Effective Use of SSDs in Database Systems

by

Pedram Ghodsnia

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2018

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Ashvin Goel Associate Professor Department of Electrical and Computer Engineering Cross-appointed to the Department of Computer Science University of Toronto
Supervisor:	Kenneth Salem Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Tim Brecht Associate Professor David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Ihab Ilyas Professor David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Lukasz Golab Associate Professor Department of Management Sciences Cross-appointed to the School of Computer Science University of Waterloo

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

With the advent of solid state drives (SSDs), the storage industry has experienced a revolutionary improvement in I/O performance. Compared to traditional hard disk drives (HDDs), SSDs benefit from shorter I/O latency, better power efficiency, and cheaper random I/Os. Because of these superior properties, SSDs are gradually replacing HDDs. For decades, database management systems have been designed, architected, and optimized based on the performance characteristics of HDDs. In order to utilize the superior performance of SSDs, new methods should be developed, some database components should be redesigned, and architectural decisions should be revisited.

In this thesis, novel methods are proposed to exploit the new capabilities of modern SSDs to improve the performance of database systems. The first is a new method for using SSDs as a fully persistent second level memory buffer pool. This method uses SSDs as a supplementary storage device to improve transactional throughput and to reduce the checkpoint and recovery times. A prototype of the proposed method is compared with its closest existing competitor. The second considers the impact of the parallel I/O capability of modern SSDs on the database query optimizer. It is shown that a query optimizer that is unaware of the parallel I/O capability of SSDs can make significantly sub-optimal decisions. In addition, a practical method for making the query optimizer parallel-I/O-aware is introduced and evaluated empirically. The third technique is an SSD-friendly external merge sort. This sorting technique has better performance than other common external sorting techniques. It also improves the SSD's lifespan by reducing the number of write operations required during sorting.

Acknowledgements

During my work on this thesis, I had the distinct honor of being supervised by Prof. Kenneth Salem whose vast knowledge and insight in the field together with his nice personality made this journey a wonderful experience. Without his invaluable help, guidance, and support I would not be able to complete this thesis. I could not have imagined having a better supervisor and mentor for my Ph.D study.

I would also like to thank my committee members, Tim Brecht, Ashvin Goel, Lukasz Golab, and Ihab Ilyas for taking the time to read and critique my thesis and for their constructive comments.

I would also like to thank my mentors and colleagues at SAP Waterloo, especially, Anil Goel, Ivan Bowman, Reza Sherkat, Anisoara Nica, John Smirnios, Peter Bumbulis, and Mohammed Abouzour, for giving me the opportunity to work in their wonderful team, and for all the help and support they gave me, and for all the valuable things I learned from them.

I would also like to thank the many friends I have met during my studies. Without you this could probably be finished earlier, but surely not as enjoyable as it was.

Last, but not least, I want to thank my family from the bottom of my heart, especially my lovely wife Somayeh who without her love, support, and encouragement, and without the sacrifices she made during the course of this journey, I would not be where I am today.

Dedication

This thesis is dedicated to my amazing son, my beloved wife, and my wonderful parents.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Hard disk drives (HDDs) have been used as a persistent storage layer in database systems for many years. In spite of decades of advancements in HDD technology, due to the mechanical nature of HDDs, they can still provide high performance only for sequential access, while suffering from a much lower performance in random I/O. A decade ago, SSDs emerged as a viable storage alternative, aiming to address the limitations of HDDs.

Because of their electrical nature, SSDs benefit from much better random I/O performance, as well as better power consumption. Due to these impressive characteristics, the research community has shown a lot of interest in the application of SSDs in I/O intensive use cases. Database systems have been designed and developed over more than three decades of research based on the I/O characteristics of HDDs. In order to benefit from the outstanding I/O characteristics of SSDs, there are many architectural aspects in database systems that need to be revisited.

1.1 Background

Flash memory is a type of memory that stores data in NAND gate arrays called *flash cells*. A variety of flash cells have been introduced by storage manufacturers in recent years. *Single-level cells* (SLC) can store only one bit by identifying only the presence or absence of the electrical current. *Multi-level cells* (MLC) can typically store two bits, by detecting four voltage levels. Although MLC devices are denser than SLC devices, they are slower as it takes longer to deal with four voltage levels. In addition, MLC devices can endure fewer *program/erase cycles* (P/E). SLC devices are therefore more expensive

and they are typically used in enterprise applications with high-performance and high-endurance requirements. *Triple-level cells* increase the data density to 3 bits by storing 9 voltage levels. Although the TLC devices are slower than MLC devices and have a shorter lifespan, they form a cheaper alternative to MLC devices for the consumer market. Very recently a new class of flash cells called quadruple-level cell (QLC) has been introduced which increases the data density to 4 bits per cell. Because of the fewer predicted number of P/E cycles in QLC devices, they are expected to be used in write-once-read-many (WORM) applications in some specific enterprise use cases like large cache servers.

Solid State Drives, also known as flash drives or flash disks, consist of multiple flash memory chips in a single enclosure. SSDs are equipped with a controller that is connected to a DRAM buffer. The controller is in charge of translating system read and write I/O requests to physical read, erase and program instructions on the flash chips. The controller also performs a variety of tasks for improving the performance and improving the lifetime of the flash chips. In order to improve the throughput of the write requests, part of the internal DRAM is used for buffering the received writes. The remaining part of the DRAM is used as a temporary memory for executing the regular tasks of the controller. In some enterprise SSDs, the DRAM is backed by super-capacitors. In the event of a power failure, the super-capacitor gives the controller enough time to persist the contents of the DRAM.

Since SSDs have no mechanical moving parts, they benefit from much lower access latency, better power efficiency, and lower heat generation. In addition, they incur faster startup times and offer better shock resistance. The internal architecture of modern SSDs allows them to improve random I/O throughput by exploiting the internal I/O parallelism. In terms of price per capacity, SSDs are more expensive than HDDs, but in terms of price per IOPS (input/output operations per second), they easily outperform HDDs.

SSDs suffer from two main issues. The first one is called the *Erase-before-write* constraint: even for changing a single sector, the entire flash block to which it belongs should be first copied into the internal memory; then the flash block should be erased, and finally the memory content should be programmed back into the erased block. This process makes writes much more expensive than reads. The second issue is the limited number of P/E cycles supported by flash chips. This problem gets worse as the density of the flash cell increases.

In order to address these two problems, the controller uses a software layer called *Flash Translation Layer* (FTL) [9, 14, 22, 53, 64, 68]. This software layer is in charge of logical-to-physical address mappings. It also is responsible for wear-leveling and power-failure recovery. To improve the write performance, whenever a new write request is received, the content of the flash block to which the corresponding sector belongs is copied into the

internal memory. Then, if a free flash block is available, it is programmed and marked as the valid block in the *address translation table* of FTL. Finally, the old block is marked as garbage. The marked blocks are erased later by a periodical garbage collector task performed by the controller. In some controllers, the garbage collector even tries to combine the existing partially-empty blocks to claim as many free blocks as possible, making them ready for the incoming writes.

The larger the capacity of the SSD, the less frequently the garbage collector needs to work. That is why the random write performance of the higher capacity SSDs is typically better than that in lower capacity devices. In order to guarantee a specific write performance, some Enterprise SSDs employ a mechanism called over-provisioning. In this method, a portion of the available flash chips is used for write optimization. This portion is not usually advertised as the available capacity of the SSD. The garbage collector tries to erase at least as many blocks as needed to maintain the cleanness of the over-provisioning capacity.

The issue of limited P/E cycles is addressed by employing the wear leveling techniques: the controller spreads writes as even as possible across all flash chips and all flash blocks in the same chip by maintaining an array of counters in the FTL layer. An SSD with a larger capacity will have a better lifetime as it takes a longer time for the first flash chip to pass its P/E cycle limit. The controller will also mark the flash chips that have already reached their maximum allowed P/E cycles and will transfer their content to healthy chips to postpone the total drive failure.

In some recent TLC-based devices, in order to improve the durability and write performance while benefiting from the high data density, a layer of SLC-based flash chips are employed for write caching, translation table maintenance, and other FTL optimizations.

The additional write operations performed for wear leveling and garbage collection are referred to as *write amplification*. A probabilistic analysis of write amplification shows that it heavily affects both the write performance and the lifetime of the SSD [27, 47].

Modern SSDs can substantially benefit from *I/O parallelism*, the ability to perform multiple I/Os simultaneously. Chen et al. have studied the substantial impact of parallelism in I/O performance of SSDs [21]. A modern SSD is capable of utilizing multiple levels of parallelism: plane, channel, package, and die levels. Almost all modern SSDs support native command queuing mechanisms (NCQ), which were first introduced in the SATA II standard [24]. These capabilities allow the SSD to accept multiple concurrent I/O requests or a burst of successive I/O requests from the operating system. The received I/O requests are queued, and the host interface will reorder them to create a favorable I/O pattern for the internal parallel organization of the SSD. In other words, increasing the *I/O queue*

depth, defined as the number of outstanding I/Os in the I/O queue at any point of time, of modern SSDs will give them a chance to exploit their internal parallel organization to improve the I/O throughput. The I/O queue depth can be increased by issuing multiple I/Os at the same time. Alternatively, issuing I/O requests with a rate faster than the rate of handling I/O requests by the device can increase the I/O queue depth. Increasing the I/O queue depth of the SSD can significantly improve the random I/O throughput of the SSD.

1.2 Research Hypotheses and Thesis Organization

In terms of price per IOPS (I/O per second), SSDs are better than HDDs. However, in terms of price per capacity, SSDs are still more expensive than HDDs, and this trend is expected to continue for the foreseeable future. The issue of cost-effectiveness of SSDs adds another constraint to the adaptation of SSDs in database systems. When replacing the entire storage layer by SSDs is not a cost-effective decision, the SSD can be used as a supplementary storage layer. When storing the entire database on the SSD is a feasible choice, the effective use of SSDs depends on proper architectural adjustments in database internals.

In this thesis, first, we target the use cases in which due to size, budgeting, or technical constraints, storing the entire database on the SSD is not feasible. In this case, our first research hypothesis is as follows:

Hypothesis 1: *SSDs can be used as a fully persistent second level buffer pool to improve the transactional throughput, checkpoint time, and recovery time and to avoid a long ramp-up time after a crash recovery in traditional database systems.*

To test our hypothesis, in Chapter 2, a novel approach for using SSDs as a persistent second level memory buffer pool is designed, implemented, and evaluated experimentally.

By mass adoption of SSDs, the price per capacity of the SSD is declining day by day. Although it is not expected to see that SSDs will become cheaper than HDDs anytime soon, the price gap between the two is gradually getting narrower. This trend will slowly improve the cost-effectiveness of storing the entire database on SSDs, especially in applications in which the size of the database is not that large. Thus, we devote the remaining part of the thesis to scenarios in which storing the entire database on the SSD is feasible. The research hypotheses of the thesis, in this case, are as follows:

Hypothesis 2: *When the entire database is stored on the SSD, we need to make the*

query optimizer SSD-aware; failing to do so will result in sub-optimal query optimizer decisions.

To test hypothesis 2, in Chapter 3, the impact of parallel I/O on optimal access path selection in the query optimizer is identified. It is shown that the optimizer's error can be significant when the optimizer knows nothing about the parallel I/O capability of the SSD. Moreover, a practical method for capturing and utilizing the I/O capability of SSDs by the query optimizer is introduced, implemented and evaluated.

Hypothesis 3: *When the entire database is stored on the SSD, we need to redesign some of the major database operators so that they can make better use of the capabilities of SSDs; failing to do so will result in performance degradation of the operator.*

To test hypothesis 3, in Chapter 4, an SSD-friendly external merge sort method is proposed, and it is experimentally shown that it outperforms the traditional external sort in a range of configurations while improving the lifetime of the SSD by reducing the number of temporary writes.

Chapter 2

SSDs as a Persistent Second Level Cache

2.1 Overview

Although the I/O performance of the SSD is superior to that of the HDD, its price per capacity is still higher. The market trends show that this price gap is expected to exist for the foreseeable future. Therefore, in large-scale databases, utilizing the SSD as an HDD supplement might be more cost-effective than replacing the HDD with the SSD. One way of using the SSD as an HDD supplement is to store the most-frequently-accessed data in the SSD and to leave the cold data in the HDD [18, 56, 74]. In this approach, every data item is stored either in the HDD or the SDD and not in both. Although storing the hot data objects on the SSD sounds appealing, the proposed approaches for realizing this idea suffer from inaccurate static decisions, costly back and forth periodical data transfers between the SSD and the HDD, and the coarse granularity of the transferred objects. In order to address these issues, utilizing the SSD as a second level cache has become a very popular alternative [13, 19, 30, 48, 49, 56, 65, 66]. In this method, the SSD is used as a caching layer between the hard disk and the memory. Whenever a memory cache-miss (first level cache-miss) takes place, the SSD cache is probed to see if the page can be found there. Due to the lower price of the SSD compared to the RAM, the size of the second level cache is supposed to be much larger than that of the memory buffer pool. Therefore, the SSD can cache a much larger number of pages. This larger cache capacity can reduce the number of expensive hard disk I/O requests significantly, resulting in a considerable jump in the overall system performance.

So far, several approaches for using the SSD as a second level cache have been proposed. Prior studies either treat the SSD as a volatile memory, exploiting only the higher I/O performance of the SSD compared to the HDD, or do not exploit the persistence of the SSD effectively. In this chapter, a novel method for exploiting the SSD as a fully persistent second level cache is introduced. The proposed method is called PC, short for the persistent cache. PC addresses the drawbacks of the prior methods. It has been patented [38].

A prototype of PC, as well as its closest prior competitor, was implemented in SAP SQL Anywhere. In the prototype, the cache management, I/O management, checkpoint mechanism, and recovery process of the database engine are modified. The experimental results show significant improvements in transactional throughput[1] and recovery time of the database.

[1] Transactional throughput refers to the number of transactions per minute that can be processed by the database system.

2.2 Summary Of Contributions

The main research contributions presented in this chapter can be summarized as follows:

1. A novel approach for using the SSD as a fully persistent second level cache in traditional RDBMSs.
2. An implementation of the proposed method, as well as its closest competitor, by modifying the I/O manager, cache manager, checkpointing mechanism, and recovery mechanism of a commercial RDBMS.
3. A comparison of the proposed approach with its competitor, under transactional workloads, showing that the proposed approach benefits from significantly higher transactional throughputs and lower recovery times.

2.3 Background

The SSD can be utilized either as a write-through or as a write-back second level cache. In a write-through cache, the contents of the SSD cache are used only to satisfy read requests. In this approach, the content of the SSD should be kept consistent with that of the HDD at all times. Therefore, whenever a dirty page (modified page) gets evicted from the memory buffer pool (first level cache) and a version of that page resides in the SSD cache (second level cache), its modifications must be reflected both on the SSD and the HDD. In contrast, in a write-back caching mechanism, there is no need to guarantee the consistency of the pages between the SSD and the HDD. Hence, pages residing in the SSD are free to be fresher than their corresponding copy in the HDD.

In write-intensive transactional workloads such as TPC-C [81], which are the focus of our study, a write-back second level cache performs much better than a write-through cache. This is because, in write-intensive workloads, the dirty pages in the second level cache tend to be re-referenced and re-dirtied several times before they get evicted. In a write-back second level cache, such pages are written to and read back from the SSD multiple times before they are finally written back to the disk. In a write-through second level cache, however, such pages are written to the HDD every time they are evicted from the memory buffer pool. This increases the I/O load of the HDD significantly, resulting in dramatic performance degradations.

Several approaches for using the SSD as a second level cache have been proposed. Koltsidas et al. studied a variety of data flow schemes (inclusive, exclusive and lazy) in

a multi-level caching approach, both theoretically and experimentally [57]. However, all caching schemes discussed in their study are based on the assumption that the SSD is an extension of RAM. Moreover, the recovery implications of the proposed schemes and the role of the checkpoint are not considered in that analysis. The process of copying dirty pages from the memory buffer pool to the disk is known as a checkpoint. Checkpoints have a considerable impact on database performance, especially in write-intensive workloads. The main goal of the checkpoint is to reduce the recovery time after a database crash.

Canim et al. exploited the SSD as a write-through cache [19]. They introduced a temperature-aware admission and replacement policy called TAC, short for temperature-aware caching. This policy tends to replace the pages that come from hot regions of the HDD with the ones belonging to the colder regions. Three different caching schemes, CW (clean-write), DW (dual-write), and LC (lazy-cleaning) are proposed by Do et al. in [30]. The performance of these approaches over different workloads has been studied using a prototype implementation over Microsoft SQL Server 2008. CW and DW, similar to TAC, are write-through while LC is a write-back caching scheme. In CW, which stands for Clean-Write, only clean pages are stored in the SSD. Whenever a dirty page is evicted from the RAM, it is written only into the disk. In DW, which stands for Dual-Write, the dirty pages evicted from RAM are written to both the SSD and the HDD. In both CW and DW, the contents of the SSD and HDD are always kept consistent. In LC, which stands for Lazy-Cleaning, the dirty pages written to the SSD cache are flushed into the HDD whenever their count passes a specific threshold. Therefore, unlike CW, DW and TAC, in LC the contents of the SSD cache can be fresher than the contents of the HDD. It is shown that in read-intensive (analytical) workloads such as TPC-H and TPC-E, the caching mechanisms DW, TAC and LC perform almost as well as each other. However, in the TPC-C benchmark, which is a write-intensive transactional workload, LC shows up to 6.8X and 5X speedup over TAC and DW respectively [30]. CW performs worse than all the other methods both in read- and write-intensive workloads.

As mentioned before, in write-intensive workloads, a write-back caching method will perform much better than a write-through caching method. However, the implementation of a write-through cache is much easier. This is because in a write-back cache the content of the SSD can be fresher than the content of the HDD. Consequently, the checkpoint and recovery logic must be modified accordingly.

2.3.1 Database Checkpointing

At a database checkpoint, all dirty pages residing in RAM are flushed into the stable storage, allowing a larger part of the transaction log file to be ignored at the recovery time. The

Checkpoint interval, which is defined as the time interval between two consecutive checkpoints, is an important tuning parameter in modern DBMSs, especially in write-intensive workloads. Choosing a longer checkpoint interval roughly corresponds to having a longer recovery time. On the other hand, since the checkpoint is a costly I/O-bound operation, choosing a very short checkpoint interval might affect the performance of the database negatively. Choosing a proper checkpoint interval in write-intensive workloads such as OLTP is more vital than in read-intensive workloads such as OLAP. In write-intensive workloads, the memory pages tend to get dirty quicker and more often. Consequently, if a checkpoint does not happen for a long time, the transaction log file grows quickly and the recovery time becomes prohibitively long. At the same time, in write-intensive workloads, the negative impact of a checkpoint on performance is potentially much higher than in read-intensive workloads.

To show the negative impact of each checkpoint in write-intensive workloads, we performed an experiment. Figure 2.1(a) illustrates the tpmC-time diagram of a TPC-C workload with 1000 warehouses when the entire database is stored in a commodity 7,200 RPM hard disk. The y-axis represents the number of new order transactions per minute in the TPC-C workload. The experiment was performed using SAP Sybase SQL Anywhere. A memory buffer pool larger than the database size is used in both experiments. Thus, after the buffer pool warm-up period, the number of writes into the disk as a result of the eviction from the memory buffer pool will become almost zero. In this way, the impact of the checkpoint on the degradation of the transactional throughput can be observed in isolation. After the memory buffer pool warms up, all the regular I/Os are entirely routed to the memory buffer pool. The checkpoint interval in both experiments is 60 minutes.

As depicted in Figure 2.1(a), each checkpoint results in a significant degradation in the transactional throughput. This is because of a burst of random I/O pressure on the disk during the checkpoint process. Figure 2.1(b) shows the same experiment when the entire database is stored on a consumer-level SSD. The extra I/O pressure can be handled perfectly well by the SSD while it presents difficulty for the HDD. We stopped each experiment after processing 80 million new order transactions. On the SSD all transactions are processed in almost half the time they are processed on the HDD.

By using the SSD as a second level write-through cache, there is no need to make any modification in the checkpoint and recovery modules of the database because, in a write-through cache, the disk content is always consistent with the SSD content. In contrast, in a write-back caching mechanism, both of these modules have to be modified to guarantee the recoverability of the database.

When the SSD is used as a volatile write-back second level cache, such as what is done

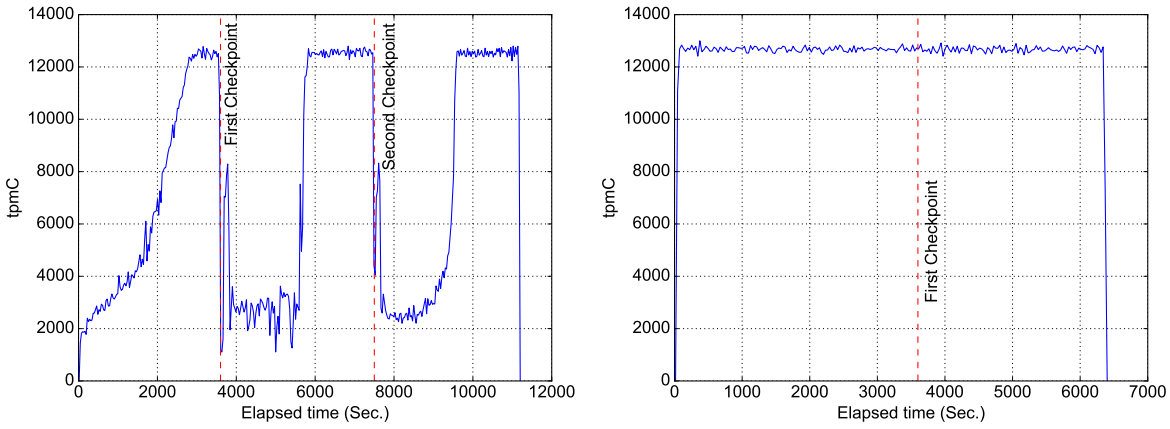


Figure 2.1: Effect of the checkpoint on TPC-C throughput. In both experiments, the entire DB fits into the memory buffer pool. In (a) and (b) the entire DB is stored on the HDD and the SSD, respectively. Y-axis represents the number of the new order transactions per minute in the TPC-C benchmark (tpmC).

in LC, the HDD will be the target of extra I/O pressure, when a checkpoint occurs. In the LC method, before flushing the dirty pages from the memory into the disk, the existing dirty pages in the SSD need to be flushed into the HDD. Since the size of the SSD is supposed to be much larger than the size of the RAM, the number of dirty pages in the SSD tends to be much larger than the number of dirty pages in the RAM. Besides, since no direct memory access between the disk and the SSD is possible, all the dirty pages residing in the SSD must be copied into memory before being flushed into the disk. Consequently, a checkpoint becomes extremely expensive.

The proposed methods in this chapter, avoids these problems by using the SSD as a non-volatile write-back second level cache. Thus, it does not need to flush dirty pages from the SSD to the HDD during checkpoints. Exploiting SSDs' non-volatility also shortens recovery times and reduces the ramp-up time of the cache after a crash.

2.4 PC: SSDs as a Persistent Cache

In this chapter we introduce a new method for using SSDs as a fully persistent cache. Our proposed method is called PC, short for persistent cache. First, the page access sequence

and caching policies enforced by PC are presented. Then, the checkpointing mechanism of PC is described. Finally, the recovery scheme of PC is explained, and its recoverability is discussed.

2.4.1 Page Access Sequence

Our design uses the same page access sequence which is used in the LC method. When the cache manager receives a page request, it first looks for the page in the first level cache. If it cannot find the page there, then the second level cache is examined. If the page is not found there, it is fetched from the disk storage. In PC, at any point of time, the memory cache content (the first level cache) is always either fresher than or as fresh as the SSD cache content, and the SSD cache content is always either fresher than or as fresh as the HDD content.

2.4.2 SSD Cache Replacement Policy

The replacement policy of the second level cache in PC is a variation of the Clock replacement policy [90]. We chose this policy because it is a very low-overhead yet effective replacement policy. It is confirmed by Canim et al. [19] that in TPC-C workload when the SSD cache is at least three times larger than the memory buffer pool, the cache hit rate of the Clock replacement policy is almost as good as that of more sophisticated policies such as ARC [69] and LRU [73].

2.4.3 SSD Cache Admission Time

In methods in which the SSD is used as a second level cache, there are two alternatives for the time of admission into the second level cache. The first alternative is to decide about the admission of the page into the second level cache at the same time that a page is admitted to the first level cache. In this case, whenever a page is admitted to the first level cache, it will be considered for admission to the second level cache as well. This decision might result in degradation of the system performance because of the potential for latch contention between the thread which is writing the page into the second level cache and the thread which is trying to modify the content of the page after its admission to the first level cache. It is worth mentioning that copying a page from the disk storage to the second level cache must be done through the first level cache because no direct memory access (DMA) between the SSD and the disk storage is possible. Therefore, copying a page from

the disk storage to the second level cache can be started only after finishing its admission into the first level cache.

The second alternative is to decide about the admission of the page into the second level cache at the time it is evicted from the first level cache. We decided to use the second alternative as it does not suffer from the latch contention problem mentioned above. This decision also reduces the impact of the cache inclusion problem [92]. The cache inclusion problem refers to the issue of having the same copy of a page in different levels of a multi-level caching scheme. Cache inclusion can result in wasting the cache capacity of the middle levels.

2.4.4 Eviction From the SSD

Whenever a new page is admitted into the second level cache and the second level cache is full, a victim needs to be selected for eviction according to the replacement policy. In PC, if the selected victim is fresher than its disk-resident version, it is flushed into the disk before the eviction. If it is as fresh as its disk-resident version, there is no need to flush it into the disk.

2.4.5 Qualified Pages for Admission

Whenever a page is evicted from the first level cache, it is considered for admission into the second level cache. In our design, in order to avoid polluting the second level cache with useless data, the candidate pages for admission should be qualified first. There are two possible qualification criteria in our design: (1) whether the victim has been originally read randomly or sequentially, and (2) whether the victim is dirty or clean. If a page is admitted to memory as a result of a high-level operator with a sequential I/O pattern, e.g. a full table scan, a flag called isRandom will be set to false in the page's metadata. In contrast, if the admitted page to memory is a result of a high-level operator with a random I/O pattern, e.g. an index scan, the isRandom flag for that page is set to true. Using this flag, at the time of admission to the second level cache, it can be determined whether a page is accessed randomly or sequentially. Since sequential reads can be done efficiently from the disk, we avoid caching the sequential pages in the second level cache. We observed that admitting the sequential pages in the second level cache not only results in wasting the capacity of the cache but also degrades the overall I/O performance. The sequential reads from the disk go to a temporary memory buffer in large blocks consisting of multiple consecutive pages. The read pages are then admitted from the temporary

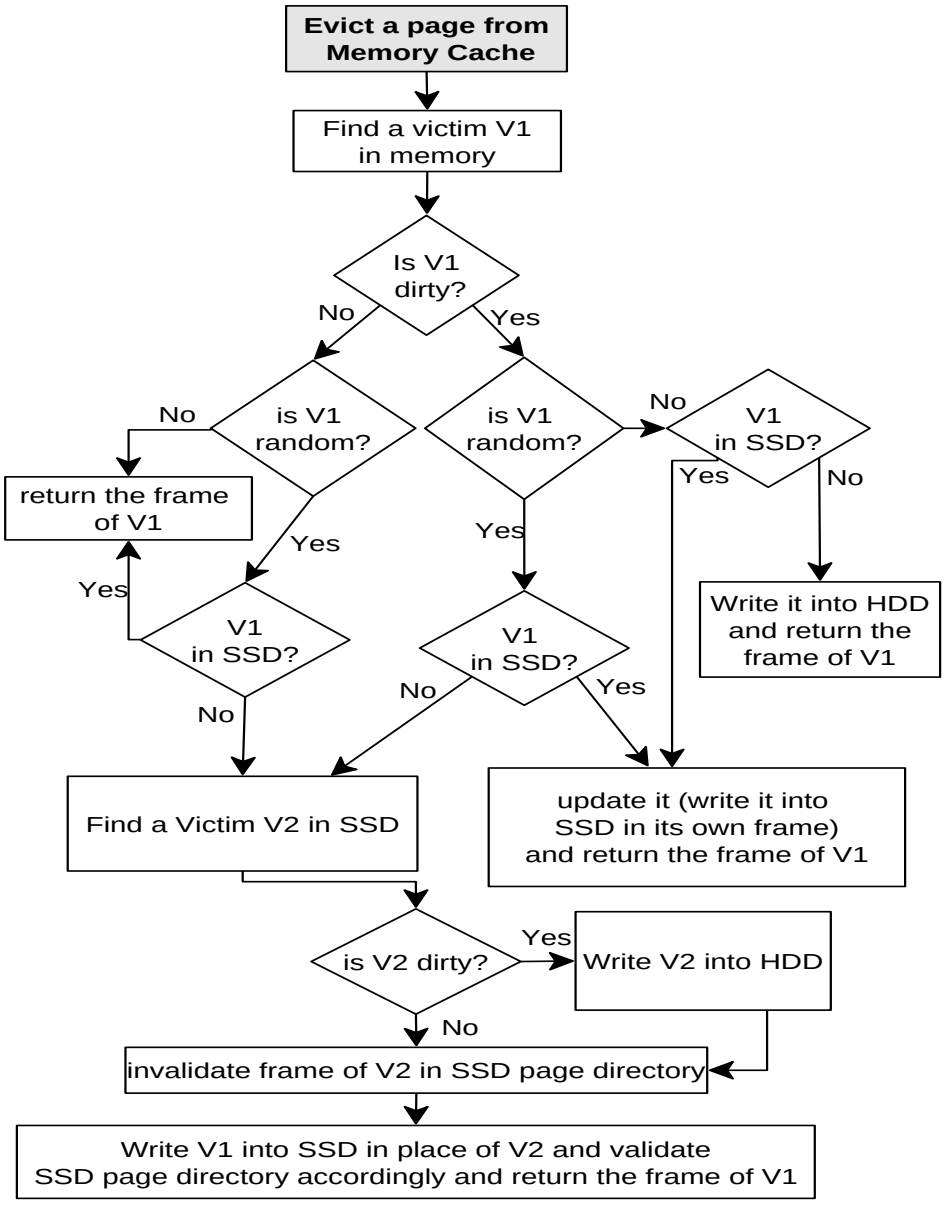


Figure 2.2: Page flow in PC method

memory buffer to the memory buffer pool one by one. After reading each block into the temporary memory buffer, before admitting each page within that block into the memory buffer pool, the content of the second level cache is examined to see if a fresher version of that page can be found there. If that is the case, the page should be fetched from the second level cache. This extra step places extra read overhead on the SSD. By disqualifying the sequential pages from admission to the second level cache, this extra overhead can be reduced. Therefore, in our design, only randomly-accessed pages will normally be qualified for admission into the second level cache. Sequential pages will be qualified only if they are dirty and an older version of them already exists in the second level cache. If an older version of a dirty sequential page is not in the second level cache then it will be flushed directly to the disk. The details of the page flow of our proposed approach are illustrated in Figure 2.2.

2.4.6 Checkpoint

Similar to LC, in PC, the SSD is used as a write-back second level cache. However, unlike LC, in PC, the SSD is treated as a persistent storage. By relying on persistence of the SSD, there is no need to copy any dirty page from the SSD to the HDD when performing a checkpoint. At the recovery time, the latest version of each page can be retrieved from the SSD. If a page is not found in the SSD, it means that its latest version is available in the HDD. This decision results in a significant improvement in the checkpoint time.

Since the checkpoint is performed only over the dirty pages residing in the RAM, the need for having an SSD cache cleaner process would be eliminated as well. Consequently, there would not be any dramatic drop in the transactional throughput as a result of starting the cleaner process. Moreover, there is no need to tune a cleaning threshold parameter anymore.

Since the SSD cache is persistent, we must decide about the target of the dirty pages flushed from the memory at the checkpoint time. Here, there are two possible choices. The first option is to write all the first level dirty pages into the second level cache no matter whether they are already there or not. An alternative is to check each dirty page individually to determine whether it is already in the second level cache. If so, then the dirty page would be flushed in-place in the second level cache. If a dirty page is not in the second level cache, then it would be written into the HDD.

At first, the former option may seem to have better checkpoint performance. However, in practice, for two reasons the latter alternative is preferable. First, the former alternative results in more admissions and eviction into and from the second level cache

during the checkpoint. In contrast, in the second alternative, all the second level cache I/Os are in-place writes. More admissions and evictions will increase the contention over the replacement queue. Note that during the checkpoint, the regular transactions are not frozen, and the admissions and evictions caused by the regular transactions will compete with those caused by the checkpointing. Second, writing all first level cache dirty pages into the second level cache will worsen the cache inclusion problem.

To achieve the benefits of both alternatives mentioned above, we decided to use a hybrid approach. In the hybrid approach, the first alternative will be applied until the second level cache gets full. Afterwards, the second alternative will be employed. An additional benefit of using the hybrid approach is that the second level cache gets warm faster. This improves the ramp-up time of the second level cache. Moreover, by using this approach, all checkpoints happening during the ramp-up time would be extremely fast as all random writes in these checkpoints would be directed to the SSD.

2.4.7 Recovery

By employing a persistent second level cache, the recovery process can be accelerated significantly. Recovery is typically a read/write intensive process. Pages will be randomly read from or written into the stable storage during the recovery. In a database with no persistent second level cache, all those reads and writes should be performed on the HDD. With a persistent second level cache, however, most of those requests can be directed to the second level cache. In other words, if a page resides in the second level cache, the I/O request would be sent to the second level cache. Otherwise, it would be sent to the disk storage. This results in a considerable performance improvement.

To benefit from the persistence of the second level cache during the recovery process, before starting the recovery, the location of a data page (i.e. whether it is on the second level cache or in the disk storage) must be known at the time the crash occurred. Thus, the page directory of the second level cache must be maintained in a reliable way.

The page directory can be maintained either in the memory or the stable storage. Maintaining the page directory in memory results in better performance. However, in this case, if the database crashes, the page directory will be lost. Upon recovery, the page directory must be reconstructed. One solution for reconstructing the page directory at the beginning of the recovery process is to scan the entire second level cache and reconstruct the page directory using the pages' metadata. However, scanning the entire second level cache is very costly. This operation is also subject to the single-page failure problem [43],

aka torn page problem or partial write problem. The torn page refers to a page that, due to a media failure or power outage, has been not entirely written into the storage media.

In our design, the page directory is stored on the SSD. Since the SSD is persistent, reconstructing the page directory by scanning the entire SSD is not necessary.

Storing the page directory on the SSD seems to be an expensive design decision. The page directory is supposed to track the changes made to the content of the second level cache. Whenever the cache content is changed the page directory should be updated, resulting in extra writes. By performing a simple trick, we have significantly reduced the number of these extra writes.

The trick is that we maintain a memory-resident version of the page directory as well as a lightweight SSD-resident version of it. The memory-resident version contains all of the information necessary for caching and for enforcing the replacement policy. The SSD-resident version contains only the information which is needed for rebuilding the memory-resident version at recovery time. In the light-weight version, we do not keep pointers, unnecessary flags, and latches. Eliminating pointers and unnecessary flags will make it impossible to rebuild the same exact cache state that exists before a crash. That is because the pointers and flags which are used for enforcing the Clock replacement policy are lost after a crash. Consequently, immediately after finishing the recovery process, the utilization of the cache will be slightly lower than before the crash, and it takes a while to reach the same cache utilization as before the crash. Compared to LC, which has an empty cahce after a crash recovery, a minor degradation in cache utilization is acceptable. This trade-off is done to reduce the number of required modifications on page directory. By using this technique, the page directory needs to be modified only on 3 occasions; at admission time, at eviction time, and whenever a clean page residing in the second level cache gets updated for the first time.

As the second level cache gets warmer the first and the second occasions become less likely to happen. When the working set of the database fits in the second level cache, the number of admissions and evictions becomes very low. The third occasion happens only if a clean page from the memory gets admitted into the second level cache and later it gets re-admitted from the second level cache to the memory and then it gets dirtied there, and then it gets evicted from the memory. This page should be updated in-place in the second level cache before eviction from the memory. In this case, the page directory needs to be updated to set a flag (fresher_than_disk flag[2]) indicating that the content of that page is fresher than the content of its HDD-resident version. The frequency of the occurrence of this situation in practice is very low. Suppose a page residing in the second level cache,

[2]This flag is used to determine if a page should be flushed to the HDD when it is evicted from the SSD.

which is already fresher than its HDD-resident version, becomes admitted into the memory, gets dirty in there, and then becomes evicted from the memory. In this case, there is no need to update the page directory because the fresher_than_disk flag is already set for that page. In this case, an in-place update of the page in the SSD suffices.

In a write-intensive workload in which the pages are modified many times if the working set of the database fits into the second level cache, then most requests to the second level cache will be the reads and in-place updates. None of these requests need a change in the page directory. The fact that maintaining the page directory in the SSD does not have a considerable negative impact on performance has been confirmed by Bhattacharjee et al. as well [13].

2.4.8 Crash Consistency

Because the page directory is stored in the second level cache, if an eviction from the first level cache results in an eviction from the second level cache, as described above, one method of proceeding is as follows. At step one, a victim must be found in the second level cache. This victim is copied into the disk to make a free frame for a victim page from the first level cache at step two. At step three, the victim page from the first level cache is copied into the second level cache, and the page directory is updated accordingly at step four. However, if a crash occurs during the steps two or three, a torn page can potentially exist on the HDD or the SSD. A torn page can result in data loss or inconsistency in the database. Torn pages can happen while writing to both SSDs and HDDs but they are less likely in SSDs. Since a torn page may put the page directory in an inconsistent state, we need to define a mechanism to avoid this inconsistency.

The page directory will be in an inconsistent state in two cases. First, when the page directory falsely reports that a page is stored in the SSD while the SSD-resident version of the page is torn. Second, when the page directory reports that a page is located in the HDD while the latest version of the page is located in the SSD and its HDD-resident version is torn.

To avoid the possible inconsistencies when a torn page happens at the crash time, if an eviction from the first level cache results in an eviction from the second level cache, after the victim page from the second level cache is copied into the disk, the page directory is invalidated for that victim pages page frame. The victim page from the first level cache is copied into the now-free frame on the second level cache, and then the page directory is updated and validated for the corresponding page frame. To employ this idea a flag called is_valid is used for each page frame in the page directory.

If a crash occurs while the victim page from the second level cache is being copied into the disk storage, a torn page on the disk storage might be possible. However, in this case, the torn page does not result in any inconsistency, because the latest version of the page is located in the second level cache and it is still valid.

Similarly, if a crash occurs after the victim page from the second level cache is successfully copied into the disk, but before the page frame can be invalidated, then no inconsistency will exist because the page directory will truly report that the latest version of the page is located in the SSD.

If a crash occurs while the page is invalidated in the page directory, then two possible situations exist. Either the invalidation will be done successfully, or it will fail. If the page is invalidated successfully, it will contain an invalid frame, and a correct copy of the page for that frame is already available on the disk. If the invalidation fails, then the page directory will correctly report that the page is still located in the SSD. The is_valid flag is just a bit in an integer variable which is used for storing the flags. A torn bit is not possible, and the value of this bit after a crash can be either one or zero.

If a crash occurs directly after the page frame is successfully invalidated, then an invalid frame will exist in the page directory, but a correct copy of that page is already safely stored in the disk, and so there will be no inconsistency in the page directory.

If a crash occurs when the victim page from the first level cache is copied into the second level cache, the contents of that page frame might become torn. However, no inconsistency will exist in this case, because the page frame for that page is already marked as invalid, and the crash recovery process will not rely on the contents of that torn page in the second level cache.

Further, if a crash occurs after the victim page from the first level cache is successfully copied into the second level cache, and before validating the page in the page directory, then the contents of that page frame will be valid, but the page directory will report that the page frame is not valid. No inconsistency will exist in this case either. The recovery process will assume that this page is located in the HDD and will re-write the HDD-resident version of the page with a clean copy which is stored in checkpoint log [7] and then the content of the page will be updated using the existing redo logs in the transactional log.

Finally, a crash may occur after successful validation of the page in the page directory. In this case, the final step of the eviction is completed, and the page directory will correctly report that a correct and up-to-date version of the page is located in the SSD. Hence, no inconsistency will exist.

2.4.9 Soundness of Crash Recovery

The crash recovery is sound if all and only committed transactions are recovered by the recovery process. PC adds a preparation phase to the recovery process of the database system. In the preparation phase, first, the memory-resident page directory is built using the SSD-resident page directory. Then, the required in-memory data structures, i.e. the FIFO queue and the hash table are built in memory over the in-memory page directory. The crash consistency mechanism described in Section 2.4.8 guarantees that at this point of time the persistent layer of the database is consistent with its persistent layer at the crash time. This means that the page directory reports the location of the latest version of every page correctly. Thus, since the consistency of the storage layer is guaranteed, assuming the recovery process of SAP SQL Anywhere is sound, we can conclude that it will remain sound after adding the preparation phase as well.

2.5 System Components

A prototype of PC has been implemented in SAP SQL Anywhere. The main components of the system are as follows:

1. **SSD cache file** is a file stored in the SSD and contains the second level cache page images.

2. **Memory-resident page directory** is a data structure which is used to keep track of the position of the cache pages in the SSD cache file. The page directory is an array of PageInfo elements. Each PageInfo is a data structure including all information we need to know about each page in the second level cache. PageInfo includes the slot number of the page in the second level cache as well as a few flags. Those flags indicate whether a page in the second level cache is fresher than its disk-resident version (is_fresher_than_disk), valid (is_valid), free (is_free), or accessed at least one time after admission (is_touched). It also includes pointers to the next and the previous pages in a hash chain, a pointer to the next free page in the linked list of free pages, as well as a latch for concurrency control. In order to improve performance and increase the concurrency by avoiding I/O bottlenecks, the memory-resident page directory is maintained in RAM. However, a lighter version of the page directory is maintained in the SSD to ensure the persistence of the mapping information.

3. **SSD-resident page directory** is a lighter version of the memory-resident page directory in which the PageInfo elements include only the slot number of each page in the SSD cache file as well as is_valid, is_free and is_fresher_than_disk flags.

4. **In-memory FIFO queue** is used to enforce the clock replacement policy. Whenever a candidate victim is dequeued from the FIFO queue, its is_touched flag is checked to see if it has been touched at least once since the last time it is enqueued. If it is not touched, then it will be evicted. Otherwise, its flag will be reset, and it will be enqueued again.

5. **In-memory hashtable** is created over the in-memory page directory for fast and concurrent lookups. The number of hash chains is determined dynamically, and it depends on the size of the SSD cache. The hash chains are build using the *next* and *pre* pointers of the PageInfo elements. Common optimization techniques are applied to maximize the concurrency of the hash table operators by minimizing the latch contention.

2.6 PC2: Optimizing PC with Batch Eviction

When the second level cache is full, a page must be evicted from the second level cache before the admission of every new page from the first level cache. When the victim is fresher than its corresponding disk-resident version, it must be flushed into the disk. Flushing a page from the SSD to the disk is a costly operation. The victim must be copied to the memory first, and then it must be flushed into the disk, and at the end, an *fsync* command must be issued to make sure that the page is written physically and persistently. At this time, the SSD slot of the victim can be reused safely for admission of a new page.

In our design, in order to reduce the impact of evictions from the second level cache, a batch of pages is evicted whenever eviction is necessary. Based on the replacement policy, a batch of candidate victims is selected, then each page in the batch is copied into the memory asynchronously. Then, they will be written into the disk asynchronously. After ensuring that all victims in the batch are written into the disk, an fsync command is issued. Finally, all SSD slots corresponding to those victims are released and added to the *free list* to be used for future admissions. Employing this technique improves performance after the second level cache becomes full and the eviction starts.

The benefits of this batch optimization are as follows: First, instead of one fsync command per write, we will need only one fsync for a batch of writes to the disk. Second,

the asynchronous reads from the SSD result in increasing the queue depth of the SSD. It is known that increasing the queue depth in SSDs significantly improves the I/O utilization by exploiting the I/O parallelism in the SSD. In the SSD which is used in our experiments, increasing the queue depth of the SSD to a number larger than 32 results in about 20 times improvement in random 4 Kbytes IOPS. Third, issuing the asynchronous write requests to the disk improves its I/O utilization. This improvement is more pronounced when a RAID array of multiple HDDs is used rather than a single HDD because, a RAID array, can perform multiple writes in parallel. The version of the PC method in which the batch eviction technique is employed is called PC2.

In the hope of improving the write performance of the HDD by reducing the number of seeks, we attempted to sort the selected victims based on their physical location in the disk before flushing them. However, it turned out that in practice the benefit of sorting is not enough to compensate its cost. This can be attributed to the fact that the batch size is much smaller than the size of the second level cache. Therefore, the chance of having many physically consecutive pages after sorting is pretty low. Moreover, according to our admission policy, we avoid admitting pages which are related to a sequential read. Consequently, the locality of the pages at the end of the queue tends to be low.

One drawback of PC2 is that during the batch eviction process the second level cache cannot admit any new page from the first level cache because no free space will be available in the SSD for admitting new pages until the entire batch is written in the HDD and fsynced. Thus, the average latency of transactions during the batch eviction process will be increased. However, immediately after finishing the batch eviction, the average latency of the transactions will be significantly improved until the next batch eviction happens. However, in-place update and read requests issued to the second level cache can still be handled while the batch eviction is in progress. This is possible with the help of latches. The PC2 method improves the throughput of the transaction processing at the cost of a negligible increase in the latency of some transactions.

2.7 Experimental Results

We implemented prototypes of the PC and PC2 methods as well as the LC method in SAP SQL Anywhere. In our implementation, the cache manager, I/O manager, checkpoint, and recovery components of SQL Anywhere are modified.

In this section, first, we compare the transactional throughput of PC, PC2, LC, an HDD-resident database, and an SSD-resident database. Then, we study the impact of PC

on recovery time. Finally, we measure the ramp-up time of LC after a crash recovery to understand how much PC wins over LC by avoiding a prolonged ramp-up period.

The objectives of the experiments presented in this section are as follows. First, we want to compare the transactional throughput of PC with that of LC. Second, we want to investigate the impact of the batch eviction mechanism of PC2 on transactional throughput. Third, we want to see that under what conditions PC can achieve the transactional throughput of an SSD-resident database. Fourth, we are interested in understanding what factors limit the transactional throughput of PC. Fifth, we want to study the recovery time of PC and how it is affected by the SSD cache size. Finally, we want to compare the ramp-up time of LC with that of PC after a crash recovery.

2.7.1 Experimental Setup

To compare the transactional throughput of PC, PC2, LC, the HDD-resident database, and the SSD-resident database, the TPC-C benchmark was used [6]. All experiments were executed on a server with an Intel Xeon Quad-Core E5420 2.50GHz processor, a RAID array of two 15K RPM HDD drives configured as RAID zero and a consumer level PCIe SSD drive which is used for the second level cache. The maximum advertised sequential throughput for read and write of the SSD drive are about 1.5 GB/s and 1.3 GB/s, respectively. The maximum random throughput for read and write of our SSD drive are 230,000 IOPS and 140,000 IOPS, respectively. Note that these numbers are the maximum possible numbers advertised by the manufacturer. In reality, the random performance depends mainly on parameters like queue depth, band size (physical address range in which the consecutive random I/Os are issued), the compressibility of data, the ratio of reads to writes, and the size of the free space on the SSD.

2.7.2 Transactional Throughput

Figure 2.3 shows the TPC-C transactional throughput of PC, PC2, LC, SSD and HDD, for three different databases, with 500, 1000 and 1500 warehouse and initial database sizes of 50GB, 100GB and 150GB, respectively. In each diagram, the HDD and the SSD curves represent scenarios in which the entire database is located on the HDD and the SSD, respectively, and no second level cache exists. The y-axis in all diagrams represents the number of NewOrder transactions executed per minute in the TPC-C benchmark. This metric is conventionally known as tpmC or tpm-C. The tpmC metric is measured by computing the average of the transaction rates over a period in which the second

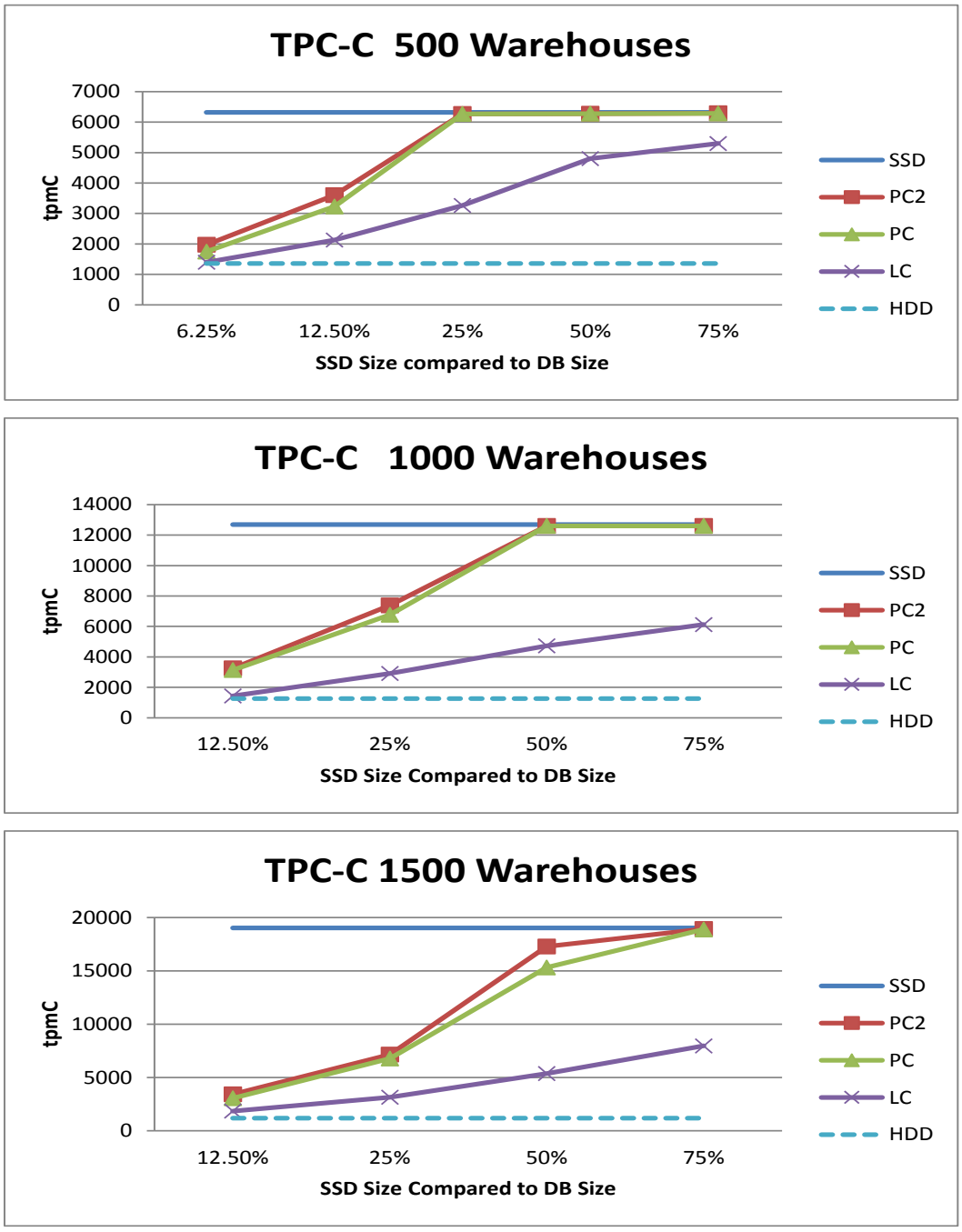


Figure 2.3: TPC-C transactional throughput under PC, PC2, LC, HDD and SSD. The HDD and SSD refer to settings in which the entire database is located on the HDD and SSD respectively and no second level cache is employed.

level cache has become full, and the tpmC graph reaches a steady state. The x-axis represents the size of the second level cache compared to the size of the database, as a percentage. In all experiments, the size of the first level cache is consistently about 3% of the size of the database. This way the impact of the size of the second level cache in transactional throughput will be isolated from the impact of the size of the first level cache. We observed that choosing larger percentages for the size of the first level cache will not have a considerable impact on transactional throughput (tpmC metric) in our workloads[3].

Each graph shows results for a different database size. In all experiments, the checkpoint interval is set to 30 minutes. This is the maximum possible checkpoint interval suggested by the TPC-C specification [81] and is usually used in the TPC-C benchmark report of the commercial database vendors. In all experiments, the reported throughput is the average throughput for two hours during the steady state. The cleaning threshold used in PC is set to 50%.

In all graphs, the SSD curve represents the maximum possible throughput for each workload-size, and the HDD curve represents a baseline, to be able to observe the amount of the effectiveness of the second level cache in different methods. At all times, the throughput of PC and PC2 is substantially better than that of LC. This is expected because, compared to PC, in LC the checkpoints are much more I/O intensive. In addition, LC employs a frequent cleaning process that increases the I/O contention even more. These two main parameters have a considerable negative impact on transactional throughput of LC.

As expected, in all experiments, PC2 performs either better than or as well as PC. Wherever the maximum throughput is achievable, PC and PC2 perform almost the same. We measured the average transaction response time of PC and PC2, and we observed that the difference between their average response times, in almost all experiments, is less than 1 ms, which is fairly negligible. PC2 sacrifices the response time of some transactions to improve the overall transactional throughput. However, this sacrifice does not have a considerable impact on the average response time.

[3]In general, in TPC-C workload, increasing the size of the memory buffer pool without improving the performance of the I/O subsystem does not have a considerable impact on transactional throughput. The TPC-C workload is a write-intensive transactional workload. In this workload, when the memory buffer pool gets full and the evictions start, since many of the eviction candidates are dirty, the I/O bandwidth to the stable storage becomes a performance bottleneck. Moreover, as we increase the size of the memory buffer pool, we will need to deal with longer checkpoint times. During a checkpoint, the transactional throughput is normally reduced. When the checkpoint time becomes longer, the transactional throughput drops for a longer period.

2.7.3 The Maximum Achievable Throughput

As we discussed before, employing the SSD as a second level cache makes sense when storing the entire database in the SSD is not feasible or cost-effective. Here, one interesting question is that if by using the SSD as a second level cache we can achieve a transactional throughput as good as when the entire database is stored on the SSD.

Figure 2.3 shows that in the 500-warehouse experiment, PC can achieve the transactional throughput of the SSD experiment, when the size of the second level cache is only 25% of the size of the database. In other words, in this workload size, instead of storing the entire database on the SSD, we can employ an SSD cache that holds only 25% of the database size and still achieve the same performance.

The maximum-throughput point shifts to 50% and 75% for the 1000-warehouse and 1500-warehouse experiments, respectively. We can conclude that, as the number of warehouses increases (as the database size gets larger), to achieve the maximum possible throughput, we need to cache a larger fraction of the database.

To be able to explain this behavior, we measured the rate of the I/Os issued to the SSD and HDD in each experiment, during the steady state. We noticed that in all experiments, there is a large gap between the rate of I/Os issued to the SSD and the maximum IOPS of the SSD. Also, we observed that, in any experiment in which the maximum throughput is not achievable, the HDD had reached its maximum IOPS. Based on these observations, we can conclude that, in our experiments, the SSD is never the bottleneck. Rather, the HDD is the bottleneck.

After the second level cache becomes warm, it handles a portion of the I/O requests that without a second level cache would have been handled by the HDD. When the second level cache is bigger, a greater portion of those I/O requests will be handled by the second level cache. After the second level cache gets full, the admission of the new pages from RAM into the SSD results in the eviction of the dirty pages from the SSD into the HDD. Suppose the rate of these extra disk I/Os as a result of evictions from the SSD is E_{iops} I/Os per second. Suppose the rate of the regular disk I/Os resulting from the first and the second level cache-misses together is CM_{iops} I/Os per second. When $CM_{iops} + E_{iops}$ is smaller than the maximum IOPS of the HDD, the maximum possible transactional throughput is achievable. Otherwise, the HDD becomes the bottleneck, resulting in reducing the transactional throughput.

In the 500-warehouse experiment, when the second level cache size is 25% of the database size, $CM_{iops} + E_{iops}$ is smaller than the maximum IOPS of the HDD. In the 1000-warehouse experiment, the maximum possible throughput is 2 times that of the 500-

warehouse experiment. This is because doubling the number of warehouses also doubles the number of terminals in the TPC-C benchmark. Therefore, the number of requested transactions per minute is doubled as well. A higher rate of requested transactions results in a higher I/O rate. Thus, in this experiment, when the second level cache size is 25%, $CM_{iops} + E_{iops}$ is no longer smaller than the IOPS of the disk. In this case, to reach the maximum throughput, we need a larger second level cache, i.e. 50%. A larger second level cache will reduce the rate of HDD I/Os by absorbing more I/Os, making it possible to achieve the maximum throughput.

In all experiments we performed, the I/O bandwidth of the SSD is underutilized. The important lesson we can learn here is that the impact of the second level cache on transactional throughput depends on the IOPS of the HDD. In other words, as the number of HDDs in the disk array increases, the maximum possible throughput can be achieved by employing a smaller second level cache. This important fact has been widely overlooked in prior studies. For instance, the authors of LC method in their experiments have used a large RAID array with 8 disks. The hardware acquisition cost, power consumption cost, and operational cost of a large array of disks could be potentially much more than the cost of the SSD. In addition, a large array of disks configured in RAID-0 is prone to component failure. This adds even more to the total cost of ownership of the hardware. In other words, using a large and powerful RAID array to improve the transactional throughput of LC, somehow defeats the purpose of using SSD as a second level cache. With the budget of a large disk array, we can buy a larger SSD and store the entire database on it.

To further confirm our argument about the impact of using a larger disk array on PC, we repeated the 500 and 1000-warehouse experiments over a RAID array with 8 disks. We observed that the maximum achievable throughput of PC shifts from 25% to 12.50%, and from 50% to 25% in 500-warehouse and 1000-warehouse workloads, respectively. In other words, in both cases, the amount of the SSD cache we will need to achieve the maximum possible throughput is almost halved when the number of disks in the RAID array is quadrupled.

The impact of the poor performance of the storage level (HDD) on the utilization of the second level cache can be reduced by employing a cleaner process for the second level cache. The cleaner process can be started during the idle I/O cycles of the system to flush the potential future dirty victims located in the second level cache into the disk. The potential future victims must be untouched and fresher than their disk-resident version to be selected for cleaning. To avoid reducing the cache hit rate, the cleaned victims should not be removed from the second level cache after being flushed into the disk.

Table 2.1: Recovery time and recovery rate comparison before and after employing the PC method

Elapsed time after the last checkpoint	Recovery time in HDD	Recovery time in PC	Recovery time speedup	tpmC Ratio to Min	Recovery rate speedup
1 Min	3:20	2:05	1.6X	4.6	7.36X
2 Min	5:25	3:05	1.75X	4.6	8.08X
3 Min	6:17	3:32	1.77X	4.6	8.18X

Table 2.2: Recovery time and recovery-rate comparison before and after employing the PC method with different SSD cache sizes. In all cases the elapsed time after the last checkpoint is 3 minutes.

Cache Size	Recovery time in HDD	Recovery time in PC	Recovery time speedup	tpmC Ratio to Min	Recovery rate speedup
6.25%	6:17	5:23	1.16X	1.32	1.54X
12.5%	6:17	4:17	1.46X	2.26	3.31X
25%	6:17	3:32	1.77X	4.6	8.18X
50%	6:17	2:53	2.17X	4.6	10.02X

2.7.4 Recovery Rate

Table 2.1 compares the recovery time of the database server with and without employing the PC method. This experiment has been done with 500 warehouses when the second level cache is 25 % of the database size. In PC, when the throughput is in steady state, after 1, 2 and 3 minutes from the end of the latest checkpoint, we killed the database engine process and measured the recovery time after the restart. We repeated the experiment 5 times, and the reported numbers are the average of the 5 trials. The maximum deviation in recovery time is 7 seconds. When the PC is used, the transactional throughput is about 4.6 times better than when no second level cache is employed. Therefore, in the same time interval after the latest checkpoint, the number of completed transactions when PC is employed is about 4.6 times more than that when no second level cache is used. The number of completed transactions has a direct impact on the recovery time. To have a fair comparison, we need to consider the recovery-rate-speedup, rather than the recovery-time-speedup. Recovery rate refers to the number of recovered transactions per second during the recovery process. To compute the recovery-rate-speedup, we need to simply multiply the recovery-time-speedup by 4.6 because the recovery rate when PC is employed is 4.6 times better than that when no second level cache is used.

Table 2.2 summarizes the impact of the cache size on recovery-time-speedups and recovery-rate-speedups with 500 warehouses. In all cases, when the throughput is in steady state, after 3 minutes from the end of the last checkpoint, we killed the database engine process and measured the recovery time after the restart. As expected, by increasing the SSD cache size, the amount of speed-up increases. This is because when the cache size is larger, a larger portion of the I/Os issued during the recovery process will be routed to the SSD, rather than the HDD.

As mentioned before, in LC, the recovery process is not modified. After each restart in LC, the content of the SSD is assumed to be clean. In other words, LC cannot benefit from the I/O performance of the SSD during the recovery process. Therefore, the recovery time of LC is the same as the recovery time of an HDD-resident database. Thus, we can use the speed-ups reported in Tables 2.1 and 2.2 to compare the recovery time and recovery rate of PC with LC as well.

2.7.5 Ramp-up Time After a Crash Recovery

Table 2.3 shows the ramp-up time of LC over different workloads when the size of the SSD cache is 50% of the database size. Ramp-up time refers to the time that LC needs to achieve the same transactional throughput as the one before the crash. Unlike in PC, in LC, after the recovery process is done, the cache is empty. For LC to reach the same transactional throughput as the one before the crash, the SSD cache should become populated again. The process of populating the SSD cache in LC is very slow. When the second level cache is empty, because of the high I/O contention on the HDD, the transactional throughput is very low. Consequently, the rate at which the pages in the memory cache get dirty is low too. In this case, the rate of eviction from the first level cache will be low as well. Therefore, at the beginning of the ramp-up period, the warming rate of the second level cache is very low. As the second level cache gets warmer, warming rate increases gradually. Moreover, in LC, during the cache warming period, the cleaner process and checkpoints add to the already high I/O load of the HDD. Besides, during the ramp-up period, some of the dirty pages in memory get clean at checkpoints before having a chance to be admitted to the SSD cache. All these facts contribute to the slow ramp-up time of the LC. As you see in Table 2.3, the ramp-up time gets longer as the workload size increases. This is because a larger workload uses a bigger cache. It takes a longer time for a bigger cache to become fully populated.

Unlike in LC, in PC, the SSD cache is already fully populated immediately after the

Table 2.3: Ramp-up time of LC when the Cache size is 50% of the database size

Workload Size	LC Ramp-up Time in hours
500-warehouse	8.2
1000-warehouse	14.2
1500-warehouse	29

recovery process. Therefore, in PC, almost[4] the same transactional throughput as the one before the crash is immediately achievable.

2.7.6 Summary of Conclusions From the Experiments

From the experiments presented in this section, we can conclude the following:

1. In all experiments, PC outperforms LC in terms of the transactional throughput. In some cases, the transactional throughput of PC is over 3 times higher than that of LC.

2. In all experiments, the transactional throughput of PC2 is either slightly better than or as good as that of PC. However, the superior transactional throughput of PC2 comes at the cost of a very minimal increase in latency of some transactions, as well as the additional implementation complexity. Since there is no significant performance difference between PC and PC2, we can conclude that replacing PC with PC2 is not a good idea.

3. In PC, by using a large enough SSD cache, we can achieve a transactional throughput as good as the transactional throughput of an SSD-resident database.

4. In experiments in which the maximum possible transactional throughput is not achieved, the throughput is limited by the random I/O capacity of the HDD. In those cases, the transactional throughput can be improved by employing a more powerful HDD array.

5. PC can improve the recovery rate of the database by over an order of magnitude. The recovery rate of the PC is improved by increasing the SSD cache size.

[4]The transactional throughput will be slightly lower at the beginning because it still takes some time for the memory cache to become warm, and also because all is_touched flags are reset after the recovery.

6. After a crash recovery, LC needs a ramp-up period as long as several hours while in PC, the SSD cache is warm immediately.

2.8 Review of Recent Literature

The PC method was originally proposed and patented in 2012. Since then, several related techniques have been proposed. In this section, we give a brief summary of that recent related work.

An enhanced version of LC is proposed by DeWitt et al. that uses the persistence of the SSD to improve the recovery and cache ramp-up time after a crash or restart [28]. In this enhanced version, three different restart mechanisms are presented. In the first method, which is called Memory-Mapped Restart (MMR), the page directory of the second level cache is stored in the SSD using a memory-mapped file. In the second method, which is called Log-based Restart (LBR), the changes to the page directory are logged in the transactional log file, and the page directory is reconstructed during the recovery using the recorded changes. The third method which is called Lazy Verification Restart (LVR), periodically and asynchronously flushes the page directory into the SSD, and after a restart, the latest stable version of the page directory is used, and its contents are verified on-demand lazily. Also, a simple technique called aggressive fill is introduced which results in up to 7X faster ramp-up time in LC method over a TPC-C workload. When the aggressive fill is employed, every single page HDD read request is expanded to read eight adjacent pages. Although the proposed restart mechanism results in up to 2.8x and 1.8x improvement in the peak-to-peak interval after a crash and a restart, respectively, the enhanced method still suffers from the expensive checkpoints and performance degradations as a result of the I/O activity of the I/O cleaning thread.

The Cost-Adjusted Caching (CAC) is a cost-based cache replacement algorithm for managing the SSD cache that together with the Greedy Dual algorithm (GD2L), which is used as a replacement policy for a memory buffer pool, aims to minimize the total I/O cost of the workload [65, 66]. To realize this goal, reference stats should be maintained in the memory for cached pages. Although CAC shows promising results in improving the cache hit rate, the impact of maintaining the reference statistics in CAC on performance when the workload gets large can be potentially high. Besides, the impact of checkpoints in degradation of the transactional throughput in CAC is unknown. As we discussed before, in write-intensive transactional workloads, checkpoints have a significant impact on transactional throughput. In CAC, similar to PC, the SSD has been considered as an extension of the disk. However, the recovery mechanism used in CAC for reconstructing the

SSD page directory is different from that in PC. In CAC, a snapshot of the page directory of the second level cache is stored periodically in the SSD. In order to reduce the number of required snapshots, an area containing k low priority pages is identified as an eviction zone on the SSD. Until the next snapshot, only the pages that fall in the eviction zone will be selected as eviction candidates. At the recovery time, the most recent copy of the page directory will be fetched from the snapshot log, and the eviction zone will be examined to update the inconsistencies in the page directory. Employing an eviction zone is a trade-off between the effectiveness of the replacement policy and the speed of checkpoints and the recovery process.

FaCE is another proposed approach for using SSDs as a second level cache [48, 49]. FaCE is designed based on this assumption that the sequential write bandwidth of the SSD is significantly better than its random write bandwidth. By focusing on this assumption, FaCE tries to improve the overall I/O performance by converting the random writes to the SSD to the sequential writes. In FaCE, a multi-version FIFO replacement policy (mvFIFO) is employed for management of the SSD cache. Pages admitted to the second level cache are always added to the rear of the FIFO queue, and the eviction candidates are selected from the head. FaCE does not perform any in-place page updates. If a new version of a page, which an older version of it already resides somewhere in the middle of the queue, gets admitted to the SSD cache, the older version gets invalidated in the page directory. In other words, all writes to the SSD are done sequentially.

The multi-version FIFO replacement policy has two main benefits. First, as claimed by the authors of FaCE, converting random writes to sequential writes will significantly reduce the write amplification factor[5] in the SSD, resulting in improving its lifespan. Second, it allows for a more efficient recovery mechanism. Because of the sequential nature of mvFIFO, the changes in the page directory as a result of admissions, evictions, and updates are done in consecutive addresses of the page directory, in time order. Therefore, these changes can be checkpointed one segment at a time. At the recovery time, the entire page directory can be recovered from the checkpoint log except for the latest segment which has been not checkpointed. The last segment can be reconstructed by scanning a small part of the FIFO queue that corresponds to that segment. This way, the impact of checkpointing the page directory on I/O bandwidth will be minimal.

Employing a multi-version FIFO queue in FaCE results in a potential underutilization of the second level cache because, at any point in time, there might exist multiple versions

[5]Write amplifications factor (WAF) refers to the ratio of the real physical writes performed on flash memory blocks in the SSD to the logical writes received by the SSD controller. The WAF depends on several parameters such as the size of writes, the locality of write and the degree of write randomness.

of the same page in the cache. This redundancy will reduce the cache hit rate. By reducing the hit rate of the SSD cache, I/O traffic to the HDD is increased. Consequently, compared to other caching methods such as PC or CAC, in which a redundant cache is not employed, in FaCE, there will potentially be higher I/O pressure on the HDD layer. Since FaCE is not able to convert the random writes issued to the HDD into sequential writes, the extra I/O pressure on the HDD should be compensated for by using a more powerful disk array. Otherwise, the disk layer will potentially become a bottleneck. As we discussed in Section 2.7.3, this will defeat the purpose of using SSDs as a second level cache. The fact that FaCE is evaluated over a RAID-0 disk array with 16 disks can support this hypothesis. Testing this hypothesis by performing a real comparison of the transactional throughput of FaCE with that of PC and CAC is an interesting direction for future work.

Another possible problem is that in FaCE, every database checkpoint results in the admission of a large number of dirty pages into the FIFO queue. In a write-intensive workload, after cache warm-up, it is very likely that many of those admitted pages already reside in the queue, resulting in a high degree of redundancy. This redundancy can significantly reduce the cache hit rate. The impact of the checkpoint can be even higher when the SSD cache is full, and therefore, a large number of evictions might be required to make room for the new dirty pages admitted to the SSD.

The advent of big-memory many-core machines has brought about a flurry of research and development into main-memory database systems in recent years [32, 70, 95] . H-Store [88] and HyPeR [50] are examples of main-memory multi-core systems developed in academia as research platforms. SAP HANA [86], Oracle TimesTen [58], Microsoft SQL Server Hekaton [29], VoltDB [89], and MemSQL [85] are some of the commercial main-memory database systems developed in recent years. By effective exploitation of terabytes of RAM as well as a large number of CPU cores, main-memory databases easily outperform their disk-resident counterparts. However, due to their high cost of ownership, main-memory databases are still mostly used in specific performance-hungry use cases. In terms of price and performance, there is a spectrum of possible database solutions. On one side of the spectrum lies the very expensive main-memory database with the highest performance. A fully SSD-based database is placed next with lower cost and performance. A disk-based database which is equipped with a second level SSD cache is placed next with even lower cost and performance. Finally, a fully disk-based database lies in the other side of the spectrum with the lowest cost and performance. Choosing the right solution for a specific use case depends on many parameters such as the database size, workload, expected performance, software architecture, and the availability and cost of resources, especially in the cloud. As the cost of the SSD and RAM decreases, choosing a solution from the hight performance side of the spectrum becomes more affordable. However, at

the time of writing this thesis, the disk-side of the spectrum is still extensively used in the industry, especially when the database is very large.

An alternative to implementing a persistent second level cache inside the database engine is to use a block layer persistent SSD cache, e.g. Fashcache [51], bcache [1], or dm-cache [2]. The benefit of using a block layer persistent cache is that it avoids the complexities associated with changing the database engine. One interesting question here is how PC would compare with a block layer persistent SSD cache. Since a block layer persistent cache works in isolation from the database engine, it cannot benefit from some useful information accessible by the database engine to improve the effectiveness of the caching scheme. For instance, the admission policy in PC accepts only random pages. Whether a specific page residing in the first level cache of the database has been read randomly or sequentially is know only by the database engine. A block layer cache will admit all pages regardless of whether they are random or sequential, resulting in underutilization of the persistent cache. As another example, in PC, at checkpoint time, for every dirty page residing in the first level cache, if a version of that page is already residing in the second level cache, it is flushed there, otherwise it is flushed to the HDD. This policy cannot be applied when a block layer cache is used because a block layer cache has no idea that a checkpoint is in progress. Therefore, it will admit all pages to the second level cache, resulting in excessive evictions from the second level cache to the HDD to make room for new admissions to the second level cache. Because of the low I/O bandwidth of the HDD, these excessive evictions will result in a significant degradation in transactional throughput during the checkpoints. Kim et al. has shown that the throughput under an OLTP workload actually degrades by 79.1% with flashcache, compared to the baseline performance [52]. An experimental comparison of PC with the existing block layer caching schemes is an interesting avenue for future work.

Chapter 3

Parallel I/O Aware Query Optimization

3.1 Overview

No matter what type of storage device is used, a natural way of improving I/O throughput is to perform I/Os sequentially[1]. In use cases in which sequential I/O is not possible, the only remaining option is random I/O, i.e. performing small I/Os from/to non-consecutive addresses. It is known that the throughput of random I/O in HDDs is very poor because HDDs are mechanical devices in which a physical head needs to move from one cylinder to another in order to perform random I/Os. These mechanical movements are slow. Moreover, in HDDs, there is almost no chance of performing more than one I/O at a time. In contrast, there are no moving parts in SSDs. Therefore, both latency and throughput of random I/O in SSDs are much better than those in HDDs. In addition, modern SSDs can substantially benefit from *I/O parallelism*, the ability to perform multiple I/Os simultaneously. Chen et al. have studied the substantial impact of parallelism in I/O performance of SSDs [21]. A modern SSD is capable of utilizing multiple levels of parallelism: plane, channel, package, and die levels. Almost all modern SSDs support native command queuing mechanisms (NCQ), which were first introduced in the SATA II standard [24].

These capabilities allow the SSD to accept multiple concurrent I/O requests or a burst of successive I/O requests from the operating system. The received I/O requests are queued up and the host interface will reorder them to make a favorable I/O pattern for the internal parallel organization of the SSD. In other words, increasing the *I/O queue depth*, defined as the number of outstanding I/Os in the I/O queue at any point of time, of modern SSDs will give them a chance to exploit their own internal parallel organization to improve the I/O throughput. The I/O queue depth can be increased by issuing multiple I/Os at the same time. Alternatively, issuing I/O requests with a rate faster than the rate of handling I/O requests by the device can increase the I/O queue depth.

Experiments show that if we increase the I/O queue depth of SSDs, we can observe a significant improvement in random I/O throughput. Figure 3.1 shows the impact of increasing queue depth on the throughput of 4 Kbytes random reads in the SSD and HDD.

[1] A sequential I/O pattern can be achieved by either a few large I/Os or many small I/Os with consecutive addresses. Each I/O request involves some processing overhead including the time it takes to receive the requests and do the required address translations. Therefore, performing fewer but larger I/Os has a lower overhead, and it usually results in a better throughput compared to performing many small I/Os. In some storage devices, the controller of the device can look at the I/O queue to see if there are many small consecutive I/Os, and then convert them to a single large I/O request which can be processed more efficiently. However, even combining several small I/Os has some extra overhead that makes the throughput of a single large I/O better than multiple small I/Os. Moreover, the controller cannot wait, more than a particular amount of time, for the next small I/Os to come and so the size of the final combined large I/O is limited. Generally, the I/O throughput improves by increasing the I/O size up to a specific threshold.

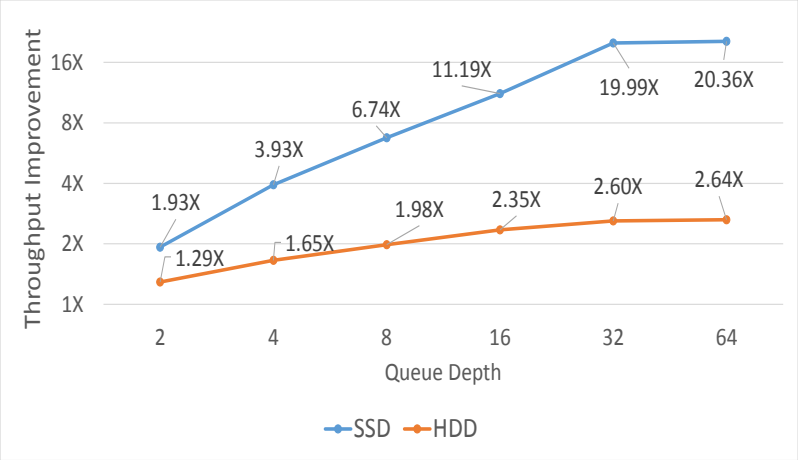


Figure 3.1: Impact of queue depth on throughput of random 4KB reads on the SSD and HDD

In this experiment, a 240GB OCZ RevoDrive 3 X2, which is a consumer-grade PCIe-based SSD, and a 500GB Western Digital 7200 RPM SATA III HDD, which is a commodity 7200 RPM HDD, are used. To increase the I/O queue depth in the random 4K read workload multiple threads are used. These threads send synchronous read requests to the storage device.

By increasing the SSD queue depth to 32, the throughput of random I/O improves by a factor of 20. This is roughly 51.7% of the throughput of the sequential I/O in this SSD. After that point, no further tangible throughput improvement is observed. However, on the HDD, by using a queue depth of 32, the throughput improves by a factor of only 2.64[2]. This is only 1.3% of the throughput of sequential I/O in the HDD. This experiment shows that unlike on the HDD, on the SSD, if we can increase the queue depth of random I/O, an I/O throughput very close to that of sequential I/O is achievable. In other words, unlike in the HDD, in the SSD, the performance gap between sequential and random I/O can be significantly reduced by exploiting I/O parallelism.

Pelley et al. argue that an SSD-oblivious query optimizer is unlikely to make significant errors in choosing the best access method [78, 79]. We will show that when the I/O parallelism is employed in the execution of the access methods, this argument is no longer true. Traditionally, most database query optimizers are designed by considering the substantial disparity between the runtime of random and sequential I/O in HDDs. Moreover, con-

[2]This little improvement in the HDD is as a result of reordering the I/O requests in the I/O queue of the device controller to reduce the number of movements between cylinders or to combine consecutive I/O requests.

ventionally, query optimizers assume that there is no considerable performance difference between parallel and non-parallel I/O. These assumptions have been proven to work very well over decades of using HDD-based storage subsystems. However, by moving from the HDD to the SSD, relying on the same assumptions will result in sub-optimal optimization decisions. In this chapter, we will study the impact of exploiting I/O parallelism on query processing and optimization.

Lee et al. [61] studied the impact of inter-query parallelism in exploiting the parallel I/O capability of SSDs. Roh et al. [82] proposed a new approach for executing multiple index scans at the same time. They showed that their proposed method increases the I/O queue depth of the SSD and consequently improves the total execution time. Their method exploits the inter-query parallelism to increase the I/O queue depth. To the best of our knowledge, prefetching and intra-query parallelism have been not studied as methods for exploiting the I/O parallelism in SSDs. In this chapter, we will see how these techniques can be employed to utilize the parallel I/O capability of SSDs.

While there are some studies showing approaches to improve I/O parallelism in database systems, the query optimization problem has not been fully addressed in this context. In particular, a query optimizer that operates on a range of storage technologies (HDD, RAID HDD, SSD, and even future technologies) must have a way to determine likely benefit of I/O parallelism in order to distribute parallelism opportunities among query operators. In this chapter we propose a novel, general, and dynamic I/O cost model for accurate I/O cost estimation of those database operators which can benefit from I/O parallelism. This model, dynamically defined by a calibration process, accurately summarizes the behavior of the storage device, no matter how much it can benefit from I/O parallelism. The proposed model has been implemented in SAP SQL Anywhere. Experiments show that leveraging the new model in the query optimizer results in selecting execution plans with up to 20 times shorter runtime.

3.2 Summary of Contributions

The contributions of this study can be summarized as follows:

1. Characterizing the impact of I/O parallelism in database scan operators, showing how the decision made by the query optimizer can be affected when parallel I/O is employed.

2. Showing that, contrary to popular belief, when the I/O parallelism is employed in the execution of the access methods, the query optimizer must be aware of the benefit of I/O parallelism in the underlying storage device.

3. Demonstrating that the intra-query parallelism and the effective use of prefetching can be used as two alternative approaches for exploiting the I/O parallelism of SSDs in index scans, to eliminate the limitations of the existing approaches.

4. Introducing a novel, general, and dynamic I/O cost model for accurate I/O cost estimation of those database operators which can benefit from I/O parallelism.

5. Introducing a practical method for initializing, calibrating and employing the proposed cost model.

3.3 Background

Choosing the optimal access method for a given query is one of the fundamental and classic problems in database systems. This problem has been studied since 1970 [33, 84]. Index scan and full table scan (hereafter, IS and FTS, respectively) are two traditional access methods which are implemented in all database systems[3]. When a relevant index is available, the database engine traverses the index to find and fetch only the required rows that satisfy the given scan predicate. Unlike IS, in FTS all rows within a table are fetched and scanned one-by-one to find those rows which satisfy the given predicate.

FTS suffers from unnecessary I/Os because it must read all table pages, whether relevant or not. In addition, for any retrieved page, all rows within the page must be evaluated against the given predicate. This results in the execution of extra CPU instructions. This problem becomes more pronounced when there are a large number of rows which are rejected by the predicate. However, FTS benefits extensively from an efficient sequential I/O pattern. Unlike FTS, in IS, with the guidance of an index, only relevant table pages are fetched. In addition, it is not necessary to scan all rows within every fetched page.

[3]In this chapter, wherever we are talking about index scan we are referring to non-clustered index scan. An index can be clustered, non-clustered, or partially clustered. In a clustered index, the physical order of rows in the table matches the ordering of the indexed key. Although using a clustered index is very efficient, creating clustered index is not a common practice, due to the extra overhead of maintaining the physical order in update-intensive workloads. Moreover, for a given table, only a single clustered index can be defined while more than one non-clustered index may exist. In a partially clustered index some statistics are computed and maintained to indicate how close the index is to a clustered index. These statistics are employed to improve the accuracy of I/O cost estimation.

Therefore, the number of CPU instructions executed for an index scan could be significantly smaller. However, index scan suffers from an expensive random I/O pattern. Due to the substantial disparity between sequential and random I/O in hard disk drives, the significantly more expensive I/O in index scan plays an important role in preferring FTS over IS over a large selectivity[4] range. Another disadvantage of the index scan is that in this method when the selectivity is large, all the table pages might be fetched and it is also very likely that the same table pages will be retrieved over and over again. When the memory buffer pool is small, these repetitive retrievals result in extra disk I/Os. Therefore, for a large enough selectivity, the total number of pages fetched using IS can be even greater than those fetched using FTS.

Traditionally, it is known that the selectivity break-even point between IS and FTS occurs at around 10 percent [34]. It means that when less than 10 percent of the rows in a table satisfy a given predicate, then it is better to use an IS to scan the table. However, recent studies [78, 79], as well as our own experiments, show that on today's modern storage devices this number is much smaller than 10 percent.

The selectivity break-even point depends on two major parameters: the size of memory buffer pool, and the number of records per page. Having a larger memory buffer pool improves the performance of the index scan as it is more likely that the re-referenced table pages can be fetched from the memory buffer pool rather than the disk. If the number of rows per page increases (or equivalently the row size gets smaller) the performance of index scan becomes worse. Hence, the break-even point shifts toward smaller selectivities. When a page contains only a single row, the number of rows retrieved is equal to the number of fetched pages. However, as the number of rows per page increases, even at small selectivity, the number of pages that must be fetched quickly approaches 100% of the table pages. When the available memory is small, this number goes beyond 100% of the table as some pages might be fetched multiple times. One study [94] proposes an analytical formula for the expected number of pages retrieved, given the size of the table, number of rows per page, and selectivity. Many commercial optimizers use their own formulas for the cost model of IS and FTS.

Some modern database management systems, like SAP SQL Anywhere, support intra-query parallelism [39, 91], which involves having more than one CPU core handle a single query simultaneously so that portions of the query result are computed in parallel on multi-processor hardware. *Parallel full table scan* and *parallel index scan* (hereafter, PFTS and

[4]Selectivity of a SQL query in the form of "SELECT * FROM T WHERE P" refers to the percentage of rows in T that satisfy the predicate P. Selectivity estimation in database management systems is a well-studied problem[20, 34, 80].

Figure 3.2: Parallel full table scan (PFTS) in SAP SQL Anywhere. Each color represents a different worker.

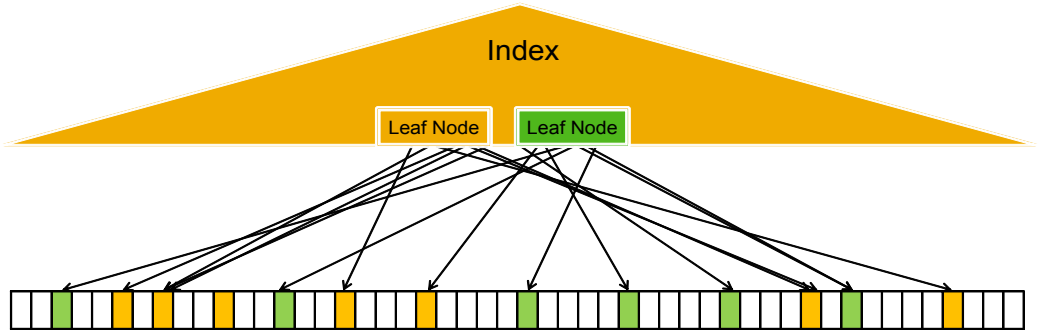


Figure 3.3: Parallel index scan (PIS) in SAP SQL Anywhere. Each color represents a different worker.

PIS, respectively) are two basic algebraic operators that can be executed in parallel in SAP SQL Anywhere. *Parallel hash join*, *parallel nested loop join*, *parallel hash filter* and *parallel hash group by* are some other operators in SQL Anywhere for which intra-query parallelism is supported. The focus of our study is on PIS and PFTS.

Figure 3.2 and Figure 3.3 show a schematic view of PFTS and PIS operators, respectively. In PFTS, each worker fetches a table page and starts processing the rows within that page. The workers fetch table pages one by one. As soon as a worker has finished processing all rows within a page, it fetches the next available page and starts processing it. To improve the I/O performance, instead of reading pages one by one from disk, a large block consisting of several consecutive pages is read at a time. A prefetching mechanism is employed that guarantees prefetching up to n blocks ahead of the current page which is being currently processed, asynchronously, where by default n is two times the number of workers used in PFTS. Therefore, when a worker requests the next available page, the page might be already in memory buffer pool.

PFTS is a CPU intensive operator. CPU utilization during a PFTS is usually close to 100%. Because of the sequential I/O pattern and efficient prefetching mechanism in PFTS, CPU will never wait for I/O. The only exception is when the number of CPU cores is so high that the storage device reaches its maximum possible bandwidth. In that case, the CPU is blocked for I/O, and the CPU utilization goes below 100%.

In PFTS, especially on the SSD, the I/O queue depth is usually smaller than the number

of workers. That is because in PFTS the speed of processing pages is usually lower than the speed of fetching pages from disk into memory. In other words, to increase the I/O queue depth, more CPU cores are required.

In PIS, one worker traverses the index from root to leaf level and finds the range of leaf pages which must be accessed. Then, leaf pages are retrieved and processed by multiple workers one by one. Each leaf page consists of (key, row_id) tuples. Each thread retrieves an index leaf page, and then goes over all row_ids in it one by one and retrieves the corresponding table page for each row_id. By profiling the I/O queue depth of the SSD during the execution of the PIS operator using n workers, a queue depth of n is clearly observable. Since the time interval between issuing consecutive I/O requests by each worker is much shorter than the I/O latency of the storage device, at any point of time, during the execution of PIS, the number of outstanding I/Os will be n. This pattern is observable in all cases except in very selective queries in which the number of leaf pages which are required to be retrieved is smaller than the number of workers. Thus, the I/O pattern of PIS with *parallel degree*[5] n is the parallel random I/O with a constant queue depth of n.

Unlike in PFTS, in PIS the pages are not read from disk in multi-page blocks. That is because, unlike in PFTS, in PIS the pages needed to be fetched are not necessarily consecutive unless the index is clustered.

3.4 Characterizing the Impact of I/O Parallelism in Scan Operators

In this section, we characterize the impact of I/O parallelism in scan operators in SAP SQL Anywhere. We will demonstrate how the query optimizer's choice for the best access method is affected by the exploitation of the parallel I/O by the IS, FTS, PIS, and PFTS access methods. In particular, we explore the magnitude of the shift from non-parallel to parallel selectivity break-even points in different configurations. Non-parallel and parallel selectivity break-even points refer to particular points in the selectivity range in which the runtime curve of the IS access method crosses the FTS runtime curve, and the runtime curve of the PIS access method crosses the PFTS runtime curve, respectively.

The main goal of the experiments presented in this section is to show why the query optimizer must be aware of the impact of I/O parallelism on the underlying storage device.

[5]Parallel degree of a database operator is defined as the number of workers used for the parallel execution of that database operator.

Table 3.1: Experimental configurations

Experiment	Table	Rows per page	Device	Memory buffer pool	Table Size
E1-HDD	T1	1	HDD	5%	1.9 GB
E1-SSD	T1	1	SSD	5%	1.9 GB
E33-HDD	T33	33	HDD	5%	1.15 GB
E33-SSD	T33	33	SSD	5%	1.15 GB
E500-HDD	T500	500	HDD	5%	781 MB
E500-SSD	T500	500	SSD	5%	781 MB

As discussed in Section 3.3, intra-query parallelism in PIS and PFTS is one way to increase the I/O queue depth. We will also demonstrate how an effective prefetching mechanism can increase the I/O queue depth in index scans.

3.4.1 Experimental Setup

As mentioned in Section 3.3, the number of rows per page and the size of the available memory buffer pool are two important factors in determining the position of the selectivity break-even point. In order to consider the impact of the number of rows per page, we fixed the size of memory buffer pool and performed three sets of experiments with different numbers of rows per page. In all experiments, the amount of buffer pool memory is about 5% of the size of the table.

The first, second and third set of experiments are performed on tables T1, T33, and T500, respectively. Tables T1 and T500 represent two extreme cases for studying the impact of a very large and very small row size in the performance of access methods. T33 represents a case in between which represents a non-extreme case. Each set consists of 2 experiments. The first one uses an HDD and the second one uses an SSD. The numbers of rows in T1, T33 and T500 are 500,000, 10,000,000, and 100,000,000, respectively. The sizes of T1, T33 and T500 are 1.9 GB, 1.15 GB, and 781 MB, respectively. Note that, since the relative size of the memory buffer pool to the table in all experiments is constant (5%), our experimental results do not depend on the table size. In our experiments, we are only interested in selectivity break-even-points. A configuration summary of all experiments is given in Table 3.1.

In all experiments the following query is used:

$$SELECT\ MAX(C1)\ FROM\ Ti\ \ WHERE\ C2\ BETWEEN\ low\ AND\ high \quad (3.1)$$

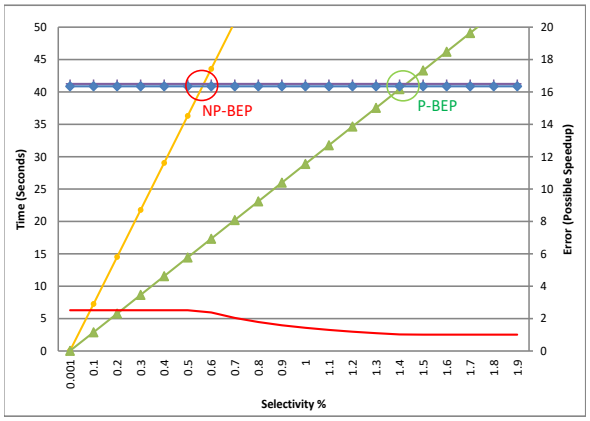
where *low* and *high* are used to control the selectivity. Tables T1, T33, and T500 include columns C1 and C2 plus some additional columns. The additional columns are used as

padding to adjust the target row size. A non-clustered index is created on C2. No index is created on C1. The data type of all columns is integer and the values in each column are a random permutation of the integers in [1,n], where n is the number of rows in the table[6]. All experiments have been done on a machine with a quad-core Xeon W3530 2.80GHz CPU, a 7200 RPM hard disk drive, and a consumer level PCIe SSD drive. The maximum advertised sequential throughput for read and write of the SSD drive are about 1.5GB/s and 1.3GB/s, respectively. The maximum random throughput for read and write of our SSD drive are 230K IOPS and 140k IOPS, respectively. These numbers are the maximum possible numbers. In reality, the random I/O performance depends mainly on parameters like queue depth, band size (physical address range in which the random I/Os are issued), the compressibility of data, the rate of reads to writes, and the size of free space on the SSD.

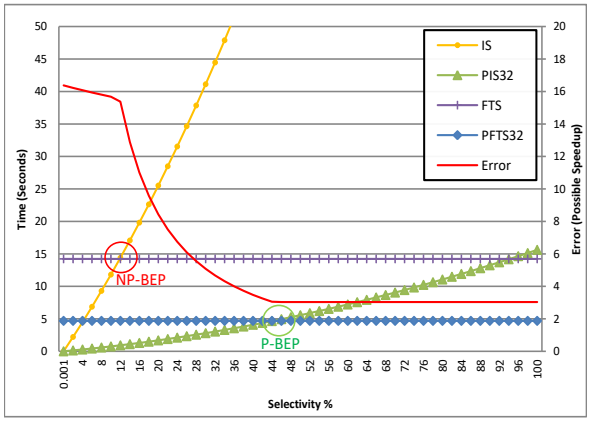
3.4.2 Experimental Results

In Figure 3.4, diagrams (a), (b), (c), (d), (e), and (f) represent experiments E1-HDD, E1-SSD, E33-HDD, E33-SSD, E500-HDD, and E500-SSD, respectively (see Table 3.1). In all diagrams, the x-axis represents a selectivity range. The selectivity range and its scale in each diagram is different. The selectivity range in each experiment is chosen such that it covers both parallel and non-parallel break-even points. In each diagram, the y-axis represents the total runtime of the execution of the Query 3.1. Each curve shows the execution time of the query executed using a different access method. PIS32 and PFTS32 refer to PIS and PFTS with a parallel degree of 32, respectively. To improve the readability of the diagrams, the curves related to parallel degrees 2, 4, 8, and 16 are omitted from all diagrams. In addition, to improve the clarity of the graphs we have replaced all curves with their corresponding best fit curves. In each diagram, the parallel and non-parallel break-even points are indicated by green and red circles, respectively. In all graphs, the amount of error, defined as possible speedup after considering the parallelism, is represented using a red curve. For any selectivity, the error is computed by dividing the execution time of the best non-parallel access method by that of the best parallel access method. In each graph, the y-axis on the right side of the graph shows the error (possible speedup) range. The minimum, maximum and range of y-axises in all graphs are similar. This makes the inter-graph comparisons easier.

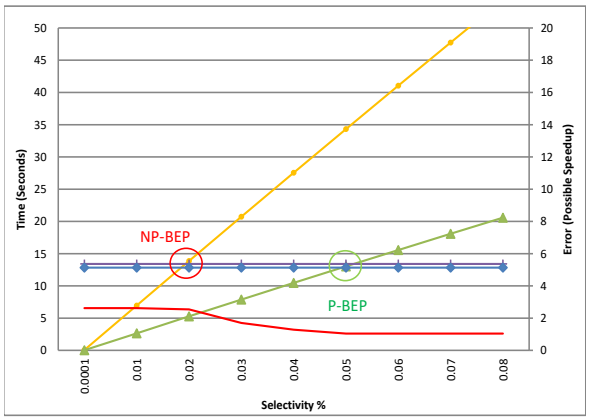
[6]To generate a random permutation, we start with a sorted array of keys in which key[i]=i (for i=1 to n). Then, for each k (k = 1 to n-1), using a uniform pseudo-random generator, we generate an integer r between k and n, and we swap key[k] with key[r].



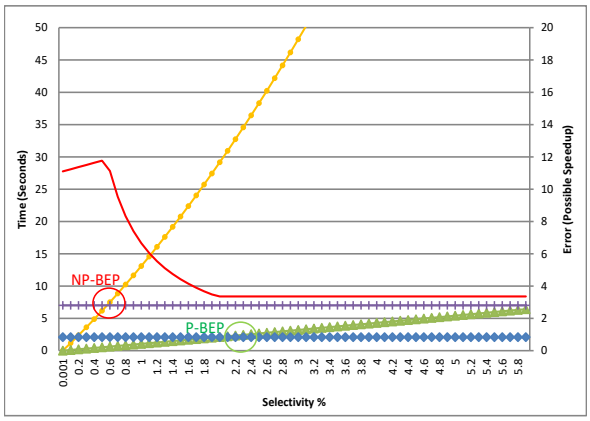
(a)E1-HDD, one row per page



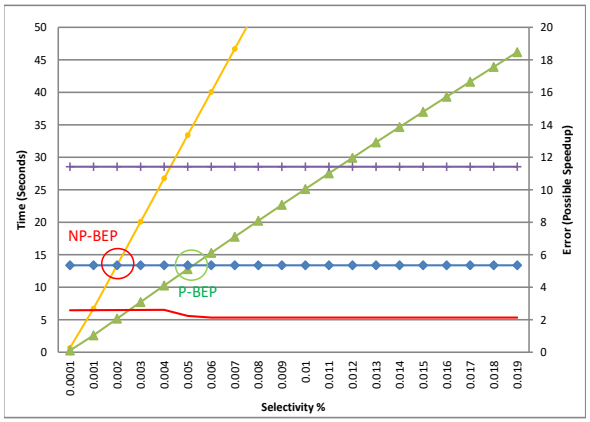
(b)E1-SSD, one row per page



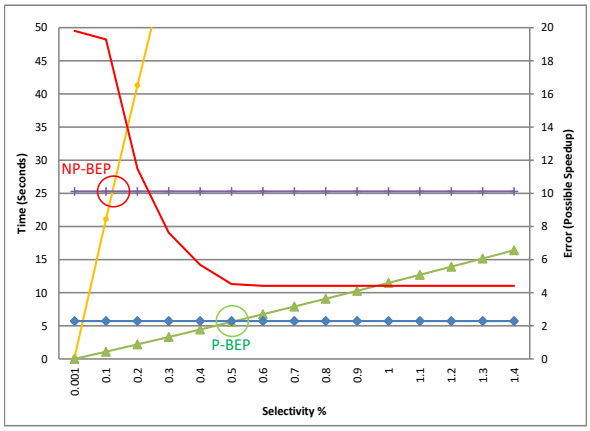
(c)E33-HDD, 33 rows per page



(d)E33-SSD, 33 rows per page



(e)E500-HDD, 500 rows per page



(f)E500-SSD, 500 rows per page

Figure 3.4: Runtime of Query 3.1 using IS, FTS, PIS32 and PFTS32 access methods over tables T1, T33 and T500 on the HDD and SSD. In each graph, the red and green circles indicated the non-parallel and parallel break-even points, respectively.

Table 3.2: Summary of non-parallel and parallel break-even points on the HDD and SSD in different experiments. NP- refers to the crossing point of IS and FTS, and P- refers to the crossing point of PIS32 and PFTS32

Rows per page	NP-HDD	P-HDD	NP-SSD	P-SSD
1	0.55%	1.4%	12%	46%
33	0.02%	0.05%	0.55%	2.05%
500	0.0045%	0.005%	0.11%	0.5%

Table 3.3: Summary of shifts in selectivity break-even points in different experiments. NP- refers to the crossing point of IS and FTS, and P- refers to the crossing point of PIS32 and PFTS32

Rows per page	$HDD\text{-}SHIFT_{NP \rightarrow P}$	$SSD\text{-}SHIFT_{NP \rightarrow P}$	$NP\text{-}SHIFT_{HDD \rightarrow SSD}$	$P\text{-}SHIFT_{HDD \rightarrow SSD}$
1	0.85%	34%	11.45%	44.6%
33	0.03%	1.5%	0.53%	2%
500	0.0005%	0.39%	0.1055%	0.495%

Before analyzing the experimental results, for the sake of clarity, let's make some definitions.

- $HDD\text{-}SHIFT_{NP \rightarrow P}$ refers to the amount of shift (in percentage) from non-parallel to parallel selectivity break-even point in the HDD
- $SSD\text{-}SHIFT_{NP \rightarrow P}$ refers to the amount of shift (in percentage) from non-parallel to parallel selectivity break-even point in the SSD
- $NP\text{-}SHIFT_{HDD \rightarrow SSD}$ refers to the amount of shift (in percentage) from non-parallel selectivity break-even point in the HDD to that in the SSD
- $P\text{-}SHIFT_{HDD \rightarrow SSD}$ refers to the amount of shift (in percentage) from parallel selectivity break-even point in the HDD to that in the the SSD

All non-parallel and parallel selectivity break-even points are summarized in Table 3.2. All the shifts in selectivity break-even points (as defined above) are summarized in Table 3.3. The summary of our observations from Figure 3.4 and Tables 3.2 and 3.3 are as follows:

1. In all graphs, there exists a selectivity range in which the amount of error is larger than 1.

2. In all SSD-based graphs, i.e. graphs (b), (d), and (f), for the entire selectivity range, the amount of error is larger than 3. Even in graphs (c) and (e) the amount of error in the entire selectivity range is larger than 1.

3. In HDD-based experiments, the maximum amount of error is 2.62, and the maximum error occurs only in a very narrow selectivity range.

4. In SSD-based experiments, the minimum and maximum errors are 3.03 and 19.29, respectively.

5. Compared to HDD-based experiments, in SSD-based experiments, the selectivity range in which the amount of error is larger than its minimum is significantly wider. This selectivity range becomes wider as the number of rows per page is reduced. For instance, in graph (b), for all selectivities smaller than 46% the amount of error is larger than 3. As the selectivity decreases, the amount of error increases and reaches to over 16 in selectivities close to zero.

6. $HDD\text{-}SHIFT_{NP\rightarrow P}$, $SSD\text{-}SHIFT_{NP\rightarrow P}$, $NP\text{-}SHIFT_{HDD\rightarrow SSD}$ and $P\text{-}SHIFT_{HDD\rightarrow SSD}$ are all inversely proportional to the number of rows per page.

7. No matter how many rows there are in each page, $SSD\text{-}SHIFT_{NP\rightarrow P}$ is larger than $HDD\text{-}SHIFT_{NP\rightarrow P}$. This difference is more pronounced as we reduce the number of pages per row.

8. No matter how many rows there are in each page, $P\text{-}SHIFT_{HDD\rightarrow SSD}$ is larger than $NP\text{-}SHIFT_{HDD\rightarrow SSD}$. This difference is more pronounced as we reduce the number of pages per row.

From the above observations, we can conclude that, in an SSD-resident database, a parallel-I/O-oblivious query optimizer may make decisions with an execution time more than 19 times worse than the optimum decision. We can also conclude that, compared to the HDD, in the SSD, the range of selectivities in which a parallel-I/O-oblivious query optimizer is prone to significant errors is much wider. This is contrary to the findings of Pelley et al. [78, 79], mainly because they have neglected to consider the impact of I/O parallelism in their analysis.

3.4.3 Employing Prefetching

In Section 3.3, we described how the I/O queue depth in index scans can be increased using intra-query parallelism. Each worker thread devotes a portion of the system resources, e.g.

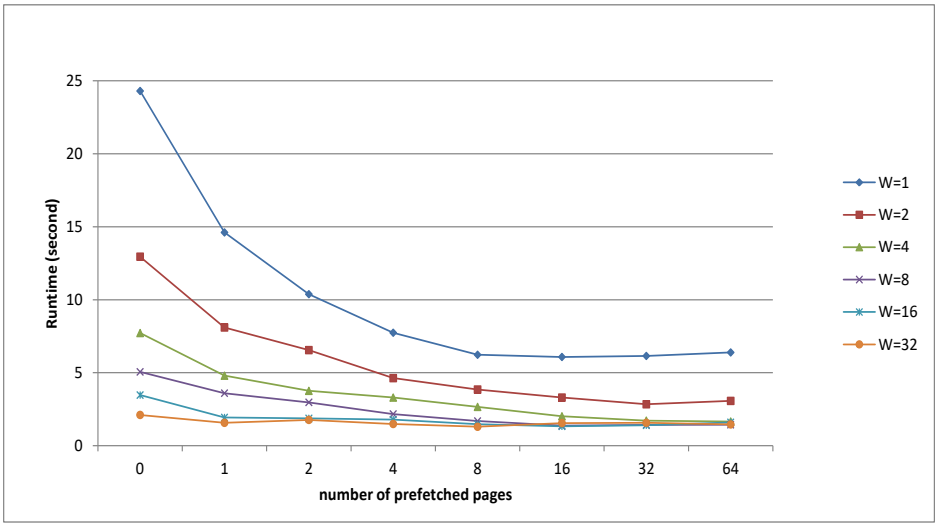


Figure 3.5: Index scan runtime with different parallel degrees when prefetching is enabled in each worker. Each curve represents a different parallel degree

the available memory, to itself. The total number of worker threads is therefore limited. If one query consumes a large number of available worker threads then other queries might not be able to start. This will reduce the concurrency of the system. Moreover, coordination and synchronization of threads introduces extra overhead. Thus, a non-parallel plan is usually preferable to a parallel query plan when their estimated costs are close to each other.

As mentioned before, because of the low performance of random I/O, the index scan is not a CPU intensive access method. While we would like to generate a high queue depth to exploit I/O parallelism, doing so with a large number of worker threads can be wasteful. An alternative approach is to employ asynchronous prefetching.

We implemented prefetching in the index scan operator of SAP SQL Anywhere, with each of W workers prefetching up to P pages that are expected to be required in the near future. In this case, the expected peak queue depth is WP. For the sake of simplicity, we only prefetch table pages referenced by a single index leaf page. As a worker gets closer to the last page which is referenced by a leaf index page, the number of prefetched table pages for that worker is reduced until it moves to another index leaf page. Since the number of (key, row_id) tuples in each index leaf page is typically large, this simplification does not have a large impact on the overall runtime of the index scan.

One interesting question here is the extent to which intra-query parallelism can be

replaced by the prefetching method we described above. To answer this question we performed an experiment. Figure 3.5 shows the impact of prefetching on index scans with different parallel degrees. Parallel degree refers to the number of workers used to execute the index scan. In this experiment, a range index scan over a large table with 80 million rows has been performed. The number of rows in each page is 33. Each curve represents the execution of index scan with a different number of workers. The x-axis indicates P, the number of prefetching requests issued by every individual worker. The y-axis shows the total execution runtime. In this experiment, the selectivity of the predicate is 0.03%.

Consider two different executions of the index scan such that in the first one W is 0 and P is n and in the second one, W is n and P is zero. Intuition suggests the same execution time in these two cases. However, this is not the case. In other words, intra-query parallelism cannot be completely replaced by prefetching[7]. That is mainly because of the mentioned simplification we made in the implementation of prefetching which results in an imperfect overlapping of I/O and CPU. Nevertheless, the good news is that by combining prefetching with intra-query parallelism, by using fewer workers, we can achieve a runtime even better than that in sole intra-query parallelism. For example, as depicted in Figure 3.5, at W=4 and P=32, the runtime is 1.62 seconds. This runtime is about 33% better than the runtime when W is 32 and P is zero (2.11 seconds). In other words, by exploiting prefetching in intra-query parallelism, we cannot only reduce the number of required workers but also achieve a better execution time.

3.4.4 Summary of Conclusions From the Experiments

From the experiments presented in this section we can conclude the following:

1. In SSDs, increasing the I/O queue depth plays an essential role in the I/O cost estimation of database access methods.

2. Increasing the I/O queue depth in SSDs can result in a considerable shift in the selectivity break-even point of database access methods. The query optimizer must be aware of this shift. Otherwise, it would end up choosing plans with much worse execution time than the optimum plan.

[7]The only exception here is when W is 0 and P is 2 which its runtime is better than when W is 2 and P is 0. This exception can be explained as follows. Intra-query parallelism has some extra overhead for worker synchronization. The cost of this extra overhead will discount the benefit of parallelism. When there are only 2 workers, the impact of this extra overhead is enough for seeing a worse runtime in intra-query parallelism compared to prefetching. As we start using more than two workers, the benefit of parallelism increases more, and the impact of the extra overhead will become proportionally lower.

3. Intra-query parallelism and prefetching are two mechanisms that can be employed to generate a higher I/O queue depth in index scan. The combination of these methods can result in a better I/O utilization with a lower negative impact on concurrency.

3.5 Queue Depth Aware Disk Transfer Time Model

In Section 3.4 we observed that the selectivity break-even point faces a significant shift when parallel access methods are employed on the SSD. Now, the question is how we can make the query optimizer aware of the potential benefit of I/O parallelism and its impact on the I/O cost estimation of the query plans.

3.5.1 DTT Model

SAP SQL Anywhere employs an I/O cost model called the disk transfer time (DTT) model to estimate the cost of I/O operations [8, 15, 16]. The DTT model summarizes disk subsystem behavior with respect to an application. The DTT model is a function $DTT(B)$ that takes the band size B as input and returns the amortized cost of reading one page randomly within band size B. Band size is the size of a contiguous disk area (in number of pages[8]) from/to which the random I/Os are going to be issued.

In the DTT model, a band size of 1 represents a sequential I/O pattern. In a DTT model calibrated for a hard disk drive, increasing the band size results in a significant increase in I/O cost. When the band size is larger, it will span more cylinders on the disk. Consequently, it is more likely that for every retrieval, the disk arm needs to be moved from one cylinder to another, resulting in higher overall seek time. By calibrating and using the DTT model, just by knowing the band size in which a database operator is going to perform its I/Os, we can have a fairly accurate estimate of the amortized cost of each individual page I/O.

Figure 3.6 shows DTT models of a consumer-level SSD and HDD. In the HDD, the difference between the minimum and maximum DTT values is about 115 times while on SSD this difference is less than 3 times. Although an SSD has no moving parts, the band size still has an impact on I/O cost. The impact of band size in the SSD depends on its internal architecture. Clustered page size and cache effects in modern SSDs can

[8]The page size is determined at the creation time of a database schema. Most database systems allow different page sizes for different database schemas. The DTT model is calibrated and stored for every created database schema separately.

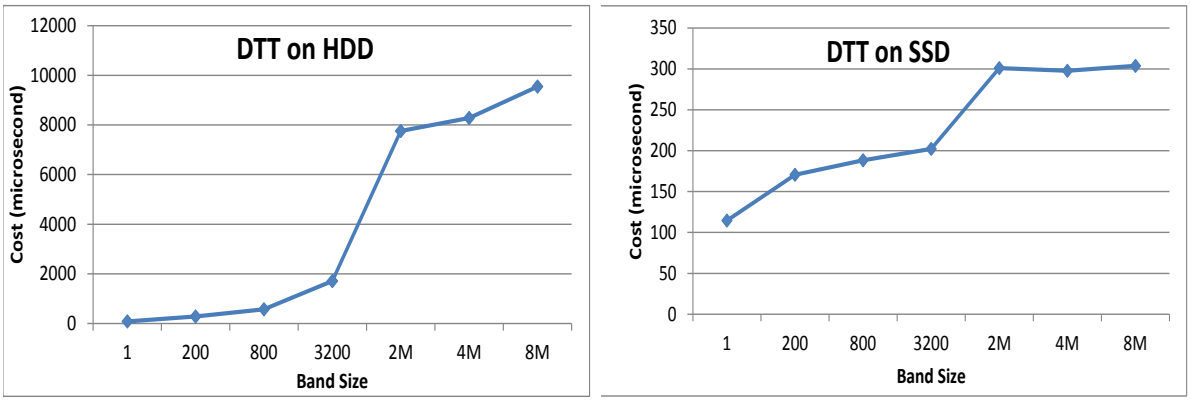


Figure 3.6: A sample DTT model for the HDD and SSD.

contribute to this impact. Suppose the clustered page size of the SSD is 128 Kbytes while the database page size is 4 Kbytes. Then, to read a 4 Kbytes database page a 128 Kbytes cluster is read into the internal buffer pool of the SSD. Thus, if the next requests fall into the same clustered page, the latency of fulfilling those requests will be reduced as they are already in the buffer pool of the SSD. Moreover, some modern SSDs benefit from an internal prefetching mechanism. The smaller the band size is, the higher is the likelihood of the usefulness of the prefetching mechanism in improving the I/O throughput. As the band size becomes bigger, the likelihood of accessing the prefetched pages in the internal buffer pool of the SSD, before they get evicted, will be reduced.

The DTT model can be calibrated easily for any particular hardware at any time. This allows the database optimizer to be adapted to the new hardware and perform more accurately after calibration. This eliminates the trouble of using inaccurate hardcoded parameters which are tuned ahead of time and used for any deployment of the database.

3.5.2 QDTT Model

Although the DTT model works very well for modeling the I/O cost of commodity hard disk drives, it is not accurate enough for modeling the behavior of modern storage devices such as solid state drives. This is because the DTT model does not capture the impact of queue depth. This will result in inaccurate optimization decisions when devices with high parallel I/O capability (like SSDs) are employed.

In order to solve this problem, we have introduced an extension of the DTT model which considers the I/O queue depth as well as band size. The new model is called queue-depth-aware disk transfer time (QDTT) [36, 37]. Unlike the DTT model, the QDTT model is a

function that takes two parameters, band_size and queue_depth, and returns the amortized cost of a random I/O within the given band_size, when the queue depth of the storage device is equal to the queue_depth. For database operators which can benefit from I/O parallelism, this new model provides a much more accurate estimation of the I/O cost. It is clear that this model will be more beneficial when the data are located on an SSD. For hard disk drives, the QDTT model maintains the same functionality as the DTT model. Therefore, the new QDTT model can be considered as a generalization of the DTT model.

Although both DTT and QDTT models are used in a similar way by the query optimizer, the interpretation of the value they return is slightly different. What the DTT model returns is the estimated I/O latency of a single I/O. However, what QDTT model returns is not exactly the I/O latency of an individual I/O. QDTT(band_size, queue_depth) takes into account the parallel I/O capability of the device. Therefore, the returned value by QDTT is smaller than the real I/O latency of each individual I/O. Let's clarify this by an example. Assume the query optimizer knows a database operator will read 100 pages randomly from a file that spans 2000 pages. To compute the time it takes to perform all 200 I/Os using the DTT model, the query optimizer multiplies 200 by $DTT(2000)$. Here it is assumed that the latency of each individual I/O is estimated as $DTT(2000)$. Now assume that the query optimizer knows that the database operator is executed using a technique that increases the I/O queue depth to 32 during the execution (e.g. PIS). In this case, the query optimizer still estimates the total I/O time of all 200 pages by multiplying 200 by $QDTT(2000, 32)$. However, here, $QDTT(2000, 32)$ is smaller than the I/O latency of each individual I/O because when the queue depth is 32, multiple I/Os will be fulfilled in parallel. Note that when k I/Os are fulfilled in parallel in n seconds, then the I/O latency of each I/O will be close to n seconds but the amortized cost of each I/O is n/k. That is why, instead of *I/O latency*, in this chapter, we have used *amortized cost* as a universal term for interpreting the return value of both DTT and QDTT models. For further clarification please see Section 3.5.5.

Figure 3.7 shows a calibrated QDTT model of an SSD drive and an HDD drive[9]. Each curve represents a different queue depth. As it is shown, by increasing the queue depth the amortized cost of one I/O (in microseconds) decreases significantly. This impact is clearly more pronounced in the SSD.

A calibrated QDTT model helps the optimizer to hide the internal complexities of the underlying storage subsystem for I/O cost estimation. The only things the optimizer needs

[9]The QDTT model is a model with a two-dimensional input and a one-dimensional output. A natural method for visualizing such models is using a 3D graph. For the sake of clarity of presentation, we decided to flatten the 3D graph and convert it to a 2D graph in which each curve represents a different queue depth.

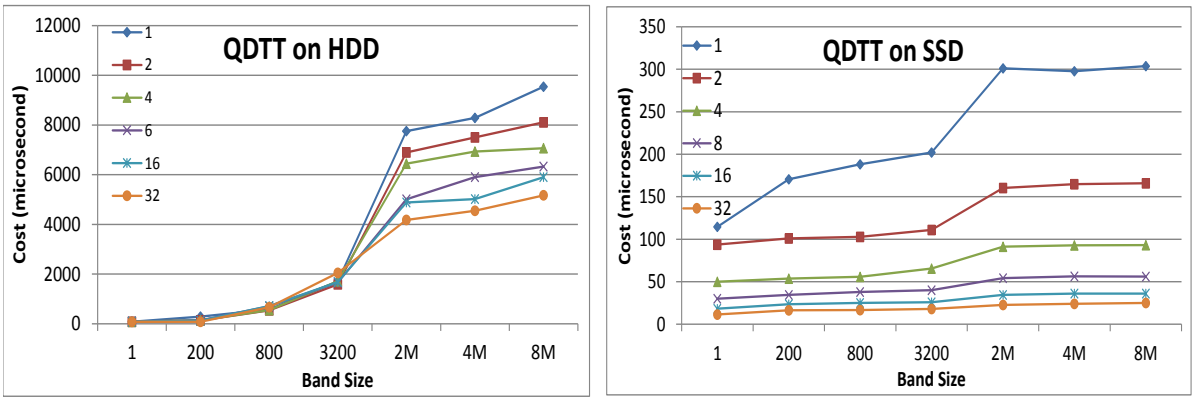


Figure 3.7: A sample QDTT model for the HDD and SSD.

to know are the band size and queue depth. The QDTT model will take care of the rest.

3.5.3 Experimenting with QDTT Model

Here we would like to perform some experiments to see the impact of using the new model in the query optimizer. In the cost estimation function of PIS and PFTS operators, there is a call to the DTT function. The cost estimation function first estimates the band size on disk from which the physical I/O fetches will be issued. This band size will be sent to the DTT model as an input parameter, and the DTT model will return the amortized cost of reading a page randomly within the given band size. Then, the number of pages needed to be retrieved during the scan is estimated. By multiplying this number by the amortized cost of reading a single page, the total I/O cost is calculated. We changed the cost estimation functions of PIS and PFTS so they use QDTT model instead of DTT model. This time, in addition to band size, the expected queue depth of the operator will be passed to the model as well. The calibrated QDTT model must know the expected queue depth, and it will return a more accurate estimated I/O cost. By having a more accurate estimation of the I/O cost, the optimizer will be able to make a better choice between PIS and PFTS operators. When the optimizer is using the DTT model, it may not realize that there might be some benefits (in terms of I/O cost) from executing the operator in parallel. From the perspective of the optimizer, the I/O cost of a parallel and non-parallel access method will be similar. Consequently, the optimizer would prefer a parallel access method over a non-parallel access method only in cases in which the CPU cost benefit of doing things in parallel will surpass the overhead of parallelism. By using the new QDTT model, the optimizer will consider the advantage of I/O parallelism as well.

Figure 3.8 shows the runtime of Query 3.1 before and after using QDTT model for

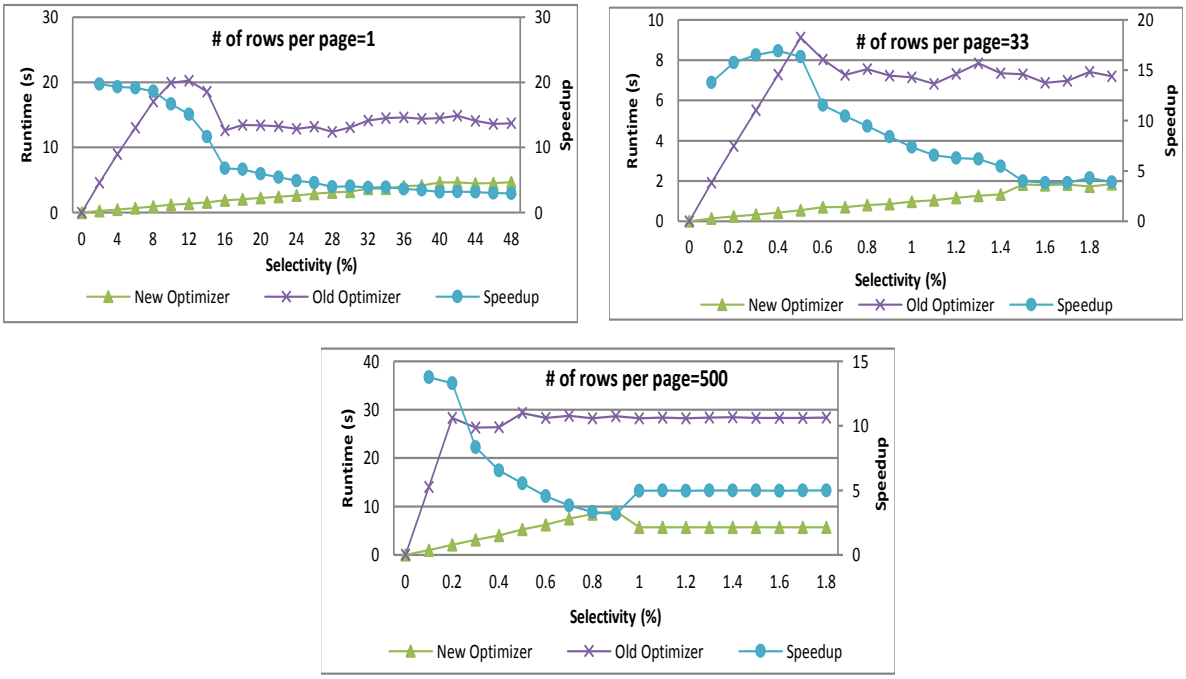


Figure 3.8: Comparing the performance of DTT-based and QDTT-based optimizers

experiments E1-SSD, E33-SSD and E500-SSD. In each diagram, the *old optimizer* curve represents the runtime of the query when the DTT model is used by the optimizer. The *new optimizer* curve represents the runtime of the query when the optimizer employs the QDTT model, and the *speedup* curve indicates how many times the query runtime has improved after utilizing the QDTT model. The y-axis on the right side of each diagram represents the amount of speedup. By using the new model, a significant improvement in query runtime is observable. The maximum speedups in E1-SSD, E33-SSD and E500-SSD are 19.7, 16.9, and 13.7, respectively. The old optimizer uses the DTT model and since it does not realize the benefit of parallel I/O it always prefers a non-parallel method over a parallel one for these experiments. Since the estimated I/O cost is much higher than the estimated CPU cost, the CPU benefit of parallel plans does not have any impact on the decision of the optimizer. Therefore, the optimizer prefers a non-parallel plan. By employing the QDTT model, the optimizer will be aware of the benefit of parallel I/O for the SDD used in these experiments. Therefore, parallel plans will be preferred to non-parallel plans based on the correct cost estimations.

In these experiments, in new optimizer, the parallel degree of PIS is set to the maximum

beneficial queue depth derived from the QDTT model, e.g. 32. The queue depth parameter passed to the QDTT function inside the I/O cost estimation function of the PIS is set to 32 as well. The parallel degree of PFTS is set to the number of available CPU cores, e.g. 8. The queue depth parameter passed to the QDTT function in the I/O cost estimation function of the PFTS is set to 8 as well. The logic behind choosing this number has been described with more details in Section 3.5.4.

In all three experiments, the amount of improvement for low selectivities is high; then it starts to drop, and finally, it becomes constant. When the amount of improvement becomes constant, both the new and old optimizers are choosing the full table scan. However, since the new optimizer chooses the parallel version of the table scan its runtime is consistently better. For a very large range of selectivities, we can observe at least 3 to 5 times improvement while for a small range of selectivities we can achieve up to 20 times improvement.

3.5.4 Application of QDTT Model in Other Operators

As the focus of our research in this chapter is on access path methods, we studied the impact of the QDTT model only on PIS and PFTS operators. However, this model can potentially be used in I/O cost estimation of other database operators as well. Parallel hash join, parallel nested loop join, parallel hash filter, and parallel hash group by are some other operators in SAP SQL Anywhere for which the intra-query parallelism is supported. Application of the QDTT model is not limited only to the parallel operators. This model can be used even for I/O cost estimation of non-parallel operators in which employing prefetching increases the I/O queue depth.

To use the QDTT model for a specific parallel operator, in the I/O cost estimation function of the operator, all invocations of the DTT function should be replaced by those to the QDTT function. Unlike the DTT function, which only takes a band size parameter, the QDTT function takes an additional parameter for queue depth. Now, the question is how to find the proper queue depth that should be passed to the QDTT function. Finding the proper queue depth parameter is something that needs to be investigated for each invocation of the QDTT function in the I/O cost estimation function of the operator[10]. To do so, we need to study the I/O pattern in each phase of the database operator.

[10]Note that there might be more than one invocation of the DTT or QDTT functions in the cost estimation function of a database operator. This happens when the operator consists of multiple phases each with a different I/O pattern.

There are two facts that make finding the proper queue depth for passing to the QDTT function non-trivial. First, the I/O queue depth during the execution of a parallel operator is not necessarily equal to the parallel degree of the operator. A parallel operator may read chunks of data sequentially and then process them in parallel. In this case, during the execution of the parallel operator, the queue depth may never go over one. Second, in some parallel operators, e.g. PIS, while increasing the parallel degree to a number larger than the number of CPU cores has no positive impact on the CPU cost of the operator, it still reduces the I/O cost by increasing the I/O queue depth. In other words, in order to increase the I/O queue depth to the maximum beneficial queue depth supported by the storage device, we might need to increase the number of workers to a number larger than CPU cores[11].

Based on these two facts, to determine the parallel degree and the QDTT queue depth parameters for a given database operator, what we need to do is as follows. First we need to carefully study the I/O patterns in every phase of the operator to figure out the average I/O queue depth that is generated in each phase. Then, we need to adjust the queue depth parameter passed to the QDTT function in each phase accordingly. Second, based on our findings about the I/O patterns of the database operator, we might need to re-adjust the parallel degree of the operator. The parallel degree is typically set to the number of available CPU cores. However, for some database operators that can benefit from an I/O queue depth larger than the number of cores, e.g. PIS, the parallel degree should be set to the maximum beneficial queue depth derived from the QDTT model.

For example, in Section 3.5.3, we determined the parallel degree and the QDTT queue depth parameters for PIS and PFTS as follows. After studying the I/O pattern of PIS, we realized that the I/O pattern of this operator, when executed using a parallel degree equal to P, is a random I/O pattern with an average queue depth very close to P. Thus, in Section 3.5.3, we forced the query optimizer to use the maximum beneficial queue depth derived from the QDTT model, i.e. 32, as the queue depth parameter sent to the QDTT function. Similarly we forced the query optimizer to set the parallel degree of the PIS to this number. Since PFTS is a CPU-bound operator, increasing its parallel degree to a number larger than the number of available CPU cores has no impact on the total execution time of this operator. Thus, we set the parallel degree of PFTS to the number of available cores, e.g. 8. After studying the I/O pattern of the PFTS we realized that during the execution of this operator with parallel degree P, the queue depth is smaller than P. The queue depth in PFTS depends on the number of rows per page. When the number of rows per page is higher, processing each page needs more CPU instructions and it will take

[11] Alternatively, increasing the queue depth might be possible by employing an effective prefetching mechanism similar to the one discussed in Section 3.4.3

longer. In other words, it takes a longer time before the CPU becomes blocked by I/O. Consequently, I/O queue depth will be reduced because the speed of processing data will be less than the speed of reading data. When the number of rows per page is smaller, the fetched pages will be processed faster because each page needs fewer CPU instructions to be processed. Consequently, the CPU will be blocked for I/O more quickly. In this case, the CPU will generate I/O requests more quickly. Thus, the I/O queue depth will be increased. We realized that, in experiments described in 3.5.3, the I/O queue depth during the execution of PFTS varies from 2 to 6, depending on the number of rows per page. Making the queue depth parameter a function of row size is possible. However, this approach adds unnecessary complications to our I/O cost estimation model. Therefore, for the sake of simplicity, we decided to consider the number of available CPU cores, i.e. 8, as the queue depth parameter passed to the QDTT function in I/O cost estimation function of the PFTS. This number is in fact the maximum possible queue depth during the execution of the PFTS operator. Although this simplification reduces the accuracy of the I/O cost estimation, in practice, this inaccuracy is negligible. As we saw in Figure 3.8, the amount of improvement in quality of the decision made by the query optimizer is still very promising. Studying the application and impact of the QDTT model on other database operators is an interesting avenue for future work.

3.5.5 Calibrating QDTT Model

The QDTT model can be calibrated by executing an SQL command. However, since the calibration command is not a standard SQL command, many users might not be familiar with it. Moreover, in a self-tuning database system like SAP SQL Anywhere, which is supposed to work with minimum user administration, it would be very desirable if the model can be calibrated automatically in proper times in which the I/O activity of the system is minimal. To achieve this goal we need to minimize the amount of resources needed for calibration. Moreover, since calibrating the entire model for all possible queue depths is very expensive, we need to minimize the number of calibration points. In this case, we need to employ a proper method to estimate the values associated with the non-calibrated points in the model based on the values of the calibrated points.

In order to calibrate the QDTT model for a band size of b and a queue depth of d we need to measure the amortized cost of a single page I/O, when the I/Os are randomly issued, within a band size of b when the average queue depth is d. The amortized cost of a single page I/O will then be calculated by dividing the total measured I/O time by the number of issued I/Os. One way of increasing the I/O queue depth is to use multiple threads. Each thread issues a synchronous page I/O, and as soon as that synchronous

I/O finished, it issues another synchronous I/O. Since the pages are just read, the total CPU time for processing a page is almost zero; hence the CPU time compared to the I/O latency is negligible. Therefore, by using n threads we can keep the average I/O queue depth constantly equal to n.

In order to minimize the amount of resources needed for calibration, instead of using multiple workers (threads) we achieved the same results by employing asynchronous prefetching using a single worker. To do so, we need n buffers. At first, a group of n asynchronous I/Os are issued by the thread to the buffers 1 to n. Then, the thread waits for the I/O associated with the first buffer to finish. As soon as that I/O was finished, the thread issues another asynchronous I/O into buffer 1 and then immediately waits for the I/O associated with the buffer 2 to finish. As soon as the I/O associated with buffer 2 was finished, the thread issues another asynchronous I/O into buffer 2 and then immediately waits for the I/O associated with the buffer 3. This process continues until the I/O associated with the nth buffer is finished. Then, the thread issues another I/O into buffer n and then waits for associated I/O to buffer 1 to complete. This circular process continues until all pages are read. We validated this method by comparing its resulting calibrated model with that of a multi-worker implementation (over multiple different drives) and noticed very similar models.

3.5.6 Bilinear Interpolation

Calibrating the QDTT model for all queue depths from 1 to 32 will increase the calibration time substantially. This will be a more serious issue when the model is calibrated on an HDD drive. Therefore, we need to reduce the number of calibration points somehow and use an interpolation method to estimate the value of non-calibrated points when needed during the cost estimation.

For the original DTT model, a linear interpolation method is used to calculate the cost associated with band sizes for which there is no calibration point in the computed model. We decided to adopt the same approach to calculate the cost associated with the queue depths for which there is no calibrated point in the QDTT model. Namely, we will first interpolate linearly on the band size and then on the queue depth. This method is also known as *bilinear interpolation*. Suppose the value of $QDTT(b, q)$ is unknown. Suppose q' is the largest queue depth smaller than q for which the value of $QDTT(b, q')$ is known. Similarly, suppose q'' is the smallest queue depth larger than q for which the value of $QDTT(b, q")$ is known. Then, to calculate the value of $QDTT(b, q)$ we use the following formula:

$$QDTT(b,q) = QDTT(b,q') + \frac{QDTT(b,q'') - QDTT(b,q')}{q'' - q'} \times (q - q')$$

Now, the question is, from 1 to 32, which queue depths can be used by the linear interpolation more accurately. We assumed that queue depths 1, 2, 4, 8, 16, and 32 are the best candidates. This is based on the fact that the number of parallel I/O units in storage devices is usually a power of 2. In other words, exponentially increasing the queue depth during the calibration process will result in a reasonably accurate model. In this model, values of important points are more accurate, and a bilinear interpolation is used for calculating values of less-important missing points. To validate our assumption we performed an extensive set of experiments on different drives.

In Figure 3.9 each diagram represents the cost of a random read for a given band size over different queue depths on an 8-spindle RAID-0 array. The red square-shaped points represent the observed I/O times associated with queue depths 1, 2, 4, 8, 16 and 32 and the blue diamond-shaped points represent the observed I/O times associated with other queue depths. For the sake of clarity, we have avoided connecting consecutive red points to each other. It is apparent that almost all blue points fall on (or close to) an imaginary line between consecutive red points. This imaginary line represents the estimated values based on a linear interpolation. In other words, we can conclude that calibrating for points 1, 2, 4, 8, 16, and 32 and employing linear interpolation for other points (blue diamond points in Figure 3.9) is a reasonably accurate approach. This confirms the validity of our assumption. We repeated the same experiments on RAID arrays of different sizes as well as on SSD and observed very similar results. For the sake of brevity, we decided not to include the result of all experiments.

3.5.7 Improving the Calibration Time

Calibrating a QDTT model is slower than calibrating a DTT model. That is because there are many more calibration points in the QDTT model. For devices which cannot benefit from parallel I/O, performing calibration for high queue depths is pointless. If we know for sure that a device cannot benefit even from a queue depth of 2, then there is no point in calculating and maintaining the accurate costs for queue depths larger than 1 because the optimizer will never use the costs associated with queue depths larger than 1 for that device.

In order to reduce the calibration time, we can take advantage of this fact and propose a control mechanism that stops the calibration process when continuing it is no longer beneficial. The mechanism works as follows. The calibration starts from queue depth 1.

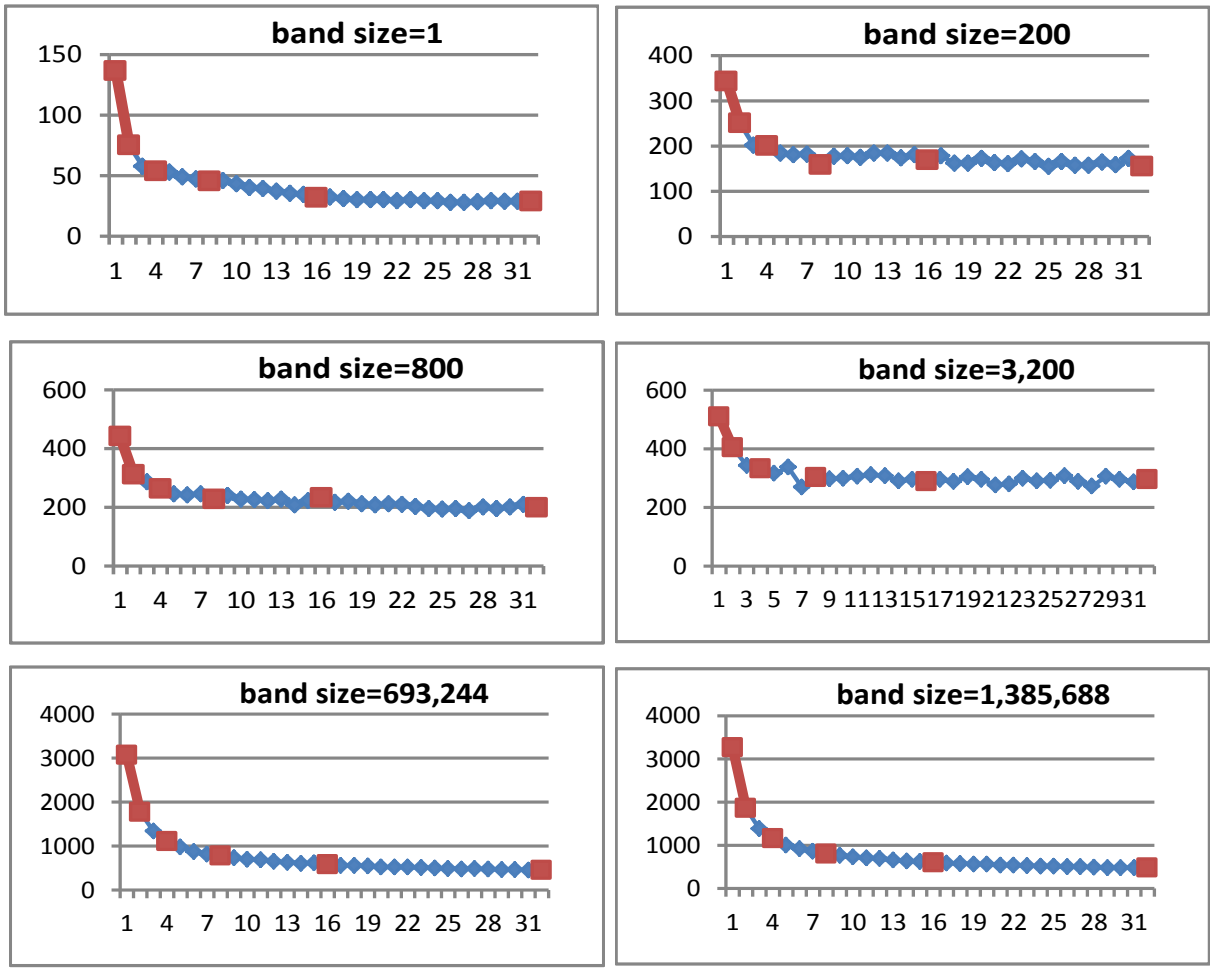


Figure 3.9: QDTT on RAID (8 spindles). The x-axis represents the queue depth and the y-axis represents the cost of reading a single page in microseconds

After calibrating all band sizes for queue depth 1, the queue depth is doubled, and the calibration will be performed for queue depth 2. For each queue depth, the calibration is done from the largest to the smallest band-size. After that the calibration of the largest band size in queue depth 2 is finished, we check the calibration point associated with the largest band size in queue depths 1 and 2. If increasing queue depth has resulted in at least T percent improvement, we will continue the calibration. Otherwise, the calibration will be stopped and a default value slightly larger than the measured costs for queue depth one is assigned to the remaining calibration points. If the calibration did not stop, after calibrating the largest band size in next queue depth we recheck the stop condition. This approach results in a significant improvement in calibration time especially for devices with weak parallel I/O capability. For example, on a single 7200 RPM HDD, the total calibration time with and without employing the stop condition takes 1.09 and 5.63 seconds respectively. This is about 5 times improvement in calibration time.

Another benefit of this approach is that it adjusts the calibration runtime dynamically based on the parallel I/O capability of the device. For example, in a drive in which the maximum beneficial queue depth is 4, calibration stops at queue depth 4. In other words, by employing this approach, the calibration process stops whenever there is no point in continuing the process. We experimentally found that 20% is a reasonable value for T.

Chapter 4

An External Merge Sort for Solid State Drives

4.1 Overview

As we discussed in Chapter 3, the gap between the throughput of the sequential and random I/O is much narrower in SSDs than in HDDs. For decades database operators have been designed based on the I/O characteristics of the HDDs. As HDDs are replaced by SSDs, some assumptions about the I/O characteristics of the storage layer will become outdated. Consequently, the design decisions of database operators need to be reconsidered.

External sort is one of the fundamental operators in database systems. In addition to ordering result sets, it is used as part of many other database operations, such as duplicate removal, uniqueness verification, rank and top, roll-up, cube, merge join, index creation for tables and materialized views, and logical and physical consistency checks [31, 44, 45, 87]. Because of the importance of external sort, it has been studied extensively since before the advent of database systems [54]. External merge sort and external distribution sort are two main types of external sort [40]. Due to the advantages of the external merge sort over external distribution sort, it has been adopted in most database systems [40].

In this chapter, we will propose a new variation of external merge sort called SSD-sort which exploits the I/O characteristics of modern SSDs. We will see in which range of configurations SSD-sort will beat the traditional external merge sort in terms of performance.

4.2 Summary of Contributions

The contributions of this study can be summarized as follows:

1. An SSD-friendly external merge sort that outperforms the traditional external merge sort in terms of time-to-first-row in all configurations, beats the traditional method in terms of total execution time in a range of configurations, and increases the lifespan of the SSD

2. A comparison of the proposed method with the traditional external merge sort showing the range of configurations in which the proposed method presents a superior performance

3. An exploration of the similarity of the last stage of SSD-sort to parallel index scan, and of the possibility of replacing external merge sort with parallel index scan when a relevant index is available

There exists a variety of external merge sort methods in which different optimization techniques and implementation tricks have been applied [10, 42, 55, 59, 60, 71, 72, 75, 76, 83, 93, 97, 96, 98]. The idea behind SSD-sort is orthogonal to those existing optimization methods. In other words, the technique used in SSD-sort is applicable to almost all existing variations of the traditional external merge sort. In this chapter, *traditional external merge sort* refers to any existing variation of the external merge sort in which there are only two phases, *run generation* and *merge*. However, for the sake of simplicity, we compare SSD-sort with a selected existing variation of the external merge sort that we believe is fairly close to optimal but not necessarily the best in all scenarios. We believe that the selected variation serves well as a baseline for our comparisons for two reasons: (1) there is no such thing as the best possible external merge sort because the performance of different variations depends on many parameters such as the distribution of data, data type, variability of the size of the data items, available resources, and the underling processing hardware, and (2) in SSD-sort we propose a technique for exploiting the I/O characteristics of the modern SSDs that is applicable to almost any existing variation of the external merge sort.

4.3 Background

Traditional external merge sort consists of two major phases: *run generation* and *merge*. Given a memory budget of B Mbytes, in the run generation phase, in each step, B Mbytes of data is fetched into memory, sorted, and written back to the disk. This process is repeated until there is no more input data. This method of run generation is called the *load-sort-store* method. In this method, a larger memory budget results in longer sorted runs and fewer of them.

In the basic version of the load-sort-store, after loading data to memory, the I/O is blocked until sorting is finished. Then, computation is blocked until the sorted data is written to disk, and then the next chunk of unsorted data is read into the memory. I/O and computation overlapping can be employed in load-sort-store by dividing the available memory into two buffers. The data is loaded into the first buffer, then at the same time that the first buffer is being sorted, the next chunk of data is loaded into the second buffer. After that sorting the first buffer is finished, sorting the second buffer is started, and at the same time, the sorted data of the first buffer is written into the disk, and then the next chunk of unsorted data is loaded into the second buffer. The same scheme is repeated until all runs are generated. Dividing the memory into two buffers makes it possible to benefit from I/O and computation overlap. However, when the memory is divided into two pieces, the length of generated runs will become shorter, and the number of generated runs will

be doubled. This will potentially result in more merging passes in the merge phase.

The most famous alternative method to load-sort-store which is used for increasing the length of sorted runs during the run generation phase is called *replacement selection* [40, 55, 60]. This method keeps track of the highest key output so far to the current run. It can then determine whether an incoming record can still be made part of the current run or whether it should be deferred to the next run. This algorithm is implemented efficiently using a min heap data structure and in average generates runs twice as large as the size of the memory buffer.

Although replacement selection can generate larger runs, in practice, quicksort-based load-sort-store is a more widely used method of run generation. This is because of its better performance compared to replacement selection. Compared to replacement selection, quicksort benefits from much better CPU cache locality of reference[1] and when it is implemented carefully it is less CPU intensive [72].

In the merge phase, in every pass, a number of sorted runs are merged to form a larger sorted run. The merging process is repeated until a single sorted run is formed. The number of sorted runs that can be merged in every pass depends on the given memory budget. Merging can be done by employing a *tree of losers* data structure [55]. In this method, the data is read randomly and is written sequentially. A memory buffer is assigned to each input run in the tree. The size of this buffer is known as the *cluster size*. In this chapter we will refer to cluster size as C. During the merging process, at first C Mbytes from each participating sorted run is read from disk into the corresponding run buffer. Whenever all data items inside a particular run buffer are consumed, an I/O read request is issued to fetch the next C Mbytes from disk into the buffer.

When the cluster size is small, the data items inside the buffers are consumed quickly, and I/O read requests will frequently be needed. The data items inside each sorted run on disk are stored consecutively. However, the order in which the run buffers are consumed is data-dependent. Therefore, consecutive I/O read requests might be from different sorted runs. This will increase the cost of I/O, especially on HDDs. When different sorted runs are stored on different cylinders (tracks), in order to fulfill each read request, the disk head should move from one cylinder to another. These seek times are expensive and will reduce the I/O throughput.

In order to improve the read I/O throughput during the merge phase, we need to

[1]The poor cache behavior of the replacement selection can be attributed to the fact that the heap usually cannot be fit in CPU cache and therefore the cache thrashes in the bottom levels of the heap. In other words, every single heapification operation leads to multiple cache misses. This results in wasting CPU cycles waiting for the cache misses and increasing the total run generation time.

increase the cluster size. In this way, many small reads will be replaced with fewer large block reads. Consequently, the number of seeks will be reduced significantly and the read I/O throughput will be improved.

When all items inside a run buffer are consumed, the merge process will be blocked until the issued read I/O for next block is finished. *Double buffering* and *forecasting* [55], and their variations [97] are the standard methods which are used to avoid blocking. These methods overlap I/O and computation. In double buffering, two run buffers are allocated for each run. While the first run buffer is being processed, an asynchronous read request will be issued to fetch the next block into the second run buffer. When all items inside the first buffer are consumed, processing the second buffer is started immediately, and another asynchronous read request is issued to fetch the next block into the first run buffer. In the forecasting method, one run buffer is allocated for each run, and one extra run buffer is allocated which is used to prefetch the next block from a run which is expected to need its next block before all other runs. To predict the target run for prefetching, the last data item of all run buffers are examined, and the run buffer with minimum (or maximum) key will be chosen for prefetching. The number of prefetching buffers can be increased to more than one to improve the I/O utilization. Also in some variations of prefetching, a separate list of last block items for each run is computed during the run generation phase. These lists are used during the merge phase to accelerate the prediction process.

Cluster size is a fraction of the given memory budget. Although larger cluster size results in better I/O throughput, when the memory budget is limited, choosing a larger cluster size reduces the number of runs that can be merged in one pass. Consequently, the number of merging passes would potentially be increased, resulting in more I/Os. Thus, the cluster size should be selected carefully.

When HDDs are replaced by SSDs, increasing the cluster size has no longer a significant impact on I/O throughput. That is because the performance gap between the throughput of the sequential and random I/Os in SSDs is much narrower than that in HDDs. Therefore, on SSDs, in general, a smaller cluster size would be more beneficial for the overall performance of the merge phase [63].

The possibility of proposing an SSD-friendly external merge sort has been studied previously. Liu et al. have proposed a flash-friendly external sorting algorithm that avoids non-necessary writes during the run generation phase by detecting the natural page runs [67]. A natural page run is a sequence of pages in which the tuples are already sorted. They map the problem of finding naturally occurring runs into the shortest distance problem in a directed acyclic graph. Since their proposed approach is computationally expensive, they use a heuristic approach to tackle the problem. The method proposed by Liu et al. is

only suitable for partially sorted relations, and it fails to show promising performance as the input data becomes more unsorted. The idea behind SSD-sort is orthogonal to that in this study. In other words, the project and fetch phases of SSD-sort can be applied to the method proposed by Liu et al as well.

A number of other previous studies have proposed flash-friendly external sorting algorithms for mobile databases and embedded systems such as sensor networks [11, 23, 77]. In these environments, usually, the main memory is very limited and flash chips are used as a raw storage layer in which the write performance is 10 to 100 times worse than the read performance. Therefore, these methods try to either eliminate writes completely or reduce the number of required writes at the cost of a significant increase in the number of reads. For example, Park and Shim propose an external sorting algorithm called FAST(1) that generates a single sorted run [77]. There is no merge step in this algorithm. This algorithm reads the entire relation multiple times. Each time it finds the K smallest elements using an in-memory heap and writes a sorted run into the disk at the end of the pass. The number of passes over data is equal to the size of the relation divided by the size of the available memory. Park and Shim propose an extended version of FAST(1) called FAST. In the first phase of FAST, the relation is divided into Q partitions, and each partition is sorted using the FAST(1) algorithm. In the second phase, the sorted runs generated in the first phase are merged to form a single sorted run. The performance gap between writes and reads in SSDs is significantly narrower than that in raw flash chips. Thus, these flash-chip-optimized methods can not beat the traditional external merge sort on SSDs. Unlike the simple flash memories which are used in embedded systems, modern SSDs employ an internal controller, a parallel architecture, an internal cache, as well as mechanisms like garbage collection and wear leveling to improve the performance of both reads and writes, and to reduce the performance gap between the two. Thus, the sorting algorithms which are designed for embedded systems are not suitable for DBMSs with modern SSDs.

Graefe et al. discuss the required modifications in external merge sort in a systems in which the SSD is used as a supplementary storage for storing the temporary generated runs [42]. The adjustments needed in buffer sizing in external merge sort and the optimal order and schemes for merging the generated runs in scenarios in which either the entire generated runs fit in the SSD, or only a portion of it fits there are discussed. Although SSD-sort has been primarily designed for a system in which the entire database is stored on the SSD, it can be used in a system configuration in which the SSD is used as a supplementary device as well. In that scenario the advice given by Graefe et al. for adjusting the buffer sizes as well as the suggested schemes for merge ordering are applicable to SSD-sort as well. In other words, SSD-sort is orthogonal to the ideas presented in this study.

FMsort is maybe the closest SSD-friendly external merge sorting approach to SSD-sort

[62]. The main idea behind FMsort, though different, is in line with the main idea behind SSD-sort. Like SSD-sort, FMSort tries to exploit the parallel I/O capability of the SSDs. However, unlike SSD-sort, in FMsort the parallel I/O is exploited only in the merge phase. In FMsort the sequence of page numbers which need to be fetched during the merge phase is precomputed during the run generation phase. This sequence is computed by keeping track of the value of the minimum key in every block of each sorted run and comparing the smallest values from the runs which are participating in a merging pass. The pages which are supposed to be needed in the near future are prefetched using asynchronous I/Os. By issuing multiple asynchronous requests at the same time, the queue depth of the SSD would be increased and the I/O throughput would be improved. The idea proposed in FMsort is orthogonal to that in SSD-sort. In other words, both methods can be applied at the same time to improve the performance further. To be fair, in our implementation of both SSD-sort and the traditional method we used this technique to improve the I/O throughput during the merge phase.

4.4 SSD-sort: an SSD-Friendly External Merge Sort

In this section, a variation of external merge sort called *SSD-sort* is proposed. It outperforms the traditional external merge sort in terms of time-to-first-row in all configurations. The proposed approach is also capable of beating the traditional method in terms of total execution time in a range of configurations. Moreover, replacing the traditional method with SSD-sort will increase the lifespan of the SSD.

The proposed approach is capable of outperforming the traditional method by generating longer runs using the same amount of memory. SSD-sort can improve the lifespan of the SSD by reducing the number of required writes. These benefits, however, will come at the cost of extra random read requests which will be compensated for by exploiting the parallel I/O capability of the SSD.

Figures 4.1 and 4.2 show the data flow of the traditional external merge sort and SSD-sort, respectively. Unlike the traditional external merge sort, which has only *run generation* and *merge* phases, SSD-sort includes two additional phases: *project* and *fetch*. The project phase happens before the run generation phase and the fetch phase happens after the merge phase.

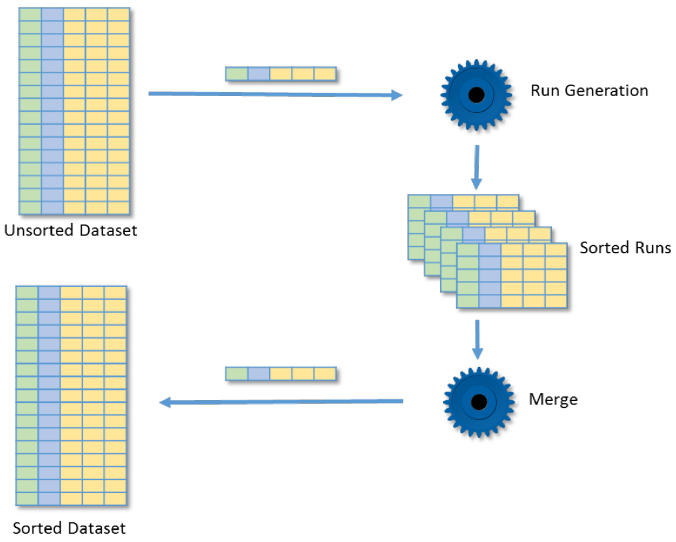


Figure 4.1: Traditional external merge sort

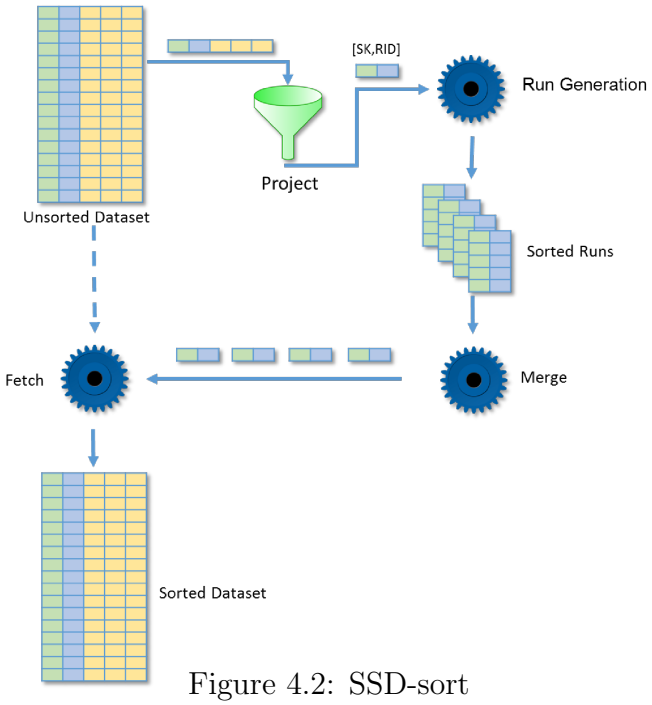


Figure 4.2: SSD-sort

4.4.1 Project Phase

The input of the project phase is a stream of rows, and its output is a stream of rows projected over the sorting key (SK) and the row id (RID). In other words, for each input row, a pair $[SK, RID]$ is extracted and streamed into the run generation phase. The sorting key can be a simple key or a compound key consisting of multiple fields. A row id is a compound number that consists of the *page number* and the *offset* (or number) of the row within the page. RID is a pointer to the physical location of the row on disk. The complete rows can be fetched from disk later on using their RIDs. The project phase can be overlapped perfectly with the sequential I/Os issued for reading the input data. It can also be perfectly pipelined with the run generation phase. Thus, the impact of this extra phase in overall execution time of the sorting operator is very minimal.

4.4.2 Run Generation Phase

During the run generation phase, the memory is filled with only the $[SK, RID]$ pairs. Load-sort-store and replacement selection methods both will work exactly the same way in SSD-sort as they do in the traditional method, except that in SSD-sort the generated runs consist of only $[SK, RID]$ pairs. Since the complete rows are not kept in memory, the extra available room can be used to maintain more items. Therefore, the generated runs are expected to be longer. This benefit will be more pronounced when the row size is larger. In addition, since only the $[SK, RID]$ pairs are written into the disk, the amount of data written to the disk will be smaller.

4.4.3 Merge Phase

In SSD-sort's merge phase the generated sorted runs are merged the same way they are merged in the traditional method. However, since the length of runs is expected to be larger, than in the traditional method, the number of runs needed to be merged will be smaller. This will reduce the number of merging passes. Moreover, since the total size of the generated runs is smaller, the amount of data required to be read and written in each merging pass will be smaller.

4.4.4 Fetch Phase

The output of the merge phase in SSD-sort will be a single sorted run consisting of $[SK, RID]$ pairs. While the last sorted run is being generated, the $[SK, RID]$ pairs will be streamed into the next phase, which is the *Fetch* phase. During the fetch phase, the row corresponding to each $[SK, RID]$ pair is fetched from disk using the RID, and is written into the final sorted result set or streamed into the next operator in the query execution pipeline. To fetch a row, the page containing that row must be read from disk. During the fetch phase, the memory buffer pool of the database engine will play an important role. If a page is already in the buffer pool, there is no need for a physical I/O request from the SSD. The fetch phase and merge phase can be perfectly pipelined. Therefore, the first row in the sorted dataset will be available almost immediately after the merge phase generates it.

Although the last phase of SSD-sort requires many random reads from the SSD, random I/O is faster on SSDs than on HDDs and many such requests can be served in parallel. Our experiments show that the speed of generating the sorted output stream of the merge phase is higher than the speed of the random I/O fetches in fetch phase[2]. In other words, assuming all pages are located in the SSD, the speed of revealing the page numbers which are going to be fetched in the near future is more than the speed of fetching those pages. This observation can be employed to exploit the parallel I/O capability of the SSD. Suppose the 4 Kbytes read IOPS of our SSD when the queue depth is 1 is 4,000 and the rate of emitting the $[SK, RID]$ pairs by the merge phase is 132,000 pairs per second. Suppose fetching each page will take about 250 microseconds. By the time the first page which contains the first row is fetched, we will know the page number of the next 32 pages. Thus, instead of issuing another single synchronous I/O we can issue 32 simultaneous asynchronous I/Os. By doing so, the queue depth of the SSD will be increased to 32. This will increase the read IOPS of the SSD to about 40,000. By employing this method, the I/O bandwidth of the fetch phase will be increased about 14 times. This will potentially compensate for the high cost of the random I/Os during the fetch phase. The idea of asynchronous prefetching in fetch phase is the key idea in designing SSD-sort. It can significantly improve the I/O throughput and execution time of the fetch phase. Lee et al. have employed a very similar prefetching idea but in merge phase of the traditional

[2]This observation happens when the size of the memory buffer pool is 75% or smaller than the size of the input data. The efficient I/O and computation overlapping in merge phase of SSD-sort makes the merge phase in most cases CPU-bound. The fetch phase is slower than the merge phase because the fetch phase is I/O-bound. In an extreme case in which the size of memory buffer pool is zero, for every $[SK, RID]$ pair generated in fetch phase, an actual physical random I/O will be required. This will make the fetch phase extremely I/O-bound.

method [62]. In our implementation of SSD-sort, which is described in next section, we have exploited parallel I/O in both merge and fetch phases of SSD-sort.

4.4.5 SSD-sort vs. the Traditional External Merge Sort

One important advantage of SSD-sort over the traditional approach is its better *time-to-first-row* or *response time*. Time-to-first-row is defined as the time it takes from the beginning of the sorting process to the point that the first row in the sorted result set is returned. Even when the total execution time of SSD-sort in a particular setting is more than that of the traditional approach, its time-to-first-row might be much shorter. In external merge sort, the first-sorted row is generated immediately after the last merging step in the merge phase starts. The number of sorted runs in SSD-sort is normally smaller than that in the traditional method. Besides, the amount of data that needs to be read and written during the merge phase of SSD-sort is typically less than that in the traditional method. Thus, the merge phase in SSD-sort is typically done faster. Consequently, the last merging step in SSD-sort starts earlier than that in the traditional method. That is why the time-to-first-row in SSD-sort is better than that in the traditional method.

The last step of the merge phase is usually the most expensive step in SSD-sort because, in this step, the corresponding row to each emitted $[SK, RID]$ pair should be fetched. Fetching rows that do not reside in memory will generate a costly random I/O pattern. However, as soon as this step starts, the generated rows can be either used by the next operator in a query execution pipeline or returned to the client application. In interactive applications and use cases in which time-to-first-row is more important than the total execution time, SSD-sort can potentially show a considerable advantage.

Another advantage of SSD-sort over the traditional approach is its superior total execution time in a range of configurations. In particular, it is more likely that SSD-sort shows a superior performance when the sorting memory is smaller, the memory buffer pool is larger, and the row size is bigger.

To understand why SSD-sort is more likely to outperform the traditional method when the sorting memory is smaller, we need to analyze the run generation and merge phases in the external merge sort. Unlike the traditional approach, SSD-sort uses $[SK, RID]$ pairs rather than the complete rows. Thus, compared to the traditional approach, SSD-sort can generate longer and fewer runs with the same amount of sorting memory. This will reduce the number of merge passes, and the amount of data that need to be read and written during the merge phase of SSD-sort. This results in a faster merge phase. Thus, when the sorting memory is smaller, SSD-sort is more likely to outperform the traditional approach.

Note that this does not mean that reducing the sorting memory improves the performance of SSD-sort. Rather, it says that, compared to the traditional method, decreasing the sorting memory has a lower negative impact on the performance of SSD-sort. In other words, in a configuration in which the execution time of both methods is equal, if we decrease the sorting memory, it is more likely that SSD-sort outperforms the traditional method.

Even in database systems in which a large amount of memory is employed, being able to sort with a limited amount of memory can decrease the overall execution time of the query. This is because, in a complex query, a larger share of the memory budget can be assigned to the more memory-hungry operators such as hash join. Moreover, processing the same query with the same performance but smaller memory footprint allows the database engine to process more queries concurrently.

To understand the impact of the row size and memory buffer pool size on the relative performance of SSD-sort, we need to analyze the fetch phase in SSD-sort. Suppose the size of the memory buffer pool is zero and the number of rows in the input data is R. Also, suppose that the row size is so big that each page contains only one big row. In this case, the number of physical page reads during the fetch phase of SSD-sort is equal to R. It means that each page in the input data is physically read only once. In other words, the input data is scanned only once. Now, assume the memory buffer pool size is still zero, but the row size is so small that each page contains K rows. Then, in the worst-case scenario, the number of physical page read I/Os issued to fetch all rows is equal to $R \times K$. In other words, each page is read K times. This is similar to scanning the entire input data K times, using a random I/O pattern. Therefore, size of the rows in the input data has a significant impact on the performance of the fetch phase of SSD-sort. The traditional method has no fetch phase. Therefore, its performance does not depend on row-size.

Now, assume the size of the memory buffer pool is greater than zero, and the number of rows per page is K. In this case, during the fetch phase of SSD-sort, each page that is read can be cached in the buffer pool. The cached pages can be reused to fulfill the upcoming row requests, avoiding the unnecessary physical page reads. In this case, the number of physical page reads during the fetch phase of SSD-sort is smaller than $R \times K$. The extent to which the number of physically read pages is smaller than $R \times K$ depends on the buffer pool size. A bigger buffer pool has a higher hit rate and can prevent more of the physical reads. Thus, the memory buffer pool size has a direct impact on the performance of the fetch phase. In other words, a large buffer pool can discount the negative effect of a small row size. Again, since the traditional method does not have a fetch phase, its performance is independent of the memory buffer pool size.

In conclusion, SSD-sort is more likely to outperform the traditional method when the sorting memory is smaller and when the rows and the memory buffer pool are bigger.

Finally, another benefit of SSD-sort over the traditional method is that SSD-sort requires fewer writes in the run generation and merge phases when more than one merge pass is required. As we discussed before, fewer writes can be translated to a more extended lifetime for SSDs. Even in configurations in which SSD-sort and the traditional method show very comparable total execution times, this advantage can be a tiebreaker.

4.5 Experimental Results

In this section, we present an experimental evaluation of SSD-sort. Our main objective is to compare the performance of SSD-sort with that of the traditional approach. In particular, we are interested to know in which ranges of configurations SSD-sort shows superior performance over the traditional method. The query optimizer of a database system can benefit from the answer to this question. By identifying the range of configurations in which SSD-sort shows better performance than the traditional method, the query optimizer can decide which sorting operator to choose in a given configuration. A *sorting configuration* refers to a combination of the following parameters: (1) row size, (2) size of the available sorting memory, (3) size of the memory buffer pool relative to the size of the input, and (4) type (or I/O capabilities) of the storage device. In the experiments presented in this section, we will investigate how changing the first three parameters will impact the total execution time and the time-to-first row of SSD-sort compared to those in the traditional method. As SSD-sort is an SSD-specific sorting method, we have included only the experiments we performed on SSD in this section. Thus, the 4th parameter is fixed in all experiments.

In order to have full control over the experiments, we have implemented a standalone version of SSD-sort. This standalone version is implemented using the STXXL package [12, 25, 26]. STXXL is a C++ standard template library for large datasets. It is, in fact, an extension of the C++ standard template library, STL. STXXL is designed for external memory computations. This library provides the basic functionality needed in external memory algorithms, e.g. streaming, pipelining, and overlapping of I/O and computation.

We compared SSD-sort with the default implementation of the traditional external merge sort which is available in STXXL package. This implementation is highly optimized and employs I/O and computation overlapping. It also uses the forecasting method (see Section 4.3) during the merge phase for predicting the next required blocks. To the best

of our knowledge, this implementation is among the best general purpose open-source implementations of the external merge sort in terms of performance.

Table 4.1 summarizes the parameters used in the implementation of the traditional method and SSD-sort in our experiments. The default implementation of the traditional method in STXXL uses a cluster size of 2 Mbytes. It is known that the optimal cluster size has steadily increased over the past decade, as the gap between latency and bandwidth has become wider [41, 46]. Because of the superior I/O latency of SSDs over HDDs, on SSDs we can choose a much smaller cluster size [63]. By choosing a smaller cluster size, using the same amount of memory, more runs can be merged in each merging pass. This will potentially reduce the number of merging passes. Consequently, the number of required I/Os will be reduced and the execution time would be improved. We experimentally determined that on SSD, a cluster size of 32 Kbytes is a reasonable choice in most cases in both SSD-sort and the traditional method[3]. Therefore, we decided to use a cluster size of 32 Kbytes in both SSD-sort and the traditional method.

For the fetch phase of SSD-sort we chose a page size of 4 Kbytes as it is a common choice for page manager of most database systems. 4 Kbytes is the smallest physical unit of I/O in many existing modern SSDs. This parameter is irrelevant in the traditional method as it does not have a fetch phase and it does not utilize the memory buffer pool.

In the run generation phase of both the SSD-sort and the traditional method, an improved version of the load-sort-store method is used in which the memory is divided into two buffers and I/O and computation are overlapped to improve the I/O utilization.

In the merge phase of both methods, the I/O read throughput is optimized using a forecasting method similar to the one used in FMSort [62]. In the forecasting method, during the run generation phase, for each run, a sequence of the last key of each block is computed. During the merge phase, these computed sequences are used to predict the next required block. This is done using a tournament tree over the computed sequences of the runs which are participating in the merge pass. The predicted next blocks are prefetched into a number of assist blocks in order to eliminate or reduce I/O blocking and to improve the I/O utilization. In FMSort the parallel I/O capability of the SSD is exploited during the merge phase by issuing multiple prefetch requests at the same time. Prefetching degree during the merge phase refers to the number of assist blocks used for

[3]To find the optimal cluster size we started with 2 Mbytes and reduced the cluster size gradually and measured the execution time of a multi-pass merge phase with every cluster size. We observed an increase in execution time after we passed 32 Kbytes. In SSD-sort, row size has no impact on optimal cluster size, but in the traditional method, when the row size is very large (e.g. 4096) a cluster size of 64 Kbytes shows slightly better results. However, considering different row sizes, on average, a 32 Kbytes cluster size shows better results.

Table 4.1: Parameters used in implementation of SSD-sort and the traditional method

Parameter name	Traditional Method	SSD-sort
Cluster Size	32 Kbytes	32 Kbytes
Run Generation Method	load-sort-store (2 buffers)	load-sort-store (2 buffers)
Page size	Irrelevant	4 Kbytes
Merge phase optimization	forecasting (8 assist blocks)	forecasting (8 assist blocks)

prefetching the predicted next pages. We experimentally determined that employing a prefetching degree larger than 8 does not result in any tangible improvement. Thus, we used 8 assist blocks in both methods.

4.5.1 Experimental Setup

All experiments have been performed on a server running Windows Server 2008 R2 operating system with 24 Gbytes of RAM and two Intel Xeon E5620 2.40 GHz. The amount of RAM used in the experiments is much less than 24 Gbytes, and it is explicitly indicated in each experiment. A 10K RPM Seagate Cheetah NS.2 SAS hard drive is used for the operating system. A 240 Gbytes OCZ Vector, which is a consumer-grade SATA III MLC SSD is used for our experiments. The input, output, and temporary data are all stored on the SSD. The maximum sequential I/O throughput of the SSD is about 270 Mbytes/sec and the maximum random I/O throughput of 4 Kbytes pages (both read and write) in the SSD is about 180 Mbytes/sec (about 46,000 IOPS). Each experiment has been repeated 5 times and the reported execution times are averages of the observed execution times. In every experiment, the maximum absolute deviation of the observed values is within 5% of the reported mean.

As mentioned before, the following parameters will impact the performance of SSD-sort:

- Row size
- Memory buffer pool size
- Sorting memory size
- I/O performance

To study the impact of row size, we have considered 4 different tables with four different row sizes; 16 bytes, 256 bytes, 2048 bytes, and 4096 bytes. For the sake of simplicity, in

Table 4.2: Experimental setup

Table Name	Row Size	Columns	Rows	Table Size
T16	16	256	134,217,728	2GB
T256	256	16	8,388,608	2GB
T2048	2048	2	1,048,576	2GB
T4096	4096	1	524,288	2GB

our experiments, each column is a 4-byte integer, and we adjust the size of the row by changing the number of columns. The first column in each table is used as the sorting key. The values generated for the sorting key follow a uniform random distribution[4]. The values of the other columns are not important in our experiments, and they are used only to increase the size of the row. We filled those columns with random numbers generated using a pseudo-random number generator. For row sizes 16 bytes, 256 bytes, 2048 bytes, and 4096 bytes, the number of columns per row would be 256, 16, 2 and 1, respectively.

If we choose a different row size for each table while keeping the number of rows in all tables equal, then we will end up having a very small table when the row size is 16B, and a very large table when the row size is 4096 bytes. If we compare the sorting time of the tables with significantly different sizes, the one which is smaller will fit better in the cache and its sorting time will be much lower than that of a larger table. To increase the fairness of our comparisons and to eliminate the impact of the table-size-related parameters such as caching, we decided to equalize the size of all tables. To do so, we added more rows to the tables with smaller row size. Table 4.2 summarizes our dataset configuration. The size of all tables T16, T256, T2048 and T4096 is 2 Gbytes. T16 is the tallest and thinnest table, and T4096 is the shortest and fattest one.

To study the impact of the memory buffer pool size, we have repeated the experiments by employing memory buffer pools that are 25%, 50%, 75%, and 100% of the input size.

To investigate the impact of sorting memory size, we have repeated the experiments using 1 Mbytes, 8 Mbytes, and 64 Mbytes of sorting memory. At first look, compared to the large amount of memory which is used in today's database servers, these three sorting memory sizes might seem to be relatively small and unrealistic. However, in practice, this is a typical range of sorting memory size which is used in many database systems. For instance, the default sorting memory used in Postgress SQL, MySQL, and Oracle database are 4 Mbytes [5], 256 Kbytes [3], and 64 Kbytes [4], respectively. The default size of the

[4]To generate a random sequence of n keys we start with a sorted array of keys in which key[i]=i (for i=1 to n). Then for each k (k = 1 to n-1), using a uniform pseudo-random number generator, we generate an integer r between k and n, and we swap key[k] with key[r].

sorting memory in database systems is typically much smaller than the total available memory for the database for three main reasons. First, allocating a major part of the available system memory for a shared memory buffer pool is in general more beneficial than allocating that memory for the working memory of individual queries. Since the memory buffer pool is shared among all queries, the data fetched into it by a query is likely to reduce the I/O cost of other existing or upcoming queries as well. Second, the remaining part of the system memory after deducting the memory buffer pool size is used for multiple purposes such as the working memory of the queries running concurrently, query optimization, plan caching, session management, networking, database cursors, etc. Granting a smaller working memory to individual queries, allows more queries to be executed concurrently. Third, in complex queries containing multiple operators, allocating a larger portion of the working memory of the query to more memory-hungry operators, e.g. hash join, improves both performance and robustness of the query execution.

Since SSD-sort is an SSD-specific sorting method, we have only presented the results of the experiments performed on the SSD.

4.5.2 SSD-sort vs. the Traditional Method

In this section, we will present the results of different sets of experiments with the goal of identifying the range of configurations in which SSD-sort can outperform the traditional method.

4.5.2.1 Identifying the Impact of Memory Buffer Pool Size and Row Size

Figure 4.3 shows the results of comparing the execution time of SSD-sort with that of the traditional method when the sorting memory is fixed to 8 Mbytes. In each graph, the execution time of SSD-sort and the traditional method is shown using a separate bar. In each bar, the blue part represents the time taken from the beginning of the sort to a point in which the first row in the result set is ready to be streamed out (time-to-first-row). The orange part represents the time needed for fetching the entire sorted result set. For SSD-sort the orange part is, in fact, the execution time of the fetch phase. Each row in the figure presents a different cache size, and each column represents a different row size. The columns from left to right correspond to tables T16, T256, T2048, and T4096 (see Table 4.2). By looking at the total execution times of SSD-sort in any row in the figure from left to right, it can be observed that the total execution time of SSD-sort improves as the row size increases. By looking at each column from top to bottom, it can be observed

that the cache size has a considerable positive impact on the total execution time of SSD-sort. A larger cache can absorb a larger number of random reads during the fetch phase of SSD-sort.

No matter how large the cache or row size, the time to first-row of SSD-sort is better than that of the traditional method. The cache size has no impact on the time-to-first row of SSD-sort because the cache is utilized only during the fetch phase. The cache also has no impact on the traditional method as it is not utilized by any phases in the traditional method.

In terms of total execution time, SSD-sort outperforms the traditional method in 10 out of 16 configurations presented in the figure. When the row size is very small (e.g. 16 bytes or 256 bytes) the execution time of the fetch phase of SSD-sort becomes prohibitively large. To improve the fetch time of SSD-sort in this case, a very large cache, i.e. close to 100% of the input size, is required. Even a cache as large as 75% of the input size cannot help SSD-sort to beat the traditional method when the row size is small. In contrast, when the row size is large (e.g. 2048 bytes or 4096 bytes), no matter how large the cache, SSD-sort outperforms the traditional method. In this case, a larger cache will increase the gap between the total execution time of SSD-sort and that of the traditional method.

4.5.2.2 Identifying the Impact of Sorting Memory Size and Row Size

Figure 4.4 shows the results of comparing the execution time of SSD-sort with that of the traditional method when the memory buffer pool size (cache size) is fixed to 50% of the input size. Each row in the figure presents a different sorting memory size. Similar to Figure 4.3, each column represents a different row size. Similar to our observation from Figure 4.3, by looking at the total execution times of SSD-sort in any row in Figure 4.4, from left to right, it can be observed that the total execution time of SSD-sort improves as the row size increases. By looking at any column from top to bottom, it can be observed that increasing the sorting memory has a very limited impact on the total execution time of SSD-sort. However, a larger sorting memory clearly improves the performance of the traditional method. This observation is in line with our discussion in Section 4.4.5 regarding the suitability of SSD-sort when the sorting memory is limited. In other words, when the sorting memory is large, SSD-sort is not able to outperform the traditional method in terms of the total execution time unless the row size and cache size are very large as well.

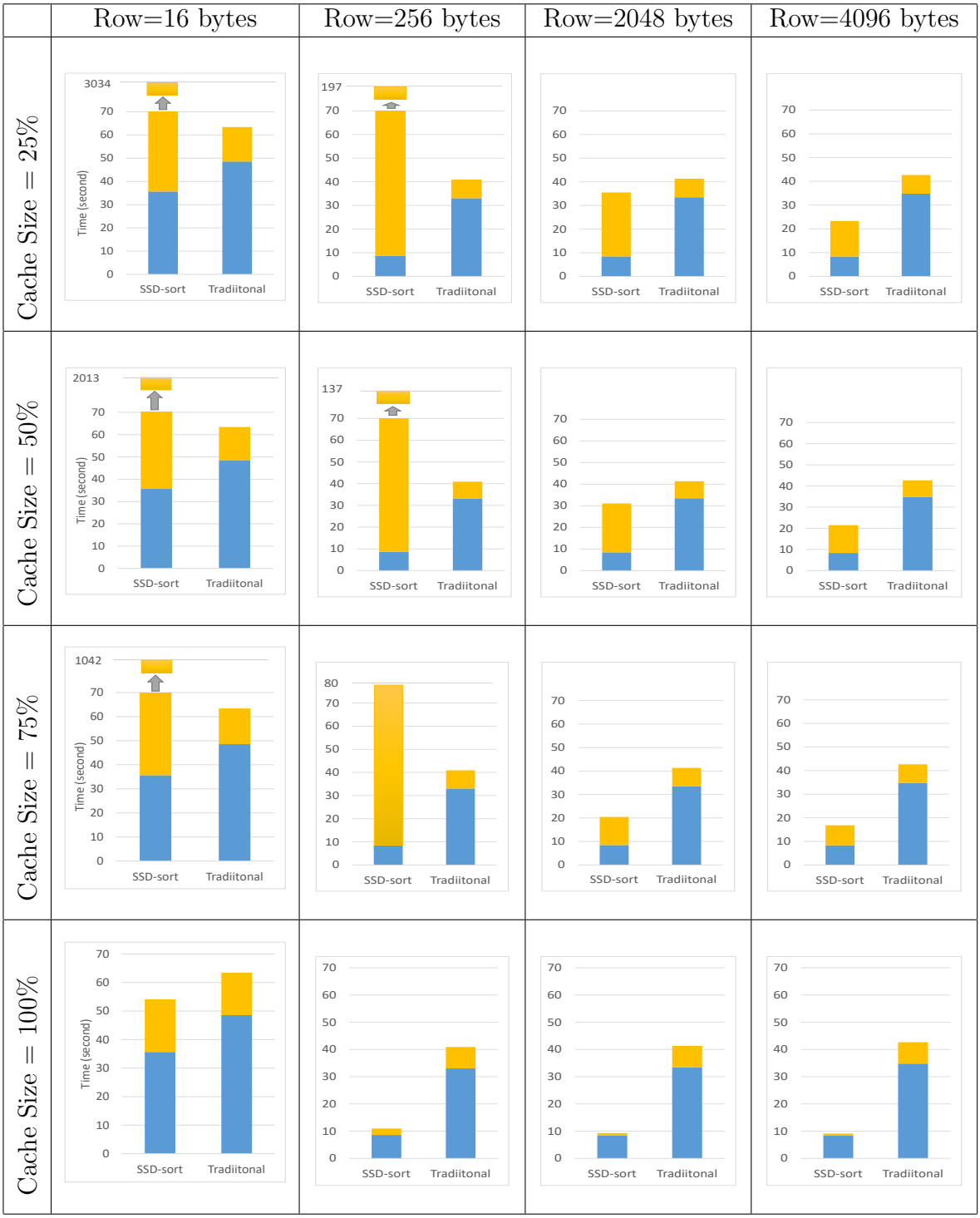


Figure 4.3: The execution time of SSD-sort vs. that of the traditional method when the sorting memory is 8 Mbytes. In each graph, the blue color represents the time-to-first-row, and the orange color represents the fetch time.

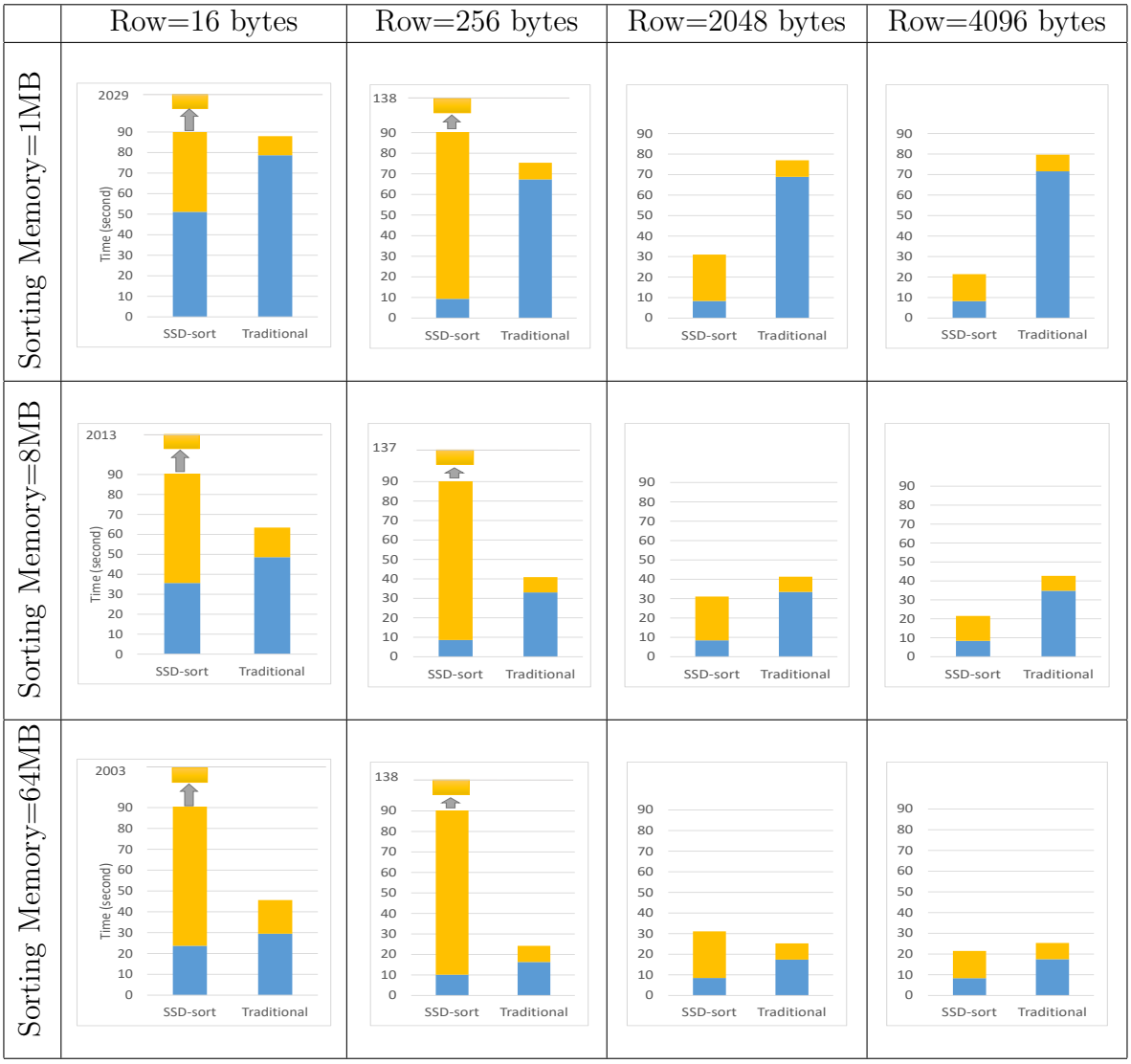


Figure 4.4: The execution time of SSD-sort vs. that of the traditional method when the memory buffer pool size is 50% of the input size. In each graph, the blue color represents the time-to-first-row, and the orange color represents the fetch time.

4.5.2.3 Identifying the Impact of All Parameters in One Picture

To have a larger picture about the range of configurations in which SSD-sort outperforms the traditional method, we measured the execution time of both methods over all possible combinations of the three parameters: row size, sorting memory, and cache size. We have summarized the total execution time and time-to-first-row speed-ups/slow-downs of SSD-sort over the traditional method in Tables 4.5 and 4.6, respectively. In these tables, speed-ups and slow-downs are highlighted by colors green and red, respectively. A number larger than 1 indicates a speed-up, and a number smaller than 1 represents a slow-down.

As expected, in terms of total execution time, SSD-sort tends to outperform the traditional method where the sorting memory becomes smaller, and/or when the cache size and row size become larger. In terms of total execution time, SSD-sort shows a superior performance in 28 out of 48 configurations. One interesting observation from Table 4.5 is that when the cache size is 100%, no matter how large the sorting memory or the row size, SSD-sort outperforms the traditional method. In contrast, when the row size is smaller than or equal to 256 bytes, unless the cache is 100%, in all other configurations the traditional method wins. Similarly, When the sorting memory is 64MB, unless the cache is 100%, in all other configurations the traditional method either wins or looses with a relatively small margin. Note that in all these experiments, the permutation of soring keys in the original table follows a uniform random distribution. This permutation is the worst-case scenario for SSD-sort because in this case, the locality of reference among consecutive sorting keys is in its lowest possible. As we will see in Section 4.5.2.4, increasing the locality of reference will improve the hit rate of the cache, resulting in a better total execution time for SSD-sort.

Table 4.6 shows that in terms of time-to-first-row, SSD-sort dominantly outperforms the traditional method in all configurations. This is a very promising observation that confirms the dominant applicability of SSD-sort in use cases in which time-to-first-row is more important than the total execution time.

4.5.2.4 Impact of Pre-sortedness on SSD-sort

In all experiments presented in Section 4.5.2, we considered a worst-case scenario in which the initial ordering of the rows is uniformly random. In practice, in many use cases, there is usually some sort of *locality of reference* in the initial ordering of the data. The locality of reference refers to the fact that two consecutive rows in the sorted result set are close to each other in the original table as well. This property results in a much better cache utilization during the fetch phase of SSD-sort. A better cache utilization will reduce the

Sorting Memory Size (Mbytes)		Row Size (Bytes)	Cache Size (Percentage of Input Data)			
			25%	50%	75%	100%
	1	16	0.03X	0.04X	0.08X	1.27X
		256	0.38X	0.55X	0.96X	6.42X
		2048	2.17X	2.49X	3.97X	8.39X
		4096	3.43X	3.72X	4.77X	8.86X
	8	16	0.02X	0.03X	0.06X	1.17X
		256	0.21X	0.30X	0.53X	3.73X
		2048	1.16X	1.33X	2.02X	4.45X
		4096	1.83X	1.98X	2.54X	4.69X
	64	16	0.02X	0.02X	0.04X	1.05X
		256	0.12X	0.17X	0.31X	1.95X
		2048	0.71X	0.81X	1.23X	2.72X
		4096	1.09X	1.18X	1.51X	2.80X

Figure 4.5: Total-time speed-ups/slow-downs of SSD-sort over the traditional method

Sorting Memory Size (Mbytes)		Row Size (Bytes)	Cache Size (Percentage of Input Data)			
			25%	50%	75%	100%
	1	16	1.54X	1.54X	1.54X	1.54X
		256	7.20X	7.20X	7.20X	7.20X
		2048	8.24X	8.24X	8.24X	8.24X
		4096	8.66X	8.66X	8.66X	8.66X
	8	16	1.36X	1.36X	1.36X	1.36X
		256	3.84X	3.84X	3.84X	3.84X
		2048	3.95X	3.95X	3.95X	3.95X
		4096	4.16X	4.16X	4.16X	4.16X
	64	16	1.24X	1.24X	1.24X	1.24X
		256	1.61X	1.61X	1.61X	1.61X
		2048	2.05X	2.05X	2.05X	2.05X
		4096	2.10X	2.10X	2.10X	2.10X

Figure 4.6: Time-to-first-row speed-ups/slow-downs of SSD-sort over the traditional method

number of required physical reads during the fetch phase. The degree of locality of reference is higher when the data is partially sorted. When the input data is partially sorted, even with a very small cache size, we might end up having very few physical reads during the fetch phase.

In order to see the impact of the locality of reference on the performance of the fetch phase of SSD-sort, we have performed a new set of experiments in which the input table is already sorted. These experiments represent an extreme case in which the locality of reference is maximum. Since locality of reference is maximum, during the fetch phase of SSD-sort, each page which is read will be reused immediately for multiple consecutive future row fetches. For instance, when the row size is 16 bytes, after fetching a page, it can be used to fulfill 256 consecutive row requests because all those rows physically reside on the same page.

In order to reduce the amount of required physical reads in the fetch phase to the size of the original table, only a very small cache is sufficient. To confirm this, we have used a 2 Mbyte cache (0.1% of the input size). To consider the impact of sorting memory size, we have repeated the experiment with 1 Mbyte, 8 Mbytes and 64 Mbytes of sorting memory.

For the sake of brevity we have presented only the speed-up/slow-down values in Table 4.3. When the sorting memory is 1 Mbyte or 8 Mbytes, no matter how large the row size, SSD-sort beats the traditional method. When the sorting memory is 64 Mbytes, in row sizes 256B, 2048B and 4096B SSD-sort still beats the traditional method. The only case in which the traditional method shows a superior performance is when sorting memory is 64 Mbytes, and row size is 16B. Even in this case, the traditional method is only 1.35X times faster than SSD-sort. Remember that when data was not sorted and the sorting memory was 64 Mbytes, even with a 512 Mbytes cache (25% of the input size) the traditional method was 33.3X better than SSD-sort (see Table 4.5). By comparing Table 4.3 with the first column in Table 4.5 (the cache=25% column), it can be seen that in spite of having a significantly larger cache(0.1% vs. 25%), when the data is pre-sorted, the speed-up values are considerably better.

This set of experiments confirms the significant impact of the locality of reference on the performance of SSD-sort.

The promising results we observed in Table 4.3 are not limited to a completely pre-sorted table. Even if we sort a partially sorted table in which any pair of unordered rows in the original table are not too far from each other, we can expect to see very similar results. As the distance of out-of-order pairs in our original table becomes longer, we can use a larger cache in fetch phase to catch up. In general, as long as the maximum distance between two out-of-order elements is within the boundaries of our cache we can expect

Sorting Memory Size (Mbytes)	Row Size (Bytes)	Cache Size = 2MB
1	16	1.48X
	256	3.76X
	2048	4.13X
	4096	4.27X
8	16	1.02X
	256	2.14X
	2048	2.25X
	4096	2.29X
64	16	0.74X
	256	1.21X
	2048	1.33X
	4096	1.40X

Table 4.3: Summary of total-execution-time speed-ups/slow-downs of SSD-sort over traditional method when the original table is already sorted

very similar results.

In practice, there are many use cases in which we need to sort a partially sorted table. One of the possible use-cases is the periodic rebuild of a clustered index which has become partially unsorted after multiple inserts, deletes, and updates. Another use case is sorting partially sorted server logs in data centers. In data centers log data is collected from many servers and brought together either immediately or periodically and stored in a central log store. The central partially sorted log is then typically sorted by timestamp before doing temporal analysis over the log data.

4.5.2.5 Impact of SSD-sort on Improving the Lifespan of SSDs

As mentioned earlier, SSD-sort can potentially improve the lifespan of SSDs by reducing the number of required writes during the sorting process. In order to realize how many writes can be saved by using SSD-sort instead of the traditional method, we measured the number of writes in both methods and divided the write volume of the traditional method by that of SSD-sort and summarized the results in Table 4.4. Changing row size will change the volume of writes only in SSD-sort. That is because, unlike the traditional

Table 4.4: Required write volume in SSD-sort compared to that in the traditional method

Table Name	Sorting Mem=1MB	Sorting Mem=8MB	Sorting Mem=64MB
Row Size=16	2.5X	2X	2X
Row Size=256	53.33X	48X	21.33X
Row size=2048	640X	256X	256X
Row size=4096	1280X	768X	512X

method, SSD-sort has a project phase. Thus, it can benefit from more saving on the number of required writes when the row size is larger. Sorting memory can have an impact on the write volume of both methods. Employing a larger sorting memory reduces the number of generated runs, and consequently, the number of merge passes in merge phase. Fewer merge passes will need fewer writes. For both methods, cache size has no impact on the write volume as the cache size only affects the I/O throughput during the fetch phase of SSD-sort which contains only random reads. As indicated in Table 4.4, when the sorting memory is 1MB, and the row size is 4096 bytes, SSD-sort will need 1280X fewer writes compared to the traditional method.

4.6 Similarity between SSD-sort and Parallel Index Scan

In Chapter 3, we studied the performance of parallel index scan (PIS) on SSDs[5]. Suppose an index IX_{c1} has been created over column C1 in table T which is stored on the SSD. Consider an SQL query in the form of "SELECT C1, C2, C3 FROM T ORDER BY C1". Since IX_{c1} does not cover columns C2 and C3, the leaf nodes in IX_{c1} are not enough to answer the above query. For answering the above query, there exist two possible execution plans. The natural choice is to perform a complete external merge sort. An alternative plan is to perform a PIS using IX_{c1}. It is believed traditionally that the cost of an index scan is much higher than that of an external sort. This belief is based on the traditional assumption about the large performance gap between the throughput of the random and sequential I/O in HDDs. When the database is stored on an SSD, based on the observations we made from our experiments, we can argue that, for processing an ORDER BY operator, a PIS is not necessarily more expensive than a full external merge sort.

[5]Here, by index we mean nonclustered index.

An index is similar to a pre-computed output of the merge phase in SSD-sort. In other words, in SSD-sort we build an index and then perform a full PIS using it; but the built index is not materialized on disk. Instead, it is used on-the-fly, as it is created, to feed the fetch phase. Thus, the execution time of performing a PIS would be very close to that of the fetch phase in SSD-sort. Therefore, for processing the ORDER BY operator mentioned above, a full PIS scan should be cheaper than a complete external merge sort which is performed using SSD-sort.

One question is whether or not a PIS is cheaper than a traditional external merge sort as well. As expected, the answer to this question depends on the row size, sorting memory size, cache size, and the degree of pre-sortedness of the table T. To find out in which configurations the PIS outperforms traditional merge sort, it is enough to compare the execution time of the fetch phase of SSD-sort with total execution time of the traditional method. Table 4.5 and Table 4.6 show the results of these comparisons when the data distribution is uniformly random, and when the data is already sorted, respectively. These tables show that there are cases in which PIS is two orders of magnitudes better than the traditional external merge sort. For example, when the row size is 4096, the sorting memory is 1 Mbyte, and the cache size is 100%, then PIS is 109.12X better. When the row size is 2048 or bigger, in all cases except one, PIS shows a superior execution time. Also, when the cache is 100%, no matter how big the sorting memory or row size, PIS is the absolute winner.

From the results of this experiment, we can conclude that when the table is stored on the SSD, the traditional belief about the superior performance of the traditional external merge sort over index scan is not accurate, and it might result in choosing execution plans with significantly higher execution times.

Note that Table 4.5 represents a worst-case scenario in which the initial distribution of rows is uniformly random. As mentioned before, in many cases, there exists some degree of locality of reference in the original table. This locality of reference results in better cache utilization during the fetch phase of SSD-sort. Similarly, it will result in a better cache utilization during PIS.

Table 4.6 represents a fully sorted table. In practice, a fully sorted table is equivalent to a clustered index[6]. When a clustered index exists, it would be the natural choice for processing the ORDER BY statement mentioned above. However, we present Table 4.6 just to see the extreme impact of the maximum locality of reference on PIS. When the

[6]To be more accurate, a sequential scan of a fully sorted table is equivalent to the sequential scan of the leaf nodes in a clustered index.

Sorting Memory Size (Mbytes)		Row Size (Bytes)	Cache Size (Percentage of Input Data)			
			25%	50%	75%	100%
	1	16	0.03X	0.04X	0.09X	4.82X
		256	0.40X	0.59X	1.09X	31.41X
		2048	2.85X	3.40X	6.43X	93.99X
		4096	5.33X	6.06X	9.43X	109.12X
	8	16	0.02X	0.03X	0.06X	3.44X
		256	0.22X	0.32X	0.59X	17.31X
		2048	1.53X	1.83X	3.45X	50.47X
		4096	2.86X	3.24X	5.05X	58.44X
	64	16	0.02X	0.02X	0.05X	2.32X
		256	0.13X	0.19X	0.35X	10.53X
		2048	0.93X	1.12X	2.11X	30.83X
		4096	1.70X	1.93X	3.00X	34.72X

Table 4.5: Speed-ups/slow-downs of nonclustered index scan over traditional external merge sort when the data is distributed uniformly random

data is pre-sorted, PIS outperforms external merge sort, regardless of the size of the cache, the sorting memory, and the row size.

Sorting Memory Size (Mbytes)		Row Size (Bytes)	Cache Size = 2MB
	1	16	3.76X
		256	6.81X
		2048	7.70X
		4096	7.75X
	8	16	2.04X
		256	3.78X
		2048	4.20X
		4096	4.16X
	64	16	1.19X
		256	2.18X
		2048	2.49X
		4096	2.54X

Table 4.6: Speed-ups/Slow-downs of nonclustered index scan over traditional external merge sort when the data is already sorted

Chapter 5

Conclusion, Discussion, and Future Work

5.1 Conclusion

This thesis has introduced systems and methods for exploiting the capabilities of modern SSDs to improve the performance of database systems.

In Chapter 2, we showed that SSDs can be used effectively as a fully persistent second level buffer pool in relational database systems to improve the transactional throughput, checkpoint time, and recovery time, and to avoid a long ramp-up time after a crash recovery. We proposed PC, a novel method for exploiting the SSD as a fully persistent second level cache. We discussed our design decisions and the logic behind them in detail and showed empirically how the effective use of the persistence of the SSD can improve the performance. Compared to LC, PC shows more than 3X improvement in transactional throughput. Employing PC can enhance the recovery rate of the database by an order of magnitude. Moreover, after a crash recovery, the SSD cache in PC is immediately warm. In contrast, LC suffers from a prolonged ramp-up time.

In Chapter 3, we showed that when the entire database is stored on the SSD, we need to make the query optimizer SSD-aware because failing to do so will result in sub-optimal query optimizer decisions. We characterized the impact of I/O parallelism in database scan operators on SSDs and HDDs. We showed empirically how query optimization in the SSD is affected when I/O parallelism is employed. Our experiments confirmed that, contrary to popular belief, an SSD-oblivious query optimizer can fail to choose the optimal access method, with a big margin of error, when parallel I/O is employed in access methods. We proposed a novel, general and dynamic I/O cost model called QDTT, for accurate I/O cost estimation of parallelizable database operators. We explained how QDTT can be efficiently calibrated and employed by the query optimizer. The QDTT model allows the optimizer to accurately choose among execution alternatives on a range of storage devices. Our experiments demonstrate that an SSD-aware query optimizer can choose plans with up to 20X better execution time, compared to its SSD-oblivious counterpart. We also demonstrate how intra-query parallelism and prefetching and their combination can be leveraged to exploit the I/O parallelism in SSDs to avoid the excessive use of valuable workers.

In Chapter 4, we showed that when the entire database is stored on the SSD, we need to redesign some of the major database operators so that they can make better use of the capabilities of SSDs because failing to do so will result in performance degradation of the operator. We proposed a variation of external merge sort, called SSD-sort, which is designed to exploit the I/O capabilities of SSDs to outperform the traditional method in a range of configurations. SSD-sort outperforms the traditional method in terms of

time-to-first-row in all configurations, achieves a better total execution time in a range of configurations, and reduces the number of required writes. In terms of total execution time, the superiority of SSD-sort becomes more evident when the sorting memory is limited, the row size and memory buffer pool size are larger, and there is a higher degree of pre-sortedness or locality of reference in the input data. SSD-sort reduces the number of required merge passes in the merge phase by generating larger and fewer runs. It also exploits the parallel I/O capability of the modern SSDs to compensate for the cost of the additional random writes during its fetch phase. Our experimental evaluations showed that time-to-first-row of SSD-sort can be up to 8.86X, and its total execution time can be up to 8.66X better than those in the traditional method. We also empirically showed that SSD-sort may require orders of magnitude fewer writes compared to the traditional method. We also experimentally showed that the traditional belief about the superiority of the external merge sort over parallel index scan (when a relevant index is available) does not hold when the table resides on an SSD. A PIS can perform up to about two orders of magnitude better than a traditional external merge sort in extreme cases. This suggests an important possible improvement in the query optimizer of the database systems, where the query optimizer tries to optimize an ORDER BY operator over an SSD-resident table, and there exists a relevant index.

5.2 Discussion and Future Work

The PC method was originally proposed and patented in 2012. Since then, several related techniques have been proposed. An analytical comparison of the different performance aspects of PC with those in the newer approaches is one interesting topic for future work.

Another exciting path for further research is to investigate the impact of having an additional layer of buffer pool on query optimization. The second level cache introduces new challenges for a cost-based query optimizer. When a multi-layer caching mechanism is employed, the cost model must be aware of the portion of each database object which is located in each caching layer. Otherwise, the estimated costs will become inaccurate. In addition, some database operators, e.g. full table scan, which are sensitive to sequential I/O patterns might become affected by this additional caching layer, mainly because there might be a fresher SSD-resident version of some pages which are sequentially read from the HDD. Those pages need to be re-fetched from the SSD using a random access pattern. Those re-fetch requests may potentially degrade the performance of full table scans.

The proposed QDTT model is based on the assumption that queries will be executed in isolation. This is a simplifying assumption that makes the problem tractable. A similar

assumption is used in the design of the DTT model. The accuracy of the I/O cost estimations based on the QDTT model will be reduced when multiple queries are being executed concurrently. For instance, suppose the maximum beneficial queue depth of a specific device is 32. Assume for every index scan operator which is done on this device the QDTT model will suggest a queue depth of 32. Suppose 4 queries are running concurrently on the system. In this case, the queue depth of the system will become 128. Increasing the queue depth to a number above the maximum beneficial queue depth will result in excessive I/O latencies. This will increase the response time of queries. Besides, if we use prefetching to increase the queue depth, the memory buffer pool might be polluted by many prefetched pages which may be evicted before they can even have a chance to be used by a scan operator. This reduces the utilization of the memory buffer pool, especially in cases in which the size of the memory buffer pool is small. A queue depth governor is needed to address this problem. The governor must share the maximum beneficial queue depth among different queries based on a specific controlling protocol. This protocol must maximize the queue depth utilization of the storage while preventing the queue depth from overloading. The controlling protocol must be employed both at the optimization time and execution time. Since the state of the system might change from optimization time to execution time, the controlling mechanism must adjust itself at execution time. Designing and implementing a queue depth governor and its related controlling protocol and measuring its impact on overall performance of the concurrent workloads is an interesting direction that we leave it to future work.

In this thesis, we considered the optimizer's decision about IS, FTS, PIS and PFTS operators. Investigating the behavior of more complex database operators and more complex queries is another interesting topic for further research.

One of the advantages of SSD-sort over the traditional method is that it is more CPU cache efficient. This is because items which are moved or replaced in memory in SSD-sort are smaller than those in the traditional method. Therefore, the workload will expose a higher level of locality of reference. This results in fewer cache misses. This argument is confirmed by Nyberg et al. as well [72]. Studying the impact of this locality of reference on the overall execution time of SSD-sort is another opportunity for future work.

Since SSD-sort works with smaller data items during the run generation phase, using co-processors such as GPUs in SSD-sort will be potentially easier and more effective. It is known that the main bottleneck of using co-processors in database operators is the expensive data transfer between the RAM and the co-processor's memory [17, 35]. By reducing the size of the input data from the entire rows to $[SK, RID]$ pairs, the data transfer cost would be reduced. Extending and evaluating a GPU-enabled version of SSD-sort is another interesting path for further research.

We showed that SSD-sort can outperform the traditional method over a range of configurations. The analysis we presented will give the query optimizer some general clues about its decision between SSD-sort and the traditional method. However, to make a more accurate decision, the query optimizer needs a cost model. The cost model should estimate the execution time based on parameters such as the row size, the sorting memory size, the memory buffer pool size, the degree of pre-sortedness in the data, the parallel I/O capability of the storage device, and even the processing power of CPUs. Designing such a complex cost model and evaluating its effectiveness on improving the query optimizer's decision is another promising avenue for further research.

Implementing a parallel version of SSD-sort and comparing its performance with a parallel version of the traditional method is another area that is worthy of further study. Since modern SSDs benefit substantially from the parallel I/O, a parallel SSD-sort might potentially show a superior performance over a parallel traditional merge sort, in a wider range of configurations. In the run generation phase of the parallel external sort, the available sorting memory is divided between the different threads. Since the negative impact of reducing the sorting memory on SSD-sort is lower than that in the traditional method, it is likely that SSD-sort shows a better parallel execution performance.

References

[1] Bcache.
https://web.archive.org/web/20180416191619/https://bcache.evilpiepirate.org/. [Online; accessed 20-Mar-2018].

[2] DM-cache.
https://web.archive.org/web/20160202123356/http://visa.cs.fiu.edu:80/tiki/dm-cache. [Online; accessed 02-Feb-2016].

[3] MySQL Server System Variables.
https://web.archive.org/web/20180427204232/https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html. [Online; accessed 27-APR-2018].

[4] Oracle 12.2 Database Reference - Sort AREA SIZE .
https://web.archive.org/web/20180427204903/https://docs.oracle.com/en/database/oracle/oracle-database/12.2/refrn/SORT_AREA_SIZE.html. [Online; accessed 27-APR-2018].

[5] PostgreSQL 9.6.8 Resource Consumption.
https://web.archive.org/web/20180309050143/https://www.postgresql.org/docs/9.6/static/runtime-config-resource.html. [Online; accessed 09-MAR-2018].

[6] TPC-C: Standard specification, revision 5.11.
https://web.archive.org/web/20180112192336/http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf. [Online; accessed 12-Jan-2018].

[7] Understanding the checkpoint log in SQL Anywhere 12.
https://web.archive.org/web/20170602145853/http://dcx.sybase.com:

80/1200/en/dbadmin/da-backup-dbs-5657414.html. [Online; accessed 02-Jun-2017].

[8] Mohammed Abouzour, Ivan T. Bowman, Peter Bumbulis, David DeHaan, Anil K. Goel, Anisoara Nica, G. N. Paulley, and John Smirnios. Database self-management: Taming the monster. *IEEE Data Eng. Bull.*, 34(4):3–11, 2011.

[9] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[10] Nicolas Anciaux, Luc Bouganim, and Philippe Pucheral. Memory requirements for query execution in highly constrained devices. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 694–705. VLDB Endowment, 2003.

[11] Panayiotis Andreou, Orestis Spanos, Demetrios Zeinalipour-Yazti, George Samaras, and Panos K Chrysanthis. Fsort: external sorting on flash-based sensor devices. In *Proceedings of the Sixth International Workshop on Data Management for Sensor Networks*, page 10. ACM, 2009.

[12] Andreas Beckmann, Timo Bingmann, Roman Dementiev, Peter Sanders, Johannes Singler, Raoul Steffen, and Markus Westphal. STXXL: Standard template library for extra large data sets. https://web.archive.org/web/20180307040151/http://stxxl.org/. [Online; accessed 07-March-2018].

[13] Bishwaranjan Bhattacharjee, Kenneth A. Ross, Christian Lang, George A. Mihaila, and Mohammad Banikazemi. Enhancing recovery using an SSD buffer pool extension. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 10–16, New York, NY, USA, 2011. ACM.

[14] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *ACM SIGOPS Operating Systems Review*, 41(2):88–93, 2007.

[15] Ivan T. Bowman, Peter Bumbulis, Dan Farrar, Anil K. Goel, Brendan Lucier, Anisoara Nica, G. N. Paulley, John Smirnios, and Matthew Young-Lai. SQL Anywhere: A holistic approach to database self-management. In *ICDE Workshops*, pages 414–423, 2007.

[16] Ivan T. Bowman, Peter Bumbulis, Dan Farrar, Anil K. Goel, Brendan Lucier, Anisoara Nica, G. N. Paulley, John Smirnios, and Matthew Young-Lai. SQL Anywhere: An embeddable DBMS. *IEEE Data Eng. Bull.*, 30(3):29–36, 2007.

[17] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. *GPU-accelerated database systems: survey and open challenges*, pages 1–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[18] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. An object placement advisor for DB2 using solid state storage. *Proc. VLDB Endow.*, 2(2):1318–1329, August 2009.

[19] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3(1-2):1435–1446, September 2010.

[20] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, pages 34–43, New York, NY, USA, 1998. ACM.

[21] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 266–277. IEEE, 2011.

[22] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. System software for flash memory: a survey. In *International Conference on Embedded and Ubiquitous Computing*, pages 394–404. Springer, 2006.

[23] Tyler Andrew Cossentine. *An efficient external sorting algorithm for flash memory embedded devices*. PhD thesis, University of British Columbia, 2012.

[24] Brian Dees. Native command queuing-advanced performance in desktop storage. *Potentials, IEEE*, 24(4):4–7, 2005.

[25] R. Dementiev and L. Kettner. STXXL: Standard template library for XXL data sets. In *In: Proc. of ESA 2005. Volume 3669 of LNCS*, pages 640–651. Springer, 2005.

[26] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard template library for XXL data sets. *Softw. Pract. Exper.*, 38(6):589–637, May 2008.

[27] Peter Desnoyers. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.

[28] David J. DeWitt, Jaeyoung Do, Jignesh M. Patel, and Donghui Zhang. Fast peak-to-peak behavior with SSD buffer pool. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1129–1140, Washington, DC, USA, 2013. IEEE Computer Society.

[29] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[30] Jaeyoung Do, Donghui Zhang, Jignesh M Patel, David J DeWitt, Jeffrey F Naughton, and Alan Halverson. Turbocharging DBMS buffer pool using SSDs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1113–1124. ACM, 2011.

[31] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison Wesley, March 2006.

[32] Franz Faerber, Alfons Kemper, Per ke Larson, Justin Levandoski, Thomas Neumann, and Andrew Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[33] Peter Gassner, Guy M. Lohman, K. Bernhard Schiefer, and Yun Wang. Query optimization in the IBM DB2 family. *IEEE Data Eng. Bull.*, 16(4):4–18, 1993.

[34] Johannes Gehrke and Raghu Ramakrishnan. Database management systems. *New York*, 2003.

[35] Pedram Ghodsnia. An in-GPU-memory column-oriented database for processing analytical workloads. In *The VLDB PhD Workshop. VLDB Endowment*, pages 54–59, 2012.

[36] Pedram Ghodsnia and Ivan T. Bowman. Estimation of query input/output (I/O) cost in database, March 14 2017. US Patent 9,594,781.

[37] Pedram Ghodsnia, Ivan T. Bowman, and Anisoara Nica. Parallel I/O aware query optimization. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of data*, Snowbird, UT, USA, 2014.

[38] Pedram Ghodsnia, Reza Sherkat, John Smirnios, Peter Bumbulis, and Anil Goel. Solid state drives as a persistent cache for database systems, September 13 2016. US Patent 9,442,858.

[39] Goetz Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120–135, 1994.

[40] Goetz Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3), 2006.

[41] Goetz Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proceedings of the 3rd International Workshop on Data Management on New Hardware*, DaMoN '07, pages 6:1–6:9, New York, NY, USA, 2007. ACM.

[42] Goetz Graefe. Sorting in a memory hierarchy with flash memory. *Datenbank-Spektrum*, 11(2):83–90, 2011.

[43] Goetz Graefe and Harumi A. Kuno. Definition, detection, and recovery of single-page failures, a fourth class of database failures. *PVLDB*, 5(7):646–655, 2012.

[44] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh. Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 152–159, Feb 1996.

[45] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[46] Jim Gray and Prashant J. Shenoy. Rules of thumb in data engineering. In *ICDE*, pages 3–10, 2000.

[47] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 10. ACM, 2009.

[48] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash-based extended cache for higher throughput and faster recovery. *Proceedings of the VLDB Endowment*, 5(11):1615–1626, 2012.

[49] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash as cache extension for online transactional workloads. *The VLDB Journal*, pages 1–22, 2015.

[50] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 195–206, Washington, DC, USA, 2011. IEEE Computer Society.

[51] Taeho Kgil and Trevor Mudge. FlashCache: a NAND flash memory file cache for low power web servers. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 103–112. ACM, 2006.

[52] Hyojun Kim, Ioannis Koltsidas, Nikolas Ioannou, Sangeetha Seshadri, Paul Muench, Clement L Dickey, and Lawrence Chiu. How could a flash cache degrade database performance rather than improve it? lessons to be learnt from multi-tiered storage. In *INFLOW*, 2014.

[53] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[54] DE Knuth. Vol. 3: Sorting and searching. *Addison-Wesley series in computer science*, 1973.

[55] DE Knuth. Art of computer programming, volume 3: Sorting and searching (2n, 1998.

[56] Ioannis Koltsidas and Stratis D Viglas. Flashing up the storage layer. *Proceedings of the VLDB Endowment*, 1(1):514–525, 2008.

[57] Ioannis Koltsidas and StratisD. Viglas. Designing a flash-aware two-level cache. In *Advances in Databases and Information Systems*, volume 6909 of *Lecture Notes in Computer Science*, pages 153–169. Springer Berlin Heidelberg, 2011.

[58] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle TimesTen: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[59] Per-Åke Larson. External sorting: Run formation revisited. *IEEE Trans. on Knowl. and Data Eng.*, 15(4):961–972, July 2003.

[60] Per-Åke Larson and Goetz Graefe. Memory management during run generation in external sorting. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 472–483, New York, NY, USA, 1998. ACM.

[61] Eun-Mi Lee, Sang-Won Lee, and Sangwon Park. Optimizing index scans on flash memory SSDs. *ACM SIGMOD Record*, 40(4):5–10, 2012.

[62] J. Lee, H. Roh, and S. Park. External mergesort for flash-based solid state drives. *Computers, IEEE Transactions on*, PP(99):1–1, 2015.

[63] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086. ACM, 2008.

[64] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[65] Xin Liu and Kenneth Salem. Hybrid storage management for database systems. *Proceedings of the VLDB Endowment*, 6(8):541–552, 2013.

[66] Xin Liu and Kenneth Salem. Integrating SSD caching into database systems. *IEEE Data Eng. Bull.*, 37(2):35–43, 2014.

[67] Yang Liu, Zhen He, Yi-Ping Phoebe Chen, and Thi Nguyen. External sorting on flash memory via natural page run generation. *Comput. J.*, 54(11):1882–1990, November 2011.

[68] Dongzhe Ma, Jianhua Feng, and Guoliang Li. Lazyftl: a page-level flash translation layer optimized for NAND flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1–12. ACM, 2011.

[69] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[70] J. Najajreh and F. Khamayseh. Contemporary improvements of in-memory databases: A survey. In *2017 8th International Conference on Information Technology (ICIT)*, pages 559–567, May 2017.

[71] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. AlphaSort: A RISC machine sort. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 233–242, New York, NY, USA, 1994. ACM.

[72] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and Dave Lomet. Alphasort: A cache-sensitive parallel external sort. *The VLDB Journal*, 4(4):603–628, October 1995.

[73] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2):297–306, 1993.

[74] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware storage layout for database systems. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 939–950, New York, NY, USA, 2010. ACM.

[75] V. S. Pai and P. J. Varman. Prefetching with multiple disks for external mergesort: simulation and analysis. In *[1992] Eighth International Conference on Data Engineering*, pages 273–282, Feb 1992.

[76] HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 618–629, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[77] Hyoungmin Park and Kyuseok Shim. Fast: Flash-aware external sorting for mobile database systems. *Journal of Systems and Software*, 82(8):1298 – 1312, 2009. SI: Architectural Decisions and Rationale.

[78] Steven Pelley. *Database and System Design for Emerging Storage Technologies*. PhD thesis, University of Michigan, 2014.

[79] Steven Pelley, Thomas F Wenisch, and Kristen LeFevre. Do query optimizers need to be SSD-aware? *ADMS'11*, 2011.

[80] Evaggelia Pitoura. Selectivity estimation. In *Encyclopedia of Database Systems*, pages 2548–2548, Boston, MA, 2009. Springer US.

[81] Francois Raab. TPC-C - the standard benchmark for online transaction processing (OLTP). In *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[82] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-tree index optimization by exploiting internal parallelism of flash-based solid state drives. *Proceedings of the VLDB Endowment*, 5(4):286–297, 2011.

[83] B. Salzberg. Merging sorted runs using large main memory. *Acta Inf.*, 27(3):195–215, December 1989.

[84] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34. ACM, 1979.

[85] Nikita Shamgunov. The MemSQL in-memory database system. In *IMDM@ VLDB*, 2014.

[86] Vishal Sikka, Franz Färber, Anil Goel, and Wolfgang Lehner. SAP HANA: The evolution from a modern main-memory data platform to an enterprise application platform. *Proc. VLDB Endow.*, 6(11):1184–1185, August 2013.

[87] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 5th Edition.* McGraw-Hill Book Company, 2005.

[88] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[89] Michael Stonebraker and Ariel Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[90] Andrew S Tanenbaum. Modern operating systems. 2001.

[91] Yun Wang. DB2 query parallelism: Staging and implementation. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 686–691. Morgan Kaufmann Publishers Inc., 1995.

[92] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the General Track of the Annual Conference on USENIX*

Annual Technical Conference, ATEC '02, pages 161–175, Berkeley, CA, USA, 2002. USENIX Association.

[93] John Yiannis and Justin Zobel. Compression techniques for fast external sorting. *The VLDB Journal*, 16(2):269–291, April 2007.

[94] PC Yue and CK Wong. Storage cost considerations in secondary index selection. *International Journal of Computer & Information Sciences*, 4(4):307–327, 1975.

[95] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, July 2015.

[96] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 376–385, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[97] Weiye Zhang and Per-Åke Larson. Buffering and read-ahead strategies for external mergesort. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 523–533, 1998.

[98] LuoQuan Zheng and Per-Åke Larson. Speeding up external mergesort. *IEEE Trans. on Knowl. and Data Eng.*, 8(2):322–332, April 1996.