. We will take a look at how data propagates through the structure by addressing all steps in Figure 6 by numerical order.

(1): On an insert or update, every key-value pair is first inserted into the WAL, which is an append-only log. This functions similar to journaling in traditional databases and guarantees persistence and consistency [120]. On recovery, the WAL can then be replayed as it contains all changes in order.

(2): After the write to the WAL is complete, the same data is written again, but then to the in-memory table, which is

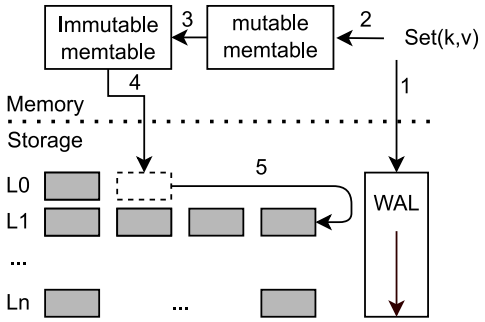

Figure 6: Common LSM-tree design, originally popularised by LevelDB. The numbers represent the steps in order of a operation that updates/inserts/deletes a pair. It is based on a figure made by Lu et al. [94]

typically small such as a few MBs [8]. The in-memory table functions as a buffer that can be used to avoid small writes to disk (reducing Q1). This is similar to buffering in hash tables and B^{ε}-trees, see Section 5.2 and Section 5.3.3.

(3): When the in-memory table reaches a certain size, it is converted to an immutable in-memory table. These immutable tables should eventually be written to storage. At the same time, a new mutable memtable is created or an old memtable is reused. The immutable in-memory tables are used to allow writes to continue to memory without a prolonged wait. Otherwise, the key-value store has to wait for the entire I/O operation to complete. New data is written to the mutable memtable, while the old immutable table is written to storage.

(4): The immutable in-memory table is written to storage as is. It is written as a sorted file, known as a Sorted String Table (SSTable). Such an operation is known as a *flush*. An SSTable is sorted on keys and contains a subset of the total range of keys. The SSTable itself is thus essentially an immutable ordered collection of key-value pairs, sorted on keys. The SSTables are stored on storage in a series of levels. Reaching from level 0, commonly referred to as *L0*, up to level N, commonly referred to as *LN*. The number of levels depends on the design and can be tweaked. Each level can only hold a certain amount of SSTables. This amount increases with each level, so higher levels can hold more data. LevelDB for example maintains a growth factor between levels of 10x by default [54]. For correctness, only 2 levels would be enough, but adding more levels helps with amortising step (5).

(5): Data can be moved to a higher level, with a process known as *compaction*. Compaction moves a number of SStables from one level to the next level. This is done with an *n-way merge* on the SSTables in the next level. In this context n-way merge approximately refers to merge sorting on multiple SSTables in the next level. Since each SSTable is limited to a certain size, this can also introduce new SSTables and in the worst case create yet another compaction for the next level. This process thus leads to cascaded write amplification

as writes have to be done for each level [128], instead of writing them just once in total. This write amplification can reach numbers as least as high as 50x in LevelDB [8]. More than 2 levels help with amortising compactions because updates move down multiple levels, not forcing an n-way merge with all data on every compaction.

Because of compaction, we can be sure that each level contains non-overlapping SSTables, except for L0, the first level. On L0, tables are simply appended. So whenever a read operation is performed, this can lead to one read in the mutable memtable, one in the immutable memtable, one for each SSTable in L0 and one for each additional level. Read operations thus have significant read amplification (Q2). This is often mitigated with the usage of AMQs such as bloomfilters.

LSM-trees are considered favourable for flash because of their append-reliant implementation and their buffering capabilities. Erase and write operations are expensive on flash and can be reduced with the usage of LSM-trees. It also reduces problems that occur with the usage of random I/O on flash as it favours sequential I/O. Some problems that can occur when random I/O are used, are expensive erasures and garbage collection. In addition, LSM-trees limit small writes with the help of buffering in DRAM, increasing throughput and lowering latency and write amplification.

At the same time, they also come with various disadvantages. LSM-trees have for example significant write and read amplification because of their levelled design (causing Q1 and Q2 issues). They also suffer from the expensive compaction procedure, which is expensive because it reduces latency of other operations, reduces available throughput and has a high software overhead.

5.4.2 Separating keys and values

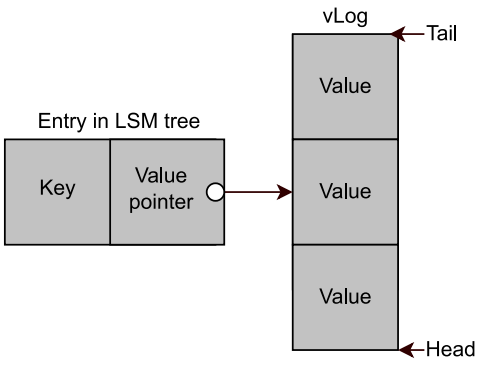

Figure 7: Separation of keys and values as used in WiscKey.

Traditional LSM-trees batch their keys and values together sequentially. Whenever a key-value pair needs to be read or written, the entire pair is read or written. This occurs not only in client operations such as update and delete but also in background operations such as compactions. This is beneficial for devices that suffer from random I/O as data stays close together, but flash is known to suffer less from this problem and suffers more from the alternative; the alternative being a higher write amplification (Q1).

Lu et al. therefore proposed *WiscKey* [94]. WiscKey separates keys and values, which prevents rewriting the values unnecessarily. Values are instead stored in a separate data structure. The LSM-tree itself is then limited to keys and data pointers only. This also has the added side effect that the tree itself becomes smaller and is thus both easier to cache and requires fewer operations to read. This design is shown to both reduce write and read amplification and increase throughput. The idea is also already implemented in various other key-value stores [24, 44, 47, 49, 88, 140]. Nevertheless, it also results in new problems and is not a silver bullet. Most problems have to do with the separate data structure that is used to store the values.

The separate data structure that values are stored in, can in theory be anything. However, a common design is to use a circular log, called a *vLog*, which is also used in WiscKey [94]. The vLog fits naturally on flash storage because of its append-only structure. An additional benefit is that in theory when a log is used, this log can also function as part of the WAL [47].

Nevertheless, this structure creates problems for scanning operations, because the data is not sorted (increasing effects of Q2). This is in various designs resolved with the usage of prefetching further scan iterations beforehand [94], but this does not remove the actual problem. Therefore, Li et al. propose to store a part of the values in another LSM-like tree [88], known as a *vTree*, that is partially sorted to get better scan performance. However, the vTree would need additional management and is still not completely sorted.

Using a vLog also causes other problems. Various works namely reported that performance can degrade when the store is update and/or delete heavy and a vLog is used at the same time. That is because updates and delete operations result in garbage collection in the vLog. The overhead of this garbage collection increases, when the values stored are small. We expect similar problems when a different data structure is used. A solution that is data structure agnostic, differentiates between small, medium and large values [24, 88, 140]. Large values are stored separately and small values are stored in place as in the traditional approach.

The garbage collection procedure remains a major performance bottleneck, even when differentiating between the value sizes. Therefore Lu. et al, proposed a more garbage collection friendly data structure [24]. They proposed mapping values into partitions by hashing the keys. So instead of using one big log, they make use of multiple partitions. This adds isolation of hot and cold data and determinism of where key-value pairs are stored. The same key is hashed to the same partition next time. Separating hot and cold data and determinism make it easier for a garbage collector (Q4) to move fewer data and thus reduces write amplification (Q1).

To conclude, key-value separation is a common optimisa-

tion that allows reducing write amplification (Q1) and better caching of the LSM-tree, but does come with additional garbage collection and can cause lower scan performance.

5.4.3 Reducing compaction overhead

Compactions are known to be one of the main bottlenecks of LSM stores. They incur many writes and reads in a short matter of time and can cause space amplification during the merge (Q1, Q3 issues). Chan et al. for example mention write amplifications at least as high as 50x during compactions [24]. In addition compactions consume CPU resources (Q6 issues) [20]. Since compactions generally happen at intervals, this causes spikes in latency and makes stable performance problematic. Therefore, various designs opt to postpone compactions or reduce the impact of compactions.

Tiering and levelling: There exist two major LSM models. These models are referred to as *levelled* and *tiered* designs [92] or alternatively as LSM-tree and *LSM-forest* designs [100]. Levelled designs follow the standard model we defined earlier, where each level above L0 contains non-overlapping sorted SSTables. However, there also exist various LSM-trees that are less strict on this rule. These go by multiple names. We will simply refer to them as LSM forests. They are known as forests because they can contain multiple logical LSM-trees on each level instead of just one. This can occur when during compaction, the trees are not fully merged, but only partially or not at all. Raju et al. noticed that in a write-dominated setting it is more important to reduce the write amplification, than it is to reduce read amplification [116]. Forest designs trade read amplification for lower write amplification, addressing Raju's concern. They thus trade Q2 for Q1.

The common idea is to postpone compaction all the way up to the last level. This increases read amplification and space amplification and reduces the scan performance; since this would require reading all SSTables on each level. Further on, it can also cause a more expensive compaction on the long run because there is one big compaction at the end. On the other hand, it does reduce the write amplification significantly in many cases. PebblesDB for example reduced write amplification by 2.3-3x compared to RocksDB [116]. Mei et al. show similar results for PebblesDB for sequential workloads and favourable results for two other LSM-forest designs, SifrDB and size-tiered Cassandra [100]. Mei et al. do note that *partitioned forest* designs such as PebblesDB and size-tiered Cassandra have higher write amplification for sequential workloads and will thus not achieve the goal of reducing write amplification for such workloads. We will get to the definition of partitioned forest, but we will first look at some additional forest designs.

To mitigate the issues that are related to reads with forest designs, PebblesDB [116] introduced the *fragmented LSM-tree* (FLSM-tree), visible in Figure 8(b). This design adds a new data structure to LSM-trees known as *guards*. Each level contains guards in addition to the trees. The guards themselves have a key range similar to SSTables, but are strictly non-overlapping, decreasing the number of reads necessary on each level and essentially creating a layered index design. To explain why it is a layered design, we explain what happens on a read. On a read, first the guards are investigated rather than the SSTables. The guard that matches the key-value pair of the guard will be picked and all SSTables of that guard will need to be read. Thus two different types of ranges are in sequence. On a compaction, the SSTables are moved to their corresponding guards on the next level. In Figure 8(b) we see such an operation with the SSTable moving to L1 and directly into the guard having the range $7 \sim \infty$. The FLSM design is still suitable for read-heavy situations unlike the standard forest design because of the usage of guards.

FLSM is known as a *partitioned LSM-forest* design. This means that trees on each LSM-tree level can overlap. For example, SSTables can move to already existing guards at a higher level on compaction. On a compaction, SSTables can also be split among guards. This can incur overhead when only a small part of an SSTable falls into a guard and the rest in another, which increases I/O amplification (in the form of Q1). It also increases write amplification on completely sequential workloads. That is because on completely sequential workloads, no merges are necessary on compactions. After all, each flushed SSTable has a new range. In that case, compaction is cheaper than creating new trees. Alternatively, a *split LSM-forest* design can be used. This has non-overlapping trees on each level. Such an idea is used in *SifrDB*, proposed by Mei et al. [100] and visible in Figure 8(a). This design aims to solve many of the earlier mentioned problems. The trees are stored separately and each tree itself is properly sorted. This allows the design to read trees in parallel, which can increase read performance and properly use the internal flash parallelism of SSDs. It also allows the store to already remove parts of the data during a merge, reducing space amplification. This removal is not possible with a standard tiered design or a levelled design. Lastly, split forests do not have problems with increased write amplification on sequential workloads. The decision to use either a tree or forest design is hard to make, also when only considering flash. In general, if the goal of a key-value store is to either process a lot of writes or reduce wear levelling caused by compaction, it might be beneficial to look at forest designs.

In memory compactions: It is also possible to leverage memory capabilities to do compactions, reducing the need for expensive I/O. Such an idea is proposed by Bortnikov et al [20]. Their design, *Accordion*, uses a pipeline of segments in memory instead of a single in-memory table. There is always one segment mutable and it is converted to an immutable segment once a certain size is reached. An immutable segment is implemented as a different data structure that has a lower memory footprint, but sacrifices modification which it no

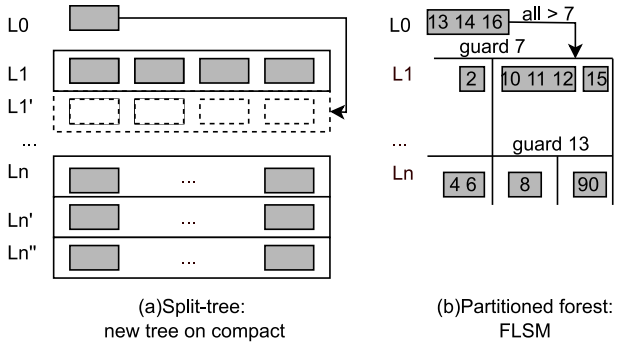

Figure 8: Multi-stage taxonomy and figure as given by Mei et al. [100]

longer needs. It thus uses only what is needed for immutable segments, good read performance. A lower footprint allows multiple immutable memtables to exist. Once a number of immutable segments exist, they are merged in memory and flushed to L0. This allows an earlier merge, already moving part of the problem that would occur in L0. It also reduces wear levelling as unnecessary writes are removed.

Delay compactions: Another idea is to delay compactions. Simply delaying compactions reduces the effect on the short run, but can have aggregating effects and can eventually cause a significant spike in long-tail latency [9, 23]. Instead delaying compactions should be based on evidence. For example, *TRIAD*, proposes to delay compactions until the overlap of SSTables in L0 is big enough [8]. This avoids writing a large number of duplicates on compaction and therefore reduces write amplification. It can, however, cause the read performance in L0 to decrease or can cause more expensive compaction later on. TRIAD also aims to reduce the impact of flushing by reusing the WAL. Instead of rewriting all of the data from the WAL, which is, in essence, a replay, it reuses the WAL on a flush. The data in the LSM-tree levels will then hold references to the data in the WAL, instead of copying the data itself. This works especially well for uniform workloads but can introduce some fragmentations. It can cause fragmentations because the WAL also contains data that is no longer valid, think for example about updates to data, which invalidates old parts of the WAL.

Scheduling compactions: The final idea we will cover is properly scheduling compactions. This idea does not aim to remove compactions, but to do compactions at the right time. It should be scheduled in such a manner that read and write operations receive as little hinder as possible from this background operation. For example, if compactions are performed too late, flushes have to wait before L0 is properly compacted to L1. On the other hand, if they are performed too early, compactions might not fully use the buffering capabilities available and compact duplicate data (hot entries that keep reoccurring).

Such a scheduling idea is proposed by Balmau et al. in the form of SILK [9]. It properly links the compaction operations to the internal flash parallelism. It always prioritises compactions on L0 because these compactions are most relevant for client operations and it allows preempting compactions. This means that for example a compaction on L2 can be stopped, to perform a compaction on L0 instead. Compactions also only get a part of all available flash bandwidth, allowing clients to always make some progress, even when multiple compactions happen at the same time. This idea is shown to stabilise the latency of the key-value store and shows no latency peaks like would happen without such a scheduling procedure.

5.4.4 Adding indexes to LSM-tree levels

Typically LSM-trees use sorted buffers on each level of the tree, organised in SSTables. It is also possible to use a more involved data structure, such as using index structures, of which we have already seen three examples. This radically changes the characteristics of the individual levels. It for example reworks the entire compaction logic needed between each level. If chosen adequately it can avoid many of the compaction problems that we discussed in Section 5.4.3.

Papagiannis et al. for example propose *Kreon* [108], which uses a B-tree on each level, accompanied with a completely different compaction operation (addressing Q1, Q2, Q4 and Q6). The design still uses an LSM-tree, but uses B-trees as indexes for each level, combining advantages of both data structures. Compactions for the B-tree indexes work as follows: instead of completely rewriting the next level on compaction, it only *spills* a part of a B-tree to the next LSM level. To explain spilling, we take a look at how compaction is done for LSM-trees. Compactions for LSM-trees read the entire next level (except for tiering) and merge sort the entire level into this next level. Spills read the current level in lexigraphic (sorted) order and inserts it into the B-tree of the next level. This prevents the reading and sorting of all of the data in the next level, only requiring parts of the tree to be updated. As a further optimisation, spilling can be done in batches. This is enabled by the lexicographic order and applies multiple changes to the B-tree before writing it back to storage, effectively amortising multiple spill operations. Such a design reduces CPU overhead but does introduce many random I/O requests. It performs many random I/O requests because of the nature of B-trees, see Section 5.3. So it only makes sense when CPU and I/O amplification are the main overhead. Similar to ordinary LSM-trees values can be stored separately, this is also done in Kreon [108]. This proves that despite switching to an index on each level, many other optimisations meant for standard LSM-trees can still be performed. Using indexes on each level leverage the idea that it is also a valid idea to add index logic to the individual levels of LSM-trees. Future work, could look at other indexing structures for the LSM-tree levels.

5.4.5 Reducing space amplification

Space amplification (Q3) is an important goal of many key-value stores. For some even more important than throughput and latency. Storage can be expensive and efficient usage of the space can keep the monetary costs down. For example, at RocksDB they shifted the focus from reducing writing amplification to reducing space amplification [48,49]. Dong et al. proposed a few techniques to achieve better space utilisation and reduce space amplification for LSM-trees.

In the multi-stage design of LSM-trees, the next level might be only marginally bigger than the previous level, which can waste space. Therefore an idea is to make use of *dynamic levelled compaction* [48,49]. This ensures that each level is at most $\frac{1}{x}$ of the next level. The "x" is configurable; the bigger the "x", the lower the space and read amplification, but the bigger the write amplification (trading Q1 for Q2 and Q3).

Another idea that is commonly used to keep space usage down is lossless compression. This can reduce the size of each file to only a fraction of its normal size. This does come with the disadvantage that data needs to be compressed on writes and decompressed on reads, which generally is done on the CPU, adding software overhead (Q6). Later on in Section 8.4 we will also discuss SSDs that have built-in compression support and therefore do not need to use the CPU of the host anymore for compression. Dong et al. proposed a *tiered compression* design meant for LSM-trees [48]. Tiered compression uses different compression on different levels. Top levels are smaller and accessed more frequently. Compression on these levels would have less effect on the space but might hamper throughput. Therefore top levels have no compression, mid levels have light compression and bottom levels have the highest compression. Lastly, cached pages should always remain uncompressed as they are frequently accessed.

The last idea we will cover is simply dropping some structures that might not be necessary in all cases. Dong et al. for example argue that bloom filters can be dropped on the last LSM-tree level [48]. Bloom filters can use quite a bit of space for each entry and the last level contains quite a few entries. At the same time, these are accessed infrequently. So in certain use cases, it can be viable to drop these filters. This idea can be generalised to all AMQs on the last level, so also for cuckoo filters or quotient filters.

A similar approach can be taken for the WAL, as it can also take a bit of space. Therefore, some designs assume a small bit of NVM, PCM or battery-packed DRAM is present [136,147], which can store the metadata instead of the SSD. This also causes the SSD to use no space for WAL at all. If no such hardware is available, it might also be possible to drop the WAL entirely under certain conditions [48]. The WAL can be dropped if there already exist other measures to keep the structure persistent and recoverable. Such as replication of the key-value store (duplicate data as a fail-safe) or databases on top of the key-value store that already provide persistency. Lastly in TRIAD data in the WAL can be reused in the SSTables [8], which prevents data duplication, but does not guarantee lower space amplification.

So in short, some common techniques we have seen for reducing space amplification on LSM-trees include: space-friendly compaction strategies, compression strategies and dropping unnecessary data structures.

6 Integrating with the flash interface

Problem	Solution	Examples
Q1,Q4,Q7	Multistream SSDs	Cassandra [62], RocksDB [142], MongoDB [104]
Q1,Q2,Q4, Q5,Q6,Q7	Rely on FTL capabilities	NVMKV [99]
Q1,Q2,Q4, Q7	Open-channel SSDs	LOCS [136], NoFTL-KV [135], FlashKV [147]
Q1,Q2,Q4, Q7	ZNS SSDs	LSM GC [29], ZenFS [18]
Q1,Q2,Q4, Q5,Q6,Q7	Near data processing	Co-KV [129]

Table 4: Overview of papers covered in Section 6.

Key-value stores are commonly stored on a file system. This file system itself then traditionally added on top of another layer, known as the *Flash Translation Layer* (FTL), which is managed on the flash device itself. For more information about the FTL, see Section 4. This leads to a layered design, which is visualised in Figure 9. This design is essentially a double-edged sword. It hides implementation details and allows both the key-value store and the file system to function with a well defined interface. However, at the same time there is a *semantic gap* between all of the layers. For example it is not always possible to communicate with the underlying layer what the difference is between hot and cold data. This in turn leads to duplicate work and leads to *auxiliary write amplification* (AWA) in the device, making optimisations non-trivial.

Therefore there has been both an academic and an industrial push to reduce the amount of layers needed and to better integrate the already existing layers. These ideas will be discussed in this section. It is important to note that all of these ideas are tied to certain devices and can therefore not be used in all cases, which is one of their main weaknesses.

6.1 Multi-stream SSD

Multi-stream SSDs are SSDs that allow the host to mark write commands with a stream hint. Normal SSDs have a single

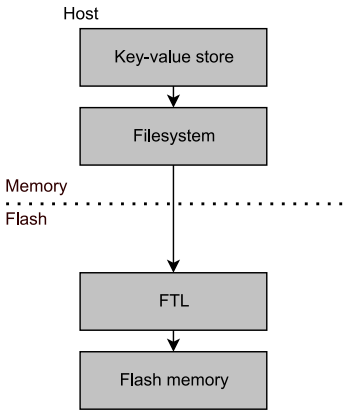

Figure 9: The traditional layered approach. Some layers have been abstracted away such as the block layer commonly used in operation systems such as GNU/Linux.

append point, also known as a *stream*, where all new data is stored. Multi-stream SSDs allow writing to multiple different streams [16, 69]. These streams can then be handled differently by the underlying FTL. This allows differentiating data and can thus help with separating hot and cold data, which in turn helps in bridging the aforementioned semantic gap (solving issues with Q1, Q4). It focuses on improving the communication between layers but does not remove any layers of itself (this communication helps with Q7). The main advantages of this improved communication will be a more efficient garbage collection and a reduced AWA.

The technology is promising, but not a lot of key-value stores are optimised specifically for such devices. Nevertheless, it has been tested on a few already existing key-value stores. For example on the wide-column store *Cassandra*. For more information on Cassandra, we recommend the paper written by Lakshman et al. [81] on the data store. Cassandra uses an LSM-tree and can use multi-stream SSDs to give different *stream ids*, used to differentiate between hot and cold streams, to different parts of the store [62, 69]. For example, the WAL can be given to one stream and bloom filters, cache and metadata to another. Lastly, as some readers might already expect they also used different streams for different LSM-tree levels. This design was shown to significantly reduce write amplification and make garbage collection more efficient. Such ideas were also at some point in time tested on RocksDB; using similar approaches as Cassandra [49, 142].

Multi-stream SSDs are also in use for B-tree designs such as WiredTiger in *MongoDB* [104]. MongoDB makes use of files to store data. They note that mapping different files to different streams is inadequate for their use case, as it does not address internal fragmentation of data. Hence they proposed an alternative, *boundary-based stream mapping*, where parts of individual files can be sent to different streams, allowing parts of files to be differentiated. Parts of a file can for instance be hot or cold.

We conclude by stating that this technology is not bound to any data structure and could also have benefits for other data structures such as hash tables. It is, for now, unsure if this technology will become bigger for key-value stores.

6.2 Native FTL capabilities

There exist various SSDs with FTLs that already provide various transactional operations and persistency guarantees. Instead of adding layers on top of these FTLs with similar functionality, it can be advantageous to make direct use of the FTL. This does not only avoid duplicate work, but it also leaves optimisations to the SSD itself (Q7). Not every SSD is made the same, so it does have advantages to leave optimisations up to the SSD vendor (this can help with Q1, Q2, Q4, Q5, Q6).

This idea was pursued in *NVMKV* [99]. This design only works for devices that have implemented certain capabilities such as *atomic multi-block writes*, *persistent trimming* operations, *exist* operations and *iterate* operations. NVMKV then maps all key-value operations such as get, put and delete to these FTL operations. Guarantees such as persistence and recoverability can already be encapsulated by existing FTL operations. Keys are in such designs mapped to hashes which are interpreted as logical addresses by the device, which the FTL then maps to the actual physical location. The device thus has complete control over where a hash is stored. Hash collisions can be resolved with operations such as linear probing, but are in general let up to the design of the store and are not relevant for this survey. Atomic multi-block writes can be used to write one file atomically, persistent trim operations can be used to make deletes persistent, iterate operations can be used for scan operations, et cetera. This makes the design of the key-value store more simplistic. The key-value store essentially becomes a thin layer on top of the FTL, or in other words, the FTL is also a kind of key-value store. Such a design can be seen in Figure 10.

Advantages of a thin layer on top of the FTL are a lower CPU usage and a lower DRAM footprint on the host (resolves Q5 and Q5). It also reduces AWA (Q1) since the device is now directly responsible for the application. Lastly, it reduces the need for multiple levels of key-value management, which reduces the complexity.

NVMKV also addresses another issue, that is absent in most other key-value stores that we cover: *Multi-tenancy*, the idea of running multiple applications on the same device. Multi-tenancy is favourable as it removes the need to use different storage devices for each application. It is also possible for multiple key-value stores to run on the same machines, which already happens in designs such as FAWN [4]. These other key-value stores also make use of the available flash channels, the DRAM and the cache of the device and the storage of the device. These key-value stores can therefore hamper each other's performance and interfere with the

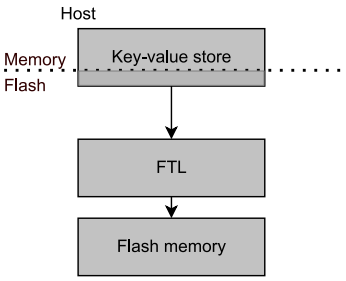

Figure 10: The layered approach when using native FTL capabilities. Part of the key-value store is managed by the underlying FTL.

scheduling of operations. NVMKV aims to solve such issues with the use of pools. FTLs can provide a pool abstraction that allows binding operations to a group, a different group can be used for each application. KV pairs of different pools will then be spread evenly across the address space of the device.

6.3 Open-channel SSD

Open-channel SSDs (OCSSD) are SSDs that allow developers to directly control the SSD [19]. They do not come with FTL interfaces and instead entirely rely on the host device managing the SSD; minus a few device specific functionalities such as bad block management. This means that applications, such as key-value stores, themselves can assert direct control over page mappings and garbage collection (solving Q7 and reducing Q1, Q2, Q4). This leads to the lean stack visible in Figure 11, provided that the file system is dropped as well. The Linux kernel allows interfacing with such an SSD through *LightNVM* [19]. We will cover a few designs that make use of such devices.

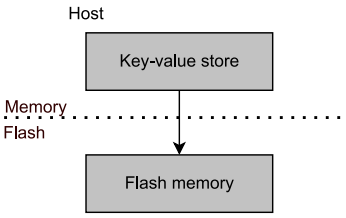

Figure 11: An example of a stack on an open-channel SSD. It directly interfaces with the memory.

A key-value store that popularised the idea of running key-value stores on OCSSDs is a design known as *LSM-tree based Key-Value Store on Open-Channel SSD* (LOCS) [136]. This store has been implemented on *Software Defined Flash* (SDF) [105], which is a type of SSD with open-channel like capabilities made by Baidu. SDF, unlike ordinary SSDs, exposes each channel as a different device to the host. This should make it possible to directly map operations to internal flash channels (see Section 4.2) and to schedule the individual operations.

LOCS internally uses an LSM-tree, see Section 5.4, and proposes various scheduling techniques that more adequately map LSM-tree operations to the individual device channels. Wang et al. discovered that it is beneficial to take the type of action into account when doing such scheduling [136]. They propose assigning a different cost to reads, writes and erasures and schedule each of these operations accordingly. Scheduling techniques could then include simple scheduling techniques such as Round Robin Scheduling [117], but also more involved ones that try to keep the queue size of each channel to a minimum, with queue size based on the sum of all assigned operations and their costs.

FlashKV [147] tries some completely different scheduling techniques but still uses an LSM-tree. It does not directly assign a cost to actions, but instead prioritises actions over other actions. It for example prioritises read operations over-write operations as these take less time. It also chooses to give erasures the lowest priority, until there is not enough space left. This is done because of the non-critical nature of the erasure operation when there is enough space available and the high cost of said operation. It also includes separate scheduling techniques for compaction, based on the workload. When the workload is write-heavy it favours compactions. Similar to other LSM-tree scheduling techniques such as proposed in SILK [9] (covered in Section 5.4.3), L0 compactions are considered to be the most important compactions and are prioritised.

FlashKV [147], LOCS [136] and SILK [9] all come with different scheduling ideas and yet all achieve significant performance gains. This raises the concern that scheduling operations and compactions are important for getting adequate throughput and latency. Key lessons that could be learned from these designs revolve around the principles of scheduling each operation differently, differentiating between client and background operations and how operations can be scheduled across channels to achieve better throughput.

Another idea that is interesting to try out is how files and data are spread across the different channels of the device. For example, writing a file to just one channel or to stripe the file's blocks across multiple channels at the same time. Such an idea was proposed by Zhang et al. [147] and can improve the read and write performance of individual files significantly. That is because each channel can read a part of a file in parallel on a read request. However, such a design does come with its own metadata and management. LOCS also showcased that it is profitable to store key ranges that are present on multiple LSM-tree levels on different channels. For example by storing a particular range on channel 0 for L0 and channel 1 for L1. Storing on different channels has benefits because this allows parallel reads on multiple LSM-tree levels [136]. This avoids having to do all reads on one channel for a read operation.

Vinçon et al. focus on an entirely different optimisation with OCSSDs. They state that with such devices it is possible to separate hot and cold data (addresses Q4). For example by sending different levels of an LSM-tree to different channels [135]. This is possible because all data can be managed from the side of the host, which allows the host to make decisions ranging from where data is stored to what data is stored. This could allow more efficient garbage collection or access patterns.

In short, we have seen various optimisation techniques for OCSSDs. Key-value stores can be made more efficient on such devices by considering how both operations and data are spread among the individual flash channels and scheduling operations accordingly.

6.4 ZNS SSD

Another device interface is the *Zoned Namespace* (ZNS) SSD [17]. ZNS SSD builds further on the *NVMe* specification with the *NVMe ZNS* specification and is a successor of open-channel SSDs. Such specifications allow standardising approaches and make optimisations transferable between devices. The main idea of the ZNS specification is to divide the capacity of the devices into distinct zones. Each zone can be read in any order, but can only be written sequentially. If a zone needs to be changed, it needs to be erased first. This specification closely follows the characteristics of flash and can thus lead to better internal placement (aiding Q4), higher write throughput, lower QoS and higher capacity. Unlike standard block interfaces and similar to open-channel SSDs, most logic can also be moved to the host (addressing Q7). Such as a *host-managed FTL* (HFTL). Optimisations discussed in Section 6.3 can thus also be applied to ZNS devices, with some translations to make use of zones instead.

Since the host can control most logic and it has an append-friendly structure, it can be beneficial to also store key-value stores on ZNS SSDs. Especially append-heavy structures such as LSM-trees are promising (potentially aiding with Q1, Q2 and Q4). As a matter of fact an LSM-tree has already been ported to ZNS, but then for a garbage collection scheme and not for a key-value store [29]. ZNS is also considered promising because ZNS is not tied to a certain device and should in principle be able to run on all types of SSDs and other hardware such as HM-SMR HDDs, provided that they implement the interface.

A key-value store that runs on a ZNS SSD was proposed by Bjørling et al. [18] by letting RocksDB use *ZenFS*, a ZNS-friendly file system. This increased the throughput and lowered the write amplification in comparison to other block-interface file systems such as *F2FS* and *XFS*; mainly because the effect of garbage collection was significantly less.

This proofs that ZNS is a viable candidate; not just for file systems, but also for key-value stores. Further research could look into the merit of optimising a key-value store for such a device without the help of any file system, similar to the designs we saw in Section 6.3. This could lead to a stack as seen in Figure 12.

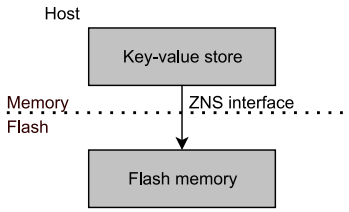

Figure 12: A potential design that could be used for ZNS devices.

6.5 Near data processing

Near data processing is the practice of moving the computation close to the data. SSDs could for example come with their own computation units such as ARM chips or FPGAs. Such devices are also referred to as computational storage and various other names [95]. It allows the logic to be almost completely moved to the device with the benefits of having the computation close to the data and a lower lookup time (answers Q7 and can reduce Q1, Q2 and Q4). A beneficial side effect is not having to rely too much on the computing power of the host (indirectly solving Q6, Q5). Therefore, it takes the complete opposite direction of LOCS and ZNS. In Figure 13 a potential stack for near data processing is illustrated. For more information about exact definitions and current works on such devices, we recommend a survey by Lukken et al [95] on such devices. We will not cover such attempts in-depth as they mostly fall outside of the scope of this survey; we will merely mention them and explain the general idea with an example.

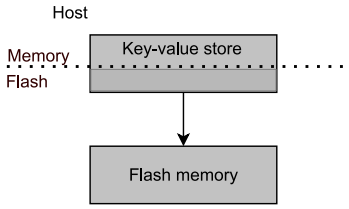

Figure 13: A key-value store design that uses near data processing.

Collaborative-KV (Co-KV) is a proposal to use near-data processing for key-value stores proposed by Sun et al [129]. Co-KV uses LSM-trees and offloads the compaction to the device. This should remove the need to move data between host and device solely for the need of compaction. We have already seen compaction to be a performance bottleneck for I/O, CPU and memory in Section 5.4.3. This design therefore also reduces the CPU and memory footprint for compaction

on the host. However, such a design does require efficient coordination between the host and the device itself.

7 Reducing software overhead

Problem	Solution	Examples
Q6	Shared nothing	Kvell [86], PrismDB [115], SplinterDB [32]
Q6	Polling or interrupts	uDepot [77], SpanDB [27]
Q6	I/O paths	uDepot [77], SpanDB [27], Kvell [86], Kreon [108], Tucana [107]
Q1,Q6	Efficient commit log	SplinterDB [32], Tucana [107], Kreon [108], SpanDB [27], Kvell [86]

Table 5: Overview of papers covered in Section 7.

SSDs are known to be significantly faster than earlier storage devices such as HDDs and tape. SSDs themselves however also do not have a homogeneous performance. There can be a significant difference in the latency and concurrency capabilities of SSDs. Some devices such as *Z-NAND* and NVMe SSDs are known to be able to achieve *ultra-low latency* (ULL) and are therefore at times referred to as ULL devices [74]. To give an example of how low low latency is: Z-NAND can achieve a memory-read latency of 3µs, which is reported to be 8x times faster than the fastest page access latency of modern multi-level cell flash storage [74]. Kourtis et al. come with similar numbers, reporting that fetching a 4KiB block on NVMe SSDs took 80µs and only 12µs on Z-NAND [77]. They even state that storage starts to challenge the performance of network I/O, as a common round-trip latency of a TCP packet over 10 Gigabit Ethernet is 25-50 µs.

These fast devices are generally directly connected with *PCIe* and the *NVMe* protocol instead of with a more traditional approach such as *SAS*/*SATA* and *AHCI* to achieve optimal performance. SAS and SATA do not match the performance that can be reached with SSDs, limiting the bandwidth that can be achieved with SSDs. PCIe is better able to match the performance of the device. Many different SSD vendors implement their SATA, SAS or PCIe communication differently requiring an abstract host-to-controller interface such as AHCI [141]. AHCI causes some performance bottlenecks and challenges, such as a single point of aggregation, some additional latency and a complicated software stack. NVMe removes this point of aggregation, simplifies the software stack (aids Q6), shortens the hardware data path and allows the software to make use the full device parallelism [141].

For these types of devices, many assumptions about flash and storage in general no longer hold. The result is that if we are to optimise key-value stores for such devices, we should focus on keeping software overhead down to relieve the CPU. Instead of trading storage for CPU, we should trade CPU for storage. We will cover a few of such ideas, which will all focus on solving Q6. Some ideas we have already covered before and will not repeat, such as the B^ϵ-tree and ideas to reduce compaction overhead in Section 5.4.3. Also note that a large part of these ideas, seem to contradict what we have determined earlier. That is because optimisations depend on the complete system architecture. So before optimising, always check where the bottleneck of the current underlying architecture is.

7.1 Shared nothing

A design that is frequently used to increase concurrency capabilities, is the *shared nothing* approach. Each thread has its own data structures and sharing of data between threads is kept to a minimum. Such an approach, ensures that threads do not have to content for the same resource. This essentially removes locking issues and allows each thread to use its full capabilities. Some of such ideas used for key-value stores include: dividing the key-value stores index structures across threads [86, 115], letting each thread handle a subset of the key space [86] and per thread WALs [32].

Kvell uses a data structure that is optimised around the idea of shared nothing [86]. They propose giving each thread a subset of the key space and its own B-tree index structure. The B-trees are supposed to be completely stored in memory, negating ordinary B-tree drawbacks. Such a design, can require some extra DRAM (worsening Q5), which can be reduced by only storing the prefixes of keys in the B-tree. This is similar to what we have seen for hash designs in Section 5.2. The design of Kvell also does not force storing data in sequence in any way, reducing the need of communication among threads, but instead allowing more random I/O to occur.

Another interesting idea, which is adjacent but not the same as shared nothing, is to make use of multiple memtables in the LSM-tree design. For example using two mutable MemTables for each individual channel [136] instead of only one mutable and one immutable table in total. This makes better use of both the internal flash parallelism and of the CPUs parallelism and does not rely on a single point of operations. With the increasing performance of storage, we can only expect to see more of such works in the future.

7.2 Polling or interrupts

A classic dilemma is the decision to use *interrupts* or *polling*. Polling does a request to the device and then regularly checks if the request is ready; all the time keeping the thread busy. Interrupts on the other hand do a request to the storage device and then temporarily context switch. Once the data is ready, the context is switched back to the original context. Interrupts make sense when the I/O is slow because it is inefficient to keep a thread busy while other work could be done at the same time. However, they do come with one major disadvantage: the need to context switch. These context switches take time and can trash the cache. Typically I/O was slow, so this decision was given less thought for storage. Yet, with the onset of faster SSDs, it raises serious questions about the use case of interrupts for flash as interrupts might actually reduce the performance.

Kourtis et al. addressed this concern and discuss using polling for fast devices [77]. Some other works also make use of asynchronous I/O, which we will discuss further in Section 7.3.1. Unfortunately, the line is blurred between the two concepts and we will therefore mainly focus on one of the two: asynchronicity. For now, we conclude by stating that asynchronous I/O can make use of polling and that the decision to use polling or interrupt will remain relevant.

7.3 I/O path

There exist a multitude of ways to interface with the storage device. Most of these go through the kernel. This adds software overhead as it adds additional complexity and depends on expensive operations such as *system calls* (syscalls). Lepers et al. even mention that with fast devices, focusing on reducing the number of syscalls instead of focusing on reducing random I/O is more lucrative [86]. I/O that goes through the kernel is also dependent on the Operation System and is unaware of the key-value stores' characteristics. This causes a semantic gap similar to what we have seen in Section 6 between the key-value store and the FTL. Similar approaches for solving the semantic gap might therefore be beneficial, such as rewriting layers for the key-value store or removing layers. In this section, We will cover a few common ways to access I/O and what problems and solutions have been identified for these approaches.

7.3.1 Asynchronous I/O

Traditionally applications made use of *synchronous I/O*. Synchronous I/O means that whoever calls the action has to wait for the action to complete. This is in the case of a local key-value a thread. In order to perform multiple operations concurrently, the application typically adds more threads. In order to achieve full performance, all flash queues need to be saturated, which are quite a few. Unfortunately, this adds expensive context switches and does not scale. *Asynchronous I/O* on the other hand means queuing a request, but not waiting for the request to finish; allowing something else to be done concurrently without requiring a context switch.

Asynchronous I/O makes it possible to simply add a request, such as a get to a request, and at the same time move on to different tasks in the same program. This more efficiently fills the device queues and allows the application to make progress. Linux allows both, a synchronous I/O backend [77] and asynchronicity with AI/O [15, 77] and io_uring [84].

Key-value stores like Kvell [86], SpanDB [27], uDepot [77] and others [84] all report using asynchronous I/O to increase performance. Using multiple asynchronous operations allows satisfying the individual queues of the flash device better.

7.3.2 User-space I/O

user-space I/O is another way to address I/O. Kourtis et al. [77] mention that such a design moves the logic entirely to the user-space, removing the kernel from the data path. This in turn reduces context switches, copying overheads and scheduling overheads. In short, it removes layers and thus potential overhead.

Kourtis et al. [77] therefore opt to use user-space I/O where possible. Unfortunately, they also mention that such a design requires access to the device, which is not always possible. This still necessitates traditional approaches for many use cases and leads to a major reason to still use synchronous I/O, compatibility. Therefore, they propose their own scheduler, known *TRT*, that can use various I/O backends, allowing the key-value store to use what is most efficient and available. A key-value store should therefore look into using the most-efficient backend, but to be compatible needs to keep its options open for now. An example of a user-space I/O framework is SPDK [144].

7.3.3 Memory-mapped I/O

A common way to access data on the disk is by using *memory-mapped I/O* (mmap). Mmap directly translates virtual addresses to logical addresses on flash and allows applications to treat storage like it is part of the memory. This can in principle be done with any of the earlier mentioned I/Os such as synchronous or asynchronous I/O.

Such a design can be a handy interface but is unfortunately hard to control. That is because the mmap implementation determines what is in memory and what is on disk at any given time, which leads to a semantic gap again. It also comes with extra operations such as translation operations. Therefore Papagiannis et al. propose using a custom implementation of mmap, known as *Kmmap* [108]. Kmmap allows assigning priorities to pages, which essentially separates hot and cold data. Separating hot and cold data helps with assigning what pages to evict. It also comes with several optimisations such as asynchronicity for writes, which make it a performant choice

for key-value stores. We thus see that memory-mapped I/O is still viable for fast devices, but it might need some slight modifications to be optimal for flash and key-value stores.

7.3.4 Direct I/O

Another type of I/O is *Direct I/O*. Direct I/O directly accesses the device, which can again be both synchronous and asynchronous. This allows the key-value store to directly determine all of the I/O operations. This can essentially remove a big part of the semantic gap, as the programmer is given more control.

Unfortunately, such a design is known to not be able to fully saturate the disk queues when requests are done synchronously. This is because as discussed in Section 7.3.1, synchronous I/O has to wait for completion. Therefore, Lepers et al. state that it would in such a case be more beneficial to *batch* operations [86]. Batching issues multiple operations at the same time. We suspect that batching can also have gains for asynchronous I/O. This not only allows more operations to be completed in one go, but it also reduces the number of syscalls. This is actually very similar to buffering techniques we have seen in data structure optimisations in Section 5. To reduce expensive syscalls, we should group multiple operations into one batch operation.

7.3.5 Caching pages in memory

Caching is a general optimisation technique that is also used heavily in key-value stores. It is a common practice to cache pages retrieved from storage to memory. Contrary to what one might believe, such caching can also reduce performance. Caching can for example increase software overhead, especially when cache misses are frequent. There is no clear consensus on how caching should be done. Therefore, we discuss a few topics and hope to make clear that there is no silver bullet.

Caching is typically done by using the page cache of the underlying system, such as the *virtual file system* (VFS). Such a design is inefficient for a number of reasons [86, 108]. Firstly, it introduces duplication since data needs to be copied between kernel and user space. Secondly, the page cache uses its own eviction policy. Such a policy is not optimised for key-value stores and can keep files in memory that might not be necessary or remove files that are in fact necessary. Thirdly, it can only issue one read at a time for a single thread or use locking in the multi-threaded case. Various works also mention various other disadvantages [77, 86, 108], but the gist is that such a cache is not aware of the use case and is not optimised for such a case in the first place.

Therefore an idea is to create a cache optimised for key-value stores and cut out the middleman. Kvell uses such an approach with its own internal page cache, which uses a *Least Recently Used* (LRU) order [86]. SplinterDB also uses a user-level cache, optimised for concurrency [32], and SpanDB uses a user-level cache made for its own file system, *TopFS* [32].

It is also possible to use mmap for caching to avoid having both a copy in the kernel and user space [107]. *Kreon* uses a modified version of mmap, Kmmap (see Section 7.3), to bypasses the Linux page cache [108]. It uses a priority-based FIFO replacement policy. Such a priority-based policy allows keeping important data in the cache, such as in the case of LSM-trees: L0 and the WAL. It also used independent banks, that are completely independent and allow for fine-grained locking, which in turn allow for full parallelism.

Common among all these cache replacements is that they add semantic knowledge to the cache and opt for more concurrency capabilities, which we assume helps with increasing the performance of key-value stores. However, we can also ask ourselves if we even need a data cache in the first place (excluding CPU caches). Such a consideration is made possible because of the speed of modern SSDs. Cached data is typically stored in DRAM because of its lower latency compared to storage, but the gap is closing. Key-value stores use less memory and have a lower software overhead when the cache is dropped (Q5 and Q6 in one). Designs without caching such as uDepot [77] have shown that such a design can have potential. In short, many designs that consider reducing the software overhead, focus on moving the cache away from the kernel path and towards the user-space or opt to remove the cache entirely.

7.4 Efficient commit log

We already discussed the usage of a *write-ahead log* (WAL) as a commit log for data structures such as LSM-trees in Section 5.4. Such a WAL does not only cause write amplification, it also incurs extra software overhead. There exist alternatives for the traditional WAL that are less of a burden on the CPU.

The first idea we will cover is turning the WAL into a concurrency friendly data structure. The WAL is by default one single shared log. This leads to resource contention when multiple threads are used. Therefore Conway et al. opted for a per-thread WAL for the design of SplinterDB [32]. In order to ensure order on recovery, they make use of *cross-referenced logs*. Each operation that is performed increments a generation number that is logged along with the data in the individual WALs, enabling a global order. SpanDB takes a similar approach by using multiple parallel WAL write streams [27].

The second idea we will cover is using *copy-on-write* (CoW) for persistence instead of a WAL [107, 108]. WAL causes each write to be done twice. Once to the WAL and once in-place to the actual data structure that is used. However, such a design does force I/O to be sequential. The main idea is therefore to trade more writes for sequential I/O (increasing Q1). CoW never does in place updates. Instead on a write, an entire structure is immediately rewritten out-of-place. This guarantees versioning and persistence because both the old

and the new data structure remain. At some point in time, the old entry can then be removed. CoW therefore only writes new data once, but it does require more random I/O. As the data is not stored sequentially as would be the case with a WAL. It thus trades fewer writes for random I/O (solving Q1 issues). We already know that flash has fewer problems with random I/O and performance can be gained, especially with faster SSDs.

The last idea we will cover is simply not using a commit log in its entirety. We have seen similar ideas in Section 5.4.5 for LSM-trees. We now cover commit logs in general. For example, Kvell completely drops the commit log because it does not buffer updates at all. Buffering does not always make sense when devices are fast enough. Instead in such a design updates are immediately flushed to disk, preventing unnecessary I/O (Q1) [86].

8 Data related optimisations

Problem	Solution	Examples
Q1,Q2,Q4	Transform access patterns	Key reshaping [73]
Q1,Q2,Q3	Use spatial locality	EvenDB [55], SlimDB [118], RocksDB [48]
Q1,Q2,Q4	Skewed data	EvenDB [55], TRIAD [8], elasticbf [89]
Q1,Q2,Q3,Q4, Q5,Q6,Q7	Compression SSDs	KallaxDB [28], B-tree compression [113]

Table 6: Overview of papers covered in Section 8.

Various novel optimisations only make sense under certain circumstances or are very specific and are not necessarily tied to a data structure. For example optimisations for certain types of datasets or certain access patterns. In this section, we will discuss these types of optimisations.

8.1 Access patterns

It has already been argued that the differences between sequential and random accesses are less severe on flash storage than on traditional storage media such as HDDs. Nevertheless, flash still performs better on sequential access patterns than on random patterns. That is because most SSDs are still limited to a block device, which means that if multiple values are stored on one block, fewer reads are necessary (reducing Q2). Further on, various data structures such as B-trees and LSM-trees perform better with sequential operations. Random operations on B-trees might for example require going back from the top of the tree to the bottom of the tree for each random operation, in the process also tampering with the cache. The difference becomes more pronounced with scan operations. Stores such as LSM-trees (and especially tiered LSM) can also result in many reads for each individual read through read amplification.

Various works have therefore proposed to prefetch results beforehand [24, 94], which can mitigate the performance issues of random I/O (Q2). Another solution that has been proposed, is to make the key pattern more sequential, removing the problem entirely. This can for example be done with key reshaping proposed by Kim et al. [73] Key reshaping uses an additional B-tree to translate keys to sequential keys, which also requires an additional key for each unique key. On an insert, a new sequential key is created, which is used for the actual put operation in the key-value store. At the same time, there is a separate store, where the old key is inserted. The old key then points to the earlier created sequential key. On a lookup, the sequential key is retrieved through this extra B-tree, which is then used on the key-value store. So each operation requires one extra operation in a translation tree. This additional tree also needs to be stored but is relatively small as keys are generally small [21]. Key reshaping is shown to significantly increase read and write performance (effecting Q1, Q2 and Q4) [73]. It should be noted that when a key-value store is only exposed to sequential patterns, some optimisations that we discussed in earlier sections no longer make sense. For example for LSM-trees it becomes possible to directly copy SSTables to the next level on compaction instead of merging into the next level, making optimisations such as tiering (partitioned forests) inefficient.

We will also shortly cover a unique problem that can occur with *3D NAND* devices. 3D NAND devices stack multiple layers of flash on top of each other, allowing flash storage to become denser. Unfortunately, this can also cause excess heat to move from one layer to another, causing vertical wear levelling. This necessitates moving data around to different places when using such technologies. Such optimisations for LSM-trees are proposed by Wang et al [137]. The lesson to learn here is that wear levelling issues can differ between devices and still needs to be accounted for.

8.2 Spatial locality

There exist a variety of patterns that are common in key-value stores. Properly using such patterns can improve performance. For example, *composite keys* are common, which are essentially keys existing out of multiple parts. Think for example about multiple files in the same directory, where the prefix is the directory and the suffix is the filename. For such types of data, spatial locality typically has a meaning. However, this is often not accounted for. LSM-trees for example group data based on temporality and not on locality. Further on, such temporal grouping is indiscriminate with regards to key popularity, causing wear levelling by excess writing of cold data

(worsening Q1).

Gilad et al. propose a data structure, known as *EvenDB*, that combines the batching of LSM-trees with spatial locality [55]. To achieve this they make use of *chunks* as the unit of I/O, which hold contiguous key ranges. This means that instead of caching or writing pages, always a range of key-value pairs is written. Each chunk has its own key-value range, SSTable and WAL, allowing related data to stay together. On a read, the appropriate key-value range must be found, so the appropriate chunk. If the chunks index is in memory, the data is read directly from this chunk. Otherwise, the index first needs to be loaded into memory. The data can then be retrieved both from the chunks SSTable or the WAL. Since a range is used, scan performance can be significantly increased as a part of the range to scan can be brought directly into memory (reducing Q2 effects). This can significantly reduce write amplification and thus wear levelling, but such a design would not work when data has no spatial locality.

Composite keys are also a type of *semi-sorted data*; data where sorting does matter, but frequently only on a part of the data. For example, if you want to query all data of one user that all use the same prefix in the keys. Semi-sorted data typically exists out of a prefix and a suffix, such as a directory and a file path for file metadata. Key-value stores are generally not optimised for such data. This can cause significant read amplification since each key-value pair can be stored elsewhere. RocksDB allows using bloom filters made specifically for prefixes [48], removing part of the bottleneck for reading. Ren et al. propose a key-value store entirely optimised for such a situation, *SlimDB* [118].

Ren et al. note that traditional LSM-trees lack optimisations to maximise read performance on SSD and provide techniques to make use of semi-sorted data to achieve better performance [118]. They use a three-level compact index instead of the standard block index used on each level in LSM-trees. This block index makes clever use of the nature of semi-sorted data. The first index stores the prefixes of the data together and uses *Entropy Coded Tries* (ECT) for compression, keeping memory footprint down (Q5). This links to the second level, which keeps offsets to the individual SSTable blocks. The last index contains the suffix of the last key in each block, these can also be compressed. Compression works in these cases because the data is quite similar to each other, for example prefixes are generally similar. The indexes are optimised for the common access patterns, such as querying for prefixes. Such a design can achieve better space utilisation (helping with Q3), lower average latency and less I/O amplification (Q1 and Q2). Such a design is also highly specialised and not generalising, but illustrates some ideas on how to optimise data stores for specific patterns.

Ideas proposed in this section can also be used for other data patterns, specific use cases such as filesystem metadata or for different devices altogether. It showcases that it still possible to make optimisations on the host side, as the indexes reside in memory, without focusing on how to make better use of the flash device.

8.3 Skewed data

Various users of key-value stores report that their access patterns are skewed [21]. A small part of the data is accessed most of the time and a large part of the data is close to never read. Skewed data has implications for the performance of the key-value store. For example, take LSM-tree based stores. Data is flushed to L0 after a set number of operations independent of the amount of unique keys. If all of these operations only operate on the same key, they essentially only flush one value to L0, which is highly inefficient. Further on, this effect will be amplified by later cascading compactions (increasing Q1). Balmau et al. therefore proposed to not flush hot entries and keep them in memory and the WAL [8]. Gilad et al. propose using a row cache for frequently accessed keys [55]. This can keep the hottest entries in memory most of the time, preventing excess I/O (concerning Q2 and Q1).

Most key-value stores try to reduce I/O cost with the help of AMQs, see Section 5.1. However, traditional AMQs such as bloom filters do not account for skewed data. Ideally, such filters should prevent unnecessary lookups, even for skewed data. Therefore Li et al. proposed ElasticBF [89]. A bloomfilter that considers skewed data and can thus drastically increase read throughput for such workloads.

8.4 SSDs with compression support

There exist flash devices with internal compression capabilities. Such devices can expose a larger address space to the user than actually physically exists on the device. This can be done with the help of virtualisation that makes use of said compression. This virtualisation can be directly implemented on the device with the help of an FTL. For example, a device might expose 32 TB of logical blocks, even when the physical space might be only 4 TB. Such a design is visible in Figure 14.

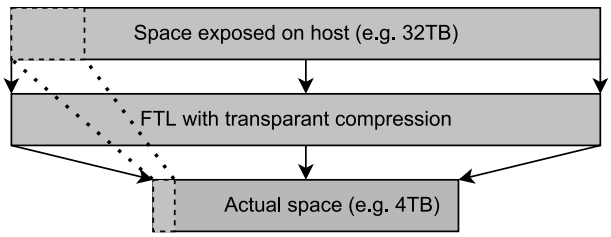

Figure 14: Example of the sparse address space when compression is used. Modified variant of a figure posted by Qiao et al [113].

Chen et al. investigated the viability of using the unique properties of such devices for key-value stores in the form of a new key-value store, *KallaxDB* [28]. This design directly

maps hashed keys to logical locations on the SSD, without using a physical hash table. Instead the address space of the device functions as the actual hash table (addressing Q7). Key-value pairs are stored at logical locations. Logical locations are defined in units of blocks; for example, KallaxDB reports 4 KB as a block size regardless of compression [28]. This design would could cause under-utilisation issues on SSDs without compression capabilities because there would be various blocks that are only partially occupied, causing fragmentation (impacting Q3). Blocks will be partially occupied because each key-value pair has to resort to using at least an entire block and frequently key-value pairs are small, sometimes only a few bytes [21]. Virtualisation that uses compression can mitigate this issue as the address space can be made sparser. There is still under-utilisation in this case, but the actual fragmentation effect is smaller because each logical block counts for less physical space on the device, for example reducing 4 KB to 1 KB. The tableless design should make it possible to also run key-value stores on devices with little memory or CPU power because there is very little logic involved on the host (mitigating Q5 and Q6 issues). Lastly, KallaxDB is shown to still achieve adequate throughput and latency, even under compression. This compression can reduce both write and read amplification (Q1, Q2 and Q4).

Qiao et al. showcase that such devices are also beneficial for B-trees [113]. They supposedly reduce the gap between LSM-trees and B-trees. To achieve this B-trees can make use of the sparse address space of the device, as the device has a larger logical space than physical space because of the aforementioned virtualisation. This can mitigate ordinary issues with operations such as random I/O, which were discussed in Section 5.3. Using the compression capabilities of the device reduces write and space amplification issues (Q1 and Q3). Therefore, Qiao et al. suggest that it warrants at least a revisit to the role of B-tree and LSM-trees for future data management systems [113]. If such SSDs remain relevant, we would expect more specialised designs for such SSDs.

9 Hybrid architectures

We have discussed various optimisations techniques for running key-value stores solely on flash storage. Nevertheless, flash storage is not the only component capable of storage. The idea that a key-value store will only be stored on flash, would be a simplification of the reality. For example, it is common for data centres, cloud providers and grid clusters alike to have a heterogeneous selection of storage devices [78]. Alternatives to flash have their own advantages and disadvantages that can be used for optimising key-value stores or simply keeping the total monetary cost down. This leads to hybrid designs, where different devices are used for different parts of the key-value store (addressing Q7). This allows key-value stores to use the advantages of different devices and mitigate their disadvantages. We will discuss a few hybrid designs that have been proposed.

Problem	Solution	Examples
Q1,Q4,Q7	Flash as cache	FlashStore [41]
Q1,Q2,Q7	Multiple flash devices	I/O Isolation scheme [72], SpanDB [27], CaseDB [134]
Q1,Q2,Q4, Q6,Q7	Integrating NVM	SSHKV [146], NoveLSM [70], RangeKV [145], SplitKV [58], prismDB [115]

Table 7: Overview of papers covered in Section 9.

A common approach we will see in such designs is that storage devices are differentiated based on their latency and throughput. In all the approaches we will discuss the faster device is considered more expensive and the slower device holds more data than the faster device. In such approaches, we identify two prominent designs: a design where the fast storage is merely used as cache between DRAM and the slower storage, and designs where the main data structure is split between the devices. The two designs are shown in Figure 15(b) and Figure 15(a) respectively. Ren et al. also identified such a distinction [115]. We will refer to these approaches as the *cache approach* and the *split approach* from now on.

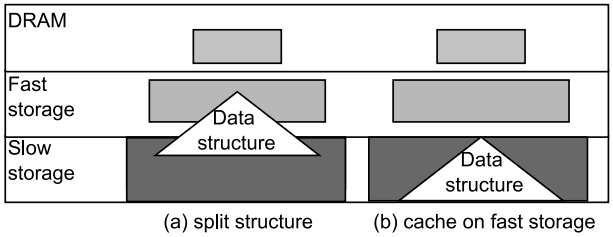

Figure 15: Two major hybrid key-value designs. The data structure can in principal be anything.

9.1 Flash as cache

Flash has up until now been more expensive than HDDs. When the amount of data that needs to be stored becomes expansive, it can become beneficial to store the bulk of the data on HDDs instead. In that case, flash can be used as an extra layer of cache in between DRAM and HDD. This requires keeping hot data on SSD and eventually destaging cold data to HDD.

Such an idea is proposed by Debnath et al. in FlashStore and follows the cache approach [41]. In this design, valid and recent data is stored on flash and indexed by a hash table in

DRAM. Old data is stored on HDD and needs to be separately indexed. This also reduces the amount of data that is stored on flash, indirectly reducing Q1 and Q4 challenges for flash.

9.2 Integrating multiple flash devices

Using different types of flash in a key-value design is also a hybrid design because, as discussed in Section 4.2, there exist multiple types of flash. Some are denser and some have lower latency. We will cover a few examples to show that differentiating flash has actual merit.

It has been proposed to use the split design on different flash devices. For example, *SpanDB* splits a LSM-tree over multiple flash devices [27]. It proposes to put the frequently accessed parts of the store, such as the WAL and L0 of an LSM-tree on faster flash and the rest on slower flash. Performance is said to improve when combining multiple devices [27], despite it probably being faster when only using fast flash. Nevertheless, it is hard to truly compare such hybrids against alternatives, especially because it is dependent on the mix of storage devices used. *CaseDB* also uses a split LSM-tree design, but combines it with the idea of key-value separation [134], discussed in Section 5.4.2. It stores all the metadata, bloom filters and indexes on fast NVMe SSDs and stores the actual values on a slower SSD. CaseDB reports better write performance than WiscKey by as much as 1.8 times and increased read performance by as much as 1.5 times [134].

To overcome limits of individual flash devices, it is also possible to combine multiple physical flash devices and treat them as one logical flash device. This allows spreading the load over multiple devices and negates limits that would occur when only using one device (such as effects of Q1 and Q2). Such an idea is an I/O isolation scheme for SSDs proposed by Kim et al [72]. This is reminiscent of the scheduling logic we saw earlier in LOCS [136] and SILK [9], see Section 6.3 and Section 5.4.3 respectively. It classifies operations and files and determines what SSD will execute a particular operation on a particular file. To give an example: they show that is possible with such a design to only send application writes to certain SSDs, to allow other SSDs to be reserved solely for application reads, which can significantly increase read throughput and decrease read latency.

9.3 Persistent memory

Persistent memory is an emerging topic with a lot of attention from the community and various applications are already optimised for it, such as key-value stores [13, 67, 148]. These devices can be put in memory slots just like DRAM, but are unlike DRAM able to persist data. Persistent memory can persist data even when the power is cut off. Another interesting property is that since they are in a memory slot, they are byte addressable (at cache line granularity). Optimising for such devices requires a radically different way of thinking. For most these devices are at the moment still unobtainable and expensive, but they are an interesting topic nonetheless. Further on, such devices are not necessarily flash themselves, but there have also been works that proposed to use both flash and persistent memory in an architecture. Persistent memory goes by many names and has various implementations. We will use the abbreviation for *Non-volatile memory*, NVM, where possible, even if it is not always accurate. We will shortly cover a few of such hybrid ideas.

We see that split and cache designs are both very common, as we uncover examples for all major data structures covered in Section 5. In all cases the part stored in NVM is altered to be more efficient on NVM, such as making use of the byte addressability. *SSHKV* [146] splits a hash table, storing the metadata on NVM and storing the values on a log on flash. *SplitKV* [58] uses a cache/split design for a B+-tree, storing hot and small items in NVM, and cold and large values on flash (reducing Q1 and Q4 in flash). *NoveLSM* [70] and *RangeKV* [145] both resort to splitting LSM-trees. NoveLSM keeps the mutable and immutable memtables in NVM and RangeKV stores a modified L0, referred to as *rangetabs*, in persistent memory.

Nevertheless, it can also be beneficial to use completely reworked data structures altogether. Split and cache designs have their merits, but might not fully use the full potential of the available stack. We will cover one such design, known as *PrismDB* [115], which combines *Quad Level Cell* (QLC) NAND with *3D Xpoint* (Optane). This combination is remarkable, as it combines both a dense flash device and a fast NVM, combining the benefits of both worlds. PrismDB uses a partitioned, shared-nothing architecture, where each thread has its own subset of keys. This separation goes down from DRAM, to NVM to flash. This allows the key-value store to more efficiently use the low latency and parallelism that NVM and the CPU can provide (addressing Q6 for NVM). In DRAM a B-tree is used, on NVM a part of the objects and metadata are stored and the rest of the values are stored on flash. It tries to keep hot objects on NVM and moves colder objects to flash through compaction, which is similar to various caching approaches. Data on flash is then stored in SSTables, similar to LSM-trees.

In short, we see that NVM can be used efficiently in combination with flash devices in order to optimise key-value stores. It can be used to split a data structure, to cache data on NVM or to create a data structure optimised for the combination of the two.

10 Measuring performance

We have already determined what we think defines performance in Section 4, yet actually measuring this performance is non-trivial. Especially with the layered design and the heterogeneity of the devices that most key-value stores make

use of. Additionally, various designs combine a multitude of ideas, add one new part, and yet test against an alternative with significantly fewer ideas. It raises serious questions about the validity of the comparison of various designs. Researchers have therefore opted for new ways to measure performance. Some have focused on specific data structures, others on general problems. This section will focus on the status of this field of studies and how key-value stores can be benchmarked in general. It will also take a look at the common workload patterns that key-value stores are typically optimised for.

The first tool we will cover is for LSM-trees, proposed by Batsaras et al. and is known as *Variable Amplification Analysis* (VAT) [11]. VAT introduces an analysis to make trade-offs for LSM parameters and their effect on among others I/O. It covers various parameters such as SSTable sizes, growth between levels, tree and forest optimisations, key-value separation optimisations and how the effects of these can be properly measured. This immediately captures a big part of the LSM optimisations we covered in Section 5.4. Such designs make it easier to reason about the optimisations. As we are speaking about such optimisations, it would also be a good idea to ensure that when stores are compared, they are compared fairly. Both should be optimised. For LSM-trees there already exist some tooling to aid in automatic optimisation, such as Monkey or Dostoevsky [38, 39].

There also exist a myriad of other pitfalls. Fortunately, some of these have been identified. Didona et al. covered various pitfalls that are made in measuring the performance of key-value stores with regards to flash [45]. They also come with advice on how to solve these pitfalls. Some major findings include that it is essential to measure write amplification from both the application and the device (AWA), tests should be run for a longer period of time, tests should mention the *state* of the SSD, test should also be done with different data set sizes, tests should consider metadata, tests should consider *over-provisioning* and tests should consider heterogeneous flash devices. The state of the SSD refers to the various states that an SSD can be in. For example, a device can be empty or it can already be filled partially. This has performance implications and must be noted down to make research reproducible. Overprovisioning implies that a part of the SSD can not be used by the user, but is used by the device itself for internal operations. This has implications for various operations such as garbage collection or out of place writes. Didona et al. also show a few examples where depending on the device, a B-tree or an LSM-tree can perform better, hence the rule on heterogeneous devices [45].

We have thus already seen an analysis tool, some automatic optimisers and a set of pitfalls. As far as we know there exists no tool that can automatically verify all constraints and ensure validity. Similarly, there are no guarantees that all recent studies or future studies will uphold the pitfalls. This is probably in part because of the vastly heterogeneous use cases of key-value stores and the difficulty in setting up such tooling. For now, it is therefore simply a set of guidelines. For future devices or stores, it would therefore be ideal if they would adhere to said guidelines and maybe add some analysis tools such as VAT.

At the moment there do exist various benchmarks that can be used to partially test key-value stores on a few common workloads. This can be used to show that on someone's hardware a key-value store performs better than another key-value store for a particular use case. In general, a good benchmark makes use of a real workload, to showcase that it is also usable in real scenarios. Such benchmarks are therefore in general provided by companies that actually use key-value stores and have a lot of data. Therefore, the workloads made by these companies reflect their workload and might not reflect edge cases or niche applications, making such benchmarks difficult. Instead, they approximate, which in the general case is enough. However, for high performing solutions, this probably requires a different workload. Some example benchmarks include Yahoo's *Yahoo Cloud Serving Benchmark* (YCSB) [33] and *db_bench* [21] from Meta, which is integrated in RocksDB.

YCSB seems to be especially prevalent. Cooper et al. recognise a set of data distributions and workloads attached to said distributions [33]. Some example distributions are *uniform* and *zipfian* distributions. Uniform evenly distributes the accesses and Zipfian distributions have skewed accesses, with more accesses in a small part of the set and a long tail tagging behind. The default workloads, named A to E, represent a set of common and distinct use cases: update heavy(A), read heavy(B), read only(C), read latest(D) and short ranges(E). This can be used to show case that when a key-value store is deployed on flash, it is suited for various use cases.

Cao et al. characterise the workloads of RocksDB and showcase that workloads differ wildy [21]. They come with various methodologies to analyse the workloads of three common use cases. Data from Facebook, ZippyDB (distributed key-value store) and UP2X (used for AI/ML) is traced to get the distribution. In all three workloads, the distributions differ significantly. Again confirming that each workload is different. However, they also conclude that spatial locality is important, making optimisations in Section 8.2 sensible.

In short, for now, measuring performance of key-value stores for flash includes using prevalent benchmarks with common workloads. At the same time measuring flash specific metrics and adhering to a set of pitfalls. Lastly, adding some extra analysis where necessary such as VAT.

11 Future Prediction

Now that we have discussed most major contributions to key-value stores with regards to flash, we would like to discuss what will happen to key-value stores in the future. We have seen trends with each new type of device, both in academia and the industry. Think about key-value stores for HDDs,

SMR HDDs, ordinary SSDs, NVMe SSDs and various other variants of SSDs. Similarly, we have seen that it takes time for each technology to be adopted by the community, where some fail and some make it, but if an idea is promising it seems to come back. Think for example about near data processing technologies or open-channel designs; ideas that both already exist for a while.

If techniques discussed in this survey become more widely adopted, they can have significant effects in the near future. They allow reducing write, read and space amplification, which aids in better throughput, latency and lower wear levelling. Further on, they allow lower CPU overheads and lower memory overheads. This would require a smaller investment in flash storage, as the storage can be more effectively utilised and breaks done at a slower pace (wear levelling), requiring less replacements. This in turn can reduce costs of the data center. At the same time a few of the solutions require less of the host, allowing a reduction in investments for these devices. This also has an effect on the amount of energy necessary, as less devices need to be used. We have already seen, that many users of key-value stores have moved to using flash storage. We therefore expect (1) that the techniques discussed in this survey will become more widespread, which would allow these improvements to occur. At the same time this does imply that key-value stores will become primarily optimised for flash storage only and not for other media (yet).

Flash storage is moving in multiple directions. On one side we see a push in making flash storage denser to account for a need to store more data in the future. For example multi-cell technologies such as QLC and PLC NAND flash. Each of these is becoming denser, yet less tolerant to wear-levelling and has a higher latency and lower throughput. On the other hand, we see a push to increase the performance of flash storage. For example, Z-NAND or faster NVMe devices. We thus see both a push to faster storage and denser storage. We expect (2) key-value stores to move along; some key-value stores that aim to address the denser storage and some that aim to address the faster storage. This will on the long term cause either various different key-value stores or key-value stores that become highly complex as they need to address different needs. Developers than also need to consider what key-value store to use along with their storage decisions and requirement decisions, adding further complexity. This requirement of different needs is going to increase on the long term, as the needs keep expanding. On the long term this does allow using different stores and storage for the appropriate needs, which more efficiently uses what is available.

We also see a push in reducing the amount of layers needed to address storage on flash storage. We have for example seen open-channel SSDs and more recently ZNS SSDs. We have also seen key-value store solutions for open-channel SSDs, such as LOCS (see Section 6.3). In the short term we expect (3) more solutions for ZNS SSDs as the technology seems promising and is already available. If this technology keeps expanding, it can on the long terms have effect on the cost and durability of storage in the data center, as such devices can reduce wear levelling and write amplification when properly used [127]. The ordinary commodity SSD, having already stood the test of time, will also not go away anytime soon, but ZNS devices can significantly increase the performance of key-value stores, yet are at the same time not significantly more expensive to produce.

Recently, quite a few storage technologies different from flash started to receive attention from the community. There is an increased interest in using alternative storage media and materials [26]. The main intent is to find a type of storage that better fits modern-day use cases, such as dealing with large amounts of archival data or cloud computing. This is becoming especially important as the amount of data that is needed in the world is ever-increasing. We also see more requirements to reduce the amount of energy needed and an interest in longevity of storage (archiving). Some examples include glass storage in the cloud for archiving with project Silica [6] or an interest in storing large amounts of data in DNA [87]. By some, holographic storage is also reconsidered as a viable storage medium [26]. We expect that more of such media will be developed or reconsidered and that some of them will be also be considered for key-value stores in the long term. Similarly, we expect some storage media that are already in use for key-value stores to become more important. Such as persistent memory. We have already seen that there have been various works, both academic and industrial, that opt to use persistent memory for key-value stores (see Section 2). Thus in short, we see that there is a push to use different types of storage, all with different characteristics and some that are not even ready for usage yet. Key-value stores being a technology to store data, will therefore both in the short, medium and long term be considered and altered to make use of these storage media. As these technologies have different characteristics, it is likely that we will also see different types of data structures, different from the LSM-tree and B+-tree discussed in Section 5. This means that we expect (4) that key-value stores will not be limited to HDD and flash technologies and should probably on the long term also not be treated as such. This indeed implies, that key-value stores can be used for all kinds of different needs in all kinds of different situations.

In short, we expect that flash will remain a major storage medium for key-value stores for a long time. We expect that novel flash-specific techniques, as discussed in this survey, will become adopted and see a widespread usage. We also expect a part of the flash storage to move to denser solutions and a part to faster solutions. We expect key-value stores to move along on the long term, which will require different approaches. Further on, we expect more flash storage devices to adopt open-channel and ZNS designs and key-value stores to become more optimised for such solutions. At the same time, we also expect other completely different storage media

to be considered at the long term for key-value stores or the other way around. This will require rethinking many designs decisions, which have been made for flash. Such redesigns are already done for persistent memory. In the end, all of these different requirements and designs will lead to vastly different implementations and probably different APIs.

12 Design guidelines for development of flash-based key-value stores

There exist various approaches that can be taken when designing key-value stores for flash. None of these approaches is necessarily wrong or right. However, we believe that there is a lot that can be learned from already existing solutions, such as the ones described in this survey. Based on these solutions, we come with a set of guidelines that can aid in creating a key-value store on flash. Note that this survey does not cover all possible use cases (see Section 13), but the general remarks provided in this section can still aid in developing a key-value store. Further on, in this survey more in-depth optimisations are marked with numbers such as *Q1* or *Q2*, which should further aid with finding solutions for requirements. Each optimisation should always be tested against the alternative, preferably with benchmarking tooling.

When designing a key-value store for flash, we recommend to first take a look at the use case and requirements of the key-value store in question. For example, should it have a high throughput, be space efficient or something else altogether. As is discussed in Section 10, different use cases typically have different data distributions. Different distributions in turn allow for different optimisations, such as is discussed in Section 5 and Section 8. Therefore, we recommend (1) to first analyse or estimate the workload of the use case. This should be about what patterns exist in the data and the balance that exists between writes, reads, updates and deletes. When the workload is known, look if there already exists a key-value store optimised for this workload that satisfies the requirements. For example, key-value stores such as RocksDB [48] or LevelDB [54]. If not, this still helps with deciding on a data structure. For example, if the workload is write-heavy, data structures such as LSM-trees or other WOIs (see Section 5) might be an option. Depending on the workload these can be further optimised, such as LSM-forests instead of LSM-trees (see Section 8.4). Similarly, if we know that the data is skewed or certain access patterns exist it might be beneficial to look into data related optimisations (see Section 8).

Once the use case is known, we recommend (2) looking at the storage hardware that will be used to store the data. Different hardware, needs to be treated differently. If the storage that will be used is heterogeneous, look into how all of these storage components can be used together effectively. For this we refer to Section 9. For example, the faster storage can be used as cache between DRAM and slower storage, an existing data structure can be split between faster and slower storage, or an entirely novel data structure can be used that is optimised for a combination. Key-value store designers should also consider whether only flash is used or also other storage media are used in this hybrid approach. For example, if persistent memory is available it might also be beneficial to look into optimisations for persistent memory. Designers should also look at the type of flash storage that is used. If the storage is fast, the bottleneck will start to move more to the host. In that case we recommend (3) to also take a look at optimisations on the side of the host. Such techniques are described in Section 7, such as making the data structures more concurrency friendly. If we know that the CPU on the host is weak, such design considerations are also be beneficial. Therefore consider (and benchmark), where the bottleneck of the key-value store is. Further on, designs should also address the memory (Q5) that is available on the host. As we have seen in Section 5, certain index data structures need to be altered to make use of the memory available. Depending on the memory available, different data structures and caching policies make sense.

Another aspect we recommend (4) looking at, are the interfaces between the key-value store and the storage. If the store should be able to work on all kinds of devices and operating systems, it should adhere to a common interface, such as a file system. However, if not and storage devices are used that allow improving the communication between storage and host, this should be considered. This can help among others with better performance and reduced wear levelling (device longevity). This includes optimisations that are described in Section 6. For example, using open-channel SSDs and dropping the file system and FTL to allow the key-value store to completely control the device.

Lastly, we recommend (5) to benchmark as described in Section 10. For example, by adhering to a set of storage pitfalls, using common benchmarks and using workloads provided by companies that use key-value stores in production. To be able to use a common benchmark, it is advisable to adhere to a common interface of a benchmark such as YCSB [33] or db_bench [21].

In short, there a few steps to consider when deciding on a key-value store. Generally, the key-value store should be designed for its particular use case. That means that it should consider the use case's workload, the storage devices available, the CPU and memory on the host devices, the interfaces between the store and storage and should be benchmarked by a common benchmark to ensure validity.

13 Open problems and future work

Key-value stores are a big research field and this survey does not address all aspects and challenges that arise when designing key-value stores. Neither did we find solutions to all challenges that we already know of from literature. The

biggest problems, which we identified as Q1-Q7, mostly reflect what is most prominent in the literature. In this section we aim to address challenges, that we either think should be looked into in a later survey or that are still open problems for key-value stores and storage in general.

We have not looked at how much energy is required when using key-value stores. Another survey could look into how key-value stores on flash have an impact on energy usage. Reducing energy usage is considered important, also for storing data [40, 76, 123, 133], which makes addressing this challenge interesting. Flash storage is said to have a lower energy consumption than alternatives such as HDDs [109], which should aid developers and administrators in making decisions on what storage medium to use. Some potential questions are: "Do key-value stores cost a large amount of energy to maintain?", "How does the energy usage of key-value stores compare to other data stores?", "How does the usage of flash impact the energy usage of key-value stores?" and "How can key-value stores for flash be altered to reduce their energy usage?".

Another issue we have not addressed in-depth is multi-tenancy. We have described one solution, NVMKV [99], in Section 6.2. However, multi-tenancy is common, such as in *Software as a Service* (SaaS) solutions [132], and most solutions we see in the literature do not address this issue. Future work can look at how key-value stores (on flash) address this challenge.

We have also not addressed security and privacy concerns that arise when using key-value stores in this survey. An important requirement for many use cases is to store data securely, so that it is hard for the non-intended to among others read, alter or deny the data. Future surveys/work could look into how techniques proposed in this paper, are secure or can be altered to be more secure or how to secure key-value stores on flash in general.

Similarly, we have addressed some forms of persistence such as WAL and CoW (see Section 7.4), but we have not focused on these parts of key-value stores. Recovery can take a different amount of time depending on the data structure. There is also a difference in reliability of various methods that can be used. Future work could look into reliability and recoverability of key-value stores with regards to flash. This is important, because it ensures that solutions discussed in this survey can be used in critical situations.

An important open problem, and a general problem in storage is how to compare solutions. For example, as discussed in Section 10, it is currently not possible to automatically verify if research upholds pitfalls that occur when benchmarking flash. Similarly, many stores have been optimised for a particular case and device, making it hard to compare key-value solutions or estimate what solution will work best. Further on, various solutions have made use of different benchmarks, increasing the problem. This is a problem we expect will remain for now, and is something that needs to be addressed.

Future work could look into key-value stores for other storage media than block-based flash, such as persistent memory or *key-value SSDs* (KVSSDs). KVSSD devices are made specifically for key-value stores, so are especially interesting. Further research could also look into other types of key-value stores when used on flash, other than traditional persistent solutions, such as ephemeral key-value stores used for caching.

Future work could also look into how flash-based key-value stores fit into the entire ecosystem that they are used in. For example, if a database is used around key-value stores, or what tools interact with key-value stores. This is important to get a good picture of what to optimise key-value stores for.

Lastly, future work could look into other data stores and how they can be optimised for flash. This makes it less difficult to make an informed decision on what data store to pick apart from key-values. A related open problem is that it is at the moment difficult to test different solutions. Future work could look into how to do this efficiently to help with making a decision on the data store and the storage media.

14 Conclusion

We conclude by answering our research question: "What is the impact of flash storage on the design choices for key-value stores?". Considering the amount of flash-specific ideas that have been proposed, some already used widely in production, we can safely say that it has had a big impact. It has become common to store key-value stores on flash storage and to optimise these stores to get more use out of the device. To give a more precise answer to the research question, we will also answer each sub-question individually:

- RQ1 - "What is the current role of flash for key-value stores?": Flash is used in a variety of roles, ranging from the main storage medium of key-value stores, to a caching layer, to slow storage used for cold data. Generally, it is used to store a large amount of data that needs to remain fast to access. Flash is widely used for key-value stores by among others data centres, game services and webshops.

- RQ2 - "How has flash influenced the design of key-value stores over the decade?": We have seen various flash specific optimisations in this survey. Key-value stores have adapted to flash specific challenges such as wear levelling and device parallelism. We thus see that key-value stores have started to adapt to the idiosyncrasies of the storage medium. The design of various key-value stores is thus directly influenced by flash storage. Data structures and other design decisions, all the same, have evolved as a result of flash specifics.

- RQ3 - "What are the main challenges involved in using key-value stores on flash and how can they be mitigated?": The main challenges that we identified are:

reducing write amplification, reducing read amplification, reducing space amplification, reducing garbage collection overhead, reducing memory overhead, reducing CPU overhead and how to integrate flash in an architecture. There exist various optimisations that tackle a few of these challenges, but none that tackles them all. Generally, a trade-off needs to be made, where one challenge is solved at the expense of another. The main challenge is then deciding what trade-off is best for the use case of a particular key-value store.

- RQ4 - “How will flash contribute to key-value stores in the future?": Flash will continue to remain relevant for key-value stores. We expect key-value stores to keep storing the majority of their data on flash for years to come. We also expect new ideas to be proposed to further optimise key-value stores for flash storage.

Glossary

- AHCI: Advanced Host Controller Interface
- AMQ: approximate membership query
- AWA: auxiliary write amplification
- CoW: copy-on-write
- DB: database
- ECT: entropy coded trie
- FIFO: first in first out
- FLSM: fragmented LSM
- FS: file system
- FTL: Flash Translation Layer
- GC: garbage collector
- HDD: hard disk drive
- HFTL: host-managed FTL
- I/O: input/output
- KV: key-value
- LBA: logical block address
- LRU: least recently used
- LSM-tree: log-structured merge-tree
- MLC: multi-level cell
- mmap: memory-map
- MS: multi-stage
- NVM: non-volatile memory
- NVMe: non-volatile memory Express
- OCSSD: open-channel SSD
- PCI: Peripheral Component Interconnect
- PCIe: PCI Express
- PLC: penta-level cell
- RA: read amplification
- SaaS: Software as a Service
- SAS: Serial Attached SCSI
- SATA: Serial ATA
- SCM: storage class memory
- SDF: software defined flash
- SLC: single-level cell
- SMR: shingled magnetic recording
- SPDK: Storage Performance Development Kit
- SSD: solid state drive
- SSTable: sorted strings table
- syscall: system call
- TLC: triple-level cell
- VFS: virtual file system
- vLog: value log
- vTree: value tree
- QLC: quad-level cell
- ULL: ultra low latency
- WA: write amplification
- WAL: write-ahead log
- WL: wear levelling
- WOI: write optimised index
- ZNS: Zoned Namespace

References

[1] AHN, J.-S., SEO, C., MAYURAM, R., YASEEN, R., KIM, J.-S., AND MAENG, S. ForestDB: A fast key-value storage system for variable-length string keys. *IEEE Transactions on Computers 65*, 3 (2015), 902–915.

[2] ALSALIBI, A. I., MITTAL, S., AL-BETAR, M. A., AND SUMARI, P. B. A survey of techniques for architecting SLC/MLC/TLC hybrid Flash memory–based SSDs. *Concurrency and Computation: Practice and Experience 30*, 13 (2018), e4420.

[3] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *NSDI* (2010), vol. 10, pp. 29–29.

[4] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), pp. 1–14.

[5] ANDERSEN, D. G., AND SWANSON, S. Rethinking flash in the data center. *IEEE micro 30*, 04 (2010), 52–54.

[6] ANDERSON, P., BLACK, R., CERKAUSKAITE, A., CHATZIELEFTHERIOU, A., CLEGG, J., DAINTY, C., DIACONU, R., DREVINSKAS, R., DONNELLY, A., GAUNT, A. L., ET AL. Glass: A new media for a new era? In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018).

[7] ARITOME, S. *NAND flash memory technologies*. John Wiley & Sons, 2015.

[8] BALMAU, O., DIDONA, D., GUERRAOUI, R., ZWAENEPOEL, W., YUAN, H., ARORA, A., GUPTA, K., AND KONKA, P. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIXATC 17)* (2017), pp. 363–375.

[9] BALMAU, O., DINU, F., ZWAENEPOEL, W., GUPTA, K., CHANDHIRAMOORTHI, R., AND DIDONA, D. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIXATC 19)* (2019), pp. 753–766.

[10] BARROSO, L., MARTY, M., PATTERSON, D., AND RANGANATHAN, P. Attack of the killer microseconds. *Communications of the ACM 60*, 4 (2017), 48–54.

[11] BATSARAS, N., SALOUSTROS, G., PAPAGIANNIS, A., FATOUROU, P., AND BILAS, A. VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. *arXiv preprint arXiv:2003.00103* (2020).

[12] BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., MEDJEDOVIC, D., MONTES, P., SHETTY, P., SPILLANE, R. P., AND ZADOK, E. Don't thrash: how to cache your hash on flash. In *3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11)* (2011).

[13] BENSON, L., MAKAIT, H., AND RABL, T. Viper: an efficient hybrid PMem-DRAM key-value store. *Proceedings of the VLDB Endowment 14*, 9 (2021), 1544–1556.

[14] BEZ, R., CAMERLENGHI, E., MODELLI, A., AND VISCONTI, A. Introduction to flash memory. *Proceedings of the IEEE 91*, 4 (2003), 489–502.

[15] BHATTACHARYA, S., PRATT, S., PULAVARTY, B., AND MORGAN, J. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium* (2003), pp. 371–386.

[16] BHIMANI, J., YANG, J., YANG, Z., MI, N., GIRI, N. K., PANDURANGAN, R., CHOI, C., AND BALAKRISHNAN, V. Enhancing ssds with multi-stream: What? why? how? In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)* (2017), IEEE, pp. 1–2.

[17] BJØRLING, M. From open-channel SSDs to zoned namespaces. In *Proc. Linux Storage Filesyst. Conf.(Vault)* (2019), p. 1.

[18] BJØRLING, M., AGHAYEV, A., HOLMBERG, H., RAMESH, A., LE MOAL, D., GANGER, G. R., AND AMVROSIADIS, G. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 689–703.

[19] BJØRLING, M., GONZALEZ, J., AND BONNET, P. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association, pp. 359–374.

[20] BORTNIKOV, E., BRAGINSKY, A., HILLEL, E., KEIDAR, I., AND SHEFFI, G. Accordion: Better memory organization for LSM key-value stores. *Proceedings of the VLDB Endowment 11*, 12 (2018), 1863–1875.

[21] CAO, Z., DONG, S., VEMURI, S., AND DU, D. H. Characterizing, Modeling, and Benchmarking RocksDBKey-Value Workloads at Facebook. In *18th*

USENIX Conference on File and Storage Technologies (FAST 20) (2020), pp. 209–223.

[22] CATTELL, R. G. G. Scalable SQL and NoSQL data stores. *SIGMOD Rec. 39* (2011), 12–27.

[23] CHAI, Y., CHAI, Y., WANG, X., WEI, H., BAO, N., AND LIANG, Y. LDC: a lower-level driven compaction method to optimize SSD-oriented key-value stores. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), IEEE, pp. 722–733.

[24] CHAN, H. H., LIANG, C.-J. M., LI, Y., HE, W., LEE, P. P., ZHU, L., DONG, Y., XU, Y., XU, Y., JIANG, J., ET AL. Hashkv: Enabling efficient updates in KV storage via hashing. In *2018 USENIX Annual Technical Conference (USENIXATC 18)* (2018), pp. 1007–1019.

[25] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 1–26.

[26] CHATZIELEFTHERIOU, A., STEFANOVICI, I., NARAYANAN, D., THOMSEN, B., AND ROWSTRON, A. Could cloud storage be disrupted in the next decade? In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)* (2020).

[27] CHEN, H., RUAN, C., LI, C., MA, X., AND XU, Y. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021), pp. 17–32.

[28] CHEN, X., ZHENG, N., XU, S., QIAO, Y., LIU, Y., LI, J., AND ZHANG, T. KallaxDB: A Table-less Hash-based Key-Value Store on Storage Hardware with Built-in Transparent Compression. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)* (2021), pp. 1–10.

[29] CHOI, G., LEE, K., OH, M., CHOI, J., JHIN, J., AND OH, Y. A New LSM-style Garbage Collection Scheme for ZNS SSDs. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)* (2020).

[30] CHUNG, T.-S., PARK, D.-J., PARK, S., LEE, D.-H., LEE, S.-W., AND SONG, H.-J. A survey of flash translation layer. *Journal of Systems Architecture 55*, 5-6 (2009), 332–343.

[31] COMER, D. Ubiquitous B-tree. *ACM Computing Surveys (CSUR) 11*, 2 (1979), 121–137.

[32] CONWAY, A., GUPTA, A., CHIDAMBARAM, V., FARACH-COLTON, M., SPILLANE, R., TAI, A., AND JOHNSON, R. SplinterDB: Closing the bandwidth gap for nvme key-value stores. In *2020 USENIX Annual Technical Conference (USENIXATC 20)* (2020), pp. 49–63.

[33] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[34] CORNWELL, M. Anatomy of a solid-state drive. *Communications of the ACM 55*, 12 (2012), 59–63.

[35] DAIM, T. U., PLOYKITIKOON, P., KENNEDY, E., AND CHOOTHIAN, W. Forecasting the future of data storage: case of hard disk drive and flash memory. *Foresight* (2008).

[36] DAVOUDIAN, A., CHEN, L., AND LIU, M. A survey on NoSQL stores. *ACM Computing Surveys (CSUR) 51*, 2 (2018), 1–43.

[37] DAVOUDIAN, A., AND LIU, M. Big data systems: A software engineering perspective. *ACM Computing Surveys (CSUR) 53*, 5 (2020), 1–39.

[38] DAYAN, N., ATHANASSOULIS, M., AND IDREOS, S. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 79–94.

[39] DAYAN, N., AND IDREOS, S. Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data* (2018), pp. 505–520.

[40] DAYARATHNA, M., WEN, Y., AND FAN, R. Data center energy consumption modeling: A survey. *IEEE Communications Surveys & Tutorials 18*, 1 (2015), 732–794.

[41] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1414–1425.

[42] DEBNATH, B., SENGUPTA, S., AND LI, J. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data* (2011), pp. 25–36.

[43] DEBNATH, B. K., SENGUPTA, S., AND LI, J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *USENIX annual technical conference* (2010), pp. 1–16.

[44] DGRAPH. Badger. https://github.com/dgraph-io/badger, last accessed on 2022-02-10.

[45] DIDONA, D., IOANNOU, N., STOICA, R., AND KOURTIS, K. Toward a better understanding and evaluation of tree structures on flash SSDs. *Proceedings of the VLDB Endowment 14*, 3 (2020), 364–377.

[46] DIPERT, B., AND HEBERT, L. Flash memory goes mainstream. *IEEE spectrum 30*, 10 (1993), 48–52.

[47] DONG, M., ZHONG, H., SUN, B., BI, S., AND CAI, Y. SardineDB: A Distributed Database on the Edge of the Network. In *Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data* (2021), Springer, pp. 186–193.

[48] DONG, S., CALLAGHAN, M., GALANIS, L., BORTHAKUR, D., SAVOR, T., AND STRUM, M. Optimizing Space Amplification in RocksDB. In *CIDR* (2017), vol. 3, p. 3.

[49] DONG, S., KRYCZKA, A., JIN, Y., AND STUMM, M. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021), pp. 33–49.

[50] EISENMAN, A., CIDON, A., PERGAMENT, E., HAIMOVICH, O., STUTSMAN, R., ALIZADEH, M., AND KATTI, S. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 65–78.

[51] ESCRIVA, R., WONG, B., AND SIRER, E. G. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication* (2012), pp. 25–36.

[52] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies* (2014), pp. 75–88.

[53] FEVGAS, A., AKRITIDIS, L., BOZANIS, P., AND MANOLOPOULOS, Y. Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal 29* (2019), 273–311.

[54] GHEMAWAT, S., AND DEAN, J. LevelDB, 2011.

[55] GILAD, E., BORTNIKOV, E., BRAGINSKY, A., GOTTESMAN, Y., HILLEL, E., KEIDAR, I., MOSCOVICI, N., AND SHAHOUT, R. EvenDB: optimizing key-value storage for spatial locality. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 1–16.

[56] GRAEFE, G., AND KUNO, H. Modern B-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering* (2011), IEEE, pp. 1370–1373.

[57] GUPTA, A., TYAGI, S., PANWAR, N., SACHDEVA, S., AND SAXENA, U. NoSQL databases: Critical analysis and comparison. In *2017 International conference on computing and communication technologies for smart nation (IC3TSN)* (2017), IEEE, pp. 293–299.

[58] HAN, S., JIANG, D., AND XIONG, J. Splitkv: Splitting IO paths for different sized key-value items with advanced storage devices. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)* (2020).

[59] HARRINGTON, J. L. *Relational database design and implementation*. Morgan Kaufmann, 2016.

[60] HE, J., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The unwritten contract of solid state drives. In *Proceedings of the twelfth European conference on computer systems* (2017), pp. 127–144.

[61] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *International Symposium on Distributed Computing* (2008), Springer, pp. 350–364.

[62] HUEN, H., CHOI, C., AND BALAKRISHNAN, V. Performance and endurance enhancements with multi-stream ssds on apache cassandra. *Samsung Semiconductor* http://www.samsung.com/us/labs/collateral/index.html (2017).

[63] IDREOS, S., AND CALLAGHAN, M. Key-value storage engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (2020), pp. 2667–2672.

[64] IM, J., BAE, J., CHUNG, C., LEE, S., ET AL. PinK: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIXATC 20)* (2020), pp. 173–187.

[65] JIN, Y., TSENG, H.-W., PAPAKONSTANTINOU, Y., AND SWANSON, S. KAML: A flexible, high-performance key-value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), IEEE, pp. 373–384.

[66] JUNG, M.-G., LEE, C.-G., PARK, D., PARK, S., NOH, J., CHUNG, W., PARK, K., AND KIM, Y. GPUKV: an integrated framework with KVSSD and GPU through P2P communication support. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (2021), pp. 1156–1164.

[67] KAIYRAKHMET, O., LEE, S., NAM, B., NOH, S. H., AND CHOI, Y.-R. SLM-DB:Single-LevelKey-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019), pp. 191–205.

[68] KANG, D., JUNG, D., KANG, J.-U., AND KIM, J.-S. μ-tree: An ordered index structure for NAND flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software* (2007), pp. 144–153.

[69] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (2014).

[70] KANNAN, S., BHAT, N., GAVRILOVSKA, A., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 993–1005.

[71] KHAN, N., YAQOOB, I., HASHEM, I. A. T., INAYAT, Z., MAHMOUD ALI, W. K., ALAM, M., SHIRAZ, M., AND GANI, A. Big data: survey, technologies, opportunities, and challenges. *The scientific world journal 2014* (2014).

[72] KIM, H., YEOM, H. Y., AND SON, Y. An i/o isolation scheme for key-value store on multiple solid-state drives. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)* (2019), IEEE, pp. 170–175.

[73] KIM, S., AND SON, Y. Optimizing Key-Value Stores for Flash-Based SSDs via Key Reshaping. *IEEE Access 9* (2021), 115135–115144.

[74] KOH, S., LEE, C., KWON, M., AND JUNG, M. Exploring System Challenges of Ultra-Low Latency Solid State Drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)* (2018).

[75] KONDYLAKIS, H., DAYAN, N., ZOUMPATIANOS, K., AND PALPANAS, T. Coconut: A scalable bottom-up approach for building data series indexes. *arXiv preprint arXiv:2006.13713* (2020).

[76] KORONEN, C., ÅHMAN, M., AND NILSSON, L. J. Data centres in future european energy systems—energy efficiency, integration and policy. *Energy Efficiency 13*, 1 (2020), 129–144.

[77] KOURTIS, K., IOANNOU, N., AND KOLTSIDAS, I. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)* (2019), pp. 1–15.

[78] KRISH, K., ANWAR, A., AND BUTT, A. R. hats: A heterogeneity-aware tiered storage for hadoop. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2014), IEEE, pp. 502–511.

[79] KUDALE, A. B+ tree Preference over B Tree. *Chicago, USA http://www. academia. edu/11575258/B_tree_preference_over_B_trees*.

[80] KUSZMAUL, B. C. A comparison of fractal trees to log-structured merge (LSM) trees. *Tokutek White Paper* (2014).

[81] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[82] LANDSMAN, D., AND WALKER, D. AHCI and NVMe as interfaces for SATA Express™ Devices, 2013.

[83] LEE, C.-G., KANG, H., PARK, D., PARK, S., KIM, Y., NOH, J., CHUNG, W., AND PARK, K. iLSM-SSD: An intelligent LSM-tree based key-value SSD for data analytics. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2019), IEEE, pp. 384–395.

[84] LEE, J., OH, G., AND LEE, S.-W. Boosting Compaction in B-Tree Based Key-Value Store by Exploiting Parallel Reads in Flash SSDs. *IEEE Access 9* (2021), 56344–56353.

[85] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), pp. 1075–1086.

[86] LEPERS, B., BALMAU, O., GUPTA, K., AND ZWAENEPOEL, W. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 447–461.

[87] LI, B., SONG, N. Y., OU, L., AND DU, D. H. Can we store the whole world's data in DNA storage? In *Proceedings of the 12th USENIX Conference on Hot Topics in Storage and File Systems* (2020), pp. 15–15.

[88] LI, Y., LIU, Z., LEE, P. P., WU, J., XU, Y., WU, Y., TANG, L., LIU, Q., AND CUI, Q. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 673–687.

[89] LI, Y., TIAN, C., GUO, F., LI, C., AND XU, Y. ElasticBF: Elastic Bloom Filter with Hotness Awareness for Boosting Read Performance in Large Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)* (2019), pp. 739–752.

[90] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), pp. 1–13.

[91] LIU, C.-Y., CHANG, Y.-M., AND CHANG, Y.-H. Read leveling for flash storage systems. In *Proceedings of the 8th ACM International Systems and Storage Conference* (2015), pp. 1–10.

[92] LIU, L., AND ZHOU, K. PTierDB: Building Better Read-Write Cost Balanced Key-Value Stores for Small Data on SSD. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), IEEE, pp. 796–801.

[93] LU, G., NAM, Y. J., AND DU, D. H. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)* (2012), IEEE, pp. 1–11.

[94] LU, L., PILLAI, T. S., GOPALAKRISHNAN, H., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in SSD-conscious storage. *ACM Transactions on Storage (TOS) 13*, 1 (2017), 1–28.

[95] LUKKEN, C., AND TRIVEDI, A. Past, Present and Future of Computational Storage: A Survey. *arXiv preprint arXiv:2112.09691* (2021).

[96] LUO, C., AND CAREY, M. J. LSM-based storage techniques: a survey. *The VLDB Journal 29*, 1 (2020), 393–418.

[97] MA, K., LIU, M., LI, T., YIN, Y., AND CHEN, H. A low-cost improved method of raw bit error rate estimation for nand flash memory of high storage density. *Electronics 9*, 11 (2020), 1900.

[98] MANSOURI, Y., TOOSI, A. N., AND BUYYA, R. Data storage management in cloud environments: Taxonomy, survey, and future directions. *ACM Computing Surveys (CSUR) 50*, 6 (2017), 1–51.

[99] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND RANGASWAMI, R. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *2015 USENIX Annual Technical Conference (USENIXATC 15)* (2015), pp. 207–219.

[100] MEI, F., CAO, Q., JIANG, H., AND LI, J. SifrDB: A unified solution for write-optimized key-value stores in large datacenter. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 477–489.

[101] MENON, P., RABL, T., SADOGHI, M., AND JACOBSEN, H.-A. CaSSanDra: An SSD boosted key-value store. In *2014 IEEE 30th International Conference on Data Engineering* (2014), IEEE, pp. 1162–1167.

[102] MICHELONI, R., CRIPPA, L., AND MARELLI, A. *Inside NAND flash memories*. Springer Science & Business Media, 2010.

[103] NATH, S., AND KANSAL, A. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of the 6th international conference on Information processing in sensor networks* (2007), pp. 410–419.

[104] NGUYEN, T.-D., AND LEE, S.-W. Optimizing mongodb using multi-streamed ssd. In *Proceedings of the 7th International Conference on Emerging Databases* (2018), Springer, pp. 1–13.

[105] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014), pp. 471–484.

[106] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[107] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIXATC 16)* (2016), pp. 537–550.

[108] PAPAGIANNIS, A., SALOUSTROS, G., GONZÁLEZ-FÉREZ, P., AND BILAS, A. An efficient memory-mapped key-value store for flash storage. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), pp. 490–502.

[109] PARK, S., KIM, Y., URGAONKAR, B., LEE, J., AND SEO, E. A comprehensive study of energy efficiency and performance of flash-based ssd. *Journal of Systems Architecture 57*, 4 (2011), 354–365.

[110] PATRIZIO, A. IDC: Expect 175 zettabytes of data worldwide by 2025. *Network World* (2018).

[111] POUYANFAR, S., YANG, Y., CHEN, S.-C., SHYU, M.-L., AND IYENGAR, S. Multimedia big data analytics: A survey. *ACM computing surveys (CSUR) 51*, 1 (2018), 1–34.

[112] PRITCHETT, D. BASE: An Acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. *Queue 6*, 3 (2008), 48–55.

[113] QIAO, Y., CHEN, X., ZHENG, N., LI, J., LIU, Y., AND ZHANG, T. Closing the B-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. *arXiv preprint arXiv:2107.13987* (2021).

[114] QIN, M., REDDY, A. N., GRATZ, P. V., PITCHUMANI, R., AND KI, Y. S. KVRAID: high performance, write efficient, update friendly erasure coding scheme for KV-SSDs. In *Proceedings of the 14th ACM International Conference on Systems and Storage* (2021), pp. 1–12.

[115] RAINA, A., CIDON, A., JAMIESON, K., AND FREEDMAN, M. J. PrismDB: Read-aware log-structured merge trees for heterogeneous storage. *arXiv preprint arXiv:2008.02352* (2020).

[116] RAJU, P., KADEKODI, R., CHIDAMBARAM, V., AND ABRAHAM, I. PebblesDB: Building key-value stores using fragmented log-structured merge trees. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 497–514.

[117] RASMUSSEN, R. V., AND TRICK, M. A. Round robin scheduling–a survey. *European Journal of Operational Research 188*, 3 (2008), 617–636.

[118] REN, K., ZHENG, Q., ARULRAJ, J., AND GIBSON, G. SlimDB: A space-efficient key-value storage engine for semi-sorted data. *Proceedings of the VLDB Endowment 10*, 13 (2017), 2037–2048.

[119] RISCH, T. Introduction to NoSQL Databases. *NoSQL-Databases.pdf* (2015).

[120] ROTHERMEL, K., AND MOHAN, C. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions.* IBM Thomas J. Watson Research Division, 1989.

[121] ROTTENSTREICH, O., AND KESLASSY, I. The bloom paradox: When not to use a bloom filter. *IEEE/ACM Transactions on Networking 23*, 3 (2014), 703–716.

[122] RUSINKIEWICZ, M., AND SHETH, A. P. Specification and Execution of Transactional Workflows. *Modern database systems 1995* (1995), 592–620.

[123] SAITO, T., SATO, K., SATO, H., AND MATSUOKA, S. Energy-aware i/o optimization for checkpoint and restart on a nand flash memory system. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale* (2013), pp. 41–48.

[124] SEEGER, M., AND ULTRA-LARGE-SITES, S. Key-Value stores: a practical overview. *Computer Science and Media, Stuttgart* (2009).

[125] SHARMA, V., AND DAVE, M. Sql and nosql databases. *International Journal of Advanced Research in Computer Science and Software Engineering 2*, 8 (2012).

[126] SIVASUBRAMANIAN, S. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), pp. 729–730.

[127] STAVRINOS, T., BERGER, D. S., KATZ-BASSETT, E., AND LLOYD, W. Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2021), pp. 144–151.

[128] SUN, H., DAI, S., AND HUANG, J. Cascaded Write Amplification of LSM-tree-based Key-Value Stores underlying Solid-State Disks. *Microprocessors and Microsystems 78* (2020), 103217.

[129] SUN, H., LIU, W., HUANG, J., AND SHI, W. Co-kv: A collaborative key-value store using near-data processing to improve compaction for the lsm-tree. *arXiv preprint arXiv:1807.04151* (2018).

[130] SYED, A., GILLELA, K., AND VENUGOPAL, C. The future revolution on big data. *Future 2*, 6 (2013), 2446–2451.

[131] TOKUTEK, I. TokuDB: MySQL performance, MariaDB Performance, 2013.

[132] TSAI, W.-T., SHAO, Q., HUANG, Y., AND BAI, X. Towards a scalable and robust multi-tenancy saas. In *Proceedings of the Second Asia-Pacific Symposium on Internetware* (2010), pp. 1–15.

[133] TSIROGIANNIS, D., HARIZOPOULOS, S., AND SHAH, M. A. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), pp. 231–242.

[134] TULKINBEKOV, K., AND KIM, D.-H. CaseDB: Lightweight Key-Value Store for Edge Computing Environment. *IEEE Access 8* (2020), 149775–149786.

[135] VINÇON, T., HARDOCK, S., RIEGGER, C., OPPERMANN, J., KOCH, A., AND PETROV, I. Noftl-kv: Tackling write-amplification on kv-stores with native storage management. In *Advances in database technology-EDBT 2018: 21st International Conference on Extending Database Technology, Vienna, Austria, March 26-29, 2018. proceedings* (2018), University of Konstanz, University Library, pp. 457–460.

[136] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), pp. 1–14.

[137] WANG, Y., TAN, J., MAO, R., AND LI, T. Temperature-aware persistent data management for LSM-tree on 3-D NAND flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*, 12 (2020), 4611–4622.

[138] WU, C.-H., CHANG, L.-P., AND KUO, T.-W. An efficient B-tree layer for flash-memory storage systems. In *International Conference on Real-Time and Embedded Computer Systems and Applications* (2003), Springer, pp. 409–430.

[139] WU, S.-M., LIN, K.-H., AND CHANG, L.-P. KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2018), IEEE, pp. 563–568.

[140] XANTHAKIS, G., SALOUSTROS, G., BATSARAS, N., PAPAGIANNIS, A., AND BILAS, A. Parallax: Hybrid Key-Value Placement in LSM-based Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 305–318.

[141] XU, Q., SIYAMWALA, H., GHOSH, M., AWASTHI, M., SURI, T., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance characterization of hyperscale applicationson on nvme ssds. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2015), pp. 473–474.

[142] YANG, F., DOU, K., CHEN, S., HOU, M., KANG, J.-U., AND CHO, S. Optimizing NoSQL DB on flash: a case study of RocksDB. In *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)* (2015), IEEE, pp. 1062–1069.

[143] YANG, M.-C., CHANG, Y.-M., TSAO, C.-W., HUANG, P.-C., CHANG, Y.-H., AND KUO, T.-W. Garbage collection and wear leveling for flash memory: Past and future. In *2014 International Conference on Smart Computing* (2014), IEEE, pp. 66–73.

[144] YANG, Z., HARRIS, J. R., WALKER, B., VERKAMP, D., LIU, C., CHANG, C., CAO, G., STERN, J., VERMA, V., AND PAUL, L. E. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2017), IEEE, pp. 154–161.

[145] ZHAN, L., LU, K., CHENG, Z., AND WAN, J. Rangekv: An efficient key-value store based on hybrid dram-nvm-ssd storage structure. *IEEE Access 8* (2020), 154518–154529.

[146] ZHAN, L., ZHANG, Y., AND YU, K. Design and implementation of SCM and SSD based hybrid key-value store. In *Proceedings of the 2019 International Conference on Artificial Intelligence and Computer Science* (2019), pp. 566–572.

[147] ZHANG, J., LU, Y., SHU, J., AND QIN, X. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Transactions on Embedded Computing Systems (TECS) 16*, 5s (2017), 1–19.

[148] ZHANG, W., ZHAO, X., JIANG, S., AND JIANG, H. ChameleonDB: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems* (2021), pp. 194–209.