A Survey of Solid State Drives and Databases

Chan Jie Ho	Zhiping Liang	Geng Liu	Ran Lu
961948	1071095	1035248	1200134

{*chanjieh, zhipliang, glliu, ralu*}@student.unimelb.edu.au

ABSTRACT

Since the emergence of the SSD, the increasing popularity of SSD has changed storage systems, and SSD is now widely used in DBMS to improve performance. Many researchers are working on building a database that has better performance with the help of SSD. This paper is a survey of SSD and databases with a detailed discussion on various topics from different DBMS and SSD architectures to indexing and query algorithms. Finally, current trends and future directions are also discussed in this paper.

Keywords: SSD; DBMS; Hybrid; Caching; Tiering; Indexing; B+Trees; Sorting; Query Processing; Joins

1. Introduction

HDD was once the only choice as database storage hardware, but as technology has evolved, a new class of storage devices has emerged. SSDs offer many advantages over HDDs, including further improvements in reading speeds and reduced power consumption (*"SSD vs HDD: Which Is Best for You? - Intel"*, n.d.). Furthermore, because SSDs do not have the exact mechanical mechanisms as HDDs, the advent of SSDs has improved DBMS performance, such as faster read speeds and shorter query times. However, SSDs also have many drawbacks, including price and lifetime are the reasons why SSDs cannot be further promoted (Michael, 2021). Therefore, improving the architecture of SSDs to improve the performance of databases is a major research trend.

1.1. Architecture

This section will discuss the relationship between SSD architectures and databases and the impact of improving SSD architectures on databases. Here, we classify the improvement directions into two categories, i.e., reducing consumption and the I/0 latency by improving or designing new SSD architecture.

FlashSSDs have also implemented internal parallelism, as depicted in Figure 1.1. It shows FlashSSDs adopt multiple channels, each linked to multiple flash memory chips. Hence two kinds of internal parallelism are channel-level parallelism between numerous channels and package-level parallelism

between ganged flash memory chips in a chunk (Kim et al. 2018). The overall performance can be improved by multiplying the number of channels and flash chips in each chunk and applied by algorithms.

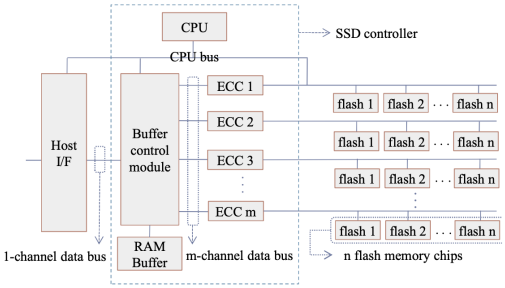

Figure 1.1 The internal architecture of flashSSDs

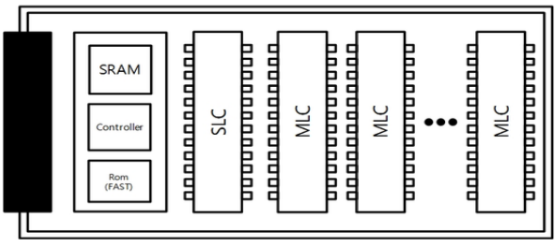

Figure 1.2 The structure of the hybrid flash SSD

1.2. Motivation

1.2.1. Reduce Consumption

SLC (single-level chip) and MLC (multi-level chip) are types of flash memory: SLC has higher performance and longer lifetime, while MLC has larger capacity and lower price. Nam et al.(2010) proposed a hybrid flash SSD (Figure 1.2) based on FAST FTL, where the benefits of both chips are combined. After modifying the original FAST algorithm (algorithm details out of scope), it is applied to the scheme using SLC chips to provide fast and durable writes and MLC chips for storage. The research team's experiments showed that the hybrid flash SSD solution far outperformed the pure MLC flash solution in terms of performance and price. It will significantly reduce DBMS expenses, allowing us to get higher performance and larger capacity at a lower price.

Since DBMSs need to read and write data quickly and persistently, their reliability is critical to the SSD or storage device used to store the data. Generally, non-volatile RAM (NVRAM) is implemented with the battery-backed DRAM and will be used to temporarily store data until the system determines the write request. However, DRAM is costly and can occupy a DIMM slot. Thus, Bae et al. (2018) proposed a novel NAND flash SSD architecture: a dual-byte and block-addressable solid-state drive (2B-SSD). This new architecture allows for access to the same file with two independent byte and block I/O paths. It also allows direct access to files on the storage device through the designed application. This new architecture allows users or DBMS to enjoy the advantages of NVRAM without taking up DIMM slots, allowing storage devices to add more RAM space and enhancing the DBMS performance. This new architecture has been tested to provide up to 2.8x throughput gain with no data loss.

1.2.2. Reduce latency

In addition to improvements in the areas mentioned above, some research teams have also focused on reducing latency. For example, Kim et al. (2021) have combined the benefits of zero-copy and page cache

and designed a new I/O stack. This new design can be used in modern low-latency SSDs and reduces the memory copy overhead. It also takes advantage of the kernel page cache through selective prefetching. These optimisations improve the hit rate and throughput of the page cache. Experimental tests have shown an 11% increase in throughput when database applications use this design.

2. Hybrid Structure

Using SSD as a replacement for HDD has become very popular in many cases. However, many DBMS adopted hybrid structures composed of HDD and SSD devices considering the trade-off between capacity and read-write speed. The two main categories of this kind of hybrid structure are SSD caching method and the SSD tiering method.

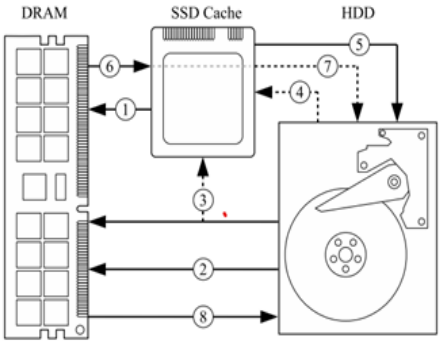

Figure 2.1 Dataflows in SSD caching

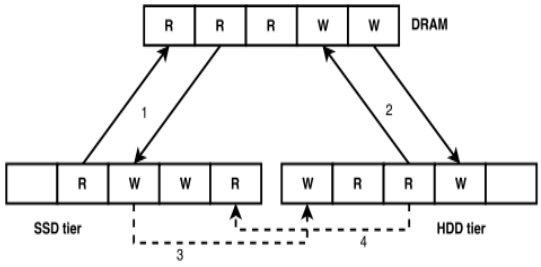

Figure 2.2 Dataflows in SSD tiering

2.1. SSD Cache method

To improve the overall performance, SSDs have been widely used as a cache between the DRAM and HDDs to keep the hot read data or the write data (Niu et al., 2018). Figure 2.1 shows the common data flow inside a caching system. When the read request arrives, its accessing data is located in the SSD. The read request can then directly access the data from the SSD (Process 1). If the data are located in HDD, the DRAM will retrieve the data from HDD (Process 2), and if the data are identified as hot data, they will be moved to the SSD device (Process 3). A background data migration may then happen when the data is identified as hot (Process 4), and the write data can then be flushed to the HDD from the SSD cache in the background (Process 5). Based on the different methods handling the new write requests, the caching methods can be categorised into SSD as Read-Only Cache and SSD as Read-Write Cache.

2.1.1. SSD as Read-Only Cache

There are many different storage architectures taking Read-Only Cache policy, such as MOLAR (Liu et al., 2013), LMsT (Tai et al., 2015), ETD-Cache (Dai et al., 2015). In this type, when a new write request

arrives and the accessing block is not located in SSD, the request is completed only when the data is recorded into HDD successfully through process 8. If the data was already cached into the SSD, the data in HDDs need to be updated, and the data in SSDs need to be discarded or updated.

To reduce the number of useless write operations, MOLAR introduced a method to wisely identify the data hotness and select the evicted data blocks from DRAM to SSD based on a control metric and a demotion count (Liu et al., 2013). Zhao, Qiao and Raicu also proposed a heuristic file-placement algorithm to improve the cache performance by using the I/O patterns of applications on an HPC system (Zhao et al., 2016). Lastly, researchers also proposed a new idea, ETD-Cache, that could give the cached data more chances to wait for accesses and keep the hot data in SSD for a long time by giving them a longer expiration time (Dai et al., 2015).

2.1.2. SSD as Read-Write Cache

There are two types of write policy, which are write-through policy and write-back policy. However, the write-through policy in SSD caches is obsolete since it is designed for volatile caches and SSDs are non-volatile (Hoseinzadeh, 2019). Therefore, an SSD R-W cache only uses a write-back policy. In this kind of architecture, new write requests are performed in the SSD cache in process 6, and data will be flushed to the disk later. However, because of the later flush, data synchronisation problems could occur when the data is written to HDD. Therefore, period write-data flushing is often used to solve the data synchronisation problems.

Many architectures are using SSD as an R-W Cache, such as LARC, SVD and RAF. They use different methods to improve the overall performance: LARC reduces the write traffic to SSD by filtering out the less accessed blocks and keeping them out of cache as well as keeping popular blocks longer in cache (Huang et al., 2016); SVD adopted the idea of transforming the random write to the sequential writes, and thus the system could cache the write operations in the SSD without harming its performance or lifespan (Lee et al., 2015); RAF split the SSD to read and write caches. The read cache will only maintain random-access data evicted from the file cache to reduce flash wear and write hits. The write cache is a circular write-through log to respond to write requests faster and handle the garbage collection (Liu et al., 2010).

2.2. SSD Tiering methods

The main difference between cache methods and tiering methods is that the data recorded in the SSD cache is temporary. There is no need to flush or copy the data to the HDD, where, in contrast, data in the SSD tier is permanent. Tiering methods usually categorise the data into hot and cold based on many

factors. Then, the hot data will reside in the tier with better performance, such as the SSD tier, and the cold data will be stored into the capacity tier, such as the HDD tier.

Figure 2.2 is an example of a tiered structure composed of both SSD and HDD tiers. Operation 1 and 2 are the data allocation operations, which are actively performed by the host or the device with some specially designed algorithms. These algorithms will categorise the data and designate the data to a suitable tier, i.e., hot data allocated to the SSD tier and cold data to the HDD tier. Another critical operation in tiering structure is data migration, which is operations 3 and 4 in this figure. Data migration will be performed in the background and relocate the data between the two tiers to control the data flows to improve the overall system performance.

The main difference among tiering methods is using different data allocation and migration algorithms: GreenDM (Li, 2014) combines flashed-based SSDs with traditional HDDs in a storage system, and it offers tunable parameters which could be used in adapting for different workloads; Tiera (Raghavan et al., 2014) is a multi-tiered cloud storage system that supports many storage policies and desired metrics for making choices regarding tradeoffs; Online OS-level Data Tiering (OODT) distributed data among multiple tiers according to the access frequency, random access frequency, metadata access frequency and read access frequency. The result shows these techniques can improve performance up to 72% compared to a pure HDD system (Salkhordeh et al., 2015). In addition, some machine learning algorithms are used recently to track and predict the data or file automatically (Herodotou et al. 2019), and this approach uses incremental learning to refine the models when new data arrives dynamically.

2.3 Comparison

In caching methods, Read-Only cache can help prolong the lifespan of the SSD device since all the requests are directly passed to HDD. Meanwhile, the read performance can be improved because the cache space utilised by the write data is saved, meaning more cache space will be allocated for the read data. For Read-Write cache, the performance could be improved since SSD can keep data permanently, and data can be kept in SSD for a long time without demoting before the space is full. However, because the number of write operations to SSD is much larger than utilising SSD as Read-Only cache, the limitation of the SSDs write endurance considerably influences the R-W cache. What is more, a write-intensive workload will make a large number of requests to HDD, which could cause the disk to become too busy with short idle periods, and the data might not be able to flush to the disk during these short idle periods (Niu et al., 2018).

In tiering methods, since the data will be identified as hot or cold and stored into different devices, there will not be a copy of data, and then the frequency of data relocation is very low. It is usually less than one time per hour or per day to optimise the data location. Thus, it will take a considerable measure of time to

move the data between different devices. On the contrary, in caching methods, the data will be identified frequently, and the data location is optimised in a short time. However, the frequent data movements will increase the workload of both HDDs and SSDs and then influence the devices' lifespan, especially SSDs. Therefore, some new architectures have been created recently, trying to combine the tiering method and caching method, and this will be one of the directions of future trends.

3. Indexing

Index, a special purpose data structure, can speed up the access of contents via algorithms and has been extensively investigated in DBMSs (Akritidis et al., 2018). Since B+ Tree is closely related to databases with SSDs, any enhancement of the data structure will reduce the read/write latency. Consequently, we are looking into the variants of B+ Tree to solve its drawback and reduce the cost.

3.1. PIO-Tree - a multi-path search algorithm

By utilising channel-level parallelism, parallel synchronous I/O can query many nodes, which is possible only if a set of search requests are provided at once (Kim et al. 2018). Under the above assumption, Kim et al. (2018) designed a search algorithm, which processes multiple requests simultaneously while searching multiple nodes from top-down. Instead of examining one node via a single search request, the enhanced algorithm looks at entries in multiple nodes through parallel synchronous I/O. The process continues recursively until reaching and retrieving search requests from leaf nodes (Figure 3.2). Thus it could improve the efficiency with the combination of node size optimisation.

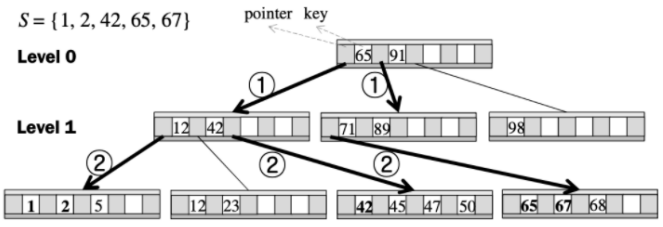

*Figure 3.2 Search requests **S** with two parallel synchronous I/Os*

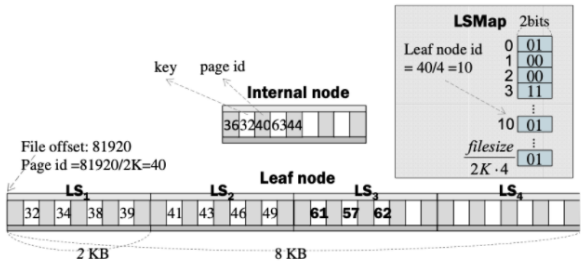

Figure 3.3 Leaf node with the size of 4

Although larger nodes enhance FlashSSD bandwidths, the latency of each I/O operation also increases; hence the tradeoff exists between shortened B+ Tree heights and increased latency (Kim et al., 2011). This issue is resolved by alleviating the cost of enlarged leaf nodes.

Kim et al. (2011) defined a resizing unit called Leaf Segment (LS) as in Figure 3.3, and each LS is the same size as that of a page and an internal node. Since key values in node entries are sorted by ascending order, the constraint is that nearly half of the entire leaf nodes need to be updated for every index-insert

operation (Kim et al., 2011). Here, it has been replaced by appending an index record right next to the most recently inserted operation queue (Figure 3.4) regardless of the order, thus reducing average page-write cost from *the number of pages / 2* to 1 page. Moreover, updating and deleting records are also appended following the same rule. As a result, the read latency of a node could be reduced significantly.

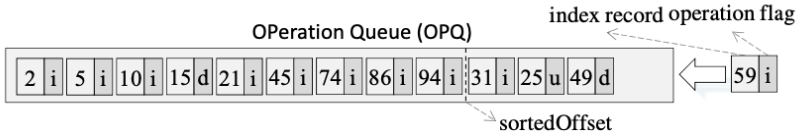

Figure 3.4 Each operation queue is stored in memory before the batch operation.

3.2. AB-Tree - a write-optimised index structure for bulk insertion

This method was found by Jiang et al. (2014), which aims to solve the problem of update propagation delay in B+ Tree due to its linked leaf nodes by inserting real data in both internal and leaf nodes and putting data entries into buckets in each node, as in Figure 3.5. Each bucket contains a data part for bulk insertion and a buffer part for caching update and delete. Moreover, Jiang et al. (2014) have adopted an approach to ensure that both parts' sizes are changed dynamically for different workloads. A modified node structure contains a pointer list of offset nodes associated with each bucket, a key list denoting the range of all buckets and a bucket list storing data entries. Therefore, the algorithm chooses the corresponding bucket and validates if it has reached a threshold. Otherwise, entities would be migrated to the leaf node of the bucket. Eventually, it inserts many entries and achieves the exact I/O cost of inserting one entry in the original B+ Tree by controlling the threshold.

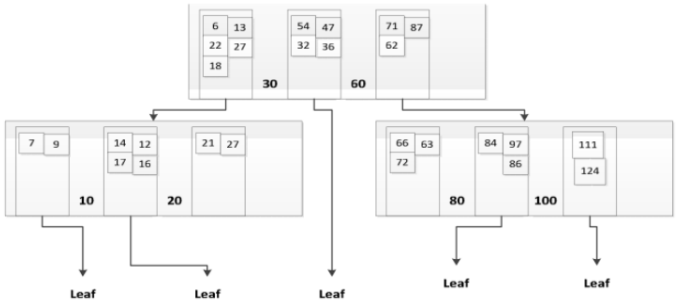

Figure 3.5 AB-Tree with a fanout of 3

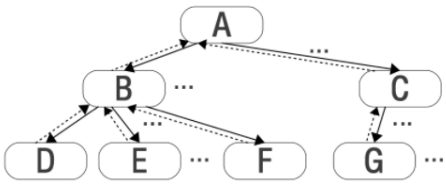

Figure 3.6 FB-tree with in-buffer dash pointers

3.3. FB-Tree - discarding all sibling pointers in leaf nodes

There are two kinds of pointers in a B+ Tree: child pointers and sibling pointers. Each sibling pointer is a pointer to the left/right leaf node with the nearest keys, and each update of a linked leaf node will result in rewriting the whole tree. With that, Jørgensen et al. (2011) from Denmark built an alternative algorithm to abandon the sibling pointers. Since any B+Tree operations such as range scan and node split can locate a

node's sibling pointers through its parents, the new algorithm achieves this by maintaining dash pointers to parents' in-memory buffer in Figure 3.6. The differences between B+ Tree and FB-Tree are:

- Every node in the buffer maintains a reference to its parent
- If a node splits or merges, pointers to the siblings are acquired through its parent
- A new node receives a virtual ID in the buffer and is evicted to disk with a real physical address

3.4. Comparison

B+ Tree Variants	PIO-Tree	AB-Tree	FB-Tree
Optimisation	Multiple simultaneous search requests and optimised node size	Put data entries into buckets in each node for bulk insertion and update	Break sibling pointers in leaf nodes and maintain dash pointers to parents
Pros	Reduce average write cost from the number of pages / 2 to 1 and shorten search time	Insert many entries via bulk insertion and achieve the exact I/O cost as inserting one entry	Each update is independent without rewriting the whole tree.
Cons	Larger nodes enhance FlashSSD bandwidths, but the latency of each I/O operation also increases.	It is challenging to decide the ratio of the buffer part of buckets for arbitrary workloads.	A large group of such writes cannot be collected into a single large write and cause problems.

4. External Sorting Algorithms

Commonly utilised for creating indexes and resolving queries (Laga et al., 2017), external sorting algorithms need to be used if the data does not fit into the main-memory capacity. They revolve around *runs* (data is split into chunks that fit in the main memory and sorted in-memory; a run is the intermediate file where the sorted chunks are written). These algorithms have two phases: the *run generation phase* and the *run merge phase*, where the runs are merged (and sorted). A key point to note is that their performance is highly dependent on how optimised their I/O cost is (Laga et al., 2017).

4.1. External MergeSort

External MergeSort creates runs in the generation phase and sorts them using buffers in the merge phase (further detail is out of scope). However, this algorithm is very I/O intensive (Laga et al., 2017), and while much research has gone into optimising traditional MergeSort, many usually assume HDDs and do not leverage the characteristics of SSDs.

4.1.1. Flash MergeSort (FMSort)

Hence, in 2016, Lee et al. have proposed FMSort, which utilises the high random I/O bandwidths that SSDs have due to exploiting internal parallelism to reduce I/O operations during the merge phase. Here, multiple data blocks are read simultaneously into main memory via multiple asynchronous I/Os. The order is calculated during the generation phase by sorting each data block's first tuple's key.

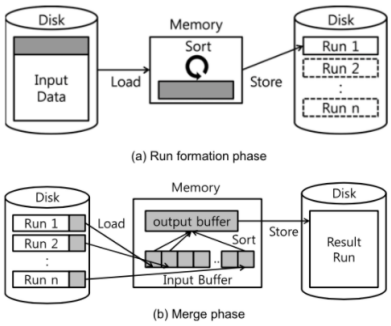

Figure 4.1 Traditional MergeSort

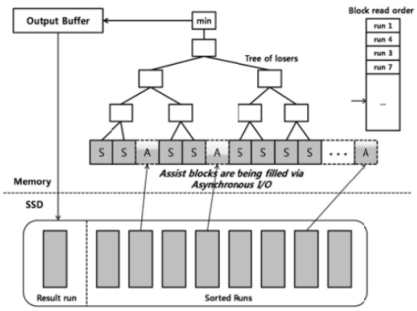

Figure 4.2. Memory Status of Merge Phase in FMSort

4.1.2. Merge ON-the-Run External Sorting (MONTRES)

With MONTRES, Laga et al. sought to improve performance in 2017 by optimising both phases to cut down on I/O cost at the cost of a small number of extra read operations.

For the generation phase, MONTRES uses random reads on SSDs to select blocks in ascending order based on their minimal value to create a min-index to allow for utilising a merge on-the-fly mechanism that evicts small values to the sorted file all through the phase to reduce the dataset size for the merge phase and adopting a continuous run expansion policy to create the largest possible runs to minimise the number of merge phase passes. MONTRES then performs the merge phase in one pass by continuously retrieving minimum values from the generated runs and outputting them to the final sorted file.

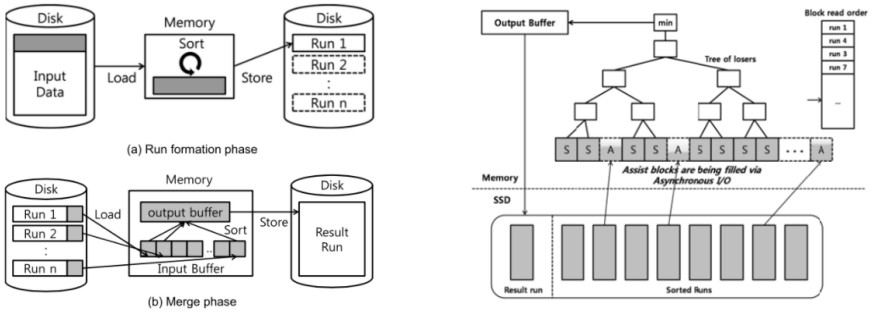

Figure 4.3 Generation Phase (left) and Merge Phase (right) of MONTRES

4.2. Active SSDs and ActiveSort

Data processing is commonly the host application's responsibility (Lee et al., 2016), like Hadoop. However, offloading parts of this process to the SSDs – Active SSD – can improve I/O throughput and remove extra data transfer, thus improving performance. Thus, Lee et al. proposed ActiveSort in 2016.

With ActiveSort, the merge phase only runs when the read requests to the final sorted data is requested by the host. Data is merged on-the-fly inside the active SSD using chunk buffers (further implementation details out of scope), and the final result of the join will then be returned to the host.

4.4. Analysis and Comparison

Lee et al. (2016) managed to outperform traditional MergeSort with FMSort by up to 4.87 times and MergeSort with double buffering up to 4.67 times. That said, the limitations of this algorithm are that it only focused on improving the merge phase and incurred an additional overhead when calculating the block read order in the generation phase. However, they justified this by stating that their research suggests that SSDs do not improve the generation phase since only sequential reads and writes are executed but can leverage its short access latency during the merge phase, where random read consistently occurs.

However, according to Laga et al. (2017), the generation phase can be optimised by directly generating some sorted data by relying on efficient random reads to search for minimal values or scanning the overall file many times to avoid write operations. They managed to improve on traditional algorithms by reducing execution times by >40% when file size to main-memory size ratio is large compared with MergeSort. They also noted that the continuous run expansion policy optimisation is efficient when a part of the data is partially sorted.

On the other hand, Lee et al. (2016) stated that on top of being easily integrated with Hadoop, ActiveSort also allows for concurrent computation in the SSD and the host application and processes I/Os efficiently by exploiting the internal architecture of SSDs, resulting in a 36.1% improvement in Hadoop applications

and a 40.4% reduction in writing operations, thus improving SSD lifespan. Also, ActiveSort transfers read and write data equal to the size of the data set, whereas MergeSort generates larger amounts since data has to be transferred between the main memory and the disk during the merge phase.

Method	Read (MB)	Write (MB)
ActiveSort	2048.3	2048.4
Merge sort	4085.9	4096.7

Table 4.1 Comparison of I/O Cost of ActiveSort and MergeSort

That said, Lee et al. (2016) noted that their implementation of ActiveSort gave a poor performance on small-sized records and when using fixed sizes for keys and records. However, they associated these limitations with their sub-par prototype SSD. Furthermore, ActiveSort also requires a lot of engineering effort in building an interface to deliver the data-specific metadata information to the SSD to enable the data to merge on the fly inside the SSD.

5. Query Processing

According to Roh et al. in 2017, reducing write operations has been heavily researched, but reducing the number of temporary write operations via optimising query processing can also be looked towards in prolonging SSD lifetimes. This is true especially with join queries as, with limited buffer cache sizes, joining large relations requires writing intermediate results to SSDs. Even major join-processing algorithms such as Hybrid Hash Join (HHJ) and Sort-Merge Join (SMJ), empirically best join algorithms, produce temporary writes.

5.1. DigestJoin

In 2009, by exploiting random reads in SSDs, Li et al. aimed to improve the overall performance of join queries with DigestJoin, where digest tables containing only the join attributes and tuple ids are used in a traditional join algorithm instead of the entire tuple, thus reduces the size of the temporary writes.

5.2. Advanced Block Nested Loop Join (ANLJ)

Then, in 2017, Roh et al. looked towards improving Block-Nested Loop Join (BNLJ) despite being sub-par to HHJ and SMJ to leverage the fact that BNLJ does not produce temporary writes. In BNLJ, for each outer relation block, all the tuples are compared with all the tuples of the inner relation. In contrast, ANLJ aims to remove temporary writes while being at least as good as HHJ and SMJ by changing how the outer relation is set and how the tuples are read (further detail out of scope).

5.3. Analysis and Comparison

In Li et al.'s paper (2009), DigestJoin outperformed SMJ in various system configurations (out of scope). However, a large issue with using digest tables is that the original tuples satisfying the join will have to be fetched post-join, which will increase the number of I/O operations. The researchers themselves stated that this increase might even exceed the amount saved in prior.

On the other hand, Roh et al. (2017) found that if the relations were sorted by join-keys, ANLJ outperforms SMJ, Grace Hash Join (GHJ), and HHJ. If not, the performance is on par, but only if the buffer size is sufficiently large, implying it may not perform well in some situations. Roh et al. (2017) believe that eliminating temporary write files is noteworthy as it significantly improves the SSD lifetime.

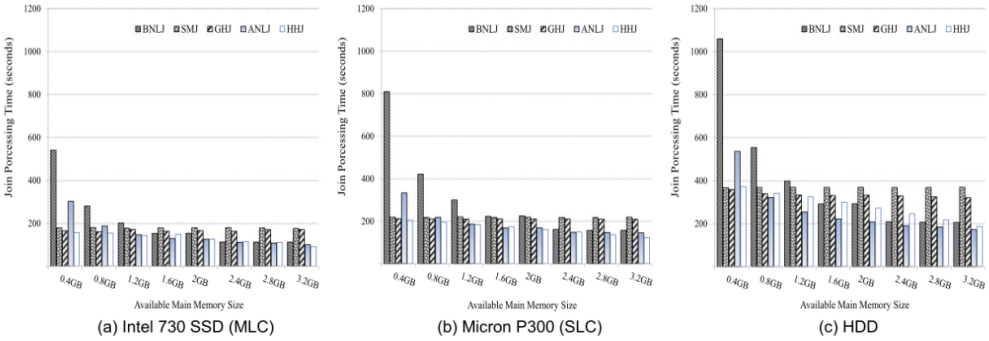

Figure 5.1 Comparison of Processing Times of Major Join Algorithms

Memory	BNLJ read	SMJ read	GHJ read	ANLJ read	HHJ read	BNLJ write	SMJ write	GHJ write	ANLJ write	HHJ write
0.4GB	23180849	4687673	4687671	10327451	4518848	0	2363088	2363086	0	2194263
0.8GB	11804705	4687662	4687658	6041841	4330901	0	2363077	2363073	0	2006316
1.2GB	8012657	4687658	4687655	4734654	4137902	0	2363073	2363070	0	1813317
1.6GB	6116633	4687656	4687654	4024988	3932108	0	2363071	2363069	0	1607523
2GB	6116633	4687654	4687653	3610268	3722730	0	2363069	2363068	0	1398145
2.4GB	4220609	4687653	4687652	3439755	3506278	0	2363068	2363067	0	1181693
2.8GB	4220609	4687652	4687652	3302539	3273471	0	2363067	2363067	0	948886
3.2GB	4220609	4687652	4687651	3048058	3036494	0	2363067	2363066	0	711909

Table 5.1 Comparison of Page I/O Counts of Join Algorithms

6. Current Trends and Future Directions

6.1. Transactions and Transactional SSDs

Transactions play a big role in ensuring the integrity of the data across crashes and reboots. However, a big obstacle for transactional APIs on HDDs is the poor read performance due to data fragmentation and extra complexity at the disk controller layer to implement these transaction protocols (*Transactional Flash*, n.d.). Thus, there has been numerous research towards creating transactional SSDs to support the transactional protocols leveraging random read accesses, such as TxSSD (Son et al., 2020) or TxFlash.

6.2. Hybrid Structure and Non-Volatile Memory (NVM)

As mentioned above, using SSD as Caching and Tiering are two important methods and have been widely applied in DBMS. Some new hybrid structures that combine caching and tiering are recently proposed to

take advantage of these two methods. For example, there has been a recent surge of interest in NVM, a type of computer memory that can hold saved data even if the power is turned off. Thus, there has been a significant amount of effort and research that has gone towards integrating NVM in SSDs. One such example is using an NVM-based cache (Tan et al., 2020) within the SSDs to further support embedded transaction protocols within the SSDs to reduce transaction over and provide fast recovery.

6.3. Architecture of SSD

Although there is already a significant number of new SSD architecture solutions available to improve database performance, the cost of SSD is still an obstacle in the current research direction and practical application. In order to solve this obstacle, we need to seek material breakthroughs in the future to reduce the cost of SSD.

7. Conclusion

From the various literature above, we can see that integrating DBMS with SSDs can improve the overall DBMS performance further, both in terms of reliability and reducing the overall DBMS expense, due to their exceptional read performance and internal parallelism leading to shorter query times. However, SSD cannot be widely used in DBMS due to its disadvantages, such as higher cost and poorer write performance. However, this situation can be improved by improving SSD or optimising DBMS.

With the above considerations, it is possible to reduce the read/write latency by optimising algorithms, such as with external sorting and joining algorithms for query processing and optimisation, indexing methods such as B+ trees, and improving the structure of caching/tiering methods. Researchers have been actively working on balancing the enhanced bandwidths and the increased latency of each I/O operation. None of these would be practical without both channel-level and package-level parallelism of SSDs. Future research will mainly focus on solving existing problems with algorithms and allowing better DBMS protocol support within the SSDs.

8. References

Bae, D., Jo, I., Choi, Y., Hwang, J., Cho, S., Lee, D., & Jeong, J. (2018). 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. 2018 ACM/IEEE 45Th Annual International Symposium On Computer Architecture (ISCA). https://doi.org/10.1109/isca.2018.00043

Dai, N., Chai, Y., Liang, Y., & Wang, C. (2015). ETD-Cache. *Proceedings of the 12th ACM International Conference on Computing Frontiers*. https://doi.org/10.1145/2742854.2742881

Fevgas, A., Akritidis, L., Bozanis, P., & Manolopoulos, Y. (2019). Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal, 29(1), 273–311*.

https://doi.org/10.1007/s00778-019-00559-8

Herodotou, H., & Kakoulli, E. (2019). Automating distributed tiered storage management in cluster computing. *Proceedings of the VLDB Endowment, 13(1), 43–56.* https://doi.org/10.14778/3357377.3357381

Hoseinzadeh, M. (2019). A Survey on Tiering and Caching in High-Performance Storage Systems. *ArXiv:1904.11560 [Cs]*. https://arxiv.org/abs/1904.11560

Huang, S., Wei, Q., Feng, D., Chen, J., & Chen, C. (2016). Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Transactions on Storage, 12(2), 1–24.* https://doi.org/10.1145/2737832

Jiang, Z., Wu, Y., Zhang, Y., Li, C., & Xing, C. (2014, September 1). *AB-Tree: A Write-Optimized Adaptive Index Structure on Solid State Disk.* IEEE Xplore. https://doi.org/10.1109/WISA.2014.42

Jørgensen, M. V., Rasmussen, R. B., Šaltenis, S., & Schjønning, C. (2011). FB-tree. *Proceedings of the 15th Symposium on International Database Engineering & Applications - IDEAS '11.* https://doi.org/10.1145/2076623.2076629

Kim, S., Lee, G., Woo, J., & Jeong, J. (2021). Zero-Copying I/O Stack for Low-Latency SSDs. IEEE Computer Architecture Letters, 20(1), 50-53. https://doi.org/10.1109/lca.2021.3064876

Laga, A., Boukhobza, J., Singhoff, F., & Koskas, M. (2017). MONTRES : Merge ON-the-Run External Sorting Algorithm for Large Data Volumes on SSD Based Storage Systems. *IEEE Transactions on Computers, 66(10), 1689–1702.* https://doi.org/10.1109/TC.2017.2706678

Lee, D., Min, C., & Eom, Y. I. (2015, January 1). *Effective SSD caching for high-performance home cloud server.* IEEE Xplore. https://doi.org/10.1109/ICCE.2015.7066359

Lee, J., Roh, H., & Park, S. (2016). External Mergesort for Flash-Based Solid State Drives. *IEEE Transactions on Computers, 65(5), 1518–1527.* https://doi.org/10.1109/tc.2015.2451631

Lee, Y.-S., Quero, L. C., Kim, S.-H., Kim, J.-S., & Maeng, S. (2016). ActiveSort: Efficient external sorting using active SSDs in the MapReduce framework. *Future Generation Computer Systems, 65, 76–89.* https://doi.org/10.1016/j.future.2016.03.003

Li, Y., On, S. T., Xu, J., Choi, B., & Hu, H. (2009, May 1). *DigestJoin: Exploiting Fast Random Reads for Flash-Based Joins.* IEEE Xplore. https://doi.org/10.1109/MDM.2009.26

Li, Z. (2014). *GreenDM: A Versatile Tiering Hybrid Drive for the Trade-Off Evaluation of Performance, Energy, and Endurance.* Undefined. https://www.semanticscholar.org/paper/GreenDM%3A-A-Versatile-Tiering-Hybrid-Drive-for-the-Li/f72e50c27d53a602073919c23899c214da7d8a72

Liu, Y., Ge, X., Huang, X., & Du, D. H. C. (2013, September 1). *MOLAR: A cost-efficient, high-performance hybrid storage cache.* IEEE Xplore. https://doi.org/10.1109/CLUSTER.2013.6702613

Liu, Y., Huang, J., Xie, C., & Cao, Q. (2010, July 1). *RAF: A Random Access First Cache Management to*

Improve SSD-Based Disk Cache. IEEE Xplore. https://doi.org/10.1109/NAS.2010.9

Michael. (2021). Advantages and Disadvantages of Solid-State Drives (SSDs) | Updato. Updato. Retrieved 26 July 2021, from https://updato.com/devices/advantages-and-disadvantages-of-solid-state-drives-ssds

Nam, B., Na, G., & Lee, S. (2010). A Hybrid Flash Memory SSD Scheme for Enterprise Database Applications. *2010 12Th International Asia-Pacific Web Conference.* https://doi.org/10.1109/apweb.2010.70

Niu, J., Xu, J., & Xie, L. (2018). Hybrid Storage Systems: A Survey of Architectures and Algorithms. *IEEE Access, 6, 13385–13406.* https://doi.org/10.1109/access.2018.2803302

Raghavan, A., Chandra, A., & Weissman, J. B. (2014, December). Tiera: Towards flexible multi-tiered cloud storage instances. In *Proceedings of the 15th International Middleware Conference* (pp. 1-12).

Roh, H., Park, S., Kim, S., Shin, M., & Lee, S.-W. (2011). B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proceedings of the VLDB Endowment, 5(4), 286–297.* https://doi.org/10.14778/2095686.2095688

Roh, H., Shin, M., Jung, W., & Park, S. (2017). Advanced Block Nested Loop Join for Extending SSD Lifetime. *IEEE Transactions on Knowledge and Data Engineering, 29(4), 743–756.* https://doi.org/10.1109/TKDE.2017.2651803

Salkhordeh, R., Asadi, H., & Ebrahimi, S. (2015). Operating system level data tiering using online workload characterization. *The Journal of Supercomputing, 71(4), 1534–1562.* https://doi.org/10.1007/s11227-015-1377-0

SSD vs HDD: Which Is Best for You? – Intel. Intel. Retrieved 24 July 2021, from https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/ssd-vs-hdd.html

Son, Y., Yeom, H. Y., & Han, H. (2020). An Empirical Performance Evaluation of Transactional Solid-State Drives. *IEEE Access, 8, 3848–3862.* https://doi.org/10.1109/ACCESS.2019.2960838

Tai, J., Sheng, B., Yao, Y., & Mi, N. (2015). SLA-aware data migration in a shared hybrid storage cluster. *Cluster Computing, 18(4), 1581–1593.* https://doi.org/10.1007/s10586-015-0461-9

Tan, Y., Tan, H., Zhu, P., Lu, Y., & He, Z. (2020). Embedded Transaction Support inside SSD with Small-Capacity Non-volatile Disk Cache. *IEEE Transactions on Knowledge and Data Engineering, 1–1.* https://doi.org/10.1109/TKDE.2020.3004518

Transactional Flash. (n.d.). Www.usenix.org. Retrieved July 28, 2021, from https://www.usenix.org/legacy/event/osdi08/tech/full_papers/prabhakaran/prabhakaran_html/

Zhao, D., Qiao, K., & Raicu, I. (2016). Towards cost-effective and high-performance caching middleware for distributed systems. *Int. J. Big Data Intell.* https://doi.org/10.1504/IJBDI.2016.077358