Data Processing Using Flash Storage: Some Opportunities and Limitations

by

Kwanghyun Park

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 05/27/2016

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

Jignesh M. Patel, Professor, Computer Sciences

Jeffrey F. Naughton, Professor, Computer Sciences

AnHai Doan, Professor, Computer Sciences

Paraschos Koutris, Assistant Professor, Computer Sciences

Yang-Suk Kee, Director of Samsung Memory Solutions Lab, Electrical Engineering

ACKNOWLEDGMENTS

It has been an intensive period of learning not just in the academic realm but also in personal growth. After an in-depth period in academia, I write this note of thanks a final touch to my thesis.

I wish to sincerely thank my great advisor Professor Jignesh M. Patel. He simultaneously serves as my academic tutor and personal counselor. He has truly taken care of me within and without academia. His insights and wisdom come through in every paragraph and sentence of this thesis.

I would also like to thank all of my committee members. By introducing many different views to this thesis, Jeffrey F. Naughton helped me broaden my research perspectives. Yang-Suk Kee is one of the important people who has supported this thesis for a long time. I also thank AnHai Doan and Paris Koutris for being on my committee and giving me valuable feedback.

I am also grateful for the huge support I've received from Microsoft's Jim Gray Systems Lab (GSL). During my four-year RA-ship at GSL, David J. DeWitt has always guided me in the right direction. I am very proud to have worked with him for such a long time. I wish to present my special thanks to all the people at GSL — Alan, Nikhil, Willis, Karthic, Rimma, and Jaeyoung. Other important collaborators on this thesis are the people at Samsung Memory Solutions Lab — Dongchul, Heekwon, and Yangwook.

I wish to thank my friends from the University of Wisconsin-Madison database group — Avrilia, Ian, Jessie, Arun, Bruhathi, Lalitha, Justin, Rasiga, Venkiteswaran, and Vinitha. I have enjoyed our discussions and learned a great deal from them and from their individual presentations. I count it a great honor to have been a member of this remarkable group.

Whenever stressed by school, I have been able to refresh myself by playing tennis with the members of the Korean Tennis Club (KOTEL). I thank all of them.

I could never have successfully finished my thesis without the devoted support I've received from my parents and sister. I deeply thank them for their staunch support, which I've received all my life.

Finally, my true love, Mina — you are my best friend and soulmate. Without you, I could not imagine any minute in my life at Madison.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Page

Page

ABSTRACT

In many data intensive workloads, I/O is a key bottleneck. In a storage hierarchy in a canonical database system, non-volatile storage devices (e.g., hard disk drives and flash solid state drives) are used as permanent data storage subsystems, whereas volatile storage devices (e.g., DRAM and CPU registers) are used to stage data from the non-volatile storage for processing by the CPU. Under this hardware architecture, non-volatile storage is connected to the rest of the system via common host I/O interfaces, such as SAS, SATA, and PCIe. Data movement cost through these I/O interfaces has become the largest performance bottleneck for many data intensive workloads. Therefore, in this thesis we explore alternative solutions to achieve high performance data processing by reducing the (expensive) data movement cost across the I/O interfaces.

In the first and second parts of this thesis, we propose a "code push down" technology to reduce the data movement cost from flash solid state drives (SSDs) to DRAM. We use the computation capability of the SSD device to push down selected database operations into the SSD devices, thereby dramatically reducing the actual data movement cost through host I/O interfaces.

Another alternative solution that we propose in the third part of this thesis is to preload necessary data (i.e., hot data) from disks to DRAM before the user query actually requests the data. In this part of thesis, we focus on how to load hot data efficiently at system restart, which could save the data movement cost at query time.

Collectively this thesis discusses some opportunities for using SSDs in data processing platforms, and develops insights about the current limitations and potential future opportunities for

using the computational processing power inside SSDs to alleviate the I/O bottleneck in data intensive workloads.

Chapter 1

Introduction

In many database system settings, I/O is a bottleneck when running data intensive workloads. Figure 1.1 shows the volatile and non-volatile storage hierarchy in a canonical database system. Non-volatile storage devices (e.g., hard disk drives and flash solid state drives) are used as permanent data storage subsystems, whereas volatile storage devices (e.g., DRAM and CPU registers) are used to stage data from the non-volatile storage for processing by the CPU. The CPU typically pulls data from DRAM to CPU registers through various levels of processor caches (e.g., L1, L2, and L3 caches). Under this hardware architecture, non-volatile storage is connected to the rest of the system via common host I/O interfaces, such as SAS, SATA, and PCIe. Data movement cost through these I/O interfaces has become the largest performance bottleneck for many data intensive workloads. Therefore, in this thesis, we explore alternative solutions to achieve high performance data processing by reducing the (expensive) data movement cost across the I/O interfaces.

The first solution that we propose is a "code push down" technology to reduce the data movement from flash solid state drives (SSDs) to DRAM. Modern SSDs often have embedded computing capabilities. As described in Figure 1.2, we use this computation capability to push down selected database operations into the SSD. As a result, the actual data movement cost through the I/O interfaces can be significantly reduced in some cases. In Chapters 2 and 3, we study how to realize this code push down paradigm with current SSD devices for data applications running in both relational (SQL Server) and non-relational (Hadoop) data processing systems. Using that study, we explore the potentials and challenges in using embedded computing in SSDs for data processing workloads.

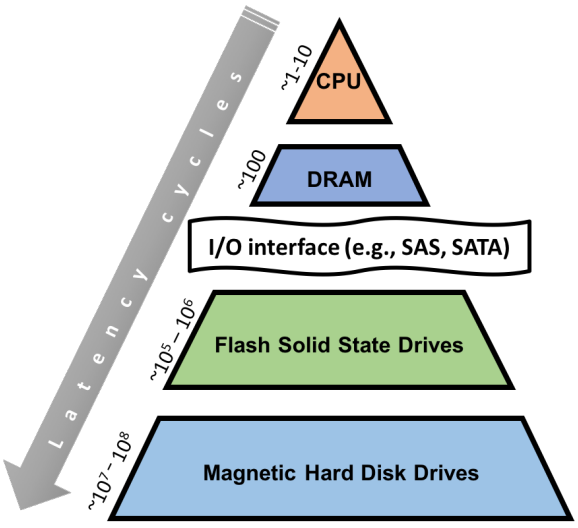


Figure 1.1: Data storage hierarchy in a database system.

Another alternative solution that we propose is to preload necessary data (i.e., hot data) from disks to DRAM (see Figure 1.3) before the user query actually requests the data. In this part of the thesis, we focus on how to load hot data efficiently at system restart, which could save the data movement cost at query time. In Chapter 4, we present a new framework for SQL Server that allows continual capturing of the state of the buffer pool, and restoring the server state quickly with a snapshot of the buffer pool at restart.

Collectively, the key goal of this dissertation is to explore methods to reduce data movement across the I/O subsystem by exploiting the computational resources that are embedded in SSDs, or by deploying methods that stage data into the DRAM right after system restart as the system restart typically wipes out data in the DRAM. The key component of this thesis consists of three parts that are described below.

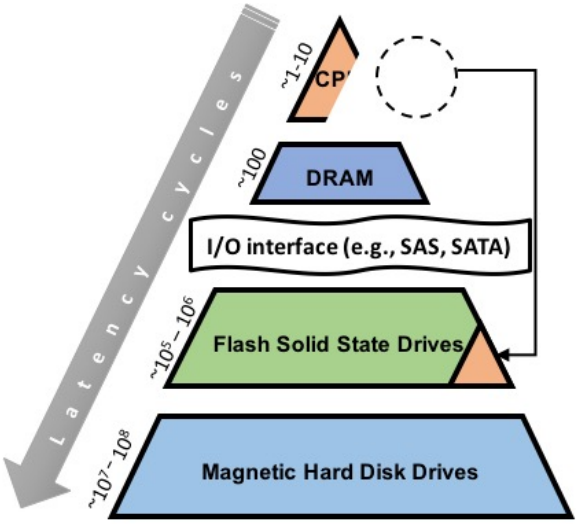


Figure 1.2: First solution: saving data movement cost from a flash SSD to DRAM by pusing down computations into a flash SSD.

1.1 Potentials and Challenges for In-storage Computing in a Relational Database Management System

Modern SSDs pack CPU processing and DRAM storage components inside the SSD to carry out routine functions (such as managing the FTL logic) for the SSD. Thus, there is a small programmable computer inside a SSD device, presenting an interesting opportunity to move computation closer to the storage. The focus of Chapter 2 is on exploring how analytical relational database workloads can exploit this in-storage computing device (a.k.a, "Smart SSDs"). This work [27, 44] shows that there some exciting opportunities for improved performance when highly-selective predicates can be pushed down into the Smart SSD.

1.2 Extending In-storage Computing for a Non-relational Data Processing System and Its Future Roadmap for Data Processing

Based on the potential performance improvements and energy gains that are achieved using the Smart SSD for relational query operations (i.e., scan, aggregate, and join), this second part of the thesis explores the opportunities and limitations for in-storage computing in a non-relational

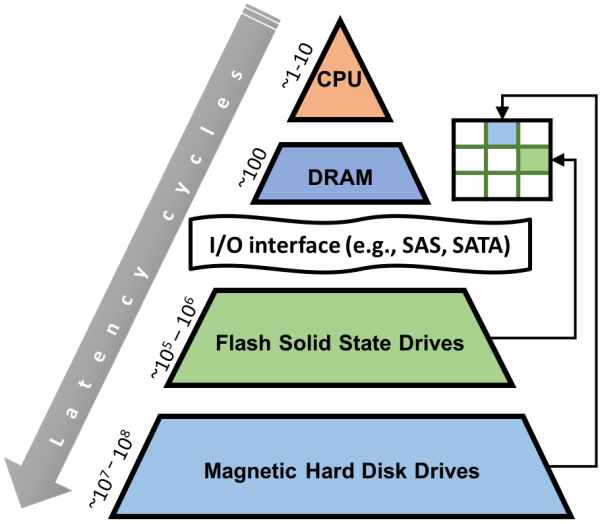


Figure 1.3: Second solution: saving data movement cost from permanent storage devices to DRAM by loading hot pages on restart.

database system (i.e., Hadoop) with unstructured data. In this chapter we also present a simple analytical model to evaluate the experimental results, and propose a roadmap to better guide hardware/software design choices for future Smart SSDs when used in data processing applications. This work is described in Chapter 3.

1.3 Aggressive Buffer Pool Warm-up after Restart in SQL Server

In many settings, a database server has to be restarted either in response to a failure event, or in response to an operational decision such as moving a database service from one machine to another. However, such restarts pose a potential performance problem as the new database server starts off with a cold buffer pool. As a result, the database application experiences a dramatic reduction in performance right after the restart, since just before the restart the database buffer pool was filled with hot pages and after the restart the database buffer pool is empty. In Chapter 4, we present a new framework for SQL Server that allows continual capturing of the state of the buffer pool, and restoring the server state quickly with a snapshot of the buffer pool at restart. Our results using

SQL Server show that our framework reduces the ramp-up time by up to 2-3X compared to the previous approaches in SQL Server and MySQL [43].

1.4 Outline

The remainder of this dissertation is organized as follows: Chapter 2 presents the potential performance improvements and energy gains from using Smart SSDs in a single node database management system (DBMS) with selected database operations such as scan, aggregation, and join. Chapter 3 continues the work to use a cluster of Smart SSDs with Hadoop MapReduce framework on unstructured data. We also propose a set of hardware/software roadmap to realize the full vision of Smart SSDs for data processing. Chapter 4 describes a set of mechanisms to reduce the ramp-up time by warming up the buffer pool aggressively on restart. Chapter 5 contains our conclusions.

Chapter 2

Potentials and Challenges for In-storage Computing in a Relational Database Management System

Data storage devices are getting "smarter." Smart Flash storage devices (a.k.a. "Smart SSDs") are on the horizon and will package CPU processing and DRAM storage inside a Smart SSD, and make that available to run user programs inside a 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 a prototype of Microsoft SQL Server 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.

2.1 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 features into a single chip (the system-on-a-chip trend). We are now on the cusp of this hardware trend sweeping over into the server world. The focus of this project is 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 focus 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.

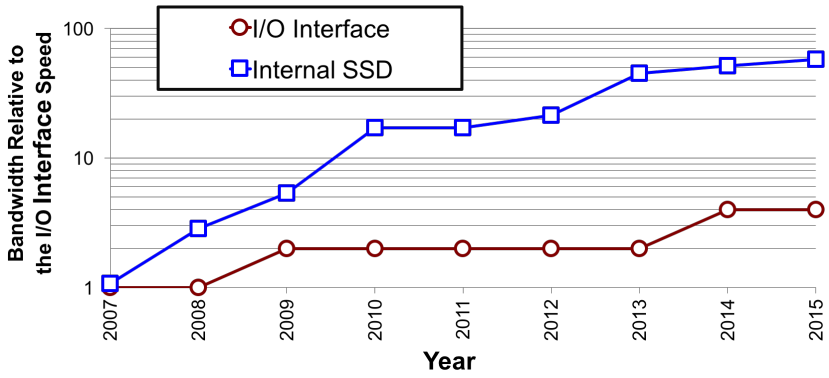


Figure 2.1: Bandwidth trends for the host I/O interface (i.e., SAS/SATA standards), and aggregate internal bandwidth available in high-end enterprise Samsung SSDs. Numbers here are relative to the I/O interface speed in 2007 (375 MB/s). Data beyond 2012 are internal projections by Samsung.

First, SSDs generally have a far larger aggregate internal bandwidth than the bandwidth supported by common host I/O interfaces (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 2.1) in the near future. Thus, pushing operations, especially highly selective ones that return few result rows, could 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 [31], and Intel Xeon processors in Exadata [19]). 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 traditional processors that are used in servers. 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 run simple selection and aggregation queries end-to-end.

Our results show that for this class of queries, we observed up to 2.7X improvement in end-to-end performance compared to using the same SSDs but without the "Smart" functionality, and up to a 3.0X reduction in energy consumption. These early results, admittedly on queries using a limited subset of SQL, 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 the 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.

Second, the firmware development process we followed to run user code in the Smart SSDs is rudimentary. This can be a potential challenge for general application developers. Before Smart SSDs can be broadly adopted, the existing development and debugging tools and runtime system (Section 2.3) 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 also extend out to query optimization, DBMS buffer pool caching policies, transaction processing, and may require re-examining how aspects such as database compression are used. In other words, the DBMS internals have to be modified to make use of Smart SSDs in a production setting.

The remainder of this chapter is organized as follows: The architecture of a modern SSD is presented in Section 2.2. In Section 2.3 we describe how Smart SSDs work. Experimental results are presented in Section 2.4. Related work is discussed in Section 2.5. Finally, Section 2.6 contains our concluding remarks and points to some directions for future work.

2.2 Background: SSD Architecture

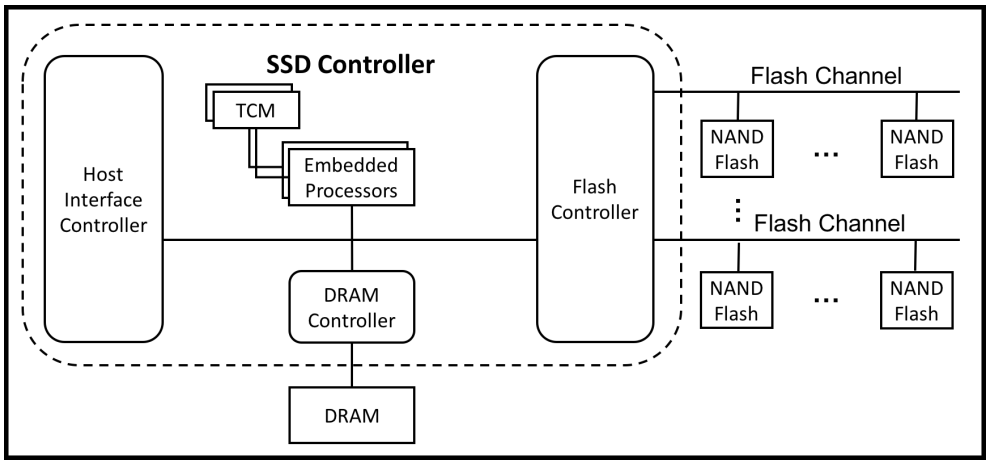


Figure 2.2: Internal architecture of a modern SSD.

Figure 2.2 illustrates the general internal architecture of a modern SSD. 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 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. There are two types of volatile memory, namely Tightly-Coupled Memory (TCM) and Dynamic Random-Access Memory (DRAM). The processing CPU is often an ARM processor with multiple cores. Each ARM core has its own dedicated TCM while the DRAM is shared by multiple cores. Access to the TCM has a far lower access latency (by a factor of 10) than the DRAM, while the size of the TCM is much smaller than the size of DRAM (by a factor of 10,000).

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. All the flash channels share access to the DRAM. Hence, data transfers from the flash channels to the DRAM (via DMA) are serialized.

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.

2.3 Smart SSDs for Query Processing

The Smart SSD runtime framework (shown in Figure 2.3) implements the core of the software ecosystem that is needed to run user-defined programs in the Smart SSDs.

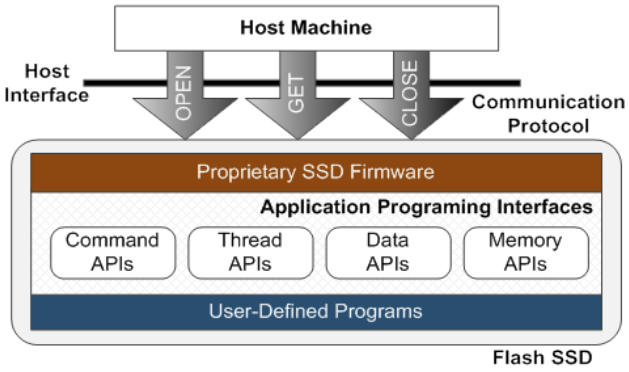


Figure 2.3: Smart SSD runtime framework.

2.3.1 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.

- OPEN, CLOSE: A session starts with an OPEN command and terminates with a CLOSE command. Once the session starts, runtime resources including threads and memory (see Thread and Memory APIs in Section 2.3.2) that are required to run a user-defined program are granted, and a unique session id is then returned to the host. Note that when one of the other Smart SSD commands (i.e., GET and CLOSE) is invoked by the host, the session id must be provided to find the corresponding session before the command is executed in the Smart SSDs. The CLOSE command closes the session associated with the session id;

it terminates any running program and releases all resources that are used by the program. Once the session is closed, the corresponding session id is invalid, and can be recycled.

- GET: The host can monitor the status of the program and retrieve results that the program generates via a GET command. This command is mainly designed for the traditional block devices (based on SATA/SAS interfaces), in which case the storage device is a passive entity and responds only when the host initiates a request. For PCIe, a more efficient command (such as PULL) could be introduced to directly leverage device-initiated capabilities (e.g., interrupts). A single GET command retrieves both the running status of the program and the results if the output is ready. With different session ids, multiple user-defined programs can be executed in parallel. Note that the programs can be blocked if no resource is available in the Smart SSD. Therefore, the polling interval should be adaptive so that it does not introduce a large polling overhead or hinder the progress of the Smart SSD operations. In our experiments, the polling interval was set to 10 msec.

2.3.2 Application Programming Interface (API)

Once a command has been successfully delivered to the device through the Smart SSD communication protocol (Section 2.3.1), the Smart SSD runtime system drives the user-defined program in an event-driven fashion. The user program can use the Smart SSD APIs for command management, thread management, memory management, and data management. The design philosophy of the APIs is to give more flexibility to the program, so that it is easier for the end-user programs to use these APIs. These APIs are briefly described below.

- Command APIs: Whenever a Smart SSD command (i.e., OPEN, GET, and CLOSE) is passed to the device, the Smart SSD runtime system invokes the corresponding callback function(s) registered by the user-defined program. For instance, the OPEN and CLOSE commands trigger user-defined open and close functions respectively. In contrast, the GET command calls functions to fill the running status of the program and to transfer results to the host if available.

- Thread APIs: Once a session is opened, the Smart SSD runtime system creates a set of worker threads and a master thread per core dedicated to the session. All threads managed by the runtime system are non-preemptive. A worker thread is scheduled when a Smart SSD command arrives (see Command APIs above), or when a data page (8KB) is loaded from flash to DRAM (see Data APIs below). Once scheduled, a user-registered callback function for that event is invoked on the thread (e.g., an open function in the event of the OPEN command). Since callback functions are designed to be "quick" functions, long-running operations that are required for each page (such as filtering) are handled by a special function that is executed in the master thread. We note that the current version of the runtime system does not support a "yield" command that gives up the processor to other threads. To simulate this behavior when the master thread is scheduled, the operation processes only a few pages, before the master thread is rescheduled to deal with the next task (which could be to process the next set of pages for the first task).

- Memory APIs: Smart SSD devices typically have two types of memory modules — a small fast TCM (e.g., ARM's Tightly-Coupled Memory), and a large slow DRAM. In a typical scenario, the DRAM is mainly used to store data pages while the TCM is used for frequently accessed metadata such as the database table schema. Once a session is open, a pre-defined amount of memory is assigned to the session, and this memory is returned back to the Smart SSD runtime system when the session is closed (i.e., dynamic memory allocation using malloc and free is not allowed.)

- Data APIs: Multiple data pages can be loaded from flash to DRAM in parallel. Here, the degree of parallelism depends on the number of flash channels employed in the Smart SSD. Once loaded, the pages are pinned to ensure that they are not evicted from the DRAM. After processing a page, it must be unpinned to release the memory required to hold the page back to the device. Otherwise, Smart SSD operations might be blocked until enough memory is available for the subsequent operations.

2.4 Evaluation

In this section, we present selected results from an empirical evaluation of Smart SSD with Microsoft SQL Server.

2.4.1 Experimental Setup

2.4.1.1 Workloads

For scan and aggregation queries, we created three synthetic tables, called Synthetic4 (having 4 integer columns), Synthetic16 (having 16 integer columns), and Synthetic64 (having 64 integer columns), each of which has 400M tuples. The sizes of these tables are 10GB, 30GB and 110GB for the Synthetic4, Synthetic16, and Synthetic64 table respectively. In addition, for two way join queries, we created two synthetic tables called Synthetic_R and Synthetic_S. Both tables have 64 integer columns. The Synthetic64_R table has 1M tuples ($\sim$ 300MB) and the Synthetic64_S table has 400M tuples ($\sim$ 120GB). The first column of the Synthetic64_R (R.Col_1) is the primary key, and the second column of the Synthetic64_S (S.Col_2) is the foreign key pointing to R.Col_1.

For analytical workloads, we used the LINEITEM and PART table from the TPC-H benchmark [16]. Because of Smart SSD's hardware and software limitations, we modified the original LINEITEM, PART table specifications as follows:

1. All variable-length string columns are converted to fixed-length string columns,
2. All decimal type columns are converted to big integers by multiplying 100,
3. All date values are converted to the number of days since the last epoch.

The LINEITEM and PART table are populated at a scale factor of 100. Thus, the LINEITEM table has 600M tuples ($\sim$ 90GB), and the PART table has 20M tuples ($\sim$ 3GB).

The tuples in the LINEITEM, PART, and synthetic tables were inserted into a relational heap table in the SQL Server without a clustered index. The tuples, by default, were stored in slotted pages using the traditional N-ary Storage Model (NSM). For the Smart SSDs, we also implemented the PAX layout [21] in which all the values of a column are grouped together within a page.

2.4.1.2 Hardware/Software Setup

All experiments were performed on a system running 64bit Windows 7 with 32GB 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 32KB L1 cache, and two 6MB 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) [6] for the three storage devices that we used in our experiments. These three devices are:

1. A 146GB 10K RPM SAS HDD,
2. A 400GB SAS SSD, and
3. A Smart SSD prototyped on the same SSD as above.

We implemented code for simple selection, aggregation, and selection with join queries, and uploaded that code into the Smart SSD. We also modified some components in SQL Server 2012 [7] to recognize and communicate with the Smart SSD. For each query that is used in this empirical evaluation, we have a special path in SQL Server that we created to communicate with the SSD using the API described in Section 2.3. Note that the results presented here are from a prototype version of SQL Server that only works on a selected class of queries.

All the results presented here are for cold experiments; i.e., there is no data cached in the buffer pool prior to running each query.

Only one of three devices is connected to the HBA at a time for each experiment. Finally, the power drawn by the system was measured using a Yokogawa WT210 unit (as suggested in [9]). We used this server hardware since it was compatible with the LSI HBA card that was needed to run the extended host interface protocol described in Section 2.3.1.

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, selection with aggregation, and selection with aggregation/join queries in the Smart SSD by using the Smart SSD APIs (Section 2.3.2). We also modified some components in SQL Server 2012 [7] to recognize and communicate with the Smart SSD through the Smart SSD communication protocol (Section 2.3.1). For each test, we measured the elapsed wall-clock time, and calculated the disk (I/O subsystem) energy consumption as well as the entire system 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. Thus, all the results presented here are for cold experiments; i.e., there is no data cached in the buffer pool prior to running each query.

2.4.2 Experimental 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 2.1. The bandwidth of the HDD and the SSD was obtained using Iometer [5]. For the Smart SSD internal bandwidth, we implemented a simple program (by using the Smart SSD APIs introduced in Section 2.3.2) to measure the wall clock time to sequentially fetch a 100GB dummy data file from flash to the on-board DRAM. Note that for this experiment, there was no data transfer between the SSD and the host. The only traffic between the host and the Smart SSD was the communication associated with issuing the Smart SSD commands (i.e., OPEN, GET, and CLOSE) to control the program.

Table 2.1: Maximum sequential read bandwidth with 32-page (256KB) I/Os.

	SAS HDD	SAS SSD	(internal) Smart SSD
Sequential Read (MB/sec)	80	550	1,560

As can be seen in Table 2.1, the internal 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 this Smart SSD could potentially deliver. As described

in Figure 2.1, over time it is likely that the gap between the SSD and the Smart SSD will grow to a much larger number than 2.8X.

We also note that the improvement here (of 2.8X) is far smaller than the gap shown in Figure 2.1 (about 10X). The reason for this gap is that the access to the DRAM is shared by all the flash channels, and currently in this SSD device only one channel can be active at a time (recall the discussion in Section 2.2), which becomes the bottleneck. One could potentially address this bottleneck by increasing the bandwidth to the DRAM or adding more DRAM busses. As we discuss below, this and other issues must be addressed to realize the full potential of the Smart SSD vision.

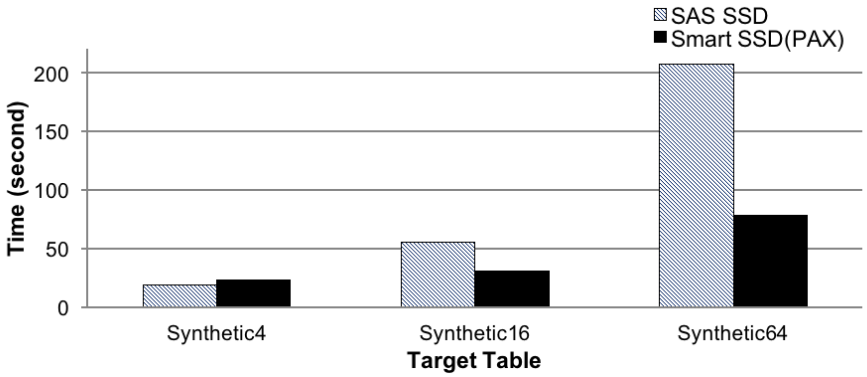


Figure 2.4: End-to-end elapsed time for a selection query at a selectivity of 0.1% with the three synthetic tables Synthetic4, Synthetic16, and Synthetic64.

2.4.2.1 Selection Query

For this experiment, we used three synthetic tables and the following SQL query:

SELECT SecondColumn

FROM SyntheticTable

WHERE FirstColumn <[VALUE]

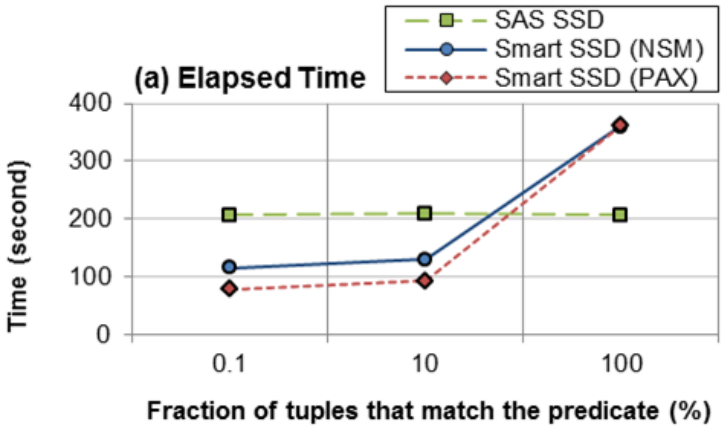
Effect of Tuple Size: Figure 2.4 shows the end-to-end elapsed time to execute the selection query at a selectivity of 0.1% with the three synthetic tables (Synthetic4, Synthetic16, and Synthetic64). As can be seen in this figure, the Smart SSD, with a PAX layout, executes the selection query on

the Synthetic64 table 2.6X faster than the regular SSD, whereas the selection on the Synthetic4 table is slower than the regular SSD. The performance improvement of the Smart SSD comes from the faster internal I/O, whereas the low computation power of the ARM core in the Smart SSD saturates its performance. In this experiment, in all the three cases, the Smart SSD improves the I/O component of fetching data from the flash chips. But, compared to the regular SSD case, the Smart SSD has to compute on the data in the pages that are fetched from the flash chips before sending it to the host. With the Synthetic64 data set, this computation cost (measured as cycles/page) is low as there are only 29 tuples on each page. However, with the Synthetic4 table, there are 323 tuples on each data page, and the Smart SSD-based execution strategy now has to spend far more processing cycles per page, which saturates the CPU. Now, the query (on the Synthetic4 table) in the Smart SSD is bottlenecked on the CPU resource. In the case of this SSD device, for the Synthetic4 data set, the throughput of the computation that can be pushed "through the CPU" in the Smart SSD is lower than the host IO interface. Consequently, the performance of this query (with 0.1% selectivity) is faster with the regular SSD.

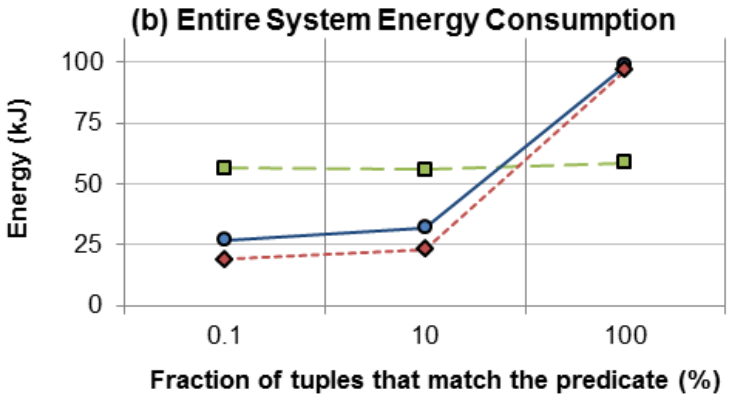
Effect of Varying the Selectivity Factor: Figures 2.5 (a), 2.5 (b), and 2.5 (c) present the end-to-end elapsed time and the energy consumed when executing the selection query at various selectivity factors on the Synthetic64 table, using the regular SSD, and the Smart SSD with the default NSM layout and the PAX layout. The energy consumption is shown for the entire system in Figure 2.5 (b), and for just the I/O subsystem in Figure 2.5 (c).

Table 2.2: Results for the SAS HDD: End-to-end query execution time, entire system energy consumption, and I/O subsystem energy consumption for a selection query on the Synthetic64 table at various selectivity factors.

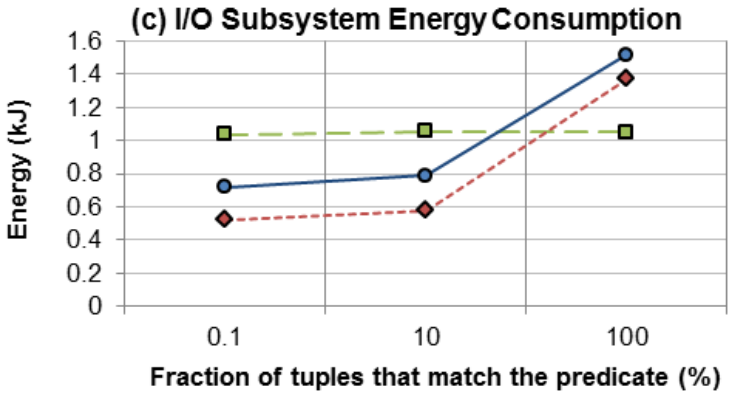
	0.1%	10%	100%
Elapsed time (seconds)	1,494	1,486	1,485
Entire System Energy (kJ)	357	358	358
I/O Subsystem Energy (kJ)	13	13	13



(a)



(b)



(c)

Figure 2.5: End-to-end (a) query execution time, (b) entire system energy consumption, and (c) I/O subsystem energy consumption for a selection query on the Synthetic64 table at various selectivity factors.

To improve the presentation of these figures, we do not show the measurements for the HDD case, as it was significantly higher than the SSD cases. Rather, we show the measurements for the HDD case in Table 2.2. As can be observed from Figure 2.5 (a) and Table 2.2, the Smart SSD provides significant improvements in performance for the highly selective queries (i.e. when few tuples match the selection predicate). The improvements are 19X and 2.6X over the HDD and the SSD, respectively when 0.1% of the tuples satisfy the selection predicate.

One interesting observation from Figure 2.5 (a) is that for the Smart SSD case, using the PAX layout provides better performance than the NSM layout, by up to 32%. As an example, for the 0.1% selection query, the elapsed times when using NSM and PAX are about 115 seconds and 78 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, called the LDM instruction [3] . Since all the values of a column in a page 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 (slow) DRAM accesses. Given the high DRAM latency in the SSD, the columnar PAX layout is more efficient than a row-based layout.

In addition, from Figure 2.5 (b) and Table 2.2, we observe that the Smart SSD provides a big energy efficiency benefits — up to 18.8X and 3.0X over the HDD and the SSD respectively, with 0.1% selectivity. Furthermore, from Figure 2.5 (c) and Table 2.2, we observe that the Smart SSD achieves a substantial I/O subsystem energy efficiency improvement. For example it reduces the energy consumption by 24.9X and 2.0X over the HDD and the SSD cases respectively, at 0.1% selectivity. The interesting observation for the I/O subsystem energy consumption is that the Smart SSD energy efficiency benefit over the SSD is not proportional to the elapsed time. In other words, the elapsed times at 0.1% selectivity when using the Smart SSD with a PAX layout and the regular SSD are about 78 seconds and 207 seconds, which shows 2.6X performance improvement. However, the I/O subsystem energy efficiency improvement is only 2.0X. That is because the Smart SSD consumes additional computation power compared to the regular SSD.

With the Synthetic4 and the Synthetic16 tables, the Smart SSD is usually slower than the regular SSD for the select query at 0.1% selectivity, and in the worst case about 2.6X slower with the NSM format. As above, the PAX format works better with the Smart SSD, and in the worst case the Smart SSD is 22% slower than the regular SSD. The reasons for this behavior are similar to the case of the Synthetic4 table shown in Figure 2.4 (See Section 2.4.2.1).

2.4.2.2 Selection with Aggregation Query

For this experiment, we used the following SQL aggregate query:

SELECT AVG (SecondColumn)

FROM SyntheticTable

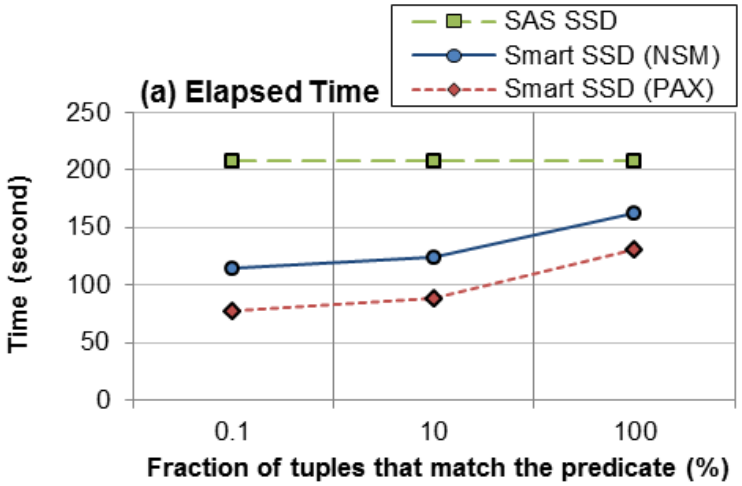
WHERE FirstColumn <[VALUE]

The results for this experiment for the Synthetic64 dataset are shown in Figure 2.6. The HDD results for this experiment are shown in Table 2.3. From Figure 2.6 and Table 2.3, we note that compared to the Smart SSD case with PAX, the HDD case takes 19.2X longer to execute the query, consumes 18.7X more energy at the whole server/system level, and about 23.3X more energy in just the I/O subsystem.

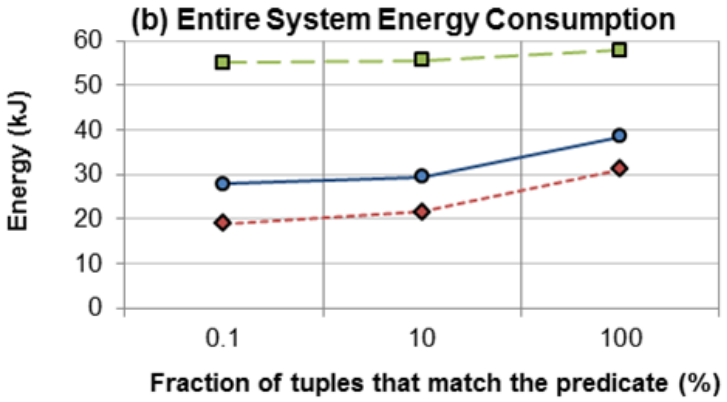
Table 2.3: Results for the SAS HDD: End-to-end query execution time, entire system energy consumption, and I/O subsystem energy consumption for a selection with aggregation query on the Synthetic64 table at various selectivity factors.

	0.1%	10%	100%
Elapsed time (seconds)	1,485	1,486	1,488
Entire System Energy (kJ)	354	353	355
I/O Subsystem Energy (kJ)	13	13	13

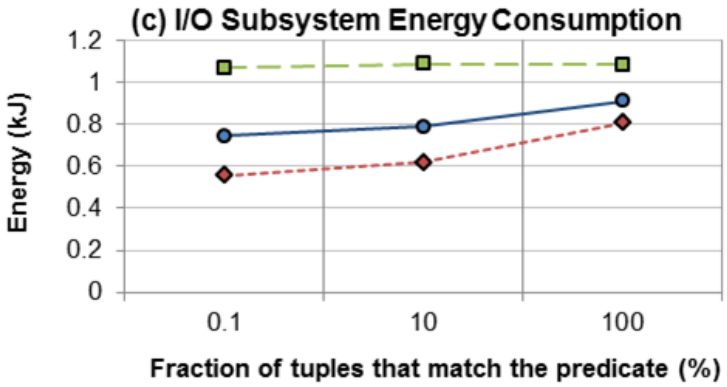
Similar to the previous results, the Smart SSD shows significant performance and energy savings over the HDD and the SSD cases. As seen in Figure 2.6 (a), the Smart SSD improves performance for the highly selective queries by up to 2.7X over the (regular) SSD case when 0.1% of



(a)



(b)



(c)

Figure 2.6: End-to-end (a) query execution time, (b) entire system energy consumption, and (c) I/O subsystem energy consumption for a selection with aggregate query on the Synthetic64 table at various selectivity factors.

the tuples satisfy the selection predicate. In addition, as shown in Figure 2.6 (b), using the Smart SSD (with PAX) is 2.9X more energy efficient than the regular SSD case when the selectivity is 0.1%. Furthermore, as can be observed from Figure 2.6 (c), the Smart SSD with PAX is 1.9X more efficient in the I/O subsystem over the (regular) SSD case, at 0.1% selectivity.

The one big difference between the simple selection query results shown in Figure 2.5 and Table 2.2, and the aggregate query results shown in Figure 2.6 and Table 2.3, is that with the aggregate query, the Smart SSD has better performance than the HDD and the SSD cases even at 100% selectivity. The reason for this behavior is that the output of the aggregation query is far smaller than the output of the selection query. Thus, the selection query has a much higher I/O cost associated with transferring data from the Smart SSD to the host, which diminishes the benefits of the Smart SSD.

With the Synthetic4 and Synthetic16 tables, similar to the selection query results, the Smart SSD is usually slower than the regular SSD for the aggregate query at 0.1% selectivity, and in the worst case about 2.5X slower with the NSM format. As above, performance is higher with the PAX format in the Smart SSD, and in the worst case the Smart SSD is 20% slower than the regular SSD. The reasons for this are also similar to the case of the Synthetic4 table shown in Figure 2.4 (See Section 2.4.2.1).

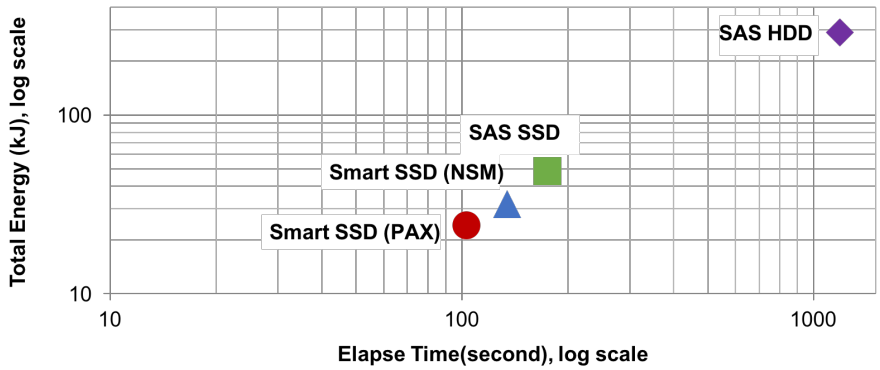


Figure 2.7: Elapsed time and entire system energy consumption for the TPC-H query 6 on the LINEITEM table (100SF).

2.4.2.3 TPC-H Query 6

For this experiment, we used the LINEITEM table and Query 6 from the TPC-H benchmark [16], using the default SHIPDATE, DISCOUNT, and QUANTITY values for the predicates in the query. This query is:

SELECT SUM (EXTENDEDPRICE * DISCOUNT)
FROM LINEITEM
WHERE SHIPDATE >= 1994-01-01 AND
SHIPDATE <1995-01-01 AND
DISCOUNT >0.05 AND
DISCOUNT <0.07 AND
QUANTITY <24

Figure 2.7 shows the results with the HDD, the SSD, and the Smart SSD (with the NSM and the PAX layouts). The Smart SSD with the PAX layout improves overall query response time by 11.5X and 1.7X over the HDD and the SSD cases respectively. Also, it provides 12.0X and 2.0X energy efficiency gains for the entire system over the HDD and the SSD respectively. The LINEITEM table contains 51 tuples in a data page, which is more than the Synthetic64 case (29 tuples/page), but less than the Synthetic16 table case (109 tuples/page). With the Synthetic16 table, the Smart SSD with the PAX layout provides about 12.5X and 1.8X performance improvements over the HDD and the SSD respectively, for the aggregate query at 0.1% selectivity factor. The selectivity factor of the TPC-H benchmark Query 6 is 0.6%. As explained in Section 2.4.2.1, the number of tuples in a data page has a big impact on the performance improvement that is achieved using the Smart SSD. So, the LINEITEM table should have provided better performance improvement than the Synthetic16 table. However, the higher selectivity of the TPC-H benchmark Query 6 (0.6% vs. 0.1%), and its more complex predicates (five predicates vs. one predicate) saturates the CPU and the memory resources in the Smart SSD. As a result, the performance improvement of TPC-H Query6 with LINEITEM table is similar to that of the aggregate query described in Section 2.4.2.2 with a 0.1% selectivity factor for the Synthetic16 table.

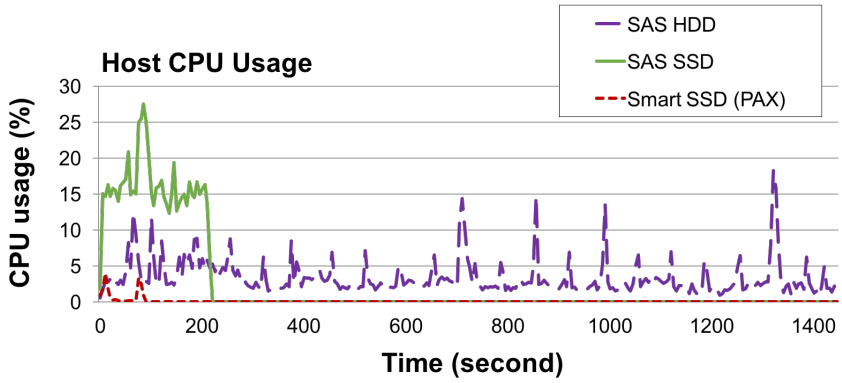


Figure 2.8: Host CPU usage for the SAS HDD, the SAS SSD, and the Smart SSD for a selection query with average query on Synthetic64 table at 0.1% selectivity factor.

2.4.2.4 Two-way Join Queries

In this section, we evaluate the impact of using the Smart SSD to run simple join queries.

Selection with Join Query For this experiment, we use the Synthetic64_R and the Synthetic64_S tables, and the following SQL query:

SELECT S.Col_1, R.Col_2

FROM Synthetic64_R R, Synthetic64_S S

WHERE R.Col_1 = S.Col_2 AND S.Col_3 <[VALUE]

Figure 2.9 shows the query plan for this query. Since the size of the Synthetic64_R table is far smaller than the Synthetic64_S table, (i.e., $|S| = 400|R|$), and the hash table for the Synthetic64_R table fits in memory, we used a simple hash join algorithm that builds a hash table on the Synthetic64 R table. As shown in Figure 2.9, the core computation of this query is carried out inside the Smart SSD and the host only collects the output from the Smart SSD. In the case of the regular SSD, we used the same query plan as the Smart SSD, but the plan was run entirely in the host.

Similar to the previous result, the Smart SSD shows significant performance over the regular SSD case. As seen in Figure 2.10, the Smart SSD (with the PAX layout) improves performance for the highly selective queries by up to 2.2X over the (regular) SSD case when 1% of the tuples in the Synthetic64_S table satisfy the selection predicate. Similar to the case of single table scan

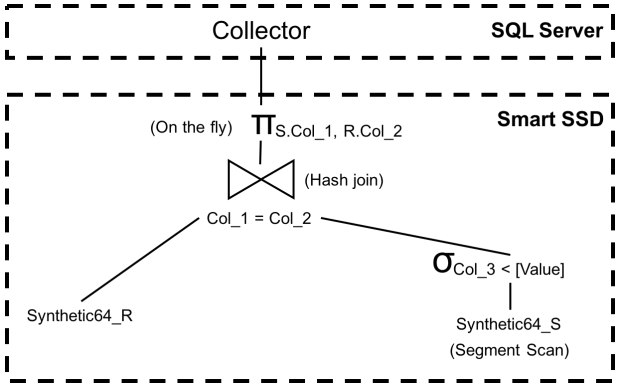


Figure 2.9: Query plan for the selection with join query in the Smart SSD.

queries, when the selectivity of the query is low (i.e., 100%), the performance of the Smart SSD is saturated because the query cost is dominated by the cost of the high volume of data that has to be transferred between the host and the Smart SSD.

TPC-H Query 14 For this experiment, we used the LINEITEM, the PART table and Query 14 from the TPC-H benchmark [16]. This query is:

SELECT 100 * sum(case when p_type like 'PROMO%'
then l_extendedprice * (1 - l_discount) else 0 end)
/ sum(l_extendedprice * (1 - l_discount))
as promo_revenue
FROM LINEITEM, PART
WHERE l_partkey = p_partkey AND l_shipdate >= date '1995-09-01'
AND l_shipdate <date '1995-09-01' + interval '1' month

Figure 2.11 shows the query plan for this query when run in the Smart SSD. Overall, this query plan is the same as the plan that was used in Section 4.2.2.1 (except that the selection was replaced by an aggregation), since the size of the PART table is also much smaller than the LINEITEM table, (i.e., $|\text{LINEITEM}| \simeq 30|\text{PART}|$), and the hash table for the PART table fits in memory.

Figure 2.12 shows the results with the (regular) SSD, and the Smart SSD (with the NSM and the PAX layouts). The Smart SSD with the PAX layouts improve the overall query response time

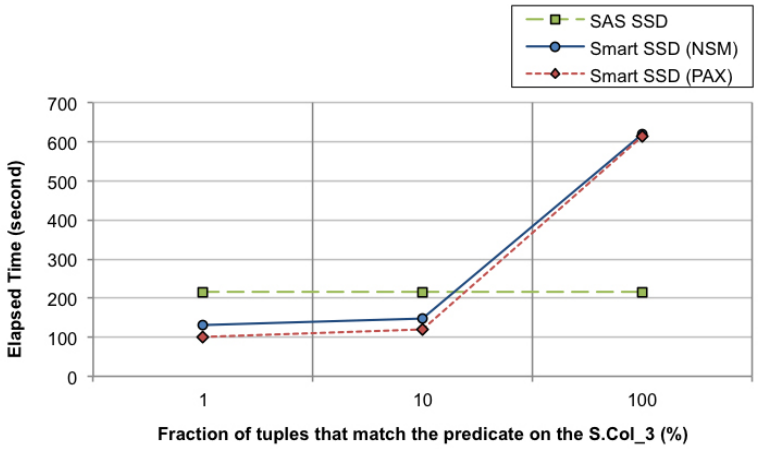


Figure 2.10: Elapsed time for the join query on the Synthetic64_R and the Synthetic64_S tables at various selectivity factors.

by 1.3X over the (regular) SSD case. As discussed in [27], the Smart SSD achieves greater benefits when the query requires fewer computations (number of CPU cycles) per data page. However, this query requires a large number of CPU cycles to be executed per page compared to the query (a simple selection with aggregation query) used in Section 2.4.2.3, which saturates the performance of the Smart SSD. That is why the performance improvement of this query is lower than the selection with aggregation query in Section 2.4.2.3 (1.3X vs. 1.8X).

2.4.3 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 4.2.2 that with the aggregate query, we observed 18.7X and 2.9X 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 (235W), then these gains become 25.1X and 11.6X over the HDD and the SSD, respectively. Figure 2.8 shows the host CPU usage for the HDD, the SSD, and the Smart SSD. The SAS SSD uses about 20% of the host CPU during the query execution time whereas the Smart SSD rarely uses the host CPU. In our experimental setup, the power consumption of the host CPU is about 65W whereas the power consumption of the general ARM core is less than 5W. This power consumption difference results

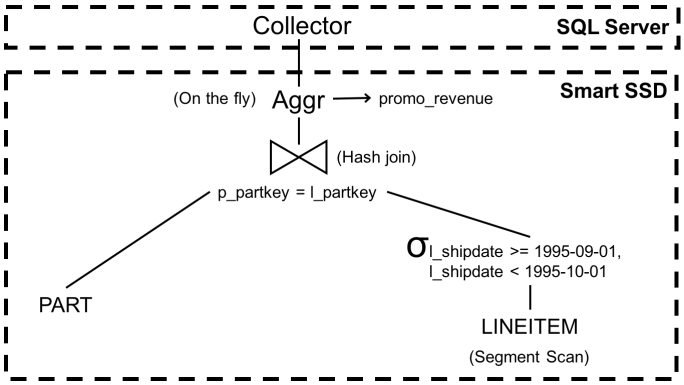


Figure 2.11: Query plan for the TPC-H query 14 in the Smart SSD.

in the Smart SSD's huge energy consumption gain (11.6X) over the SAS SSD at the entire system level.

A crucial observation that we made is that the processing capabilities inside the Smart SSD quickly became a performance bottleneck, in particular when the selection predicate matches many input tuples or when there is a large amount of processing to be done per page of data (e.g., the Synthetic4 table). For example, as seen in Figure 2.5 (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. Also, as discussed in Section 2.4.2.1, the Smart SSD achieves greater benefits when the query requires fewer computations per data page.

The development environment that is required to run code inside the Smart SSD needs further development. A large part of the tool that we used in this study was developed hand-in-hand with Samsung for this project. To maximize the performance that we could achieve with the Smart SSD, we had to carefully plan 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 TCM. Similarly, we used a hardware-debugging tool called Trace32, a JTAG in-circuit (ICD) debugger [17], which is far more primitive than the regular debugging tools (e.g., Visual Studio) available to database systems developers.

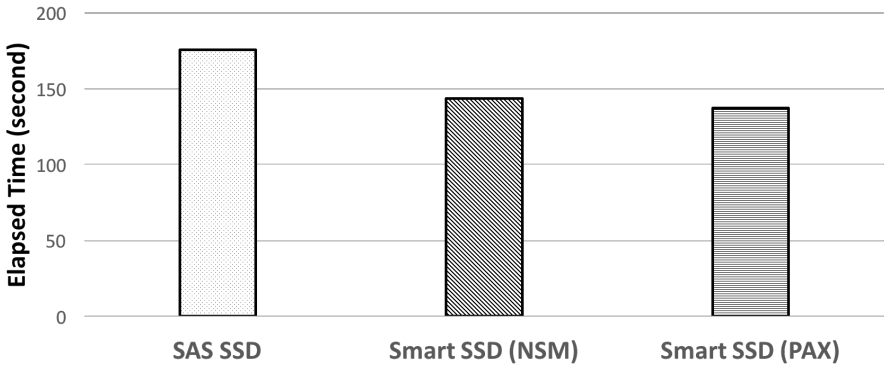


Figure 2.12: Elapsed time for the TPC-H query 14 on the LINEITEM and PART table (100SF).

On the DBMS side, the implication of using a Smart SSD for query processing has other ripple effects. One key area is around caching in the buffer pool. If there is a copy of the data in the buffer pool that is more current than the data in the SSD, pushing the query processing to the SDD may not be feasible. Similarly, queries with any updates can't be processed in the SSD without appropriate coordination with the DBMS transaction manager. If the database is immutable then some of these problems become easier to handle.

In addition, there are other implications for the internals of existing DBMSs, including query optimization. If all or part of the data is already cached in the buffer pool then pushing the processing to the Smart SSD may not be beneficial (from both the performance and the energy consumption perspectives). In addition, even when processing the query the usual way is less efficient than processing all or part of the query inside the Smart SSD, we may still want to process the query in the host machine as that brings data into the buffer pool that can be used for subsequent queries.

Finally, using a Smart SSD can change the way in which we build database servers/appliances. For example, if the issues outlined above are fixed, and Smart SSDs in the near future have both significantly more processing power and are easier to program, then one could build appliances that have far fewer compute and memory resources in the host server than what typical servers/appliances have today. Thus, pushing the bulk of the processing to Smart SSDs could produce a data processing system that has higher performance and potential a lower energy consumption profile than traditional servers/appliances.

At the extreme end of this spectrum, the host machine could simply be the coordinator that stages computation across an array of Smart SSDs, making the system look like a parallel DBMS with the master node being the host server, and the worker nodes in the parallel system being the Smart SSDs. The Smart SSDs could basically run lightweight isolated SQL engines internally that are globally coordinated by the host node. Of course, the challenges associated with using the Smart SSDs (e.g. buffer pool caching and transactions as outlined above) must be addressed before we can approach this end of the design spectrum.

2.5 Related Work

Since Jim Gray's 2006 prediction [32] that "tape is dead, disk is tape, and flash is disk", various DBMS internal components have been revisited for flash SSDs to improve the DBMS performance (e.g., for query processing [28, 48], index structures [20, 38, 49], and page layout [37]). In particular, a promising and well-established way of using the SSDs in a DBMS is to extend the main-memory buffer pool [22, 24, 29, 30, 36]. With an SSD buffer pool extension, pages that are evicted from the main-memory buffer pool are selectively cached in the SSDs to be served for subsequent accesses on the pages. The industry has released commercial storage appliances including Oracle Exadata [19], Teradata Virtual Storage System [12], and IBM XIV Storage System [4] that use similar ideas. As revealed in [30], however, the SSD buffer pool extensions are mainly beneficial for OLTP workloads, and not data warehousing workload, which is the focus of this chapter. A nice overview of techniques that use flash memory for DBMSs is described in [35].

Over a decade ago, the concept of in-storage processing, which involves combining on-disk computational power with memory to execute all or part of application functions directly in the device, was propose in the Active Disks [45, 46] and the Intelligent Disks [33] projects. The studies proposed to exploit the excess computational power of the embedded processors in disks for useful data processing (offloaded from the host) to mainly reduce the data traffic between the host and the device. For example, Riedel et al. demonstrated performance gains for data computational tasks (e.g., filtering, image processing [46], and primitive database operations such as scan, aggregation [45]) in this environment. Since then, however, the computational power of disk controllers has not been improved significantly [23], and therefore none of the approaches have been commercially successful.

Similar efforts of moving computation closer to the data have been realized with the help of special-purpose or commodity hardware to improve the performance of database processing. Mueller et al. [41, 40] proposes an FPGA-based approach, in which an FPGA is located between the disk and the host. In this approach, the data from the disk is pre-processed before it is fed to the host processors, and as a result, some of the computational work can be offloaded from the host. A

commercial product based on this idea can be found in [31]. Another approach that uses additional commodity processors in storage servers is Oracle Exadata [19]. By pushing down some database operations from database servers to storage servers, the amount of data traffic can be significantly reduced. Our work follows in this same direction, but directly uses processing that can be directly built as part of the SSD manufacturing process.

Recently, several studies have explored the feasibility of in-storage processing on flash SSDs [25, 34]. These studies propose using a dedicated hardware logic (that is placed inside a flash controller) to accelerate the scan operation. A commercial SoC designer was used to demonstrate performance and energy gains by simulating the hardware logic. In [23], an analytical model was presented to examine the energy-performance trade-offs when data analysis tasks are carried out on the SSD-resident processors in a High Performance Computing (HPC) context. Lessons from these studies can be used to guide the future development of additional processing inside the Smart SSD for database related data processing.

2.6 Conclusion

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 (up to 2.7X in some cases) and a significant reduction in energy consumption for the entire server (up to 3.0X 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, TCM and DRAM) so that the DBMS code can run more effectively inside the SSD. These enhancements are absolutely crucial to achieve the 10X or more benefit that Smart SSDs have the potential of providing (see Figure 2.1). 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 the 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, considering the

impact of various storage layout, 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 and Smart SSDs present an interesting additional new axis for thinking about how to build future high-performance database servers/appliances. Our results indicate that there is a significant potential benefit for database hardware and software vendors to come together to explore this opportunity.

Chapter 3

Extending In-storage Computing for a Non-relational Database System and Future Roadmap for Smart SSDs

In the previous chapter, we explored the potential performance improvements and energy gains that are achieved using the Smart SSD [27, 44] for relational query operations with structured data. In this chapter, we explore the opportunities and limitations of using Smart SSDs for a non-relational database system (i.e., Hadoop [1]) with unstructured data. We also present a simple analytical model to evaluate our experimental results and propose a roadmap for hardware/software design choices with Smart SSD.

3.1 Introduction

As shown in the previous chapter, using a flash SSD as a computing component (a.k.a "Smart SSD") showed a great potential for a relational database system, which largely manages structured data. We showed both the potential performance improvements and energy gains by pushing down various database operations (e.g., scan, aggregation and join) into a flash SSD device.

This part of the thesis explores the opportunities and challenges for in-storage computing with a cluster of Smart SSDs in a non-relational database system (i.e., Hadoop [1]). We have implemented an initial prototype in the Hadoop MapReduce framework [1] running on Samsung Smart SSDs. Our results show that for a `grep` application using eight Smart SSDs, we observed up to 2.8X improvement in end-to-end performance compared to using the same SSDs but without the "Smart" functionality (i.e., regular SSDs).

In this part of the thesis, we also design a cost model that explains the problems in data processing with Smart SSDs. Current SSDs have various limitations that hinder the opportunities associated with pushing computation into the SSD. Based on the cost model, we propose a list of hardware/software recommendations to realize the full vision of the Smart SSD for data processing.

On a cautionary note, as was pointed out in the previous chapter, the market place for Smart SSDs will require significant changes from both the hardware and software vendors. The focus of this part of the thesis is not to speculate on how long it will take for the market place to fully utilize the potential for Smart SSDs, but rather to provide a more detailed analysis of the benefits and limitations of the Smart SSD approach, so that both the Smart SSD hardware vendors and the database software developers can make "smarter" designs about if and how to use Smart SSDs for data processing.

The remainder of this chapter is organized as follows: The internal architectural details of a modern SSD and a sample actual Samsung Smart SSD properties are presented in Section 3.2. In Section 3.3, we describe how we have made Smart SSDs work with the Hadoop 3.0 framework [2]. Experimental results and analytics are presented in Section 3.4. Finally, Section 3.5 contains our concluding remarks and points to a list of hardware/software recommendations to realize the full vision of the Smart SSD for data processing.

3.2 Overview of the Current Smart SSD

In this section, we describe the detailed properties of an actual current Smart SSD to better understand the internal architectural aspects of a Smart SSD. Then, we develop a simple cost model for in-storage data processing with Smart SSDs based on these properties.

3.2.1 Smart SSD Properties

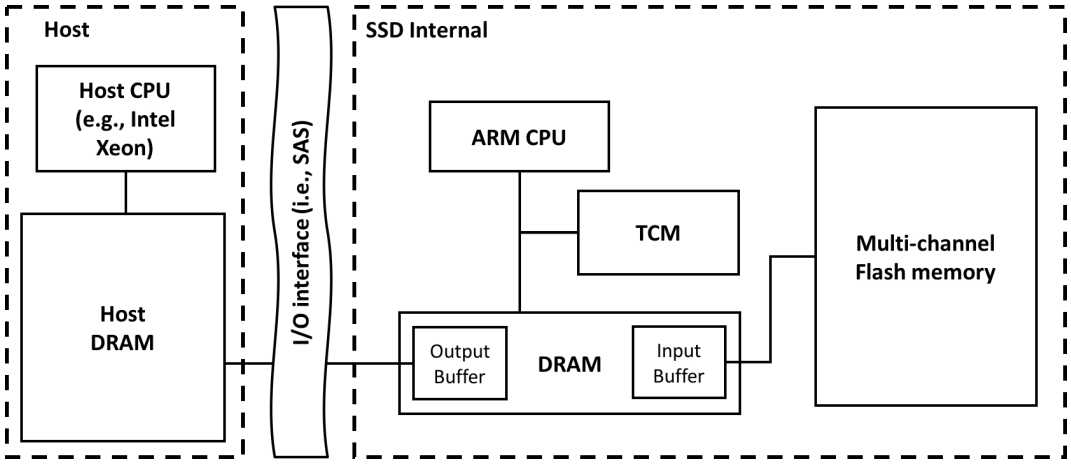


Figure 3.1: Smart SSD hardware architecture.

As shown in Figure 3.1, inside a flash SSD device there are two types of volatile memory, namely Tightly-Coupled Memory (TCM) and Dynamic Random-Access Memory (DRAM). The processing CPU is often an ARM processor with multiple cores. Each ARM core has its own dedicated TCM while the DRAM is shared by multiple cores. Access to the TCM has a far lower access latency (by a factor of 10) than the DRAM, while the size of the TCM is much smaller than the size of DRAM (by a factor of 10,000). There are also hundreds of gigabytes of multi-channel flash memory where the data is stored permanently. The main benefit of the in-storage data processing approach comes from the larger aggregate internal bandwidth than the bandwidth supported by common host I/O interfaces (see Figure 2.1). However, the downside of the in-storage data processing is the potential performance bottlenecks caused by the low-performance embedded processor that doesn't have L1/L2 caches (unlike the processor in the host), and the high latency

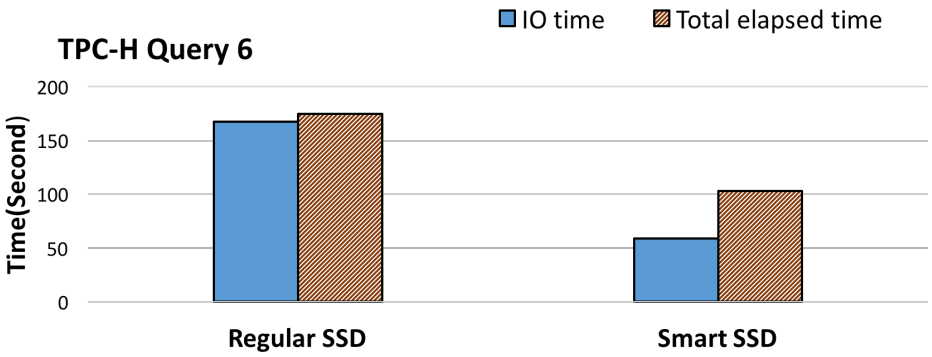
cost for accessing the DRAM memory. In other words, when there is a large amount of processing to be done or a large number of DRAM memory accesses per page of data, the performance of the Smart SSD drops significantly (compared to using the SSD as just a storage device and computing in the host).

To further consider the performance bottlenecks as well as the large internal bandwidth of the Smart SSD, we define the CPU time and I/O time for Smart SSD-based data processing as follows: **The CPU time** is the total elapsed time not only to process data in the CPU, but also to read data from the underlying memory hierarchies, and **the I/O time** is the total elapsed time to load data from the flash memory to the DRAM inside the Smart SSD.

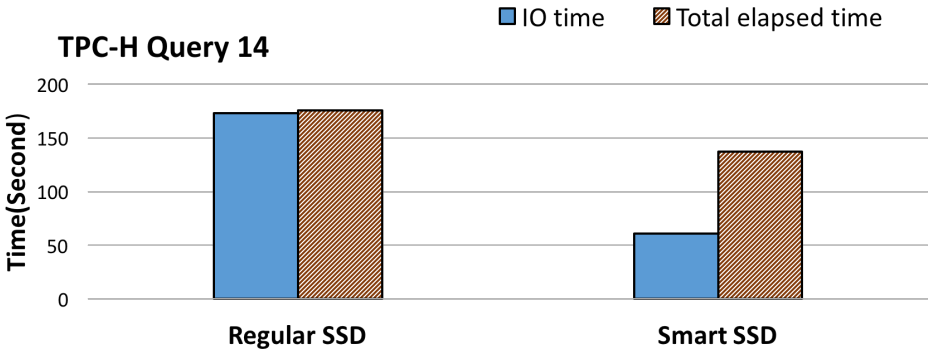
Figure 3.2 demonstrates the potential CPU time overhead for data processing inside the Smart SSD. Figure 3.2 shows the experimental results for TPC-H benchmark [16] Query 6 and Query 14 with a regular SSD (the same SSD as the Smart SSD but without the "Smart" functionality) and a Smart SSD (see Section 2.4.2.3 and Section 2.4.2.4 for more details about these experiments).

In the regular SSD, where all computations are executed in the host CPU, both queries are I/O bound. However, in the Smart SSD, where all computations are performed inside the SSD, there are multiple dominant factors, including the I/O cost. From these results, we can determine that an I/O-bound workload using a regular SSD may become a CPU-bound workload when using a Smart SSD. Another interesting observation is that in-storage query processing for the TPC-H Query 6 (scan and aggregation query) is more beneficial than it is for the TPC-H Query 14 (scan, join, and aggregation query) because the TPC-H Query 14 requires more computations and more DRAM accesses per data page (as that query has an additional join operation).

To find the main factors of the CPU cost for data processing inside the Smart SSD, we performed two types of experiments. In the first experiment, we loaded 100GB of data from the flash memory to the Smart SSD DRAM, and read a certain number of tuples in a 8KB page from DRAM (x-axis). In the second experiment, we added a for-loop with each DRAM access, and placed a simple add calculation within that for-loop. The result shows that the CPU time is minimally affected by the addition calculation. Also, the total CPU time is almost proportional to the number of DRAM accesses per page fetch. This is because the access to the DRAM is shared by all the



(a)



(b)

Figure 3.2: End-to-end (a) TPC-H Query 6 query execution time and (b) TPC-H Query 14 query execution time with a regular SSD and a Smart SSD.

flash channels and the ARM CPU, and currently in our SSD device the CPU competes with the flash channels to read data from DRAM. Therefore, we can conclude that DRAM access is the main bottleneck for the total CPU time under the *current* Smart SSD architecture for simple data processing tasks.

Based on the properties of the flash SSD device and the observations from our experiments, we suggest two basics rules for data processing inside the Smart SSD:

- **DRAM rule**: Minimize the number of (expensive) DRAM accesses.
- **Cache rule**: Utilize other memory hierarchy such as the TCM to avoid the expensive DRAM access. Note that inside the Smart SSD there is no OS or transparent hardware-based caching, so the database program has to manage the hardware directly.

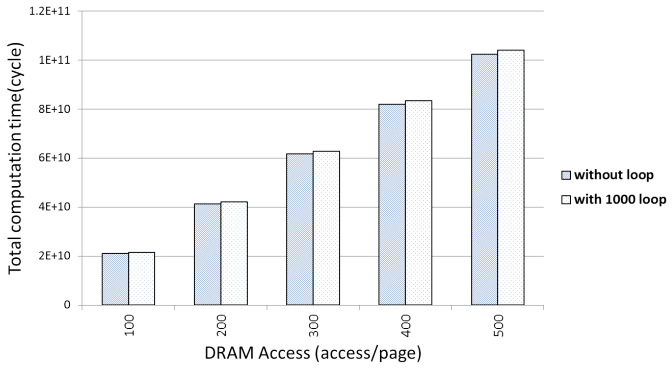


Figure 3.3: CPU time to process 100GB data with the various number of DRAM accesses per page.

3.2.2 Smart SSD Cost Model

To evaluate the performance of a specific Smart SSD data processing operation, we used a cost model that includes the I/O cost and the CPU cost for data processing inside the Smart SSD. This cost model was used to assess Hadoop applications with Smart SSDs and estimate storage organizations for Smart SSDs. Even though this cost model is based on the *current* Samsung

Smart SSD, it can provide the foundation for realizing the full potential of the Smart SSD for data processing in the future.

3.2.2.1 Smart SSD Cost Model Assumptions

Building an accurate cost model from scratch is a challenging task. Thus, we made the following assumptions to simplify this challenge:

- **Complete parallel data processing with multiple SSDs**: The cost model for distributed data processing with multiple SSDs assumes a completely parallel in-storage computing with multiple SSDs.
- **Absence of data movement between SSDs**: This cost model assumes that there is no data movement cost when using multiple SSDs.

Table 3.1: List of variables used and their definitions.

Variable	Definition
C_{FD}	Cost to load a page into the DRAM from flash chips (seconds/page)
N_P	Number of page fetches
C_D	Cost to access data in the DRAM (cycles/access)
C_T	Cost to access data in the TCM (cycles/access)
C_{DT}	Cost to copy data from the DRAM to the TCM (cycles/byte)
N_D	Number of access to the DRAM
N_T	Number of access to the TCM
D_{DT}	Size of data transferred from the DRAM to the TCM (bytes)
N_{SSD}	Number of SSDs

Now we define a cost model for data processing with Smart SSDs based on the properties and assumptions. This cost model will be used for evaluating Hadoop applications using Smart SSDs and estimating storage organizations for Smart SSDs.

$$TotalCost = \frac{I/Ocost + CPUcost}{\#ofSSDs} =$$

$$\frac{\{C_{FD} \times N_P\} + \{C_D \times N_D + C_{DT} \times D_{DT} + C_T \times N_T\}}{N_{SSD}}. \quad (3.1)$$

To validate the cost model, we used the following numbers (from our current Samsung Smart SSD device) in Table 3.2, and for the TPC-H Query 6 and Query 14 (for which we also have actual experimental results). As shown in Figure 3.4, the actual experimental results are relatively close to the values that are calculated from the cost model (within 10%).

Table 3.2: Properties of the Samsung Smart SSD.

C_{FD}	C_D	C_T	C_{DT}
2240	70	3.5	6

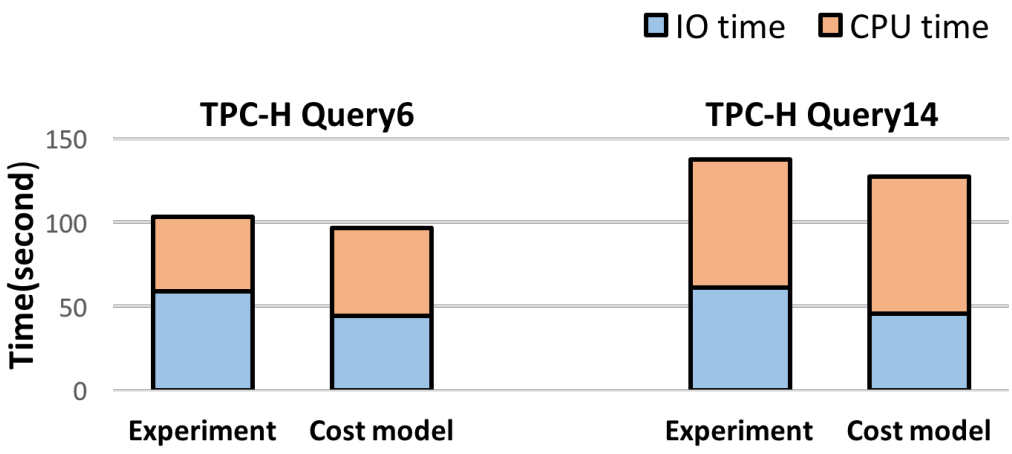


Figure 3.4: Validation of the Smart SSD cost model.

3.3 Hadoop MapReduce Framework with Smart SSDs

In this section, we describe our system design that integrates Smart SSDs in the Hadoop MapReduce framework [1] for a limited set of jobs/applications (i.e. this is is not a full-fledged integration of any MapReduce job with Smart SSD).

3.3.1 Hadoop MapReduce Framework

First, we briefly describe the general Hadoop MapReduce framework. Hadoop is a data processing framework that implements the MapReduce [26] programming model. MapReduce is designed to process large data sets with a parallel and distributed algorithm on a cluster. Hadoop runs on the Hadoop Distributed File System (HDFS), which is the primary distributed storage used by Hadoop applications. Hadoop typically processes a submitted job through three main phases: Map, Shuffle, and Reduce. In the initial Hadoop data processing phase, HDFS splits large input data into a set of small data blocks [1]. In the Map phase, each data block is processed by each Map task. All Map tasks are executed in parallel on a cluster of Hadoop DataNodes. Map tasks apply a user-defined Map function to its own input data block and generate a set of output data (i.e., intermediate key-value pairs). An output data (key-value pairs) from a Map task is then partitioned into the number of Reducers. In the Shuffle phase, each Map task's partitioned output data is transferred to each Reduce task based on the intermediate key. Reduce tasks sort and merge their own input data before they apply a Reduce function in the Reduce phase. The final output from each Reduce task is written in an HDFS file.

3.3.2 Offloading Map Tasks into Smart SSDs

As illustrated in Figure 3.5, we offload Map tasks to the Smart SSDs. The reasons that we only pushed down Map tasks into the Smart SSDs are twofold: (1) Map tasks take a large amount of input data and often produce relatively small output data; (2) Reduce tasks require sort and merge

[1] The size of each data block is configurable and the default size in Hadoop 3.0 [2] is 128MB

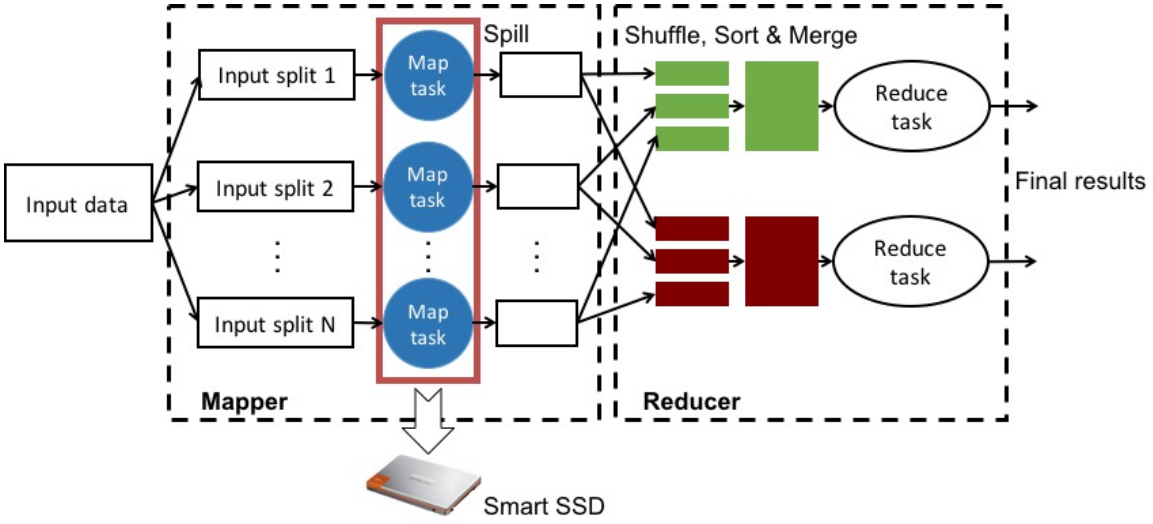


Figure 3.5: Offloading Map tasks into Smart SSDs.

operations that are not favorable to Smart SSDs. In Section 3.4.3.1, we will explain why sorting is not a good application for Smart SSDs.

Figure 3.6 depicts the overall workflow of the Hadoop MapReduce framework with Smart SSDs. Because of the language discrepancy between Hadoop implementation language and the Smart SSD firmware language (Java vs. C++), we implemented an additional interface layer called the Smart SSD Agent (see Figure 3.6) by adopting Java Native Interface (JNI) [39]. This Smart SSD Agent communicates with both the Hadoop system and the Smart SSDs. The Smart SSDs execute Map tasks once they take input split file information from the Smart SSD Agent. Each Smart SSD Map task processes each unstructured data block (128MB in size). As mentioned in Section 3.2, there are two types of volatile memory that we can utilize for data processing inside the Smart SSD. Based on the cost model and the rules that we proposed in Section 3.2, we can simply calculate the data processing cost with and without the TCM.

Assume that we process an 8KB page of unstructured data. Because the 8KB page is unstructured data, the CPU reads every byte to process the page, which results in 8K DRAM accesses per page. In this case, the CPU cost is $C_D \times N_D$. Alternatively, based on the *Cache rule* (see Section 3.2.1), we can utilize the TCM to avoid expensive DRAM accesses. In this case, the CPU cost

includes the memory copy cost ($C_{DT} \times D_{DT}$) from the DRAM input buffer to the TCM as well as the TCM access cost ($C_T \times N_T$). Based on a simple calculation using the numbers in Table 3.2, we recommend the following rule for data processing inside the Smart SSD:

- **Memory copy rule**: Copy input pages from the DRAM input buffer to the TCM if the number of DRAM accesses per page is greater than 1000.

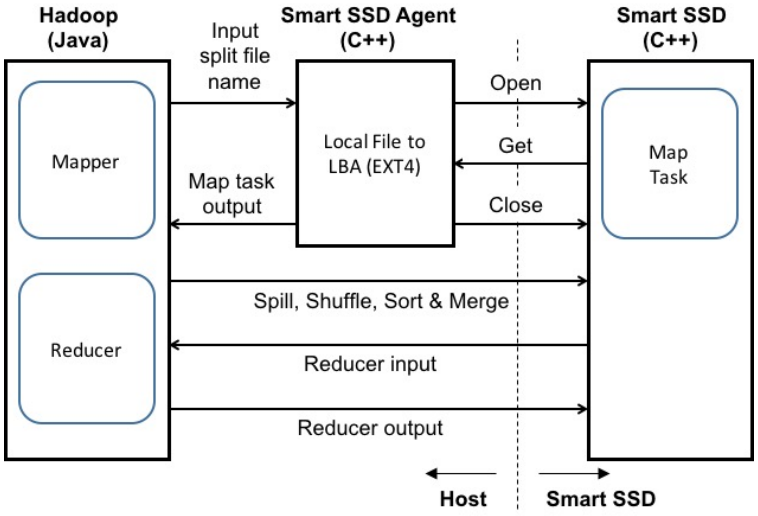


Figure 3.6: Software design for Hadoop with Smart SSDs.

Once each Smart SSD Map task is completed, the Smart SSD returns its output to the Smart SSD Agent instead of writing directly to the flash memory because the *current* Samsung Smart SSD does not support an internal write API. The rest of the Hadoop MapReduce procedure is same as in the case of the regular Hadoop MapReduce framework.

3.4 Evaluation

In this section, we present results from an empirical evaluation of Smart SSDs using the Hadoop MapReduce framework. In addition, we discuss further applications of the cost model and storage organization solutions for data processing inside the Smart SSD.

3.4.1 Experimental Setup

3.4.1.1 Workloads

For all experiments, we used Wikipedia data (∼100GB) from the Purdue PUMA benchmark [10]. Among the many Hadoop applications, we chose the `grep` application to evaluate the performance of Smart SSDs using the Hadoop MapReduce framework. The reasons that we chose `grep` are threefold: (1) the Map task execution time is much longer than the Reduce task time; (2) `grep` is a representative data intensive application among Hadoop applications; and (3) the output data of a Map task is much smaller than the input data. Thus, `grep` is a good candidate to show benefits from using a Smart SSD, which allows us to answer the question of how far can current Smart SSDs benefit such applications. This method also allows us to then use these results to explore (analytically) the roadmap for future Smart SSDs to benefit such and other data-intensive applications.

The search keyword for the `grep` application was "Microsoft" and this keyword rarely appeared (∼6000 times) in Wikipedia data (∼100GB). Thus, the size of the output data from each Map task is almost zero.

3.4.1.2 Hardware/Software Setup

We performed experiments with Hadoop 3.0 [2] framework with two base Hadoop cluster settings. The first configuration is a single DataNode with various number of SSDs and the second configuration is two DataNodes with various number of SSDs (see Figure 3.7). For each node (both NameNode and DataNode), we used a low cost commodity machine with an Intel i5 processor (3.20 GHz, 4 cores) and 32 GB memory. Each DataNode is equipped with Smart SSDs, each of which is a 400GB SSD with SAS 6Gb interface. All nodes run Ubuntu 14.04 LTS (64 bits).

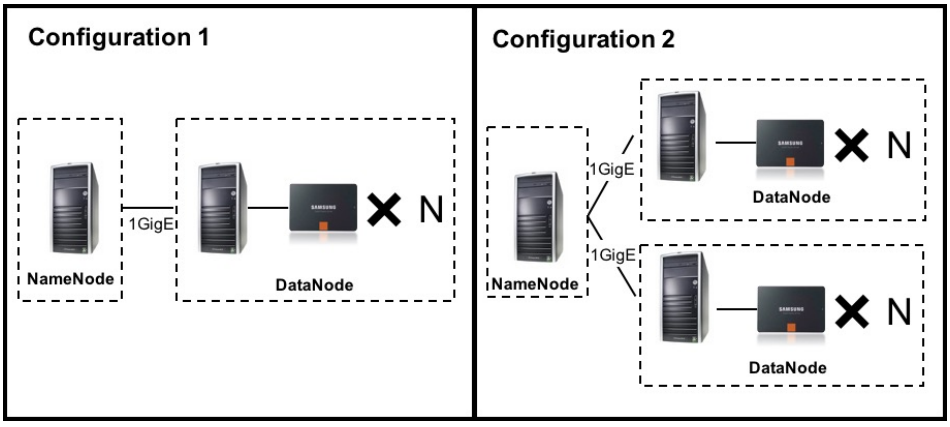


Figure 3.7: Hadoop cluster configurations.

All the results presented here are for cold experiments; i.e., there is no data cached in the DRAM prior to running each operation.

For all experiments, we used default the Hadoop 3.0 [2] configurations except for the block replication factor. We set up the replication factor as 1 for a single DataNode and 2 for two DataNodes.

3.4.2 Experimental Results

3.4.2.1 Single DataNode

For this experiment, we set up a Hadoop cluster with a NameNode and a DataNode (see Configuration 1 in Figure 3.7). The DataNode had various number of SSDs (i.e., N = 1, 2, 4, and 8), each of which stores HDFS data blocks. We ran a Hadoop `grep` job with a large text file ($\sim$100GB) extracted from Wikipedia pages.

Effect of adding SSDs: Figure 3.8 presents the end-to-end elapsed time when executing a `grep` job with various number of SSDs. As can be seen in Figure 3.8, the performance of Smart SSDs increases linearly as the number of SSDs increase, while the performance of regular SSDs remains the same. The reason for this is that all Smart SSDs run perfectly in parallel; therefore, the aggregate processing capabilities of Smart SSDs increase as the number of SSDs increases. On the other hand, with regular SSDs, the processing capabilities depend on the host CPU in a DataNode.

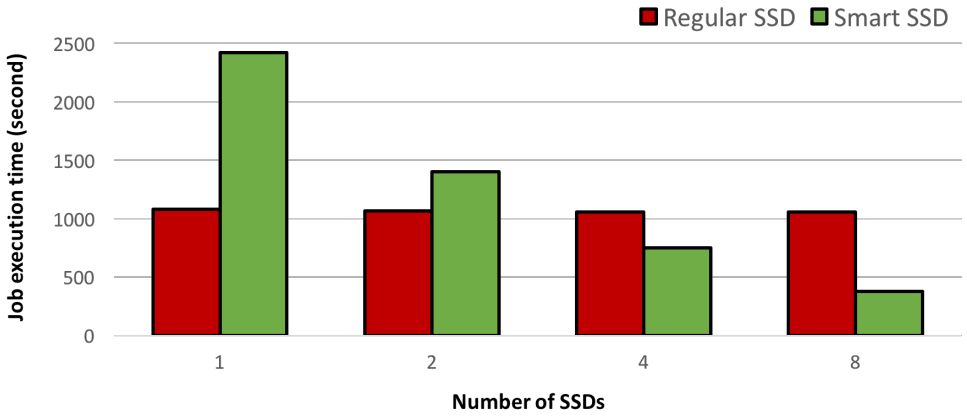


Figure 3.8: End-to-end `grep` execution time on the Wikipedia data (∼100GB) with various number of SSDs in a single DataNode.

Therefore, it does not change even after adding more SSDs to the DataNode. As can be seen in Figure 3.8, when N = 8, Smart SSDs execute the `grep` job about 2.8X faster than regular SSDs, and this gap will likely increase as more SSDs are added.

Effect of memory copy cost and TCM access cost for data processing inside the Smart SSDs: Because the input data for a `grep` job is an unstructured plain text file, a `grep` operation must access every byte in the input file and find matching strings. Thus, the number of DRAM accesses per 8KB page is greater than 1000. Based on the *Memory copy rule* (see Section 3.3.2), we copy each input data page from the DRAM input buffer into the TCM and then process it to evaluate the `grep` operation. Figure 3.9 describes the main bottlenecks in performing the `grep` job inside Smart SSDs based on the cost model. There are three main cost in performing the `grep` job inside a Smart SSD: (1) internal I/O cost, (2) the cost of copying input pages from the DRAM input buffer to the TCM, and (3) the cost of accessing TCM to process the input pages. The internal I/O cost is negligible because of the huge aggregate internal I/O bandwidth in Smart SSDs. The memory copy cost and the TCM access cost are two main bottlenecks in performing a `grep` job inside Smart SSDs.

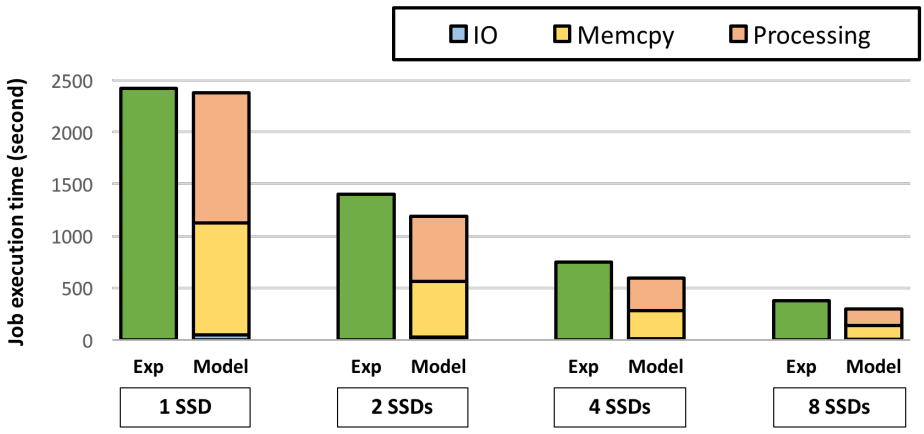


Figure 3.9: Smart SSD `grep` execution time and analysis with the cost model.

3.4.2.2 Multiple DataNodes

For this experiment, we set up a Hadoop cluster with a NameNode and two DataNodes (see Configuration 2 in Figure 3.7) where each DataNode has four SSDs (i.e., N = 4). We ran a Hadoop `grep` job with a large text file (∼100GB) extracted from the Wikipedia pages.

Effect of adding DataNodes: Figure 3.10 presents the end-to-end elapsed time when executing a `grep` job with eight SSDs. In this figure, for the single DataNode setting, we used Configuration 1 (see Figure 3.7) where N = 8. For the two DataNodes setting, we used Configuration 2 (see Figure 3.7) where N = 4. As can be seen in Figure 3.10, the performance of regular SSDs increases as the number of DataNodes increases, while the performance of Smart SSDs remains the same. The reason for this is that the aggregate processing capabilities with Smart SSDs does not change because the total number of SSDs is eight in both experiments. On the other hand, with regular SSDs, the processing capabilities that depend on the host CPU in DataNodes increased by adding an additional DataNode.

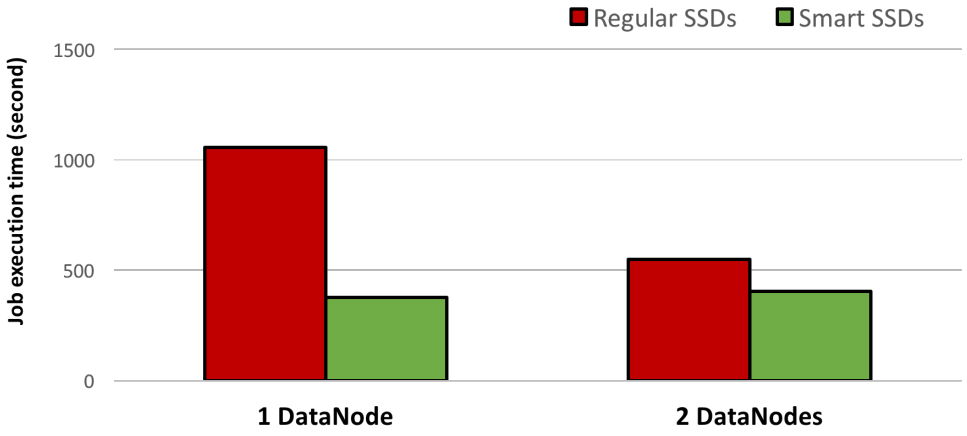


Figure 3.10: End-to-end `grep` execution time comparison between a single datanode and two datanodes with eight SSDs.

3.4.3 Discussion

A crucial observation is that the lack of processing capability inside the Smart SSD can be overcome by adding more SSDs. For example, as seen in Figure 3.8, a single Smart SSD's performance was worse than that of a single regular SSD. However, multiple Smart SSDs outperformed regular SSDs (up to 2.8X with 8 Smart SSDs) due to their aggregate processing capabilities. Recent servers are equipped with more than 24 storage device slots. We can therefore predict about 8X speedup (compared to regular SSDs) when using the aggregate processing capabilities of 24 Smart SSDs.

3.4.3.1 Other Hadoop Applications

Wordcount Figure 3.11 presents the experimental results for a Hadoop `wordcount` [13] job that counts the number of occurrences of each word in given input data. We used Configuration 2 (see Figure 3.7), where each DataNode has four SSDs (i.e., N = 4). Smart SSDs underperformed regular SSDs because the Wikipedia data had too many distinct words; therefore, the output data from each Map task was too large. This high selectivity value ($\sim$100% selectivity) of the Map task saturated the performance of Smart SSDs (refer to Section 2.4.2.1).

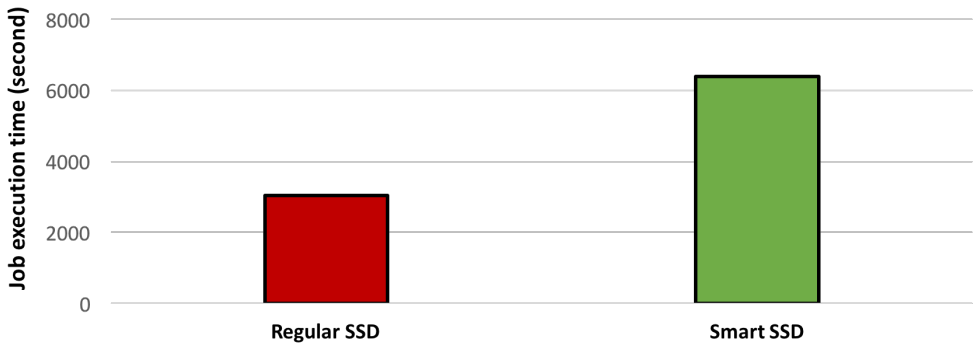


Figure 3.11: End-to-End `wordcount` execution time on the Wikipedia data (∼100GB) with two DataNodes, each of which has four SSDs.

Sorting Sorting is an important operation in a data processing system. We estimate the performance of a sorting operation in the Smart SSD based on the cost model. We note that sorting cannot be currently implemented/performed with the current Samsung Smart SSD because it does not support an internal write API. Even if we assume there is an internal write API for the Smart SSD, sorting inside the Smart SSD would be very slow because the size of DRAM inside the Smart SSD is much smaller than the host DRAM size (by a factor of 1,000).

Assume we sort $8M$ 8KB pages (∼60GB), each of which has 80 data entries. Assume also that the host DRAM size is about 64GB and the Smart SSD's DRAM size is about 64MB. Then, sorting in the host requires only a single pass, while sorting in the Smart SSD requires an external sorting. If we perform a two-way external merge sort, it will require about $\log_2 8M$ passes, each of which requires $2 \times 8M \times 80$ DRAM accesses. Based on the cost model, we can estimate the sorting execution time for this example, as shown in Figure 3.12. In this example, we conservatively assume the host DRAM access latency is 100 cycles and the host CPU clock cycle is 3.2 GHz. The Smart SSD will be about 4X slower than the regular SSD.

3.4.3.2 Smart SSD Storage Organizations

Based on the cost model, we explore a number of key dimensions related to storage organizations for the Smart SSD. These dimensions are described below.

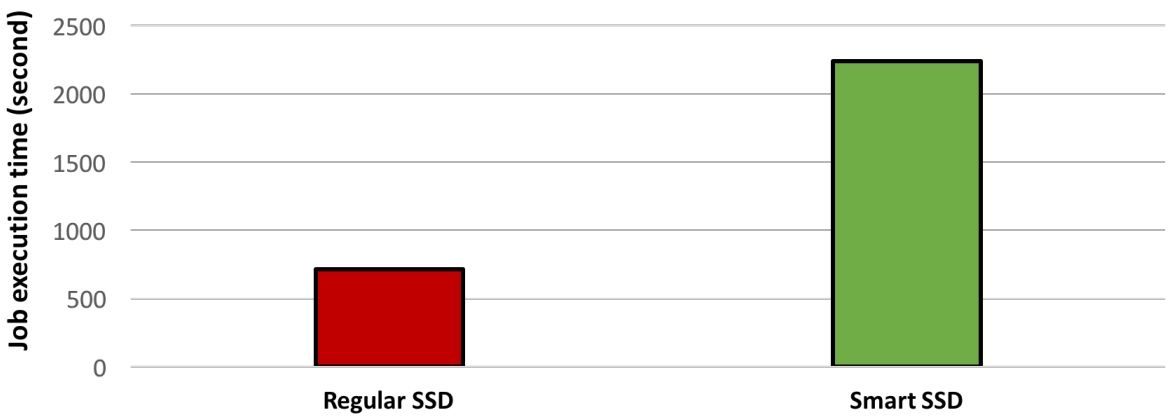


Figure 3.12: Estimated sorting execution time with a regular SSD and a Smart SSD based on the cost model.

Row-Stores vs. Column-Stores The relative benefits of row-store and column-store organization for the Smart SSD can be briefly estimated based on the cost model. Assume a simple query that scans a heap table (with 64 integer columns). The query has a selection predicate on the first column of the table, and the selectivity of this query is nearly 0%. Now imagine there are two data storage models: row-store and column-store. The table size in the row-store is 100GB, while the size of the first column in the column-store is about 1.5GB. We can estimate the performance of the Smart SSD and the regular SSD with both row-store and column-store data (see Figure 3.13). Using row-store data, the Smart SSD provides a performance benefit over the regular SSD, while the regular SSD outperforms the Smart SSD in a column-store data storage model.

NSM vs. PAX We evaluate the traditional N-ary Storage Model (NSM) and the PAX layout [21] in which all the values of a column are grouped together within a page. Figure 3.14 estimates the NSM and the PAX layout using the same example as described in the previous paragraph based on the cost model. Because all the values of a column in a page are stored contiguously in the PAX layout, we are able to use the LDM instruction [3] [2] to load multiple values at once, reducing the

[2]The load multiple instruction (LDM) allows loading data into any subset of the 16 general-purpose processor registers from memory, using a single instruction.

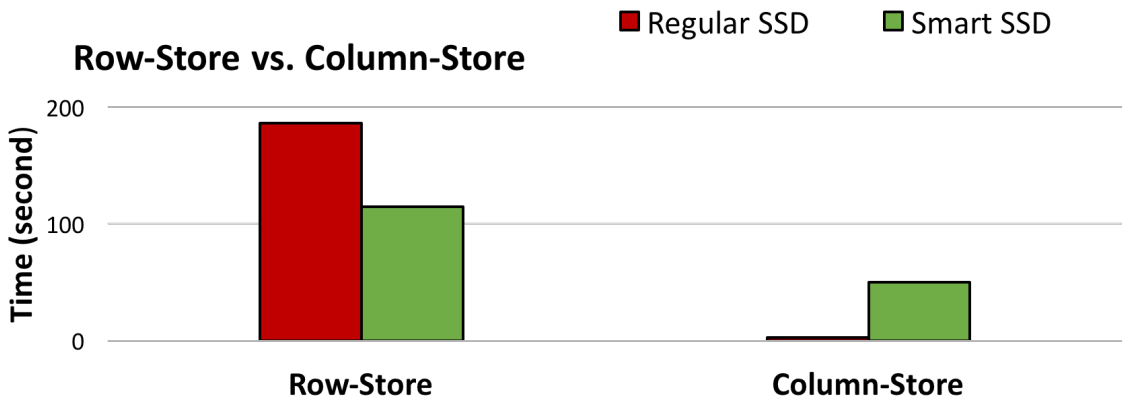


Figure 3.13: Estimation for storage organizations with a regular SSD and a Smart SSD.

number of (slow) DRAM accesses. Given the high DRAM access latency in the SSD, the columnar PAX layout is more efficient than a row-based NSM for the Smart SSD (see Figure 3.14).

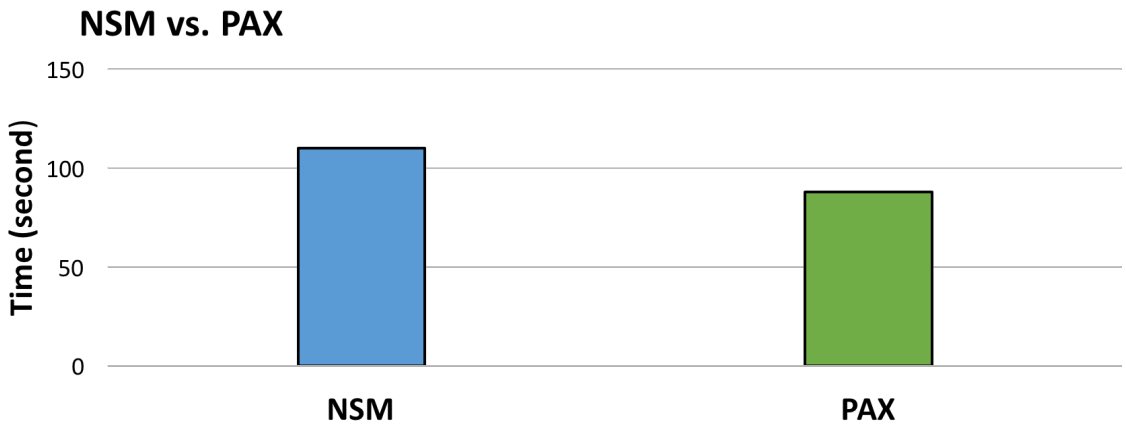


Figure 3.14: Estimation for the NSM and PAX data layout with a Smart SSD.

3.5 Conclusion and Roadmap for Future Smart SSD Design

In this chapter, we demonstrated the potential performance improvement of using Smart SSDs in a non-relational database system. Because of the low computation capabilities in Smart SSDs and the characteristics of unstructured data processing, using a single Smart SSD does not offer a performance improvement compared to the use of a regular SSD. However, as shown in Figure 3.8, the aggregate computation capabilities of a cluster of Smart SSDs could provide a significant performance improvement over the use of regular SSDs. In our experiments with eight Smart SSDs, we showed that Smart SSDs could improve the performance of a `grep` job by up to 2.8X over the use of regular SSDs. Also, based on the cost model, we estimate that the performance improvement will be much larger when adding more SSDs (up to 8X with 24 SSDs).

3.5.0.1 Roadmap for Future Smart SSD Design

Based on the Smart SSD's current software/hardware properties and our analysis using the cost model, we propose a roadmap to guide hardware/software design choices for future Smart SSDs as follows:

1. We need a Direct Memory Access (DMA) memory copy operation to reduce the expensive memory copy cost from the DRAM input buffer to the TCM (see Figure 3.9).

2. Currently, the internal Smart SSD I/O is serialized with CPU computations. This results in degradation of the Smart SSD's performance. Thus, we need asynchronous internal I/O for the Smart SSD functionality.

3. Larger DRAM size is needed to reduce certain computation, such as the number of passes when sorting inside the Smart SSD. Without a large DRAM size, computing inside the Smart SSD faces a sheer algorithmic challenge as algorithms (like sorting) are very sensitive to the available memory size.

4. Row-store, especially PAX, is more beneficial for the Smart SSD, as shown in Figure 3.13 and Figure 3.14.

5. The current Smart SSD cannot run multiple Smart SSD processes at the same time. To support concurrent queries and multi-tenant databases, a multiprocessing feature should be included in future Smart SSDs.

Chapter 4

Aggressive Buffer Pool Warm-up after Restart in SQL Server

In many settings, a database server has to be restarted either in response to a failure event, or in response to an operational decision such as moving a database service from one machine to another. However, such restarts pose a potential performance problem as the new database server starts off with a cold buffer pool. As a result, the database application experiences a dramatic reduction in performance right after the restart, since just before the restart the database buffer pool was filled with hot pages and after the restart the database buffer pool is empty. To address these issues, traditional database systems use mechanisms such as SQL Server's aggressive page expansion and MySQL's buffer pool preloading. However, these approaches have key limitations including long warm-up times, possible early hot page eviction, user query performance saturation, and failure restart. In this chapter, we present a new framework for SQL Server that allows continual capturing of the state of the buffer pool, and restoring the server state quickly with a snapshot of the buffer pool at restart. Our empirical evaluation demonstrates that our method reduces the time to regain peak performance by a factor of 2X or more over the previous approaches.

4.1 Introduction

In production deployments, database servers restart for many reasons, including software updates and hardware/software failure. In cloud settings, such restarts are more frequent due to operational considerations, such as the need to move a database service from one machine to another. Such restarts result in performance degradation as the restarted service starts with a cold buffer pool. Figure 4.1 describes the long ramp-up issue that is associated with restarts.

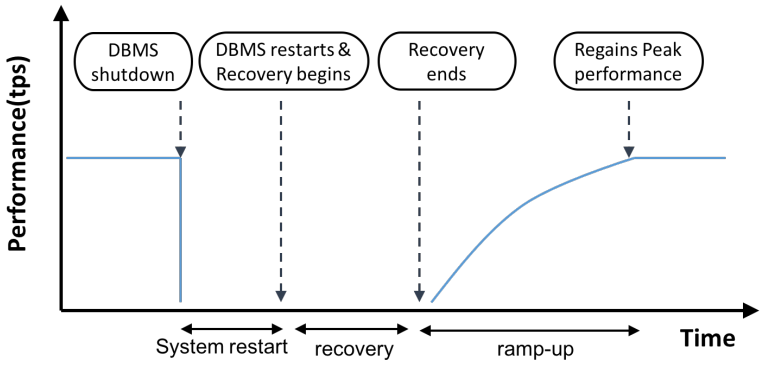


Figure 4.1: DBMS performance trend on restart.

To address the issue of long ramp-up times, traditional database management systems (DBMSs) try to warm up the database buffer pool quickly on restart. For example, Microsoft SQL Server uses an aggressive page expansion technique. This technique warms up the buffer pool by expanding every single page read request to multiple (8 in SQL Server 2014) adjacent pages, including the requested page before the buffer pool is filled [30]. Thus, this technique aims to fill the cold buffer pool faster. Another technique, and one that MySQL takes, is called buffer pool preloading. This technique captures the metadata of the buffer pool pages during shutdown, and preloads the pre-captured buffer pool pages when the system restarts [8]. However, these existing approaches have a number of shortcomings, including that the pages that are fetched may not have the highest utility (from the subsequent buffer hit rate perspective), thus limiting the effectiveness of these methods. (See Section 4.3 for a more detailed discussion.)

In this chapter, we present a set of mechanisms to reduce the ramp-up time and evaluate the performance of our method compared to the existing methods used in SQL Server and MySQL.

The key contributions of this chapter are as follows:

- We propose checkpointing the metadata of the cached buffer pool pages (we define these pages as hot pages in this chapter) periodically, and use this information to warm up the buffer pool aggressively on restarts.

- We design a Warm-up Estimator that pre-estimates if our framework provides a speedup over the base SQL Server approach.
- We design and develop new LRU shifting and Piggybacking mechanisms (see Section 4.4.3) to efficiently warm up the buffer pool.
- We perform an extensive evaluation of our framework with various hardware settings, including Cloud virtual hard disks (VHDs), HDDs, and SSDs, using an OLTP (TPC-E) workload. Our results using SQL Server show that our framework reduces the ramp-up time by up to 2-3X compared to the approaches in SQL Server and MySQL.

The remainder of this chapter is organized as follows: Section 4.2 describes background information, and Section 4.3 illustrates the differences between our idea and other approaches. Section 4.4 describes the details of our framework. Section 4.5 presents the performance evaluation and discussion. Related work is covered in Section 4.6, and our concluding remarks are in Section 4.7.

4.2 Background

In this section, we describe characteristics of I/O subsystems, Microsoft SQL Server, and MySQL that are relevant to the framework presented in Section 4.4.

Table 4.1: Maximum sustainable random-read performance in IOPS when using 8KB, 64KB and 512KB requests for azure VHD, HDD and SSD, respectively.

	8KB	64KB	512KB
Azure VHD	500	500	500
SAS HDD	342	295	102
SATA SSD	19026	3714	464

4.2.1 I/O Subsystems

To better understand the proposed framework (which is described in Section 4.4), and the results (which are described in Section 4.5), the random read performance of three underlying storage devices that are used in our analysis are shown in Table 4.1. These storage devices are Microsoft Azure [18] Standard VHD, a 10K RPM SAS HDD, and a SATA SSD. The performance numbers shown in the table are IOPS that is measured using IOmeter [5]. As can be seen in Table 4.1, the three devices show different performance trends as the I/O request size varies. For example, since Azure VHD[1] can provide up to 500 IOPS per disk regardless of the request size, using the 512KB request size showed much higher throughput compared to the cases when smaller request sizes are used. On the other hand, similar throughputs were measured when using 64KB and 512KB requests on the SSD. These I/O subsystem characteristics are considered in our framework to reduce the ramp-up time (see Section 4.4.2.1).

[1]Microsoft AZURE provides different IOPS numbers for different storage services. For example, it provides 300, 500, and 5000 IOPS for Basic, Standard, and Premium storage services, respectively.

4.2.2 Microsoft SQL Server

Microsoft SQL Server aggressively expands a single page read request to an (consecutive) eight-page request when the buffer pool is not full, so as to fill up the pool as quickly as possible. In addition, the buffer pool manager maintains the last two references to each buffered page (i.e., LRU-1, LRU-2). This reference information is used to implement the buffer pool page replacement policy, which is based on the LRU-K replacement algorithm [42], with K = 2.

4.2.3 MySQL

MySQL employs a cache preloading mechanism in order to avoid a lengthy ramp-up period after restarting the system [8]. This cache preloading mechanism has three major stages. First, the system captures the buffer pool state by recording the list of page identifiers (i.e. pageID) of the pages in the buffer pool when the system shuts down. This option can be set to only record a fraction of the most recently used pages that are in the buffer pool. Second, on restart, the system sorts the list in an ascending order based on the page ID. Lastly, the pages are preloaded to restore the buffer pool state when the system restarts.

4.3 The Need for a New Framework

In this section, we describe the limitations with the SQL Server's page expansion and MySQL's buffer pool preloading techniques for the two possible restart scenarios that are shown in Figure 4.2. Then, we explain how our approach addresses these limitations.

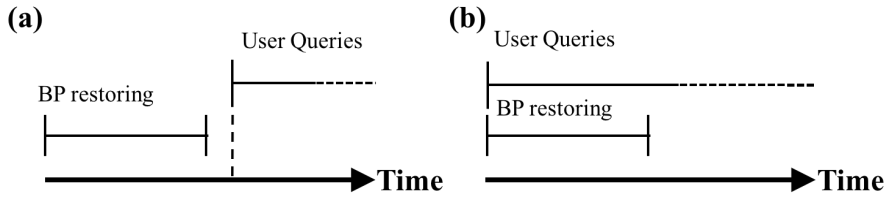


Figure 4.2: Possible scenarios: (a) User queries are processed only after the buffer pool restoring step is completed (i.e., after the buffer pool is warmed up). (b) User queries start to be processed right after the system restarts (In this case the buffer pool restoring process runs simultaneously in background).

4.3.1 Where SQL Server Falls Short

SQL Server expands every single page read request to load adjacent eight pages until the buffer pool is full. However, since this technique does not capture the buffer pool state before shutdown, it is not applicable to Scenario (a) in Figure 4.2. In addition, the buffer pool inevitably ends up containing several cold pages mixed with hot pages when the warm-up phase is done: In the worst case, 7/8th of the buffer pool pages could be cold.

4.3.2 Where MySQL Falls Short

The MySQL buffer pool preloading technique helps warm up the buffer pool before processing user queries (Scenario (a)). However, it still has a number of drawbacks. First, since the technique issues only single-page I/O requests to warm up the buffer pool, it could take a large amount of time to fill up the buffer pool with hot pages. For example, one of our experiments shows that more than an hour was necessary to completely warm up the 48 GB buffer pool on AZURE VHDs, while

our framework needs only 9 minutes (see Section 4.5.2.1). Second, MySQL does not use the LRU values that the preloaded hot pages had before the system restarts, which may result in hot pages getting evicted sooner than it would have with this information. Third, there is no throttle-control mechanism between processing user queries and executing the buffer pool restoring process. As a result, when user queries and the buffer pool restoring process run concurrently (Scenario (b)), we observed that user queries were rarely processed because the I/O subsystem was saturated by the buffer pool restoring process (See Figure 4.14). Finally, the buffer pool state is captured only when the system is cleanly shutdown; thus, the system still needs a long ramp-up time when restarting from a failure.

4.3.3 Key Aspects of the New Framework

Our framework addresses several limitations with the existing methods in MySQL and SQL Server. First, since our framework captures the buffer pool state periodically (see Section 4.4.1), it has the hot page metadata information available right after restart. Thus, restoring the buffer pool state can be done in advance if there is enough time before user queries start to run, as in Scenario (a). Second, we address the issue of possible long warm-up times with MySQL by finding the best I/O request size instead of using single page I/Os. Third, we assign LRU values to the preloaded hot pages by applying a LRU-Shifting technique (see Section 4.4.3.2). Fourth, our piggybacking technique (see Section 4.4.3.3) helps warm up the buffer pool without saturating the user queries' performance. Lastly, since our framework checkpoints the metadata of the cached buffer pool pages periodically, we can aggressively warm up the buffer pool even on failure restarts.

4.4 The New Framework

Figure 4 shows the architecture of the framework that we propose in this chapter.

Table 4.2: Notations used in the Warm-up Planner.

P = $\{p_1, ..., p_n\}$, a list of the hot pages' metadata where each consists of Page ID (which consists of a database ID, a file ID, and a page number in the file), LRU-1, and LRU2.
S = $\{s_1, ..., s_m\}$, where each is a candidate I/O request size (e.g., 8KB, 64KB, or 512KB in our example)
R = $\{r_1, ..., r_m\}$, where is the random read performance in IOPS when s_i-sized requests are used.
C = $\{C_1, ..., C_m\}$, where each is a list of I/O requests for s_i
H = $\{H_1, ..., H_m\}$,, where each is a list of hot pages, $H_i \subseteq P$, and $H_i \subseteq C_i$.
S_E = Page expansion size of DBMS (i.e., 64KB for SQL Server)

4.4.1 Buffer Pool State Recorder: BSR

The main role of Buffer Pool State Recorder (BSR) is to periodically record the metadata (i.e., page ID, LRU-1, and LRU-2) of the cached pages in the buffer pool to an on-disk file.

4.4.2 Warm-up Planner

The Warm-up Planner consists of three subcomponents: a Warm-up I/O Planner, a Warm-up Estimator, and a Warm-up Arranger. The Warm-up Planner is executed as a part of the recovery process when the system restarts.

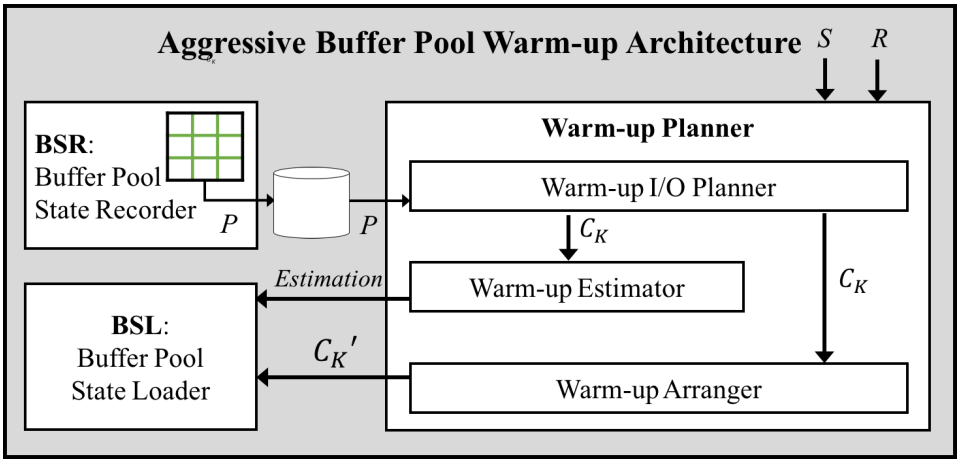


Figure 4.3: The framework has three components: Buffer Pool State Recorder (BSR), Warm-up Planner, and Buffer Pool State Loader (BSL).

4.4.2.1 Warm-up I/O Planner

We use an example to illustrate the utility of the Warm-up I/O Planner. Suppose that the system has a 100MB buffer pool (12,800 pages, with 8KB page size), and these buffer pages are laid out on disk as shown in Figure 4.4 (i.e., 8 buffer pool pages are evenly distributed across 16 consecutive disk pages, followed by 48 disk pages containing no buffer pool pages). Once the system has been restarted, the buffer pool needs to be warmed up by loading the pages that were previously cached in the buffer pool. Here, the expected warm-up time can be varied based on the I/O request size. For example, when there are three candidate I/O request sizes (8KB, 64KB, and 512KB), each candidate size requires 12800, 3200 (12800/4, 4 hot pages in every 8 page chunk), and 1600 (12800/8, 8 hot pages in every 64 page chunk) I/O requests to fill up the 100MB buffer pool, respectively. With the IOPS numbers in Table I, we can calculate the total warm-up time. In AZURE VHD, for instance, the warm-up time is 25.60 seconds (12800IOs/500IOPS), 6.40 seconds (3200IOs/500IOPS), and 3.20 seconds (1600IOs/500IOPS) when using 8KB, 64KB, and 512KB I/O request sizes, respectively. Therefore, the 512KB is the best I/O request size for AZURE VHDs, while 64KB and 8KB are the best I/O request sizes for HDDs and SSDs,

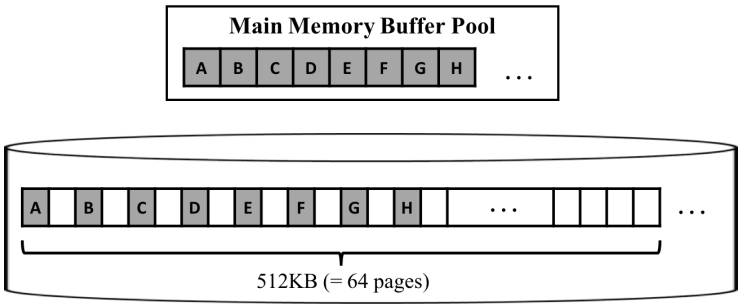


Figure 4.4: A sample buffer pool layout on disk when the Warm-up Planner is executed.

respectively (see Figure 4.5). The details for the Warm-up I/O Planner is presented in Algorithm 1.

1. The algorithm sorts P in an ascending order based on the ID of each page.

2. It then clusters the sorted pages[2], P', according to the candidate I/O request sizes, S, (i.e., 8KB, 64KB or 512KB in our experiments), and finds the best I/O request size, s_K, by using the IOPS numbers per each request size.

3. It returns the best I/O request chunks, $C_K = \{c_{K,1}, ..., c_{K,p}\}$ where each $c_{K,i}$ contains the start page ID, the number of pages, and the list of LRU-1, LRU-2 values of the hot pages. The choice of the I/O request size can be tuned based on each request, but for simplicity we only pick one I/O size in the experiments presented in Section 5.

4.4.2.2 Warm-up Estimator

Warm-up Estimator, which is described in Algorithm 2, determines if using our framework results in a speedup over SQL Server's aggressive page expansion strategy (with an assumption that all the expanded pages, 8 pages, are hot pages). For example, the random read performance for the page expansion size, r_E, is 295 IOPS (see Table 4.1). Then, the expected warm-up speed of the page extension, A, is $s_E \times r_E$ = 64 KB/IO $\times$ 295 IOPS = 18,880 KB/sec. In our case, we

[2]Note that we apply early chunk pruning that discards clusters containing few hot pages. Since is 64KB (8 pages), we discard clusters that have less than 8 hot pages.

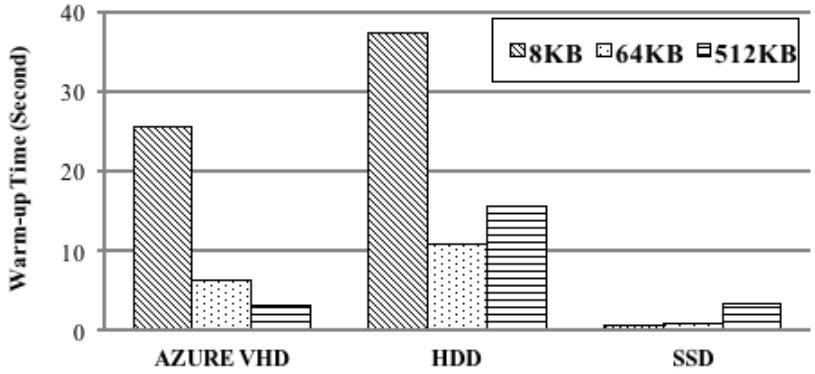


Figure 4.5: Estimated total warm-up time with the buffer page layout shown in Figure 4.4.

need the average hot pages per each I/O request chunk (AVG Hot Page Density), which is the total hot pages over the total I/O request chunks, $\frac{H_K}{C_K}$. Thus the warm-up speed of our framework, B, is KB/sec. If B is greater than A (i.e., if AVG Hot Page Density is greater than Break-Even Hot Page Density, $\frac{H_K}{C_K} \times r_K$), we can guarantee that the warm-up speed our framework brings is faster compared to simply using the aggressive page expansion. The benefit of this technique is explained in Section 4.5.3.2 and Section 4.5.4.

4.4.2.3 Warm-up Arranger

The Warm-up Arranger module sorts the list of I/O request chunks, C_K, based on how important they are. The criteria that we chose are as follows: 1. average of LRU-2 values, 2. the number of hot pages, 3. sum of LRU-2 values, and 4. max of LRU-2 values. Also we build a hash index for each hot page with the corresponding I/O request chunk for Piggybacking (see Section 4.4.3.3).

4.4.3 Buffer Pool State Loader: BSL

The Buffer Pool State Loader (BSL) is executed right after the recovery process and the Warm-up Planner are completed. It issues actual I/O requests based on the two outputs from the Warm-up Planner — the sorted I/O request chunks ($C_K^{'}$) and $Estimation$ (see Figure 4.3). This BSL component considers the two cases illustrated in Figure 4.2, which allows it to: a) either warm up the buffer pool before accepting user queries, or b) warm up the buffer pool while concurrently

Table 4.3: Algorithm 1. The Warm-up I/O Planner

Input: P, S, R

Output: C_K

1 $P' \leftarrow$ SortPages(P);

2 **for** i to m **do**

3 | $C_i \leftarrow$ MakePageClusters (P', s_i);

4 **end**

5 $s_K \leftarrow$ FindBestIORequestSize(C, R);

6 Return

admitting new user queries before the warm-up process is completed. Note that in Scenario (b), if the value of Estimation is False, the BSL does not issue I/O requests for preloading hot pages since it cannot guarantee a speedup over the user query's buffer pool warm-up as explained in Section 4.4.2.2. Also, the BSL component employs techniques such as discarding cold pages, LRU-shifting, and piggybacking, which are briefly described below.

4.4.3.1 Discarding cold pages

Since each I/O request chunk possibly contains cold pages, we discard such cold pages after each I/O request is completed, thereby not polluting the buffer pool with cold pages.

4.4.3.2 LRU-shifting

Since SQL Server uses its own (simulated clock) tick for LRU values, we save the current time tick, T, as an output of the BSR method. On restart, we use T as the start time tick so that we can simply use the pre-saved LRU values for the preloaded hot pages after the system restarts.

Table 4.4: Algorithm 2. The Warm-up Estimator

Input: C_K

Output: $Estimation$

1 $A \leftarrow s_E \times r_E$;

2 $B \leftarrow \frac{H_K}{C_K} \times r_K$;

3 **if** $B > A$ **then**

4 | $Estimation \leftarrow True$;

5 **else**

6 | $Estimation \leftarrow False$;

7 Return

4.4.3.3 Piggybacking

As shown in Figure 4.6, when an I/O request for a page from the user query arrives at time T, we search a list of I/O request chunks (which are indexed by the page ID), and issue a request for a chunk c instead of the page p, if there is a chunk c that contains p.

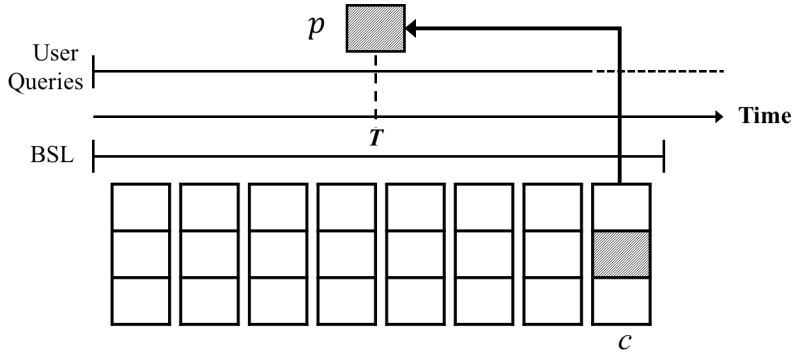


Figure 4.6: Piggybacking.

4.5 Evaluation

In this section, we present results from implementing our framework as well as MySQL's buffer pool preloading in Microsoft SQL Server 2014 [14]. We evaluate these two methods compared to the baseline (the default SQL Servers aggressive page expansion with the user queries). Table 4.5 shows the implementation details about our framework, which we call "SQLServer/Warm" in this section, and MySQL's buffer pool preloading in SQL Server.

4.5.1 Experimental Setup

4.5.1.1 Hardware / Software Setup

We ran the experiments on three different I/O subsystems — Microsoft Azure Cloud VHDs, 10K RPM SAS HDDs, and SATA SSDs. For the cloud VHD evaluation, we used a Microsoft Azure A8 virtual machine instance [18]. The system runs SQL Server 2014 on the 64-bit Windows O/S (Windows Server 2012 R2 Datacenter) with 48GB of memory dedicated to the DBMS. The databases were created on a file group that spans eight 1TB VHDs. Four additional 1TB VHDs were dedicated to the OS, the transactional log, the tempdb [11], and the BSR (Section 4.4.1), respectively. For the HDD evaluation, all experiments were performed on a system using the same OS and DBMS configuration as the cloud evaluation. The system has two Intel Xeon L5630 2.13GHz quad core processors. For the OS and the transactional log, we used two 146GB HDDs, respectively. The databases were created on a file group that spans eight 300GB HDDs. Two additional 300GB HDDs were dedicated to the tempdb, and the BSR (Section 4.4.1), respectively. For the SSD evaluation, we used the same machine/settings as the HDD evaluation while the databases were created on a file group that spans four 500GB SSDs instead of eight 300GB HDDs.

4.5.1.2 Workload

For the workload, we used the TPC-E benchmark [15], which is a read-intensive OLTP workload. We used three different sizes of dataset: 10K customers database ($\sim$100GB), 20K customers database ($\sim$200GB), and 80K customers database ($\sim$800GB). We measured the number of

(Trade-Result) transactions executed within a second (tpsE). As per the TPC specification, we set the recovery interval to 7 minutes. The performance is sampled every second. For MySQL and SQLServer/Warm, we warmed up the buffer pool for a long enough time (∼3 hours) to achieve stabilized peak performance and captured the buffer pool state. Then we shut down and restarted the system (clean restart). In Scenario (a), MySQL and SQLServer/Warm warmed up the buffer pool before the user queries start to run, while in Scenario (b) the user queries start right after the recovery process is completed. In addition to this, Buffer Pool State Recorder (BSR) was added to the regular checkpoint process, and IOPS numbers specified in Table 4.1 are used as inputs (S, R) to the Warm-up Planner.

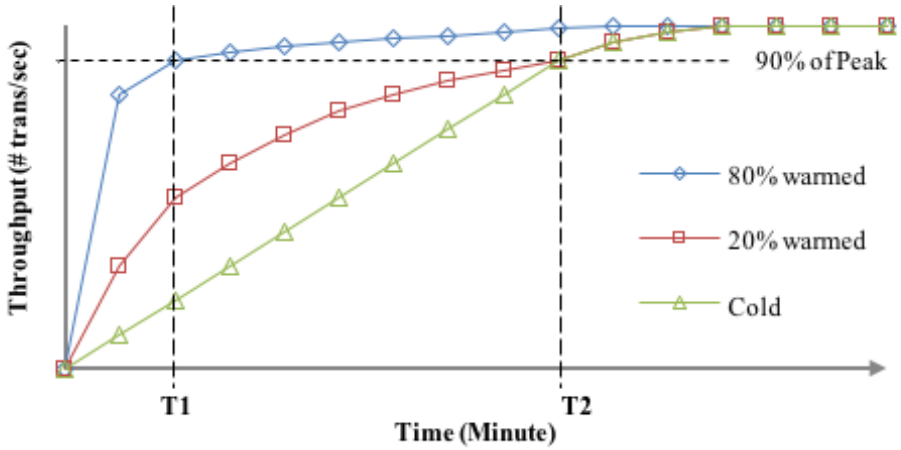


Figure 4.7: Example for the ramp-up time metrics.

4.5.1.3 Metrics

The goal of our framework is to reduce the ramp-up time while increasing the transaction throughput during that period. To evaluate the effectiveness of our framework with respect to this goal, we define two metrics:

- Peak 90% time is defined as the elapsed time for the user queries to reach 90% of peak performance.

- Average throughput measures the average transaction per second (tpsE) from the beginning of the user queries to the peak 90% time of baseline. In our experiments, we use SQL Server's aggressive page expansion as the baseline.

Figure 4.7 gives a synthetic example of why these two metrics are important in practice. It shows a sketch of hypothetical throughputs (transactions per second) over time on restart. Assume that we have three different states for the buffer pool on restart. One is an 80% warmed buffer pool, where 80% of the hot pages are loaded to the memory before the user queries start to run. The other two cases correspond to a 20% warmed buffer pool scenarios and a cold buffer pool scenario. T1 is the peak 90% time for the 80% warmed buffer pool scenario, and T2 is the peak 90% time for the 20% warmed buffer pool and the cold buffer pool scenarios. In this situation, the 80% warmed buffer pool case is the best outcome since it delivers the shortest peak 90% time (T1) as well as the highest average throughput (assume that the baseline is the cold buffer pool case). Between the 20% warmed and the cold buffer pool scenarios, the former is better since it provides higher average throughput than the cold buffer pool scenario, while both achieve the same peak 90% time.

4.5.2 Cloud VHD Evaluation

In this section, we evaluate SQLServer/Warm and compare it to the baseline and MySQL approaches on the cloud VHD I/O subsystem for two different scenarios (see Figure 4.2). For all three datasets (10K, 20K, and 80K customer databases), the Warm-up I/O Planner (see Section 4.4.2.1) returns 512KB (64 pages) as the best I/O request size.

4.5.2.1 Scenario (a)

Figure 4.8 shows the first one hour of performance behavior since the user queries started. We only show the 10K customer dataset result since the other cases (20K and 80K customer databases) demonstrated similar patterns. As shown in Figure 4.8, MySQL and SQLServer/Warm quickly reach peak performance when the system restarts since the buffer pool was warmed up for those cases. Figure 4.9 shows the total warm-up time for MySQL and SQLServer/Warm for 10K, 20K,

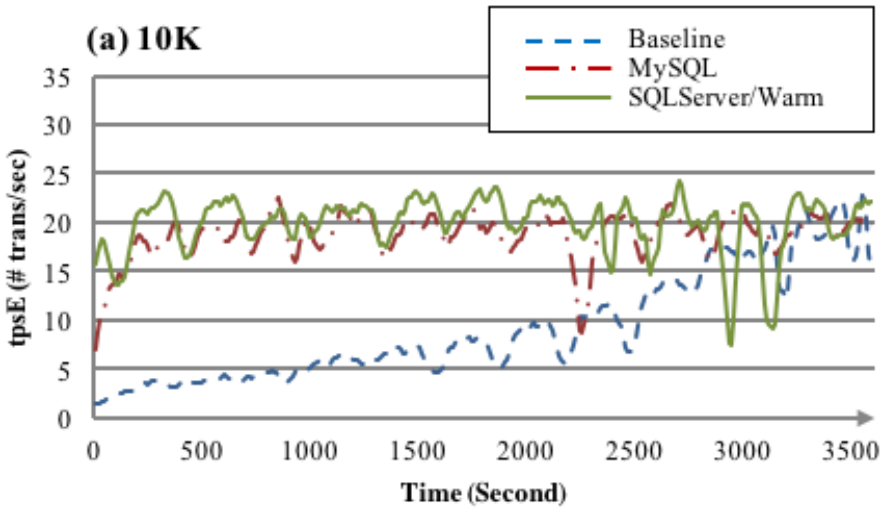


Figure 4.8: Performance trend of the cloud VHD scenario (a) with 10K customer database.

and 80K customer databases. As can be seen in Figure 4.9, SQLServer/Warm reduces the warm-up time by up to 10X compared to MySQL.

4.5.2.2 Scenario (b)

In our experiments, the Break-Even Hot Page Density (see Section 4.4.2.2) of the cloud VHD I/O subsystem was 64KB (8 pages), and the AVG Hot Page Densities for 10K, 20K, and 80K customer databases are 40 pages, 25 pages, and 16 pages, respectively. Therefore, all three cases guarantee a faster warm-up behavior over the baseline by using our framework since all three AVG Hot Page Densities are greater than the Break-Even Hot Page Density. Figure 4.10 shows the peak 90% time and the average throughput for all three on the cloud VHD I/O subsystem. As shown in Figure 4.10, SQLServer/Warm reduces the peak 90% time by 2.98X, 2.42X, and 1.55X, and provides 1.88X, 1.35X, and 1.22X average throughput improvement over the baseline for 10K, 20K, and 80K customer databases, respectively. With the 10K customer database, where the biggest AVG Hot Page Density is obtained, the best performance improvement is achieved compared to other two cases (20K and 80K). Also SQLServer/Warm reduces the peak 90% time by 2.4X, 2X, and 1.2X and provides 1.6X, 1.1X, and 1.04X average throughput improvement over MySQL for

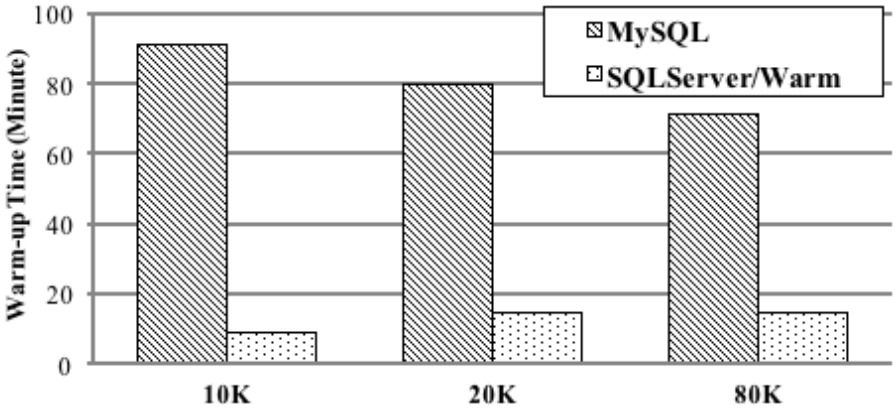


Figure 4.9: Warm-up time for the cloud VHD scenario (a).

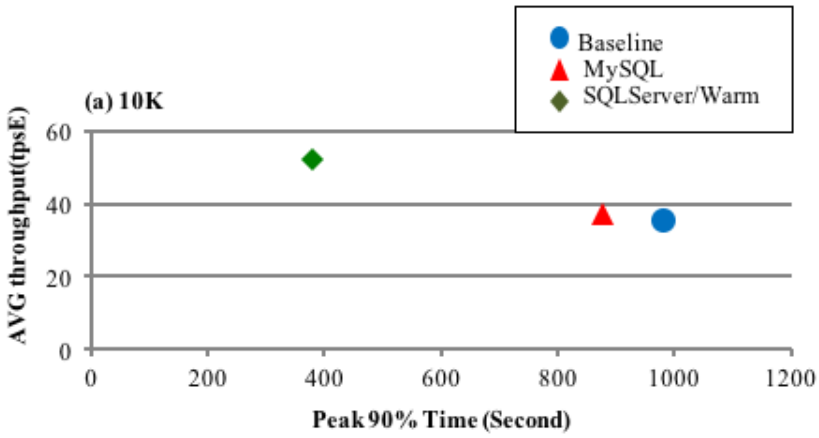
the 10K, 20K, and 80K customer databases, respectively. The reason for this improvement is because SQLServer/Warm not only warms up the buffer pool faster, but also applies the LRU-shifting technique to hold hot pages longer and the Piggybacking technique to warm up the buffer pool with user queries.

4.5.3 HDD Evaluation

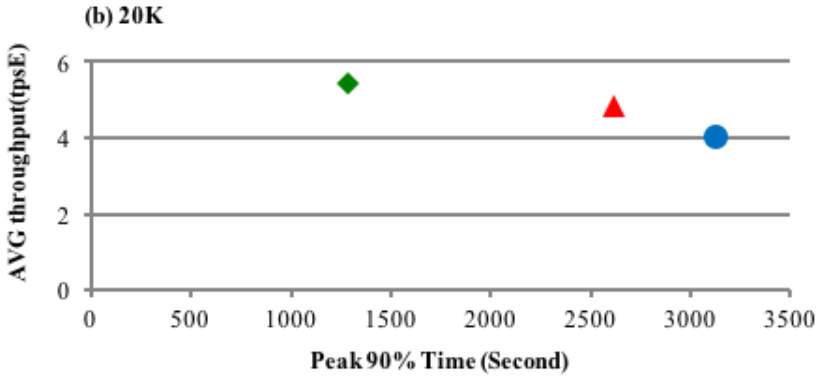
In this section, we compare SQLServer/Warm to the baseline and MySQL on the local HDD I/O subsystem with two different scenarios (see Figure 4.2). For all three datasets (10K, 20K, and 80K customer databases), the Warm-up I/O Planner (see Section 4.4.2.1) returns 512KB (64 pages) as the best I/O request size.

4.5.3.1 Scenario (a)

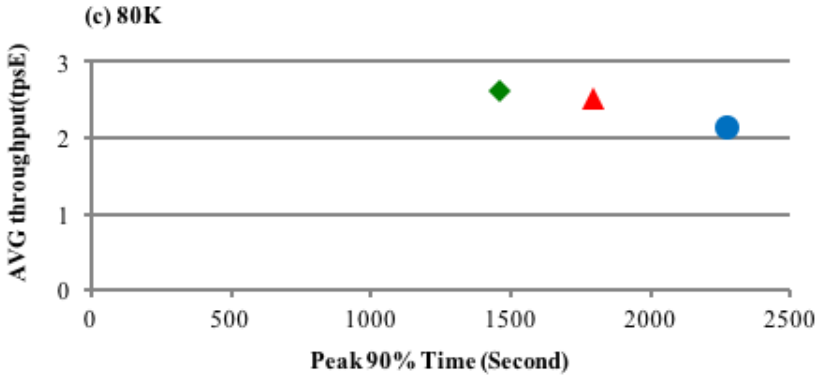
Similar to the results on the cloud VHD evaluation, the peak performance is immediately reached with MySQL and SQLServer/Warm when the system restarts (see Figure 4.11). However, on the HDD I/O subsystem environment, SQLServer/Warm requires more ramp-up time than MySQL for 10K and 20K databases since MySQL takes advantage of the fast sequential read speed of HDDs. In other words, MySQL sorts the hot pages in an ascending order based on the page ID and reads the hot pages in a sequential manner, while SQLServer/Warm orders the hot pages based



(a)



(b)



(c)

Figure 4.10: Peak 90% time and AVG throughput for the cloud VHD Scenario (b).

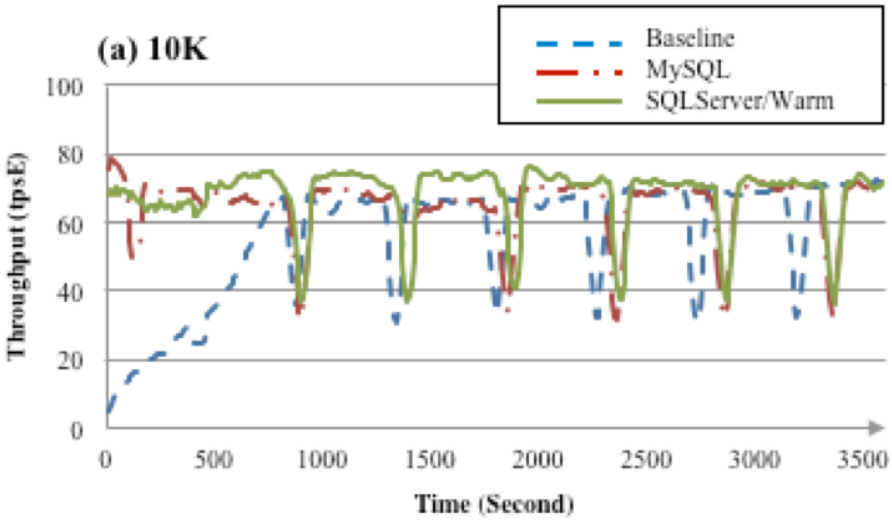


Figure 4.11: Performance trend of the HDD scenario (a) with 10K customer database.

on the hotness defined in Section 4.4.2.3, and reads the hot pages in a random manner. Interestingly, in the case of the 80K database, SQLServer/Warm reduces the warm-up time by 2.7X over MySQL.

4.5.3.2 Scenario (b)

In our experiments, the Break-Even Hot Page Density (see Section 4.4.2.2) for the HDD I/O subsystem 1) is 186KB (∼23 pages), and the AVG Hot Page Densities for 10K, 20K, and 80K customer databases are 41 pages, 24 pages, and 16 pages, respectively. Therefore, the 10K and the 20K cases guarantee a faster warm-up speed over the baseline while the 80K case does not. Figure 4.13 shows the speedup with the 10K and the 20K customer databases over the baseline and MySQL methods. However, SQLServer/Warm provides almost the same performance as the baseline with the 80K customer database. In this case, the Warm-up Estimator returns False, thus the BSL does not issue any I/Os for preloading hot pages. Figure 4.13 shows a point that is marked within a dotted circle. This point shows the result when BSL issues I/O requests for preloading hot pages with the user queries, which is worse than the baseline as expected. As shown in Figure 4.13, SQLServer/Warm reduces the peak 90% time by 2.6X and 1.4X over the baseline

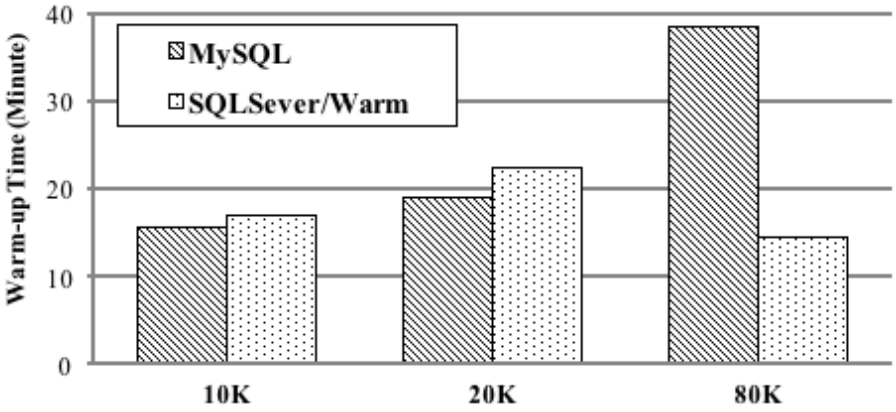
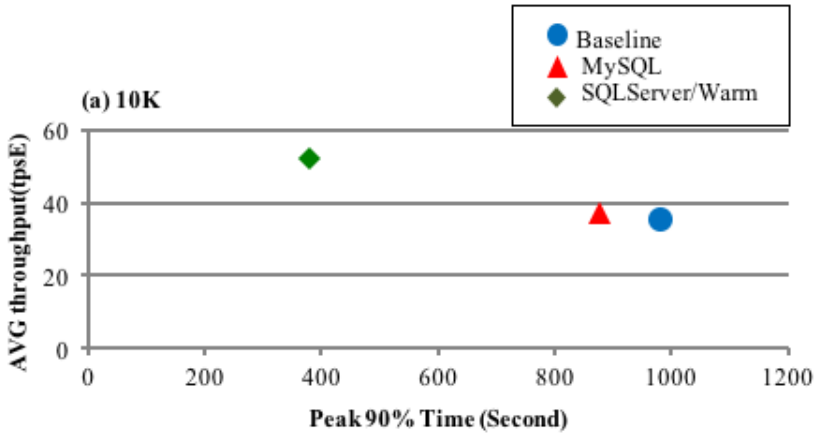


Figure 4.12: Warm-up time for the HDD scenario (a).

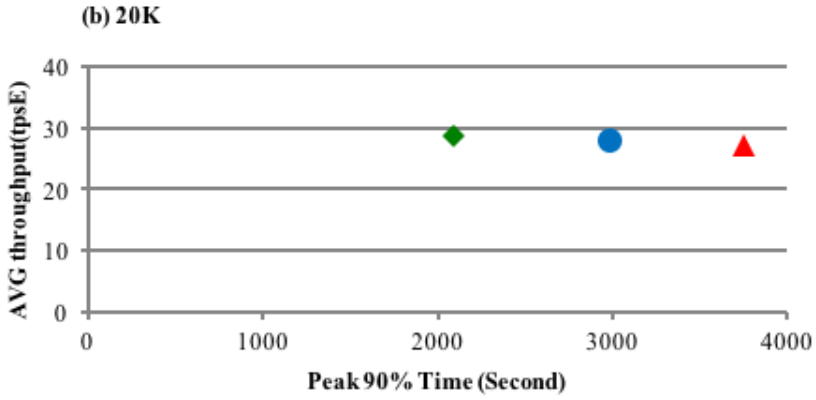
with the 10K and 20K customer databases, respectively. It provides a similar peak 90% time as the baseline with the 80K customer database. The improvement difference (2.6X vs. 3X) with the 10K customer database between HDD and VHD stems from the I/O subsystem characteristics. In other words, HDD's throughput for random reads with 512KB I/Os is only 2.8X bigger than 64KB I/Os, while VHDs throughput for the random read with 512KB I/Os is 8X bigger than 64KB I/Os. SQLServer/Warm saves the peak 90% time by up to 2.3X over MySQL and provides up to 1.4X average throughput improvement over MySQL by applying the LRU-shifting and Piggybacking as well as utilizing faster warm-up speed (512 KB/IO $\times$ 102IOPS vs. 8 KB/IO $\times$ 342IOPS).

4.5.4 SSD Evaluation

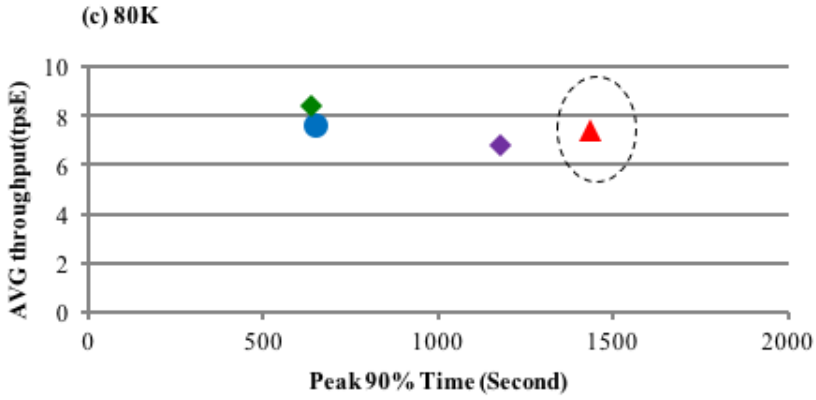
In this section, we evaluate SQLServer/Warm compared to the baseline and MySQL on the local SSD I/O subsystem with two different scenarios (see Figure 4.2). For all three datasets (10K, 20K, and 80K customer databases) the best I/O request size is 8KB, which is because of the fast 8KB random-read speed of SSDs. Therefore, both MySQL and SQLServer/Warm use single page read requests when warming up the buffer pool and require the same elapsed time to warm up the buffer pool in Scenario (a). In Scenario (b), the Break-Even Hot Page Density for SSD is 13KB ($\sim$1.5 pages), and the AVG Hot Page Densities for 10K, 20K, and 80K customer databases are all 1 page, which does not guarantee speedup over the baseline. In fact, in the case of the 80K database



(a)



(b)



(c)

Figure 4.13: Peak 90% time and AVG throughput for the HDD Scenario (b).

Scenario (b), when the BSL issues I/O requests for preloading hot pages with the user queries it shows an 8% degradation in peak 90% time, and an average of 2.3% reduction in throughput. However, based on the value (False) of the Estimator, the BSL does not issue I/O requests, thus SQLServer/Warm shows the almost same performance as the baseline.

4.5.5 Discussion

4.5.5.1 The impact of LRU-shifting and Piggybacking

Figure 4.14 shows the experiments with and without LRU-shifting and Piggybacking. In both experiments (the cloud VHD and the HDD settings with 10K customer database), SQLServer/Warm is worse than the baseline without the LRU-shifting and Piggybacking techniques. Between these two techniques, the impact of using the LRU-shifting is greater than that of Piggybacking based on our experimental results.

4.5.5.2 Warm-up Planner Overhead

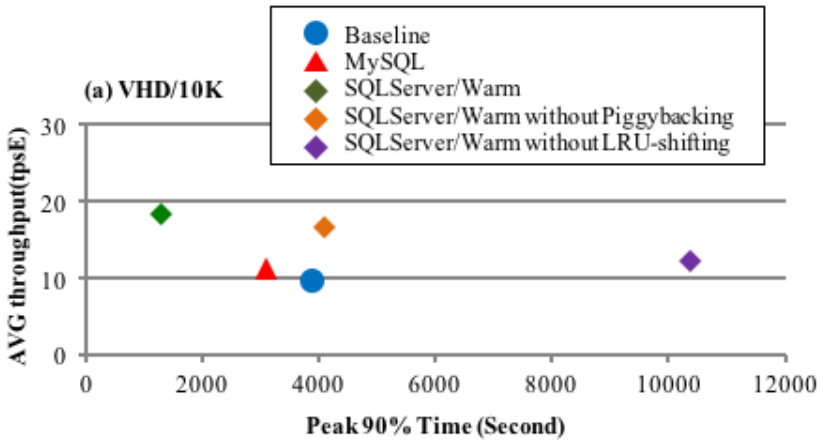
The sample result shown in the Table 4.6 demonstrates that the overhead associated with the Warm-up Planner is low. This is because the Warm-up Planner and the recovery process run in parallel.

4.5.5.3 Sustained Peak Performance

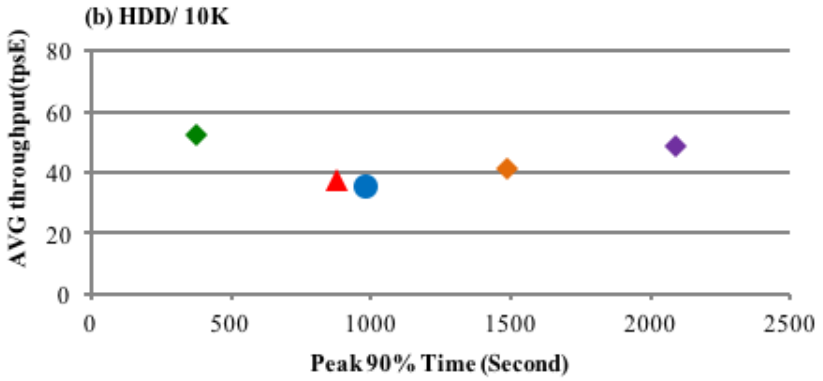
We omitted the sustained peak performance of the three cases (i.e., Baseline, MySQL and SQLServer/Warm) because only a small performance difference was observed as expected. The sustained peak performances of MySQL and SQLServer/Warm were within 6% of the peak performance of the baseline in all experiments.

4.5.5.4 Failure Restart

In the failure restart scenario, since MySQL does not have the buffer pool state metadata, it shows the same performance as the baseline. Therefore, we omit the experimental results with failure restart, which is just simple comparison between SQLServer/Warm and the baseline.



(a)



(b)

Figure 4.14: Impact of LRU-shifting and Piggybacking.

Table 4.5: Implementation details about three buffer pool warm-up mechanisms.

	Baseline	MySQL	SQLServer/Warm
Recording buffer pool state	×	✓ At clean shutdown	✓ At regular check-pointing (BSR)
Background process to load hot pages	×	✓ Sequential read	✓ BSL
User Query with page expansion until the buffer pool is filled	✓	✓	✓

Table 4.6: Recovery time with and without the warm-up planner on restart after a clean shutdown event with a 48GB buffer pool and an 80K customer TPC-E run in cloud VHDs.

Recovery without the Warm-up Planner	71.43 seconds
Recovery with the Warm-up Planner	72.82 seconds

4.6 Related Work

Buffer pool preloading in MySQL [8] is similar to our framework in that it captures the meta-data of the buffer pool pages and preloads the pre-captured buffer pool pages when the system restarts. However, our framework addresses drawbacks in that approach such as long warm-up time, possible early hot page evictions, user query performance saturation, and failure restarts. Previous work has introduced techniques (e.g., [22], [29]) to reduce the ramp-up time by restarting from SSDs. Other relevant techniques include Windows SuperFetch [47] and Bonfire [50], which warm up the cache to reduce system boot time and application launch time. Our methods go beyond these to take a more holistic approach to faster restart and efficiently servicing of incoming user queries while the restart process is in progress.

4.7 Conclusion

Database servers restart for many reasons. Such restarts can result in a dramatic fluctuation in performance until the buffer pool is warmed up. This is a serious issue in settings where one has to start servicing incoming queries as quickly as possible. In this chapter, we proposed a set of mechanism to improve the behavior of database servers/services when restarting. We have compared our framework to other existing approaches (SQL Server's aggressive page expansion and MySQL's buffer pool preloading), and conducted an evaluation of these approaches, with a variety of I/O subsystem settings (cloud VHDs, HDDs, and SSDs) using an OLTP workload (TPC-E). Our empirical evaluation demonstrates that our new mechanism reduces the time to regain peak performance by a factor of 2X or more with cloud VHDs and local HDDs.

Chapter 5

Conclusions and Future Work

In this thesis, we have suggested alternative solutions to achieve high performance data processing by saving the expensive data movement cost in a data processing platform. We have integrated those solutions into existing database systems.

5.1 Conclusions

5.1.1 Moving Computation Closer to the Storage

Modern SSDs pack CPU processing and DRAM storage components inside the SSD to carry out the routine functions (such as managing the FTL logic) for the SSD. Thus, there is a small programmable computer inside a SSD device, presenting an interesting opportunity to move computation closer to the storage. The I/O bus standards (such as SAS, SATA, and PCIe) evolve slower than the speed of the internal network that is used inside the SSD. Without mechanisms to push computation inside the flash SSD, we are essentially doomed to be "drinking from a narrow straw" when using SSDs for data intensive workloads.

In Chapter 2 and Chapter 3, we explored how relational and non-relational database systems can exploit Smart SSDs. We have built prototypes of SQL Server and Hadoop that selectively push computation (i.e., relational query operations and Map tasks) to Smart SSDs. We also present results that show how using the Smart SSD in this way improves not just the performance, but also reduces the energy that is required for data processing. As energy consumption is a critical factor for database appliance and cloud services, using Smart SSDs also provides an interesting opportunity to design energy-efficient data processing systems.

5.1.2 Aggressive Buffer Pool Warmup on Restart

Database servers restart for many reasons. Such restarts can result in a dramatic fluctuation in performance until the buffer pool is warmed up. This is a serious issue in settings where one has to start servicing incoming queries as quickly as possible. In this chapter, we proposed a set of mechanism to improve the behavior of database servers/services when restarting. We have compared our framework to other existing approaches (SQL Server's aggressive page expansion and MySQL's buffer pool preloading), and conducted an evaluation of these approaches, with a variety of I/O subsystem settings (cloud VHDs, HDDs, and SSDs) using an OLTP workload (TPC-E). Our empirical evaluation demonstrates that our new mechanism reduces the time to regain peak performance by a factor of 2X or more with cloud VHDs and local HDDs.

5.2 Future Work

There are still many interesting open directions for future work. In Chapter 2, we explored and quantified the potential advantages of using Smart SSDs for a relational database management system. Thus, an interesting direction for this part of the thesis is to examine the impact of running concurrent queries inside the Smart SSD. Furthermore, studies on appropriate designs for a query optimizer, a transaction manager, and a buffer pool manager inside the Smart SSD would be an interesting direction for future work.

Using Smart SSDs in a distributed data management system (presented in Chapter 3) requires extension of our cost model to include data movement cost between Smart SSDs. Quantifying the benefits of using Smart SSDs from the overall price/performance perspective would be another interesting future work in this chapter.

Finally, the work in Chapter 4 was evaluated using a standalone SQL Server instance. However, integrating our new framework into real cloud database systems such as Azure DB is an interesting direction for future work. In cloud settings, system restarts are more frequent due to operational considerations, such as the need to move a database service from one machine to another. Such

restarts result in performance degradation as the restarted service starts with a cold buffer pool. Therefore, our new framework would be helpful to mitigate the performance degradation at restart.

LIST OF REFERENCES

[1] Apache hadoop. http://hadoop.apache.org/.

[2] Apache hadoop 3.0.0. http://aajisaka.github.io/hadoop-project/.

[3] Arm developer suite. http://infocenter.arm.com/help/topic/com.arm.doc.dui0068b/DUI0060.pdf.

[4] Ibm xiv storage system. http://www.ibm.com/systems/storage/disk/xiv/index.html.

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

[6] Lsi, sas 9211-4i hba. http://www.lsi.com/channel/products/storagecomponents/Pages/LSISAS9211-4i.aspx.

[7] Microsoft sql server 2012. http://www.microsoft.com/sqlserver.

[8] Mysql preloading the buffer pool. https://dev.mysql.com/doc/refman/5.7/en/innodb-preload-buffer-pool.html.

[9] Power and temperature measurement setup guide. http://spec.org/power/docs/SPEC-Power_Measurement_Setup_Guide.pdf.

[10] Purdue puma benchmarks. https://engineering.purdue.edu/ puma/datasets.htm.

[11] tempdb. https://msdn.microsoft.com/en-us/library/ms190768.aspx.

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

[13] title =.

[14] Tmicrosoft sql server 2014. http://www.microsoft.com/en-us/servercloud/products/sql-server/.

[15] Tpc benchmark e (tpc-e). http://www.tpc.org/tpce.

[16] Tpc benchmark h (tpc-h). http://www.tpc.org/tpch.

[17] Trace32, lauterbach development tools. http://www.lauterbach.com.

[18] Windows azure. https://msdn.microsoft.com/enus/library/azure/dn197896.aspx.

[19] A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server. Technical report, Oracle Corp, 2012.

[20] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: An optimized index structure for flash devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

[21] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001.

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

[23] Simona Boboila, Youngjae Kim, Sudharshan S Vazhkudai, Peter Desnoyers, and Galen M Shipman. Active flash: Out-of-core data analytics on flash storage. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–12. IEEE, 2012.

[24] Mustafa Canim, George A Mihaila, Bishwaranjan Bhattacharjee, Kenneth A Ross, and Christian A Lang. Ssd bufferpool extensions for database systems. *Proceedings of the VLDB Endowment*, 3(1-2):1435–1446, 2010.

[25] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102. ACM, 2013.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[27] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[28] Jaeyoung Do and Jignesh M Patel. Join processing for flash ssds: remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 1–8. ACM, 2009.

[29] Jaeyoung Do, Donghui Zhang, Jignesh M Patel, and David J DeWitt. Fast peak-to-peak behavior with ssd buffer pool. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 1129–1140. IEEE, 2013.

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

[31] P. Francisco. The netezza data appliance architecture: A platform for high performance data warehousing and analytics. In *IBM Redbook*, 2011.

[32] Jim Gray. Tape is dead, disk is tape, flash is disk, ram locality is king. *Gong Show Presentation at CIDR*, 15:231–242, 2007.

[33] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, September 1998.

[34] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory ssds. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, 2011.

[35] Ioannis Koltsidas and Stratis D Viglas. Data management over flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1209–1212. ACM, 2011.

[36] Ioannis Koltsidas and Stratis D Viglas. Designing a flash-aware two-level cache. In *Advances in Databases and Information Systems*, pages 153–169. Springer, 2011.

[37] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 55–66. ACM, 2007.

[38] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.

[39] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[40] Rene Mueller and Jens Teubner. FPGA: what's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 999–1004. ACM, 2009.

[41] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

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

[43] Kwanghyun Park, Jaeyoung Do, Nikhil Teletia, and Jignesh M Patel. Aggressive buffer pool warm-up after restart in sql server. In *Proceedings of the workshop on Cloud Data Management*. IEEE, 2016.

[44] Kwanghyun Park, Yang-Suk Kee, Jignesh M Patel, Jaeyoung Do, Chanik Park, and David J Dewitt. Query processing on smart ssds. *IEEE Data Eng. Bull.*, 37(2):19–26, 2014.

[45] E. Riedel, C. Faloutsos, and D. F. Nagle. Active disk architecture for databases. Technical report, CMU, 2000.

[46] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[47] Mark Russinovich. Inside the windows vista kernel: Part 3. *Microsoft TechNet Magazine*, 2007.

[48] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A Shah, Janet L Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 59–72. ACM, 2009.

[49] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An efficient b-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):19, 2007.

[50] Yiying Zhang, Gokul Soundararajan, Mark W Storer, Lakshmi N Bairavasundaram, Sethuraman Subbiah, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Warming up storage-level caches with bonfire. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 59–72, 2013.