Optimizing Databases by Learning Hidden Parameters of Solid State Drives

Aarati Kakaraparthy Jignesh M. Patel
University of Wisconsin, Madison
{aaratik, jignesh}@cs.wisc.edu

Kwanghyun Park Brian P. Kroth
Microsoft Gray Systems Lab, Madison
{kwpark, bpkroth}@microsoft.com

ABSTRACT

Solid State Drives (SSDs) are complex devices with varying internal implementations, resulting in subtle differences in behavior between devices. In this paper, we demonstrate how a database engine can be optimized for a particular device by learning its hidden parameters. This can not only improve an application's performance, but also potentially increase the lifetime of the SSD. Our approach for optimizing a database for a given SSD consists of three steps: *learning* the hidden parameters of the device, *proposing rules* to analyze the I/O behavior of the database, and *optimizing* the database by eliminating violations of these rules.

We obtain two different characteristics of an SSD, namely the *request size profile* and the *location profile*, from which we learn multiple internal parameters. Based on these parameters, we propose rules to analyze the I/O behavior of a database engine. Using these rules, we uncover sub-optimal I/O patterns in SQLite3 and MariaDB when running on our experimental SSDs. Finally, we present three techniques to optimize these database engines: (1) *use-hot-locations* on SSD-S, which improves the SELECT operation throughput of SQLite3 and MariaDB by 29% and 27% respectively; it also improves the performance of YCSB on MariaDB by 1%-22% depending on the workload mix, (2) *write-aligned-stripes* on SSD-T, reduces the wear-out caused by SQLite3 write-ahead log (WAL) file by 3.1%, and (3) *contain-write-in-flash-page* on SSD-T, which reduces the wear-out caused by the MariaDB binary log file by 6.7%.

PVLDB Reference Format:
Aarati Kakaraparthy, Jignesh M. Patel, Kwanghyun Park, and Brian P. Kroth. Optimizing Databases by Learning Hidden Parameters of Solid State Drives. *PVLDB*, 13(4): 519–532, 2019.
DOI: https://doi.org/10.14778/3372716.3372724

1. INTRODUCTION

Solid State Drives (SSDs) are widely used persistent storage devices, typically built with NAND Flash [7] and more recently with 3D XPoint memory [2]. They are integrated circuit assemblies with no mechanical parts, and are faster than hard disks as a result [1, 21, 32]. The behavior of SSDs is greatly influenced by their hierarchical architecture (§2.1), and the properties of the storage medium used (§2.2). For instance, a distinctive feature of SSDs is the abundance of internal parallelism, which is a consequence of their hierarchical organization.

Behind the block interface of SSDs, the internal operation and the organization of flash memory varies from one device to another. There is no silver bullet when it comes to completely determining the internal operation of an SSD. Different manufacturers use different internal policies with hidden parameters, which results in a spectrum of device characteristics. However, external measurements can be used to study these characteristics, and can sometimes reveal information regarding the internal parameters of the device.

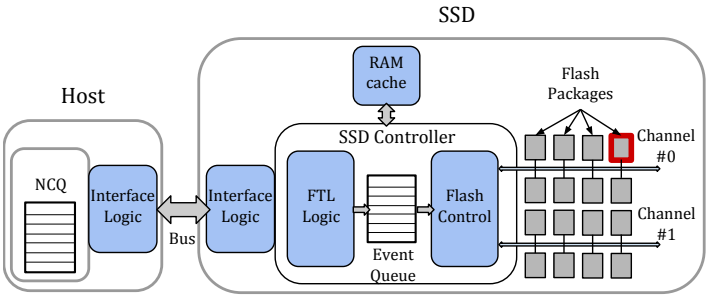
While previous work (§2.2) prescribes general guidelines to use SSDs effectively, it is possible to further optimize an application for a specific device by studying its unique characteristics. The aim of this paper is to describe ways to learn hidden parameters of an SSD, and demonstrate how these parameters can be used to optimize a database engine for a given device. The benefits of optimizing a database for an SSD are twofold: it can improve the immediate performance of the database engine through better utilization of the SSD's internal parallelism, and can have long-term benefits such as increasing the lifetime of the device.

To motivate why optimizing for a particular SSD could be useful, consider a datacenter with a million SSDs of the same make [49] running a common service. Let us assume that the average lifetime of an SSD in the fleet is one year [39]. A 1% improvement in lifetime would equal an additional 3.65 days for an individual SSD, but 3,650,000 machine days for the whole fleet. Thus, even the slightest improvement in lifetime (or performance) by optimizing the service for that particular make of SSD can cumulatively be very beneficial.

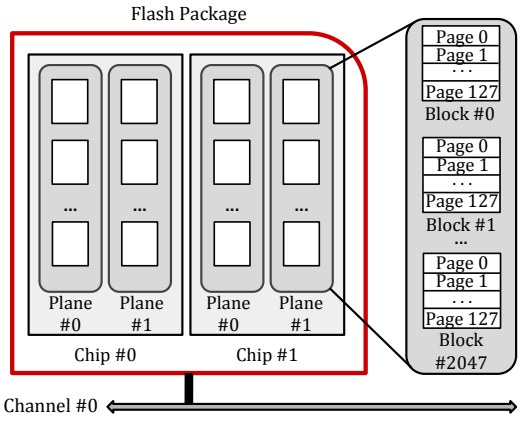
Our contribution towards optimizing a database system for a given SSD consists of the following three parts (§3):

1. ***Obtaining SSD characteristics and learning internal parameters*** (§3.1): We study two different characteristics of an SSD, namely the *request size profile* (§3.1.1) and the *location profile* (§3.1.2, [25]). From these characteristics, we learn five different internal parameters: the desirable write request sizes, the stripe size, the chunk size, the flash page size, and hot locations in the logical address space. We have performed experiments on SSDs from four different manufacturers, and the internal parameters learned have been summarized in Table 1.

Proceedings of the VLDB Endowment, Vol. 13, No. 4
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3372716.3372724



(a) The architecture and internal operation of an SSD. The Host issues commands to the SSD through the interface (NVMe, SATA, etc). The FTL processes the commands and issues events to the flash control, which operates the multiple internal channels (buses). The RAM cache controlled by the FTL is used for intermediate storage during the operation of the SSD. Together, the FTL and Flash Control constitute the SSD controller.



(b) Hierarchical organization of flash memory inside a flash package. Attached to each *channel* inside an SSD are multiple *flash packages*. Each flash package has multiple *chips*, which in turn have multiple *planes*. Each plane has multiple *flash blocks*, each of which is a collection of *flash pages* (size 2KB to 16KB).

Figure 1: The Architecture of an SSD and Hierarchical Organization of Flash Memory. The block interface is implemented by the Flash Translation Layer (FTL). Flash pages (typically 2KB to 16KB in size) and flash blocks (order of MBs) are the unit of read(write) and erase operations respectively, and are fixed for a device. Flash memory is organized hierarchically over channels, packages, chips, and planes (in that order).

2. ***Proposing rules to analyze the I/O patterns of a database engine*** (§3.2): Informed by the parameters learned, we propose rules (§3.2.1) to analyze the I/O behavior of a database when running on a particular SSD. These rules depend on the internal parameters, and thus vary between devices[1]. By identifying I/O calls that violate these rules, we uncover sub-optimal patterns that could potentially be corrected to improve performance.
3. ***Optimizing disk data structures to improve performance*** (§3.3): We examine the data structures used by the database engine, and describe techniques to modify them such that the sub-optimal I/O patterns found are eliminated. We present three techniques (§3.3), namely *use-hot-locations* (Fig. 9), *write-aligned-stripes* (Fig. 11), and *contain-write-in-flash-page* (Fig. 10).

In this paper, we have studied two popular open-source databases, namely SQLite3 [15] and MariaDB (with InnoDB storage module) [11,13]. While the former is a minimalistic database engine with uses spanning from mobile applications to data science platforms like pandas [4,16], the latter is widely deployed in multiple large-scale enterprises [3]. We apply the proposed rules to study these database engines, and present techniques to optimize them on our experimental SSDs. These techniques have been evaluated in §4.

Applying the technique *use-hot-locations* to optimize SQLite3 and MariaDB on SSD-S increased their SELECT operation throughput by 29% and 27% respectively in the presence of memory buffering (§4.1). In addition to this, we benchmarked MariaDB using YCSB [18], and obtained an improvement in performance ranging from 1%-22% for different workloads (Fig. 13b). On the other hand, the techniques *write-aligned-stripes* and *contain-write-in-flash-page* reduce the device wear out caused by the log files of SQLite3 and MariaDB by 3.1% and 6.7% respectively (§4.2). Thus we demonstrate that the knowledge of internal parameters can be used to tune a database engine for a given SSD.

[1]We refer to a particular make of an SSD as "a device".

2. BACKGROUND

In this section, we give an overview of the hierarchical architecture of SSDs (§2.1), followed by an in-depth discussion on their internal operation (§2.2). We also discuss some common recommendations for applications informed by the general structure and behavior of SSDs.

2.1 The Hierarchical Architecture of SSDs

Figure 1 shows the architecture of an SSD. The Flash Translation Layer (FTL) is a key component responsible for managing the resources inside the SSD (Fig. 1a). Internally, NAND flash cells are aggregated into larger units of *flash pages* (Fig. 1b), which usually range from 2KB to 16KB in modern SSDs [6,31,45]. Flash pages are further aggregated into *flash blocks*, which are typically in the order of MBs in size. Flash pages and blocks are the smallest units of read(write) and erase operations respectively inside an SSD (discussed further in §2.2).

This hierarchical organization of flash memory is typical to SSDs[2] (Fig. 1), and greatly influences their behavior. SSDs have multiple internal channels (i.e., buses) which can be operated independently, each of which is shared by multiple flash packages (also known as dies). Each flash package consists of chips, which in turn consist of multiple planes. Overall, we have four levels of internal parallelism, corresponding to each level of the storage hierarchy:

- **Channel-level**: Multiple internal channels (or buses) can be operated simultaneously and independently. Each channel is operated by at most one attached flash package at any time.
- **Package-level**: Access to all the packages on a single channel can be interleaved, and commands can be processed by packages simultaneously.

[2]Intel Optane SSD (SSD-I in Table 1) is not flash based. It is built with 3D XPoint Memory, but also has a hierarchical internal architecture with multiple channels [30].

Table 1: SSDs used in experiments and their internal parameters learned. In all expressions used, i is an integer. The desirable write request sizes and the stripe size are learned from the *request size profile* (§3.1.1), whereas the chunk size, hot locations and the flash page size are learned from the *location profile* (§3.1.2).

Label	Name	Capacity	Interface	Desirable Write Request Sizes	Stripe Size	Chunk Size	Hot Locations	Page Size
SSD-S	Samsung 960 EVO	512GB	NVMe	32KB×i	64KB	64KB	64KB×i+32KB	-
SSD-I	Intel Optane 905P	480GB	NVMe	1KB×i	-	4KB	4KB×i	4KB
SSD-T	Toshiba XG5	256GB	NVMe	64KB×i	64KB	4KB	4KB×i	4KB
SSD-M	Micron M500	120GB	SATA	64KB×i	64KB	4KB	4KB×i	4KB

- **Chip-level & Plane-level**: Chips inside a flash package, and planes within chips can be accessed simultaneously.

The choice of physical parameters like the size of flash pages, number of channels, as well as the logic of the FTL are trade secrets of SSD manufacturers. Although the specific details may vary from one make of SSD to another, the internal operation follows similar principles informed by the properties and organization of flash memory.

2.2 Internal Operation of SSDs and Common Recommendations for Applications

Three important low-level operations inside an SSD are *read*, *write/program*, and *erase*. Read and write operations are executed in units of *flash pages*. Unlike HDDs, flash memory is never overwritten directly; it has to be erased before writing again[3]. The erase operation is performed in units of *flash blocks*, and data is written sequentially in an erased block in units of flash pages. The number of write-erase operations that can be performed on flash memory are limited, thus causing the SSD to *wear out* over time.

The FTL is responsible for implementing the block interface of SSDs. This requires the FTL to physically store the data onto the flash memory[4], and also maintain the the mapping of the logical block address to the physical flash address (*logical-to-physical address mappings*) where data stored. Thus, the FTL plays a critical role of deciding the physical layout of data on flash memory, which is important in determining the immediate performance of the SSD.

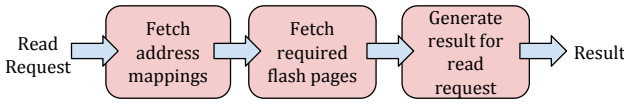
Figure 2 details the steps performed by an SSD to satisfy a read/write request. In both cases, *address translation* is performed by fetching the (logical-to-physical) address mappings corresponding to the logical address, followed by fetching the flash pages (if any) containing data. Accessing flash pages takes a significant portion of the time in satisfying a request. Hence the physical layout of data in the flash memory hierarchy, as well as the number of flash pages internally accessed is important in determining the immediate performance of an SSD (see Fig. 4 for an example).

SSD manufacturers have varying FTL implementations with different physical data layout policies. However, the following recommendations for applications have been made [20, 24, 25, 31, 37] based on the general operation of SSDs:

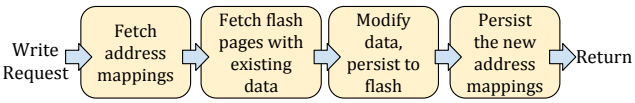
- ***Issue large or sequential write requests***. This allows the SSD to have more compact address mappings by storing information for larger logical units (*clustered pages* [23, 37] instead of flash pages).

[3]However, Intel's 3D XPoint Memory need not be erased before being overwritten.

[4]Aside from managing the physical layout of data, the FTL is also responsible for various background tasks like garbage collection and wear leveling [26].



(a) Satisfying a read request. The SSD performs address translation by fetching the necessary logical-to-physical address mappings. Once the physical address(es) is(are) obtained, the flash pages containing the required data are fetched. The result for the read request is assembled and returned.



(b) Satisfying a write request. The SSD first performs address translation by fetching the necessary logical-to-physical address mappings. If the write request overwrites existing data, the flash pages containing the existing copy of the data are fetched. The updated data is assembled and written to flash memory. The address mappings corresponding to the updated data are persisted and the older mappings invalidated.

Figure 2: Steps performed internally by an SSD to satisfy a read and write request.

- ***Issue write requests with temporal locality***. This can result in faster address translation for subsequent read requests, as the address mappings are likely to be co-located and already present in the SSD's RAM cache.
- ***Issue large or multiple concurrent read requests***. Large read requests are likely to span over multiple sub-units inside an SSD, utilizing internal parallelism. The same holds true for multiple concurrent read requests.
- ***Issue large or multiple concurrent write requests***. This leads to better utilization of the internal parallelism of the SSD, and better performance as a result.
- ***Issue sequential read requests***. SSDs often prefetch data internally [46], resulting in better performance for sequential accesses.
- ***Issue aligned write requests that are multiples of the flash page size***. For these requests, the SSD can skip fetching the existing pages being overwritten completely (step 2 in Fig. 2b). In contrast, requests smaller than the flash page size would need to be padded to fill the page causing *write amplification*, as more flash memory is being used than necessary. This can also cause *read amplification* for subsequent read requests.

The last recommendation requires knowledge of the flash page size, which is a *hidden* internal parameter of an SSD and is not readily shared by the manufacturers. Our aim in this paper is to go beyond generic recommendations, and make recommendations specific to a device by learning its characteristics through measurements (see §3.2).

Experiment 1 The Request Size Profile of an SSD

```
1: procedure GetRequestSizeProfile
2:     /* Generate files with increasing request sizes */
3:     fileSize ← 1GB
4:     filesCreated ← []
5:     numFiles ← 10
6:     for fId in 1:numFiles do
7:         requestSize ← 2^(fId−1) KB
8:         filename ← {requestSize}-seq
9:         file ← open(filename)
10:        filesCreated.append(filename)
11:        numReqs ← fileSize/requestSize
12:        for i in 1:numReqs do
13:            data ← malloc(requestSize)
14:            offset ← (i − 1) × requestSize
15:            write(file, data, offset)
16:            fsync(file)
17:
18:    // Issue read requests to the files in a random
19:    // order to avoid read-ahead inside the SSD
20:    readReqSize ← 1MB
21:    numReadReqs ← fileSize/readReqSize
22:    randOrder ← random_shuffle(1 : numReadReqs)
23:    for filename in filesCreated do
24:        file ← open(filename)
25:        latencies ← [ ]
26:        for idx in randOrder do
27:            offset ← (idx − 1) × readReqSize
28:            startTime ← time.now()
29:            data ← read(file, readReqSize, offset)
30:            endTime ← time.now()
31:            latencies.append(endTime − startTime)
32:        plot(latencies, filename)
```

3. OPTIMIZING DATABASES FOR AN SSD

In this section, we describe our approach towards optimizing databases for an SSD. We first measure the SSD's characteristics and learn some internal parameters (§3.1). Informed by these parameters, we describe rules to identify any sub-optimal I/O requests issued by the database engine (§3.2). Finally, we propose techniques to eliminate these sub-optimal I/O requests, and improve the performance of the database engine on the SSD (§3.3). The SSDs used for experiments in this section, as well as their parameters learned have been summarized in Table 1. SSD-S, SSD-T, and SSD-M are NAND flash based, whereas SSD-I is built with 3D XPoint memory.

3.1 Learning SSD Parameters

We study two types of characteristics of an SSD, which help us infer various internal parameters. First, we describe how to obtain the *request size profile* (§3.1.1), from which we learn the *desirable write request sizes* and the *stripe size* of an SSD. Next, we obtain the *location profile* (§3.1.2) of the SSDs, from which we identify their respective *chunk sizes*, *hot locations*, and *flash page sizes*.

3.1.1 The Request Size Profile

Experiment 1 describes how to obtain the request size profile of an SSD. We start by creating files of size 1GB with sequential write requests of sizes ranging from 1KB to 512KB (filename *512kb-seq* refers to a 1GB file created with sequential write requests of size 512KB; likewise for other request sizes). Following this, the latency of read requests to all the files is measured. This experiment helps determine which write request sizes are desirable for a given SSD; a higher latency of read requests indicates an undesirable write request size during file creation.

Experiment 2 The Location Profile of an SSD

```
1: procedure GetLocationProfile
2:     filename ← 512KB-seq
3:     file ← open(filename)
4:     fileSize ← 1GB
5:     chunkSizeMin ← 4KB
6:     chunkSizeMax ← 512KB
7:     offsetUnit ← 1KB
8:     expChunkSize ← chunkSizeMin
9:     while expChunkSize <= chunkSizeMax do
10:        numChunks ← fileSize/expChunkSize
11:        randChunks ← random_shuffle(1:numChunks)
12:        numOffsetGroups ← chunkSize/offsetUnit
13:        for i in 0:(numOffsetGroups-1) do
14:            latencies ← []
15:            for j in randChunks do
16:                chunkOffset← (j − 1) × chunkSize
17:                offset ←chunkOffset+i× offsetUnit
18:                startTime ← time.now()
19:                data ← read(file, chunkSize,offset)
20:                endTime ← time.now()
21:                latencies.append(endTime − startTime)
22:            plot(expChunkSize, latencies, offsetGroup=i)
23:        expChunkSize ← 2 × expChunkSize
```

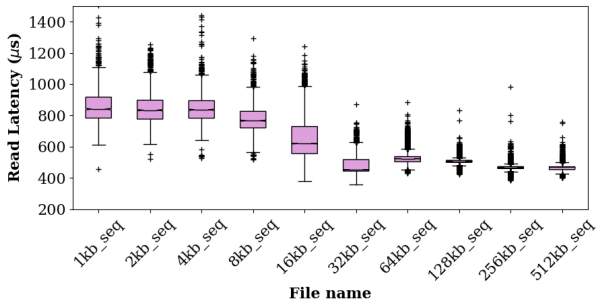
Fig. 3 shows the request size profile of all the SSDs, and we find that different devices have different characteristics. For a given SSD, we identify the request size at which minimum latency is attained, and all sizes greater than that, as *desirable write request sizes*. Two factors contribute to higher latency of a read request, namely read/write amplification and higher address translation time. The request size at which the lowest latency is obtained suggests absence of these factors, and is thus desirable. Thus, for SSD-S, request sizes 32KB and above are desirable, whereas for SSD-T and SSD-M, the desirable request sizes are 64KB and above.

Next, we attempt to learn the *stripe size* of the SSDs. We define the stripe size as the unit of decision of physical layout inside an SSD. Two (aligned) stripes of data have similar layout in terms of the sub-units occupied at each level of the flash hierarchy. The latency of read requests depends on the internal parallelism utilized inside the SSD, which in turn depends on the physical layout of data. Thus, similar latency of read requests indicates a similarity in physical layout of data. Therefore, for SSD-T, files *64kb-seq* to *512kb-seq* have similar physical layout, indicating that 64KB is the stripe size. By a similar argument, the stripe size of SSD-S and SSD-M is also 64KB.

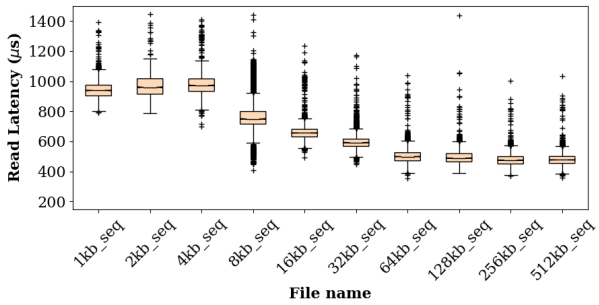
Unlike the others, SSD-I has constant latency for all the files. Flash-based SSDs can incur read/write amplification, as flash pages are units of read/write operations. However, unlike flash memory, Intel's 3D XPoint memory is byte-addressable and need not be erased before overwriting, and this is perhaps the reason behind the difference in SSD-I's behavior. Although we find that none of the request sizes are particularly undesirable for SSD-I, this experiment is insufficient to learn its stripe size.

3.1.2 The Location Profile

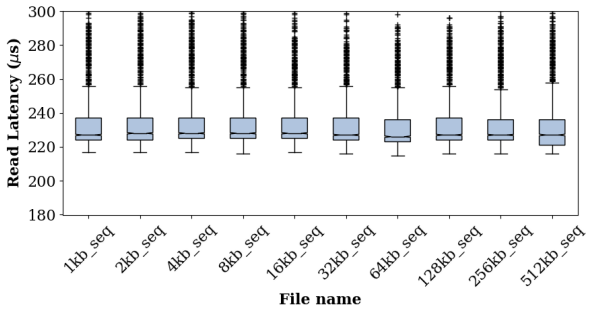
We define the *chunk size* of an SSD as the amount of contiguous data stored on a single channel. Due to the hierarchical architecture of SSDs, the latency of read requests (of chunk size) can vary at different logical address locations. Fig. 4 shows the multiple cases that can arise. If a request



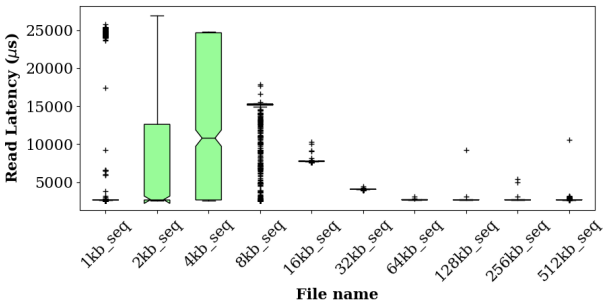
(a) SSD-S. Request sizes 32KB and up are desirable. The latency becomes roughly constant after request size 64KB, indicating that it is the stripe size.



(b) SSD-T. Request sizes 64KB and up are desirable. The latency becomes roughly constant after request size 64KB, indicating that it is the stripe size.

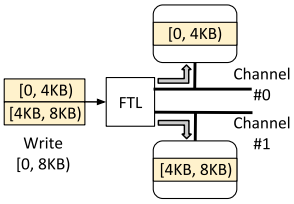


(c) SSD-I. All request sizes are desirable. This experiment is insufficient to learn the stripe size of this SSD.

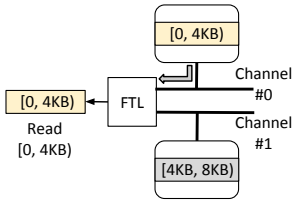


(d) SSD-M. Request sizes 64KB and up are desirable. The latency becomes roughly constant after request size 64KB, indicating that it is the stripe size.

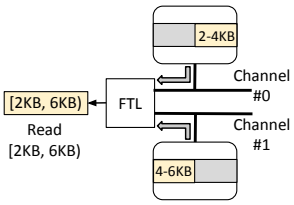
Figure 3: The Request Size Profile. We learn the *stripe size* and *desirable request sizes* from Experiment 1.



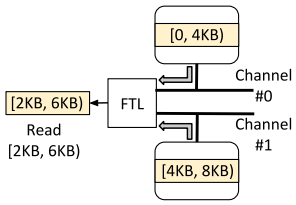
(a) An example of an 8KB write request to an SSD with chunk size 4KB. Two chunks of 4KB are stored on each channel.



(b) A 4KB read request aligning with a chunk; data on Channel #0 alone is accessed.



(c) A 4KB read request offset into a chunk. If the flash page size is 2KB, required pages are accessed in parallel, which will be faster than Fig. 4b.



(d) A 4KB read request offset into a chunk. If the flash page size is 4KB, both the chunks(pages) will be fetched. Will be slower than Fig. 4b.

Figure 4: Impact of channel-level parallelism on the latency of read requests at different logical address locations. If the flash page size is smaller than the chunk size, channel-level parallelism reduces latency by doubling the bandwidth (Fig. 4c). Otherwise, it increases latency due to read amplification (Fig.4d).

internally spans over multiple channels, its latency can be relatively lower or higher depending on the flash page size and chunk size of the SSD. This is the motivation behind Experiment 2, which is inspired by Chen et al. [25].

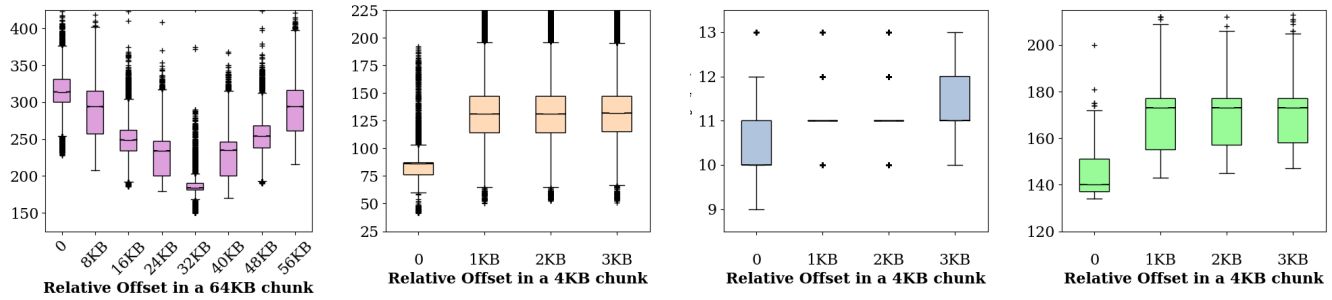
In the experiment, we repeatedly guess a chunk size and issue read requests (of chunk size) at different logical address locations, giving rise to the location profile of an SSD (for that chunk size). We define an *offset group* as the group of logical addresses with the same relative offset into a chunk[5]. If a significant variation in latency between different offset groups is observed, it indicates an influence of channel-level parallelism in some form.

Fig. 5 shows the result of this experiment. For SSD-S, we observe that for a chunk size of 64KB, offset group 32KB has 39% lower latency than offset group 0. Thus, the 32KB location group constitutes the set of *hot locations* on SSD-S. This behavior is similar to Fig. 4c, where channel-level parallelism helps reduce latency. On the other hand, the remaining SSDs have behavior similar to Fig. 4d with minimum latency at offset group 0, with a flash page size and chunk size of 4KB. We find that the reduction in latency at hot locations varies from one device to another, and this has been reported in Fig. 5.

3.2 Rules to Analyze Database I/O Patterns

In the previous section, we described how to learn some internal parameters of an SSD. In this section, we use these parameters to propose rules for applications when running on a given SSD (§3.2.1).

[5]For example, offset group 1KB for chunk size 4KB refers to all logical address locations of the form 4KB×i+1KB.



(a) SSD-S. Latency of reads at offset group 32KB is 39% lower than offset at group 0. The chunk size is 64KB, and the hot locations are at offset group 32KB.

(b) SSD-T. Latency of read requests at offset group 0 is 38% lower compared to others. The chunk size is 4KB, and the hot locations are at offset group 0.

(c) SSD-I. Latency of read requests at offset group 0 is 9% lower compared to others. The chunk size is 4KB, and the hot locations are at offset group 0.

(d) SSD-M. Latency of read requests at offset group 0 is 20% lower compared to others. The chunk size is 4KB, and the hot locations are at offset group 0.

Figure 5: The Location Profile. We learn the *chunk size* and *hot locations* of the SSDs from Experiment 2.

We first run a benchmark experiment (§3.2.2) to obtain the I/O patterns of a database engine, followed by applying the rules to analyze the I/O requests (§3.2.3 and §3.2.4). Requests that violate the rules are considered sub-optimal, and identifying such requests reveals opportunities for improving the performance of a database engine on a particular SSD. Thus, by using this approach, we study SQLite3 and MariaDB on two of our experimental SSDs, namely SSD-S and SSD-T, and we find sub-optimal I/O patterns in both of these database engines.

3.2.1 Rules to Identify Sub-Optimal I/O Requests

We propose the following rules for applications to follow when issuing I/O requests to a particular SSD. Requests violating these rules are considered sub-optimal.

- ***Rule 1***: *Issue write requests that are multiples of the minimum desirable write request size.* Requests not obeying this rule either cause write amplification, or require more flash memory for storing logical-to-physical address mappings.
- ***Rule 2***: *Issue read requests of chunk size at hot locations whenever possible.* Hot locations in the logical address space provide significantly lower latencies, and the application should use this advantage if it can.
- ***Rule 3***: *Issue write requests aligned with stripe boundaries, preferably of stripe size.* Overwriting stripes completely is desirable, as any existing data and address mappings can be invalidated without fetching additional flash pages. Also, unaligned requests can disturb the regularity of hot locations (discussed further in Rule 4).
- ***Rule 4***: *Do not issue unaligned write requests of chunk size.* The distribution of chunks over channels determines where hot locations occur in the logical address space. Issuing write requests not aligned with chunks can destroy the regularity of hot locations.
- ***Rule 5***: *Issue write requests such that the number of flash pages modified are minimum.* For instance, write requests smaller than the flash page size will internally be padded up to a page, and thus require at least a single flash page. However, a small write request spanning two flash pages will require modifying both the pages, causing greater write amplification than necessary, and should be avoided.

It should be noted that these rules help with maximizing the performance of the *SSD alone*. However, the requirements of the application also need to be considered for an overall improvement in performance. For instance, it would be incorrect to conclude that the database engine should always issue an I/O of 64KB (stripe size) instead of say, 16KB. Doing so might entail more I/O time than necessary and could degrade the overall performance.

3.2.2 The Benchmark Experiment

We run a benchmark experiment and record the I/O calls issued by the database engine for subsequent analysis using the above rules. This experiment helps isolate the insert/select operation throughput of the database engine as a measure of its performance. The experiment starts with an initial database containing 10 million key-value pairs in a single table. The key and value sizes are 32-byte and 100-byte respectively, and the initial size of the data is about 1.2GB. The experiment involves a single client running multiple iterations, with each iteration containing the following two phases:

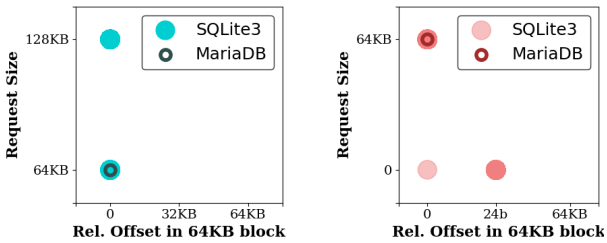
1. Insert 50,000 new key-value pairs into the database in a random order.
2. Run 50,000 select queries generated randomly on the set of keys present in the database.

The size of the database increases by about 6MB (50,000 × 132B) during an iteration. The same workload has been run in all cases by using a fixed random seed. All the select operations are run in the same transaction, whereas the number of insert operations per transaction varies (details regarding this have been specified wherever applicable).

This experiment is similar to SQLite's benchmarking of its backend library [10]. However, informed by previous work [43], we perform operations on a non-empty database instead. Required measurements (like measuring insert/select operation throughput, recording I/O calls using *strace* [12], etc.) are made during the experiment.

3.2.3 Analyzing the I/O behavior of B$^+$-Tree Indices

We study the I/O calls issued by SQLite3 and MariaDB to their primary index database files, to find any sub-optimal requests violating the rules (§3.2.1) on SSD-S. In both of these systems, the primary index file is structured as a B$^+$-Tree [9,14], and is divided into *database pages*. Each database



(a) Write requests of sizes 64KB and 128KB are made aligned with a 64KB block (rel. offset 0). These correspond to updates to database pages.

(b) Aligned read requests of size 64KB to the database pages are made. The remaining small requests are issued by SQLite3 to access its database header fields.

Figure 6: I/O profile of SQLite3 and MariaDB B$^+$-Tree index files during an iteration of the benchmark experiment for a B$^+$-Tree page size of 64KB. Pages are laid out contiguously in the index file, resulting in aligned requests that are multiples of 64KB. Both the databases violate Rule 2, as read requests are not issued to hot locations on SSD-S.

page corresponds to a node in the B$^+$-tree (Fig. 9a), and the leaf nodes contain data rows.

We investigate the I/O behavior of both the database engines for a B$^+$-Tree (database) page size of 64KB. This matches the stripe size of most of our SSDs, which will be useful for a more comprehensive discussion involving the proposed rules. A single iteration of the benchmark experiment (§3.2.2) has been run with each insert operation in a separate transaction. Both database engines have been run in their default journaling modes (i.e, rollback journaling for SQLite3, and undo/redo logging in MariaDB w/InnoDB).

Fig. 6a shows the write requests made by both database engines to their corresponding primary index files. We see that all the write requests are issued at aligned offsets, and are a multiple of the stripe size of SSD-S (64KB). Thus, none of the rules have been violated while issuing write requests.

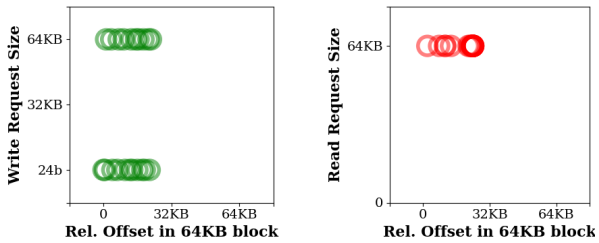
Fig. 6b shows the read requests issued to the B$^+$-tree index file by both the engines. The read requests of size 64KB at aligned offsets correspond to accessing pages from the index file. These requests violate Rule 2, as they are not issued to hot locations on SSD-S. The smaller read requests of size 100B and 4B are issued by SQLite3 to its database header. These requests are much smaller than a flash page and will cause read amplification. MariaDB doesn't issue any small read requests to its B$^+$-Tree database file.

3.2.4 *Analyzing the I/O Behavior of Log Files*

We consider the logging behavior of SQLite3 and MariaDB on SSD-T. We have studied SQLite3 in its write-ahead logging (WAL) mode[6] [17], whereas we run MariaDB in its default mode involving undo/redo logging.

Fig. 7a show SQLite3's write requests to the WAL file. The 64KB requests correspond to writing dirty database pages to the log, and the smaller write requests of 24B (which violate Rule 1) are for writing the log frame headers (Fig. 11a). Both these requests are increasingly unaligned with stripe boundaries, and violate Rule 3. All read requests (Fig. 7b) are of size 64KB, and are increasingly offset into

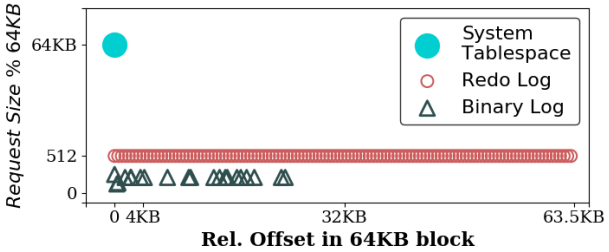
[6]SQLite3 in the rollback journal mode issues a mix of small and large writes to the log, similar to the WAL mode (see Fig. 7a). However, we only study the WAL file as a representative example.



(a) Two types of write requests are issued: 24B requests for frame headers, and 64KB requests to write dirty database pages to the log.

(b) Read requests of size 64KB are issued to dirty database pages (size 64KB) during checkpointing. Note that requests are unaligned.

Figure 7: I/O profile of the SQLite3 WAL log file in WAL mode during an iteration of the benchmark experiment for a B$^+$-tree page size of 64KB. The log file is organized in frames, where a frame contains a header (24B) and a dirty page (64KB). Rules 1 & 3 are violated by the write requests on SSD-T.



(a) Write requests to the system tablespace file are multiples of 64KB (stripe size for SSD-S) aligning with internal stripes, and don't violate any rules. The write requests to the redo log are of size 512B (aligned with 512B boundaries), and violate Rules 1 & 3. The binary log on the other hand contains small write requests of varying sizes, which violate Rules 1, 3, and 5.

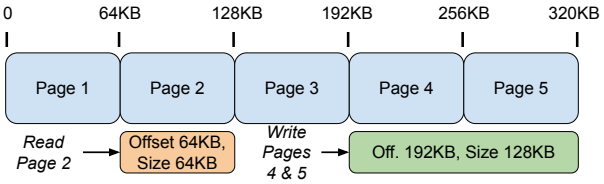
Figure 8: Write profile of MariaDB/InnoDB logs. The logs are contained in three files: (1) the System Tablespace file contains the undo log, doublewrite buffer, and the change buffer, (2) the Redo Log, and (3) the Binary Log for database replication. Read requests to these files follow a similar pattern.

the stripe boundary. These requests are not issued at a 1KB boundary, and will cause read amplification inside the SSD.

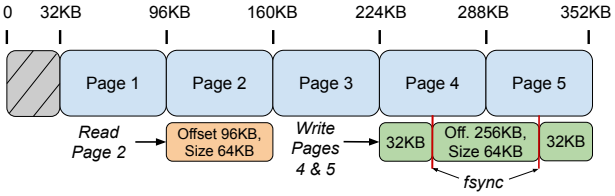
Compared to SQLite3, MariaDB has multiple log files to provide MVCC (undo logs), ensure crash recovery (redo logs) , avoid broken pages (doublewrite buffer), and provide log shipping to replicas (the binary log). These logs are contained in different tablespace files described in Fig. 8, and the I/O requests to these files are discussed below.

1. ***The System Tablespace (ibdata)***: Write requests of size 64KB aligning with stripe boundaries are issued, and none of the rules are violated[7].
2. ***The Redo Log (ib_logfile)***: Write requests of size 512B are issued at offsets aligning with 512B boundary (thus contained in a single flash page). Rules 1 & 3 are violated.
3. ***The Binary Log ({hostname}-bin)***: Small write requests of varying sizes at varying offsets are issued to this file. Rules 1, 3, and 5 are violated.

[7]Rule 2 would have been violated on SSD-S, but not on SSD-T as its hot locations are at offset group 0. This file's layout is similar to the B$^+$-Tree index file (Fig. 9a). However, we do not recommend applying the technique *use-hot-locations* to this file (see §4.1.3).

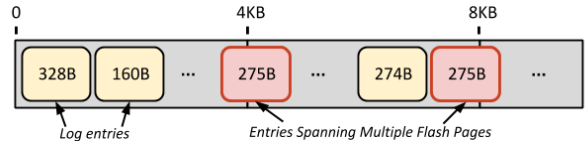


(a) The original layout of the database file. Pages are laid out contiguously, aligning with 64KB chunks. Read and write requests are issued to addresses at offset group 0.

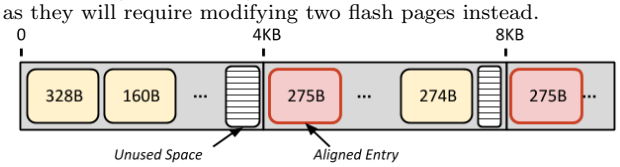


(b) The modified layout of the database file. All database pages are offset by 32KB, such that read requests to pages are issued to the 32KB offset group which are the hot locations for SSD-S. Write requests continue to align with a 64KB boundary to retain the internal chunk pattern (this is ensured by *fsync*, see §4.1.2).

Figure 9: *use-hot-locations* on SSD-S. In both SQLite3 and MariaDB, the original layout of the B$^+$-Tree database file contains contiguous aligned pages of size 64KB. Hot locations on SSD-S can be utilized if the pages are offset by 32KB. Write requests to pages need to be modified to satisfy Rule 3.



(a) The original layout of the InnoDB binary log file containing entries of varying sizes. Each entry internally requires the SSD to write a single flash page. However, entries spanning multiple flash pages (shown in red), will cause greater write amplification, as they will require modifying two flash pages instead.

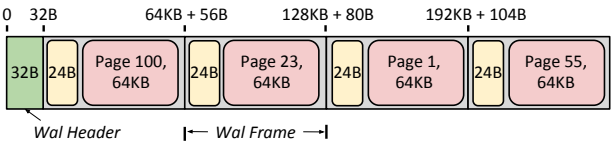


(b) The modified layout of the InnoDB binary log file. Entries spanning multiple flash pages are offset to align with the next page boundary. This results in a small varying amount of unused space at the end of each flash page, but reduces the write amplification caused by these entries to a single flash page.

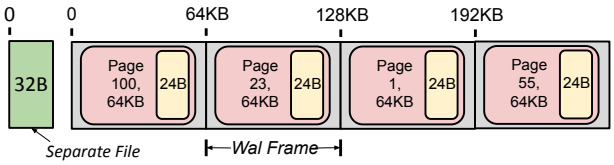
Figure 10: *contain-write-in-flash-page* on SSD-T (flash page size 4KB). The MariaDB/InnoDB binary log file has been modified to eliminate log entries spanning over multiple flash pages. This reduces the write-amplification caused by the database engine.

3.3 Techniques to Optimize Performance

Having studied the I/O behavior of SQLite3 and MariaDB, we now propose techniques to improve their performance by eliminating sub-optimal I/O patterns. This is achieved by examining the data-structures used and making modifications to them to obtain the desired I/O behavior, while also considering the database engine's performance. We propose the following techniques:



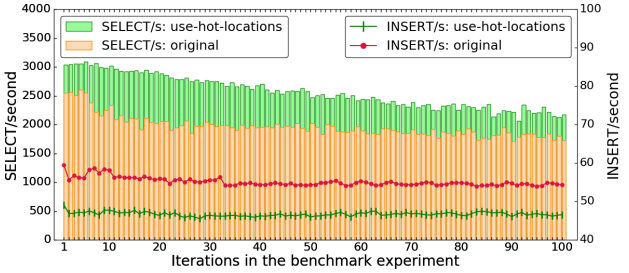
(a) The original layout of the WAL log file. The file has multiple frames, each with a frame header (24B) and a dirty page (64KB). As the log frame is not a multiple of stripe size (64KB), read/write requests become increasingly unaligned and violate Rule 3 (Fig. 7).



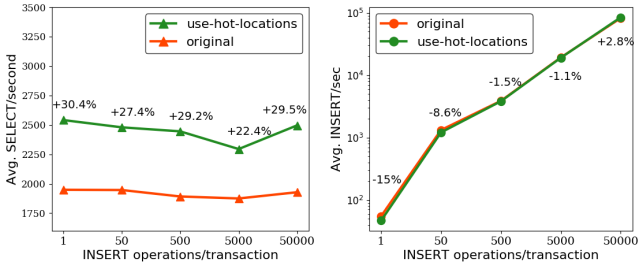
(b) The modified layout of the WAL log file. The frame header is a small amount of memory, and can be included in the database page without reducing the capacity of the page by much. The new frame size is 64KB, which is equal to the stripe size. The log header of 32B is stored in a separate file to ensure that the log frames align with stripe boundaries. Writing the file header also causes write amplification, and is a one-time unavoidable cost.

Figure 11: *write-aligned-stripes* on SSD-T. The SQLite3 WAL log file originally contains frames of size 64KB+24B. We include the 24B frame header in the page, to issue write requests of stripe size.

1. ***use-hot-locations* on SSD-S**: Fig. 9a shows the original layout of the B$^+$-Tree database file of both SQLite3 and MariaDB, where database pages align with internal stripes of SSD-S. This violates Rule 2 (see Fig. 6b), as read requests are issued at stripe boundaries instead of the hot locations (32KB offset group on SSD-S). To place the pages at hot locations, we offset them by 32KB in the modified layout of the file (Fig. 9b). However, in order to comply with Rule 3, write requests should continue to align with stripe boundaries, as shown in Fig. 9b. This technique is ineffective on the remaining SSDs as their hot locations are at offset group 0 (Rule 2 isn't violated).
2. ***write-aligned-stripes* on SSD-T**: The SQLite3 WAL file (Fig. 11a) has a 32B header followed by multiple *frames*, each of which has a 24B frame header and a 64KB dirty page. These small header fields result in write amplification, as well as make the read/write requests to the dirty pages increasingly unaligned, thus violating Rules 1 & 3 (Fig. 7). We propose including the frame header at the end of the database page (Fig. 11b), and writing the WAL header (of 32B) to a separate file. This eliminates all violations of rules on SSD-T, as shown in Fig. 11b. This technique will be effective on SSD-S and SSD-M as well, as they have the same stripe size as SSD-T.
3. ***contain-write-in-flash-page* on SSD-T**: The MariaDB redo log and binary log both violate Rules 1 & 3 (Fig. 8), as the requests are small and unaligned. However, issuing larger write requests would increase the latency of commit operation, and degrade the DB engine's performance. Additionally, the binary log also violates Rule 5, as shown in Fig. 10a. This violation can be eliminated by modifying the layout of the binary file as in Fig. 10b, where entries spanning multiple flash pages are offset to align with the next page boundary. This technique will also be effective on SSD-M, with a flash page size of 4KB.



(a) Select and Insert operation throughput over the iterations in a single run of the benchmark experiment with every insert operation in a separate transaction. Total 100 iterations are executed with 50,000 select and insert operations issued in each iteration. For select(insert) operations, we see that the performance *use-hot-locations* is consistently higher(lower) compared to the original. The average increase(decrease) in throughput is 30.4%(15.2%) over all the iterations. The loss in throughput of insert operations is caused by the additional *fsync* (Fig. 9b).



(b) Select and Insert operation throughput over multiple runs of the benchmark experiment with varying granularity of transactions in the insert phase. The gain/loss in throughput during each run has been indicated. For select operations, we obtain an almost constant improvement, with a median increase of 29.2%. For insert operations, as we increase the number of insert operations/transaction over the runs, the overhead of the additional *fsync* call becomes amortized and is compensated by the benefit of faster seek time. The loss in insert throughput falls under 2% from 500 insert operations/transaction onwards.

Figure 12: The performance of SQLite3 with *use-hot-locations* on SSD-S. The B$^+$-Tree index file is modified such that pages are stored at hot locations (Fig. 9b). This leads to a reduction in latency while accessing pages, and a median increase of 29.2% in select operation throughput as a result. The overhead of the additional *fsync* (see Fig. 9b and §4.1.2) leads to a 15% decrease in throughput of insert operations, but is compensated by faster seek time as we increase the number of insert operations per transaction.

4. EVALUATION

We evaluate the proposed techniques, namely *use-hot-locations* (§4.1), *write-aligned-stripes* (§4.2), and *contain-write-in-flash-page* (§4.3). While the former technique improves the immediate performance of both the database engines on SSD-S, the latter two eliminate write amplification and increase the lifetime of SSD-T.

4.1 use-hot-locations on SSD-S

We apply the technique *use-hot-locations* (Fig. 9) to both SQLite3 and MariaDB, and modify the layout of the B$^+$-Tree database file as shown in Fig. 9b. Improvement in performance is measured as the increase in operation throughput when running the benchmark experiment (§3.2.2) on SSD-S. We first describe our experimental setup (§4.1.1), followed by discussing the performance of SQLite3 (§4.1.2) and MariaDB (§4.1.3) with the modified database file layout, and finally summarize our observations (§4.1.4).

4.1.1 Experimental Setup

As before, SQLite3 and MariaDB have been configured with a B$^+$-Tree database page size of 64KB in their default journaling modes (rollback journaling and undo/redo logging respectively). The benchmark experiment (described in §3.2.2) has been run for 100 iterations, and the throughput of insert and select operations during each iteration has been measured[8]. This experiment helps us isolate and quantify the impact of *use-hot-locations*, without interference from any other factors.

Theoretically, the benefit of *use-hot-locations* would be the same as the reduction in latency of read requests, i.e. 39% on SSD-S (corresponds to a 64% increase in throughput of select operations). However, applications seldom run solely on disk, and we measure the benefit of this technique in the presence of main memory. Thus, we configure the buffer pool size of SQLite3 and MariaDB to 128MB; this size is heuristically chosen to be about 10% of the initial database size before the start of the experiment (1.2GB).

Modifying the database file layout to use hot locations on SSD-S required changing the I/O modules of SQLite3 (file *os_unix.c*) and MariaDB (file *innobase/os/os0file.cc*), to offset I/O requests to the B$^+$-Tree index file by 32KB. Additionally, in both the database engines, the module for flushing dirty pages from the buffer pool to the database file was modified to issue aligned write requests as shown in Fig. 9b. Overall, we found that this technique could be incorporated with about 200 lines of code in both of these database libraries.

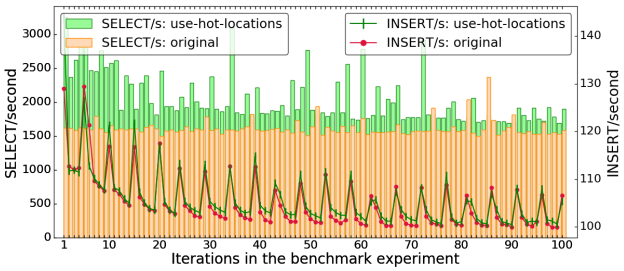
4.1.2 Performance of SQLite3

We run the benchmark experiment with each insert operation in a separate transaction on SQLite3 with (and without) *use-hot-locations* on SSD-S. Figure 12a shows the throughput of select and insert operations over multiple iterations of the benchmark experiment. On an average, we see that the throughput of select operations is 30.4% higher with *use-hot-locations* compared to the original layout. As the database pages coincide with hot locations in the modified file layout, they can be accessed faster (by 39% on SSD-S) thus reducing the latency of select operations.

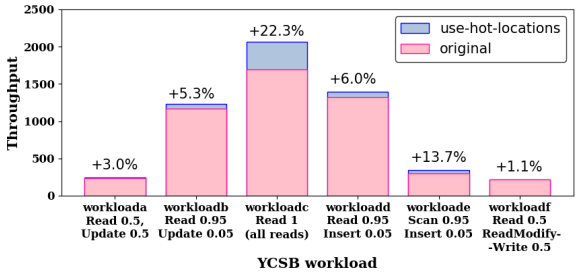
On the other hand, the throughput of insert operations is 15% lower than the original layout. The first step of an insert operation is a seek on the B$^+$-Tree (similar to select), which will be faster with *use-hot-locations*. However, an additional *fsync* needs to be issued at the time of commit (Fig. 9b), to ensure that write requests align with stripe boundaries. This *fsync*[9] acts like a write barrier which guarantees that adjacent write requests are not merged by the OS; they would otherwise overwrite the stripe boundary and violate Rule 3. Thus, the overhead of the *fsync* outweighs the benefit of faster seek in this case.

[8]Throughput of insert/select operations in an iteration is measured as $\frac{\text{number of operations}}{\text{time taken in seconds}} = \frac{\text{50,000 operations}}{\text{time taken in seconds}}$.

[9]Only one additional fsync call is necessary. Non-adjacent write requests are not merged by the OS and can be issued together.



(a) Select and Insert operation throughput over the iterations in a single run of the benchmark experiment with each insert operation in a separate transaction. Total 100 iterations are executed with 50,000 select and insert operations issued in each iteration. For select(insert) operations, we see that the performance *use-hot-locations* is higher compared to the original. The average improvement in throughput is 26.6%(0.9%) over all the iterations. There is no loss in insert operation throughput unlike SQLite3.



(b) We run the YCSB benchmark on MariaDB with 1 million records. Each of the *workloads(a-f)* affect 100K records. We report an average performance over five runs of the benchmark, and the gain in performance for each workload has been indicated in the plot. The highest gain of 22.3% is obtained for *workloadc* (all reads), while the lowest is obtained for *workloada* and *workloadf* (3% and 1.1% respectively) as writers block readers.

Figure 13: The performance of MariaDB with *use-hot-locations* on SSD-S. We obtain an increase of 26.6% and 0.9% in select and insert operation throughput respectively. Unlike SQLite3 (in rollback journaling), where data is written directly to the B$^+$-Tree index file on commit, MariaDB (InnoDB) asynchronously copies updates to the database file using background threads, hiding the overhead of additional *fsync* (see §4.1.3).

To reduce the overhead of *fsync*, we measure the throughput of insert (and select) operations by running the benchmark experiment five times (Fig. 12b) (100 iterations are run each time) while varying the number of insert operations running in a transaction from 1 to 50,000 (i.e, varying the number transactions in the insert phase of an iteration from 50,000 to 1). Increasing the number of insert operations in a transaction (i.e., *batching* insert operations) amortizes the overhead *fsync* and improves the insert operation throughput. We find that the loss in throughput becomes less than 2% when running 500 insert operations per transaction, and ultimately, we obtain a *3% increase* in throughput when running all 50,000 insert operations in a single transaction. Also, as expected, the performance of select operations does not depend on the granularity of transactions in the insert phase, and we obtain a median increase of 29.2% in the throughput of select operations.

4.1.3 Performance of MariaDB

Once again, the benchmark experiment has been run with each insert operation in a separate transaction on MariaDB with (and without) *use-hot-locations* on SSD-S. Fig. 13a shows the throughput of select and insert operations over multiple iterations of the benchmark experiment. Similar to SQLite3, we obtain an average 26.6% increase in throughput of select operations.

However, unlike SQLite3, we do not incur any loss in insert operation throughput. This is because MariaDB (InnoDB) writes updates to the *redo log* at the time of commit, and asynchronously copies the updates to the database file using background threads. Thus, the overhead of the additional *fsync* is incurred in the background, and the observed performance of the database engine remains unchanged.

To further evaluate the benefit of *use-hot-locations* on MariaDB, we run the YCSB benchmark [18] with a million database records (Fig. 13b). Each of the *workloads(a-f)* (refer to Fig. 13b for the composition of these workloads) affect 100K records of the database. The highest gain obtained is 22.3% for *workloadc* (all reads), while the lowest is obtained for *workloada* and *workloadf* (3% and 1.1% respectively) as writers seemingly block readers in both these workloads.

The System Tablespace file (Fig. 8a) of MariaDB (InnoDB) also has a layout similar to the primary index file (Fig. 9a), and one might argue that we could apply *use-hot-locations* to this file as well. However, the pages from this file are seldom accessed (unless there is a transaction accessing the undo log), and using hot locations for this file would not improve the performance by much.

4.1.4 Summary

In conclusion, while the select operation throughput increases (by 29% and 27% for SQLite3 and MariaDB respectively) with *use-hot-locations*, the throughput of insert operations can vary depending upon how the database engine handles updates. In the case of SQLite3, the overhead of the additional *fsync* reduced the insert operation throughput by 15%, as dirty pages were written to the database file directly. However, MariaDB (InnoDB) commits any updates to the redo log instead, and the observed performance of insert operations is not affected by the *fsync*.

It should be noted that the reduction in insert operation throughput of SQLite3 results from the additional *fsync* call, for lack of a better way to ensure that write requests align with stripe boundaries. This exemplifies the lack of synergy between the database system and the underlying storage, thus requiring workarounds to obtain the desired behavior. With a better contract between the system and the storage device, such an overhead could perhaps be avoided.

4.2 write-aligned-stripes on SSD-T

We apply the technique *write-aligned-stripes* (Fig. 11) to SQLite3 and modify the layout of the WAL log file as shown in Fig. 11b. As the log frames are modified to be of stripe size (64KB for SSD-T), we eliminate the violation of Rule 3 while writing to this file.

4.2.1 Experimental Setup

Again, SQLite3 has been configured with a database page size of 64KB in the write-ahead logging (WAL) mode. The benchmark experiment (§3.2.2) has been run for 20 iterations with a single insert operation per transaction. Auto-checkpointing in SQLite3 has been turned off (by setting

Table 2: *write-aligned-stripes* on SSD-T. While the observed performance remains the same, write amplification has been eliminated entirely by modifying the layout of the WAL log file (Fig. 11b).

Property	Original	Optimized (write-aligned-stripes)	Difference (Optimized vs Original)
Initial DB Size	19424 DB pages (of size 64KB)	19448 DB pages (of size 64KB)	increases by 0.12%
Final DB File Size (after 20 iterations)	21418 DB pages (of size 64KB)	21436 DB pages (of size 64KB)	increases by 0.08%
Avg. Insert Operation Throughput	288 Insert/s	290 Insert/s	increases by 0.7%
Avg. WAL Checkpoint Time	0.087s	0.09s	increases by 3.4%
Avg. WAL frames written per iteration	101133	101105	decreases by 0.03%
Write Amplification per iteration	50000 flash pages	0	*decreases by 100%*

wal_autocheckpoint to zero [5]), and an explicit checkpoint is performed at the end of every iteration.

Various measurements are made during every iteration, and have been summarized in Table 2. Modifying the layout of the WAL log file required making changes to the write-ahead logging module of SQLite3 (file *wal.c*), along with reserving space at the end of every database page using `SQLITE_TESTCTRL_RESERVE` [8] operation. Overall, this change could be incorporated with less than 100 lines of code in a highly modular manner, without affecting any other sub-modules of SQLite3.

4.2.2 Observed Performance

Table 2 shows the result of this experiment. We first discuss the directly measurable performance metrics of the database engine.

1. ***The Initial and Final DB file sizes***: As 24B of space is reserved at the end of a database page in the new layout (Fig. 11b), its capacity is reduced by a small amount. Thus, we see that the size of the database file is slightly larger compared to the original format.
2. ***Average throughput of Insert Operations***: We find that the throughput with the modified WAL layout is slightly higher (0.7%). This is to be expected as the size of the log frame is smaller in the modified layout, leading to faster commit operations as lesser amount of data needs to be written to the log.
3. ***Average WAL checkpoint time***: At the end of every iteration, we checkpoint the WAL log and measure the time taken to complete this operation. We find that the time taken for the checkpoint operation on an average is quite low, and is approximately the same in both cases (3.4% higher for the modified layout).
4. ***Average number of WAL frames written per iteration***: Again, we find that roughly same number of frames are written to the log in both cases. However, the objective of recording this is to estimate the amount of flash wear-out caused during the experiment.

Thus, we find that the observed performance in both cases is about the same. However, the goal of *write-aligned-stripes* is to eliminate write amplification, which we describe next.

4.2.3 Write Amplification and Estimated Wear-out

We measure write amplification as the number of partially written flash pages. In the original layout of the WAL log file, the 24B header causes a frame to spill over the 64KB boundary. WAL frames are written to the log file contiguously on a commit. Thus, the amount of overflow beyond the 64KB boundary on a commit in the original layout is:

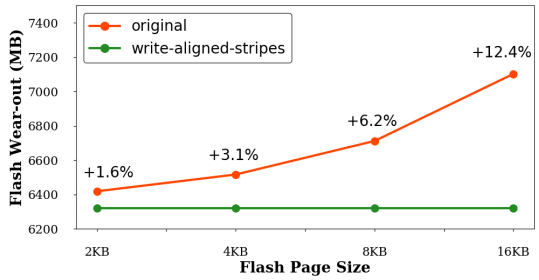


Figure 14: Measuring the benefit of *write-aligned-stripes* in terms of reduction in wear-out. The total flash wear-out caused during the experiment is estimated for different flash page sizes. Decreasing the wear-out corresponds to increasing the lifetime of the SSD. Thus, the increase in lifetime ranges from 1.6% to 12.4%.

$$\text{Overflow} = 24B \times \text{Number of frames written}$$

In the experiment, a single insert operation has been run per transaction, and the number of frames written on a commit is low (two frames on average). Thus, the overflow is $< 100B$, which can easily be contained in a single flash page irrespective of its size (the size of a flash page ranges from 2KB to 16KB [6, 31, 45]). Therefore, we report the write amplification in the original layout as 50,000 *flash pages*, as a single additional flash page is written on a commit operation. The modified layout of the WAL log file has no overflow and no write amplification as a result.

Although the write amplification can be accurately measured in units of flash pages, the total wear-out caused depends on the flash page size. We calculate the wear-out as:

$$\text{Flash wear-out} = \text{Number of frames written} \times 64KB + \text{write amplification} \times \text{flash page size}$$

Fig. 14 shows the total flash wear-out caused during the experiment for possible flash page sizes. The difference in wear-out between the original and modified layout depends on the size of the flash page, and ranges from 1.6% to 12.4%. By reducing the wear-out, one can increase the lifetime of an SSD. Thus, the increase in the lifetime also ranges from 1.6% to 12.4%, depending on the flash page size.

This analysis holds true for any SSD, as the amount of data written to disk during the experiment remains unchanged. In particular, SSD-T has a flash page size of 4KB, and its lifetime should improve by 3.1% through this technique. Thus, transparency regarding basic parameters such as flash page size can help applications make informed decisions, while benefiting the SSD as well.

4.3 contain-write-in-flash-page on SSD-T

We apply the technique *contain-write-in-flash-page* (from Fig. 10) to the MariaDB binary log file on SSD-T. We run the benchmark experiment for 20 iterations, which results in about one million write operations to the binary log file (one write request for each insert operation's commit) of size 275B on an average. 6.7% of these write requests fall on the flash page boundary violating Rule 5. Although the actual size of the binary log is about 262MB, the total flash wear-out caused is $1.067M \times$*flash page size*$(4KB) = 4.1GB$. Thus, by offsetting these write requests to align with the next flash page boundary, we reduce the write amplification caused by the binary log by 6.7%[10] (262MB). This technique can be applied on any SSD, by making a conservative guess (say 1KB) if the flash page size is not known.

5. RELATED WORK

There is a fair amount of existing work on adapting B-Trees for flash-based SSD storage [19,35,38,40,44,48] based on general recommendations described in §2.2. A large portion of the work focuses on optimizing write operations to the SSD. Small random write operations deteriorate the performance obtained for reasons like write amplification, low bandwidth utilization, and increased address translation overhead. These can be overcome through buffering to issue large write requests [19,35,38,48], and through logging [40].

In [44], authors attempt to improve select performance by utilizing the internal parallelism of SSDs. This is achieved by issuing multiple read requests corresponding to multiple concurrent search operations on the B-Tree. In contrast, using hot locations reduces the latency of a single request through effective utilization of channel-level parallelism.

The internal operation of SSDs and its impact on applications has been extensively studied [20,24,25,31,37,42]. Multiple authors have studied factors like read and write amplification [33,47], the impact of address translation [20,29,31], performance of sequential vs. random requests [20,24,25,37], and proposed rules for applications based on them. These recommendations are based on the general behavior of SSDs, and have been summarized in §2.2.

The hierarchical structure of SSDs [20, 25, 34, 37] and utilizing the multiple levels of parallelism [20,34] to improve the performance of the device have been of interest in the past. Multiple authors have taken a white box approach [20, 26, 34,36,41] to propose how internal operations such as address translation should be handled. These studies emphasize the importance of channel-level parallelism [34], and have helped us build an analytical model of SSDs to reason about the performance obtained in different scenarios.

However, in the real world, commercial SSDs are a black boxes and we have attempted to learn their characteristics and parameters through measurements. Our experiment for obtaining the *request size profile* (§3.1.1) is informed by previous work which describes the utilization of RAID-like striping schemes [20, 27] to distribute data at multiple levels of the storage hierarchy, whereas obtaining the *location profile* (§3.1.2) has been adapted from [25].

A similar approach of treating SSDs as a black box, and inferring the internal parameters through measurement has been taken in [25, 37]. Chen et al. [25] propose experiments to identify the chunk size and number of channels in an SSD. However, their main focus is to show the benefit of issuing concurrent requests to an SSD, and their recommendations make limited to no use of these parameters. Unlike [25], Jaehong et al. [37] attempt to find the the clustered page and block size through various measurements, as well as use these parameters to tune the Linux I/O scheduler. Their objective is similar to our work, i.e., making SSD-specific optimizations informed by internal parameters. However, there are two differences between our work and [37]. First, the parameters considered in our work are different from [37]. Second, our recommendations for SSD-specific optimizations are at the *application-level*; for instance, techniques like *write-aligned-stripes* cannot be directly implemented in the OS, and decisions such as applying *use-hot-locations* to the primary index file alone, and not to the System Tablespace file can only be made by the application.

In recent years, open-channel SSDs [22] have been developed, which do not have an FTL and instead give the user full control over available resources. However, a vast majority of SSDs used are manufactured by commercial vendors, who do not reveal their internal policies. With the advent of persistent memory technologies such as 3D XPoint memory [30], one can expect hybrid devices to be developed, possibly requiring newer techniques to learn their parameters as the complexity if these devices increases.

6. CONCLUSIONS AND FUTURE WORK

In conclusion, we have shown that optimizing a database engine for a particular SSD can be beneficial. Our contributions are threefold; *learning* hidden parameters of SSDs (§3.1), proposing *rules* specific to a device (§3.2), and *optimizing* database engines for the device using these rules (§3.3). We have studied SQLite3 and MariaDB as part of our work, and we propose three techniques to optimize these database engines, namely *use-hot-locations* (Fig. 9), *write-aligned-stripes* (Fig. 11), and *contain-write-in-flash-page* (Fig. 10). These techniques increased the throughput of select operations by 27-29% on SSD-S, and reduced the wear-out caused to SSD-T by 3.1% and 6.7% respectively.

As future work, it is possible to implement the proposed rules and techniques in an independent pluggable layer for applications to use. We envision this layer to be highly configurable, as often applications have the broader view of their performance goals and some corrections (eg. *write-aligned-stripes*) can be best made by them. Overall, SSDs are complex devices with a range of characteristics. We would like to reiterate that there is no silver bullet to learn all the internal parameters of every SSD. However, we believe that such an attempt is worthwhile, and we have found that besides benefiting the database engine they can improve the expected performance and the lifetime of the SSD as well. The parameters learned and the recommendations outlined in this paper are by no means complete, and the most reliable source of this information are vendors themselves. Thus, we would like to end with a recommendation for SSD manufacturers to be more transparent regarding basic parameters such as the flash page size and the stripe size.

7. ACKNOWLEDGMENTS

This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab and by the National Science Foundation under grant OAC-1835446. Some of the experiments reported in this work were run on CloudLab [28].

[10]The benefit of this technique will vary, as the size of write requests to the binary log depends on the workload.

8. REFERENCES

[1] HDD vs SSD: What does the future for storage hold? https://www.backblaze.com/blog/ssd-vs-hdd-future-of-storage/.

[2] Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series.html.

[3] MariaDB Success Stories. https://mariadb.com/kb/en/library/mariadb-success-stories.

[4] *pandas*: Python Data Analysis Library. https://pandas.pydata.org/.

[5] PRAGMA Statements supported by SQLite. https://www.sqlite.org/pragma.html.

[6] Samsung K9XXG08UXA Flash Datasheet. http://www.samsung.com/semiconductor.

[7] Solid-state Drive. https://en.wikipedia.org/wiki/Solid-state_drive.

[8] SQLite Testing Interface Operation Codes. https://www.sqlite.org/c3ref/c_testctrl_always.html.

[9] SQLite3 Database File Format. https://www.sqlite.org/fileformat.html.

[10] SQLite4 LSM Benchmark. https://sqlite.org/src4/doc/trunk/www/lsmperf.wiki.

[11] Storage Engines: MariaDB Knowledge Base. https://mariadb.com/kb/en/library/storage-engines.

[12] strace. https://strace.io/.

[13] The InnoDB Storage Engine. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[14] The Physical Structure of an InnoDB Index. https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html.

[15] The SQLite3 Database Engine. https://sqlite.org/index.html.

[16] Working With SQLite Databases using Python and Pandas. https://www.dataquest.io/blog/python-pandas-databases/.

[17] Write-Ahead Logging in SQLite3. https://www.sqlite.org/wal.html.

[18] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB.

[19] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1):361–372, 2009.

[20] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA.

[21] A. Baxter. SSD vs HDD. https://www.storagereview.com/ssd_vs_hdd.

[22] M. Bjørling, J. Gonzalez, and P. Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017.

[23] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 217–228, New York, NY, USA, 2009. ACM.

[24] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[25] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277, Feb 2011.

[26] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A Survey of Flash Translation Layer. *J. Syst. Archit.*, 55(5-6):332–343, May 2009.

[27] C. Dirik and B. Jacob. The Performance of PC Solid-state Disks (SSDs) As a Function of Bandwidth, Concurrency, Device Architecture, and System Organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM.

[28] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[29] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 229–240, New York, NY, USA, 2009. ACM.

[30] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[31] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144, New York, NY, USA, 2017. ACM.

[32] P. Hernandez. SSD vs HDD: Price Comparison. http://www.enterprisestorageforum.com/storage-hardware/ssd-vs-hdd-price-comparison.html.

[33] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write Amplification Analysis in Flash-based Solid State Drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM.

[34] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and

C. Ren. Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, June 2013.

[35] Z. Jiang, Y. Wu, Y. Zhang, C. Li, and C. Xing. AB-Tree: A Write-Optimized Adaptive Index Structure on Solid State Disk. In *2014 11th Web Information System and Application Conference*, pages 188–193, Sept 2014.

[36] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 161–170, New York, NY, USA, 2006. ACM.

[37] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). *IEEE Trans. Comput.*, 61(5):636–649, May 2012.

[38] Y. Li, B. He, R. J. Yang, Q. Luo, and K. Yi. Tree Indexing on Solid State Drives. *PVLDB*, 3(1-2):1195–1206, 2010.

[39] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pages 177–190, New York, NY, USA, 2015. ACM.

[40] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 410–419, New York, NY, USA, 2007. ACM.

[41] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, Aug. 2008.

[42] I. L. Picoli, C. V. Pasco, B. T. Jónsson, L. Bouganim, and P. Bonnet. uFLIP-OC: Understanding Flash I/O Patterns on Open-Channel Solid-State Drives. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 20:1–20:7, New York, NY, USA, 2017. ACM.

[43] D. Purohith, J. Mohan, and V. Chidambaram. The Dangers and Complexities of SQLite Benchmarking. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 3:1–3:6, New York, NY, USA, 2017. ACM.

[44] H. Roh, S. Park, S. Kim, M. Shin, and S.-W. Lee. B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *PVLDB*, 5(4):286–297, 2011.

[45] B. Tallis. Micron 3D NAND Status Update. `https://www.anandtech.com/show/10028/micron-3d-nand-status-update`.

[46] A. J. Uppal, R. C. Chiang, and H. H. Huang. Flashy prefetching for high-performance flash drives. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[47] Wikipedia. Write Amplification. `https://en.wikipedia.org/wiki/Write_amplification`.

[48] C.-H. Wu, T.-W. Kuo, and L. P. Chang. An Efficient B-tree Layer Implementation for Flash-memory Storage Systems. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[49] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J.-G. Lou, M. Chintalapati, and D. Zhang. Improving service availability of cloud systems by predicting disk error. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 481–493.