THE ROLE OF FLASH MEMORY IN DATABASE MANAGEMENT SYSTEMS

by

Jaeyoung Do

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2012

Date of final oral examination: 11/30/12

The dissertation is approved by the following members of the Final Oral Committee:

Jignesh M. Patel, Professor, Computer Sciences

David J. DeWitt, John P. Morgridge Professor, Emeritus, Computer Sciences

Jeffrey F. Naughton, Professor, Computer Sciences

Michael Swift, Assistant Professor, Computer Sciences

Rafael Lazimy, Associate Professor, Operations and Information Management

To my family.

Acknowledgments

Throughout this intellectual journey, I have learned a lot of lessons and experiences from dozens of remarkable individuals whom I wish to acknowledge.

First and foremost, I wish to thank my wonderful advisor, Professor Jignesh M. Patel. He has been supportive since the day I began working on the first class project taught by him. It has truly been a pleasure to work with him, and I will never forget the enthusiasm, sincerity, and encouragement that he showed me. Along with Jignesh, David J. DeWitt has been very helpful and encouraging during my time at Wisconsin. It is an honor to have had opportunities to write very successful papers with him. I would also like to thank Colin Dewey for taking a keen interest in guiding my master thesis. Jeffrey F. Naughton and Michael Swift deserve special thanks for the many useful discussions to point out various improvements on my thesis. I am also very thankful to Rafael Lazimy for taking the time to be on my defense committee.

A special word of thanks should go to my family. Throughout my life, my parents and sister have provided love, faith, and pride in me. Their invaluable support and advice have helped in more ways than I can count, and they have sacrificed to give me all the opportunities possible. I love them all.

I would like to thank all my collaborators with whom I worked at various stages of my research period – Donghui, Alan, Srinath, Nikhil, Rimma, Dimitris, and Eric from Microsoft Jim Gray Systems Lab, and Chanik and Yangsuk from Samsung. I'd also like to thank my friends from the Wisconsin DB group – Avrilia, Ian, Willis, Jessie, Spyros, Khai, and

Kwanghyun. I am very proud that I was a member of this special group!

During my long stay at Wisconsin, I had the good fortune of meeting Pastor Jin's family with whom we prayed for each other. His son (a.k.a my dear brother) Daniel deserves special thanks for the many pleasant socials that we have shared together. I'd also like to thank Nam for the great times arguing spiritedly over various technical issues and entrepreneurship.

Finally, I would like to thank Pinpi who has always been by my side with tremendous love and respect.

Contents

List of Tables

List of Figures

THE ROLE OF FLASH MEMORY IN DATABASE MANAGEMENT SYSTEMS

Jaeyoung Do

Under the supervision of Professor Jignesh M. Patel

At the University of Wisconsin-Madison

For the first time in the history of data processing systems, traditional hard disk drives (HDDs) are under pressure from flash storage in the form of solid state drives (SSDs) as the home for persistent data. SSDs fill the big gap in latency between the main memory and the HDDs, especially for random I/O accesses. However, the internals of database management systems (DBMSs) are traditionally designed and optimized for HDDs – for example, DBMS query processing methods go to great lengths to employ mechanism that generate sequential I/O accesses to the underlying I/O storage subsystem. The focus of this dissertation is on exploring how SSDs can be effectively integrated into existing DBMSs to improve the performance of database query processing methods.

The first part of this thesis examines how traditional join algorithms are impacted when the database is stored entirely in SSDs. The next component of this thesis deals with extending the DBMS buffer pool to gracefully spill evicted pages from the main-memory database buffer pool into a SSD-based cache. In that architecture HDDs are still used to store the database, and the DBMS employs the SSD to accelerate the DBMS query processing performance. This approach allows building database configurations in which one can balance the amount of SSD, which is typically more expensive than HDDs from the $/GB perspective, to improve performance over a pure HDD-based I/O subsystem. We also examine how the SSD buffer pool accelerator can be used to improve the performance of the DBMS when restarting the DBMS following a system crash or a server shutdown event. The basic idea here is to carefully reuse pages that were cached in the SSD prior to the crash or shutdown, while preserving ACID transaction semantics. The final part of this thesis explores accelerating DBMS query processing using *Smart SSDs*. These SSDs have processing components that

are built inside the SSDs and can be made available to run user programs. We explore how Smart SSDs can be used to selectively run certain database query processing operators to improve the overall query performance.

Abstract

For the first time in the history of data processing systems, traditional hard disk drives (HDDs) are under pressure from flash storage in the form of solid state drives (SSDs) as the home for persistent data. SSDs fill the big gap in latency between the main memory and the HDDs, especially for random I/O accesses. However, the internals of database management systems (DBMSs) are traditionally designed and optimized for HDDs – for example, DBMS query processing methods go to great lengths to employ mechanism that generate sequential I/O accesses to the underlying I/O storage subsystem. The focus of this dissertation is on exploring how SSDs can be effectively integrated into existing DBMSs to improve the performance of database query processing methods.

The first part of this thesis examines how traditional join algorithms are impacted when the database is stored entirely in SSDs. The next component of this thesis deals with extending the DBMS buffer pool to gracefully spill evicted pages from the main-memory database buffer pool into a SSD-based cache. In that architecture HDDs are still used to store the database, and the DBMS employs the SSD to accelerate the DBMS query processing performance. This approach allows building database configurations in which one can balance the amount of SSD, which is typically more expensive than HDDs from the $/GB perspective, to improve performance over a pure HDD-based I/O subsystem. We also examine how the SSD buffer pool accelerator can be used to improve the performance of the DBMS when restarting the DBMS following a system crash or a server shutdown event. The basic idea here is to carefully reuse pages that were cached in the SSD prior to the crash or shutdown, while

preserving ACID transaction semantics. The final part of this thesis explores accelerating DBMS query processing using *Smart SSDs*. These SSDs have processing components that are built inside the SSDs and can be made available to run user programs. We explore how Smart SSDs can be used to selectively run certain database query processing operators to improve the overall query performance.

Chapter 1

Introduction

For over three decades, storage system has primarily consisted of a simple hierarchy with DRAM and magnetic hard disk drives (HDDs). This traditional storage hierarchy has often been a bottleneck in high-performance database management systems (DBMSs). The main obstacle in the way of high query performance for very large databases has been the performance gap between the access times of DRAM and disks. Although disk technology has achieved an enormous growth in areal density[1] and produced dramatic reductions in the cost per bit[2], there has been very little improvement in disk access time (around a 3.5-fold improvement over the last 20 years), mainly due to the mechanical components in disks. Consequently, there is a big performance gap, for both access latency and bandwidth, between DRAM and disks, which is many orders of magnitudes today.

One solution to solve this storage system performance issue has been to use a "disk-farm" consisting of a large number of disks. The surprising success of the capacity increase and price decrease of disks has made it attractive to combine multiple disks in RAID configurations to compensate for the bandwidth and latency limitations in enterprise storage servers. This trend, however, brings along with it a different set of problems, including huge power consumption and large space usage. For example, in the year 2020, systems for data-centric workloads

[1] A 35-million-fold increase from 1957 to 2003 [39].
[2] A 4-million-fold decrease from 1957 to 2003 [39].

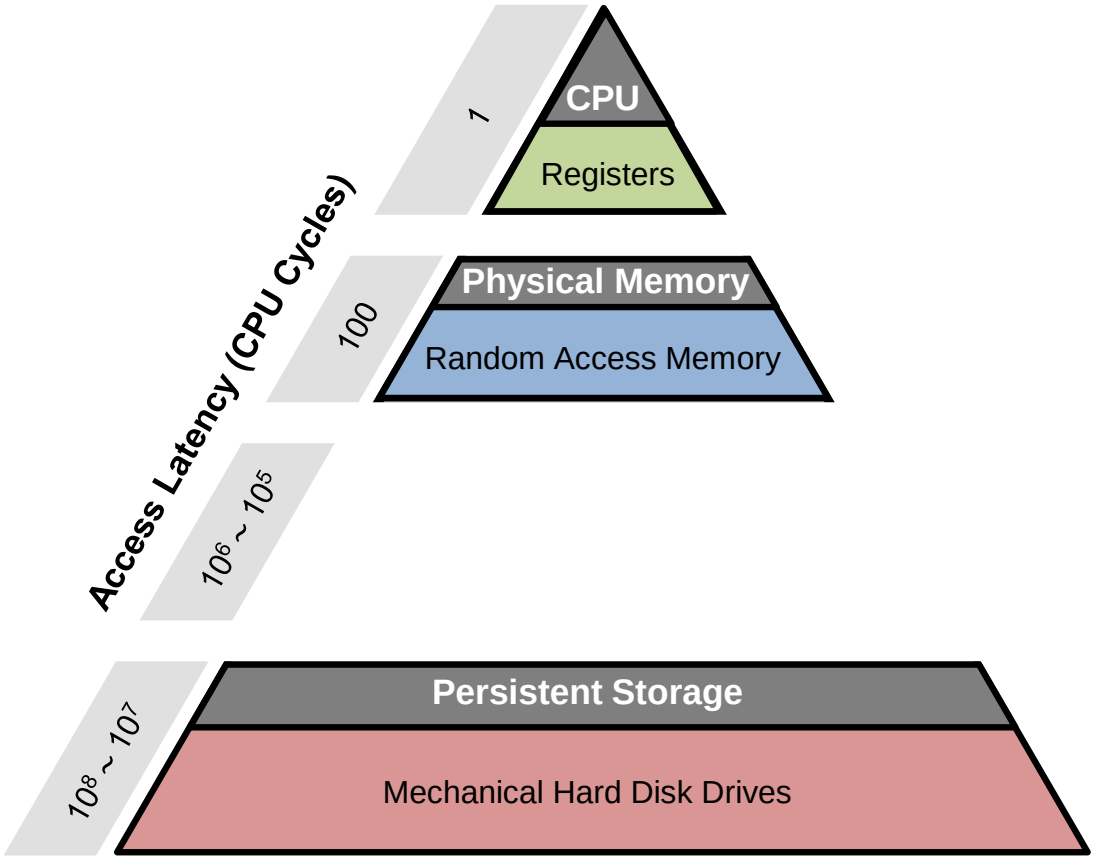
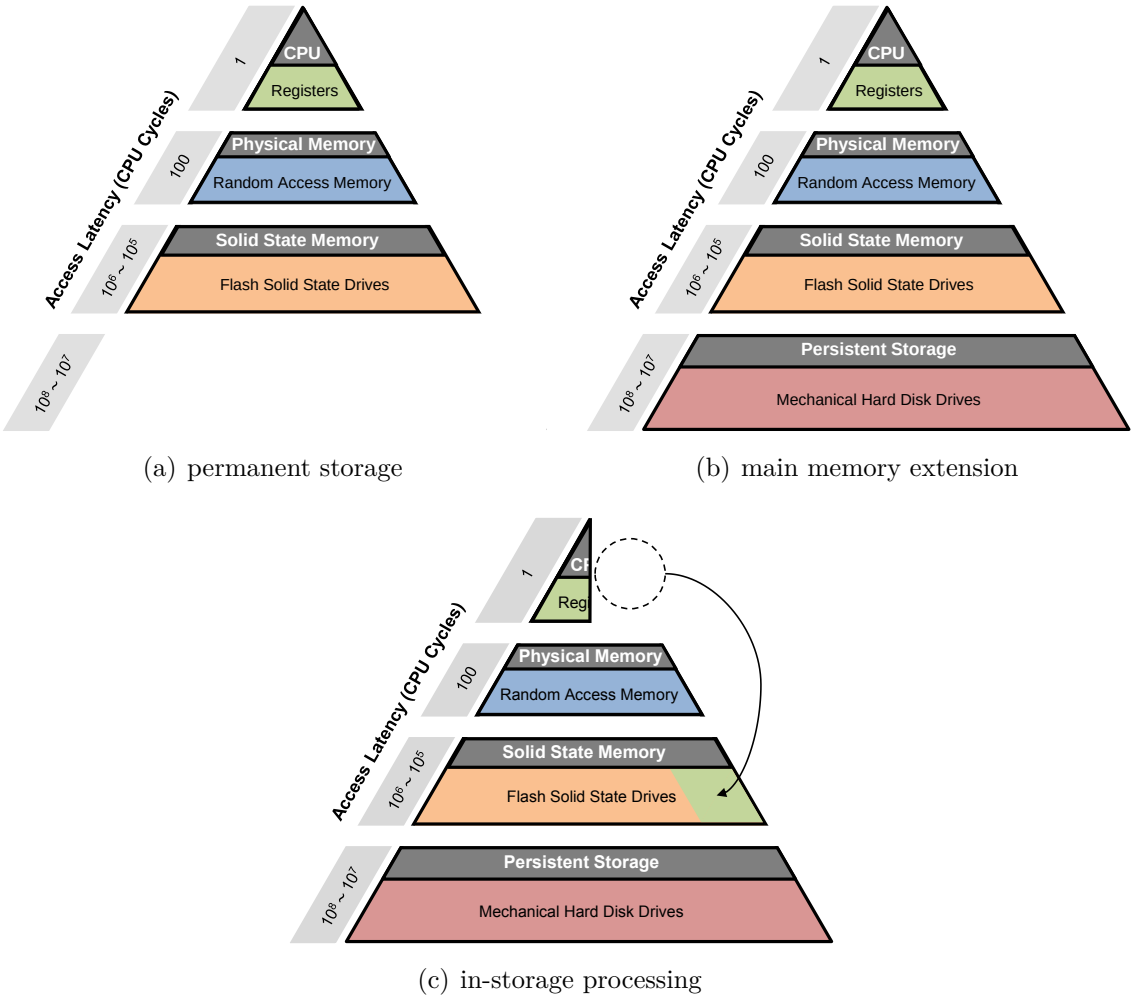


Figure 1.1: Traditional memory hierarchy consisting of CPU, DRAM, and disks. The access latencies in cycles based on a today's multi-GHz CPU are obtained from [2]

(such as analysis of extremely large-scale graphs [41] or large scientific experiments [38]) are expected to be able to perform 2 billion I/O per second (IOPS). With such demand, at least 5 million disks are needed, consuming 22 MW of power and 23K square feet space usage [33].

Thus, simply adding more disks cannot fundamentally address the storage performance bottleneck. Even worse, if this trend continues, the storage server cost (including the installation and energy costs) is likely to severely increase the total cost of ownership (TCO) for enterprise servers, becoming an even more dominant component.

Against this backdrop, the traditional memory hierarchy shown in Figure 1.1 is now undergoing an unprecedented change with the introduction of new flash memory technology such as flash solid state drives (SSDs). SSDs are non-volatile storage devices, and have higher



(a) permanent storage

(b) main memory extension

(c) in-storage processing

Figure 1.2: Various novel memory hierarchies that work with flash SSDs.

I/O performance than disks while being less expensive than DRAM. These characteristics make SSDs an interesting option to mitigate the performance gap between DRAM and disks, by complimenting or completely replacing them.

The key challenge when incorporating SSDs into DBMSs is that they have very different characteristics compared to disk. For instance, SSDs are constructed entirely using flash chips and have no mechanical moving parts, resulting in a high bandwidth for random accesses. As existing DBMSs and their internal algorithms have been designed with disks consisting of rotating platters in mind, the fundamental principles of the 30-year-old storage architecture have to be revisited to make DBMSs work effectively with SSDs.

The goal of this dissertation is to explore how SSDs can be effectively integrated into existing DBMSs to improve the performance of database query processing methods. To achieve this goal, we evaluate several novel DBMS architectures that work with SSDs as shown in Figure 1.2 (a) – (c).

1.1 Join Processing for SSDs

SSDs provide an attractive alternative to traditional HDDs for DBMS applications (Figure 1.2 (a)). Naturally there is substantial interest in redesigning critical database internals, such as join algorithms, for SSDs. However, before designing new algorithms, we must consider the lessons that we have learned from over three decades of designing and tuning join algorithms for disk-based systems. In this work, we recall some of the lessons in the context of ad hoc join algorithms, namely block nested loops join, sort-merge join, Grace hash join and hybrid hash join, and examine the implications of these lessons in designing join algorithms for SSDs.

This work was accepted into the Proceedings of the International Workshop on Data Management on New Hardware (DaMoN), 2009 and won the best paper award [28]. The content of this work can be found in Chapter 2 of this dissertation.

1.2 Turbocharging DBMS Buffer Pool Using SSDs

SSDs can also serve as an accelerator to improve the I/O performance compared to a pure HDD-based I/O subsystem (Figure 1.2 (b)). In this work, we propose and systematically explore designs for using SSDs to improve the performance of a DBMS buffer manager by extending the main-memory buffer pool to an SSD buffer pool. We propose three designs for the SSD buffer pool that differ mainly in the way that they deal with dirty pages that are evicted from the main-memory buffer pool. An evaluation of the proposed methods shows that selectively caching pages in the SSD buffer pool produces big performance gains in the steady state.

This work was accepted into the Proceedings of the ACM SIGMOD Conference, 2011 [31]. This work is described in Chapter 3.

1.3 Fast Peak-to-Peak Restart for SSD Buffer Pool Extension

Using SSDs to extend the main-memory buffer pool can improve the performance of DBMSs (Figure 1.2 (b)). However, simple methods that using the SSD buffer pool methods, commonly discard data that is in the SSDs during system reboots (either when recovering from a crash or restarting after a shutdown). Consequently, these methods require a long "ramp-up" period to regain peak performance. In this work we propose two schemes that work with SSD buffer pool designs to reuse the pages that are cached in the SSD buffer pool after a server restart. Our results show that the proposed methods reduce the ramp-up time significantly, without impacting the actual peak performance that can be achieved when using the SSD buffer pool designs.

This work was presented at the New England Database Summit (NEDB Summit), 2012 [29], and accepted into the Proceedings of the IEEE International Conference on Data Engineering (ICDE), 2013 [30]. The content of this work is found in Chapter 4 of this dissertation.

1.4 Query Processing inside Smart SSDs

In the near future, SSDs will have computing resources that are made available to run general user-defined programs (Figure 1.2 (c)). The focus of this work is on exploring the opportunities and challenges associated with exploiting this functionality for relational analytic query processing with these "Smart SSDs". Our results demonstrate that significant performance and energy gains can be achieved by pushing selected query processing components inside

the SSDs. We also identify various changes that Smart SSD device manufacturers can make to increase the benefits of using Smart SSDs for data processing applications, and suggest possible research opportunities for the database community.

This work is described in Chapter 5 of this dissertation.

1.5 Outline

The remainder of this dissertation is organized as follows: Chapter 2 show how and when traditional disk-based techniques can be reused for join algorithms running on data that is resident in SSDs. Chapter 3 introduces designs to use SSDs as an extension to the main-memory buffer pool, and Chapter 4 continues the work to reuse SSD buffer pool pages even after a system restart. Chapter 5 presents a way to offload database operations onto Smart SSDs. Chapter 6 concludes the dissertation.

Chapter 2

Join Processing for SSDs

Flash solid state drives (SSDs) provide an attractive alternative to traditional magnetic hard disk drives for DBMS applications. Naturally there is substantial interest in redesigning critical DBMS internals, such as join algorithms, for flash SSDs. However, we must carefully consider the lessons that we have learnt from over three decades of designing and tuning algorithms for disk-based systems, so that we continue to reuse techniques that worked for disks and also work with SSDs.

In this chapter, we focus on recalling some of these lessons in the context of *ad hoc* join algorithms. Based on an actual implementation of four common ad hoc join algorithms on both a magnetic disk and a flash SSD, we show that many of the "surprising" results from disk-based join methods also hold for flash SSDs. These results include the superiority of block nested loops join over sort-merge join and Grace hash join in many cases, and the benefits of blocked I/Os. In addition, we find that simply looking at the I/O costs when designing new flash SSD join algorithms can be problematic, as the CPU cost is often a bigger component of the total join cost with SSDs.

2.1 Introduction

Flash solid state drives (SSDs) are actively being considered as storage alternatives to replace or dramatically reduce the central role of magnetic hard disk drives (HDDs) as the main choice for storing persistent data. Jim Gray's prediction of "Flash is disk, disk is tape, and tape is dead" [36] is coming close to reality in many applications. Flash SSDs, which are made by packaging (NAND) flash chips, offer several advantages over magnetic HDDs including faster random reads and lower power consumption. Moreover, as flash densities continue to double as predicted in [44], and prices continue to drop, the appeal of flash SSDs for DBMSs increases. In fact, vendors such as Fusion-IO and HP sell flash-based devices as I/O accelerators for many data-intensive workloads.

The appeal of flash SSDs is also attracting interest in redesigning various aspects of DBMS internals for flash SSDs. One such aspect that is becoming attractive as a research topic is join processing algorithms, as it is well-known that joins can be expensive and can play a critical role in determining the overall performance of the DBMS.

While such efforts are well-motivated, we want to approach a redesign of database query processing algorithms by clearly recalling the lessons that the community has learnt from over three decades of research in query processing algorithms. The focus of this chapter is on recalling some of the important lessons that we have learnt about efficient join processing in magnetic HDDs, and determining if these lessons also apply to joins using flash SSDs. In addition, if previous techniques for tuning and improving the join performance also work for flash SSDs, then it also changes what are interesting starting point for comparing new SSD-based join algorithms.

In the case of join algorithms, a lot is known about how to optimize joins with magnetic HDDs to use the available buffer memory effectively, and to account for the characteristics of the I/O subsystem. Specifically, the paper by Haas *et al.*[40] derives detailed formulae for buffer allocation for various phases of common join algorithms such as block nested loops join, sort-merge join, Grace hash join, and hybrid hash join. Their results show that the

right buffer pool allocation strategy can have a huge impact - upto 400% improvements in some cases. Furthermore, the relative performance of the join algorithms changes once you optimize the buffer allocations - block nested loops join is much more versatile, and Grace hash join is often not very competitive.

The dangers of forgetting these lessons could lead to an incorrect starting point for comparing new flash SSD join algorithms. For example, the comparison of RARE-join [66] with Grace hash join [47] to show the superiority of the RARE-join algorithm on flash SSDs is potentially not the right starting point. (It is possible that the RARE-join is superior to the best magnetic HDD-based join method when run over flash SSDs, but this question has not been answered conclusively.) As we show in this chapter, in fact even block nested loops join far outperforms Grace hash join in most cases, on both magnetic HDDs and flash SSDs. Cautiously, we note that we have only tried one specific magnetic HDD and one specific flash SSD, but even this very first test produced interesting results.

The focus of this chapter in on investigating four popular *ad hoc* join algorithms, namely block nested loops join, sort-merge join, Grace hash join, and hybrid hash join, on both flash SSDs and magnetic HDDs. We start with the best buffer allocation methods that are proposed in [40] for these join algorithms, and first ask the question: "What changes for these algorithms as we replace a magnetic HDD with a flash SSD?" Then, we study the effect of changing the buffer pool sizes and the page sizes and examine the impact of these changes on these join algorithms. Our results show that many of the techniques that were invented for joins on magnetic HDDs continue to hold for flash SSDs. As an example, *blocked* I/O is useful on both magnetic HDDs and flash SSDs, though for different reasons. In the case of magnetic HDDs, the use of blocked I/O amortizes the cost of disk seeks and rotational delays. On the other hand, the benefit of blocked I/O with flash SSDs comes from amortizing the latency associated with the software layers of flash SSDs, and generating fewer erase operations when writing data.

The remainder of this chapter is organized as follows. In Section 2.2, we briefly introduce

the characteristics of flash SSDs. Then we introduce the four classic join algorithms with appropriate assumptions and buffer allocation strategies in Section 2.3. In Section 2.4, we explain and discuss the experimental results. After reviewing related work in Section 2.5, we summarize this work in Section 2.6.

2.2 Characteristics of Flash SSD

Flash SSDs are based on NAND flash memory chips and use a controller to provide a persistent block device interface. A flash chip stores information in an array of memory cells. A chip is divided into a number of *flash blocks*, and each flash block contains several *flash pages*. Each memory cell is set to 1 by default. To change the value to 0, the entire block has to be erased by setting it to 1, followed by selectively programming the desired cells to 0. Read and write operations are performed at the granularity of a flash page. On the other hand, the time-consuming erase operations can only be done at the level of a flash block. Considering the typical size of a flash page (4 KB) and a flash block (64 flash pages), the erase-before-write constraint can significantly degrade write performance. In addition, most flash chips only support 10^5~10^6 erase operations per flash block. Therefore, erase operations should be distributed across the flash blocks to prolong the service life of flash chips. These kinds of constraints are handled by a software layer known as *flash translation layer* (FTL).

The major role of the FTL is to provide address mappings between the *logical block addresses* (LBAs) and flash pages. The FTL maintains two kinds of data structures: *A direct map* from LBAs to flash pages, and *an inverse map* for re-building the direct map during recovery. While the inverse map is stored on flash, the direct map is stored on flash and at least partially in RAM to support fast lookups. If the necessary portion of the direct map is not in RAM, it must be swapped in from flash as required.

While flash SSDs have no mechanically moving parts, data access still incurs some latency, due to overheads associated with the FTL logic. However, latencies of flash SSDs are typically

much smaller than those of magnetic HDDs [21].

2.3 Joins

In this section, we introduce four classic ad hoc join algorithms that we consider in this chapter, namely: block nested loops join, sort-merge join, Grace hash join, and hybrid hash join. The two relations being joined are denoted as R and S. We use $|R|$, $|S|$ and B to denote the sizes of the relations and the buffer pool size in pages, respectively. We also assume that $|R| \leqslant |S|$. Each join algorithm needs some extra space to build and maintain specific data structures such as a hash table or a tournament tree. In order to model these structures, we use a multiplicative *fudge factor*, denoted as F.

Next we briefly describe each join algorithm. We also outline the buffer allocation strategy for each join algorithm. The I/O costs for writing the final results are omitted in the discussion below, as this cost is identical for all join methods. For the buffer allocation strategy we directly use the recommendations by Haas *et al.* [40] (for magnetic HDDs), which shows that optimizing buffer allocations can dramatically improve the performance of join algorithms (by 400% in some cases).

2.3.1 Block Nested Loops Join

Block nested loops join first logically splits the smaller relation R into same size chunks. For each chunk of R that is read, a hash table is built to efficiently find matching pairs of tuples. Then, all of S is scanned, and the hash table is probed with the tuples. To model the additional space required to build a hash table for a chunk of R we use the fudge factor F, so a chunk of size $|C|$ pages uses $F|C|$ pages in memory to store a hash table on C.

The buffer pool is simply divided into two spaces; one space, $\mathrm{I}_{\mathtt{outer}}$, is for an input buffer with a hash table for R chunks, and another one, $\mathrm{I}_{\mathtt{inner}}$, is for an input buffer to scan S. Note that reading R in chunks of size $\frac{\mathrm{I}_{\mathtt{outer}}}{\mathsf{F}}$ $(= |C|)$ guarantees sufficient memory to build a

hash table in memory for that chunk [22].

2.3.2 Sort-Merge Join

Sort-merge join starts by producing sorted runs of each R and S. After R and S are sorted into runs on disk, sort-merge join reads the runs of both relations and merges/joins them. We use the tournament sort (*a.k.a.* heap sort) in the first pass, which produces runs that on average are twice the size of the memory used for the initial sorting [48]. We also assume $\mathsf{B} > \sqrt{\mathsf{F}|\mathsf{S}|}$ so that the sort-merge join, which uses a tournament tree, can be executed in two passes [67].

In the first pass, the buffer pool is divided into three parts: an input buffer, an output buffer, and working space (WS) to maintain the tournament tree. During the second pass, the buffer pool is split across all the runs of R and S as evenly as possible.

2.3.3 Grace Hash Join

Grace hash join has two phases. In the first phase, it reads each relation, applies a hash function to the input tuples, and hashes tuples into buckets that are written to disk. In the second phase, the first bucket of R is loaded into the buffer pool, and a hash table is built on it. Then, the corresponding bucket of S is read and used to probe the hash table. Remaining buckets of R and S are handled in the same way iteratively. We assume $\mathsf{B} > \sqrt{\mathsf{F}|\mathsf{R}|}$ to allow for a two-phase Grace hash join [67].

There are two sections in the buffer pool during the first partitioning phase: one input buffer and an output buffer for each of the k buckets. We subdivide the output buffer as evenly as possible based on the number of buckets, and then give the remaining pages, if any, to the input buffer. In the second phase, a portion of the buffer pool (WS') is used for the $\mathfrak{i}^{\mathsf{th}}$ bucket of R and its hash table, and the remaining pages are chosen as input buffer pages to read S.

2.3.4 Hybrid Hash Join

As in the Grace hash join, there are two phases in this algorithm, assuming $\mathsf{M} > \sqrt{\mathsf{F}|\mathsf{R}|}$. In the first phase, the relation R is read and hashed into buckets. Since a portion of the buffer pool is reserved for an in-memory hash bucket for R, this bucket of R is not written to a storage device while the other buckets are. Furthermore, as S is read and hashed, tuples of S matching with the in-memory R bucket can be joined immediately, and need not be written to disk. The second phase is the same as Grace hash join.

During the first phase, the buffer pool is divided into three parts: one for the input, one for the output of k buckets excluding the in-memory bucket, and the working space (WS) for the in-memory bucket. The buffer allocation scheme for the second phase is the same as Grace hash join.

2.3.5 Buffer Allocation

Table 2.1 shows the optimal buffer allocations for the four join algorithms, for each phase of these algorithms. Note that block nested loops join does not distinguish between the different passes.

In this chapter, we use the same buffer allocation method for both flash SSDs and magnetic HDDs. While these allocations may not be optimal for flash SSDs, our goal here is to start with the best allocation strategy for magnetic HDDs and explore what happens if we simply use the same settings when replacing a magnetic HDD with a flash SSD.

2.4 Evaluation

In this section, we compare the performance of the join algorithms using the optimal buffer allocations when using a magnetic HDD and a flash SSD for storing the input data sets. Each data point presented here is the average over three runs.

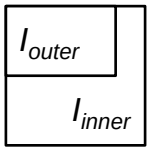
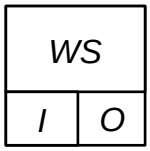
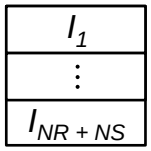
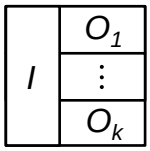
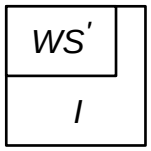
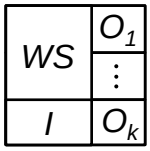
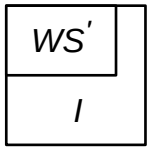
Algorithms	First Phase/Pass		Second Phase/Pass	
BNL		$I_{inner} = \lceil \frac{\sqrt{y\|S\|(y\|S\|+B(y+\|S\|))}-y\|S\|}{y+\|S\|} \rceil$ $y = \frac{Dl}{Dx}$ $I_{outer} = B - I_{inner}$		
SM		$I = O = \lceil \frac{(\sqrt{2z}-4)\cdot B}{z-8} \rceil$ $z = \frac{(Dl+Ds)\cdot F(\|R\|+\|S\|)}{Dl\cdot B}$ $WS = B - I - O$		$I \simeq \lfloor \frac{B}{NR+NS} \rfloor$
GH		$k = \lceil \frac{\|R\|F+\sqrt{\|R\|^2F^2+4B\|R\|F}}{2B} \rceil$ $O = \lfloor \frac{B}{k+1} \rfloor$ $I = B - k\cdot O$		$WS' \simeq \lceil \frac{F\|R\|}{k} \rceil$ $I = B - WS'$
HH		$I = O = \lceil 1.1\sqrt{B} \rceil$ $k = \lceil \frac{F\|R\|-(B-I)}{B-I-O} \rceil$ $WS = B - I - k\cdot O$		$WS' \simeq \lceil \frac{F\|R\|-WS}{k} \rceil$ $I = B - WS'$

Table 2.1: Buffer allocations for join algorithms from Haas *et al.* [40]: BNL, SM, GH, and HH stand for block nested loops join, sort-merge join, Grace hash join, and hybrid hash join, respectively. Ds, Dl, and Dx denote average seek time, latency, and page transfer time for magnetic HDDs, respectively.

2.4.1 Experimental Setup

We implemented a single-thread and light-weight database engine that uses the SQLite3 [5] page format for storing relational data in heap files. Each heap file is stored as a file in the operating system, and the average page utilization is 80%. Our engine has a buffer pool that allows us to control the allocation of pages to the different components of the join algorithms. The engine has no concurrency control or recovery mechanisms.

Our experiments were performed on a Dual Core 3.2 GHz Intel Pentium machine with 1 GB of RAM running Red Hat Enterprise 5. For the comparison, we used a 5,400 RPM

Comments	Values
Magnetic HDD cost	$129.99 (0.36 $/GB)
Magnetic HDD average seek time	12 ms
Magnetic HDD latency	5.56 ms
Magnetic HDD transfer rate	34 MB/s
Flash SSD cost	$230.99 (3.85 $/GB)
Flash SSD latency	0.35 ms
Flash SSD read transfer rate	120 MB/s
Flash SSD write transfer rate	80 MB/s
Page size	2 KB ~ 32 KB
Buffer pool size	100 MB ~ 600 MB
Fudge Factor	1.2
orders table size	5 GB
customer table size	730 MB

Table 2.2: Device characteristics and parameter values.

TOSHIBA 320 GB external HDD and a OCZ Core Series 60 GB SATA II 2.5 inch flash SSD.

We used wall clock time as a measure of execution time, and calculated the I/O time by subtracting the reported CPU time from the wall clock time. Since *synchronous* I/Os were used for all tests, we assumed that there is no overlap between the I/O and the computation. We also used *direct* I/Os so that the database engine transfers data directly from/to the buffer pool bypassing the OS cache (so there is no prefetching and double buffering). With this setup all join numbers repeated here are "cold" numbers.

2.4.2 Data Set and Join Query

As our test query, we used a primary/foreign key join between the TPC-H [9] customer and the orders tables, generated with a scale factor of 30. The customer table contains 4,500,000 tuples (730 MB), and the orders table has 45,000,000 (5 GB). Each tuple of both tables contains an unsigned 4 byte integer key (the customer key), and an average 130 and 90 bytes of padding for the customer and the orders tables respectively. The data for both tables were stored in random order in the corresponding heap files.

	Buffer Pool Size					
	100 MB		200 MB		300 MB	
Algorithms	Join	I/O	Join	I/O	Join	I/O
BNL	1.64X	2.86X	1.59X	2.62X	1.72X	3.04X
SM	1.41X	1.81X	1.45X	2.05X	1.44X	2.06X
GH	1.34X	1.54X	1.29X	1.59X	1.41X	1.62X
HH	1.45X	1.77X	1.55X	1.90X	1.35X	1.51X

	Buffer Pool Size					
	400 MB		500 MB		600 MB	
Algorithms	Join	I/O	Join	I/O	Join	I/O
BNL	1.73X	2.87X	1.67X	2.79X	1.65X	2.61X
SM	1.45X	2.08X	1.43X	2.04X	1.48X	2.20X
GH	1.33X	1.62X	1.39X	1.77X	1.30X	1.55X
HH	1.51X	1.89X	1.50x	1.78X	1.65X	2.09X

Table 2.3: Speedups of total join times and I/O times with flash SSDs

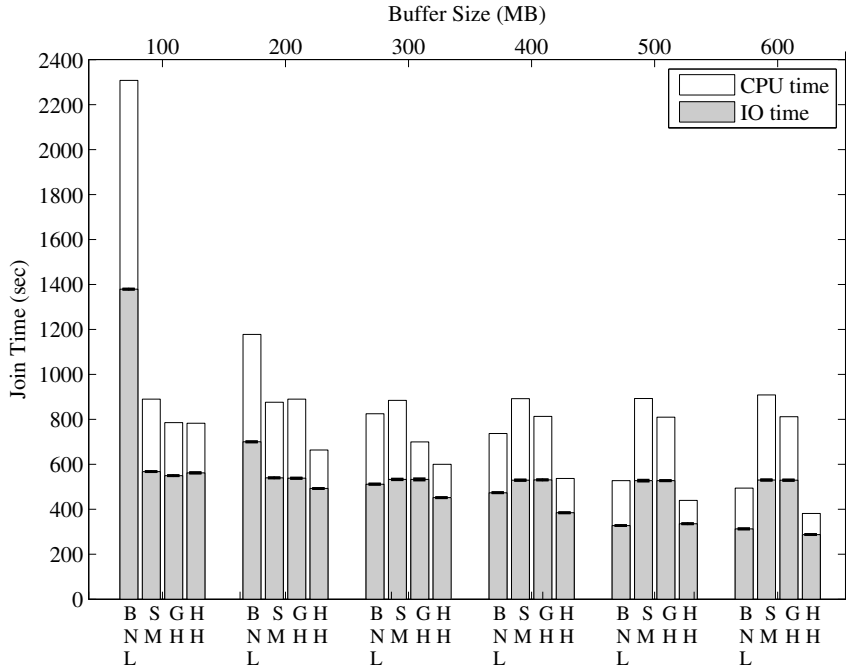
Characteristics of the magnetic HDD and the flash SSD, and parameter values used in these experiments are shown in Table 2.2.

2.4.3 Effect of Varying the Buffer Pool Size

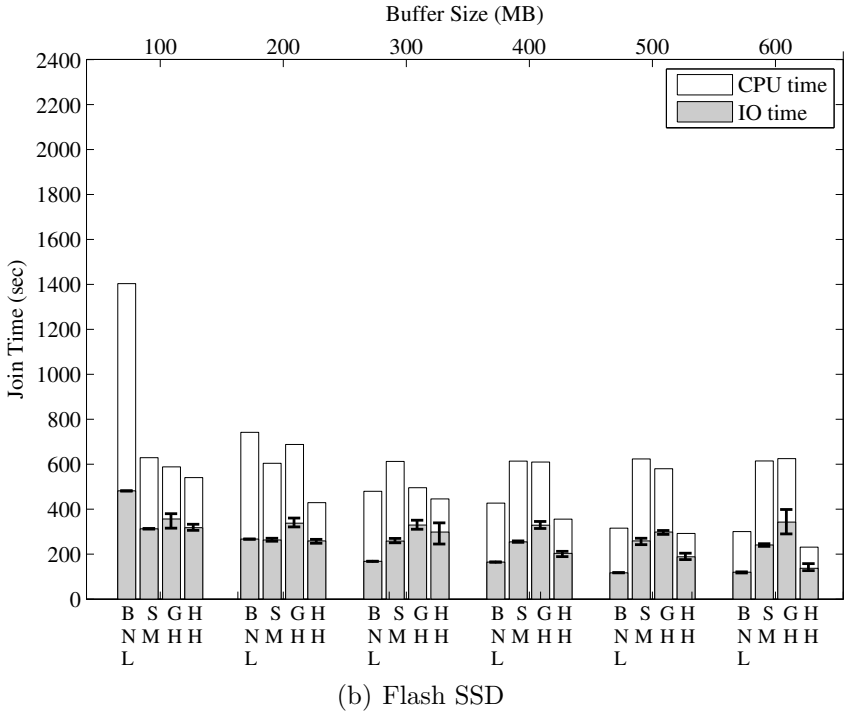
The effect of varying the buffer pool size from 100 MB to 600 MB is shown in Figure 2.1, for both the magnetic HDD and the flash SSD. We also used *blocked I/O* to sequentially read and write multiple pages in each I/O operation. The size of the I/O block is calculated for each algorithm using the equations shown in Table 2.1.

In Figure 2.1 error bars denote the minimum and the maximum measured I/O times (across the three runs). Note that the error bars for the CPU times are omitted, as their variation is usually less than 1% of the total join time.

Table 2.3 shows the speedup of the total join times and the I/O times of the four join algorithms under different buffer pool sizes. The results in Table 2.3 show that replacing the magnetic HDD with the flash SSD benefits all the join methods. The block nested loops join whose I/O pattern is sequential reads shows the biggest performance improvement, with



(a) Magnetic HDD



(b) Flash SSD

Figure 2.1: Varying the size of the buffer pool (8 KB page, blocked I/O)

	Algorithms					
	Sort-Merge Join		Grace Hash Join		Hybrid Hash Join	
BP Size	1st Phase	2nd Phase	1st Phase	2nd Phase	1st Phase	2nd Phase
100 MB	1.52X	3.00X	1.34X	2.27X	1.57X	2.61X
200 MB	1.83X	2.81X	1.43X	2.19X	1.66X	3.09X
300 MB	1.86X	2.79X	1.47X	2.12X	1.34X	2.32X
400 MB	1.90X	2.63X	1.47X	2.13X	1.70X	2.91X
500 MB	1.81X	2.86X	1.59X	2.44X	1.63X	2.83X
600 MB	2.00X	2.89X	1.31X	2.68X	1.98X	2.84X

Table 2.4: Speedups of I/O times with flash SSDs, broken down by the first and second phases. BP Size denotes the buffer pool size.

speedup factors between 1.59X to 1.73X. (Interestingly, a case can be made that for sequential reads and writes, comparable or much higher speedups can be achieved with striped magnetic HDDs, for the same $ cost [64].)

Other join algorithms also performed better on the flash SSD compared to the magnetic HDD, with smaller speedup improvements than the block nested loops join. This is because the write transfer rate is slower than the read transfer rate on the flash SSD (See Table 2.2), and unexpected erase operations might degrade write performance further.

As an example, different I/O speedups were achieved in the first and the second phases of the sort-merge join as shown in Table 2.4. While the I/O speedup of the second phase was between 2.63X and 3.0X due to faster random reads, the I/O speedup in the first phase (that has sequential writes as the dominant I/O pattern), was only between 1.52X and 2.0X, which reduced the overall speedup for sort-merge join.

In the case of Grace hash join, all the phases were executed with lower I/O speedups than those of the sort-merge join (See Table 2.4). Note that the dominant I/O pattern of Grace hash join is random writes in the first phase, followed by sequential reads in the second phase. While the I/O speedup between 2.12X and 2.68X was observed for the second phase of Grace hash join, the I/O speedup of its first phase was only between 1.31X and 1.59X due to expensive erase operations. This indicates that algorithms that stress random reads, and

avoid random writes as much as possible are likely to see bigger improvements on flash SSDs (over magnetic HDDs).

While there is little variation in the I/O times with the magnetic HDD (See the error bars in Figure 2.1(a) for the I/O bars), we observed higher variations in the I/O times with the flash SSD (Figure 2.1(b)), resulting from the varying write performance. Note that since random writes cause more erase operations than sequential writes, hash-based joins show wider range of I/O variations than sort-merge join. On the other hand, there is little variation in the I/O costs for block nested loops join regardless of the buffer pool size, since it does not incur any writes.

Another interesting observation that can be made here (Figure 2.1) is the relative I/O performance between the sort-merge join and Grace hash join. Both have similar I/O costs with the magnetic HDD, but sort-merge join has lower I/O costs with the flash SSD. This is mainly due to the different output patterns of both join methods. In the first phase of the joins, where both algorithms incur about 80% of the total I/O cost, each writes intermediate results (sorted runs for sort-merge join, and buckets for Grace hash join) to disk in different ways; sort-merge join incurs sequential writes as opposed to the random writes that are incurred by Grace hash join. While this difference in the output patterns has a substantial impact on the join performance with the flash SSD because random writes generate more erase operations than sequential writes, the impact is relatively small with the magnetic HDD.

In addition, we can not argue that the sort-merge join algorithm is better than the Grace hash join algorithm on flash SSDs based solely on the I/O performance results (as in [66]). While looking at only the I/O characteristics is sometimes okay for magnetic HDDs (when the join algorithm is I/O bound), using flash SSDs makes the CPU costs more prominent. As seen in Figure 2.1(b), the CPU cost now dominate the I/O cost in most cases for block nested loops join and sort-merge join. In general, with flash SSDs the balance between CPU and I/Os changes, which implies that when building systems with flash SSDs we may want

to consider adding more CPU processing power.

From Figure 2.1, we notice that hybrid hash join outperformed all other algorithms on both the magnetic HDD and the flash SSD, across the entire range of buffer pool sizes that we tested. Hybrid hash join is 2.81X faster than Grace hash join with a 600 MB buffer pool, indicating that comparing only with Grace hash join (as done in [59, 66]) could be misleading. Finally note that in our experiments, with the flash SSD, block nested loops join is faster than sort-merge join and Grace hash join for large buffer pool sizes, but slower than hybrid hash join, which is more CPU efficient (though incurs a higher I/O cost!).

In summary:

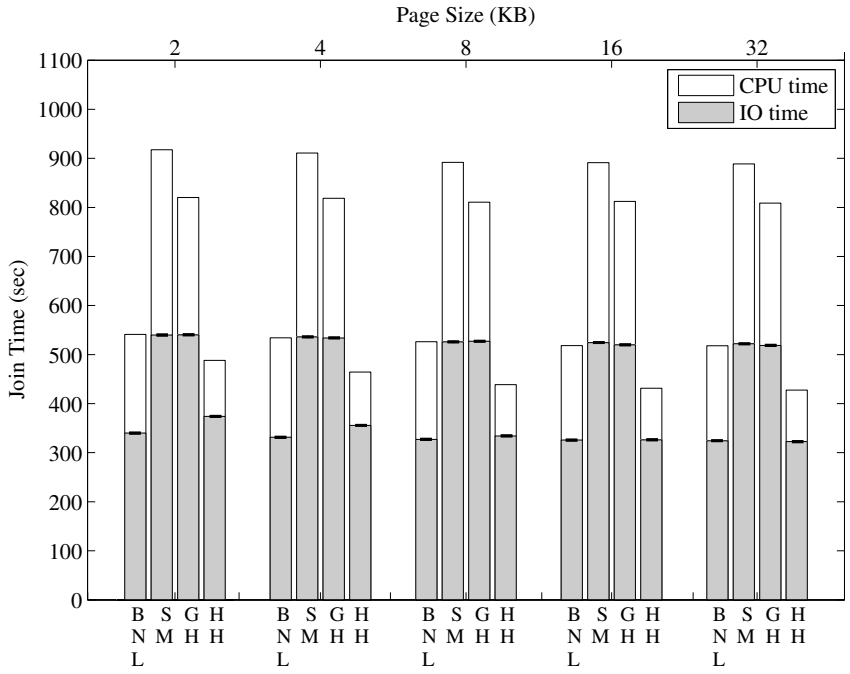
- Joins on flash SSDs have a greater tendency to become CPU-bound (rather than I/O-bound), so ways to improve the CPU performance, such as better cache utilization, is of greater importance with flash SSDs.
- Trading random reads for random writes is likely a good design choice for flash SSDs.
- Compared to sequential writes, random writes produce more I/O variations with flash SSDs, which makes the join performance less predictable.

2.4.4 Effect of Varying the Page Size

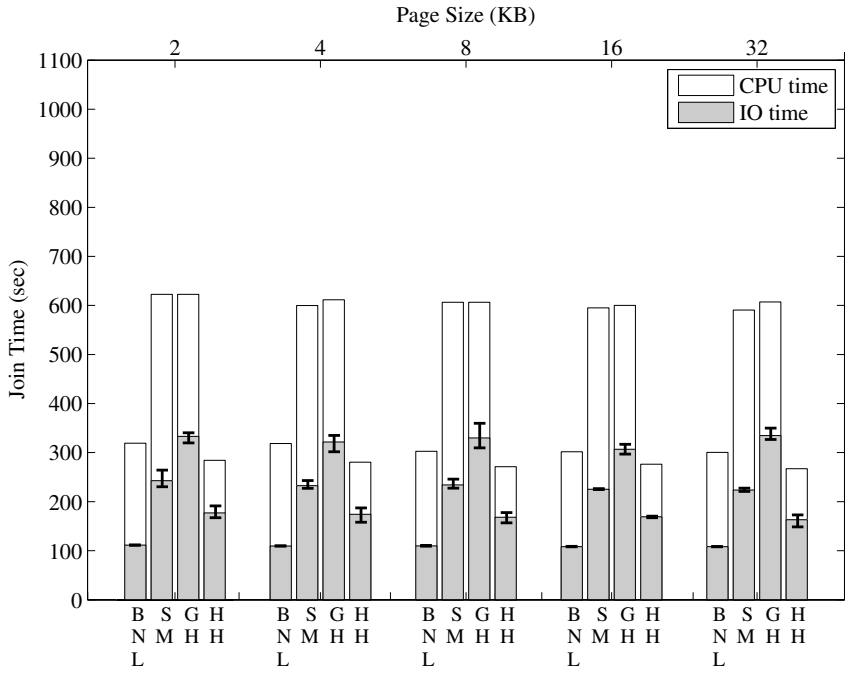
In this experiment, we continue to use blocked I/O, but vary the page size from 2 KB to 32 KB. (The maximum page size allowed by SQLite3 is currently 32 KB.) The size of the I/O block is also calculated using the equations shown in Table 2.1, as in the previous section. The results are shown in Figure 2.2 for a 500 MB buffer pool.

As can be seen from Figure 2.2, when blocked I/O is used, the page size has a small impact on the join performance in both the magnetic HDD and the flash SSD cases.

The key lesson here is that if blocked I/O is used, the database system can likely set the page size based on criteria other than join performance.

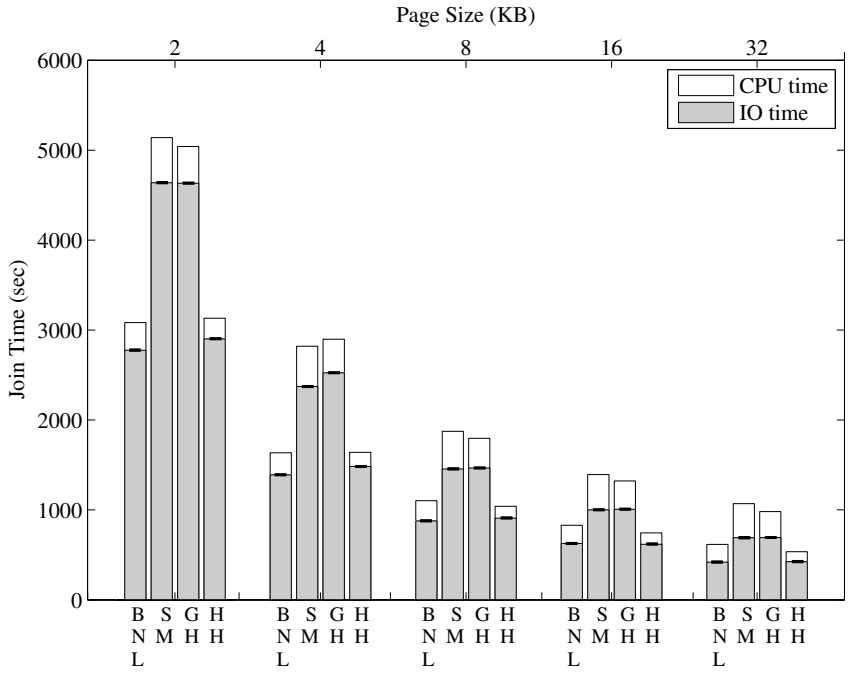


(a) Magnetic HDD

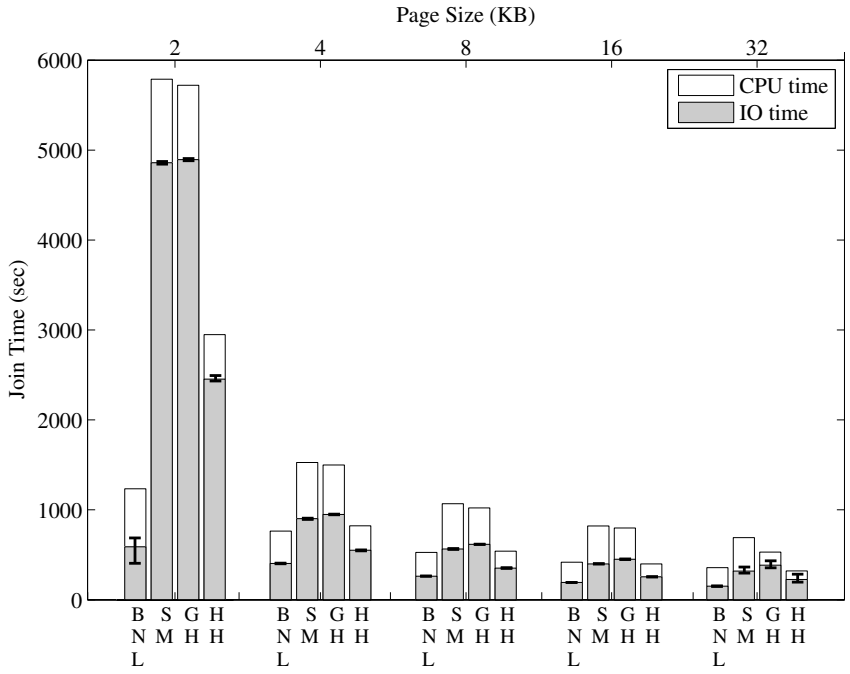


(b) Flash SSD

Figure 2.2: Varying the page size (500 MB Buffer Pool, blocked I/O)



(a) Magnetic HDD



(b) Flash SSD

Figure 2.3: Varying the page size (500 MB Buffer Pool, page sized I/O)

	Magnetic HDD		Flash SSD	
Page Size	User	Kernel	User	Kernel
2 KB	187.5 sec	108.4 sec	204.4 sec	324.0 sec
4 KB	192.8 sec	49.0 sec	205.5 sec	157.9 sec
8 KB	190.5 sec	27.9 sec	183.6 sec	80.6 sec
16 KB	186.6 sec	15.5 sec	188.8 sec	39.8 sec
32 KB	187.2 sec	8.6 sec	185.9 sec	22.4 sec

Table 2.5: CPU times for block nested loops join on the magnetic HDD and the flash SSD

2.4.5 Effect of Blocked I/O

The major reason for using blocked I/O with magnetic HDDs is to amortize the high cost of disk seeks and rotational delays. An interesting question is: "Does blocked I/O still make sense for flash SSDs?" To answer this question, we repeated the experiment described in the previous section with page sized I/O instead of blocked I/O. These results are shown in Figure 2.3.

Comparing Figure 2.3 with Figure 2.2, we can clearly see that blocked I/O is still valuable for the flash SSD, often improving the performance by 2X. The reasons for this are: a) the software layer of the flash SSD still incur some latency [21], making larger I/O size attractive, and b) the write operations with larger I/O sizes generate fewer erase operations, as the software layer is able to manage a pool of pre-erased blocks more efficiently.

When the page size is 2 KB, the performance of sort-merge join and Grace hash join on the flash SSD is worse than on the magnetic HDD. When the I/O size is less than the flash page size (4 KB), every write operation is likely to generate an erase operation, which severely degrades performance. This results re-confirms the observation (but for joins) that blocks should be aligned to flash pages [21].

We also observed that the CPU costs on the flash SSD and on the magnetic HDD are different for the same page size. CPU costs are generally larger with the flash SSD, due to the complex nature of FTL on the flash SSD. As described in Section 2.2, FTL not only provides the ability to map addresses, but also needs to support many other functionalities

such as updating map structures, maintaining a pool of pre-erased blocks, wear-leveling and parallelism to improve performance. As an example, Table 2.5 shows CPU times of the block nested loops join, broken down by the time spent in the user and the kernel modes. (Other join methods showed similar behavior.) From this table, we observe that the kernel CPU times are larger with the flash SSD. This gap between the CPU costs for the flash SSD over the magnetic HDD is larger for smaller page sizes, since the smaller page sizes result in more I/O requests, which keeps the FTL busier with frequent direct map look-ups and indirect map updates. On the other hand, the gap is smaller with larger page sizes, and eventually the CPU costs for the SSD over the magnetic disk are almost the same when blocked I/O is used as in Figure 2.2.

In summary:

- Using blocked I/O significantly improves the join performance on flash SSDs over magnetic HDDs.
- The I/O size should be a multiple of the flash page size.

2.5 Related Work

There is substantial related work on the use of flash memory for DBMSs. Graefe [34] revisits the famed five-minute rule based on flash and points out several potential uses in DBMSs. Lee *et al.* [53] suggests a log-based storage system that can convert random writes into sequential writes. In that paper they presents a new design for logging updates to the end of each flash erase blocks, rather than doing in-place updates to avoid expensive erase operations. Lee *et al.* [54] observe the characteristics of hash join and sort-merge join in a commercial system and conclude that for that system, sort-merge join is better suited for flash SSDs; however the implementation details of the hash join method in the commercial system is not know. Myer [59] examines join performance on flash SSDs under a set of realistic I/O workloads

with Berkeley DB on a fixed-sized buffer pool. Of the hash-based join algorithms, only Grace hash join is considered in this study.

Shah *et al.* [66] show that for flash memory the PAX [15] layout of data pages is better than a row-based layout for scans. They also suggest a new join algorithm for flash SSDs. No direct implementation of the algorithm is presented, but the potential benefits of the new algorithm are presented by using an analytical model, and comparing it to Grace hash join. Tsirogiannis *et al.* [68] presents a new pipelined join algorithm in combination with the PAX layout. A key aspect of their new algorithm is to minimize I/Os by retrieving only required attributes as late as possible. They show that their algorithm is much more efficient on flash SSDs compared to hybrid hash join, especially when either few attributes or few rows are selected in the join result.

More recently, Bouganim *et al.* [21] have provided a collection of nine micro-benchmarks based on various I/O patterns to understand flash device performance.

2.6 Summary

In this chapter, we have presented and discussed the performance characteristics of four well-known ad hoc join algorithms on a magnetic HDD and a flash SSD. Based on our evaluation, we conclude that

- buffer allocation strategy has a critical impact on the performance of join algorithms for both magnetic HDDs and flash SSDs.
- despite the absence of mechanically moving parts, blocked I/O plays an important role for flash SSDs.
- both CPU times and I/O costs must be considered when comparing the performance of join algorithms as the CPU times can be a larger (and sometimes the dominating) proportion of the overall join cost with flash SSDs.

Many of these observations are lessons that we have learnt from previous work on optimizing join algorithms for magnetic HDDs, and continue to be important when studying the performance of join algorithms on flash SSDs, though with different emphases.

Chapter 3

Turbocharging DBMS Buffer Pool Using SSDs

In the previous chapter, we investigated what happens when magnetic HDDs are replaced with flash SSDs in the context of ad hoc join algorithms. In this chapter we propose and systematically explore designs for using SSDs to improve the performance of a DBMS buffer manager. We propose three alternatives that differ mainly in the way that they deal with the dirty pages evicted from the buffer pool. We implemented these alternatives, as well another recently proposed algorithm for this task (TAC), in SQL Server 2012 R2, and ran experiments using a variety of benchmarks (TPC-C, E and H) at multiple scale factors. Our empirical evaluation shows significant performance improvements of our methods over the default HDD configuration (up to 8X), and up to a 5X speedup over TAC.

3.1 Introduction

After three decades of incremental changes in the mass storage landscape, the memory hierarchy is experiencing a disruptive change driven by flash Solid State Drives (SSDs). SSDs have already made significant inroads in the laptop and desktop markets, and they are also making inroads in the server markets. In the server markets, their appeal is in improving I/O

performance and reducing energy consumption [25, 45]. The research community has taken note of this trend, and over the past year there has been substantial research on redesigning various DBMS components for SSDs [49, 53, 57, 68].

At first glance, Jim Gray's prediction of "flash is disk, disk is tape, and tape is dead" [35, 36] seems to have come true. However, SSDs are still in the early adoption phase, especially in large data centers [11]. One reason is that SSD storage today is at least an order of magnitude more expensive per byte than hard disk storage. Another related reason is that due to the excellent sequential read performance of modern hard disk drives, higher sequential read performance can be achieved with striped disks at a lower price than with SSDs [65]. These observations suggest that SSDs and disks are likely to coexist in large data centers for a long time [42].

SSDs can be used by a DBMS in a number of ways without modifying the engine code, including storing some of the hot relations/indices [23], storing logs [54], storing temp storage [54], and so forth. These approaches are simple to deploy but have the drawback that decisions are made statically and at a coarse granularity.

Our focus in this chapter is on improving DBMS performance by using SSDs as an extension to a traditional database buffer pool. The decisions on what data to store in the SSD are made at run time and dynamically change as the working set migrates. In addition, decisions are made at the finer granularity of pages, rather than the coarser granularity of tables or files.

We present and explore three design alternatives for this task. We call these alternatives clean-write (CW), dual-write (DW), and lazy-cleaning (LC), respectively. The designs differ in the way they handle dirty pages that are evicted from the memory buffer pool. The CW design never writes dirty pages to the SSDs. The DW and LC designs cache dirty pages in the SSDs. When a dirty page is evicted from the buffer pool, the DW design writes the dirty pages simultaneously to both the database and to the SSD (like a write-through cache), whereas the LC design writes dirty pages to the SSDs first, and then lazily copies pages from

the SSD to the database (like a write-back cache). In addition to these design alternatives, suitable replacement policies are employed to identify and keep the most beneficial pages cached in the SSDs.

Recently, Canim *et al.* [24] proposed a way of using an SSD as a second-level write-through cache. They propose a new method called Temperature-Aware Caching (TAC) as their admission/replacement policy. TAC tracks the accesses to pages at a block level (in their implementation a block is a group of 32 contiguous pages) and records this as the temperature [26] of that block. TAC aims to cache the high temperature blocks in the SSD. Although our design alternatives, like the TAC design, aim to use SSDs as a storage accelerator that bridges the performance gap between memory and disks, our work extends that paper in several ways.

First, we thoroughly explore the space of alternative designs for using SSDs to extend the buffer pool. In 2009, an internal Microsoft SSD committee discussed six alternative designs. After intensive discussions, three designs (CW/DW/LC) were chosen to be implemented. In addition, we also implemented the TAC design [24]. In some cases, one of our designs (LC) outperformed TAC by 5X.

Second, we conduct experiments with a larger hardware configuration that better represents what a typical customer might deploy. While the study by Canim *et al.* [24] used 1 GB memory, a 12 GB SSD, 3 disks, and a 48 GB database (to conduct the TPC-C experiment), we used 20 GB memory, 140 GB SSD, 8 disks, and a 400 GB database. We believe that it is important to investigate performance at this larger scale, for several reasons. One reason is that the DBMS chooses different plans at different scales, so smaller-scale experiments may not be indicative of larger scale experiments. Another reason is that running experiments at a larger scale, while surprisingly painful due to the long running times of test runs, revealed issues that would not appear at smaller scale. These issues include a performance drop due to log-file growth, painfully long rampup/shutdown/checkpoint times, performance variations due to data placement in outer/inner tracks of the disks, and so forth.

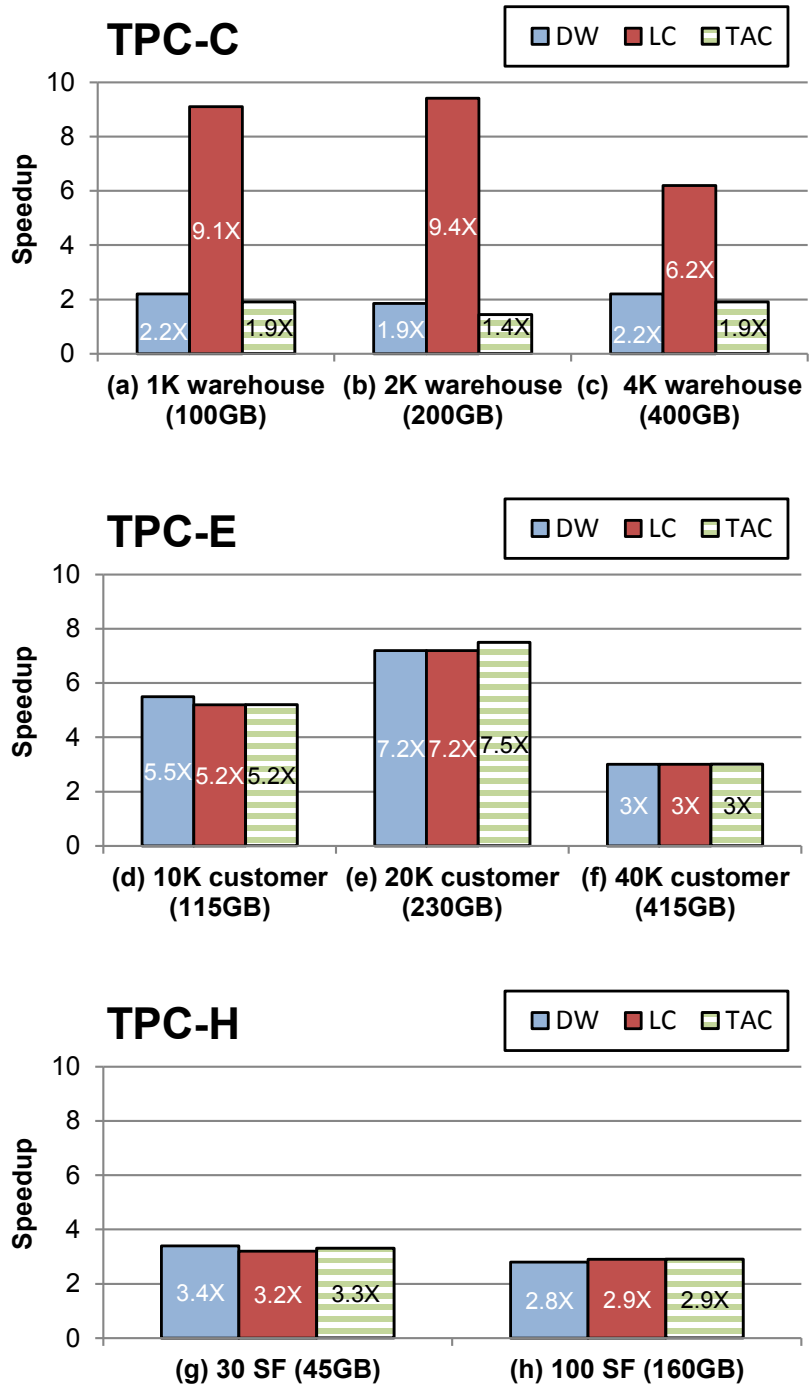


Figure 3.1: Speedups of DW, LC and TAC over the case that does not using any SSDs. We omit the CW results since it always performs worse than DW and LC in our experiments.

Third, we experiment with a broader range of benchmark workloads. Canim *et al.* [6] considered only TPC-C (48 GB) and TPC-H (10 SF) scale factors as their experimental workloads. By contrast, we investigate: TPC-C (100 GB, 200 GB, 400 GB), TPC-E (115 GB, 230 GB, 415 GB), and TPC-H (30 SF, 100 SF). Although TPC-E and TPC-C are both OLTP workloads, they differ from each other substantially in that TPC-E is read intensive while TPC-C is update intensive.

Fourth, re-implementing TAC in a different database management system and experimenting with it on different hardware constitutes a contribution in its own right. The database community has recently identified the lack of experimental reproducibility as a weakness that detracts from the overall credibility of our field. By independently investigating TACs performance we hope to make a small contribution toward rectifying this weakness.

Returning to the SSD designs, our experimental results show that all the SSD designs provide substantial performance benefits for OLTP and DSS workloads, as illustrated by the sample of our results shown in Figure 3.1. For example, on a 2K warehouse (200 GB) TPC-C and a 20K customer (230 GB) TPC-E databases, we observe about 8X performance improvement over the no-SSD performance; for the 2K warehouse TPC-C database, the LC design provides 5.4X better throughput relative to the previous work (TAC).

The remainder of this chapter is organized as follows: our design alternatives are presented in Section 3.2. Section 3.3 describes implementation details and lessons learned during the process of implementing and refining these designs. Experimental results are presented in Section 3.4. Related work is described in Section 3.5, and Section 3.6 contains our summary.

3.2 SSD Design Alternatives

In this section we describe our three SSD designs: the clean-write, the dual-write and the lazy-cleaning designs. We begin by reviewing key aspects of the buffer manager and disk manager in a traditional DBMS, and then explain how our designs fit in with these existing

disk and buffer managers.

3.2.1 Storage Module Overview

In a DBMS, the query execution engine makes calls to the buffer manger to request pages (for reading or writing), and the buffer manger in turn interacts with a disk manager that manages the interface to the physical storage devices. The buffer manager manages an in-memory buffer pool. When a page is requested, the buffer manager first checks the buffer pool, and the page is returned to the caller if the requested page is found. Otherwise, if there is a free frame in the buffer pool, the buffer manager requests that the disk manager read the desired page (from disk) into the free buffer frame. If there is no free frame, the buffer manager picks a victim buffer page based on a replacement policy, and evicts that victim page. If the evicted page is dirty, then the evicted page is written to the disk before its frame is reused. The disk manager handles read and write requests passed to it from the buffer manager, by issuing asynchronous I/Os, to maximize both CPU and I/O resources.

3.2.2 SSD Manager

All of our designs use a modified architecture that employs a new storage module component, called the SSD Manager. The SSD manager interacts with the buffer and disk managers, as illustrated in Figure 3.2. Like the (main-memory) buffer manager that uses a replacement policy to keep the most "desirable" pages in the buffer pool, the SSD manager uses an SSD replacement policy and an SSD admission policy to selectively cache pages in the SSD. The SSD replacement policy is used to determine which page to evict from the SSD when the SSD is full. On the other hand, the SSD admission policy plays a complimentary role and determines when a page should be considered for caching in the SSD. The admission of a page may then trigger an invocation of the replacement policy to evict a page when the SSD is full.

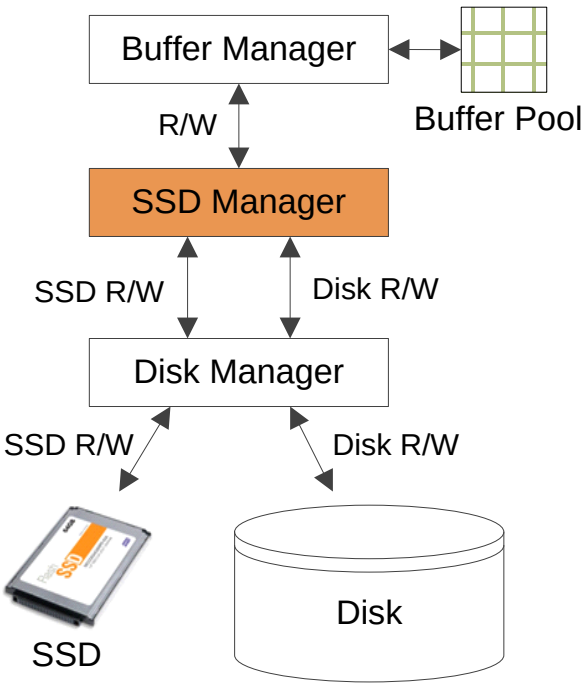


Figure 3.2: The relationship between the buffer, SSD, and disk managers along with the main-memory buffer pool, the SSD storage subsystem, and the disk storage subsystem.

When the buffer manager gets a request for a page, it first examines the main-memory buffer pool for that page. If the page is found in the buffer pool then a handle to the page is returned to the caller. However, if the page is not found in the buffer pool, then the SSD manager first checks if a copy of the page is cached in the SSD. If so, the page is read from the SSD and its usage information is updated (for use by the SSD replacement policy). Otherwise, the page is fetched from the disk.

In the case that a clean page in the main-memory buffer pool is modified (dirtied), if the SSD also has a copy of the clean page, then the copy of the page in the SSD is no longer valid. In this case the SSD manager looks for the copy of the clean page in the SSD, and if found, the page is invalidated and the space on the SSD is reclaimed as free space.

If there are no or few free buffers in the (in memory) free buffer list, the buffer manager picks a (victim) page for eviction. This victim page is evicted to the SSD manager, which in turn makes a decision about caching it in the SSD based on the SSD admission policy

(described below). If the page is admitted for caching, then it is (asynchronously) written to the SSD. Of course, if the SSD is full, the SSD manager chooses and evicts a victim based on the SSD replacement policy before writing the page to the SSD.

The *SSD admission policy* used in this study allows configurations in which the SSD can deliver much higher random performance than a disk (100X in our experimental setup), but provides smaller performance gains for sequential reads I/Os. As mentioned in the introduction, higher sequential read performance is easily and more economically achieved with a small number of striped disks. The SSD manager, therefore, differentiates pages that are read from disk using random accesses (such as through non-clustered index lookups) versus sequential accesses (such as through table scans), and only pages fetched using random I/Os are considered for caching in the SSD.

In general, precisely classifying I/Os into the random and sequential categories is not easy. It becomes more difficult in a multi-user system where concurrent sequential I/Os can interleave [56], and with sophisticated I/O subsystems that optimize the order in which received I/Os are executed [27]. In this study, we take a simple approach that leverages the "read-ahead" mechanism that is often already in place in a modern DBMS e.g. SQL Server. Typical read-ahead mechanisms are already finely tuned to incorporate the complexities mentioned above, and will automatically be triggered during a sequential scan. In our designs, we simply exploit the existing read-ahead mechanism in SQL Server. Thus, when reading a page from disk, the SSD manager marks the page as "sequential" if it is read via the read-ahead method. Otherwise, the page read is marked as "random." As an alternative, we could classify a page as sequential if the page is within 64 pages (512 KB) of the preceding page [60]. Our results indicate that leveraging the read-ahead mechanism is much more effective. For example, when we issued a sequential-read query, we observed that while the read-ahead mechanism was 82% accurate in identifying sequential reads, the method proposed in [60] was only 51% accurate. As the SSD replacement policy, we use LRU-2 [61], which is widely used by commercial DBMSs.

3.2.3 The Three Design Alternatives

Now, we explain the three SSD design alternatives. These designs mainly differ in the way they deal with dirty pages that are evicted from the main-memory buffer pool, and the implications they have for the checkpoint and recovery logic as a result of retaining the dirty pages. To graphically represent the target storage devices for pages evicted from the memory buffer pool, we use a quad chart in the C/D-R/S space, where C, D, R and S stand for "clean", "dirty", "random", and "sequential", respectively. Each page in the main-memory buffer pool is mapped to one quadrant of the chart. Below we use the terms "random" and "sequential" to refer to pages that are accessed from the database stored on disk using random and sequential I/Os respectively.

3.2.3.1 The Clean-Write (CW) Design

The CW design is illustrated in Figure 3.3 (a). With this design, only clean pages are cached in the SSD. Whenever a dirty page is evicted from the main-memory buffer pool it is only written to the disk. Hence, the copy of each page in the SSD is identical to the copy on the disk. This design, consequently, requires no changes to the checkpoint and recovery logic. Since modified pages are never stored in the SSD, CW mainly benefits applications whose working sets are composed of randomly accessed pages that are infrequently updated.

3.2.3.2 The Dual-Write (DW) Design

The DW design shown in Figure 3.3 (b) differs from the CW design in that dirty pages are written both to the SSD as well as back to the database on disk, treating the SSD as a "write-through" cache for dirty pages. As with CW, clean pages that are selected for caching on the SSD are written only to the SSD. Since dirty pages evicted from the main-memory buffer pool are written to the SSD and the disk, the copy of each page in the SSD is identical to the copy on the disk, and therefore, no change to the checkpoint and recovery logic is required.

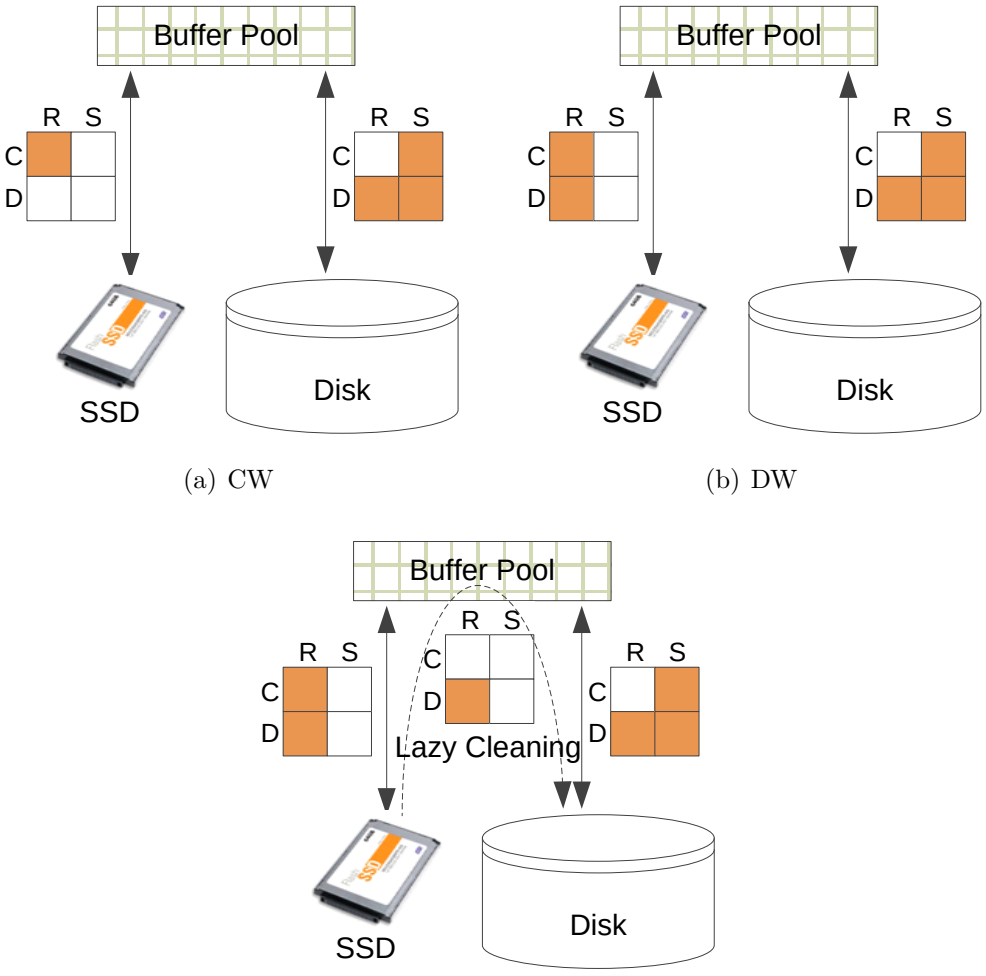


Figure 3.3: Overview of the three SSD design alternatives: (a) the clean-write design, (b) the dual-write design, and (c) the lazy-cleaning design.

3.2.3.3 The Lazy-Cleaning (LC) Design

The LC design depicted in Figure 3.3 (c) differs from the DW design in that dirty pages that are selected for caching on the SSD are written only to the SSD. A background "lazy-cleaning" thread is in charge of copying dirty SSD pages to the disk when they are evicted from the SSD, thus implementing a "write-back" cache. As with CW and DW, clean pages that are selected for caching on the SSD are written only to the SSD. Since the SSD may have the latest copy of a page, there are checkpoint and recovery implications associated with this design.

Since SQL Server 2008 R2 we used in our experiemnts (Section 3.4) relies on a "sharp" check pointing mechanism [37], the LC design requires modifying the checkpoint logic to flush all the dirty pages in the SSD to disk when taking a checkpoint. With very large SSDs this can dramatically increase the time to perform the sharp checkpoint. An advantage of the sharp checkpoint (over the fuzzy checkpoint policy described below) is that restart time (the time to recover from a crash) is fast, and, as a result, the restart time for the LC design is not impacted.

A fuzzy checkpoint policy, on the other hand, does not require that dirty pages in the SSD are forced to disk when a checkpoint is performed. Thus, the impact is fairly low on the time it takes to compute a checkpoint with the LC design. However, the restart time can be larger as the dirty pages that were resident in the SSD buffer pool at the time of the crash must be reconstructed from the log and the copy of the pages in the hard disk drives if the recovery component does not use the pages in the SSD during restart. An interesting direction of future work is to consider the implications of using the pages cached in the SSD during restart.

Therefore, to limit the time required for a checkpoint (in case of the sharp checkpoint policy) and for a system restart after a crash (in case of the fuzzy checkpoint policy), it is desirable to control the number of dirty pages in the SSD. The most "eager" LC method flushes dirty pages as soon as they are moved to the SSD, resulting in a design that is similar

to the DW design. At the other extreme, in the "laziest" LC design, dirty pages are not flushed until there is no SSD space for new pages. This latter approach has a performance advantage for workloads with repeated re-accesses to relatively "hot" pages that are dirtied frequently. However, delaying these writes to disk for too long can make the recovery time unacceptably long.

To mitigate this effect, the behavior of the background LC thread is controlled using a threshold parameter λ (expressed as a % of the SSD capacity). When the number of dirty pages in the SSD is above this threshold λ, then the LC cleaner thread wakes up and flushes pages to the disk so that the number of remaining dirty pages is slightly below this threshold (in our implementation about 0.01% of the SSD space below the threshold). In the evaluation section (Section 3.4) we investigate the effect of this tunable parameter.

3.2.3.4 Discussion

Each of the three designs has unique characteristics. The CW design never writes dirty pages to the SSD, and therefore, its performance is generally worse than the DW and LC designs unless the workload is read-only, or the workload is such that updated pages are never re-referenced. It is, however, simpler to implement than the other two designs because there is no need to synchronize dirty page writes to the SSD and the disk (required for DW), or to have a separate LC thread and to change the checkpoint module (required for LC).

The DW and LC designs differ in how they handle dirty pages that get evicted from the main-memory buffer pool. The LC design is better than DW if dirty SSD pages are re-referenced and re-dirtied, because such pages are written to and read back from the SSD multiple times before they are finally written back to disk, while DW writes such pages to disk every time they are evicted from the buffer pool. The DW design, on the other hand, is better than LC if the dirty SSD pages are not re-dirtied on subsequent accesses. Since pages cannot be directly transferred from the SSD to the disk , pages being copied from the SSD back to disk must first be copied from the SSD into main-memory and then copied

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6
Main Memory	P	P′			P	P′
SSD			P	P′	P	P′
Disk	P	P	P	P	P	P

Figure 3.4: Six possible relationships amongst (up to) 3 copies of a page in memory, SSD, and disk. P denotes a copy of a page, and P' denotes a modified version of the page, so P $\neq$ P' (P' is newer).

from main-memory to disk. Thus, DW saves the cost of reading the dirty pages from the SSD into the main-memory if they are not dirtied subsequently. In addition, DW is easier to implement than LC since it does not require an additional cleaner thread and does not require changing the checkpointing and recovery modules.

Note that the DW and LC designs obey the write-ahead logging (WAL) protocol, forcibly flushing the log records for that page to log storage before writing the page to the SSD.

With the data flow scheme explained in Section 3.4, there may exist up to three copies of a page: one in memory, one in the disk, and one in the SSD. Figure 3.4 illustrates the possible relationships among these copies. First, if a page is cached in main-memory and disk, but not the SSD, the memory version could be the same as the disk version (case 1), or newer (case 2). Similarly, if a page is cached in the SSD and disk, but not in the memory, the SSD version is either the same as the disk version (case 3), or newer (case 4). When a page is cached in all three places, the main-memory version and the SSD version must be the same because whenever the memory version becomes dirty, the SSD version, if any, is immediately invalidated. As a result, the two cases where the memory/SSD version is the same as the disk version (case 5) or newer (case 6) are possible. As might be noticed, while all the cases apply to the LC design, only case 1, 2, 3, and 5 are possible for the CW and DW designs.

3.2.4 Comparison with TAC

In this section we present a brief comparison of our designs (CW, DW, and LC) and TAC [24], which is intended to highlight the differences in terms of data flows.

The data-flow scheme used in our designs differs from TAC in the following ways: First, when a page that is cached in both the main-memory and the SSD is modified, TAC does not immediately reclaim the SSD space that contains the copy of the page, while our designs do. Hence, with update-intensive workloads, TAC can waste a fraction of the SSD. As an example, with the 1K, 2K and 4K warehouse TPC-C databases, TAC wastes about 7.4 GB, 10.4 GB, and 8.9 GB out of 140 GB SSD space to store invalid pages.

Second, for each of our three designs clean pages are written to the SSD only after they have been evicted from the main-memory buffer pool. The TAC design, on the other hand, attempts to writes pages to the SSD immediately after they are read from disk. We have observed that this approach (at least in our implementation of TAC) can create latch contention between the thread that is trying to write the page to the SSD and the other thread that wants to use the page to process transactions. Once the page is placed in the main-memory, if the former thread grabs the latch on the page, forward processing (aka the second thread) can be delayed since pages cannot be used while they are being read/written. Consequently, the latch wait times for page latches are higher with TAC. For example, with the TPC-E workloads we have observed that TAC has about 25% longer average page latch wait times than our designs.

3.3 Implementation Details

In this section, we describe the implementation details of the SSD manager (the shaded component in Figure 3.2). We first introduce the data structures required for the three design alternatives, and then present some of the optimization techniques and lessons learnt during the course of designing, implementing, and evaluating these designs.

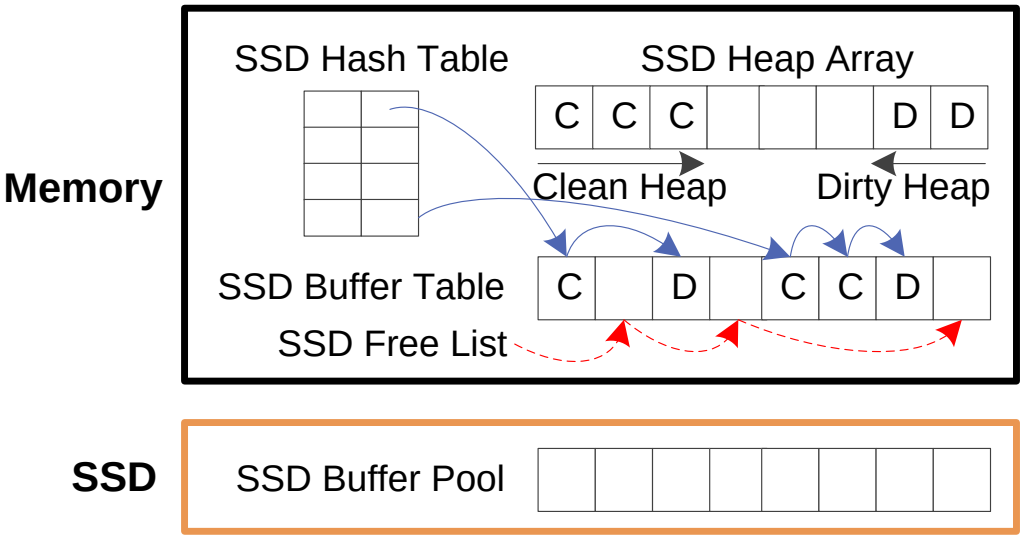


Figure 3.5: The data structures used by the SSD Manager.

3.3.1 Data Structures

The SSD manager uses the following five data structures as illustrated in Figure 3.5.

- The **SSD buffer pool** is an array of S frames that are page-sized regions in which the database pages are cached. It resides on the SSD as a big file. This file is created on the SSD when the DBMS starts for the first time.

- The **SSD buffer table** is an array of S records corresponding to the frames in the SSD buffer pool. Each record has a page id, a dirty bit that describes the state of the matching page stored in the SSD, the last two access times, a latch, and some pointers such as the "next-free pointer" used in the SSD free list (to be described below). The size of each record is 112 bytes in our implementation.

- To facilitate fast lookups, the records in the SSD buffer table that have the same hash index (transformed from their corresponding page IDs) are organized into a linked list (see the solid blue arrows in Figure 3.5). A hash table, called the **SSD hash table**, is used to point to the heads of such lists.

- The records in the SSD buffer table that correspond to free frames in the SSD buffer pool are organized into a linked list called the **SSD free list** (see the dashed red arrows.)

- The SSD manager has to find a victim page when the SSD is full or when cleaning dirty pages (in LC). To quickly identify a victim page, an **SSD heap array** is used. This heap has nodes that point to SSD frames in the SSD buffer table, and is used to quickly determine the oldest page(s) in the SSD, as defined by some SSD replacement policy (LRU-2 in our case). This SSD heap array is divided into two regions: the clean and dirty heaps. The **clean heap** stores the root (the oldest page that will be chosen for replacement) at the first element of the array, and grows to the right. The **dirty heap** stores the root (the oldest page that will be first "cleaned" by the LC thread) at the last element of the array, and grows to the left. This dirty heap is used only by LC for its cleaner thread and for checkpointing (CW and DW only use the clean heap).

3.3.2 Checkpoint Operations

SQL Server 2008 R2 we used in all experiemnts employs sharp checkpoints. Thus, in case of the LC design, all the dirty pages in the SSD are flushed to the disk during a checkpoint. Note that during a checkpoint, LC stops caching new dirty pages evicted from the main-memory buffer pool in order to simplify the implementation.

In the DW design, during a checkpoint operation, dirty pages in the main-memory buffer pool that are marked as "random" are written both to the disk and to the SSD (as opposed to writing only to the disk). Although this technique extends the DW policy, in which only evicted pages, and not checkpointed pages, are flushed to the SSD, this modification has the advantage of potentially filling up the SSD faster with useful data.

3.3.3 Optimizations

Our designs also incorporate a number of optimizations to the basic design, as described below.

3.3.3.1 Aggressive Filling

To prime the SSD with useful data when a DBMS is first run (cold start), we use a technique called "aggressive filling." With this technique, all pages that are evicted from the main-memory buffer pool are cached in the SSD until the SSD is filled up to a certain threshold (τ). After this threshold is reached, only pages that are "qualified" for caching, as dictated by the admission policy, are stored in the SSD. The same technique with different filling threshold is used in [24] and also incorporated in our TAC implementation.

3.3.3.2 SSD Throttle Control

The SSD can become a bottleneck under high load. So we continuously monitor the SSD queue length to limit the load on the SSD. With this technique, no additional I/Os are issued to the SSD when the number of pending I/Os to the SSD is more than a certain threshold (μ). In other words, all I/Os bypass the SSD, and directly read/write pages to/from the disk. As an exceptional case, however, when the SSD version of a requested page is newer than the disk version (only possible in LC), the page is read from the SSD for correctness. The same technique is incorporated in our TAC implementation.

3.3.3.3 Multi-page I/O

When reading multiple data pages that are consecutive on the disk, the I/O manager can optimize these requests and issue only one I/O call. In our designs, the individual pages in a read request are examined to see which pages are in the SSD. At first, when we implemented our designs, we broke the request into a split set of reads. Consider, for example, a request to read 6 consecutive pages. If the 3^{rd} and the 5^{th} pages were in the SSD, five separate

reads were performed (3 reads to disk and 2 read to the SSD). Unfortunately this degraded performance since the disk can handle a single large I/O request more efficiently than multiple small I/O requests.

Thus, to avoid breaking a single multi-page read into multiple smaller reads, only the leading and trailing pages from the requested list of pages (in the I/O request call) are trimmed if they are in the SSD, and the remaining consecutive pages are read from the disk in one I/O. In the case that some of the pages in the middle of the disk read request have newer SSD versions than the disk versions, additional read I/Os are issued to read them from the SSD, and the older disk versions are removed from the memory as soon as the disk read completes.

3.3.3.4 SSD Partitioning

When one thread modifies an SSD data structure, other threads that try to access the same data structure must be blocked (via a latch wait). To increase concurrency, the SSD buffer pool is partitioned into N pieces, where N is the number of CPU hardware threads. Different partitions share the SSD hash table, but each partition has its own segment of the SSD buffer table and SSD heap array. This SSD partitioning increases concurrency, because when one segment of the SSD heap array is latched for reorganization, other segments can still be accessed.

3.3.3.5 Group Cleaning in LC

As mentioned in Section 3.2.3.3, the LC thread is invoked when the number of dirty pages in the SSD is more than a threshold (λ). In our implementation of this LC thread, it gathers up to α dirty SSD pages with consecutive disk addresses, and writes them to the SSD using a single I/O operation. The purpose of this technique is to reduce the number of I/O calls that are made to the disks.

READ	Random	Sequential
8 HDDs	1,015	26,370
SSD	12,182	15,980

WRITE	Random	Sequential
8 HDDs	895	946
SSD	12,374	14,965

Table 3.1: Maximum sustainable IOPS for each device when using page-sized (8KB) I/Os. Disk write caching is turned off.

3.4 Evaluation

In this section, we compare and evaluate the performance of the three design alternatives using various TPC benchmarks.

3.4.1 Experimental Setup

3.4.1.1 H/W and S/W Specifications

We implemented the three SSD design alternatives (the CW, DW, and LC designs) and the TAC design (TAC) [24] in SQL Server 2008 R2, and compared these with the case of not using the SSD at all (noSSD).

For the evaluation, we performed experiments on a 2.27 GHz Intel dual CPU quad-core Xeon processor (Nehalem) running 64-bit Windows Server 2008 R2 with 24 GB of DRAM. The databases, including TempDB, were created on file groups striped across eight 1 TB 7,200 RPM SATA hard disk drives (HDDs). We used two additional HDDs - one disk was dedicated to the OS and the other for the DBMS log storage. For the SSD buffer pool, we used a 160GB SLC Fusion ioDrive. In all experiments, 20 GB of DRAM was dedicated to the DBMS, and 140 GB of the SSD was used for the SSD buffer pool.

To aid the analysis of the results that are presented below, the I/O characteristics of the SSD and the collection of the 8 HDDs are given in Table 3.1. All IOPS numbers were obtained using Iometer [3]. In all the experiments, we used the optimization techniques described in Section 3.3.3 (for both our designs and TAC). Table 3.2 shows the parameter values used in our evaluations.

Symbol	Description	Values
τ	Aggressive filling threshold	95%
μ	Throttle control threshold	100
N	Number of SSD partitions	16
S	Number of SSD frames	18,350,080 (140 GB)
α	Maximum allowable number of dirty SSD pages gathered in an LC write request	32
λ	Dirty fraction of SSD space	1% (E, H), 50% (C)

Table 3.2: Parameter values used in our evaluations.

In all the experiments, the CW design performs worse than the DW and LC designs. For example, for the 20K customer TPC-E database, CW was 21.6% and 23.3% slower than DW and LC, respectively. Thus, in the interest of space, we omit additional CW results in the remainder of this section.

3.4.1.2 Datasets

To investigate and evaluate the effectiveness of the SSD design alternatives, we used various OLTP and DSS TPC benchmark workloads, including TPC-C [7] (the original OLTP benchmark), TPC-E [8] (the new TPC OLTP benchmark) and TPC-H [9] (the standard DSS benchmark).

One observation in our evaluation is that (as expected) the overall performance heavily depends on the relationship between the working set size of the database, the DBMS buffer pool size, and the SSD buffer pool size. Thus, we designed the experiments to cover the following interesting cases: (a) when the database size is smaller than the SSD buffer pool size, (b) when the working set size is close to the SSD buffer pool size, and (c) when the working set size is larger than the SSD buffer pool size. Based on this design, for TPC-E we chose the 10k, 20k and 40k customer databases (corresponding to 115 GB, 230 GB and 415 GB, database sizes respectively). For TPC-C, we selected the 1K, 2K and 4K warehouse databases with database sizes of 100 GB, 200 GB and 400 GB, respectively. For TPC-H, we

only used two scale factors: 30 and 100, corresponding to database sizes of 45 GB and 160 GB, respectively. Note these scale factors are "officially" allowed by the TPC-H specification. The next allowed scale factor (300 SF) requires more than a week of run time for some data points with our hardware, and we omitted this run; as shown below, the trend with TPC-H is fairly clear with the two scale factors that we ran.

For each workload, we used different performance metrics according to the TPC specifications. For TPC-C and TPC-E, we measured the number of new orders that can be fully processed per minute (tpmC), and number of (Trade-Result) transactions executed within a second (tpsE), respectively. For TPC-H, we measured elapsed wall-clock time to complete the Power and the Throughput Tests, and transformed it to the TPC-H Composite Query-per-Hour Performance Metric (QphH). After each execution, we restart the DBMS to clean up the main-memory and SSD buffer pools.

Since SQL Server 2008 R2 uses "sharp checkpoints" it flushes the dirty pages in the buffer pool to the disk during a checkpoint. To isolate the effects of checkpointing from the main goal of this empirical evaluation (namely, to quantify the benefits of the various SSD caching designs), we set the interval between checkpoints to a large number for the update heavy TPC-C benchmark. Effectively, this setting turns off checkpointing.

In general we saw a significant reduction in performance (about 2X or more) when using checkpointing with TPC-C. This observation also highlights an important direction for future work: using the contents of the SSD during recovery task, thereby requiring a sharp checkpoint to only flush the dirty pages in the main-memory. In addition, to improve performance it may be enough to flush these pages to the SSD instead of to the disks. In principle this modification can be achieved by adding the SSD buffer table data structure, shown in Figure 3.5, along with a few other pieces of metadata information, to the checkpoint record. Exploring the implication of this with respect to fuzzy and sharp checkpoint performance is beyond the scope of this work.

While checkpointing was effectively turned off for the TPC-C experiments, for the TPC-E

and H experiments the recovery interval was set to 40 minutes. With this setting, the DBMS issues a checkpoint roughly every 40 minutes.

3.4.2 TPC-C Evaluation

In this section, we evaluate the effectiveness of our SSD designs when running TPC-C. Each algorithm was run for 10 hours and we measured the tpmC as required by the benchmark. For LC, the parameter λ was set to 50%, meaning that the LC thread starts working once the number of dirty SSD pages exceeds 50% of the SSD buffer pool.

Figures 3.1 (a) – (c) presents the steady-state throughput speedups of the SSD designs (DW, LC and TAC) relative to the noSSD case on the 1K, 2K and 4K warehouse databases, respectively. The speedups are calculated with the average throughputs over the last two hours of executions. Note that published TPC benchmark results use this method of reporting the average steady state behavior.

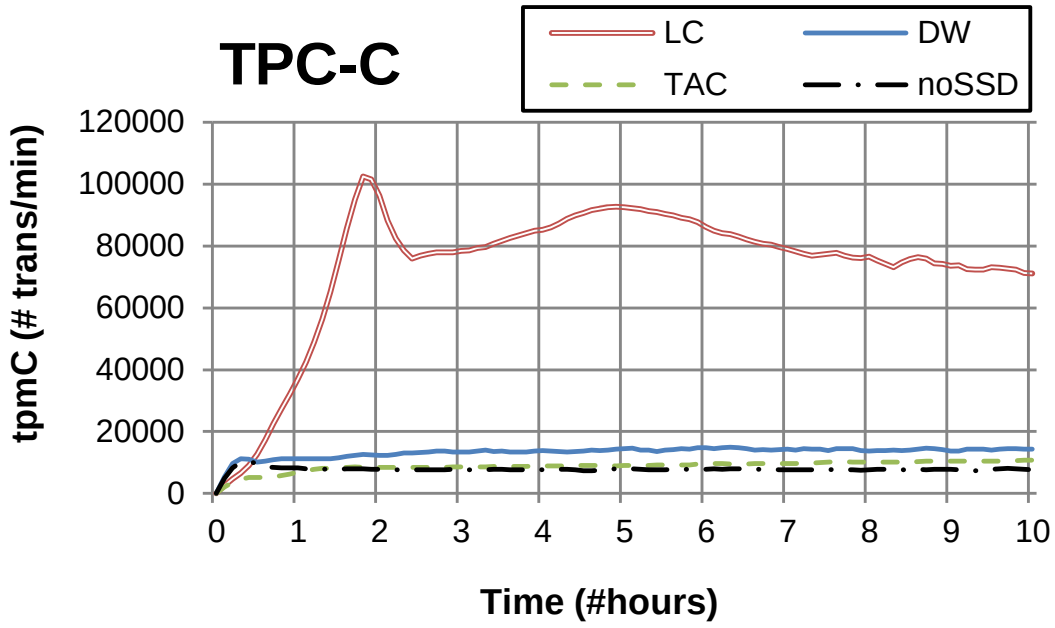
As can be seen in Figures 3.1 (a) – (c), LC clearly outperforms all the methods for each TPC-C configuration. LC is 8X faster for the 2K warehouse database over the noSSD case (while LC read pages from the SSD at the spped of 74.2 MB/s with 89% SSD hit ratio, the read speed of the 8 disks was only 6.7 MB/s in the noSSD case). LC provided 4.3X and 5.4X better throughput relative to DW and TAC, respectively. For all the databases, DW also performed better than TAC, but worse than LC.

The write-back approach (LC) has an advantage over the write-through approaches (DW and TAC), if the dirty SSD pages are frequently re-referenced and re-dirtied. The TPC-C workload is highly skewed (75% of the accesses are to about 20% of the pages) [55], and is update intensive (every two read accesses are accompanied by a write access). Consequently, dirty SSD pages are very likely to be re-referenced, and as a result, the cost of writing the pages to the disk and reading these pages back from the disk is saved with LC. For example, with the 2K warehouse, with the LC design, about 83% of the total SSD references are to dirty SSD pages. Thus for this update-intensive workload, the SSD design adopting a

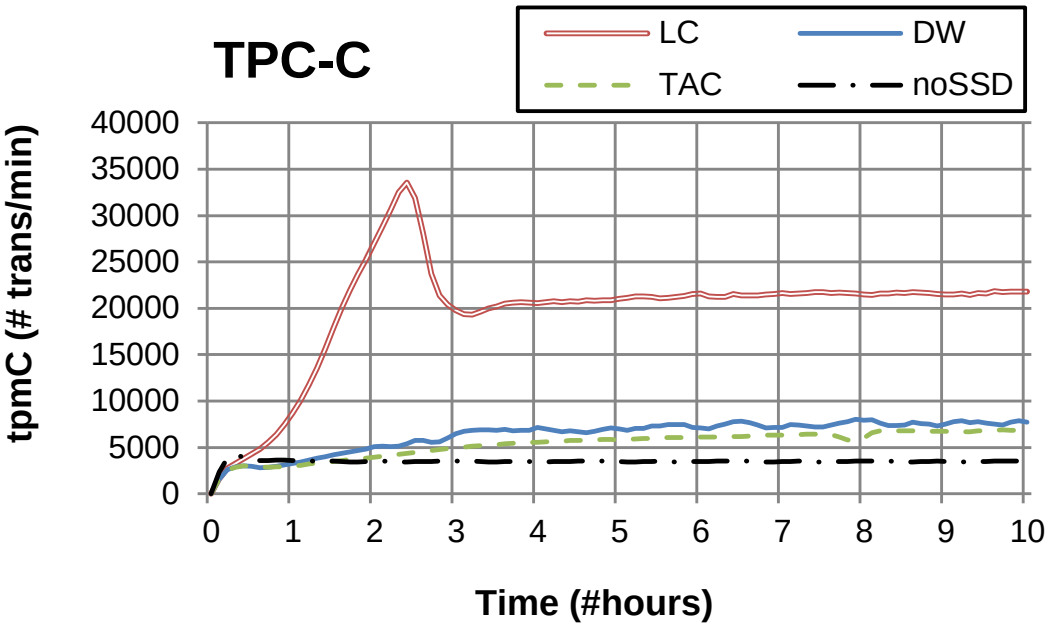
write-back policy (such as LC) works far better than a write-through policy (such as DW and TAC).

While DW and TAC are both write-through approaches, DW performs better than TAC for TPC-C. We suspect that the main reason lies in the difference in page flows. In TAC, a page is written to the SSD under two circumstances: (a) right after being loaded from the disk; and (b) upon eviction from the buffer pool, if the page is dirty, and if an invalid version of the page exists in the SSD. In a system (such as SQL Server) that uses asynchronous I/Os, there may exist a gap between the time a page has been loaded from the disk and the time the page starts to be written to the SSD. If a forward-processing transaction grabs the latch to the page and dirties it first, TAC cannot write the page to the SSD because that would make the SSD version of the page newer than the disk version (which conflicts with its write-through policy). As a result, when this dirty page is later evicted from the main-memory buffer pool, TAC does not write it to the SSD because there is not an invalid version of the page in the SSD. A similar problem occurs when some dirty pages are generated on-the-fly, e.g. during B+ tree page split. TAC does not write such pages to the SSD because they were not read from the disk in the first place. The DW design does not have these problems as it writes the dirty pages to both the SSD and the disk when they are evicted from the buffer pool. This difference has a bigger impact for TPC-C than for TPC-E or TPC-H, because TPC-C is update intensive.

Another reason why DW may be better than TAC is that TAC wastes some SSD space to store the invalid pages. When a page in the main-memory buffer pool is modified, DW uses a physical invalidation mechanism (the SSD frame is appended to the free list), whereas TAC uses a logical invalidation mechanism (the SSD frame is marked as invalid but is not released). Consequently, after the SSD is full, TAC may choose a valid SSD page for replacement, even though some invalid SSD pages exist.



(a) 2K warehouses (200 GB)



(b) 4K warehouses (400 GB)

Figure 3.6: 10-hour test run graphs with the TPC-C databases: (a) 2K warehouses and (b) 4K warehouses, respectively.

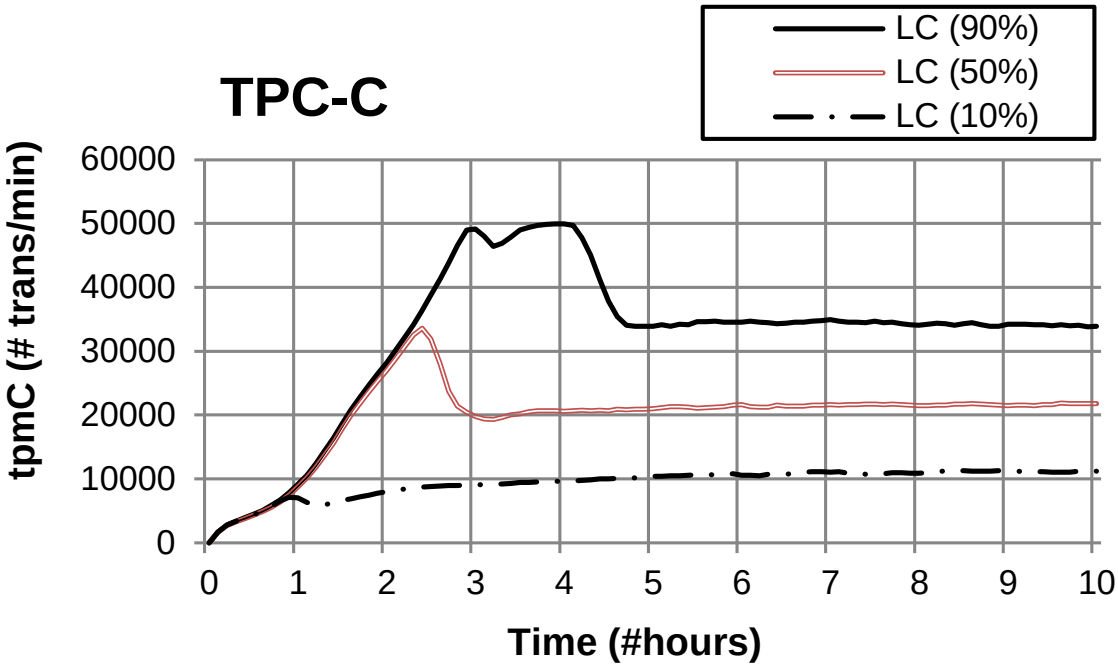


Figure 3.7: The effect of allowing more dirty pages in the SSD buffer pool with the TPC-C 4k warehouse database.

3.4.2.1 Ramp-Up Period

To analyze the behavior of each SSD design during the full 10-hour execution (recall Figures 3.1 (a) – (c) are the average throughput over the last hour), we present the test run graphs of six-minute average throughput versus elapsed wall clock time in Figures 3.6 (a) and (b). To conserve space, we only present the results for the 2K and 4K warehouse databases. To make the graphs more readable, each curve is smoothed using the moving average of three adjacent data points.

As can be observed in Figures 3.6 (a) and (b), the performance of the LC design drops significantly after 1:40 hours and 2:30 hours with the 2K (Figure 3.6 (a)) and 4K (Figure 3.6 (b)) warehouse databases, respectively. Before these drops, the number of dirty SSD pages was lower than the 50% threshold (λ), so the SSD could completely absorb the dirty pages being evicted from the main memory to the disk. Once the number of dirty pages in the SSD is greater than λ, the LC thread kicks in and starts to evict pages from the SSD to the disk

(via a main-memory transfer as there is not direct I/O from the SSD to the disks). As a result, a fraction of the SSD and disk bandwidth is consumed by the LC thread, resulting in less SSD and disk bandwidths being available for the normal transaction processing.

To evaluate the impact of the parameter λ on the LC design, we tested LC with different dirty fractions (λ = 10%, 50%, 90%). As illustrated by Figure 3.7, the speedup of LC increases as more dirty pages are allowed in the SSD. For example, with λ = 90%, LC improves performance by 3.5X and 1.3X (steady-state throughput) over the 10% and 50% cases, respectively. Larger values of λ allow LC to cache more dirty pages, and thereby issue fewer I/Os to evict dirty SSD pages. Since the TPC-C workload has a high re-reference and update rates, by keeping an evicted (from the main-memory buffer pool) dirty page in the SSD for a longer period of time, the LC design is likely to absorb more read/write requests associated with the dirty pages. For example, while the LC thread with λ = 90% issued about 470 IOPS to the disks, about 808 and 726 IOPSs were issued for the 10% and 50% cases, respectively. Thus with higher λ settings, LC can use more of the SSD and disk bandwidths to process transactions, resulting in higher overall transaction throughput rates.

3.4.3 TPC-E Evaluation

This section presents the results of running the TPC-E benchmarks for 10 hours. Checkpointing was turned on (triggered roughly every 40 minutes), and the LC thread was set to start working once the number of dirty SSD pages exceeded 1% of the SSD space (we varied this λ parameter and it did not have a significant impact). The results for the TPC-E runs are summarized in Figures 3.1 (d) – (f).

As can be seen in Figures 3.1 (d) – (f), all the designs provide significant (and similar) performance gains over the noSSD case, about 5.5X, 7.2X, and 3X on the 10K, 20K and 40K customer databases, respectively. Notice that compared to the TPC-C results, now the advantage of LC over DW and TAC is gone. This is because, while the TPC-C workload is update-intensive and highly skewed, updates constitute a lower fraction of the TPC-E

workload.

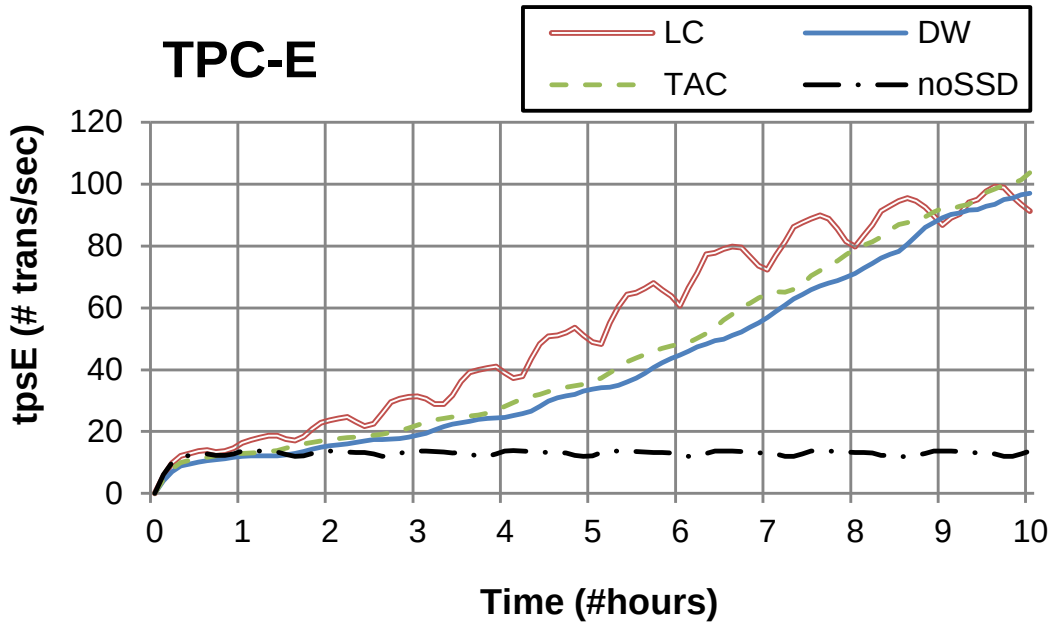
Comparing TAC with DW, we notice that with TPC-E the performance gain with DW is relatively lower. This again is due to the fact that there are less updates. So the chance that a page loaded from the disk is dirtied before writing to the SSD is smaller.

Also notice in Figures 3.1 (d) – (f) that the performance gains are the highest with the 20K customer database. In case of the 20K customer database, the working set nearly fits in the SSD. Consequently, most of the random I/Os are offloaded to the SSD, and the performance is gated by the rate at which the SSD can perform random I/Os. As the database size decrease/increase, however, we observe reduced performance gains for all designs. As the working set becomes smaller, more and more of it fits in the memory buffer pool, so the absolute performance of the noSSD increases. If the working set gets larger, a smaller fraction of it fits in the SSD, so although there is a lot of I/O trffic to the SSD, it can capture an increasingly smaller fraction of the total traffic. Thus the SSD has an increasingly smaller impact on the overall performance, and the performance is gated by the collective random I/O speed of the disks.

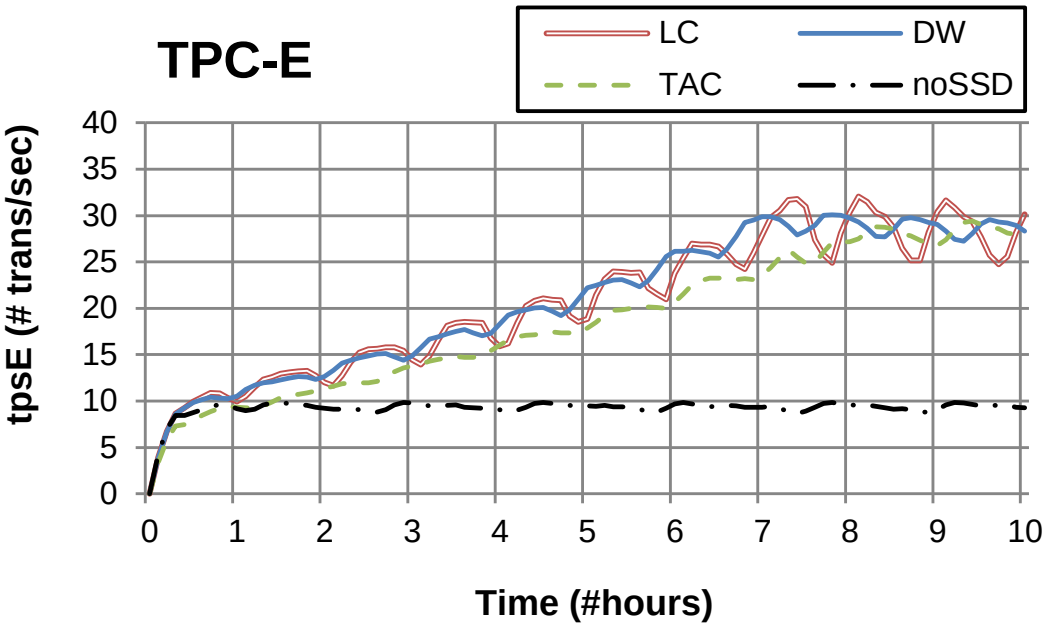
3.4.3.1 Ramp-Up Period

As with TPC-C (cf. Section 3.4.2.1) we now show the full 10-hour run for TPC-E. Again in the interest of space we only show the results with the 20K and the 40K customer databases. These results are shown in Figures 3.8 (a) and (b) respectively.

As can be seen in Figures 3.8 (a) and (b), the "ramp-up" time taken for the SSD designs is fairly long (and much longer than with TPC-C). In the case of the DW design, steady state was achieved only after about 10 and 7 hours with the 20k and 40k customer databases, respectively. The reason for this long ramp-up time stems from the poor random-read bandwidth of the disks. During most of the ramp-up period, the SSD buffer pool is continually getting populated with pages that are evicted from the memory buffer pool because of the aggressive filling technique (Section 3.3.3.1). This means that the speed with



(a) 20K customers (230 GB)



(b) 40K customers (415 GB)

Figure 3.8: 10-hour test run graphs with the TPC-E databases: (a) 20K customers and (b) 40K customers, respectively.

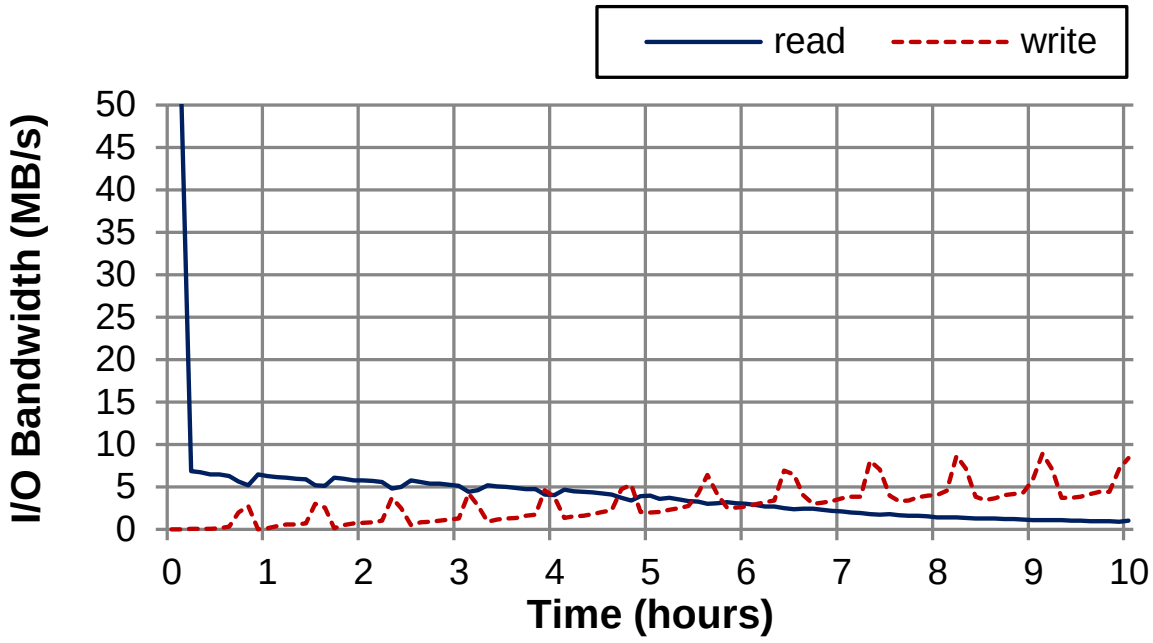
which the SSD can be populated directly depends on the rate at which pages are evicted from the main-memory buffer pool, which in turn is constrained by the speed of reading pages (not in the SSD) from the disks using random accesses (as most of the misses tend to result in an random disk I/O).

Note that the ramp-up time with the 40K customer database is shorter compared to the 20K customer database. With the 20K customer database the working set nearly fits in the SSD, which results in repeated re-references with updates to the SSD pages during the ramp-up period. When this happens, first the page in the SSD must be invalidated. Later, when that page is evicted from the main-memory buffer pool it is written back to the SSD. Thus, during the ramp-up period a sizable chunk of the disk I/O bandwidth is taken up by these re-references, and as a result filling up the SSD with new page is slower.

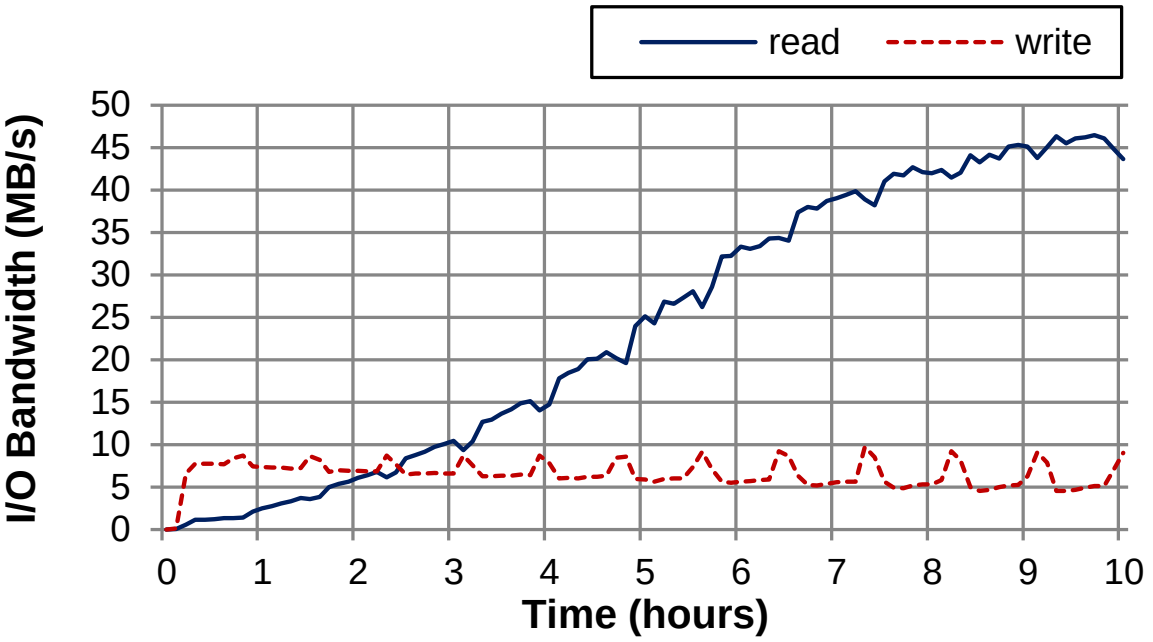
3.4.3.2 I/O Traffic

Next we dig deeper into the dynamic I/O characteristics during the entire run of the TPC-E benchmark to better understand the traffic that both the disks and the SSD see during the run of the benchmark. In the interest of space, we only focus on the 20K customer database with DW as the insights from this case is fairly representative of what happens with the other datasets and the other SSD designs.

Figures 3.9 (a) and (b) show the read and write traffic to the disks and to the SSD, respectively. First, we notice that in Figure 3.9 (a), initially the disks start out by reading data a 50 MB/s, but drastically dropped to 6 MB/s. This is due to SQL Server's feature that before the buffer pool is filled up, every single-page read request is expanded to an eight-page read request. After the buffer pool fills up, pages started to get evicted from the buffer pool, and the disk write traffic starts to go up as the new pages that are brought in result in evictions of existing pages. Notice also that in Figure 3.9 (b), the SSD read traffic steadily goes up till steady state is achieved (at 10 hours) - this is the time it takes to fill up the SSD. In both figures, notice the spikes in the write traffic, which is the I/O traffic



(a) disks



(b) SSD

Figure 3.9: I/O traffic to the disks and SSD on the TPC-E 20K customer database with the DW design.

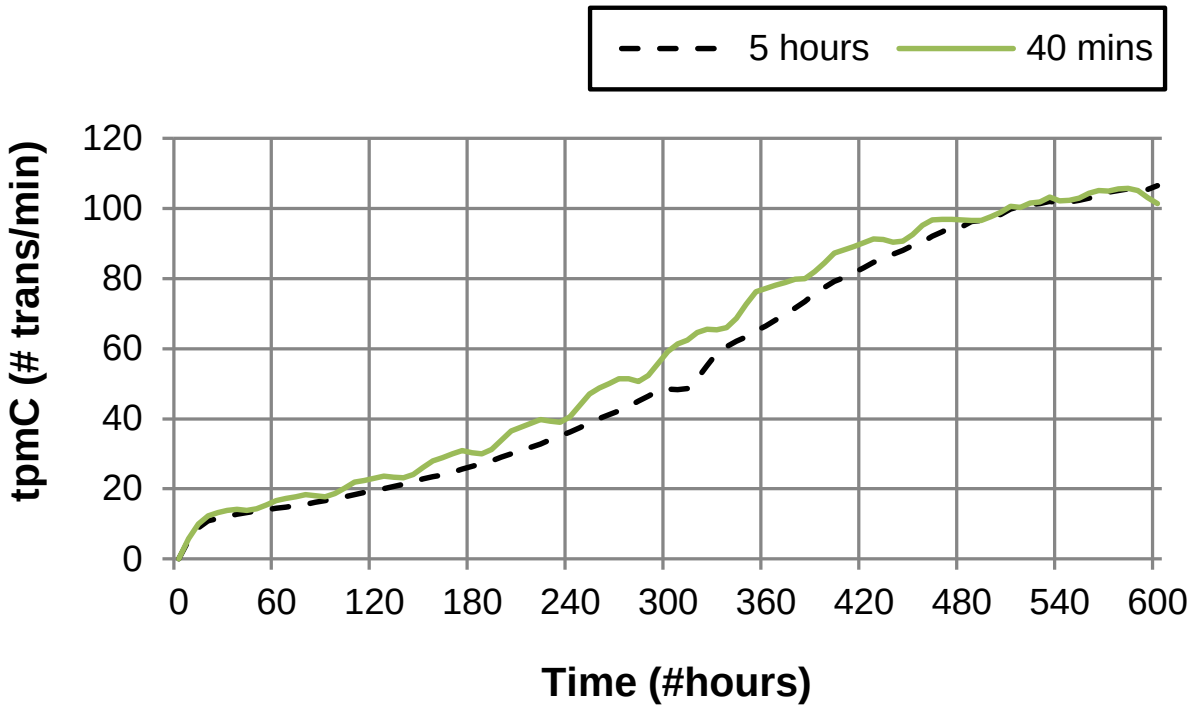
generated by the checkpoints.

In steady state (which is after 10 hours) the bottleneck is the aggregated (random) traffic to the disks, which is 6.5 MB/s, or 832 IOPS, close to the maximum that the disks can collectively support (See Table 3.1). Also notice that the SSD bandwidth is not saturated in the steady state. At most, the SSD sees 46 MB/s of read traffic and 9.7 MB/s of write traffic, but it can support 12,182 IOPS (= 95 MB/s) with 1 page reads (similar for 1 page writes, see Table 3.1). This observation indicates that a high quality SSD does not always guarantee the maximum performance improvement with TPC-E as long as the disk subsystem is a bottleneck.

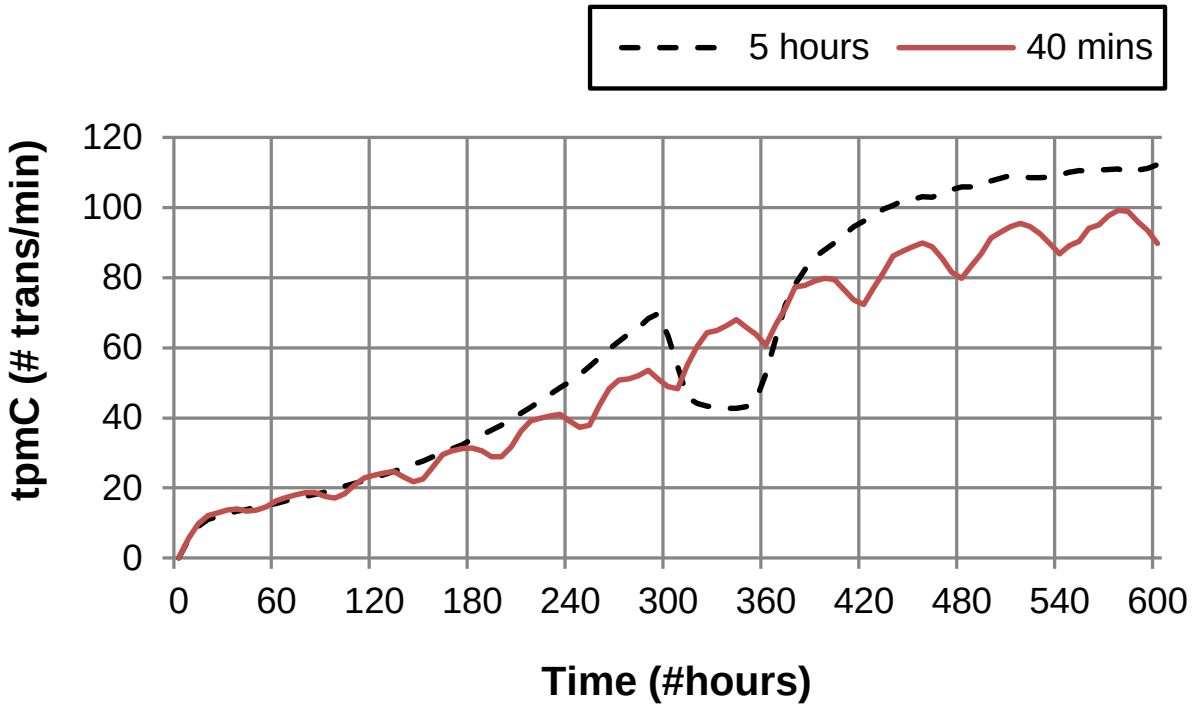
3.4.3.3 Effects of Checkpointing

In Figures 3.8 (a) and (b), we observed that all the designs (including the noSSD) have periodic dips in performance because the checkpointing method is invoked roughly every 40 minutes. The reason for this behavior is that when taking a checkpoint, the DBMS flushes all the dirty pages in the main-memory buffer pool to the disks (i.e., sharp checkpoint), which in turn results in a performance drop. Amongst the SSD designs, checkpointing has a larger effect on LC (the dips are sharper and longer) than DW and TAC because during a checkpoint, LC has to write all the dirty pages in the SSD buffer pool to the disks as well.

Next, we wanted to examine what would happen if we used a very large checkpoint interval (5 hours) to get a perspective of the "pure" performance of the SSD designs with TPC-E. Indirectly, this experiment is also interesting as it provides insight into the performance impact if one were to change the actual checkpointing code to leverage using the contents of the SSD for recovery. In this experiment, we only consider the DW and LC designs, since the previous results show that DW and TAC have similar checkpointing behaviors. In addition, the "dirty fraction" for LC (λ) is increased from 1% to 50% to allow the SSD to accumulate more dirty pages, thereby not incurring additional disk I/Os to the disks to stay below the 1% threshold.



(a) DW



(b) LC

Figure 3.10: Effect of increasing the checkpoint interval on the TPC-C 20K customer database for (a) DW and (b) LC.

The results for this experiment are shown in Figures 3.10 (a) and (b) along with those of the default checkpoint interval (40 minutes). Comparing the DW curves (Figure 3.10 (a)), we notice that for the first 10 hours using a smaller recovery interval improves the overall performance because of the DW checkpoint mechanism, which results in dirty "random" pages being pushed to the SSD during the checkpoint (see Section 3.3.2). Since in the first 10 hours the SSD does not get filled to its maximum capacity, the DW checkpointing mechanism indirectly leads to warming up the SSD. However, after the first 10 hours using a larger recovery interval results in better throughput. The reason for this behavior is that after the first 10 hours, the SSD is filled and writing pages during the checkpoint process may bump out pages from the SSD that may be access subsequently. As a result, performance drops with the shorter recovery interval. In the case of LC (Figure 3.10 (b)), using the larger recovery interval initially (for the first 5 hours) improves performance. But then the checkpointing process kicks in. The checkpointing task here takes much longer to complete than using a smaller checkpoint interval, since it has to flush a fairly large number of pages from the SSD to the disks.

3.4.4 TPC-H Evaluation

Next we evaluate the alternative SSD designs with the TPC-H workload. For this evaluation as required by the benchmark specification [9], we first ran a power test that measures the raw query execution power of a single-user system by running a query steam (the sequential execution of each and every one of the 22 TPC-H queries). Then, we ran a throughput test that measures the ability of a multi-user system by executing some number of query streams in parallel. Before/after each query stream, we inserted/deleted records from the database as specified in the specification. The checkpoint and LC thread configurations are the same as for the TPC-E evaluation (cf. Section 3.4.3).

Figures 3.1 (g) and (h) show the performance speedups of the SSD designs over the noSSD case for the 30 SF and 100 SF databases, respectively. As can be seen in the figures,

TPC-H 30 SF	LC	DW	TAC	noSSD
Power Test	5978	5917	6386	2733
Throughput Test	5601	6643	5639	1229
QphH@30SF	5787	6269	6001	1832

TPC-H 100 SF	LC	DW	TAC	noSSD
Power Test	3836	3204	3705	1536
Throughput Test	3228	3691	3235	953
QphH@100SF	3519	3439	3462	1210

Table 3.3: TPC-H Power and Throughput test results.

the three designs bring significant and similar speedups (about 3X), on both the 30SF and 100SF databases. The reason why the SSD designs bring significant speedups even for the TPC-H workload, which is dominated by sequential I/Os, is that some queries in the TPC-H workload are dominated by key lookups in the LINEITEM table which are mostly random I/O accesses. The reason why the three designs have similar performance is that the TPC-H workload is read intensive.

Table 3.3 shows the detailed results of the TPC-H evaluation. As can be seen in the table, the SSD designs are more effective in improving the performance of the throughput test than the power test, because in the throughput test there are multiple streams which tend to create more random I/O accesses. In the 30SF database, for example, DW is 2.2X better than noSSD for the power test, but is 5.4X better than noSSD for the throughput test.

3.5 Related Work

Using Flash SSDs to extend the buffer pool of a DBMS has been studied in [24, 43, 50]. Canim *et al.* [24] proposed the TAC method and implemented it in IBM DB2. The page flow is as follows:

i Upon a page miss (in the main memory buffer pool), the SSD buffer pool is searched before the disk. That is, if the page is in the SSD, it is loaded from the SSD to the

memory. Otherwise it is loaded from the disk.

ii After reading a page from the disk to the buffer pool, if the page qualifies for the SSD, it is written to the SSD.

iii When a buffer-pool page is updated, the corresponding SSD page, if any, is logically invalidated. That is, the SSD page is marked as invalid but not evicted.

iv When a dirty page is evicted from the buffer pool, it is written to the disk as in a traditional DBMS; at the same time, if the page has an invalid version in the SSD, it is also written to the SSD.

The SSD admission and replacement policies both utilize the "temperature" of each extent (of 32 consecutive disk pages), which is a number indicating how hot the extent is. Initially every extent has temperature=0. Upon every page miss, the temperature of the extent, which contains the page that is accessed, is incremented by the number of milliseconds that can be saved by reading the page from the SSD instead of the disk. Before the SSD is full, all pages are admitted to the SSD. After the SSD is full, a page is admitted to the SSD if its extent's temperature is hotter than that of the coldest page in the SSD. That coldest SSD page is chosen for replacement. Simulation-based evaluations were conducted using TPC-H and TPC-C. Real implementation in IBM DB2 was evaluated using a 5 GB ERP database and a 0.5k-warehouse (48 GB) TPC-C database. While TAC is similar to one of our implemented designs (DW), our work improves upon [24] in several ways. First, we implemented not only the write-through approach (DW), but also the write-back approach, which turns out to up to 5.4X more effective than TAC for TPC-C during forward processing. Second, we conducted experiments in a more realistic setup and therefore the results are more practical to the customers. Third, our scheme is advantageous in performance in a numerous ways, including allowing more dirty pages to be admitted to the SSD, using less memory to store statistics, and allowing a larger fraction of the SSD space to store valid data.

Holloway [43] proposed another SSD buffer-pool extension scheme, which we call the "rotating-SSD design", and implemented it in MySQL. The design was motivated by the relatively slow random write performance of non-enterprise class SSDs. The SSD buffer pool is organized as a circular queue. A logical pointer called nextFrameNo is used to indicate which SSD frame should be allocated next. This pointer will rotate through all frames in the SSD buffer pool sequentially, wrapping around at the end. When a page (dirty or clean) is evicted from the buffer pool, it is written to the SSD frame that nextFrameNo points to. If the frame already contains a valid page, the page is evicted from the SSD first. In case the page to be evicted does not exist in memory and is newer than the disk version, it is written to the disk before being evicted. Note that the eviction of the SSD page stored at nextFrameNo takes place even if the SSD page is frequently accessed. The design sacrifices the quality of the SSD replacement policy to acquire truly sequential write behavior in the SSD. We believe enterprise SSDs suffer less and less from the earlier problem that the random write performance is two to three orders of magnitude worse than the sequential write performance. So the design is less interesting now.

Koltsidas and Viglas [50] proposed three SSD buffer-pool extension designs: the inclusive approach (which is somewhat similar to the TAC's page flow in that a page in the memory buffer pool is included in the SSD), the exclusive approach (where a page never exists in both the buffer pool and the SSD), and the lazy approach (somewhat similar to our LC design). In the exclusive approach, whenever a page is read from the SSD to memory, the SSD version of the page is unnecessarily removed. Later when the page is evicted from the buffer pool, it has to be written to the SSD again. In [50], these three methods were never implemented in a DBMS to characterize the effectiveness of them for different workloads.

Also related is the HeteroDrive system [46], which uses the SSD as a write-back cache to convert random writes to the disk into sequential ones

Instead of using the SSD as a cache, an alternative usage in a DBMS is to replace (some of) the data disks. In the buffer-pool extension case, the permanent "home" of the data

is still the disks. On the contrary, in the disk-replacement approach, the "home" of all or selected data is changed to the SSD. The high cost of SSDs makes it prohibitively expensive to completely replace existing disks with SSDs for very large databases [42, 60]. So an important issue is what data should be migrated to the SSD. Koltsidas and Viglas [49] presented a family of on-line algorithms to decide the optimal placement of a page and studied their theoretical properties. Canim *et al.* [23] presented an object placement advisor that collects I/O statistics for a workload and uses a greedy algorithm and dynamic programming to calculate the optimal set of tables or indexes to migrate.

The effect of using the SSD to replace the storage of the transaction log or the temp space was studied in [54].

Since Jim Gray's 2006 prediction [35] that "tape is dead, disk is tape, and flash is disk", substantial research has been conducted to improve the DBMS performance by using flash SSDs. Some of the work are: revisiting the five-minute rule based on SSD [34], benchmarking the SSD performance [21], examining method for improving various DBMS internals such as query processing [28, 68], index structures [14, 57, 69], and page layout [53].

Recently the industry has released several storage solutions integrated with SSD to accelerate the performance of traditional disk-based storage. Oracle Exadata [63] uses the SSD as a smart cache for frequently accessed disk pages. Teradata Virtual Storage [6] continuously migrates hot data to the SSD. Solaris ZFS file system [19] uses the SSD as a second-level cache that extends the DRAM cache.

3.6 Summary

In this chapter, we evaluated four alternative designs that use the SSD to extend SQL Server's buffer pool, across a broad range of benchmark databases. One of these designs includes a recently proposed design called TAC [24]. In our setting, with these designs we have observed up to 8X speedup for OLTP workloads and up to 3X speedup for OLAP workloads. A number

of factors affect the overall speedup, including the gap in the random I/O performance between the SSD and the disk subsystem, and the ratio of the working set size of a database over the SSD buffer pool size. If the random I/O performance gap increases/decreases, the speedup provided by the SSD designs will increase/decrease as well. The best speedup can be achieved when the working set size is close to the SSD buffer pool size. In our experiments and settings, among the four designs, LC (a write-back cache design) is the best option for update-intensive workloads, as the LC design benefits from allows caching of dirty pages in the SSD and serving up references to these pages from the SSD (rather than the disk). For read-intensive workloads, DW, TAC and LC have similar performance, as they all cache the pages that are randomly accessed in SSD.

Chapter 4

Fast Peak-to-Peak Restart for SSD Buffer Pool Extension

In the previous chapter, we studied a promising usage of Flash SSDs in a DBMS to extend the main memory buffer pool by caching in the SSD selected pages that are evicted from the buffer pool. These schemes have been shown to produce big performance gains in the steady state. Simple methods for using the SSD buffer pool throw away the data in the SSD when the system is restarted (either when recovering from a crash or restarting after a shutdown), and consequently need a long "ramp-up" period to regain peak performance. A recent method to address this limitation is to use a memory-mapped file to store the metadata (called the SSD buffer table) about the contents of the SSD buffer pool, and to recover the metadata at the beginning of recovery. However, this method can result in lower sustained performance, because every update to the SSD buffer table may incur a random I/O operation. In this chapter we propose two new designs. One design reconstructs the SSD buffer table using transactional logs. The other design asynchronously flushes the SSD buffer table, and upon restart, lazily verifies the integrity of the data cached in the SSD buffer pool. We have implemented the previously proposed scheme and these two new schemes in SQL Server 2012. For each design, both the write-through and the write-back caching policies were

implemented. Using two OLTP benchmarks (TPC-C and TPC-E), our experimental results show that our designs produce up to 3.8X speedup on the interval between peak-to-peak performance, with negligible performance loss; in contrast, the previous approach has a similar speedup but up to 54% performance loss.

4.1 Introduction

Using a flash SSD to extend the main memory buffer pool is well established as a way to improve the performance of DBMSs (e.g., [17, 24, 31, 52]). With an SSD buffer-pool extension, a DBMS still treats the disks as the permanent "home" of data. However, when pages are evicted from the buffer pool, selected pages are cached in the SSD (called the SSD buffer pool). Subsequent access to such pages can be served by fetching them from the SSD. Generally, the SSD buffer pool is used to cache pages that are likely to be accessed in the future with a random I/O access pattern. Consequently, these methods result in improved performance when the random I/O speed of the SSD is (much) faster than the aggregate random I/O speed of the disks.

However, such schemes may have a long "peak-to-peak interval" when restarting the DBMS from a crash or from a shutdown. As Figure 4.1 illustrates, the peak-to-peak interval has three components: shutdown, recovery, and ramp-up.

On restart, a simple scheme for SSD buffer-pool extension can simply throw away all the pages in the SSD buffer pool and recover the system from the data in the disks (in the normal fashion). However, in this case the ramp-up time can be very long (of the order of many hours in our experiments). Another approach to restarting when using an SSD buffer-pool extension is to keep, at all times, an accurate catalog/metadata of the pages that are cached in the SSD buffer pool in some well-known persistent location. Then, on restart, we can reload the SSD buffer pool metadata, called the SSD buffer table, and reuse the pages that were previously cached in the SSD. In fact, Bhattacharjee *et al.* [18] recently proposed such a scheme in

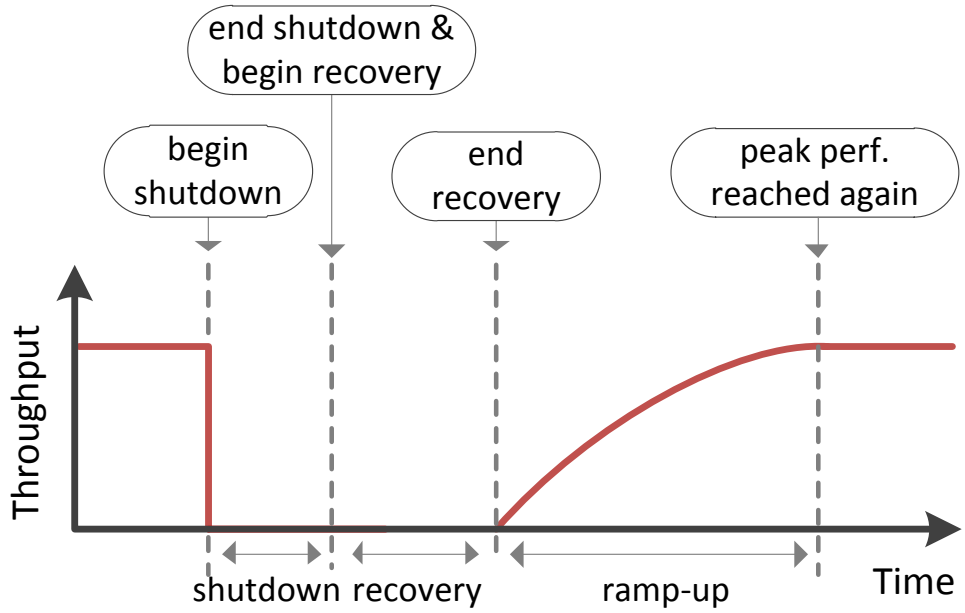


Figure 4.1: The peak-to-peak interval is the total time of shutdown (= 0 if system has crashed), recovery, and ramp-up.

which the SSD buffer table is implemented as a memory-mapped file. The memory-mapped file can be stored in the SSD with the SSD buffer pool, or it can also be stored on disk, or a dedicated SSD. This approach, which we call the Memory-Mapped Restart (MMR) scheme, does in fact reduce the peak-to-peak interval. However, one drawback of this approach is that it has the potential to generate a large amount of additional I/O traffic for every change that is made to the SSD buffer table. Consequently, in some cases, the overall peak performance with this approach can be lower compared to the case when the SSD is not used to speed up the restart process. In other words the actual peak that is achieved in Figure 4.1 can be lower.

What we need is a fast mechanism to reduce the restart (shutdown and recovery time in Figure 4.1), and the ramp-up time, without impacting the actual peak performance that can be achieved when using the SSD buffer-pool extension. In this chapter we propose two such methods, called the Log-based Restart (LBR) and Lazy-Verification Restart (LVR).

The main idea behind the LBR method is to flush the SSD buffer table during the

checkpoint operation, and to log the updates made to the SSD buffer table in the regular database transactional log. Upon restart, the SSD buffer table can be reconstructed from the log. The major challenge in this design is to figure out the protocol to checkpoint, log, and recover. On the other hand, the main idea behind the LVR method is to asynchronously flush the SSD buffer table periodically. During a restart, then, the contents of the SSD buffer table are lazily verified "on demand." A key challenge in this design is dealing with invalid SSD buffer table records that are recovered from the most recent flush.

In Chapter 3, we examined four pure SSD buffer-pool extension (no restart from the SSD) designs: LC (a write-back approach), DW (a write-through approach), CW (a simple approach that only caches clean pages in the SSD), and TAC, which was proposed by Canim *et al.* [24]. The findings pointed to LC being better for OLTP workloads that fit the TPC-C model, and DW being better for OLTP workloads that fit the TPC-E workloads. (SSD buffer pool extension also helps warehousing workloads, and all the schemes have similar performance in that case; in the interest of space, we only focus on OLTP workloads.)

So for a comprehensive study, we need SSD-restart schemes that work with both DW and LC. In this chapter, we evaluate the performance for the three SSD recovery designs (MMR, LBR, and LVR), against both DW and LC, using both the TPC-C and the TPC-E workloads.

The key contributions of this chapter are as follows.

- We propose two new methods for restarting from SSDs.
- We make the two new restart methods and the existing memory-mapped method work with both DW and LC.
- We carry out an extensive evaluation of the three SSD restart design alternatives, and the original SSD buffer-pool extension case, for both the DW and LC policies, using TPC-C and TPC-E. Using this evaluation, we identify the benefits and drawback of

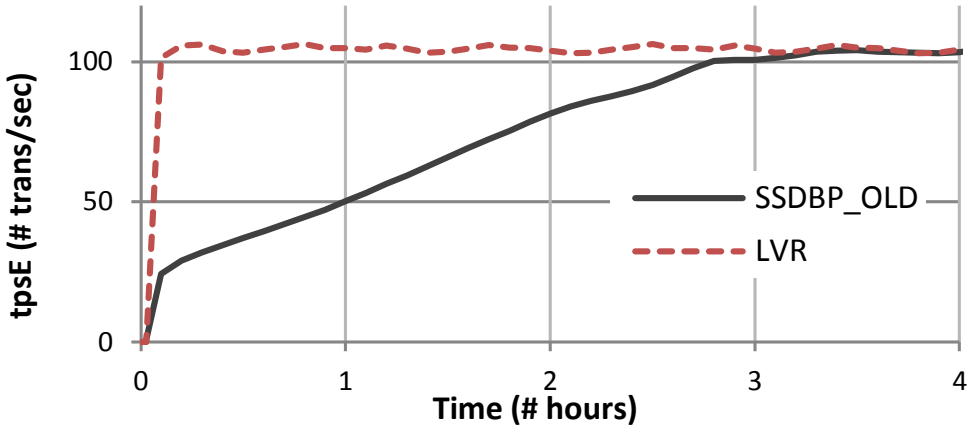


Figure 4.2: After a shutdown with a 20K customer TPC-E database in the DW design. SSDBP_OLD denotes the SSD buffer-pool extension used in Chapter 3 (without restarting from the SSD or aggressive fill).

each approach producing a comprehensive study of these methods. Our study shows that LVR+DW is generally the best scheme.

In addition, we propose a simple idea, called "aggressive fill", that we had overlooked in our previous work. This method dramatically improves the ramp-up time in all cases that we study in this chapter.

Collectively our contributions show that we can restart from the SSD without negatively impacting peak performance. Figure 4.2 is a representative result that summarizes our overall contribution. Here we show the original DW design in Chapter 3, compared to the proposed LVR method on DW. The LVR method has the same peak performance, but has a 13X speedup in the time to reach peak performance (of this, LVR contributes to 3.8X of the speedup, with the remainder performance improvement coming from the aggressive fill technique).

The remainder of the chapter is organized as follows: Section 4.2 describes background information. Section 4.3 describes the three SSD-restart designs. Section 4.4 contains the performance evaluation and analysis. Section 4.5 describes related work. Section 4.6 contains our summary.

Acronym	Meaning
DW	The Dual-Write design (Chapter 3) of SSD buffer-pool extension, a write-through policy.
LC	The Lazy-Cleaning design (Chapter 3) of SSD buffer-pool extension, a write-back policy.
SSDBP	The original SSD buffer-pool extension design that regards the SSD buffer pool empty at restart.
MMR	Memory-Mapped Restart [18] (Section 4.3.2).
LBR	Log-based Restart (Section 4.3.3).
LVR	Lazy-Verification Restart (Section 4.3.4)
FC	A record in the SSD buffer table. The name comes from "Flash Cache".

Table 4.1: Acronyms commonly used in this chapter.

4.2 Background

For ease of reference, the commonly used acronyms that are used in this chapter are listed in Table 4.1. Note that each of the three SSD-restart designs (MMR, LBR, LVR), as well as SSDBP, has both a DW version and an LC version. So we essentially evaluate eight designs. Next, we describe aspects of the SQL Server 2012 recovery protocol that is relevant to the schemes presented in Section 4.3

4.2.1 Recovery in SQL Server 2012

4.2.1.1 Data Structures

The **transaction log** is a persistent sequence of log records. Each log record is uniquely identified by an ever-increasing log-sequence number (LSN), which we denote as the *currentLSN*. Every time a page is changed, an UPDATE log record is generated to indicate the change to the page. Once the log record is created, the LSN of the record is written to the header of the page (*pageLSN*). Then, the *pageID* (the ID of the page that is being updated, which consists of a database ID, a file ID, and a page number in the file), *prevPageLSN* (the LSN of the page before the update took place), and the redo/undo information are appended to the log record. Before writing a page to disk, as per Write-Ahead Logging (WAL), the

1. Write a BEGIN_CHKPT log record. Let the LSN of the log record be *beginChkptLSN*.
2. Flush the log.
3. Write the transaction table to the log.
4. Set *oldestDirtyLSN* = min{*recLSN* of a dirty page}.
5. Set *oldestTxLSN* = min{*beginLSN* of a transaction}.
6. Write an END_CHKPT log record.
7. Write *oldestDirtyLSN*, *oldestTxLSN*, and *beginChkptLSN* to the boot page of the database file.
8. Let *truncationLSN* = min{*oldestDirtyLSN*, *oldestTxLSN*, *beginChkptLSN*}. Truncate the log to remove the log records older than *truncationLSN*.

Figure 4.3: The indirect checkpoint algorithm.

UPDATE log records for that page are flushed to the log disk. In addition, after a page is safely written to disk, a request to create a BUF_WRITE log record for that page is made. When there are a sufficient number of these BUF_WRITE log requests (2048 in the current implementation), a BUF_WRITE log record is created with *pageIDs* of all the pages in that "batch", and their minimum *pageLSNs*. The BUF_WRITE log record indicates that the disk versions of the referenced pages are at least as new as the minimum *pageLSN*.

The **dirty page table** stores information about dirty pages in the main memory buffer pool. Each record in the dirty page table stores a *pageID*, a *recLSN* (i.e., recovery LSN – the LSN of the log record that first caused the page to be dirty), and a *lastLSN* (the LSN of the last update made to the page). The *recLSN* indicates the starting point for updates that are potentially not yet reflected in the disk version of that page.

The **transaction table** stores information about active transactions. Each record in the transaction table stores the *beginLSN* (the LSN of the first log record in the transaction) and the *endLSN* (the LSN of the last log record in the transaction).

4.2.1.2 Checkpoints

SQL Server 2012 [4] employs a light-weighted checkpoint scheme called "indirect checkpoint." In this scheme, a background-writer thread continually flushes old dirty pages to the disk, in increasing order of the *recLSN*. The actual checkpoint is a fuzzy checkpoint (e.g., [16, 37, 58]), and the background writer method (a) allows for faster checkpoints, and (b) enables a more continuous and smooth write traffic to the disk subsystem. The pseudocode for the indirect checkpoint algorithm is given in Figure 4.3. Note that the dirty page table is not written to the transactional log during a checkpoint unlike in ARIES [58].

4.2.1.3 Recovery

As in ARIES [58], the recovery algorithm used in SQL Server 2012 has three phases: analysis, redo, and undo.

1. **Analysis Phase:** The analysis algorithm scans the log forward from min{*oldestDirtyLSN*, *beginChkptLSN*} and builds the dirty page table as follows: When processing an UPDATE log record, if the page has an entry in the dirty page table, then the entry's *lastLSN* is updated, else a new entry is created with the *lastLSN* set to the log record's *currentLSN*. Upon encountering a BUF_WRITE log record, the algorithm removes the entry for the referenced page from the dirty page table (if any), if the entry's *lastLSN* is smaller than or equal to the log record's *pageLSN*.

   In the interest of space, we omit discussion about other operations, such as recovering the transaction table, recovering the lock table, and processing other types of log records.

2. **Redo Phase:** The redo algorithm scans the log forward from the smallest *recLSN* in the dirty page table constructed by that analysis phase. For each UPDATE log record, if the updated page is referenced in the dirty page table, and if the dirty page table entry's *recLSN* is smaller than or equal to the log record's *currentLSN*, then the redo

algorithm requests for the page (loading the page to the buffer pool if not already in there). If the page's *pageLSN* is smaller than the log record's *currentLSN*, then the update is applied to the page in the buffer pool. If the *pageLSN* is larger than or equal to the log record's *currentLSN*, then the redo action is simply skipped.

Note that if the *pageLSN* is smaller than the log record's *currentLSN*, then this *pageLSN* must be equal to the log record's *prevPageLSN*. Intuitively, if a redo is about to be performed, the page must be "ready", in the sense that it must be in the state right before the logged update is performed. This case may seem obvious in this discussion, but as will be described in Section 4.3.4, requires careful handling.

3. **Undo Phase:** The undo phase rolls back the updates of the "loser" transactions, using the same algorithm as in ARIES [58].

4.2.2 SSD Buffer-Pool Extension

This section reviews the DW and LC SSD buffer-pool extension designs that are proposed in Chapter 3. Our SSD restart methods in this chapter build on these designs. For a comparison of DW and LC with other methods, please see Chapter 3.

The main idea behind the SSD buffer pool extension is to use the SSD to cache pages that are evicted from the buffer pool. To manage data in the SSD, the SSD manager is introduced, which is a software layer between the buffer manager and the I/O manager.

4.2.2.1 Data Structures

Figure 4.4 shows the data structures used by the SSD manager.

The SSD buffer pool is an array of frames that are page–sized regions in which the database pages are cached. It resides on the SSD as a single file. This file is created (or opened if it already exists) on the SSD when the DBMS is started.

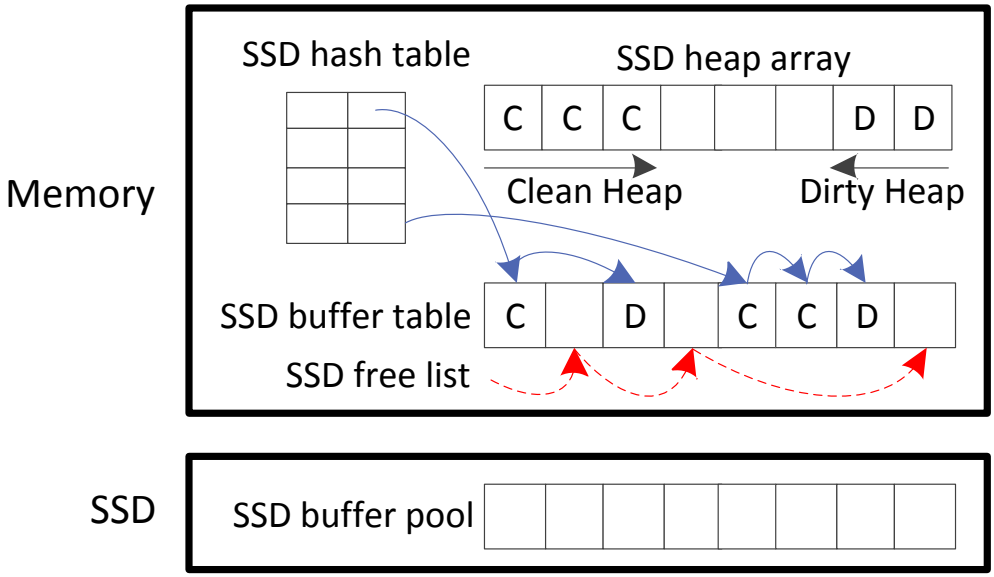


Figure 4.4: The data structures used by the SSD Manager.

The SSD buffer table is an array of records corresponding to the frames in the SSD buffer pool. Each SSD buffer–table record, called an FC, has the following key fields:

```
struct FC {
  State state;
  PageID pageID;
  int lastUseTime;
  int nextToLastUseTime;
  LSN recLSN;
  ...
};
```

The *state* field indicates the state of the FC, which can be FREE, CLEAN, or DIRTY (discussed in more details below). The *lastUseTime* and *nextToLastUseTime* fields are used for the LRU-2 replacement policy. The *recLSN* field has the same meaning as the *recLSN* field of an entry in the dirty page table. (If the dirty SSD page is loaded into the main

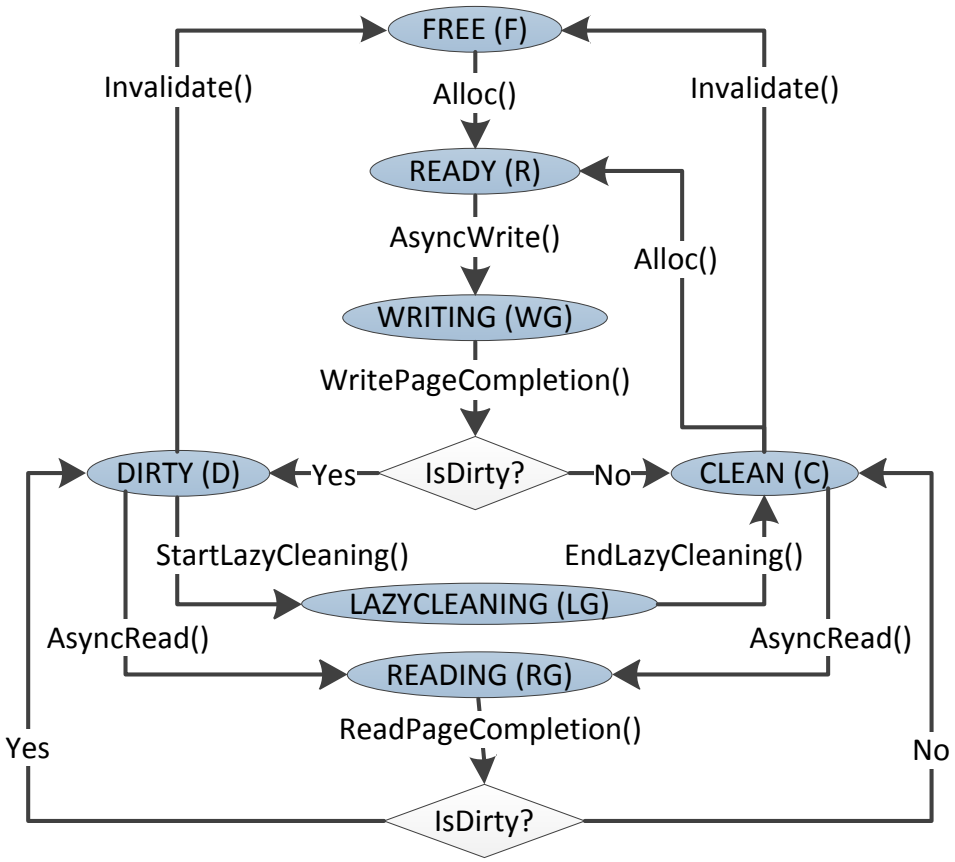


Figure 4.5: FC states and their transitions. Note that LAZYCLEANING and DIRTY states are only valid in the LC design.

memory buffer pool, and if the page is updated, the SSD frame will be invalidated but an entry will be inserted into the dirty page table, along with the value of this *recLSN* field, and not with the value of the newest update log record.)

The *SSD free list* is a linked list of free FCs in the *SSD buffer table*. The SSD hash table enables fast translations of a *pageID* to the SSD frame, if any, that caches the specified page. The *SSD heap array* embeds a dirty heap and a clean heap. The dirty heap stores references to the dirty FCs, where the heap root has the smallest *recLSN*. The clean heap stores references to the clean FCs, where the heap root has the smallest *nextToLastUseTime*.

4.2.2.2 FC State

Beyond the FREE, CLEAN or DIRTY states for an FC, there are four other FC states: READY, READING, WRITING and LAZYCLEANING, to deal with the use of asynchronous IOs in SQL Server. Figure 4.5 illustrates the seven FC states and their transitions. For example, when a FREE or a CLEAN frame is allocated to cache a new page, the FC state is set to READY. After an asynchronous SSD write request is issued, the FC state is changed to WRITING. Once the write is done, the state is changed to CLEAN if the copy of the page in the SSD is identical to the copy on the disk. Otherwise, it is changed to DIRTY.

4.2.2.3 DW and LC

Both these designs share the data structures described in the previous section. Note that the *recLSN* field of each FC and the dirty heap are only used in the LC design.

The two designs also share in common the following behavior. The disk subsystem is the "permanent home" of the data pages. On restart, both the main memory buffer pool and the SSD buffer pool are considered empty. When a page is requested to be loaded into the buffer pool, if a copy of the page exists in the SSD buffer pool, then it is loaded from there; otherwise, it is loaded from the disks. When a clean page is evicted from the buffer pool, it is cached in the SSD buffer pool if it meets certain admission criteria (which generally tries to cache pages that are likely to be re-accessed later using a random I/O pattern). To allocate an SSD frame to store a page, if there exists at least one free SSD frame, then the head of the SSD free list is used; otherwise, the root of the clean heap is chosen for replacement. When a clean SSD frame is chosen for replacement, the page content is simply discarded (because it is identical to the disk version of the page). When a page is modified in the buffer pool, the SSD frame that caches the page, if any, is *invalidated* (i.e. marked FREE, with necessary operations on the related data structures, e.g. to remove an entry from the SSD hash table).

The two designs mainly differ in the way they deal with dirty pages that are evicted from the buffer pool. In the DW design, dirty pages evicted from the buffer pool are written both to

the SSD buffer pool and to the disks. In effect, the SSD buffer pool acts as a "write-through" cache for dirty pages. In the LC design, on the other hand, dirty pages evicted from the buffer pool are written only to the SSD. A background *lazy cleaner* thread is in charge of copying dirty SSD pages to the disks. In effect, the SSD buffer pool acts as a "write-back" cache.

4.3 Restart Design Alternatives

Now, we explain the three SSD restart design alternatives. Section 4.3.2 explains the Memory-Mapped Restart (MMR) scheme that was adapted from Bhattacharjee *et al.* [18]. Section 4.3.3 and 4.3.4 introduce our new restarting designs, Log-based Restart (LBR) and Lazy-Verification Restart (LVR), respectively. These three schemes work with both DW and LC as described below. But, before we can use the DW and LC designs as described in Section 4.2, we need to make some changes to these designs to allow for correct restart from the SSD. In the next section, we first describe these basic changes.

4.3.1 Pitfalls in Using the SSD after a Restart

When the contents of the SSD buffer pool is reused after a restart, there are a number of pitfalls to watch out for, in both the DW and the LC designs, which require changes to these base designs. These changes are described below.

In DW (originally pointed out by [18] for the TAC design), suppose that the system crashes after the SSD write has completed, but before the disk write has completed. Now, after a restart, the SSD page is newer than the disk page, but the system does not know about it because the FC entry is marked as CLEAN. So when the SSD page is replaced, it is simply discarded. Later when the page is needed, an older version will be loaded from the disks, unexpectedly. The page may not even be recoverable, because the update log records may have been truncated. The solution is to delay modifying the FC until both the SSD

write and the disk write have completed.

In LC, our first implementation resulted in extremely long redo time. (About 30 hours in one experiment!) The reason for the long redo time was that the dirty pages in the main memory buffer pool were organized as a sorted list (by *recLSN*) as in the original SQL Server code. In the original SQL Server code, newly generated dirty pages are inserted to one end of the list, because they get ever-increasing *recLSNs*; and the entries from the list are extracted from the other end (to be flushed). However, when the SSD buffer pool is reused during a restart, dirty pages are no longer generated in increasing order of *recLSN*. In particular, if a dirty page is loaded from the SSD, the FC's *recLSN* will be used. So essentially for every dirty page in the SSD, a linked list (of millions of entries) has to be traversed to keep the list ordered. This turns out to be very expensive. The solution we adopted is to replace the sorted list with a heap.

Two other parts in LC also needed changes. The generation of the BUF_WRITE log records was modified such that, upon completing a write of a dirty page to the SSD, the system does NOT generate a BUF_WRITE log record, because the disk version of the page is still old. Instead, a BUF_WRITE log record is generated after the lazy cleaner thread finishes copying a dirty SSD page to the disks. Also, the checkpoint logic was modified to consider the dirty pages in the SSD buffer pool when computing the *oldestDirtyLSN* value, in addition to the dirty pages in the main memory buffer pool. In particular, *oldestDirtyLSN* is the oldest *recLSN* of the dirty pages in the main memory buffer pool and in the SSD buffer pool.

4.3.2 Memory-Mapped Restart (MMR)

Bhattacharjee *et al.* [18] proposed to extend their TAC SSD buffer pool extension design [24] to reuse the cached pages in the SSD buffer pool upon a restart, by storing the SSD buffer table as a memory-mapped file. We use this memory-mapped-file idea to extend the DW and LC designs introduced in Chapter 3, and call the resulting design Memory-Mapped Restart

(MMR). The details for this scheme are described below.

4.3.2.1 The FC Fields to Harden

The fields in an FC that should be hardened include: *state*, *pageID*, *lastUseTime*, and *nextToLastUseTime*.

The SSD buffer table is broken into two parts: the memory-mapped part that stores the above four to-be-hardened fields of each FC, and the volatile part that stores the remaining fields.

4.3.2.2 Memory-Mapped File Implementation

When the database system starts up, several Windows APIs are used: `CreateFile()` with WRITE_THROUGH and NO_BUFFERING flags is used to create a file for the memory-mapped part of the SSD buffer table, `CreateFileMapping()` is used to create a file mapping object, and `MapViewOfFile()` is used to map it to the address space. After each important update to the memory-mapped part of the SSD buffer table, `FlushViewOfFile()` is called to flush the modified FC to the file.

4.3.2.3 When to Harden

Flushing changes to the memory-mapped file may be expensive, and can hinder the sustained peak performance during regular forward processing. Hence, instead of flushing changes on every FC state transition (See Figure 4.5), we try to minimize the number of flushes without affecting correctness. For example, when a clean SSD frame is about to be replaced, we have to flush the state change. Otherwise, after a restart, the recovered FC may erroneously indicate that the SSD frame is clean (with pageID of the old page that was being replaced). On the other hand, when a free SSD frame is about to receive a new page, the state change need not be flushed. At a restart, the recovered FC will be in the FREE state (instead of READY). This is exactly what is expected, because the write to the SSD frame did not finish,

	DW	**LC**
1) When writing a page to a free SSD frame	1) WR → C	1) WR → C/D
2) When writing a page to a clean SSD frame	1) C → R	1) C → R
	2) WR → C	2) WR → C/D
3) When modifying an SSD page	1) C → F	1) C/D → F
4) When lazycleaning an SSD page	N/A	1) LR → C

Table 4.2: Cases that require flushing the FC state changes. F, C, D, R, WR and LR stand for the FREE, CLEAN, DIRTY, READY, WRITING and LAZYWRITING states, respectively.

and therefore the SSD page cannot be reused. Even if a recovered FC is found to be in the READY state, its state will need to be changed to FREE.

Table 4.2 summaries all possible cases that require flushing state changes in the DW and LC designs, respectively. For example, when writing a page to a clean SSD frame (Case 2) in the DW design, each of two state changes (from CLEAN to READY and from WRITING to CLEAN) is flushed.

4.3.2.4 Recovery

The recovery algorithm is the same as in Section 4.2.1.3, with the addition that, at the beginning of the analysis phase, the SSD buffer table is recovered.

To recover the SSD buffer table, the memory-mapped part of the SSD buffer table is loaded from persistent storage. Next, the SSD buffer table is processed as follows: First, every FC state must be one of FREE, CLEAN, or DIRTY. FC entries in the READY and the WRITING states are treated as FREE, while FC entries in the LAZYCLEANING and the READING states are treated as DIRTY. Second, the data structures (described in Section 4.2.2.1) are rebuilt. In particular, references to the CLEAN/DIRTY FCs are inserted to the clean/dirty heap, and also inserted to the SSD hash table; and, free frames are linked together in the SSD free list.

A final detail is how to recover the *recLSN* field for each dirty FC. One possible solution is to include this field in the memory-mapped part of the SSD buffer table. But this approach

would mean more data being flushed during regular forward processing. In MMR, we chose to recover this field of each dirty FC at the end of the analysis phase, when all the dirty pages and their *recLSN* values are in the dirty page table.

4.3.3 Log-Based Restart (LBR)

The main idea behind LBR is to checkpoint the SSD buffer table during a normal database checkpoint, and to log the updates made to the SSD buffer table in the database transaction log. The up-to-date SSD buffer table can be reconstructed during the analysis phase, along with the construction of the dirty page table.

4.3.3.1 The FC Fields to Harden

The fields in an FC that should be hardened (or flushed) include: *state*, *pageID*, *lastUseTime*, and *nextToLastUseTime*. This list is exactly the same as in MMR. The hardening appears in the form of newly introduced log records, as discussed below.

4.3.3.2 SSD Log Records

During forward processing, the following four types of new log records are generated.

- **SSD_CHKPT** During a checkpoint, the whole SSD buffer table is hardened to the transactional log. More specifically, for each FC, the four fields pointed out in Section 4.3.3.1 are hardened. Theoretically a single log record is enough. But to make sure the log record size is not too large, a sequence of SSD_CHKPT log records are used. Each such log record hardens a pre-specified number (we use 64) of FCs.
- **SSD_PRE_WRITE_INVALIDATE** Before a page is written to the SSD, an SSD frame is allocated. If there is no free SSD frame available, a clean frame is chosen to be replaced, and an SSD_PRE_WRITE_INVALIDATE log record is generated, with a single value: the SSD *frameNo*.

As the name indicates, this type of log records is generated when an SSD frame is invalidated because a write to the page is about to take place, not because of other reasons. In particular, another case when an SSD frame needs to be invalidated is when the page is modified in the buffer pool. In the latter case, an existing type of log records, i.e. the UPDATE log record, will be generated, and therefore no new SSD log record is needed.

- **SSD_POST_WRITE** An SSD_POST_WRITE log record, describing the metadata for the page, is generated after a page is written to the SSD. The data fields associated with this log record are those pointed out Section 4.3.3.1, plus the SSD *frameNo*.

- **SSD_LAZY_CLEANED** This type of log records is specific to the LC design. After a dirty SSD page is lazily cleaned, an SSD_LAZY_CLEANED log record is generated, which stores the SSD *frameNo*.

4.3.3.3 When To Flush

Among the new types of SSD log records, only the SSD_PRE_WRITE_INVALIDATE log record must be flushed to disk, before the thread that generates the log record can continue. The reason for this requirement is a thread generating an SSD_PRE_WRITE_INVALIDATE log record will then writing a page to the SSD frame. If the log is not flushed before the page is written to the SSD, and a crash takes place, upon a restart the system will believe that the SSD frame contains the old page (before the pre-write invalidation took place), an obvious inconsistency.

The other three types of log records do not require an immediate log flush for the following reasons. The SSD_CHKPT log record does not need to be flushed immediately, because if a crash happens, at recovery the system can use a previous checkpoint – the recover might take longer, but the system will still recover correctly. The SSD_POST_WRITE log does not require an immediate log flush, because if a crash happens, at recovery the system will

regard this SSD frame as FREE (even though the SSD frame contains a valid page), without affecting correctness. The SSD_LAZY_CLEANED log record does not require an immediate log flush, since at recovery, the system will regard this SSD frame as DIRTY (even though the frame is CLEAN). The consequence is that when the page is evicted from the SSD, it needs to be written, wastefully, to the disks.

To reduce frequent log flushes, we introduce the **Group–Writing Optimization**. Multiple (up to eight in our implementation) write requests to the SSD, that require replacing some existing clean frames, are gathered (and the issuing thread suspended) so that a single log flush is performed before the write requests are issued. To prevent stalling, a timeout feature is used such that no SSD write request is delayed more than the timeout duration.

4.3.3.4 Recovery

The recovery algorithm is modified from Section 4.2.1.3, by recovering the SSD buffer table during the analysis phase. In particular, this section describes the behavior of the analysis algorithm, upon encountering the five types of log records (the four new SSD log records, plus the UPDATE log records).

- At the beginning of the analysis phase, all the SSD frames are marked as FREE.
- Before the last BEGIN_CHKPT log record in the log is encountered, the analysis algorithm is the same as that described in Section 4.2.1.3. After the last BEGIN_CHKPT log record is encountered, the analysis algorithm handles the five types of log records as follows:
- To process an SSD_CHKPT log record, the 64 (or so) FCs are recovered from the values stored in the log record. In case the FC state is DIRTY, the *recLSN* field is recovered via a lookup in the dirty page table. The SSD hash table is updated as well, unless the FC state is FREE.

- To process an SSD_PRE_WRITE_INVALIDATE log record, the corresponding FC is invalidated.
- To process an SSD_POST_WRITE log record, the algorithm used is exactly the same as the one used in the processing of an SSD_CHKPT log record, with the difference that here a single FC is processed.
- To process an SSD_LAZY_CLEANED log record, the FC state is changed from LAZYCLEANING to CLEAN.
- To process an UPDATE log record, in addition to the existing actions (Section 4.2.1.3), the FC that references this page, if any, needs to be invalidated.

4.3.4 Lazy-Verification Restart (LVR)

The main idea behind the LVR scheme is to use a background thread, called the *FC flusher thread*, to asynchronously harden the SSD buffer table to a persistent storage called the *SSD buffer-table file*. Upon a restart, the scheme recovers the SSD buffer table by loading from the SSD the buffer table file, before starting the analysis phase.

Due to the asynchronous nature of the flushing of the SSD buffer table, the recovered SSD buffer table may contain incorrect information. To guarantee correctness, when the content in the SSD buffer table are potentially out-of-date, the LVR recovery scheme must ensure the following two properties:

Property 4.1 (Safe-to-Reuse)**.** *The databases should be consistent, if the design chooses to reuse a page in the SSD buffer pool upon a restart.*

Potential violations of the Safe-to-Reuse property include the following:

- **Violation_A:** A recovered FC has a *pageID* that is different from that of the actual SSD page. If one page (with the FC's *pageID*) is requested but a different page (that is in the SSD buffer pool) is delivered, the databases will not be consistent.

- **Violation_B:** A recovered FC has a *pageLSN* that is different from that of the actual SSD page. The database may be inconsistent because, during redo, an UPDATE log record may be erroneously applied to a wrong version of the page.

- **Violation_C:** A dirty SSD page is considered clean. Before a restart, an SSD page may be newer than its disk version, and the FC was correctly marked as dirty. But upon the restart, an old version of the FC may be recovered, which may indicate that the SSD frame is clean. The consistency of the database will now be jeopardized because, when the page is evicted from the SSD, it will not be written to the disks, leaving an old version of the page in the system.

- **Violation_D:** The existence of an old SSD page, together with log truncation, may lead to data loss. The scenario is that an SSD frame storing a valid page is invalidated because the memory version of the page was modified. Later, the page gets written directly to the disks, bypassing the SSD. Upon restart, the old page in the SSD is found. Furthermore, assume that the recovered FC matches the old page. The log may have been truncated such that the recovery algorithm now does not encounter any UPDATE log record for that page. This leads to inconsistency because the system will believe the (old) SSD page is up-to-date.

In addition, the system should also ensure an inverse property:

Property 4.2 (Safe-to-Discard)**.** *The databases should be consistent, if the design chooses to discard a page in the SSD buffer pool upon a restart, even if the SSD page is newer than the disk version.*

The above Safe-to-Discard property is trivial to guarantee because of our modifications described at the end of Section 4.3.1. Recall that the checkpoint algorithm determines the *oldestDirtyLSN* as the minimum *recLSN* of the dirty pages in the main memory buffer pool and the SSD buffer pool. Hence, when an SSD page is newer than the disk version before a

restart, the *oldestDirtyLSN* will be sufficiently small, such that upon a restart all the log records that are needed to bring the old disk page up-to-date will be available to the recovery algorithm.

Sections from 4.3.4.1 to 4.3.4.5 present the LVR design, and shows how this design avoids the four violations to the Safe-to-Reuse property. 4.3.4.6 discusses three pitfalls related to Violation_D, i.e. having an SSD page older than the disk version.

4.3.4.1 The FC Fields to Harden

The fields in an FC that should be hardened include: *state*, *pageID*, *lastUseTime*, and *nextToLastUseTime*. These are the same as in the MMR and the LBR designs.

In addition, the LVR scheme also hardens the following two new fields: a *blank* flag, and *beforeHardeningLSN*. The *blank* flag is used to indicate whether an FC in the SSD buffer-table file was never written to, after the file was created. The *beforeHardeningLSN* field is the LSN of the tail of the log, before the value of the FC was hardened. The latter field is essential to avoid Violation_D.

4.3.4.2 The FC Flusher Thread

The FC flusher thread repeatedly scans the SSD buffer table in chunks, and hardens the FCs in each chunk. Figure 4.6 shows the algorithm to harden one chunk.

- **Step 1** remembers the LSN of the tail of the log. This LSN is then assigned to all the hardened FCs in this chunk (Step 3f). The usage of this field is to avoid Violation_D, as will be discussed in the recovery algorithm (Section 4.3.4.4).

- **Step 2** of the algorithm is needed to deal with the FCs for which a latch cannot be acquired. At Step 3b, the FC flusher thread tries to latch an FC, in preparation for copying its data out for flushing. It is not a good option for the FC flusher thread to acquire an infinite latch, because that would cause the design to take a long time to

1. Set *beforeHardeningLSN* = the LSN of the tail of the log.
2. Load (the old values of) the chunk of FCs from the SSD buffer-table file to a temporary space in memory.
3. For each SSD frame belonging to the chunk:
   a) Let *currentFC* denote the FC in the SSD buffer table, and *tmpFC* denote the FC in the temporary space.
   b) Try to latch the *currentFC*, without waiting. If the latching fails, skip it and go to the next frame.
   c) If *currentFC.state* = FREE, *tmpFC.state* = FREE, and *tmpFC.blank* = FALSE, release the latch and skip the FC.
   d) Set *state*, *pageID*, *lastUseTime*, *nextToLastUseTime*, and *recLSN* of *tmpFC* by copying from *currentFC*.
   e) Release the latch on *currentFC*.
   f) Set *tmpFC.beforeHardeningLSN* = *beforeHardeningLSN*.
   g) Set *tmpFC.blank* = FALSE.
4. Write the chunk of tmp FCs to the SSD buffer-table file.

Figure 4.6: The algorithm of the FC Flusher Thread used in the Lazy-Verification Restart (LVR) method to harden one chunk of FCs.

flush the SSD buffer table. But to enable the FC flusher thread to skip a busy FC, it needs to know what information to flush for a skipped FC. (Here we assume multiple FCs are flushed using one IO operation.) The safest thing would be to treat a skipped FC as FREE in the group flush. But, that would overwrite the useful information for this FC, in the SSD buffer table file, leading to a smaller SSD re-utilization upon a subsequent restart. The solution in LVR is that, in Step 2, the FC flusher thread reads a group of FCs from the SSD buffer table file. So, for the skipped busy FCs, the original metadata in the SSD buffer table file is written back.

- **Step 3** updates information in the loaded chunk, with information in the SSD buffer table, for the FCs where a latch could successfully be acquired.

- **Step 4** hardens the chunk. After hardening a chunk, the FC flusher thread may pause to yield the CPU for other activities.

The default values in the LVR scheme are as follows: The SSD buffer table is divided into 1024 chunks. If a checkpoint is taking place, the FC flusher thread pauses every 64 chunks, and yields the CPU. If no checkpoint is taking place, the FC flusher thread pauses every 8 chunks, sleeping for 500 milliseconds.

4.3.4.3 Checkpoints

The checkpoint algorithm in LVR is slightly modified from the algorithm in Section 4.2.1.2. The modification is to make sure that the FC flusher thread finishes a complete pass of hardening the SSD buffer table during a checkpoint. In particular, at the beginning of a checkpoint, the updated checkpoint algorithm takes a snapshot of where the FC flusher thread is at; and before the END_CHKPT log record is generated, the checkpointing thread is blocked until the FC flusher thread goes past the snapshot location. The modification is needed to avoid Pitfall 4.3 (discussed below in Section 4.3.4.6).

4.3.4.4 Recovery

The recovery algorithm is shown in Figure 4.7.

- **Step 1** recovers the initial values of the SSD buffer table by loading it from the SSD buffer table file, before verification.

- **Step 2** marks the FCs UNVERIFIED, but only if the system is recovering from a shutdown. The UNVERIFIED frames will then be lazily verified later, when the actual SSD pages are loaded into memory, either during the redo phase or during forward processing (Section 4.3.4.5). If the system is recovering from a crash, the recovery algorithm will eagerly verify the integrity of the SSD pages (in Step 6 of the recovery algorithm). So, there is no need to mark the FCs as UNVERIFIED here. To be able

1. Initialize entries in the SSD buffer table, by loading from the SSD buffer table file. The blank entries are treated as FREE.
2. If the server is recovering from a shutdown:
   a) Mark every FC, which is not FREE, as UNVERIFIED.
3. Invalidate every FC whose max{*pageLSN*, *beforeHardeningLSN*} < *truncationLSN*.
4. Build the SSD hash table. In the process, if it is found that two FCs have the same *pageID*, invalidate the one with an earlier *pageLSN*.
5. Build the SSD heap array and SSD free list.
6. If the server is recovering from a crash, for each FC that is not FREE:
   a) Load the SSD page to memory.
   b) If the FC and the page have different *pageID*, invalidate the FC.
   c) If the FC and the page have different *pageLSN*, invalidate the FC.
   d) If the page has an incorrect checksum, then invalidate the FC.
7. Perform the analysis algorithm as discussed in Section 4.2.1.3 with the following additional operation. Upon encountering a BUF_WRITE log record, invalidate the FC, if any, with the same *pageID* but with a *pageLSN* < the log record's *pageLSN*.
8. Perform redo and undo.
9. Start the lazy cleaner thread (for LC only), and the FC flusher thread.

Figure 4.7: The recovery algorithm used in the Lazy-Verification Restart (LVR) method.

to tell whether the system is recovering from a crash, one method is to store a flag called *ShutdownComplete* in a persistent location. The flag is set to true at the end of a server shutdown, and set to false at the beginning of the recovery process. The recovery algorithm knows that it is recovering from a shutdown, if and only if the flag (before the recovery algorithm set it to be false) is set to true.

The recovery algorithm uses two levels of verifications. A "shallow" verification is performed in Step 3. It is shallow because the verification is done purely by studying the metadata. A "deep" verification is performed in Step 6 (if recovering from a crash), or in case a page is loaded from the SSD (Section 4.3.4.5), by studying the information stored in the actual SSD pages.

- **Step 3** uses a shallow verification to avoid Violation_D, i.e. it deals with the case that a page found in the SSD buffer pool (upon a restart) may be older than the disk version. The correctness of the solution comes from the fact that it guarantees no useful UPDATE log records to the page may have be truncated. Here an UPDATE log record is useful if it is needed to bring the old SSD page up-to-date; i.e. if its *currentLSN* > the FC's *pageLSN*.

  Theorem 4.1. *Given a recovered FC, if* $\max\{$*pageLSN, beforeHardeningLSN*$\} \geqslant$ *truncationLSN, no UPDATE log records to the page, later than pageLSN , may have been truncated.*

  Proof. If the FC's *pageLSN* $\geqslant$ *truncationLSN*, the theorem is obviously true. Any log record later than *pageLSN* is also later than *truncationLSN*, and cannot have been truncated.

  It remains to prove the theorem is correct when *pageLSN* < *truncationLSN* $\leqslant$ *beforeHardeningLSN*. If the theorem were wrong, there must exist an UPDATE log record to this page, whose *currentLSN* is later than *pageLSN* but earlier than *truncationLSN*. When that UPDATE log record was at the tail of the log, the logged update should be

applied to the page, invalidating any SSD frame that stores (an older version of) the page. The fact that at a later time, when *beforeHardendingLSN* was at the tail of the log, an FC (the one that exists in the SSD buffer-table file) was observed to reference the same page indicates that the page re-written to the SSD (after the previous invalidation) must have a *pageLSN* $\geqslant$ the UPDATE log record's currentLSN. This contradicts with the statement that the UPDATE log record's LSN is later than the FC's *pageLSN* $\square$

The intuition behind Step 2 and Theorem 4.1 is that, however old the *pageLSN* of an SSD page may be, as long as the system knows that the page was valid at some later time (*beforeHardeningLSN*), and this later "time" is newer than the *truncationLSN*, then it is safe to reuse the SSD page. This guarantee is important in maximizing the reutilization rate of the pages previously cached in the SSD buffer pool. Without it, most of the clean pages in the SSD would have to be discarded at a restart, because they have old *pageLSNs*.

- **Step 4** builds the SSD hash table. The step also eliminates conflicts, as multiple FCs may have the same *pageID*. Without eliminating the conflicts, if a page is requested, the system would not know which SSD frame to load the page from.

- **Step 5** builds the other data structures. In particular, the SSD heap array and the SSD free list.

- **Step 6** (only applicable if recovering from a crash) uses a "deep" verification to make sure that, after this step, all the frames (that are not FREE) in the SSD buffer table are safe to use. The step scans the SSD buffer pool, and for each frame that has not been invalidated, Step 6(a) loads the SSD page. Then, Step 6(b) verifies the *pageID* making sure that the recovered FC's *pageID* matches the *pageID* stored in the actual page (this step avoids Violation_A). Step 6(c) verifies the *pageLSN* (this step avoids Violation_B). Note that by verifying the *pageLSN*, the algorithm also avoids Violation_C, for the following reason: If an SSD frame stores a dirty page, but the recovered FC shows

that the page is clean, then the recovered FC store a different *pageLSN*, leading to the invalidation of the FC.

In addition, Step 6(d) validates the checksum. Note that SQL Server already has a checksum scheme (to avoid *torn writes*), but here in LVR the scheme has to be modified. In SQL Server, after a page (either from the SSD or from the disks) is loaded to memory, if the checksum value stored in the page is different from the checksum computed over the page content, then the page is reloaded, up to four times, before media recovery is performed. In Step 6(d) in LVR, a page that is loaded from the SSD may fail the checksum test because the recovered FC contains wrong information. For instance, suppose that the system crashed while a write was taking place to a frame in the SSD buffer pool. During recovery, this SSD page will fail the checksum test; but, it is meaningless to read the bad page from the SSD again and again. In Step 6(d), such an FC is simply invalidated.

- **Step 7** performs analysis. The handling of BUF_WRITE log record is modified to avoid Pitfall 4.1 as will be discussed in Section 4.3.4.6.
- **Step 8** performs the traditional redo and undo phases.
- **Step 9** starts the lazy cleaner thread and the FC flusher thread. Note that the lazy cleaner thread is started at the end of the recovery algorithm. If it were started at the beginning, then the design would suffer from Pitfall 4.2 (see Section 4.3.4.6).

4.3.4.5 Lazy Verification

As the steps 2 and 6 of the recovery algorithm (Section 4.3.4.4) show, after the system recovers from a shutdown, some FCs may be marked as UNVERIFIED. The integrity of the corresponding SSD pages will be lazily verified when the SSD page is loaded into main memory, either during the redo phase or during forward processing. The verification is the

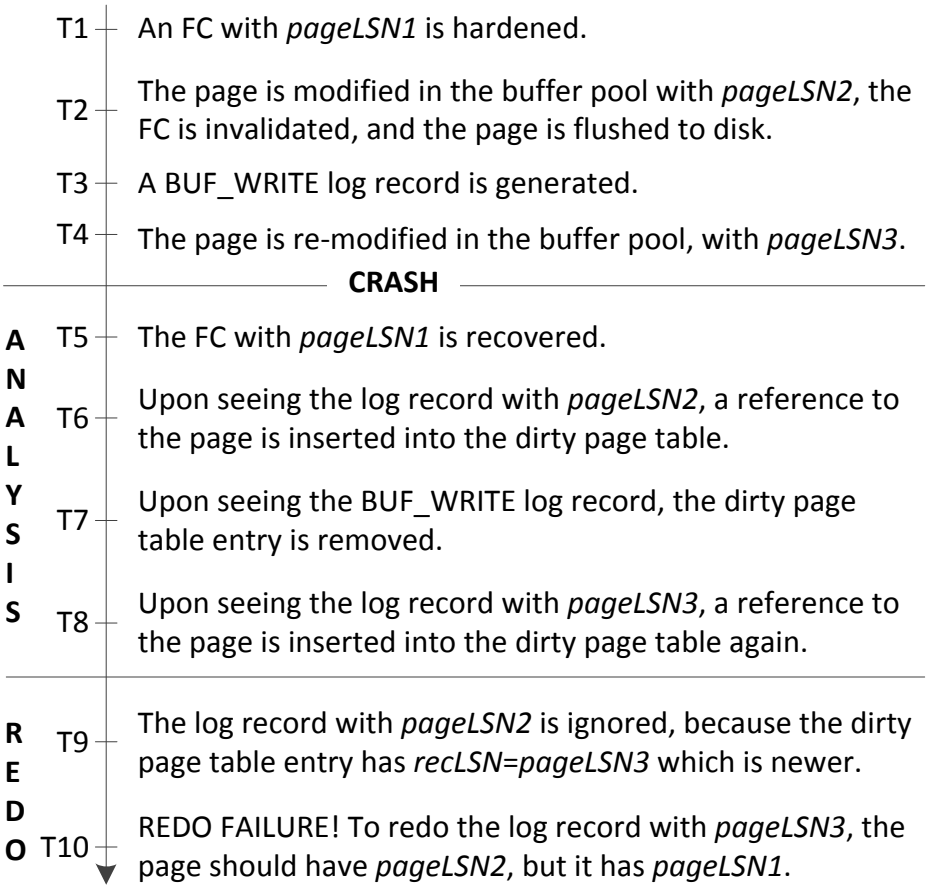


Figure 4.8: An old version of a page in the SSD may result in a redo failure.

same as steps 6(a) – 6(d) of the recovery algorithm. If the verification fails, then the FC is invalidated, and the page is loaded from the disk(s).

4.3.4.6 Pitfalls

There are several pitfalls to avoid when handling old SSD pages.

Pitfall 4.1. *An older version of a page in the SSD, in combination with BUF_WRITE log records, may result in a redo failure.*

In the presence of BUF_WRITE log records, an old SSD page may result in a redo failure, as illustrated in Figure 4.8. Intuitively, a BUF_WRITE log record indicates that a page has

been flushed to the disks; therefore the redo algorithm will skip older update log records. However, this behavior is not ok if the redo algorithm has an old SSD page and expects to use the update log records to bring the page up-to-date.

To avoid this pitfall, LVR modifies the analysis algorithm such that when processing a BUF_WRITE log record, the older version of the page in the SSD, if any, is invalidated first.

Pitfall 4.2. *The lazy cleaner thread, if it is working during the analysis phase, may replace a newer disk page with an older SSD page, and may lead to a redo failure in the future.*

The pitfall can also be illustrated using the example shown in Figure 4.8. Imagine that at T1, the SSD page with *pageLSN1* is dirty. During the analysis, but before T7, this old SSD page may be written to the disk by the lazy cleaner thread, overwriting the newer disk version. At T7, according to the solution to Pitfall 4.1, the FC is invalidated, hoping that a later redo action can access the correct version of the page from the disks. Unfortunately, the redo action at T10 will still fail, because the disk page is now also old.

To avoid this pitfall, LVR starts the lazy cleaner thread at the end of the recovery algorithm, or at least after the analysis phase.

Pitfall 4.3. *Most pages in the SSD may fail to be reused, if the server is restarted right after a checkpoint.*

Recall that (Section 4.3.4.4) during the analysis phase, an FC is invalidated, if $\max\{$*pageLSN*, *beforeHardeningLSN*$\} <$ *truncationLSN*. Also recall that (Section 4.2.1.3) *truncationLSN* $=$ $\min\{$*oldestDirtyLSN*, *oldestTxLSN*, *beginChkptLSN*$\}$. If there are no dirty pages or pending transactions before the restart, then *truncationLSN* is equal to *beginChkptLSN*, which may be newer than both the *pageLSN* and the *beforeHardeningLSN* of most of the FCs (since a checkpoint was issued right before the restart). That is why most of the FCs may fail to be reused after the restart. This is a performance pitfall, rather than a correctness pitfall. However, without solving the problem, the reutilization rate of the SSD pages can be close to zero.

LVR avoids this pitfall as follows: After a BEGIN_CHKPT log record is generated, LVR remembers where the FC flusher is processing. The generation of the END_CHKPT log record is postponed until the FC flusher thread finishes a complete pass of the SSD buffer table. The solution circumvents Pitfall 4.3 because the *beforeHardendingLSN* of an FC will now be newer than the *beginChkptLSN*. To minimize the delay in generating the END_CHKPT log record, during a checkpoint LVR increases the eagerness of the FC flusher thread. In our experiments with a 140GB SSD buffer-pool, the introduced delay is only several seconds.

4.4 Evaluation

This section compares the peak-to-peak interval and the peak performance of the three SSD restart designs, against the default-restart method without reusing data previously cached in the SSD.

4.4.1 Experimental Setup

We implemented the three SSD restart designs (MMR, LBR and LVR), as well as the default restart method (denoted as SSDBP, for the default no-restart SSD buffer-pool extension), in SQL Server 2012 CTP3. For each of the four methods, both a DW version and an LC version were implemented. For LC, the "Dirty Fraction" parameter was set to 20%, meaning that the LC thread starts working once the number of dirty SSD pages exceeds 20% of the SSD buffer pool size.

The experiments were performed on an HP ProLiant DL180 Server box with 2.13 GHz Intel dual quad-core Xeon processors (Nehalem) running 64-bit Windows Server 2008 R2 with 32 GB of DRAM (24 GB of DRAM was reserved for SQL Server). The databases were created on a filegroup that spans eighteen 300 GB 10,000 RPM SAS hard disk drives (HDDs). Two additional HDDs were dedicated to the OS and the transactional log, respectively. The

READ	Random	Sequential
18 HDDs	2,718	188,244
SSD	12,182	15,980

WRITE	Random	Sequential
18 HDDs	2,610	2,970
SSD	12,374	14,965

Table 4.3: Maximum sustainable IOPS for each device when using page-size (8KB) I/Os. Disk write caching is turned off.

SSD buffer pool used 140 GB out of a 160 GB SLC Fusion ioDrive. The SSD buffer-table file is stored on the same Fusion device.

Table 4.3 shows the IOPS of the SSD and the aggregate of the 18 HDDs (obtained using Iometer [3]).

For the workload, we used the TPC-C [7] and TPC-E [8] benchmarks, which are update-intensive and read-intensive OLTP workloads respectively. In both cases, the database size was around 200 GB (the TPC-C database has 2K warehouses; the TPC-E database has 20K customers). At this setting the database is larger than the main memory and the SSD buffer pool, and provides the most insightful experimental point to study the SSD buffer pool extension designs explained in Chapter 3.

For each workload, we used different throughput metrics and recovery intervals according to the TPC specifications. For TPC-C we measured the number of new orders that can be fully processed per minute (tpmC), and set 30 minutes for the recovery interval. For TPC-E we measured the number of (Trade-Result) transactions executed within a second (tpsE), and set 7.5 minutes for the recovery interval. The throughput is sampled using a one minute interval.

In all experiment, for LBR, we used the Group-Writing Optimization see Section 4.3.3.3) to (gather 8 write requests to the SSD. For LVR, the hardening interval (see Section 4.3.4.2) was set to 500 ms.

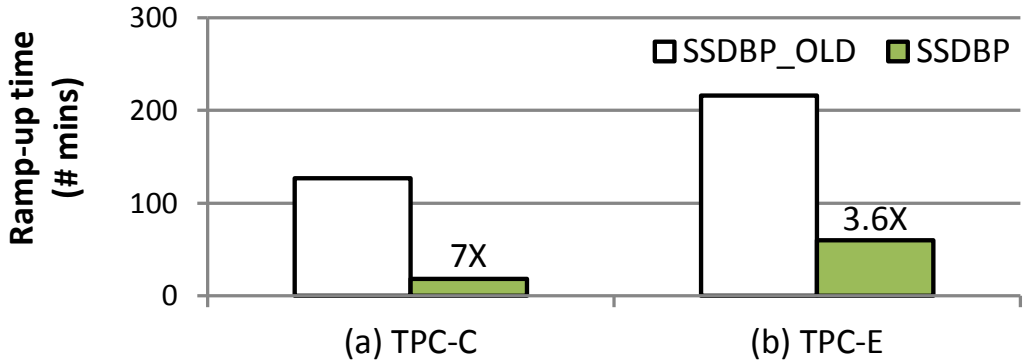


Figure 4.9: Ramp-up times on the TPC databases with DW. SSDBP_OLD does not use aggressive fill (as in Chapter 3). SSDBP uses aggressive fill.

4.4.1.1 The Impact of Aggressive Fill

In Chapter 3, we showed that the ramp-up time of an OLTP workload when using an SSD buffer pool extension if very long. The reason for this behavior is that to reach peak performance, the (large) SSD buffer pool needs to be filled, and to fill the SSD buffer pool, the data must be first loaded from the slow HDD subsystem. Since the SSD buffer pool largely caches pages that are accessed using a random I/O access pattern, it implies that the rate at which the SSD buffer pool fills up is gated by the random I/O performance of the HDD subsystem.

A simple but powerful idea, called "aggressive fill", can be used to significantly shorten the ramp-up time. In particular, before the SSD buffer pool is filled, every one-page HDD read request is expanded to read multiple (8 in our implementation) adjacent pages, including the requested page[1]. As Figure 4.9 shows, this technique reduced the ramp-up time by 7X and a 3.6X for the TPC-C and TPC-E databases, respectively.

In the remainder of this section, aggressive fill is used in all the implementations; and SSDBP (with aggressive fill) will be used as baseline when evaluating the effectiveness of the three SSD-restart designs.

[1]This technique is already implemented in the Enterprise versions of SQL Server to quickly fill the main memory buffer pool – here, we extend this technique to also aggressively fill the SSD buffer pool.

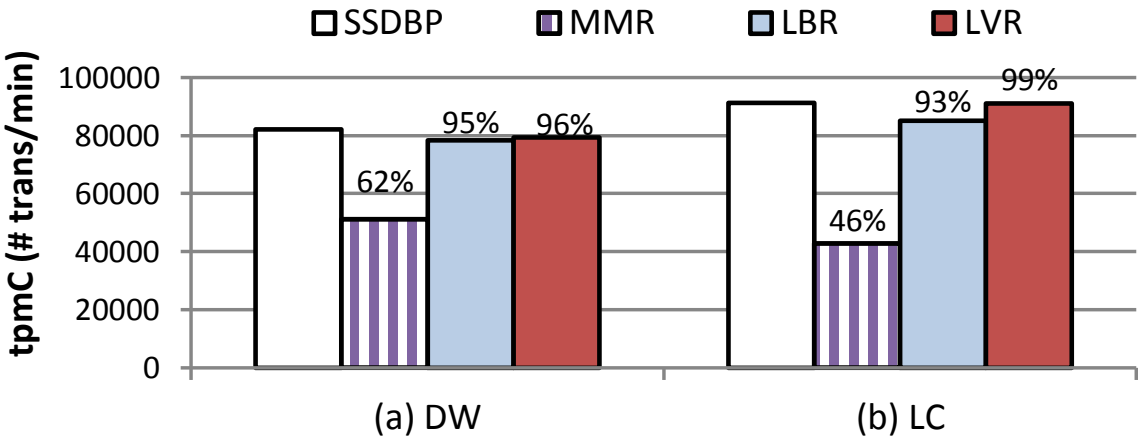


Figure 4.10: (TPC-C) Throughput after restarting from a shutdown. (Restarting from a crash is similar.)

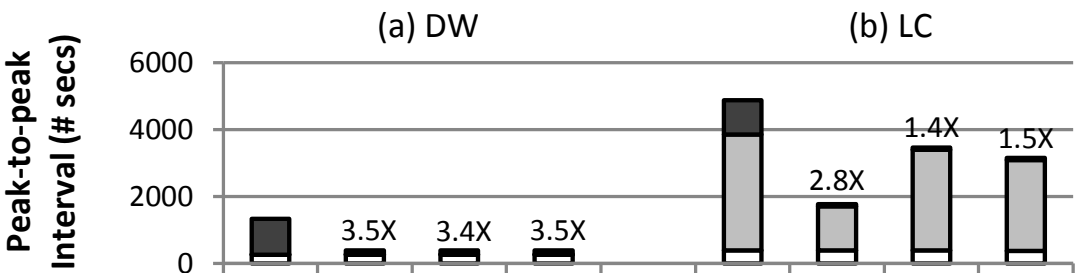
4.4.2 TPC-C Evaluation

4.4.2.1 Sustained Throughput

Figure 4.10 presents the steady-state throughput of the eight designs (four for DW and four for LC), for TPC-C, after restarting from a shutdown. (The results when restarting after a crash are similar, and omitted in the interest of space.) As can be seen in Figure 4.10, MMR's performance is only 62% and 46% as that of SSDBP, for DW and LC, respectively. LBR and LVR both have performance close to that of SSDBP, although LVR is slightly better.

Forward processing in MMR is hindered due to the additional I/O traffic that is required to synchronize the in-memory SSD buffer table with the memory-mapped file. For example, the percentage of busy time of the SSD increased from 79% (SSDBP) to 95% (MMR) in LC. A dedicated SSD could be used to store only the memory-mapped file (in some experiments in [18] a dedicated Fusion I/O device was used only for this purpose), but this option can be very expensive.

Forward processing in LBR is slowed down as LBR has to wait for certain SSD log records to be flushed to the log disk, which increases the cost associated with running each transaction. However, due to the Group-Writing Optimization (Section 4.3.3.3), LBR only lost 5% - 7%



	SSDBP	MMR	LBR	LVR		SSDBP	MMR	LBR	LVR
Ramp-up	1080	120	120	120		1020	60	60	60
Recovery	0.05	2.38	12.1	5.13		3455	1314	3019	2714
Shutdown	253	259	257	259		394	392	381	378

Figure 4.11: (TPC-C) Peak-to-peak interval (in seconds) for the case of restarting from a shutdown.

of the performance (over SSDBP). We have observed that without this optimization, LBR may lose up to 30% in performance over SSDBP.

Note that the LVR scheme (like MMR) also puts additional load on the SSD in order to harden the SSD buffer table. However, by controlling how eagerly the SSD buffer table is hardened (recall that the hardening interval is set to 500 ms), the algorithm can control the additional traffic that it adds to the SSD. For example, in this experiment, with LC the SSD busy time went up by just 3% (from 79% in the SSDBP case, to 82% with LVR). From this observation, we can also infer that if the SSD is significantly busy LVR could overload the SSD, degrading the sustained performance. However, LVR can also be modified to adjust the hardening interval dynamically using the current SSD utilization – such modifications are part of future work. In addition, LVR gathers information from many FCs and use one I/O to harden all of them. Such infrequent, large-size I/O pattern is performance-friendly to today's block devices.

4.4.2.2 Peak-to-peak Interval

Figure 4.11 compares the peak-to-peak intervals of the restart schemes, when restarting after a shutdown. For each of the three SSD-restart designs, the speedup over the SSDBP case is

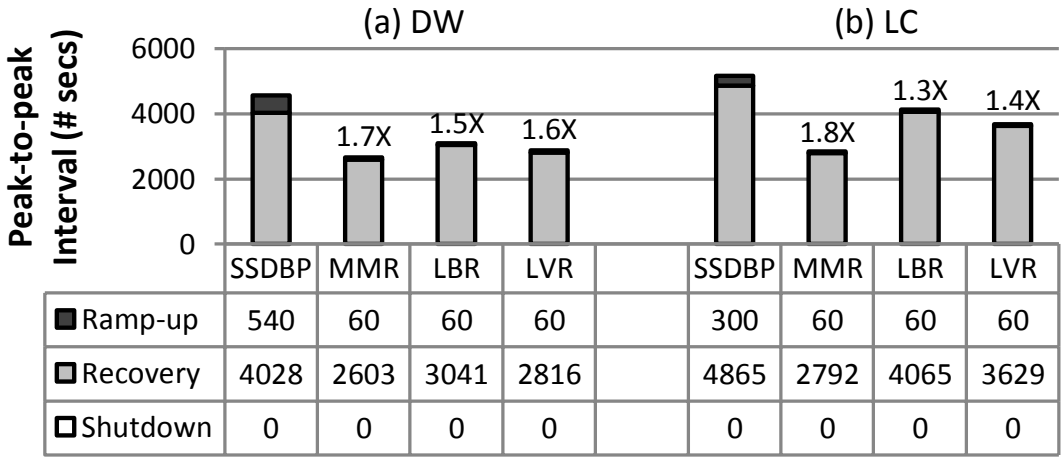
labeled on top of its bar. The SSD-restart designs provide significant speedup over SSDBP, ranging from 1.4X to 3.5X. With the DW policy, the three SSD-restart designs brought similar speedups (about 3.5X). With the LC policy, MMR has the best speedup (2.8X), while LBR and LVR have less speedup (about 1.4X). Figure 4.11 also shows the detailed breakdown of the peak-to-peak restart interval.

The shutdown time is a few hundred seconds in all the cases. The time is needed mainly to flush the dirty pages in the memory buffer pool.

The recovery time exhibits a big difference, between the DW case and the LC case. With the DW policy, the SSD-restart designs have recovery time in the order of seconds. With the LC policy, the SSD-restart designs have recovery time in the order of 1000 seconds. The difference lies in the value of *oldestDirtyLSN*. In the DW case, *oldestDirtyLSN* is NULL, because all the dirty pages in memory were flushed before the shutdown. In the LC case, *oldestDirtyLSN* is not NULL, as there were dirty pages in the SSD at shutdown – the policy allows the dirty pages to stay in the SSD. So in the LC case, during recovery a much longer log segment needs to be scanned.

Zooming into the recovery time of the DW case, we notice that while SSDBP has almost instant (0.05 sec) recovery time, the SSD-restart designs have several seconds of recovery time. The reason is that the SSD-restart schemes need to reconstruct the SSD buffer table. LVR's SSD buffer-table file is twice as large as MMR's memory-mapped file. So LVR's recovery time (5.13 sec) is twice as long as MMR's recovery time (2.38 sec). LBR's SSD_CHKPT log records collectively has the same size as MMR's memory-mapped file, but LBR's log records are stored in a disk, which is slower than Fusion. That is why LBR's recovery time (12.1 sec) is longer.

Zooming into the recovery time of the LC case, we notice that SSDBP has the longest restart time. This shows that the SSD-restart schemes did receive benefit by reusing the SSD pages during recovery. MMR has the shortest recovery time (2.8X better than SSDBP, instead of 1.4X or 1.5X for LBR and LVR). But this may be due to the fact that its sustained



	SSDBP	MMR	LBR	LVR		SSDBP	MMR	LBR	LVR
Ramp-up	540	60	60	60		300	60	60	60
Recovery	4028	2603	3041	2816		4865	2792	4065	3629
Shutdown	0	0	0	0		0	0	0	0

Figure 4.12: (TPC-C) Peak-to-peak interval (in seconds) for the case of restarting from a crash.

performance was about half of the performance of LBR and LVR – a dirty page may have more UPDATE log records to redo.

For the ramp-up time, the SSD-restart schemes brought an order of magnitude speedup. This is expected because the SSD-restart schemes start with a warm SSD buffer pool, while SSDBP starts with a cold SSD buffer pool. Note that after recovery, we gathered throughput data every minute. So all the ramp-up time data points reported in the chapter are multiples of 60 seconds.

Figure 4.12 compares the peak-to-peak intervals of the restart schemes, when restarting after a crash. The SSD-restart designs exhibit significant speedup over SSDBP, ranging from 1.3X to 1.8X. MMR has the best speedup over SSDBP, but LBR and LVR are close. A major distinction from the shutdown case is that, for DW, the recovery time is in the order of 1000 seconds, instead of seconds. This is because, for a crash recovery, *oldestDirtyLSN* is no longer NULL.

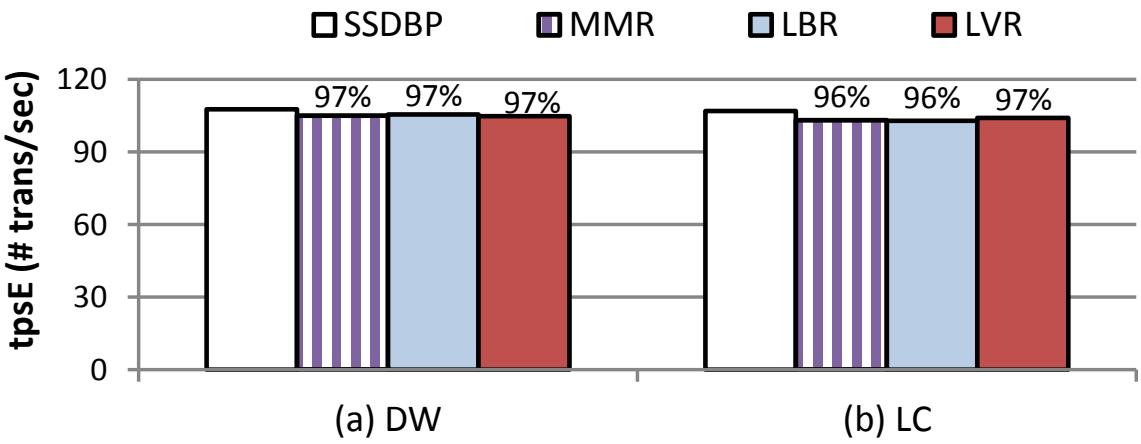
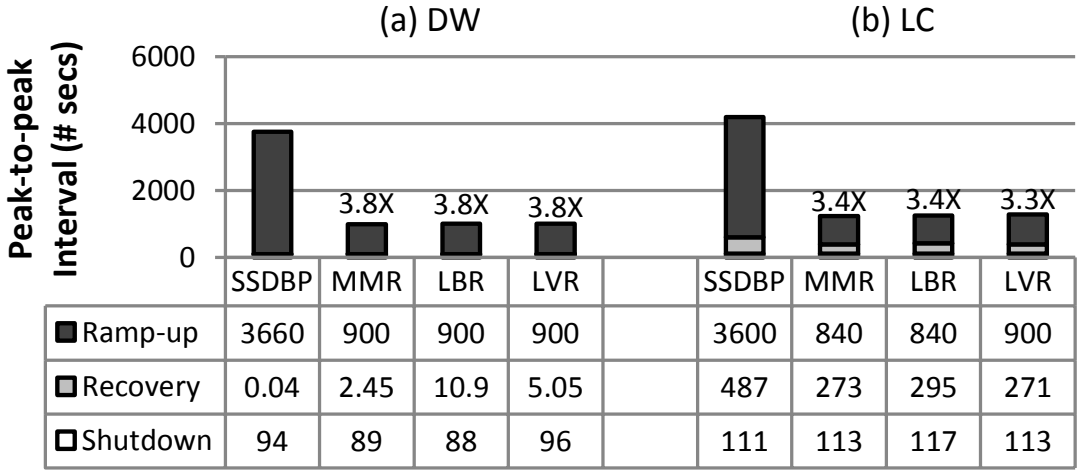
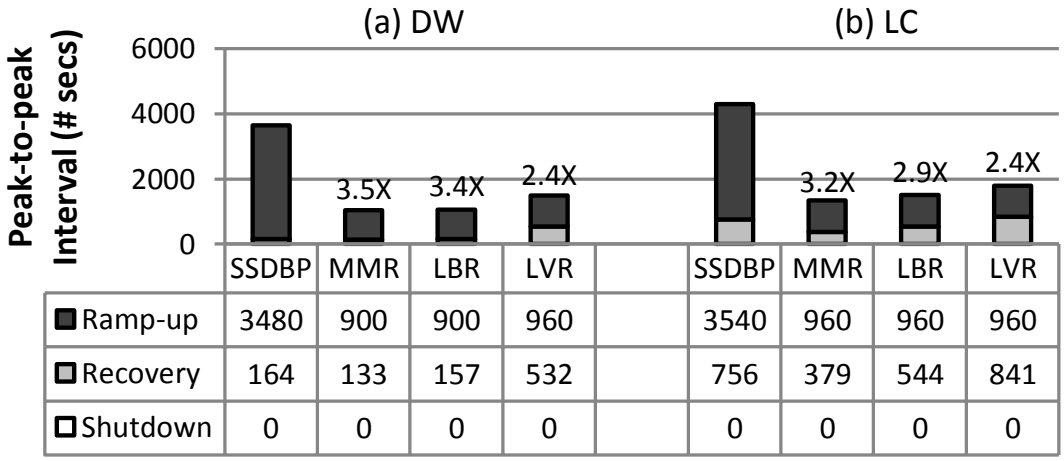


Figure 4.13: (TPC-E) Throughput after restarting from a shutdown. (Restarting from a crash is similar.)



	SSDBP	MMR	LBR	LVR		SSDBP	MMR	LBR	LVR
Ramp-up	3660	900	900	900		3600	840	840	900
Recovery	0.04	2.45	10.9	5.05		487	273	295	271
Shutdown	94	89	88	96		111	113	117	113

Figure 4.14: (TPC-E) Peak-to-peak intervals (in seconds) for the case of restarting from a shutdown.



	SSDBP	MMR	LBR	LVR		SSDBP	MMR	LBR	LVR
Ramp-up	3480	900	900	960		3540	960	960	960
Recovery	164	133	157	532		756	379	544	841
Shutdown	0	0	0	0		0	0	0	0

Figure 4.15: (TPC-E) Peak-to-peak interval (in seconds) for the case of restarting from a crash.

4.4.3 TPC-E Evaluation

4.4.3.1 Sustained Throughput

Figure 4.13 shows the sustained throughput, for TPC-E, after restarting from a shutdown. All the SSD-restart schemes, including MMR, have a sustained throughput very close to that of SSDBP. The reason why MMR did not lose significant performance is due to the fact that TPC-E is not update intensive. So the additional traffic to the SSD, to flush the updates to the SSD buffer table, was less, compared with the TPC-C case. For example, with MMR+DW, the SSD busy time increased only by 5% (contrast 95% - 79% = 16% in the TPC-C case).

4.4.3.2 Peak-to-peak Interval

Next we turn our attention to the peak-to-peak intervals of the SSD-restart schemes, for the case of restarting after a shutdown and after a crash, shown in Figure 4.14 and Figure 4.15, respectively.

In the shutdown case (Figure 4.14), all the SSD-restart schemes have a similar speedup over SSDBP. With the DW policy, the speedup is 3.8X. With the LC policy, the speedup is

around 3.4X.

In the crash case (Figure 4.15), MMR achieved a higher improvement than LBR and LVR. In particular, with the DW policy, MMR had a 3.5X speedup (over SSDBP), and LVR had a 2.4X speedup; with the LC policy, MMR had a 3.2X speedup, and LVR had a 2.4X speedup. In both cases, LBR had a speedup between MMR and LVR. The reason why LVR had a worse speedup is that for crash recovery, LVR had a constant overhead of scanning through the SSD buffer pool (to do deep verification as discussed in Section 4.3.4.4). In our experiments, LVR spent 380 seconds on the deep verification.

4.4.4 Discussion

Each of the three SSD-restart designs has unique characteristics. The MMR method could significantly lower the forward-processing performance if the SSD is a system bottleneck (as can happen for update-intensive workloads such as TPC-C). MMR is, however, simpler to implement than the other two schemes because there is no need to generate new types of log records (as is required for LBR), or require a separate thread to periodically flush the contents in the SSD buffer table (as is required for LVR).

If it is certain that the server only will run read-heavy workloads, MMR may be the best option. Otherwise, both MMR and the LBR designs can hinder the sustained peak performance for different reasons; MMR has to flush every update to the SSD buffer table, and LBR has to flush certain log records to the log disk. But, as we have shown in our experiments the MMR scheme has a bigger (negative) impact on the sustained peak performance (compared to LBR) as it has to flush more often than LBR. We have also seen that both LBR and LVR have a very small impact on the sustained peak performance.

The LBR and LVR methods use different locations for the persistent storage when hardening the contents of the SSD buffer table. The LBR design logs the updates made to the SSD buffer table in the transactional log (which is in general located in HDDs). The LVR design, on the other hand, flushes the SSD buffer table to an SSD-resident file. Depending

on which one is a potential system bottleneck, one design could achieve better throughput than the other (in our all experiments, the performance loss caused by LBR and LVR was within 5%).

However, overall we believe that the LVR method is better than LBR for the following reasons: First, LBR may require a larger log space when processing update-intensive workloads, as it generates log records whenever pages are written to the SSD buffer pool (e.g., on the TPC-C database with LC, the log-space size used by LBR and LVR is 89.6 GB and 70 GB, respectively).

Second, LVR is the only design with the flexibility that hardening the SSD buffer table does not require synchronization with regular forward processing, which in turn implies that it has a smaller impact on the sustained peak performance. In additions, with LVR one can control how frequently the SSD buffer table is hardened, thereby providing a controlled way of putting additional load on the SSD I/O subsystem.

Finally, supporting multiple databases and recovering them in parallel is challenging with LBR. In SQL Server 2012 different databases have different transactional logs, and the checkpoints and recovery of different databases is performed independently. LBR may require the recovery of multiple databases to be synchronized. As a comparison, it is relatively easy to make MMR and LVR support parallel recovery of multiple databases.

4.5 Related Work

Using flash SSDs to extend the buffer pool of a DBMS has been a topic of active research interest (e.g., [17, 24, 31, 52]), and commercial appliance design such as Oracle Exadata [63] and Teradata Virtual Storage System [6]. Flash SSD has also been targeted for other uses in a database management system, including using the SSD to permanently store some of the data in the database (e.g., [23, 28, 49]), storing database transactional logs [54], and as a second level file cache [19]. The SIGMOD'11 tutorial by Koltsidas and Viglas [51] provides a

nice recent overview of data management techniques that leverage flash memory. Restarting from the SSD is a relatively new topic of research, and the first paper on this topic was published by Bhattacharjee *et al.*[18]. Their proposed technique is the MMR technique that we evaluate in this chapter.

4.6 Summary

In this chapter, we evaluated three alternative schemes that leverage the non-volatile feature of flash SSDs by reusing the SSD buffer pool pages after a server restart. These three schemes include the previously proposed MMR method, and two new methods, LBR and LVR, that we propose in this chapter. Each of the three SSD-restart designs was implemented on top of both the previously proposed state-of-the-art methods, DW and LC, buffer pool extension designs. We have carried out a thorough investigation of these methods using both a read-intensive workload (TPC-E) and update-intensive workload (TPC-C). Our results point to the combination of DW and LVR as a leading candidate to enable fast restart from SSD.

Chapter 5

Query Processing on Smart SSDs

Data storage devices are getting "smarter." Smart Flash storage devices (a.k.a. "Smart SSD") are on the horizon and will package CPU processing and DRAM storage inside the Smart SSD and make that available to run user programs inside the Smart SSD. The focus of this chapter is on exploring the opportunities and challenges associated with exploiting this functionality of Smart SSDs for relational analytic query processing. We have implemented an initial prototype of Microsoft SQL Server 2012 running on a Samsung Smart SSD. Our results demonstrate that significant performance and energy gains can be achieved by pushing selected query processing components inside the Smart SSDs. We also identify various changes that SSD device manufacturers can make to increase the benefits of using Smart SSDs for data processing applications, and also suggest possible research opportunities for the database community.

5.1 Introduction

It has generally been recognized that for data intensive applications, moving code to data is far more efficient than moving data to code. Thus, data processing systems try to push code as far below in the query processing pipeline as possible by using techniques such as early selection pushdown and early (pre–)aggregation, and parallel/distributed data processing

systems run as much of the query close to the node that holds the data.

Traditionally these "code pushdown" techniques have been implemented in systems with rigid hardware boundaries that have largely stayed static since the start of the computing era. Data is pulled from an underlying I/O subsystem into the main memory, and query processing code is run in the CPUs (which pulls data from the main memory through various levels of processor caches). Various areas of computer science have focused on making this data flow efficient using techniques such as prefetching, prioritizing sequential access (for both fetching data to the main memory, and/or to the processor caches), and pipelined query execution.

However, the boundary between persistent storage, volatile storage, and processing is increasingly getting blurrier. For example, mobile devices today integrate many of these functionalities into a single chip (the SoC trend). We are now on the cusp of this hardware trend sweeping over into the server world. In this chapter, we focus on the integration of processing power and non-volatile storage in a new class of storage products known as Smart SSDs. Smart SSDs are flash storage devices (like regular SSDs), but ones that incorporate memory and computing inside the SSD device. While SSD devices have always contained these resources for managing the device for many years (e.g., for running the FTL logic), with Smart SSDs some of the computing resources inside the SSD could be made available to run general user-defined programs.

The goal of this chapter is to explore the opportunities and challenges associated with running selected database operations inside a Smart SSD. The potential opportunities here are threefold.

First, SSDs generally have a far larger aggregate internal bandwidth than the bandwidth supported by the host I/O interface (typically SAS or SATA). Today, the internal aggregate I/O bandwidth of high-end Samsung SSDs is about 5X that of the fastest SAS or SATA interface, and this gap is likely to grow to more than 10X (see Figure 5.1) in the near future. Thus, pushing operations, especially highly selective ones that return few result rows, could

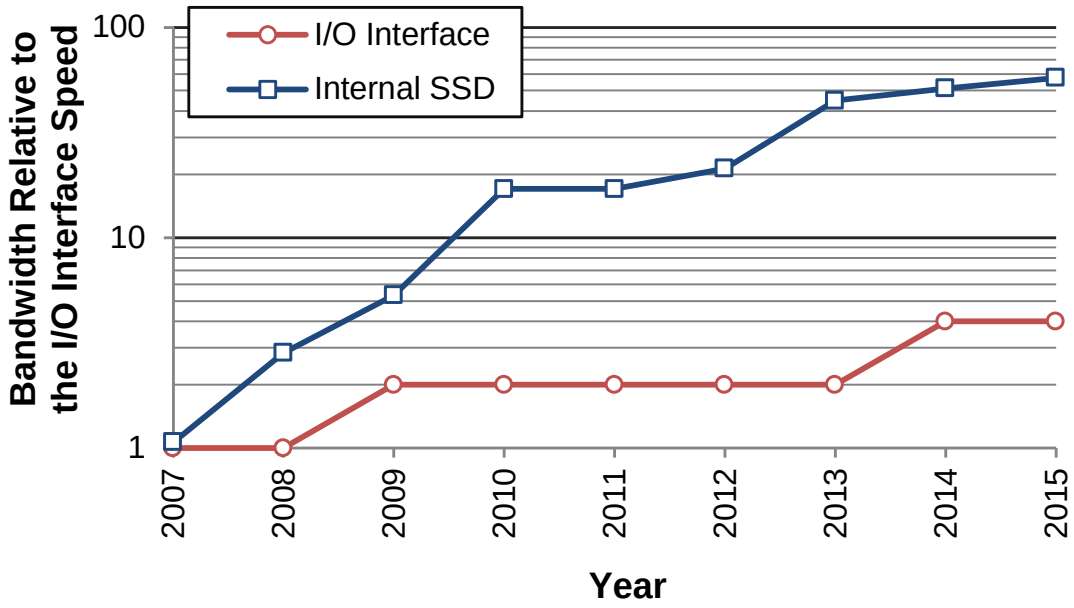


Figure 5.1: Bandwidth trends for the host I/O interface (i.e. SAS/SATA standards), and aggregate internal bandwidth available in high-end enterprise Samsung SSDs. Numbers here are relative to the I/O interface speed in 2007 (375 MB/s). Data beyond 2012 are internal projections by Samsung.

allow the query to run at the speed at which data is getting pulled from the internal (NAND) flash chips. We note that similar techniques have been used in IBM Netezza and Oracle Exadata appliances, but these approaches use additional or specialized hardware that is added right into or next to the I/O subsystem (FPGA for Netezza [32], and Intel Xeon processors in Exadata [62]). In contrast, Smart SSDs have this processing in-built into the I/O device itself, essentially providing the opportunity to "commoditize" a new style of data processing where operations are opportunistically pushed down into the I/O layer using commodity Smart SSDs.

Second, offloading work to the Smart SSDs may change the way in which we build balanced database servers and database appliances. If some of computation is done inside the Smart SSD, then one can reduce the processing power that is needed in the host machine, or increase the effective computing power of the servers or appliances. Smart SSDs use simpler processors like ARM, that are generally cheaper (from the $/MHz perspective) than the

processors used in a server. Thus, database servers and appliances that use Smart SSDs could be more efficient from the overall price/performance perspective.

Finally, pushing processing into the Smart SSDs can reduce the energy consumption of the overall database server/appliance. The energy efficiency of query processing can be improved by reducing its running time and/or by running processing on the low power processors that are typically packaged inside the Smart SSDs. Lower energy consumption is not only environmentally friendly, but often leads to a reduction in the total cost of operating the database system. In addition, with the trend towards database appliances, energy starts becoming an important deployment consideration when the database appliances are installed in private clouds on premises where getting additional (many kilowatts of) power is challenging.

To explore and quantify these potential advantages of using Smart SSDs for DBMSs, we have started an exploratory project to extend Microsoft SQL Server to offload database operations onto a Samsung Smart SSD. We wrote simple selection and aggregation operators that are compiled into the firmware of the SSD. We also extended the execution framework of SQL Server to develop a simple (but with limited functionality) working prototype in which we could runs simple selection and aggregation queries end-to-end. Our results show that for this class of queries, we observed up to 2.1X improvement in end-to-end performance compared to using the same SSDs but without the "Smart" functionality, and up to a 2.6X reduction in energy consumption. These early results, admittedly on queries using a limited subset of SQL (e.g., no joins), demonstrate that there are potential opportunities for using Smart SSDs even in mature commercial and well-optimized relational DBMSs.

Our results also point out that there are a number of challenges, and hence research opportunities, in this new area of running data processing programs inside the Smart SSDs.

First, the processing capabilities available inside the Smart SSD that we used are very limited by design. It is clear from our results that adding more computing power into the Smart SSD (and making it available for query processing) could further increase both

performance and energy savings. However, the SSD manufacturers will need to determine if it is economical and technically feasible to add more processing power – issues such as additional cost per device and changes in the device energy profile must be considered. In a sense, this is a chicken-and-egg problem since the SSD manufacturers will add more processing power only if more software makes use of an SSD's "smart" features while the software vendors need to become confident in the potential benefits before investing the necessary engineering resources. We hope that our work provides a starting point for such deliberations.

The firmware development process we followed to run the user code in the Smart SSDs is rudimentary. This can be a potential challenge to general application developers. Before Smart SSDs can be broadly adopted the existing development and debugging tools and runtime system (Section 5.3) need to be much more user-friendly. Further, the ecosystem around the Smart SSDs including communication protocols and the programming, runtime, and usage models need to be investigated in-depth.

Finally, the query execution engine and query optimizer of the DBMS must be extended to determine when to push an operation to the SSD. Implications of running operations in the Smart SSDs may also influence DBMS buffer pool caching policies, and may require re-examining how aspects such as database compression are used.

The remainder of this chapter is organized as follows: The architecture of a modern SSD is presented in Section 5.2. In Section 5.3 we describe how Smart SSDs work. Our current key experimental results are contained in Section 5.4. Section 5.5 contains our summary.

5.2 SSD Architecture

Figure 5.2 illustrates the general internal architecture of modern SSDs. There are three major components: SSD controller, flash memory array, and DRAM. The SSD controller has four key subcomponents: host interface controller, embedded processors, DRAM controller, and flash memory controllers. The host interface controller implements a bus interface protocol

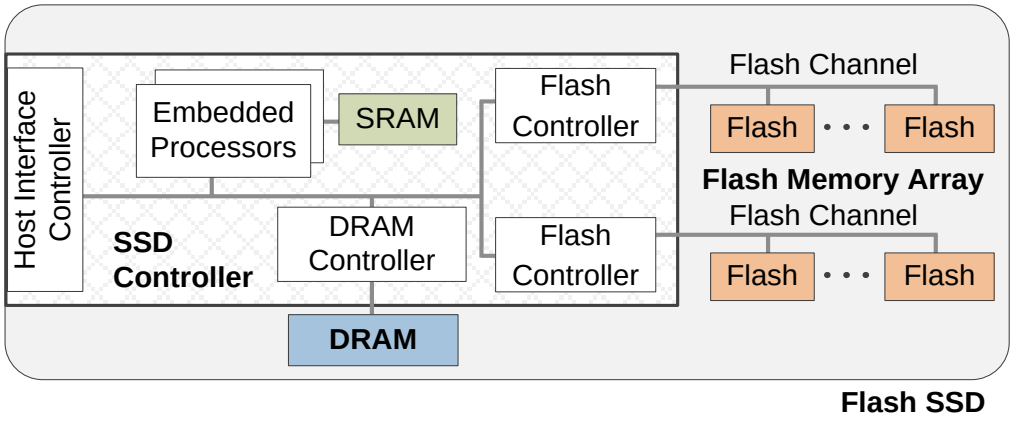


Figure 5.2: Architecture of a modern SSD

such as SATA, SAS, or PCI Express (PCIe). The embedded processors are used to execute the SSD firmware code that runs the host interface protocol, and also runs the Flash Translation Layer (FTL), which maps Logical Block Address (LBA) in the host OS to the Physical Block Address (PBA) in the flash memory. Time-critical data and program code is stored in the SRAM. The processor of choice is typically a low-powered 32-bit RISC processor, like an ARM series processor, which typically has multiple cores. The controller also has on-board DRAM memory that has higher capacity (but also higher access latency) than the SRAM. The flash memory controller is in charge of data transfer between the flash memory and DRAM. Its key functions include running the Error Correction Code (ECC) logic, and the Direct Memory Access (DMA). To obtain higher I/O performance from the flash memory array, the flash controller uses chip-level and channel-level interleaving techniques.

The NAND flash memory array is the persistent storage medium. Each flash chip has multiple blocks, each of which holds multiple pages. The unit of erasure is a block, while the read and write operations in the firmware are done at the granularity of pages.

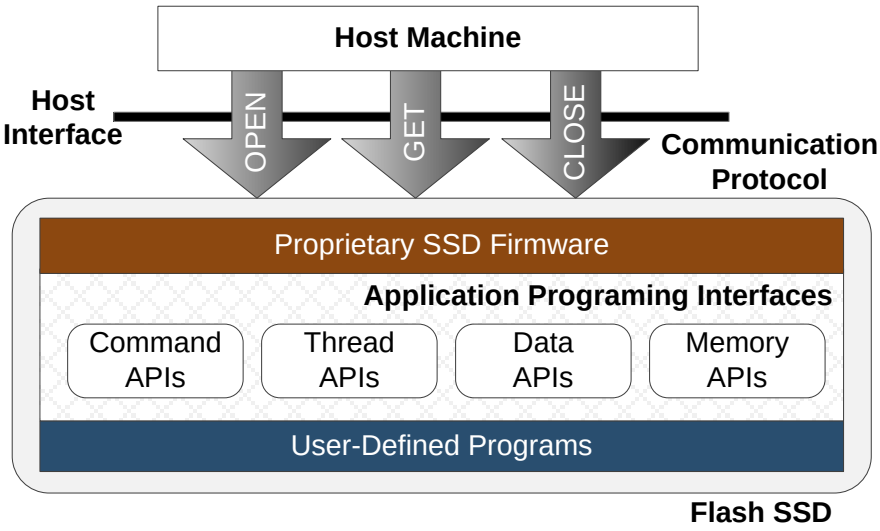


Figure 5.3: Smart SSD RunTime Framework

5.3 Smart SSDs for Query Processing

The Smart SSD runtime framework (shown in Figure 5.3) implements the core of the software ecosystem that is needed to run user-defined programs in the Smart SSDs.

5.3.1 Communication Protocol

Since the key concept of the Smart SSD is to convert a regular SSD into a combined computing and storage device, we needed a standard mechanism to enable the processing capabilities of the device at run-time. We have developed a simple session-based protocol that is compatible with the standard SATA/SAS interfaces (but could be extended for PCIe). The protocol consists of three commands – OPEN, GET, and CLOSE.

5.3.2 Application Programming Interface (API)

Once a command is successfully delivered to the device through the Smart SSD communication protocol (Section 5.3.1), the Smart SSD runtime system drives the user-defined program in an event-driven fashion via the Smart SSD APIs. The design philosophy of the APIs is to

give more flexibility to the programs, so that it is easy for end-user programs to use these APIs.

5.4 Evaluation

In this section, we show our key end-to-end results.

5.4.1 Experimental Setup

5.4.1.1 Workloads

For our experiments, we used the LINEITEM table defined in the TPC-H benchmark [9]. Our modifications to the original specifications are as follows: 1) we used a fixed-length char string for the variable-length column, L_COMMENT, 2) all decimal numbers were multiplied by 100 and stored as integers, 3) all date values were converted to the numbers of days since epoch. These changes resulted in 148 byte tuples. The LINEITEM data was populated at a scale factor of 100 (600M tuples, 90 GB), and inserted into a SQL Server heap table (without a clustered index). By default, the tuples are stored in slotted pages using the traditional N-ary Storage Model (NSM). For the Smart SSDs, we also implemented the PAX layout [15] in which all the values of a column are grouped together within a page.

5.4.1.2 Hardware/Software Setup

All experiments were performed on a system running 64bit Windows 7 with 32 GB of DRAM (24 GB of memory is dedicated to the DBMS). The system has two Intel Xeon E5430 2.66GHz quad core processors, each of which has a 32 KB L1 cache, and two 6 MB L2 caches shared by two cores. For the OS and the transactional log, we used two 7.5K RPM SATA HDDs, respectively. In addition, we used a LSI four-port SATA/SAS 6Gbps HBA (host bus adapter) for the three storage devices we used for testing: 1) a 146 GB 10K RPM SAS HDD, 2) a 400 GB SAS SSD, and 3) a Smart SSD prototyped on the same SSD. Only one of three

	SAS HDD	SAS SSD	(Internal) Smart SSD
Seq. Read (MB/sec)	80	550	1,560

Table 5.1: Maximum sequential-read bandwidth with 32-page (256KB) I/Os.

devices is connected to the HBA at a time for each experiment. Finally, system power draw was measured using a Yokogawa WT210 unit (as suggested in [13]). We used this server hardware since it was compatible with the LSI RAID controller card that was needed to run the extended host interface protocol described in Section 3.1. We recognize that this box has a very high base energy profile (235W in the idle state) for our setting in which we use a single data drive; hence, we expect the energy gains to be bigger when the Smart SSD is used with a more balanced hardware configuration. But, this configuration allowed us to get initial end-to-end results. We implemented simple selection and selection with aggregation queries in the Smart SSD by using the Smart SSD APIs (Section 5.3.2). We also modified some components in SQL Server 2012 to recognize and communicate with the Smart SSD through the Smart SSD communication protocol (Section 5.3.1). For each test, we measured the elapsed wall-clock time, and calculated the server energy consumption by summing the time discretized real energy values over the elapsed time. After each test run, we dropped the pages in the main-memory buffer pool to start with a cold buffer cache on each run.

5.4.2 Preliminary Results

To aid the analysis of the results that are presented below, the I/O characteristics of the HDD, SSD, and Smart SSD are shown in Table 5.1. The bandwidth of the HDD and the SSD was obtained using Iometer [3]. The Smart SSD internal bandwidth was calculated by measuring the elapsed time to sequentially read 100 GB of dummy data inside the SSD (this is the internal bandwidth that is available to a user program). As can be seen in Table 5.1, the sequential read bandwidth of the Smart SSD is 19.5X and 2.8X faster than that of the HDD and the SSD, respectively. This value can be used as the upper bound of the

performance gains that the Smart SSD could potentially deliver.

5.4.2.1 Selection Query

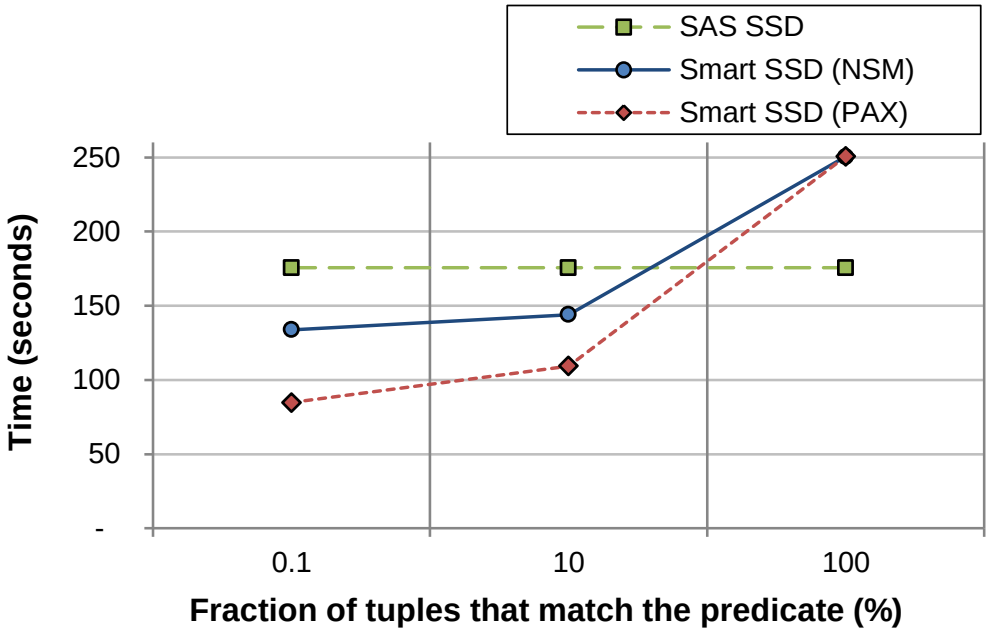
For this experiment, we used the following SQL query:

- **select** l_orderkey **from** lineitem **where** l_orderkey < [value]

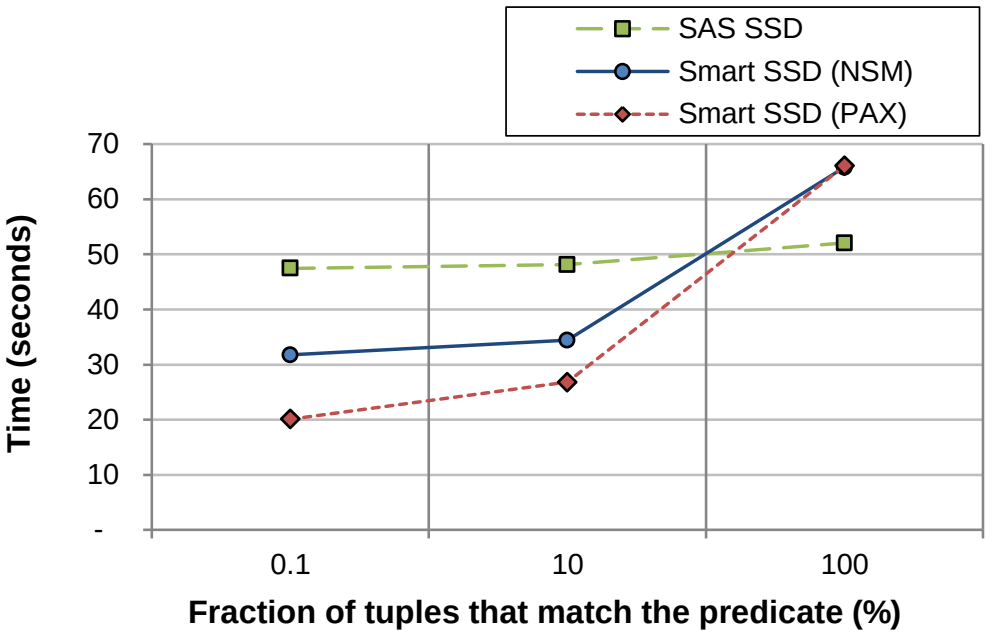
Figure 5.4 (a) and (b) present the end-to-end elapsed time and energy consumption to execute the selection query at various selectivity factors. To improve the presentation of Figure 5.4 (a) and (b), we do not show the measurements for the HDD case. For Figure 5.4 (a) the run time with the HDD was 1,188 seconds across all the selectivity factors, and in Figure 5.4 (b) the HDD configuration consumed around 295 kJoules across all the selectivity factors.

As can be seen in Figure 5.4 (a), the Smart SSD provides significant improvements in performance for the highly selective queries (i.e. where few tuples match the selection predicate): up to 14X and 2.1X over the HDD and SSD, respectively when 0.1% of the tuples satisfy the selection predicate.

One interesting observation for the Smart SSD case is that using the PAX layout provides better performance than NSM, by up to 58%. As an example, for the 0.1% selection query, the elapsed times when using NSM and PAX are about 134 seconds and 85 seconds, respectively. Unlike the host processor that has L1/L2 caches, the embedded processor in our Smart SSD does not have these caches. Instead, it provides an efficient way to move consecutive bytes from the memory to the processor registers in a single instruction (i.e., the LDM instruction [12]). Since all the values of a column are stored contiguously in the case of the PAX layout, we were able to use the LDM instruction to load multiple values at once, reducing the number of memory accesses. Given the high DRAM latency in the SSD (Section 5.2), the columnar PAX layout is more suitable than a row-based layout. In addition, we also noticed that the



(a) Elapsed Time



(b) Energy Consumption

Figure 5.4: End-to-end (a) elapsed time and (b) energy consumption for a selection query on LINEITEM (100 SF) at various selectivity factors.

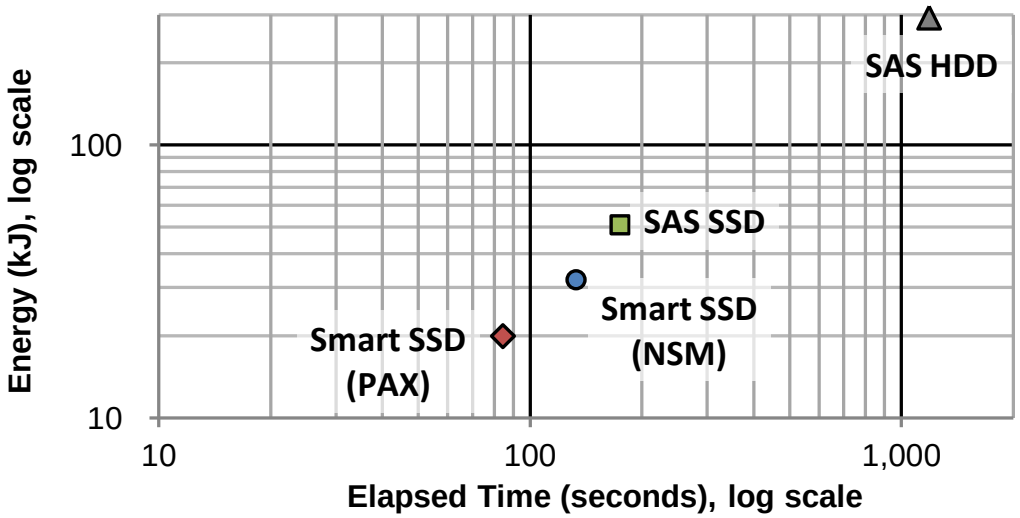


Figure 5.5: Elapsed time and energy consumption for a selection with AVERAGE query on LINEITEM (100 SF) with 1% selectivity.

Smart SSD provides a big energy efficiency benefits – up to 14.5X and 2.4X over the HDD and SSD respectively, with 0.1% selectivity; see Figure 5.4 (b).

5.4.2.2 Selection with Aggregation Query

For this experiment, we used the following SQL query:

- **select avg**(l_orderkey) **from** lineitem **where** l_orderkey < [value]

The results for this experiment when 1% of the tuples satisfy the selection predicate are shown in Figure 5.5. Similar to the previous results, the Smart SSD shows significant (> 2X) performance and energy savings over the HDD and the SSD.

5.4.3 Discussion

The energy gains are likely to be much bigger with more balanced host machines than our test-bed machine. Recall from the discussion in Section 5.4.2.1 that we observed 14.5X and 2.4X energy gains for the entire system, over the HDD and the SSD, respectively. If we only

consider the energy consumption over the base idle energy, then these gains are 19.4X and 10.6X over the HDD and the SSD, respectively. In addition, a Smart SSD can benefit from the amortization of the high static energy consumption over multiple disks in real database appliances, leveraging this low energy profile at runtime.

In addition, we noticed that the processing capabilities inside the Smart SSD quickly became a performance bottleneck, in particular when the selection predicate matches many input tuples. For example, as seen in 5.4 (a), when all the tuples match the selection predicate (i.e., the 100% point on the x-axis), compared to the regular SSD the query runs 43% slower on the Smart SSD. In this case, the low-performance embedded processor without L1/L2 caches and the high latency cost for accessing the DRAM memory quickly became bottlenecks.

The development environment for running code inside the Smart SSD needs further development – much of the tool was developed hand-in-hand with this project. For instance, reaching a 2X improvement in performance required detailed planning of the layout of the data structures used by the code running inside the Smart SSD to avoid having crucial data structures spill out of the SRAM. Similarly, we used a hardware-debugging tool called Trace32, a JTAG in-circuit (ICD) debugger [10], which is far more primitive than the regular debugging tools (e.g., Visual Studio) available to database systems developers.

5.5 Summary

The results in this chapter show that Smart SSDs have the potential to play an important role when building high-performance database systems/appliances. Our end-to-end results using SQL Server and a Samsung Smart SSD demonstrated significant performance benefits (> 2X in some cases) and a significant reduction in energy consumption for the entire server (> 2X reduction in some cases) over a regular SSD. While we acknowledge that these results are preliminary (we only tested a limited class of queries and on only one server configuration), we also feel that there are potential new opportunities for crossing across the traditional

hardware and software boundaries with Smart SSDs.

A significant amount of work remains. On the SSD vendor side, the existing tools for development and debugging must be improved if Smart SSDs are to have a bigger impact. We also found that the hardware inside our Smart SSD device is limited, and that the CPU quickly became a bottleneck as the Smart SSD that we used was not designed to run general purpose programs. The next step must be to add in more hardware (CPU, SRAM and DRAM) so that the DBMS code can run more effectively inside the SSD. These enhancements are absolutely crucial to achieving the 10X or more benefit that Smart SSDs have the potential of providing (see Figure 5.1). The hardware vendors must, however, figure out how much hardware they can add to fit both within their manufacturing budget (Smart SSDs still need to ride the "commodity" wave) and associated power budget for each device.

On the software side, the DBMS vendors need to carefully weigh the pros-and-cons associated with using smart SSDs. Significant software development and testing time will be needed to fully exploit the functionality offered by Smart SSDs. There are many interesting research and development issues that need to be further explored, including extending the query optimizer to push operations to the Smart SSD, designing algorithms for various operators that work inside the Smart SSD, considering the impact of concurrent queries, examining the impact of running operations inside the Smart SSD on buffer pool management, etc. To make these longer-term investments, DBMS vendors will likely need the hardware vendors to remove the existing roadblocks.

Overall, the computing hardware landscape is changing rapidly. Smart SSDs present an interesting additional new axis for thinking about how to build future high-performance database servers/appliances, and this current work shows that there is a lot of potential for database hardware and software vendors to come together to explore this opportunity.

Chapter 6

Conclusions

This dissertation has explored and evaluated various architectures that employ flash solid state drives (SSDs) to improve the performance of database management systems (DBMSs). In particular, the work presented in this dissertation has focused on how SSDs can be effectively integrated into existing DBMSs to improve database query processing performance.

6.1 Contributions

6.1.1 Role I: Permanent Storage

The appeal of SSDs as an alternative to magnetic hard disk drives (HDDs) is attracting interest in redesigning various aspects of DMBS internals that are largely tuned for HDDs. Before redesigning them, however, we should first focus on the lessons that we have learnt from over three decades of optimizing database query processing based on the disk characteristics, and determine if these lessons also apply to SSDs. Chapter 2 investigated some of the lessons in the context of four well-known ad hoc join algorithms, namely block nested loops join, sort-merge join, Grace hash join, and hybrid hash join on both HDDs and SSDs. Based on the observations, we concluded that i) buffer allocation strategy has a critical impact on the performance of join algorithms for both HDDs and SSDs, ii) blocked I/O continues to play an

important role for SSDs despite the absence of mechanical moving parts, and iii) both CPU times and I/O costs must be considered when comparing the performance of join algorithms as the CPU times can be a dominating component of the overall join cost when using SSDs.

6.1.2 Role II: Main Memory Extension

Today's DBMSs use HDDs as their primary means of storage, and DRAM as volatile storage to cache frequently used data items. However, in data sets with large working sets, only a relatively small amount of data can be stored in DRAM because of its high cost. In such cases, the query performance speed is gated by by accesses to the underling (far slower) spinning disk subsystems. To remedy this performance bottleneck, Chapter 3 presented several designs to use SSDs as an extension to the main-memory DBMS buffer pool (i.e., SSD buffer pool). An evaluation of these designs showed that these designs can achieve significant performance improvements over a pure HDD-based I/O subsystem. We also uncovered a number of factors can affect the overall speedup, including the gap in the random I/O performance between the SSDs and the HDDs, and the ratio of the working set size of a database over the SSD buffer pool size.

One interesting observation from the buffer-pool extension work is that after a system restart, it takes a very long time (i.e., the ramp-up time) to warm-up the SSD buffer pool with useful data. As a result, it take a long time to reach peak performance after server restart. In Chapter 4, we investigated various schemes (that work with the SSD buffer pool designs proposed in Chapter 3) to reuse the pages that were cached in the SSD buffer pool before the restart event. We carried out a thorough investigation of these schemes, and our experimental results showed that the ramp-up time was dramatically reduced with negligible performance loss.

6.1.3 Role III: In-Storage Processing

Finally, in Chapter 5, we focused on the integration of processing power and non-volatile storage in a new class of storage products known as *Smart SSDs*. Smart SSDs are regular SSDs, but additionally have computing and memory inside the devices that is made available to run general user-defined programs. In the context of a simple class of SQL queries, namely selection queries and queries with selection and aggregation, Chapter 5 discussed the opportunities and challenges associated with running these queries inside the Smart SSDs. Our results demonstrated that in some cases, running selected query processing inside the Smart SSD can improve the overall query performance, and also reduce the overall energy that is used to run the query.

6.2 Future Directions

There are interesting directions for future work associated with each component of this thesis.

The work in Chapter 2 discusses the performance characteristics of four well-known ad hoc join algorithms on one specific HDD and one specific SSD. Thus, an interesting direction for future work is to expand the range of hardware to include enterprise-level HDDs and SSDs. Furthermore, deriving detailed analytical models for existing join algorithms on SSDs would be helpful to explore, since the optimal buffer allocations for SSDs may differ from those for disks. Using SSDs as a buffer pool extension (the methods presented in Chapters 3 and 4) has a number of implications on other aspects of a DBMS. One observation from our work is that unless a large fraction of the random I/Os are offloaded to the SSDs, it is likely that HDDs will remain as a system bottleneck. In such scenario, the performance of using mid-range (cheap) SSDs may be on par with that of using expensive high-end enterprise SSDs. It would also be interesting to generalize that work to a hierarchy of non-volatile storage with different price v/s performance characteristics, and to extend that work to general data processing environments outside the traditional DBMS boundary (e.g., HDFS [20], or Hive

[1]) Finally, the work in Chapter 5 presents a new style of data processing where database operations are pushed down into Smart SSDs. This work can be expanded to cover the full range of database operations including joins, and to use multiple Smart SSDs in parallel.

Bibliography

[1] Hive. http://hive.apache.org.

[2] HLNAND. http://www.hlnand.com.

[3] Iometer. http://www.iometer.org.

[4] Microsoft SQL Server. http://www.microsoft.com/sqlserver/.

[5] SQLite3. http://www.sqlite.org.

[6] Teradata Virtual Storage. http://www.teradata.com/t/brochures/Teradata-Virtual-Storage-eb5944.

[7] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc.

[8] TPC Benchmark E (TPC-E). http://www.tpc.org/tpce.

[9] TPC Benchmark H (TPC-H). http://www.tpc.org/tpch.

[10] Trace32, Lauterbach. http://www.lauterbach.com.

[11] SSDs: Still not a Solid State Business. http://www.eetimes.com/electronics-news/4206361/SSDs--Still-not-a--solid-state--business, 2010.

[12] ARM Developer Suite. http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/DUI0068.pdf, 2011.

[13] Power and Temperature Measurement Setup Guide. http://spec.org/power/docs/SPEC-Power_Measurement_Setup_Guide.pdf, 2012.

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

[15] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.

[16] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan-Kauffman Publishers, 2nd edition edition, 2009.

[17] B. Bhattacharjee, M. Canim, C. A. Langand, G. A. Mihaila, and K. A. Ross. Storage Class Memory Aware Data Management. *IEEE Data Eng. Bull.*, 33(4):35–40, 2010.

[18] B. Bhattacharjee, K. A. Ross, C. A. Lang, G. A. Mihaila, and M. Banikazemi. Enhancing Recovery using an SSD Buffer Pool Extension. In *DaMoN*, pages 10–16, 2011.

[19] R. Bitar. Deploying Hybrid Storage Pools with Sun Flash Technology and the Solaris ZFS File System. `http://www.oracle.com/technetwork/systems/archive/o11-077-deploying-hsp-487445.pdf`, 2011.

[20] D. Borthakur. The Hadoop Distributed File System: Architecture and Design. `http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf`, 2007.

[21] L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. In *CIDR*, 2009.

[22] K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *VLDB*, pages 323–333, 1984.

[23] M. Canim, B. Bhattacharjee, G. A. Mihailaand, C. A. Lang, and K. A. Ross. An Object Placement Advisor for DB2 Using Solid State Storage. *PVLDB*, 2(2):1318–1329, 2009.

[24] M. Canim, G. A. Mihaila, B. Bhattacharjee, , K. A. Ross, and C. A. Lang. SSD Bufferpool Extensions for Database Systems. *PVLDB*, 3(2):1435–1446, 2010.

[25] Computer World. HP Offers SSDs in ProLiant Server Line. `http://www.computerworld.com/s/article/9138209/HPoffersSSDsinProLiantserverline`, 2008.

[26] G. P. Copeland, W. Alexander, E. E. Boughter, and T. W. Keller. Data Placement In Bubba. In *SIGMOD*, pages 99–108, 1988.

[27] B. Dees. Native Command Queuing - Advanced Performance in Desktop Storage. *IEEE Potentials*, 24(4):4 – 7, 2005.

[28] J. Do and J. M. Patel. Join Processing for Flash SSDs: Remembering Past Lessons. In *DaMoN*, pages 1–8, 2009.

[29] J. Do, D. Zhang, J. M. Patel, and D. J. DeWitt. Racing to the Peak: Fast Restart for SSD Buffer Pool Extension. In *NEDB Summit*, 2012.

[30] J. Do, D. Zhang, J. M. Patel, and D. J. DeWitt. Fast Peak-to-Peak Restart for SSD Buffer Pool Extension. In *ICDE*, 2013.

[31] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging DBMS buffer pool using SSDs. In *SIGMOD*, pages 1113–1124, 2011.

[32] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. http://www.redbooks.ibm.com/redpapers/pdfs/redp4725.pdf, 2011.

[33] R. F. Freitas and W. W. Wilcke. Storage-Class Memory: The Next Storage System Technology. *IBM Journal*, 52(4):439–447, 2008.

[34] G. Graefe. The Five-Minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *DaMoN*, page 6, 2007.

[35] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King. http://research.microsoft.com/en-us/um/people/gray/talks/FlashisGood.ppt, 2006.

[36] J. Gray and B. Fitzgerald. Flash Disk Opportunity for Server Applications. *ACM Queue*, 6(4):18–23, 2008.

[37] J. Gray and A. Reuter. *In Transaction Processing: Concepts and Techniques, Chapter 12: Advanced Transaction Manager Topics*. Morgan Kaufmann Publishers, 1993.

[38] R. F. Green, W. N. Brandt, D. E. V. Berk, D. P. Schneider, and P. S. Osmer. AGN Science with the Large Synoptic Survey Telescope. *ASPCS*, 373:707–714, 2007.

[39] E. Grochowski and R. D. Halem. Technological Impact of Magnetic Hard Disk Drives on Storage Systems. *IBM Systems Journal*, 42(2):338–346, 2003.

[40] L. M. Haas, M. J. Carey, M. Livny, and A. Shukla. Seeking the Truth About ad hoc Join Costs. *VLDB J.*, 6(3):241–256, 1997.

[41] B. Hayes. Graph Theory in Practice. *American Scientist*, 88(2):104–109, 2000.

[42] S. R. Hetzler. The Storage Chasm: Implications for the Future of HDD and Solid State Storage. In *IBM Journals*, 2009.

[43] A. L. Holloway. *In Adapting Database Storage for New Hardware, Chapter 4: Extending the Buffer Pool with a Solid State Disk*. PhD thesis, University of Wisconsin-Madison, 2009.

[44] C. Hwang. Nanotechnology Enables a New Memory Growth Model. *Proceedings of the IEEE*, 91(11):1765 – 1771, Nov. 2003.

[45] Information Week. Google Plans to Use Intel SSD Storage in Servers. http://www.informationweek.com/news/storage/systems/showArticle.jhtml?articleID=207602745, 2008.

[46] S.-H. Kim, D. Jung, J.-S. Kim, and S. Maeng. HeteroDrive: Reshaping the Storage Access Pattern of OLTP Workload Using SSD. In *IWSSPS*, 2009.

[47] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and Its Architecture. *New Generation Comput.*, 1(1):63–74, 1983.

[48] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[49] I. Koltsidas and S. Viglas. Flashing Up The Storage Layer. *PVLDB*, 1(1):514–525, 2008.

[50] I. Koltsidas and S. Viglas. The Case for Flash-Aware Multi-Level Caching. Technical report, University of Edinburgh, 2009.

[51] I. Koltsidas and S. Viglas. Data Management over Flash Memory. In *SIGMOD*, pages 1209–1212, 2011.

[52] I. Koltsidas and S. Viglas. Designing a Flash-Aware Two-Level Cache. In *ADBIS*, pages 153–169, 2011.

[53] S.-W. Lee and B. Moon. Design of Flash-Based DBMS: an In-Page logging Approach. In *SIGMOD*, pages 55–66, 2007.

[54] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *SIGMOD*, pages 1075–1086, 2008.

[55] S. T. Leutenegger and D. M. Dias. A Modeling Study of the TPC-C Benchmark. In *SIGMOD*, pages 22–31, 1993.

[56] C. Li, K. Shen, and A. E. Papathanasiou. Competitive Prefetching for Concurrent Sequential I/O. In *EuroSys*, pages 189–202, 2007.

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

[58] C. Mohan, D. J. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[59] D. Myers. *On the Use of NAND Flash Memory in High-Performance Relational Databases*. Master's thesis, MIT, 2008.

[60] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. I. T. Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *EuroSys*, pages 145–158, 2009.

[61] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD*, pages 297–306, 1993.

[62] Oracle. A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. `http://www.oracle.com/technetwork/database/exadata/exadata-technical-whitepaper-134575.pdf`, 2012.

[63] Oracle. Exadata. `http://www.oracle.com/us/products/database/exadata`.

[64] M. Polte, J. Simsa, and G. Gibson. Comparing Performance of Solid State Devices and Mechanical Disks. In *PDS Workshop*, 2008.

[65] M. Polte, J. Simsa, and G. Gibson. Enabling Enterprise Solid State Disks Performance. In *WISH*, 2009.

[66] M. A. Shah, S. Harizopoulos, J. L. Wiener, and G. Graefe. Fast Scans and Joins using Flash Drives. In *DaMoN*, pages 17–24, 2008.

[67] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Trans. Database Syst.*, 11(3):239–264, 1986.

[68] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query Processing Techniques for Solid State Drives. In *SIGMOD*, pages 59–72, 2009.

[69] C.-H. Wu, T.-W. Kuo, and L.-P. Chang. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. *ACM Trans. Embedded Comput. Syst.*, 6(3), 2007.