Chapter 2

Survey of Storage, Indexing, and Database Systems

2.1 Storage Technologies

2.1.1 Hard disk drives

Hard disk drives (HDDs) were introduced by IBM with the release of IBM 350 in 1956 [33]. Its capacity was 3.75 MB, and [34] puts the cost at US$34500. Adjusted for inflation, that equals a today's value of about $300000 [41] (US$80000 per MB). Since then, prices have fallen and capacity has increased tremendously. Today, for example, a Western Digital 4TB HDD with 128 MB cache, an HDD designed for data centers and tested for 24×7×365 reliability at workloads up to 550TB per year, costs about US$210 (US$0.0000525 per MB) [3]. That is an improvement of astounding 1.28-billion-to-one in price to capacity ratio. Admittedly an extreme comparison, but it serves as a testament to the long development of HDDs. And cheaper 4TB consumer HDDs are available. The WD disk is also available with 8 TB capacity for about $550, and that is not the maximum possible capacity of HDDs. New technologies help to increase the density of disks and thus increase capacity. One example is Seagate Shingled Magnetic Recording (SMR) which increases capacity by more than 25% by further maximizing the number of tracks per inch on a single disk [64]. Hence, for databases that need to store vast amounts of data at low costs, HDDs are an excellent choice.

Unfortunately, access time has not improved at the same rate as capacity. The IBM 350 had an access time of about 600 ms [33]. Today, it is still in the range of a few milliseconds, typically about 5–10ms [36] [66], and is not expected to see any drastic improvement anytime soon. Also, an HDD consists of many mechanical parts, notably spinning magnetic disks and the disk arm. All of these components are prone to physical movements and have a limited lifetime.

2.1.2 Solid state drives

Solid state drive (SSDs) use flash memory and are purely electronic devices. They contain none of the moving, mechanical components present in HDDs, thus eliminating spinning disks and vulnerability to physical impacts. Flash memory brings several additional improvements over magnetic disks, such as faster access times, lower latency, lower power consumption, completely silent operation, and potentially uniform random access speed [48]. The capacity of SSDs was initially too limited to seriously consider using them in big database systems. But the capacity has improved drastically over the last years. In March 2016, Samsung announced PM1633a SSD for enterprise storage systems with a staggering capacity at 15.36TB [61], making it the world's largest capacity SSD. Just as impressive is its claimed reliability of 1 DWPD (drive writes per day), which means that 15.36 TB can be written every day without failure. That amounts to more than 5500 TB in a year; more than 10 times the reliability guaranteed by Western Digital for their Gold data center HDD. The price for the PM1633a SSD is $10000.

NAND in the type of flash memory used by most SSDs. NAND memory is organized into blocks. One block usually consists of 64 or 128 pages, each of size 2KB or 4KB [32][40][36]. Reads and writes are at performed in unit of pages, and erases in unit of blocks. NAND flash memory has some distinctive features and limitations compared to DRAM flash memory and magnetic disk. One limitation particularly affecting database systems is that the data in NAND flash memory cannot be updated in-place. Writes can flip bits only from 1 to 0, thus an erase operation setting all bits in a block to 1 must be performed before it is possible to write to any of the pages in that block [15]. This dependency cycle is called erase-before-write and prevents efficient overwrite of pages in-place; rewrite of a single page requires the entire block in which it resides it to be completely erased before the page can be written back. The erase-before-write cycle leads to poor write performance due to the high latency of the erase operation and the overhead involved in backing up the other pages in the block, which should be left unaffected by the rewrite of a single page write.

The erase-before-write cycle is unfortunate not only for performance, but also for the lifespan of an SSD because each NAND flash memory block can sustain only a limited number of erase-write-cycles. Just how many cycles a block endures differ, but is often put at about 10K [40] [15] [31]. However, as bit density has increased over the last years to make higher capacity SSDs possible, the typical lifespan has started to decrease. Some disks now have a lifespan expectancy of only 5K cycles [4], making it even more vital to optimize writes such that expensive hardware maintenance and replacement costs can be avoided.

Flash Translation Layer, Wear Leveling, Garbage Collection

Frequent updates to a small set of pages blocks will wear the blocks holding those pages faster than the other blocks [15]. The optimal write distribution is to write to all blocks uniformly over time; no blocks would then be statistically more likely to fail than others. Unfortunately, uniform distribution is not the nature of writes in reality – especially not for data structures that update pages in-place. To prolong the lifetime

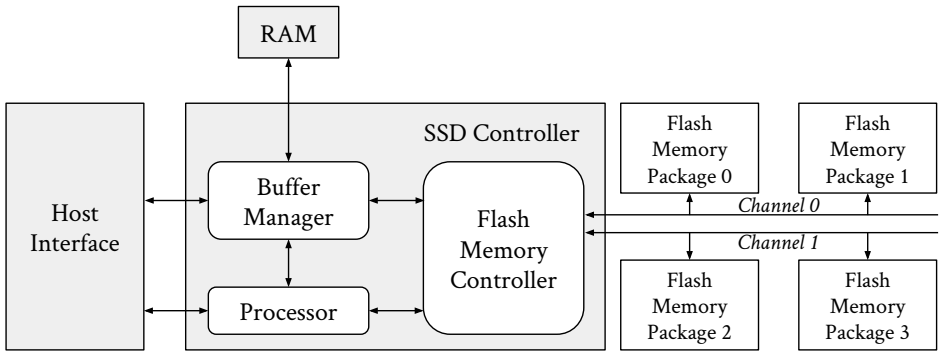


Figure 2.1: Architecture of a solid state drive.

and reduce latency, SSDs come with a firmware layer called flash translation layer (FTL) that implements wear leveling. Wear leveling distributes writes over the entire disk to avoid erase-write cycles repeatedly the same set of blocks.

Each block on a disk can be roughly divided into one of two categories depending on the data it contains: static block (infrequent writes) or dynamic block (frequent writes). Wear leveling is typically divided into three categories [31] [47]:

- **No wear leveling**: Fixed mapping from logical addresses in the operating system to physical addresses on the SSD. Modifying data in a block requires the block to be fetched from disk, erased on disk, modified in memory, and written back to disk in its original location. No wear leveling causes blocks with dynamic data to wear out much faster than blocks with static data.
- **Dynamic wear leveling**: Dynamic mapping from logical to physical addresses. Each time a block is about to be rewritten, the original block is marked as invalid, and the updated block is written to a new location. Dynamic wear leveling distributes writes better than no wear leveling, but only for dynamic blocks. Static blocks will never or very seldom be mapped to a new location.
- **Static wear leveling**: Same as dynamic wear leveling, but it also moves static blocks periodically.

Wear leveling comes at a cost. Static wear leveling gives the longest lifetime of the three, but has the slowest performance and most complex design. As in so many other areas, a tradeoff must be made.

Static and dynamic wear leveling both implement updates out-of-place by maintaining a mapping table between logical and physical pages. To make pages available for later writes, the FTL runs a garbage collector responsible for cleaning up old blocks that are considered eligible for erase. When to erase a block is not a straightforward decision. Erasing a block only when all its pages are invalid is not efficient as a vast

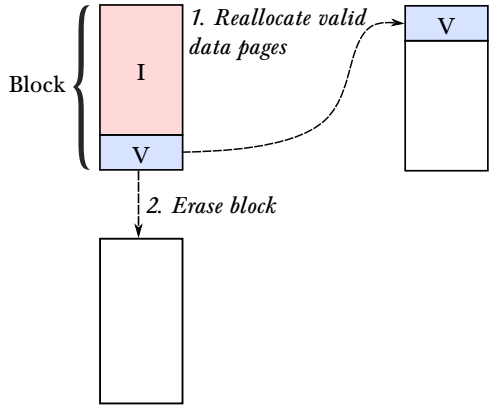


Figure 2.2: Illustration of garbage collection in an SSD.

amount of blocks will contain both invalid and valid pages. The final decision is made by the garbage collector algorithm. The algorithm can, for example, decide to erase a block when a certain percentage of its pages are invalid. Valid pages must then be copied to a free block before the old block is erased. Figure 2.2 provides an illustration of such an operation.

Garbage collection has overhead. Moving pages to new blocks generate additional read and write operations, and these operations generate write amplification. Write amplification for flash memory is defined as "the average of actual number of page writes per user page write" [32]. In Figure 2.2, suppose that I pages have already been updated out-place and rewritten to another location; they are thus now invalid in the old block. But to erase the entire block, the remaining V pages must first be reallocated. Hence, in order to rewrite I user pages, the total number of physical pages that actually has to be written is $V + I$. The write amplification can be written as [32]:

$$WA = \frac{V + I}{I} = 1 + \frac{V}{I} \tag{2.1}$$

V/I is the actual write amplification the garbage collector produces when reallocating valid pages.

2.1.3 Internal Parallelism of an SSD

Due to physical limitations, the maximum write speed to a single NAND chip is limited to around 35-40MB/s [2] [15]. To achieve higher write speeds, manufacturers started to parallelize and interleave packages of NAND chips [31] [15]. Figure 2.3 gives an overview of how the internal parallelism of an SSD works. To exploit the parallelism on the design, SSDs perform read and write operations in units of clustered pages to enlarge the unit size of the operations. A clustered page is a combination of several physical pages. Since each physical page resides on different NAND flash chips, the clustered page size is the same or a multiple of the physical page size

of NAND flash memory [40]. Similarly, a clustered block consists of several blocks from different NAND chips. Erase operations operate in units of clustered blocks to improve garbage collection performance by erasing blocks on multiple chips in parallel.

The SSD in Figure 2.3 features two channels and four chips with one plane per chip. Data is read and written in chunks of size equal to block size. The write operation writes *{A, B, C, D}* sequentially and fills a clustered block. The operation is striped across four planes and exploits the full potential of the parallelism and interleaving.

2.1.4 Exploiting The Internal Parallelism of SSDs

Programs are often parallelized to improve performance and better exploit the potential of multi-core processors. But I/O performance does not necessarily increase from parallelization of programs even if it is supported by the underlying data structure. As just metnioned, the write performance of an SSD increases with the size of the write request because the SSD controller can better exploit the internal parallelism of the SSD. Hence, if one large write request of size multiple of the clustered block size is split into smaller write requests each of size smaller than a clustered block, the overall write performance can decrease. On the other hand, if all write requests are small and cannot be batched into one large write operation, writing from multiple threads is likely to yield better performance.

How data is written affects the read performance. Writing data in one operation such that the SSD controller can stripe data across channels not only improves write performance, but also for subsequent operations reading the same data. Figure 2.3 illustrates that reading $\{A, B, G, H\}$ yields better performance than reading $\{A, B, E, F\}$ since the first read will better benefit from the internal parallelism of the SSD. Hence, if possible, data likely to be read in one operation should be written together such that it can be optimally parallelized by the SSD controller.

2.2 The Importance of Sequential Writes

Generally, sequential writes are significantly faster than random writes, both on SSDs and HDDs. The performance gap between the two write patterns depends on the size of the write requests and type of secondary storage; the smaller the write request, the greater the difference. The difference is more profound on HDDs due to the expensive head seek time; for small read and write requests, the head seek time dwarfs the transfer time.

Even though SSDs have far better random write performance than HDDs, random writes can generate internal fragmentation on SSDs, leading to reduced performance [8] [15]. The implication of random writes on an SSD is not necessarily obvious immediately, and one might be led to believe that random writes are just as fast as sequential writes if the SSD is brand new or freshly erased. To see the real impact of random writes on SSDs, it is necessary to benchmark the SSD over a long time. Figure 2.4 shows how drastic the performance of a new SSD can change over time. During the first 15 minutes or so, the SSD shows exceptional performance with random

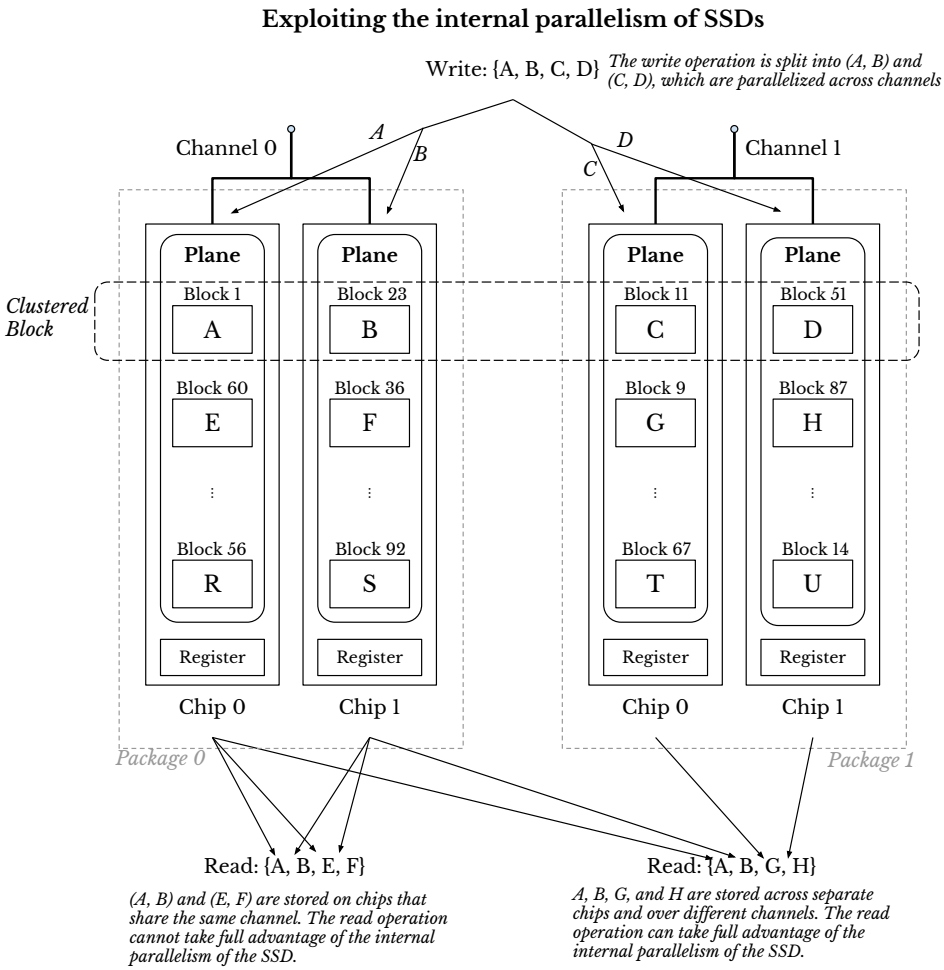


Figure 2.3: Illustration of a NAND flash package

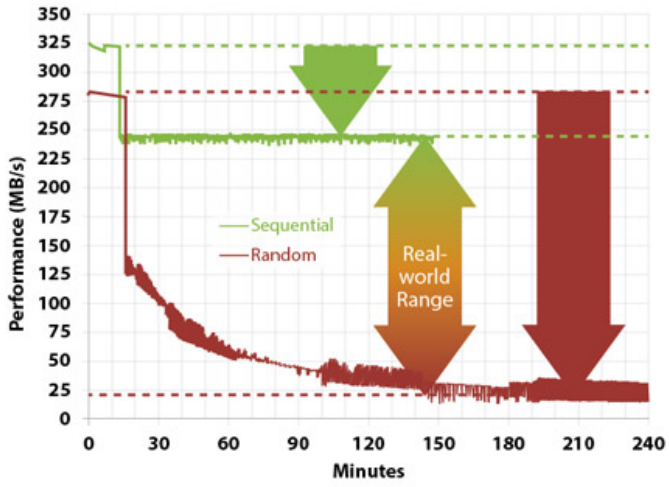


Figure 2.4: Performance of random and sequential writes to an SSD. Source: Seagate [65]

write speeds between 275MB/s and 300MB/s and sequential write speeds around 325MB/s. Both perform so well that the small difference is arguably negligible in practice for a database system.

Note, however, that during those first 15 minutes, there is no old data on the disk and no need for garbage collection. The scenery changes completely as the disk begins to reach full capacity and garbage collection begins. The performance of both is reduced but whereas the sequential write speed drops to 250MB/s, the random write speed plunges to about 25MB/s. Consequently, sequential writes are preferable over random writes on both HDDs and SSDs; both for performance and to prolong the lifetime of the disks. Previous work by [8][15][70] confirms that random write performance drops by orders of magnitude after a period of extensive random updates, but that write performance once again improves after a period of extensive sequential writes.

The concepts of wear leveling, garbage collection, and write amplification are important when discussing random versus sequential writes. If a database performs all updates in-place, the FTL will have to work intensively to ensure wear leveling, perform garbage collection, and maintain an up to date mapping between logical block addresses (LBA) — i.e., the user/host address space — and physical block addresses (PBA). However, if the workload writes strictly sequential in the order of LBAs, no complex garbage collection is required because flash blocks are invalidated block by block as writes proceed [32][65], and the write amplification WA in Equation (2.1) will approach 1.

The size of the write requests is important to consider as it has an impact on performance. A write request should ideally be a multiple of the clustered page size, as in Figure 2.5a. Figure 2.5b and 2.5c show alignments of write requests that are

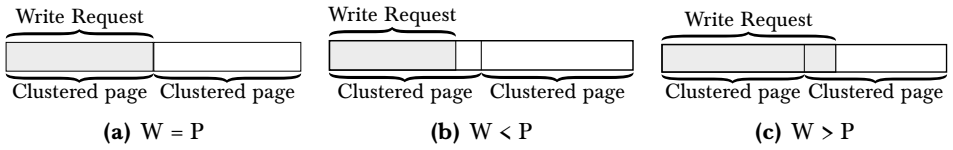


Figure 2.5: Examples of alignments of write requests to clustered page size. W is the size of a write request, P is the size of a clustered page.

smaller and larger than the clustered page size, respectively. Write requests that are aligned to clustered pages can be written to disk with no further write overhead. In contrast, write requests that are not aligned generate overhead because the SSD controller needs to read the rest of the content in the last clustered page and combine it with the updated data before all the data can be written to a new clustered page. Such read-modify-write operations increase write latency, and the overhead of the operations can be observed in Figure 2.7 which shows how latency varies with the size of write requests. It is important to observe the periodic drops in latency in Figure 2.7. For example, the Samsung SSD in Figure 2.7 (a) shows a drop in latency whenever the write request is a multiple of 16KB, indicating a clustered page size of 16KB because the writes would then be aligned to the clustered pages. Similar reasoning for the Transcend SSD in Figure 2.7 (b) makes it likely that it has a clustered page size of 128KB. Write requests not a multiple of the clustered page size are slower because they lead to additional read-modify-write operations.

The size of the write request is relevant because it affects fragmentation and the advantage of sequential writes over random writes. Random writes can perform just as well as sequential writes, also over longer periods of time, if all write requests are equal to or multiple of the size of a clustered block. Figure 2.6 illustrates how the size of write requests affects fragmentation on an SSD. The advantage of sequential writes becomes apparent in Figure 2.6a if the write request size is not a multiple of the clustered block size; writing sequential enables the disk to fill up blocks regardless of the size of the write.

If, however, the sizes of the random writes are multiples of the clustered block size, internal fragmentation within blocks is prevented. Random writes of such sizes will also benefit from the internal parallelism of the SSD since the writes are large enough to be striped over multiple channels and chips.

Random writes aligned to clustered blocks will still cause fragmentation between clustered blocks, as seen in Figure 2.6b and 2.6c. This fragmentation is called external fragmentation and will hurt performance on HDDs because it results in more head seeks. But on SSDs where the issue with random writes lies in the increased complexity of garbage collection, the complexity is now no more complex than it is for sequentially filled blocks. Hence, as long as write requests are smaller than the size of a clustered block, sequential writes perform better than random writes. Figure 2.8 and 2.9b show how much bandwidth improves as write requests increase in size. The performance of both random and sequential writes increase with larger write requests, and when the size of a write request is equal to or larger than the clustered block size, random and sequential writes achieve similar bandwidths. 16MB or 32MB is the clustered block

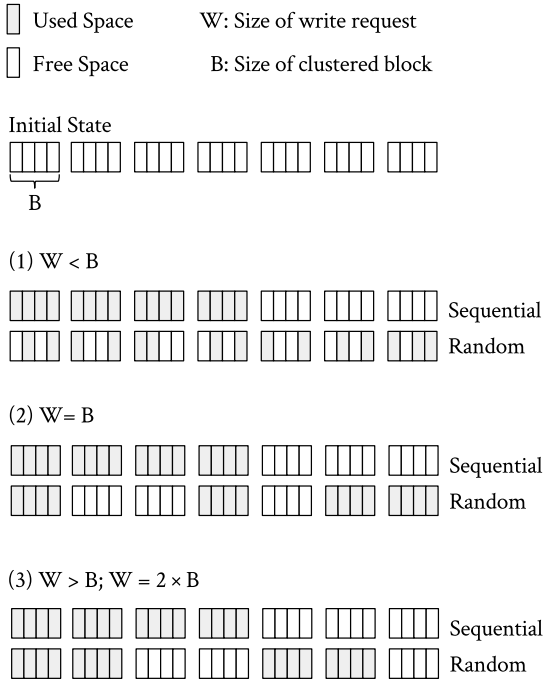
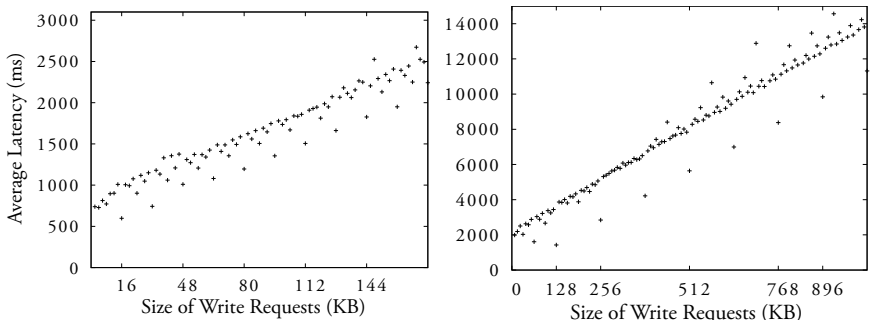


Figure 2.6: How sequential and random write requests affect fragmentation in blocks. Drawing based on: [40]



(a) Samsung MCCOE64G5MPP (64GB) **(b)** Transcend TS64GSSD25S-M (64GB)

Figure 2.7: Latency as a function of size of write requests. Source: [40]

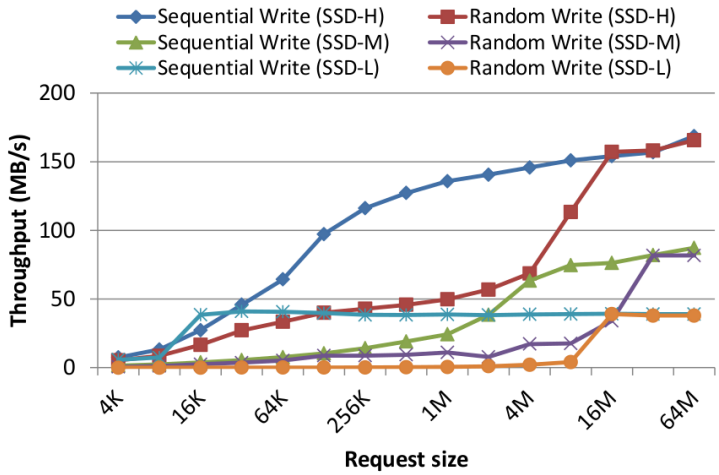
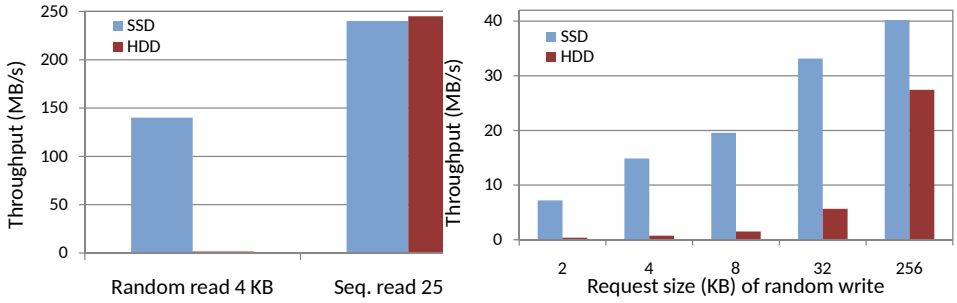


Figure 2.8: How write request size affects write performance to SSDs. Source: [48]

size for most SSDs. The clustered block size seems to be 16MB for the SSD-H and SSD-L in Figure 2.8; and 32MB for SSD-M in 2.8.

Based on this insight, random writes can indeed be justified over sequential writes if the write requests are larger than or equal to the clustered block size. However, how to achieve write requests of such sizes in a database system is not straightforward. As will be evident when reviewing and evaluating the different data structures in this and later chapters, one of the essential keys to great write performance is to organize data in such a way that each write request distributes the overhead of the write over multiple inserts batched together.



(a) Reads. **(b)** Random writes.

Figure 2.9: Comparison of how size of read and write requests affect throughput on HDDs and SSDs.

2.2.1 Over-provisioning

Garbage collection runs in the background during idle time to erase and free blocks before new write requests arrive. But if the disk is full and random writes continue to dominate over sequential writes, garbage collection becomes such a complex operation that it could turn into a bottleneck for writes. An erase operation has a higher latency than a write operation, and the disk could end up in a state in which write requests constantly consume all free blocks before the garbage collector has managed to erase enough blocks for subsequent write request, resulting in limited write throughput.

One way to mitigate this problem is to set a side a spare area on disk that the garbage collector can use as a buffer. This approach is known as explicit over-provisioning, and works by simply formatting the disk to a logical partition of size smaller than the full physical capacity of the disk [31]. The remaining, unpartitioned space will be invisible to the user, but visible and accessible to the SSD controller and its garbage collector. The buffer can be used to absorb writes during workload peaks. Benchmarking indicates that reserving 25% disk space as spare area for intensive random write workloads, or 10-15% for more balanced workloads is enough for increased performance [67]. Many manufacturers implement over-provisioning out-of-the box with no ability for the user to access or modify the spare area. For example, Intel's SSD DC S3700 has 264GB of NAND memory, but only 186GB is exposed to the user (advertised capacity is 200GB). The remaining spare area is used to improve performance, consistency, and endurance.

2.3 B-tree

The B-tree is a generalization of the binary search tree and was introduced in 1970 as a data structure intended to store data on hard disk drives [5]. It was designed to achieve high locality to minimize head seek times, and is therefore suitable for read intensive systems and applications such as databases and file systems. In contrast to a binary search tree that can contain no more than two children, an internal node in

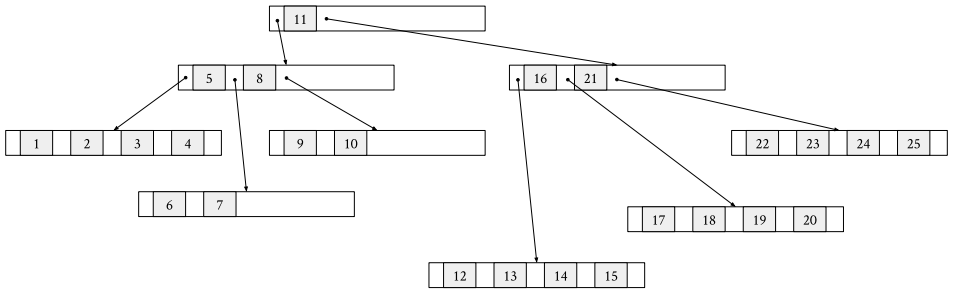


Figure 2.10: Illustration of a B-tree.

a B-tree may contain any number of children (i.e., child nodes, or subtrees).

The minimum number of children an internal node may have is referred to as the branching factor of the tree. The branching factor can be anywhere between a handful to several thousands depending on implementation and configuration. A B-tree used in a database usually has a high branching factor, often in the hundreds, to assure that the tree will remain shallow but wide as it grows so that only a few nodes need to be visited when searching for an element.

Each internal node in a B-tree contains a number of keys and their associated data. The keys also act as separation values to separate its subtrees, and are used to guide searches through the tree. An internal node with k keys must have $k+1$ subtrees. For example, a node with 3 subtrees must have 2 keys, k_1 and k_2, to separate the subtrees. All keys in the leftmost subtree will be less than k_1, all keys in the middle subtree will be between k_1 and k_2, and all keys in the rightmost subtree will be greater than k_2. Figure 2.10 illustrates the structure.

Space and performance

The slow access time of HDDs makes it desirable to keep I/O to a minimum. And despite the lower access time of SSDs, the disk still becomes a bottleneck if all reads were to consult the disk. It is therefore advantageous to keep as much of the tree as possible in memory. Yet in many cases there is simply not enough memory to hold the entire tree, so only a subset of the nodes can be in memory at any time. If queries were to vary widely over a key range, each query will take different paths down the tree. Nodes visited in each query are thus unlikely to be the same as the ones visited in the preceding query, and many of the nodes in the search path may have to be fetched from disk. Consequently, the $O(\log N)$ search time of the B-tree is not necessarily that fast in reality because many of the operations will involve I/O.

2.4 B$^+$-Tree

The B$^+$-tree attempts to solve the I/O problem of the B-tree by keeping more of the internal nodes in memory. A B$^+$-tree is almost identical to a B-tree, except that a B$^+$-tree stores all keys and data in the leaf nodes. Each internal node contains only

pointers to its children, and the separator keys are copies of the keys in the leaf nodes. These copies cause a B+-tree to use slightly more total space than a B-tree. However, the internal nodes themselves can constitute as little as 0.5% of the total space occupied by the entire tree. A B+-tree with hundreds of child pointers and with prefix and suffix truncation employed is likely to have at least 99% of its internal nodes fit in memory [29]. With close to all internal nodes in memory, it is realistic to expect that maximum one I/O is required to access a leaf node – a significant reduction in I/O cost compared to the original B-tree.

Most of today's popular B-tree databases use a B+-tree. To simplify notation, any reference to the B-tree in this text as of now is a reference to the B+-tree, unless explicitly specified otherwise.

2.4.1 Implementations of B+-Tree

InnoDB

One implementation of the B+-tree is InnoDB, the default storage engine in MySQL 5.5.5 and later versions, replacing the previous storage engine MyISAM. Figure 2.11 illustrates the structure of the B-tree in InnoDB. To speed up range queries, pages in the same level are chained together as a doubly-linked list. Records on a page are singly-linked together in logically ascending order. The infimum and supremum are system records and represent the lowest and highest possible keys of a page.

2.4.2 Fragmentation

Fragmentation is an inherent problem to all B-trees. As roughly depicted in Figure 2.10, a node is allocated more space than just what is needed for the initial records stored in that node. For example, InnoDB leaves about 1/16 of a page free for future insertions and updates [53]. For sequential inserts, a leaf page will not be updated again after it has reached its full capacity because none of subsequently inserted keys will fall within the key range of that page. Consequently, pages can be filled completely and will not risk being splitted. In InnoDB, index pages are about 15/16 full for sequential inserts. For random inserts, however, subsequently inserted keys may very well fall within the key range of any of the previously allocated pages, causing node splits which lead to a lot of wasted space due to internal fragmentation. Pages in InnoDB for random inserts are therefore between 1/2 to 15/16 full [53].

The impact of sequential and random inserts on an InnoDB database in practice can be witnessed in Figure 2.12 and 2.13. Figure 2.12a clearly shows that sequential inserts cause pages to be allocated and updated in order, and filled completely. The pattern shown in Figure 2.12b for random inserts stands in strong contrast. After the inserts, all pages have been recently modified. While this in itself is unfortunate as it implies a lot of random writes, it also leads to a larger database size, seen from the increased number of allocated pages. The increased space usage is caused by the aforementioned internal fragmentation. Figure 2.13a confirms that sequential inserts cause practically all pages to be completely filled, whereas Figure 2.13b confirms that random inserts yield more page splits and fragmentation; consequently, many of the

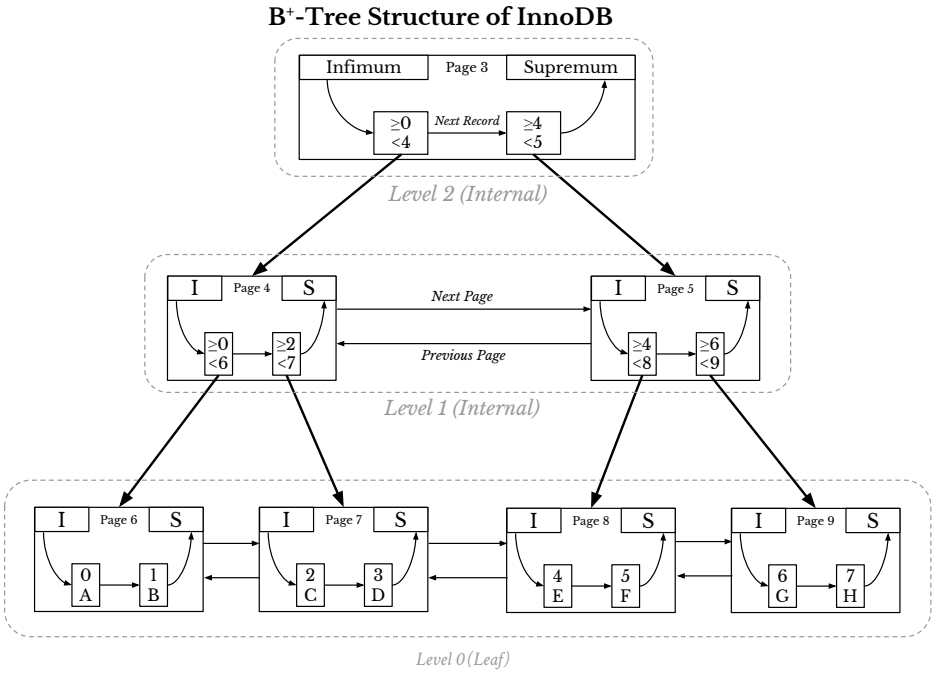
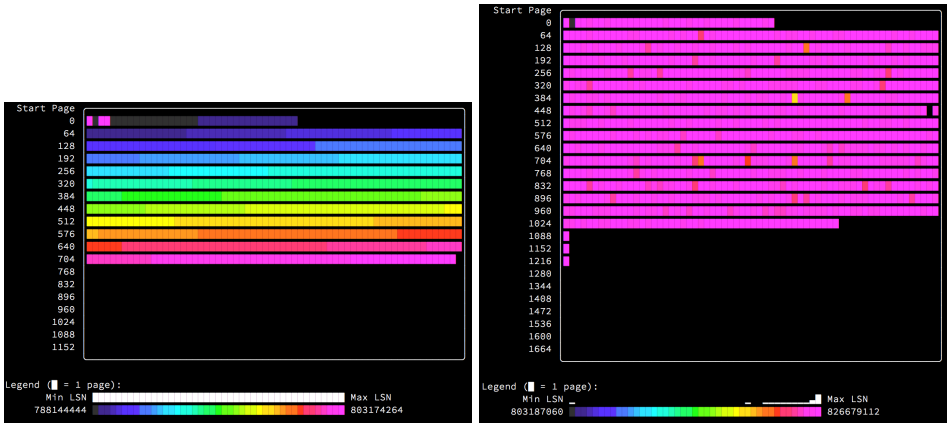


Figure 2.11: Illustration of the B+-Tree implementation in InnoDB. Illustration based on: [16]



(a) Sequential inserts. **(b)** Random inserts.

Figure 2.12: How random and sequential inserts update pages in a B-tree. The heatmap on the bottom shows the LSN for each page. Source: [17].

pages have a severely low fill factor. If some records in one of those pages are deleted and the fill factor drops below the `MERGE_THRESHOLD`, which is 50% by default, InnoDB will attempt to contract the index to free the page [53] – another time consuming operation which hurts overall performance.

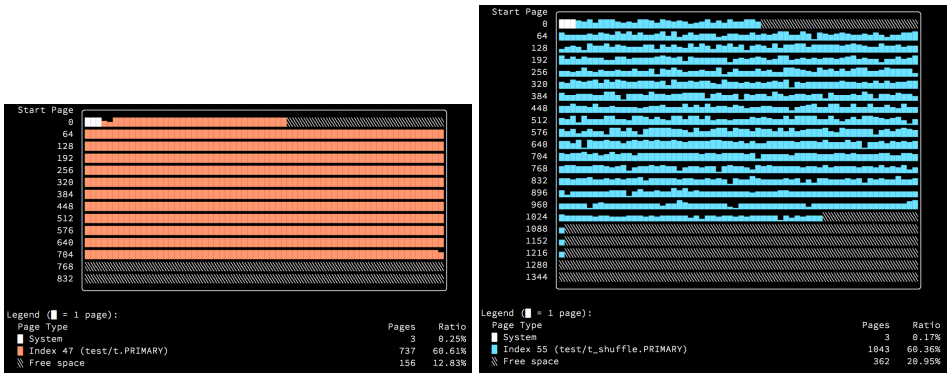
FB-Tree

Figure 2.14 marks two types of fragmentation: internal and sector fragmentation. Internal fragmentation occurs because the size of a node is larger than the size of the values it contains, as just mentioned. The FB-tree [38], a B-tree variant optimized for SSD, eliminates this type of fragmentation by allowing variable sized nodes and allocate just the required amount of space. However, SSDs consists of sectors, and it is not possible to allocate space in units smaller than a sector. This constrain results in sector fragmentation. Nevertheless, The FB-tree's approach helps to avoid most of the fragmentation because approximately just 68% of each node in a B-tree is filled on average [38].

Figure 2.15 shows one node with size of three segments and another of size two being evicted from memory. The FB-tree has a storage manager that reorganizes these for an out-of-place update and writes them to a free, contiguous space on the SSD. The out-of-place writes help to avoid wasted space between nodes. The idea is quite similar to the garbage collector in the flash transition layer, but the storage manager operates on sectors of the SSD rather than entire blocks.

2.4.3 Copy-On-Write B-Tree

The random writes incurred by random updates are caused by the in-place update strategy of the InnoDB storage engine. An in-place update means that an updated



(a) Sequential inserts. **(b)** Random inserts.

Figure 2.13: How random and sequential inserts affect fill factor of pages in a B-tree. The heatmap on the bottom shows the LSN for each page. Source: [17].

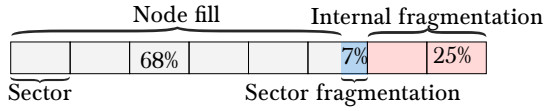


Figure 2.14: Types of fragmentation on SSDs.

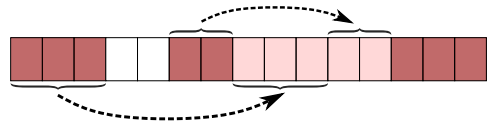


Figure 2.15: Illustration of how the storage manager in the FB-tree reorganizes data in sectors. Each block is one sector. Source: [38]

page is written back to its original position. Overwriting data in-place is possible on HDDs, but increases complexity of garbage collection and wear leveling on SSDs where data cannot be directly overwritten. An alternative is to update out-of-place by using copy-on-write (COW). A copy-on-write B-tree does not overwrite pages directly; instead, the updated page is written to a new location on disk. When the write has completed successfully, the old, original page is marked as obsolete and may be freed. Copy-on-write can be implemented in two ways: copy-on-write random (COW-R) and copy-on-write sequential (COW-S) [10]. Common to both of them is that updates are out-of-place, but they differ in their approach as to where to write updated pages.

The COW-R may overwrite previously freed pages with new, updated pages. The advantage is that deleting records will free up space that can be used in the immediate future. A disadvantage is that the tree becomes fragmented with time, thus increasing random I/O even for sequential writes and reads. The COW-R design is used by LMDB [10].

The COW-S design writes to only one or a few active log segments. It causes less random I/O than COW-R but requires background threads to copy valid data from previously written log segments; similar to how the garbage collector in an SSD cleans up partly old blocks and copies out any valid pages along the way to write them to new blocks. However, copying of valid data from old to new log segments increases both space and write amplification.

2.4.4 Write-Optimized B-Tree

A major issue that can arise in a COW B-tree with a standard B-tree structure is that all incoming pointers to a given page must to be updated whenever that page is written to a new location. For example, consider a COW B-tree implementation whose structure is similar to InnoDB's shown in Figure 2.11. If page 6 is updated and written to a new location, the pointers from page 4 and 7 to page 6 must be updated to persist the change. Thus, also page 4 and 7 must be fetched into memory, updated, and written to disk. However, also they are written to new locations, triggering incoming pointer updates from page 3, 5, and 8, all of which in turn must be written to new locations, and so on. Hence, a single small update in a COW B-tree gives rise to a snowball of pointer updates throughout the entire tree all the way to the root node, leading to massive write amplification.

The Write-Optimized B-Tree [29] introduces small modifications from the log-structured file system (LSF) to the B-tree to make out-of-place updates more efficient. To prevent the snowball of pointer updates, the write-optimized B-tree introduces fences to all nodes and removes all pointers between siblings, preserving only the essential parent-to-child pointers. Each leaf node holds two fences. The fences define the range of keys that may be found and inserted into that node. One of the fences is an inclusive bound, the other an exclusive bound. The fences are also stored in the leaf node's parent as separator keys. Internal nodes thus consist of fences at the edges and separator keys within. Stored between all adjacent separator keys is a parent-to-child pointer, pointing to one of its children. The fences make it much faster and simpler to handle page migrations, splitting a node, or merging two nodes. Moving a node to a new location requires no updates to its siblings; the only pointer

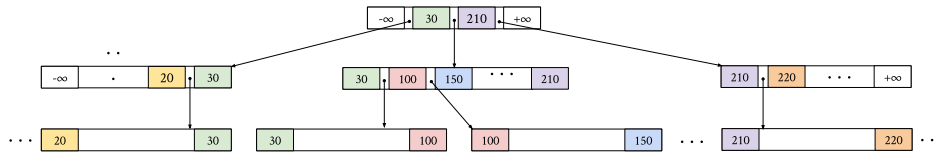


Figure 2.16: Illustration of the write-optimized B-tree. Illustration based on: [29]

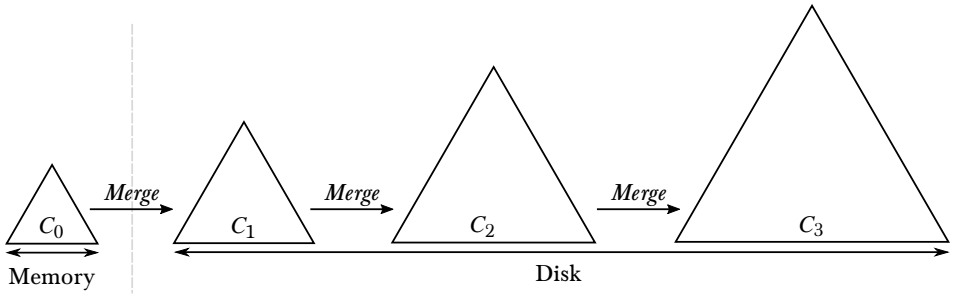


Figure 2.17: The concept of an LSM-Tree.

that must be updated is its parent's parent-to-child pointer. This update is acceptable since the parent is in memory as it must have been visited during the search down the tree.

Fences require more space than the pointers in a B$^+$-tree. But fences also make it very easy to apply prefix and suffix truncation to reduce space. If a node is split, it is necessary to propagate only the minimal prefix of the branching key, not the entire key. Because fences define the highest and lowest possible key values in a given node, the keys stored in that node can be compressed using prefix truncation. If both truncation techniques are implemented, the nodes in a write-optimized B-tree can be of approximately the same size as the ones in a B$^+$-tree.

2.5 LSM-Tree

The log-structured merge-tree (LSM-tree) was introduced in 1996 as a key-value data structure optimized for write-heavy workloads. The original paper [54] on the LSM-tree describes the subtrees as components, of which C_0 is the in-memory component and C_1, C_2, ..., etc. are on-disk components. An LSM-tree with more than one component on disk is called a multicomponent LSM-tree. Inserting a record into C_0 has no I/O cost in itself, but with time as C_0 fills up and reaches its maximum size, its entries must be merged into the C_1 tree through a so-called rolling merge process.

Records in the in-memory C_0 components are stored in order and updates are performed in-place. By contrast, updates to the on-disk components are performed out-of-place to ensure sequential writes. Batching records together in memory enables the LSM-tree not only to replace small random writes by larger sequential writes, but also more easily perform write requests of specific sizes – such as the clustered block

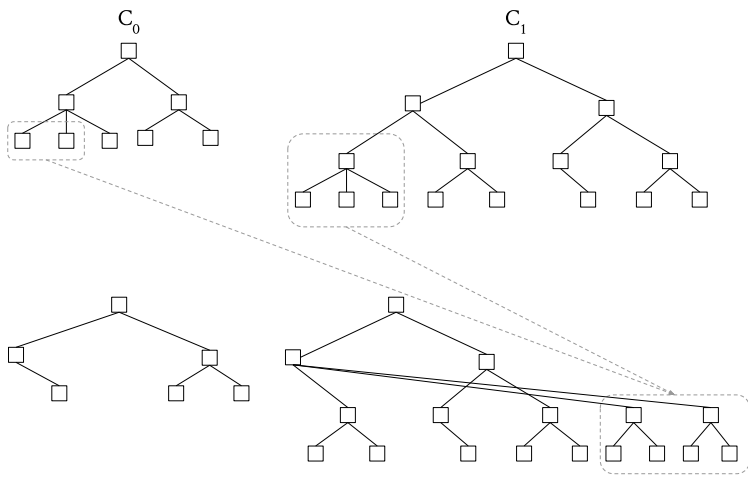


Figure 2.18: Illustration of the rolling merge process in an LSM-tree. Illustration based on: [54].

size discussed in Section 2.2.

Figure 2.17 illustrates the concept of a multicomponent LSM-tree. The components contain data in chronological order. New data is first inserted into to C_0. When C_0 reaches a defined threshold size, all its entries are merged into the larger C_1. C_1 continue to grow until it reaches its maximum size, at which point entries begin to merge from C_1 to C_2, and so on. Hence, newest entries are found in C_0, oldest entries in C_3. Except for C_0, each component is larger than its preceding component.

Figure 2.18 shows an example of a step in a rolling merge process in which entries from the C_0 are merged into C_1. The rolling merge process first allocates space for a block referred to as the emptying block, which acts as a buffer for entries from C_0, and another block called the filling block, into which the merged entries from C_0 and C_1 are inserted. The merge process iterates through entries in C_0 and C_1 to produce an output in which entries are stored in-order. When the filling block is full, it is appended to the end of C_1, and a new filling block is allocated. The process runs until all entries from C_0 are merged into C_1.

Because an entry is stored only in volatile memory between time of insertion into C_0 and time of merging into C_1, a recovery file saved to secondary storage logging all insertions into C_0 is required to ensure durability in the event of a system failure. The recovery file is often implemented as a write ahead log (WAL) into which every update is appended to ensure sequential writes. An append only design helps to keep the design simple and efficient. As soon as an entry in C_0 has been successfully merged into the C_1 tree and written to disk, its entry in the WAL is obsolete and can be discarded.

2.5.1 Costs

Whereas an insert in a B-tree involves at least one I/O operation (write back of the leaf node), the batching of entries in C_0 in an LSM-tree amortizes the I/O cost of a single insert over multiple inserts. The degree of amortization depends on the size of entries and size of components. If M is the average number of entries from C_0 inserted into a leaf node page in C_1 during a rolling merge as depicted in Figure 2.18, then M is determined by

$$M = \frac{S_p}{S_e} \times \frac{S_0}{S_0 + S_1}, \tag{2.2}$$

where S_e is the size of a single entry, S_p is the page size of the disk, and S_0 and S_1 are sizes of components C_0 and C_1, respectively. Thus, M increases with larger disk page size, smaller entries, or a bigger C_0 component. Because large, sequential writes are desirable, M should not be too small as then only a few number of entries would be merged into each C_1 page. Seeing that M becomes smaller the larger S_1 is compared to S_0, S_0 and S_1 should not differ too much. That, however, is easier said than done since the capacity of disks are usually vastly greater than that of RAM. This is why C_1 is broken up into smaller components to form a multicomponent LSM-tree. The size ratio of adjacent components in such an LSM-tree will be more beneficial, yet still allow an overall large database.

2.6 RocksDB

RocksDB is an open source embedded key-value store that uses an LSM-tree to organize its data. Its main goal is to be an optimized database engine for SSDs. Development of RocksDB was initiated by Facebook in 2012 as a fork of Google's LevelDB 1.5, and was open sourced in November 2013 [7]. A number of features were added and improved, including:

- Multithreaded compaction
- Multithreaded insertions into the memtable
- Extensive control over bloom filters
- Improved write lock
- Better utilization of flash storage (SSD and memory)

Particularly the increased parallelism is a major improvement that has resulted in significantly better write performance with lower latency and higher throughput. Running multiple compaction processes in parallel utilizes the parallelism available on today's multi-core processors, as well as the disk bandwidth. On fast flash storage devices, write rates have been observed to increase as much as 10 times compared to single-threaded compaction.

RocksDB also offers a plethora of configuration options to let users effortlessly customize the database without having to spend valuable time rewriting components.

Figure 2.19: The block-based table format in RocksDB.

Storage formats, compression algorithms, compaction style, and Bloom filters are some of the most essential elements that are easily configurable and that have a significant impact on read and write performance, and space usage as well.

SSTable

The components in the original LSM-tree from Section 2.5 are in RocksDB implemented and referred to as SSTables. The default SSTable format in RocksDB is the block-based table format which stores key-value pairs in sorted order along with metadata and indexes. The layout of the block-based table is shown in Figure 2.19.

Key-value pairs are partitioned across data blocks located at the beginning of the file. Next are the meta blocks holding Bloom filters, statistics, compression, and other metadata. The metaindex block is an index to the metablocks and contains one entry for every other meta block, whereas the index block contains one entry for every data block. The key of each entry in the index is a string that is greater than or equal to the last key in that data block and before the first key in the successive data block, enabling a search to efficiently locate the correct data block through the index.

Memtable

The in-memory component C_0 from the original LSM-tree is in RocksDB called memtable. All updates are first inserted into the memtable. The memtable accepts new updates until it reaches its maximum size, at which point RocksDB creates a new memtable into which subsequent updates are inserted. The old memtable is marked as frozen, meaning that it is immutable, and all its entries are written to disk as an SSTable, either immediately or at a later time, depending on the configuration.

The default index structure for the memtable is a skip list that stores its entries in sorted order. Having all key-value pairs stored in sorted order is beneficial so that the memtable can be searched using binary search, and is efficient when all records are to be flushed out to disk in an SSTable.

Write Ahead Log

All updates to the memtable are also appended to a recovery file to ensure durability in the event of a crash, in which case the recovery file is simply replayed on reboot to restore the database. The recovery file is implemented as a write ahead log (WAL) that serializes memtable operations to persistent storage as log files [22]. When a memtable is full and flushed to disk the corresponding WAL file is first achieved and then later purged entirely from disk. There may be a large number of WAL files at

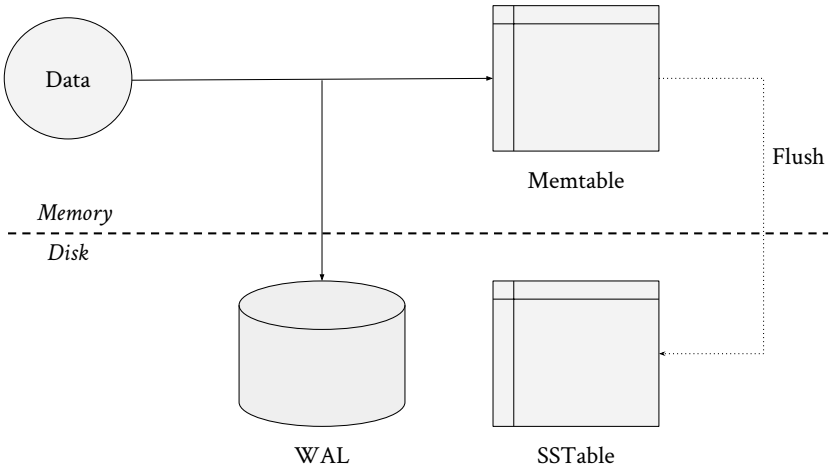


Figure 2.20: Illustration of how RocksDB inserts updates to the memtable and the write ahead log before they are flushed to disk in batches as SSTables.

any given time, and they are all stored in the WAL directory with a sequence number to ensure that the operations are replayed in the correct order during recovery.

Write Ahead Log Format

A WAL file consists of several blocks, each containing a number of records of variable lengths. If a record does not fit into the available space in a block and the size of the record is less than an entire block, the space is padded with empty data and the record is inserted into the next block; but if the record is larger than an entire block, the record will be split across several blocks. Reads and writes operate in unit of chunks, each of size of one disk block. Figure 2.21 shows an example of a WAL file with six records stored over two blocks.

Figure 2.22 shows the format of each individual WAL record. The different fields contain the following information:

- **CRC** 32 bit hash value of the data block computed using CRC algorithm.
- **Size** Length of the data block.
- **Type** Type of record. Valid types are: zero, full, first, last, middle. The type is used to allow records of size larger than one block to span over multiple blocks.
- **Data** Byte stream of data.

2.6.1 Compaction and Organization of SSTables

Compaction is the equivalent of the rolling merge process from Section 2.5. Compaction takes two or more SSTables as input, merges their entries together to obtain

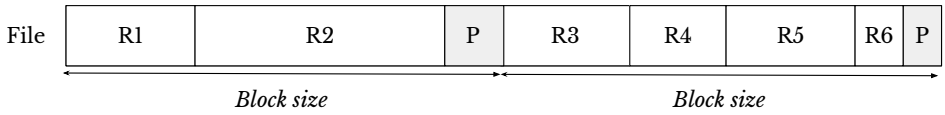


Figure 2.21: The write ahead log file format in RocksDB.

Figure 2.22: The write ahead log record format in RocksDB.

a total ordering defined by a comparator, and writes out the merged entries in new immutable SSTables. Keys marked as deleted are excluded from the output, and if a key exists in two or more input SSTables, only the latest value is included in the output. RocksDB offers three different compaction styles of SSTables: leveled, universal, and FIFO [60]. Each has its strengths and weaknesses, and which to choose depends on the workload, performance priorities, and hardware.

Leveled Compaction

Leveled compaction is the original compaction style implemented in LevelDB, and it remains so in RocksDB. The style is similar to the multicomponent LSM-tree. Leveled compaction stores SSTables in sorted runs. A sorted run contains multiple SSTables that do not overlap in key range and whose entries form a total ordering. Each sorted run is stored as a separate level, such as level 1 (L_1) and 2 (L_2) in Figure 2.23. The most recent data resides in level 0 (L_0), and the oldest data in the last level (L_2). One level in Figure 2.23 may be though of as one component in Figure 2.17.

Figure 2.23 illustrates the first three levels in a LevelDB or RocksDB database. Memtables are flushed to disk as SSTables in L_0. To increase the write performance of the flush operations, any merge operations are skipped, and the SSTables in L_0 are instead stored separately as sub-levels $L_{0,0}$, $L_{0,1}$, $L_{0,2}$ and may overlap in key range. The overlap may occur only in L_0; in higher levels, no SSTables in the same level will overlap in key range. A compaction process from level L_i to level L_{i+1}, denoted

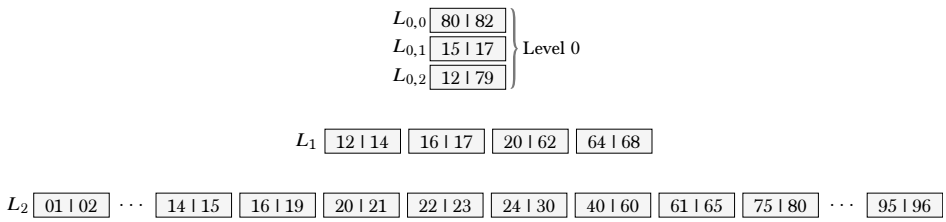


Figure 2.23: Illustration of leveled compaction in RocksDB.

$L_1 \rightarrow L_{i+1}$ from here on, picks an SSTable from L_i and merges its records with SSTables in level L_{i+1} that overlap in key range.

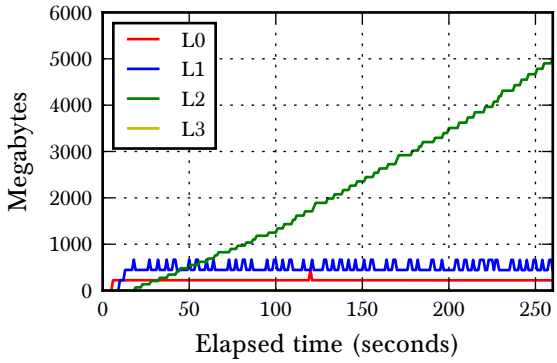
Due to the possible overlaps in key range in L_0, read operations must check all SSTables at L_0. At higher levels, maximum one SSTable at each level can contain any given key since there is no overlap in key ranges. Searching L_0 is thus much more expensive than searching higher levels; hence, the number of SSTables at L_0 should be limited. Where to set the limit becomes a tradeoff between write and read performance; many SSTables at L_0 allows rapid flushing of memtables from memory and great write performance, but decreases subsequent read performance. By contrast, fewer SSTables at L_0 causes more pressure on the compaction processes, which can decrease write performance if the compaction rate cannot keep up with the insertion rate.

Space amplification with leveled compaction Because it takes time for an update to propagate from the lowest level to the highest level, duplicate keys may exist in the tree. This leads to space amplification, which is the ratio of database size to the actual size of all its entries. For example, if database size is 2GB but it contains many duplicate keys so the valid entries actually occupy no more than 1.2GB, it has a space amplification of 1.67. In RocksDB, each level has a target size that is a constant multiple of the target size of the previous level. The default value is 10, meaning that if the target size of level 1 is 1 GB, the target sizes of level 2, 3, and 4 are 10 GB, 100GB, and 1000 GB, respectively. Determining the exact space amplification is nontrivial. But assuming that the database eventually reaches a steady state, i.e., the database size is stable or grows slowly over time by inserting about the same amount of – or little more – data than what is deleted, most data will reside in the last level, and the amount of data deleted and inserted in higher levels will be approximately equal. Therefore, the size of the last level serves as a good estimate of the size of the actual user data in the database at any given time, and the space amplification can be seen as the total size of the database divided by the size of last level. With target sizes as exemplified above, the space amplification would be $\frac{1+10+100+1000}{1000} = 1.111$ But note that to make the numbers in the calculation valid, an user would have to fine tune the target sizes to ensure that the last level would equal the size of the user data.

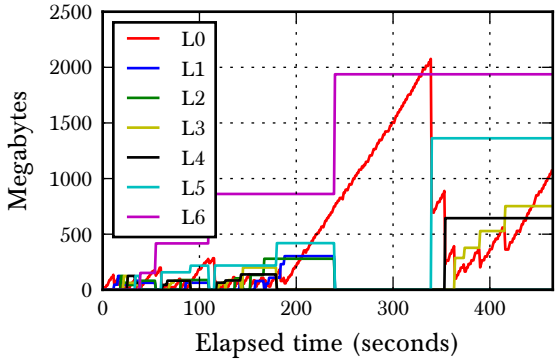
Universal Compaction

With leveled compaction, SSTables in a sorted run do not overlap each other's *key* range. By contrast, with universal compaction, SSTables in a sorted run do not overlap each other's *time* range but may very well overlap each other's key range because each SSTable can cover the entire key range. The SSTables in a sorted run still contain all entries in sorted order. Input to compactions are two or more adjacent SSTables from a sorted run, and the output is another sorted run whose time range ranges from the earliest entry in the first input SSTable to the latest entry in the last input SSTable.

The structure of an LSM-tree with universal compaction looks strikingly different from one with leveled compaction. Figure 2.24 shows how the levels in an LSM-tree evolve in size with the two compaction styles. The values were obtained from real benchmark statistics. With leveled compaction, compaction is triggered as soon as a



(a) Leveled compaction.



(b) Universal compaction.

Figure 2.24: Comparison of levels in an LSM-tree with leveled and universal compaction.

level size reaches its maximum size to merge some of its SSTables to the next level, causing a well-defined growth pattern. The LSM-tree with universal compaction in Figure 2.24b exhibit an entirely different structure. The tree reaches a much deeper depth faster, but with many empty levels in between the lowest and highest level.

FIFO Compaction

The simplest compaction style of the three. All files are stored in level 0, but data is kept only for a certain amount of time, after which it is deleted, effectively causing the database to behave similar as a time-to-live cache. Size is also a constraint; the oldest SSTable is deleted as soon as the database size reaches its confiured maximum size.

2.6.2 Bloom Filter

Bloom filter is a data structure used to rapidly determine whether an element may or may not be present in a dataset. That the filter can only determine whether an element *may* be in a dataset, and not whether it actually *is*, gives rise to false positives. A false positive occurs when the filter indicates an element to be in a dataset but is actually absent. This is a consequence of its simple, but space efficient design.

The Bloom filter consists of a bitmap of m bits and k unique hash functions, of which each function should map to one of the m bits with a uniform random distribution. To add an element, all k hash functions are called to get k bitmap positions. The bits at these positions are set to 1. To check for an element, all k hash functions are called to get k bitmap positions. If any of the bits at these positions is 0, the element is guaranteed not to be in the set. If all bits at these positions are 1, the element may be present, because they must have been set to 1 at the time the element was inserted. However, the bits may also have been set to 1 because hash values of other elements also mapped to these positions. If so, it is a false positive.

An example of a Bloom filter with 5 bits per key, 3 hash functions, and 3 keys (x, y, z) is shown in Figure 2.25. Checking for a yields a negative result because not all bits mapped to by the hash functions are 1. By contrast, all hash functions for z map to positions at which the bits are set to 1, so z may – and is, in this case – present. But the hash functions also map b to positions at which all bits are 1, despite b was never added; hence, a false positive.

To achieve a uniformed random distribution, it is necessary to scale the Bloom filter with the number of entries in the dataset. Otherwise, the false positive rate will grow high as entries are added to the dataset, and eventually any benefits of the filter would be lost. Given a Bloom filter of size appropriate to the number of entries, the false positivity rate is usually no more than 1% – more than low enough for most use cases to justify the space and time required to maintain the filter.

RocksDB offers Bloom filters either on each block in each SSTable (block-based filter) or on entire SSTable files (full filter) [21]. Full filter are more effective because they allow checking the Bloom filter prior to the SSTable index, potentially saving one lookup. The drawback is that all keys from all SSTable blocks must be in memory when creating the Bloom filter.

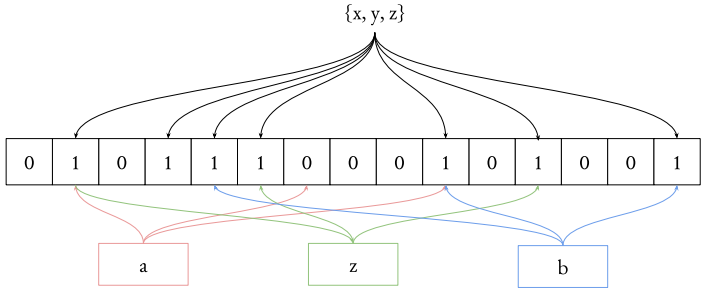


Figure 2.25: Example of a Bloom filter with the dataset x, y, z and $m = 15$ bits. There are $k = 3$ different hash functions.

2.7 MyRocks

MyRocks is an open source project initiated by Facebook that integrates RocksDB as a new MySQL storage engine [45]. Facebook uses MySQL to manage many petabytes of data, but the serious space and write amplification of InnoDB became more of a problem as Facebook started to store more of their databases on flash storage. RocksDB was considered the most viable alternative with its superior performance in both areas. However, RocksDB does not support important features such as replication or an SQL layer. As a result, development of MyRocks began to use RocksDB as backend storage for MySQL, while still benefiting from all features of MySQL.

MyRocks has already proved to be reliable and viable alternative to InnoDB. MyRocks is currently used in production at Facebook. In addition, MariaDB, a database made by the original developers of MySQL and currently the fastest growing open source database [44], is currently working on adding support for MyRocks to benefit from its superior compression ratios and I/O performance, fast bulk loading of data, and read-free replication of slaves [57].

2.8 Write Amplification

The exact definition of write amplification depends on the topic. For SSDs, the type of write amplification usually discussed is *flash* write amplification caused by internal garbage collection and wear leveling, topics that were reviewed in Section 2.1.2. The type of write amplification to be adressed here applies for database systems. It is defined as the ratio of bytes written to storage versus bytes written to database [21]. Hence, a write amplification of 1 means that number of bytes written to disk is equal to number of bytes written to database, whereas a write amplification of 2 means that twice the amount of data written to the database is written to disk. Write amplification can be determined from observing the write speed to the database and the write speed to disk [21]. For example, if data is inserted into the database at a rate of 10MB/s, but the write rate to disk is 50MB/s, the write amplification is 5. High write amplification not only hampers write performance, but also shortens the lifetime of SSDs because

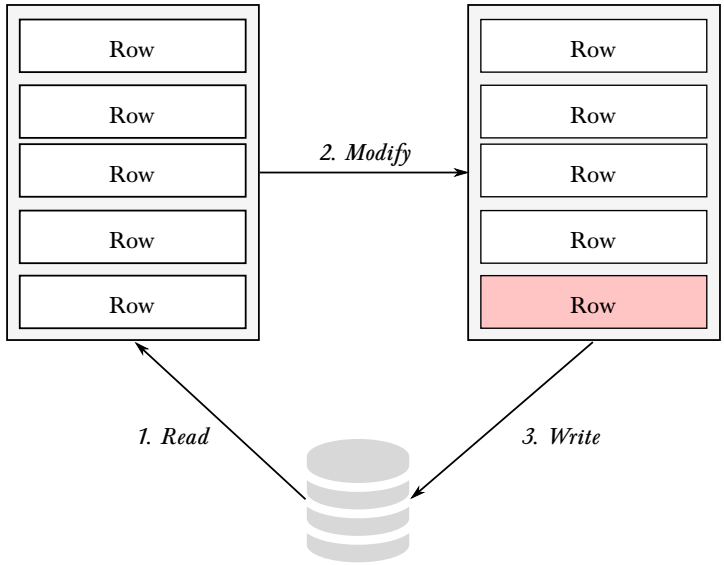


Figure 2.26: Illustration of write amplification for a B-tree in InnoDB. Illustration based on: [46].

data requires more pages, increasing the number of erase-write cycles when the data is later deleted or updated.

2.8.1 B-tree

B-trees usually have a high write amplification. Figure 2.26 explains why. No matter how much of the data is updated in a page, the entire page must be fetched to memory, modified, and written in its entirety back to disk. For example, if 100 bytes are changed and the page size is 4096 bytes, the write amplification is $4096/100 \approx 400$. For a B-tree, the write amplification is inversely proportional to the size of the entries, making it less of a concern the larger the values. Because of this, the write performance of a B-tree is often better the larger the entries.

2.8.2 LSM-tree

The write amplification of an LSM-tree is usually much lower than that of a B-tree. Figure 2.27 shows an example of the write amplification in a RocksDB database configured to use leveled compaction.

It is hard to determine the exact write amplification in an LSM-tree because it depends on the size ratio between adjacent levels, the key range of the SSTables that is about to undergo compaction, and type of compression for each level. The SSTables in L_0 will eventually be merged together to generate a list of non-overlapping SSTables in level L_1. With time, SSTables will undergo another compaction to L_2, and so on. Figure 2.27 shows that the key range of the third SSTable in L_1 overlap

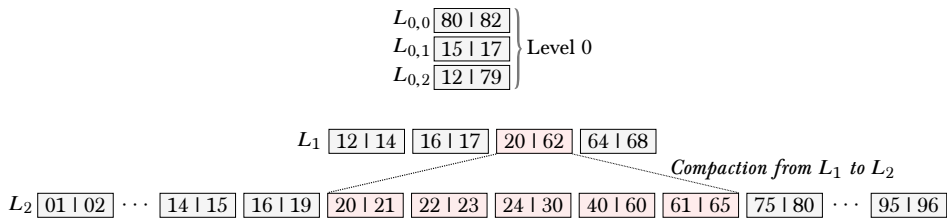


Figure 2.27: Illustration of write amplification for an LSM-tree with leveled compaction in RocksDB.

with four SSTables in L_2, generating a write amplification of 4. However, the write amplification for any of the other SSTables could have been less, equal, or greater depending on their key range and the number of overlapping SSTables at L_2. The write amplification can be lower if the compressio algorithm used for L_2 achieves higher compression ratios than the compression algorithm for L_1.

The amplification factor (AF) is the size ratio between adjacent layers:

$$AF = \frac{\text{Size}(L_{i+1})}{\text{Size}(L_i)} \tag{2.3}$$

The default amplification factor in RocksDB and LevelDB is 10 [21]. With a constant amplification factor, the number of SSTables in each level grows exponentially larger. In the worst case scenario, the key range of an SSTable at L_i overlaps with the key range of all SSTables at L_{i+1}, which gives write amplification equal to the number of SSTables at L_{i+1}. Consequently, the amplification factor affects the write amplification directly. For a database with k levels, the total write amplification for an element to reach level k can amount up to [72]:

$$k \times (AF + 1). \tag{2.4}$$

In contrast to the B-tree whose write amplification is inversely proportional to the record size, the write amplification of an LSM-tree is constant and independent of the record size. This property makes LSM-trees especially well-suited for small to medium sized records.

The universal compaction is a better choice than leveled compaction if low write amplification is critical. However, it comes at the expense of higher read amplification and space amplification.

Garbage Collection

A lower amplification factor would be a natural choice to reduce write amplification in Equation 2.3. However, a tradeoff must be made between the amortized cost of garbage collection – which is reduced the larger the amplification factor – and the write amplification. The cost of garbage collection is explained by the steps [54] [66]:

1. An item is moved from L_i to L_{i+1} at most once. All merge costs arise from such operations.

2. The compaction cost for an SSTable at L_i is proportional to the cost of scanning the overlapping key range at L_{i+1}. On average, the cost is $R_i = \frac{|L_{i+1}|}{|L_i|}$
3. Given N levels, the total size of all data in all levels is approximately $|L_0 \times \prod_{i=1}^{N-1} R_i|$.

It is desirable to keep the cost of R_i as low as possible, but at the same time keep the total index size constant. The optimal is to set $R_i = \sqrt[N-1]{\frac{|data|}{|L_0|}}$. It follows that the amortized cost of insertion is $O\left(\sqrt[N-1]{\frac{|data|}{|L_0|}}\right)$

2.9 Write Stalls and Write Stops

In an LSM-tree, there is a risk that the background flush threads and background compaction threads may not be able to keep up with the rate at which records accumulate in memtables. If no limit is imposed, the number of memtables and SSTables can grow unbounded. If compactions fall behind, duplicate keys may begin to accumulate over several levels, and deleted keys will not be removed from SSTables in higher levels. Both cause an increase in space amplification. Also read amplification will increase from the scattered SSTables and the inefficient index structure. Eventually, to allow compactions to finish, the database may decide to reduce the insert rate to the memtable or to stop further inserts altogether. The first scenario is called a write stall, the other a write stop. It it usually desirable to impose write stalls before write stops such that writes can continue to operate as normal, just at a lower throughput.

There are several parameters in RocksDB that can be adjusted to control write stalls and write stops. Some of these are [23]:

- **Number of memtables**. Writes are stopped if number of immutable memtables is equal to or greater than `max_write_buffer_number`. Writes are stopped until the number of immutable memtables falls below the limit again. Further, if `max_write_buffer_number` is greater than 3, and number of immutable memtables is equal to or greater than `max_write_buffer_number - 1`, writes are stalled.
- **Number of SSTables in level 0**. Writes are stalled when number of SSTables in level 0 is equal to or greater than `level0_slowdown_writer_trigger`. Writes are stopped when the number of SSTables in level 0 is equal to or greater than `level0_stop_writer_trigger`.
- **Number of bytes awaiting compaction**. Writes are stalled when number of estimated bytes pending for compaction is equal to or greater than `soft_pending_compaction_bytes`. Writes are stopped when the estimate is equal to or greater than `hard_pending_compaction_bytes`.

Common to all is that any write stall will reduce the maximum write rate to the configured `delayed_write_rate`. If maximum number of memtables is easily reached, `max_write_buffer_number` can be increased to permit more immutable memtables,

given sufficient memory space. `max_background_flushes` controls the number of background threads that flush memtables to L_0; it can be increased if disk bandwidth and CPU is not fully utilized. If a bottleneck is caused by too many SSTables in L_0 or too many pending compaction bytes, options to experiment with are `max_background_compactions` to control number of background compaction threads, `write_buffer_size` to set the size of the memtable (a large memtable will reduce the write amplification), and `min_write_buffer_number_to_merge`.

2.9.1 Gear Scheduling

More refined approaches have been developed and proposed to prevent write stalls. One is the gear scheduler, introduced with the bLSM-tree [66], which synchronizes downstream merges with incoming writes. The gear scheduler is illustrated in Figure 2.28. In the bLSM-tree, the fraction of C_0 in use acts as an progress indicator. Both an upper and a lower threshold exist. If the size of C_0 falls below the lower threshold, downstream merges are paused. If it rises above the upper threshold, backpressure is applied to reduce insert rate. Such an approach ensures that records are merged smoothly from the lowest to the highest level in an LSM-tree.

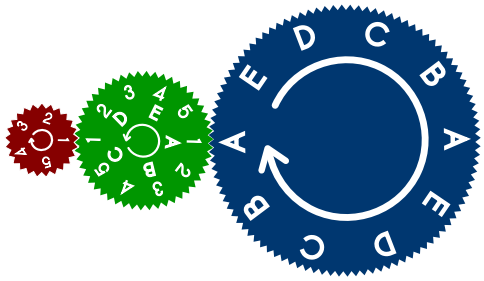


Figure 2.28: The gear scheduler in the bLSM-Tree. Source: [66]

2.10 Read Performance

Just as writes should have as little overhead as possible for inserts, it is desirable to minimize the I/O cost for reads. Read amplification is used to determine the I/O cost involved in fetching a record from disk. The RocksDB documentation defines read amplification as number of disk reads per query [21]. For example, if a single query reads 10 pages, the read amplification is 10. Because disk reads are performed in unit of pages, at least one page must be fetched. As it can be difficult to record the number of pages accessed in each query, the read amplification can be estimated from number of bytes read from disk divided by page size .