A Case for Flash Memory SSD in Enterprise Database Applications

Sang-Won Lee[†] Bongki Moon[‡] Chanik Park[§] Jae-Myung Kim[¶] Sang-Woo Kim[†]

[†]School of Information & Communications Engr.
Sungkyunkwan University
Suwon 440-746, Korea
{wonlee,swkim}@ece.skku.ac.kr

[‡]Department of Computer Science
University of Arizona
Tucson, AZ 85721, U.S.A.
bkmoon@cs.arizona.edu

[§]Samsung Electronics Co., Ltd.
San #16 Banwol-Ri
Hwasung-City 445-701, Korea
ci.park@samsung.com

[¶]Altibase Corp.
182-13, Guro-dong, Guro-Gu
Seoul, 152-790, Korea
jmkim@altibase.com

ABSTRACT

Due to its superiority such as low access latency, low energy consumption, light weight, and shock resistance, the success of flash memory as a storage alternative for mobile computing devices has been steadily expanded into personal computer and enterprise server markets with ever increasing capacity of its storage. However, since flash memory exhibits poor performance for small-to-moderate sized writes requested in a random order, existing database systems may not be able to take full advantage of flash memory without elaborate flash-aware data structures and algorithms. The objective of this work is to understand the applicability and potential impact that flash memory SSD (Solid State Drive) has for certain type of storage spaces of a database server where sequential writes and random reads are prevalent. We show empirically that up to more than an order of magnitude improvement can be achieved in transaction processing by replacing magnetic disk with flash memory SSD for transaction log, rollback segments, and temporary table spaces.

Categories and Subject Descriptors

H. Information Systems [**H.2 DATABASE MANAGEMENT**]: H.2.2 Physical Design

General Terms

Design, Algorithms, Performance, Reliability

*This work was partly supported by the IT R&D program of MIC/IITA [2006-S-040-01] and MIC, Korea under ITRC IITA-2008-(C1090-0801-0046). The authors assume all responsibility for the contents of the paper.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

Keywords

Flash-Memory Database Server, Flash-Memory SSD

1. INTRODUCTION

Due to its superiority such as low access latency, low energy consumption, light weight, and shock resistance, the success of flash memory as a storage alternative for mobile computing devices has been steadily expanded into personal computer and enterprise server markets with ever increasing capacity of its storage. As it has been witnessed in the past several years, two-fold annual increase in the density of NAND flash memory is expected to continue until year 2012 [11]. Flash-based storage devices are now considered to have tremendous potential as a new storage medium that can replace magnetic disk and achieve much higher performance for enterprise database servers [10].

The trend in market is also very clear. Computer hardware manufacturers have already launched new lines of mobile personal computers that did away with disk drives altogether, replacing them with flash memory SSD (Solid State Drive). Storage system vendors have started lining up their flash-based solutions in Terabyte-scale targeting large-scale database servers as one of the main applications.

Adoption of a new technology, however, is often deterred by lack of in-depth analysis on its applicability and cost-effectiveness, and is even considered risky when it comes to mission critical applications. The objective of this work is to evaluate flash memory SSD as stable storage for database workloads and identify the areas where flash memory SSD can be best utilized, thereby accelerating its adoption as an alternative to magnetic disk and maximizing the benefit from this new technology.

Most of the contemporary database systems are configured to have separate storage spaces for database tables and indexes, log data and temporary data. Whenever a transaction updates a data object, its log record is created and stored in stable storage for recoverability and durability of the transaction execution. Temporary table space stores

temporary data required for performing operations such as sorts or joins. If multiversion read consistency is supported, another separate storage area called rollback segments is created to store previous versions of data objects.

For the purpose of performance tuning as well as recoverability, these distinct storage spaces are often created on physically separate storage devices, so that I/O throughput can increase, and I/O bottlenecks can be detected and addressed with more ease. While it is commonly known that accessing data stored in secondary storage is the main source of bottlenecks in database processing, high throughput of a database system cannot be achieved by addressing the bottlenecks only in spaces for tables and indexes but also in spaces for log, temporary and rollback data.

Recent studies on database availability and architecture report that writing log records to stable storage is almost guaranteed to be a significant performance bottleneck [13, 21]. In on-line transaction processing (OLTP) applications, for example, when a transaction commits, all the log records created by the transaction have to be force-written to stable storage. If a large number of concurrent transactions commit at a rapid rate, the log tail will be requested to be flushed to disk very often. This will then lengthen the average wait time of committing transactions and delay the release of locks further, and eventually increase the overall runtime overhead substantially.

Accessing data stored in temporary table spaces and rollback segments also takes up a significant portion of total I/O activities. For example, queries performing a table scan, join, sort or hash operation are very common in a data warehousing application, and processing those queries (except simple table scans) will require a potentially large amount of intermediate data to be written to and read from temporary table spaces. Thus, to maximize the throughput of a database system, it is critical to speed up accessing data stored in those areas as well as in the data space for tables and indexes.

Previous work has reported that flash memory exhibits poor performance for small-to-moderate sized writes requested in a random order [2] and the best attainable performance may not be obtained from database servers without elaborate flash-aware data structures and algorithms [14]. In this paper, in contrast, we demonstrate that flash memory SSD can help improve the performance of transaction processing significantly, particularly as a storage alternative for transaction log, rollback segments and temporary table spaces. To accomplish this, we trace quite distinct data access patterns observed from these three different types of data spaces, and analyze how magnetic disk and flash memory SSD devices handle such I/O requests, and show how the overall performance of transaction processing is affected by them.

While the previous work on *in-page logging* is targeted at regular table spaces for database tables and indexes where small random writes are dominant [14], the objective of this work is to understand the applicability and potential impact that flash memory SSD has for the other data spaces where sequential writes and random reads are prevalent. The key contributions of this work are summarized as follows.

- Based on a detailed analysis of data accesses that are traced from a commercial database server, this paper provides an understanding of I/O behaviors that are dominant in transaction log, rollback segments, and temporary table spaces. It also shows that this I/O pattern is a good match for the dual-channel, superblock design of flash memory SSD as well as the characteristics of flash memory itself.

- This paper presents a quantitative and comparative analysis of magnetic disk and flash memory SSD with respect to performance impacts they have on transactional database workloads. We observed more than an order of magnitude improvement in transaction throughput and response time by replacing magnetic disk with flash memory SSD as storage media for transaction log or rollback segments. In addition, more than a factor of two improvement in response time was observed in processing a sort-merge or hash join query by adopting flash memory SSD instead of magnetic disk for temporary table spaces.

- The empirical study carried out in this paper demonstrates that low latency of flash memory SSD can alleviate drastically the log bottleneck at commit time and the problem of increased random reads for multiversion read consistency. With flash memory SSD, I/O processing speed may no longer be as serious a bottleneck as it used be, and the overall performance of query processing can be much less sensitive to tuning parameters such as the unit size of physical I/O. The superior performance of flash memory SSD demonstrated in this work will help accelerate adoption of flash memory SSD for database applications in the enterprise market, and help us revisit requirements of database design and tuning guidelines for database servers.

The rest of this paper is organized as follows. Section 2 presents a few key features and architecture of Samsung flash memory SSD, and discusses its performance characteristics with respect to transactional database workloads. Section 3 describes the experimental settings that will be used in the following sections. In Section 4, we analyze the performance gain that can be obtained by adopting flash memory SSD as stable storage for transaction log. Section 5 analyzes the patterns in which old versions of data objects are written to and read from rollback segments, and shows how flash memory SSD can take advantage of the access patterns to improve access speed for rollback segments and the average response time of transactions. In Section 6, we analyze the I/O patterns of sort-based and hash-based algorithms, and discuss the impact of flash memory SSD on the algorithms. Lastly, Section 7 summarizes the contributions of this paper.

2. DESIGN OF SAMSUNG FLASH SSD

The flash memory SSD (Solid State Drive) of Samsung Electronics is a non-volatile storage device based on NAND-type flash memory, which is being marketed as a replacement of traditional hard disk drives for a wide range of computing platforms. In this section, we first briefly summarize the characteristics of flash memory as a storage medium for databases. We then present the architecture and a few key

features of Samsung flash memory SSD, and discuss its performance implications on transactional database workloads.

2.1 Characteristics of Flash Memory

Flash memory is a purely electronic device with no mechanically moving parts like disk arms in a magnetic disk drive. Therefore, flash memory can provide uniform random access speed. Unlike magnetic disks whose seek and rotational delay often becomes the dominant cost of reading or writing a sector, the time to access data in flash memory is almost linearly proportional to the amount of data irrespective of their physical locations in flash memory. The ability of flash memory to quickly perform a sector read or a sector (clean) write located anywhere in flash memory is one of the key characteristics we can take advantage of.

On the other hand, with flash memory, no data item (or a sector containing the data item) can be updated in place just by overwriting it. In order to update an existing data item stored in flash memory, a time-consuming erase operation must be performed before overwriting. The erase operation cannot be performed selectively on a particular data item or sector, but can only be done for an entire block of flash memory called *erase unit* containing the data item, which is much larger (typically 128 KBytes) than a sector. To avoid performance degradation caused by this erase-before-write limitation, some of the data structures and algorithms of existing database systems may well be reconsidered [14].

The read and write speed of flash memory is asymmetric, simply because it takes longer to write (or inject charge into) a cell until reaching a stable status than to read the status from a cell. As will be shown later in this section (Table 1), the sustained speed of read is almost twice faster than that of write. This property of asymmetric speed should also be considered when reviewing existing techniques for database system implementations.

2.2 Architecture and Key Features

High bandwidth is one of the critical requirements for the design of flash memory SSD. The dual-channel architecture, as shown in Figure 1, supports up to 4-way interleaving to hide flash programming latency and to increase bandwidth through parallel read/write operations. An automatic interleaving hardware logic is adopted to maximize the interleaving effect with the minimal firmware intervention [18].

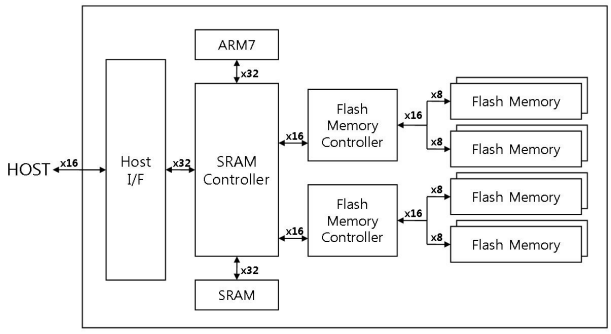


Figure 1: Dual-Channel Architecture of SSD

A firmware layer known as flash translation layer (FTL) [5, 12] is responsible for several essential functions of flash memory SSD such as address mapping and wear leveling. The address mapping scheme is based on super-blocks in order to limit the amount of information required for logical-to-physical address mapping, which grows larger as the capacity of flash memory SSD increases. This super-block scheme also facilitates interleaved accesses of flash memory by striping a super-block of one MBytes across four flash chips. A super-block consists of eight erase units (or large blocks) of 128 KBytes each. Under this super-block scheme, two erase units of a super-block are allocated in the same flash chip.

Though flash memory SSD is a purely electronic device without any moving part, it is not entirely latency free for accessing data. When a read or write request is given from a host system, the I/O command should be interpreted and processed by the SSD controller, referenced logical addresses should be mapped to physical addresses, and if mapping information is altered by a write or merge operation, then the mapping table should be updated in flash memory. With all these overheads added up, the read and write latency observed from the recent SSD products is approximately 0.2 msec and 0.4 msec, respectively.

In order to reduce energy consumption, the one-chip controller uses a small amount of SRAM for program code, data and buffer memory.[1] The flash memory SSD drives can be interfaced with a host system through the IDE standard ATA-5.

2.3 Flash SSD for Database Workload

Typical transactional database workloads like TPC-C exhibit little locality and sequentiality in data accesses, a high percentage of which are synchronous writes (*e.g.*, forced-writes of log records at commit time). Such latency hiding techniques as prefetching and write buffering become less effective for this type of workload, and the performance of transactional database applications tends to be more closely limited by disk latency than disk bandwidth and capacity [24]. Nonetheless, for more than a decade in the past, the latency of disk has improved at a much slower pace than the bandwidth of disk, and the latency-bandwidth imbalance is expected to be even more evident in the future [19].

In this regard, extremely low latency of flash memory SSD lends itself to being a new storage medium that replaces magnetic disk and improves the throughput of transaction processing significantly. Table 1 shows the performance characteristics of some contemporary hard disk and flash memory SSD products. Though the bandwidth of disk is still two to three times higher than that of flash memory SSD, more importantly, the read and write latency of flash memory SSD is smaller than that of disk by more than an order of magnitude.

As is briefly mentioned above, the low latency of flash memory SSD can reduce the average transaction commit time and improve the throughput of transaction processing significantly. If multiversion read consistency is supported, rollback data are typically written to rollback segments sequentially in append-only fashion and read from rollback segments randomly during transaction processing. This pe-

[1]The flash memory SSD drive tested in this paper contains 128 KByte SRAM.

Storage	hard disk†	flash SSD‡
Average Latency	8.33 ms	0.2 ms (read) 0.4 ms (write)
Sustained Transfer Rate	110 MB/sec	56 MB/sec (read) 32 MB/sec (write)

†Disk: Seagate Barracuda 7200.10 ST3250310AS, average latency for seek and rotational delay;
‡SSD: Samsung MCAQE32G8APP-0XA drive with K9WAG08U1A 16 Gbits SLC NAND chips

Table 1: Magnetic disk vs. NAND Flash SSD

culiar I/O pattern is a good match for the characteristics of flash memory itself and the super-block scheme of the Samsung flash memory SSD. External sorting is another operation that can benefit from the low latency of flash memory SSD, because the read pattern of external sorting is quite random during the merge phase in particular.

3. EXPERIMENTAL SETTINGS

Before presenting the results from our workload analysis and performance study in the following sections, we describe the experimental settings briefly in this section.

In most cases, we ran a commercial database server (one of the most recent editions of its product line) on two Linux systems (kernel version 2.6.22), each with a 1.86 GHz Intel Pentium dual-core processor and 2 GB RAM. These two computer systems were identical except that one was equipped with a magnetic disk drive and the other with a flash memory SSD drive instead of the disk drive. The disk drive model was Seagate Barracuda 7200.10 ST3250310AS with 250 GB capacity, 7200 rpm and SATA interface. The flash memory SSD model was Samsung Standard Type MCAQE32G8APP-0XA with 32 GB capacity and 1.8 inch PATA interface, which internally deploys Samsung K9WAG08U1A 16 Gbits SLC NAND flash chips (shown in Figure 2). These storage devices were connected to the computer systems via a SATA or PATA interface.

Figure 2: Samsung NAND Flash SSD

When either magnetic disk or flash memory SSD was used as stable storage for transaction log, rollback segments, or temporary table spaces, it was bound as a raw device in order to minimize interference from data caching by the operating system. This is a common way of binding storage devices adopted by most commercial database servers with their own caching scheme. In all the experiments, database tables were cached in memory so that most of IO activities were confined to transaction log, rollback segments and temporary table spaces.

4. TRANSACTION LOG

When a transaction commits, it appends a commit type log record to the log and force-writes the log tail to stable storage up to and including the commit record. Even if a no-force buffer management policy is being used, it is required to force-write all the log records kept in the log tail to ensure the durability of transactions [22].

As the speed of processors becomes faster and the memory capacity increases, the commit time delay due to force-writes increasingly becomes a serious bottleneck to achieving high performance of transaction processing [21]. The response time $T_{response}$ of a transaction can be modeled as a sum of CPU time T_{cpu}, read time T_{read}, write time T_{write} and commit time T_{commit}. T_{cpu} is typically much smaller than IO time. Even T_{read} and T_{write} become almost negligible with a large capacity buffer cache and can be hidden by asynchronous write operations. On the other hand, commit time T_{commit} still remains to be a significant overhead, because every committing transaction has to wait until all of its log records are force-written to log, which in turn cannot be done until forced-write operations requested by other transactions earlier are completed. Therefore, the amount of commit-time delay tends to increase as the number of concurrent transactions increases, and is typically no less than a few milliseconds.

Group commit may be used to alleviate the log bottleneck [4]. Instead of committing each transaction as it finishes, transactions are committed in batches when enough logs are accumulated in the log tail. Though this group commit approach can significantly improve the throughput of transaction processing, it does not improve the response time of individual transactions and does not remove the commit time log bottleneck altogether.

Log records are always appended to the end of log. If a separate storage device is dedicated to transaction log, which is commonly done in practice for performance and recoverability purposes, this sequential pattern of write operations favors not only hard disk but also flash memory SSD. With no seek delay due to sequential accesses, the write latency of disk is reduced to only half a revolution of disk spindle on average, which is equivalent to approximately 4.17 msec for disk drives with 7200 rpm rotational speed.

In the case of flash memory SSD, however, the write latency is much lower at about 0.4 msec, because flash memory SSD has no mechanical latency but only a little overhead from the controller as described in Section 2.3. Even the *no in-place update* limitation of flash memory has no negative impact on the write bandwidth in this case, because log records being written to flash memory sequentially do not cause expensive merge or erase operations as long as clean flash blocks (or erase units) are available. Coupled with the low write latency of flash memory, the use of flash memory SSD as a dedicated storage device for transaction log can reduce the commit time delay considerably.

In the rest of this section, we analyze the performance gain that can be obtained by adopting flash memory SSD as stable storage for transaction log. The empirical results from flash memory SSD drives are compared with those from magnetic disk drives.

4.1 Simple SQL Transactions

To analyze the commit time performance of hard disk and flash memory SSD drives, we first ran a simple embedded SQL program on a commercial database server, which ran on two identical Linux systems except that one was equipped with a magnetic disk drive and the other with a flash memory SSD drive instead of the disk drive. This embedded SQL program is multi-threaded and simulates concurrent transactions. Each thread updates a single record and commits, and repeats this cycle of update and commit continuously. In order to minimize the wait time for database table updates and increase the frequency of commit time forced-writes, the entire table data were cached in memory. Consequently, the runtime of a transaction excluding the commit time (*i.e.*, $T_{cpu} + T_{read} + T_{write}$) was no more than a few dozens of microseconds in the experiment. Table 2 shows the throughput of the embedded SQL program in terms of transactions-per-seconds (TPS).

no. of concurrent transactions	hard disk		flash SSD	
	TPS	%CPU	TPS	%CPU
4	178	2.5	2222	28
8	358	4.5	4050	47
16	711	8.5	6274	77
32	1403	20	5953	84
64	2737	38	5701	84

Table 2: Commit-time performance of an embedded SQL program measured in transactions-in-seconds (TPS) and CPU utilization

Regarding the commit time activities, a transaction can be in one of the three distinct states. Namely, a transaction (1) is still active and has not requested to commit, (2) has already requested to commit but is waiting for other transactions to complete forced-writes of their log records, or (3) has requested to commit and is currently force-writing its own log records to stable storage.

When a hard disk drive was used as stable storage, the average wait time of a transaction was elongated due to the longer latency of disk writes, which resulted in an increased number of transactions that were kept in a state of the second or third category. This is why the transaction throughput and CPU utilization were both low, as shown in the second and third columns of Table 2.

On the other hand, when a flash memory SSD drive was used instead of a hard disk drive, much higher transaction throughput and CPU utilization were observed, as shown in the fourth and fifth columns of Table 2. With a much shorter write latency of flash memory SSD, the average wait time of a transaction was shortened, and a relatively large number of transactions were actively utilizing CPU, which in turn resulted in higher transaction throughput. Note that the CPU utilization was saturated when the number of concurrent transactions was high in the case of flash memory SSD, and no further improvement in transaction throughput was observed when the number of concurrent transactions was increased from 32 to 64, indicating that CPU was a limiting factor rather than I/O.

4.2 TPC-B Benchmark Performance

In order to evaluate the performance of flash memory SSD as a storage medium for transaction log in a more harsh environment, we ran a commercial database server with TPC-B workloads created by a workload generation tool. Although it is obsolete, the TPC-B benchmark was chosen because it is designed to be a stress test on different subsystems of a database server and its transaction commit rate is higher than that of TPC-C benchmark [3]. We used this benchmark to stress-test the log storage part of the commercial database server by executing a large number of small transactions causing significant forced-write activities.

In this benchmark test, the number of concurrent simulated users was set to 20, and the size of database and the size of database buffer cache of the server were set to 450 MBytes and 500 MBytes, respectively. Note that this setting allows the database server to cache the entire database in memory, such that the cost of reading and writing data pages is eliminated and the cost of forced writing log records remains dominant on the critical path in the overall performance. When either a hard disk or flash memory SSD drive was used as stable storage for transaction log, it was bound as a raw device. Log records were force-written to the stable storage in a single or multiple sectors (of 512 bytes) at a time.

Table 3 summarizes the results from the benchmark test measured in terms of transactions-per-seconds (TPS) and CPU utilization as well as the average size of a single log write and the average time taken to process a single log write. Since multiple transactions could commit together as a group (by a group commit mechanism), the frequency of log writes was much lower than the number of transactions processed per second. Again, due to the group commit mechanism, the average size of a single log write was slightly different between the two storage media.

	hard disk	flash SSD
Transactions/sec	864	3045
CPU utilization (%)	20	65
Log write size (sectors)	32	30
Log write time (msec)	8.1	1.3

Table 3: Commit-time performance from TPC-B benchmark (with 20 simulated users)

The overall transaction throughput was improved by a factor of 3.5 by using a flash memory SSD drive instead of a hard disk drive as stable storage for transaction log. Evidently the main factor responsible for this improvement was the considerably lower log write time (1.3 msec on average) of flash memory SSD, compared with about 6 times longer log write time of disk. With a much reduced commit time delay by flash memory SSD, the average response time of a transaction was also reduced considerably. This allowed

transactions to release resources such as locks and memory quickly, which in turn helped transactions avoid waiting on locks held by other transactions and increased the utilization of CPU. With flash memory SSD as a logging storage device, the bottleneck of transaction processing now appears to be CPU rather than I/O subsystem.

4.3 I/O-Bound vs. CPU-Bound

In the previous sections, we have suggested that the bottleneck of transaction processing might be shifted from I/O to CPU if flash memory SSD replaced hard disk as a logging storage device. In order to put this proposition to the test, we carried out further performance evaluation with the TPC-B benchmark workload.

First, we repeated the same benchmark test as the one depicted in Section 4.2 but with a varying number of simulated users. The two curves denoted by `Disk-Dual` and `SSD-Dual` in Figure 3 represent the transaction throughput observed when a hard disk drive or a flash memory SSD drive was used as a logging storage device, respectively. Not surprisingly, this result matches the one shown in Table 3, and shows the trend more clearly.

In the case of flash memory SSD, as the number of concurrent transactions increased, transaction throughput increased quickly and was saturated at about 3000 transactions per second without improving beyond this level. As will be discussed further in the following, we believe this was because the processing power of CPU could not keep up with a transaction arrival rate any higher than that. In the case of disk, on the other hand, transaction throughput increased slowly but steadily in proportion to the number of concurrent transactions until it reached the same saturation level. This clearly indicates that CPU was not a limiting factor in this case until the saturation level was reached.

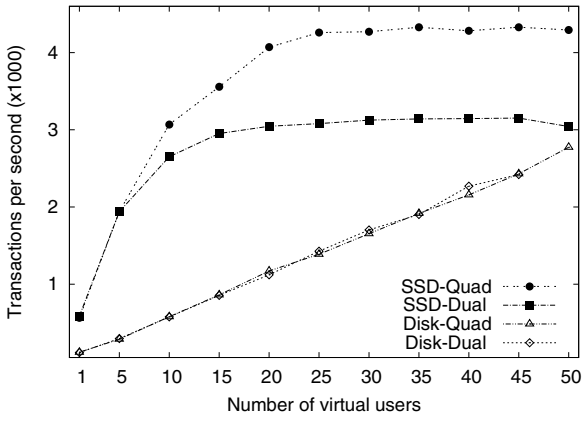


Figure 3: Commit-time performance of TPC-B benchmark : I/O-bound vs. CPU-bound

Next, we repeated the same benchmark test again with a more powerful CPU – 2.4 GHz Intel Pentium quad-core processor – instead of a 1.86 GHz dual-core processor in the same setting. The two curves denoted by `Disk-Quad` and `SSD-Quad` in Figure 3 represent the transaction throughput observed when the quad-core processor was used.

In the case of disk, the trend in transaction throughput remained almost identical to the one previously observed when a dual-core processor was used. In the case of flash memory SSD, the trend of `SSD-Quad` was also similar to that of `SSD-Dual`, except that the saturation level was considerably higher at approximately 4300 transactions per second.

The results from these two benchmark tests speak for themselves that the processing speed of CPU was a bottleneck in transaction throughput in case of flash memory SSD, while it was not in the case of disk.

5. MVCC ROLLBACK SEGMENT

Multiversion concurrency control (MVCC) has been adopted by some of the commercial and open source database systems (*e.g.*, Oracle, PostgreSQL, SQL Server 2005) as an alternative to the traditional concurrency control mechanism based on locks. Since read consistency is supported by providing multiple versions of a data object without any lock, MVCC is intrinsically non-blocking and can arguably minimize performance penalty on concurrent update activities of transactions. Another advantage of multiversion concurrency control is that it naturally supports snapshot isolation [1] and time travel queries [15, 17]. [2]

To support multiversion read consistency, however, when a data object is updated by a transaction, the original data value has to be recorded in an area known as *rollback segments*. The rollback segments are typically set aside in stable storage to store old images of data objects, and should not be confused with undo log, because the rollback segments are not for recovery but for concurrent execution of transactions. Thus, under multiversion concurrency control, updating a data object requires writing its before image to a rollback segment in addition to writing undo and redo log records for the change.

Similarly, reading a data object can be somewhat costlier under the multiversion concurrency control. When a transaction reads a data object, it needs to check whether the data object has been updated by other transactions, and needs to fetch an old version from a rollback segment if necessary. The cost of this read operation may not be trivial, if the data object has been updated many times and fetching its particular version requires search through a long list of versions of the data object. Thus, it is essential to provide fast access to data in rollback segments so that the performance of database servers supporting MVCC are not hindered by increased disk I/O activities [16].

In this section, we analyze the patterns in which old versions of data objects are written to and read from rollback segments, and show how flash memory SSD can take advantage of the access patterns to improve access speed for rollback segments and the average response time of transactions.

5.1 Understanding the MVCC Write

When a transaction updates tuples, it stores the before images of the updated tuples in a block within a rollback

[2]As opposed to the ANSI SQL-92 isolation levels, the snapshot isolation level exhibits none of the anomalies that the SQL-92 isolation levels prohibit. Time travel queries allow you to query a database as of a certain time in the past.

segment or an extent of a rollback segment. When a transaction is created, it is assigned to a particular rollback segment, and the transaction writes old images of data objects sequentially into the rollback segment. In the case of a commercial database server we tested, it started with a default number of rollback segments and added more rollback segments as the number of concurrent transactions increased.

Figure 4 shows the pattern of writes we observed in the rollback segments of a commercial database server processing a TPC-C workload. The x and y axes in the figure represent the timestamps of write requests and the logical sector addresses directed by the requests. The TPC-C workload was created for a database of 120 MBytes. The rollback segments were created in a separate disk drive bound as a raw device. This disk drive stored nothing but the rollback segments. While Figure 4(a) shows the macroscopic view of the write pattern represented in a time-address space, Figure 4(b) shows more detailed view of the write pattern in a much smaller time-address region.

The multiple slanted line segments in Figure 4(b) clearly demonstrate that each transaction writes sequentially into its own rollback segment in the append-only fashion, and concurrent transactions generate multiple streams of such write traffic in parallel. Each line segment spanned a separate logical address space that was approximately equivalent to 2,000 sectors or one MBytes. This is because a new extent of one MBytes was allocated, every time a rollback segment ran out of the space in the current extent. The length of a line segment projected on the horizontal (time) dimension varied slightly depending on how quickly transactions consumed the current extent of their rollback segment.

The salient point of this observation is that consecutive write requests made to rollback segments were almost always apart by approximately one MBytes in the logical address space. If a hard disk drive were used as storage for rollback segments, each write request to a rollback segment would very likely have to move the disk arm to a different track. Thus, the cost of recording rollback data for MVCC would be significant due to excessive seek delay of disk.

Flash memory SSD undoubtedly has no such problem as seek delay, because it is a purely electronic device with extremely low latency. Furthermore, since old images of data objects are written to rollback segments in append-only fashion, the *no in-place update* limitation of flash memory has no negative effect on the write performance of flash memory SSD as a storage device for rollback segments. Of course, a potential bottleneck may come up, if no free block (or clean erase unit) is available when a new rollback segment or an extent is to be allocated. Then, a flash block should be reclaimed from obsolete ones, which involves costly erase and merge operations for flash memory. If this reclamation process happens to be on the critical path of transaction execution, it may prolong the response time of a transaction. However, the reclamation process was invoked infrequently only when a new rollback segment or an extent was allocated. Consequently, the cost of reclamation was amortized over many subsequent write operations, affecting the write performance of flash memory SSD only slightly.

Note that there is a separate stream of write requests that appear at the bottom of Figure 4(a). These write requests followed a pattern quite different from the rest of write requests, and were directed to an entirely separate, narrow area in the logical address space. This is where metadata of rollback segments were stored. Since the metadata stayed in the fixed region of the address space, the pattern of writes directed to this area was in-place updates rather than append-only fashion. Due to the *no in-place update* limitation of flash memory, in-place updates of metadata would be costly for flash memory SSD. However, its negative effect was insignificant in the experiment, because the volume of metadata updates was relatively small.

Overall, we did not observe any notable difference between disk and flash memory SSD in terms of write time for rollback segments. In our TPC-C experiment, the average time for writing a block to a rollback segment was 7.1 msec for disk and 6.8 msec for flash memory SSD.

5.2 MVCC Read Performance

As is mentioned in the beginning of this section, another issue that may have to be addressed by database servers with MVCC is an increased amount of I/O activities required to support multiversion read consistency for concurrent transactions. Furthermore, the pattern of read requests tends to be quite random. If a data object has been updated by other transactions, the correct version must be fetched from one of the rollback segments belonging to the transactions that updated the data object. At the presence of long-running transactions, the average cost of read by a transaction can get even higher, because a long chain of old versions may have to be traversed for each access to a frequently updated data object, causing more random reads [15, 20, 23].

The superior read performance of flash memory has been repeatedly demonstrated for both sequential and random access patterns (*e.g.*, [14]). The use of flash memory SSD instead of disk can alleviate the problem of increased random read considerably, especially by taking advantage of extremely low latency of flash memory.

To understand the performance impact of MVCC read activities, we ran a few concurrent transactions in snapshot isolation mode on a commercial database server following the scenario below.

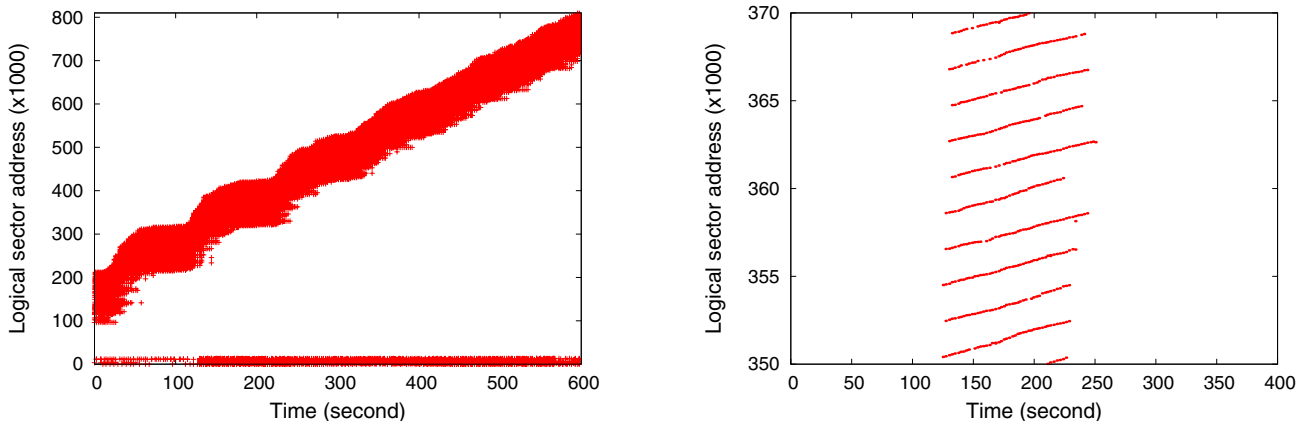
(1) Transaction T_1 performs a full scan of a table with 12,500 data pages of 8 KBytes each. (The size of the table is approximately 100 MBytes.)

(2) Each of three transactions T_2, T_3 and T_4 updates each and every tuple in the table one after another.

(3) Transaction T_1 performs a full scan of the table again.

The size of database buffer cache was set to 100 MBytes in order to cache the entire table in memory, so that the effect of MVCC I/O activities could be isolated from the other database accesses.

Figure 5 shows the pattern of reads observed at the last step of the scenario above when T_1 scanned the table for the second time. The x and y axes in the figure represent the timestamps of read requests and the logical addresses of sectors in the rollback segments to be read by the requests. The pattern of read was clustered but randomly scattered across quite a large logical address space of about one GBytes. When each individual data page was read from the table,



(a) Macroscopic view (b) Microscopic view

Figure 4: MVCC Write Pattern from TPC-C Benchmark (in Time×Address space)

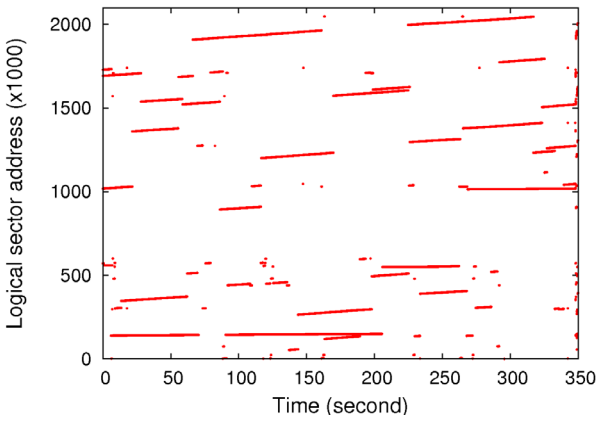


Figure 5: MVCC Read Pattern from Snapshot Isolation scenario (in Time×Address space)

T_1 had to fetch old versions from all three rollback segments (or extents) assigned to transactions T_2, T_3 and T_4 to find a transactionally consistent version, which in this case was the original data page of the table before it was updated by the three transactions.

	hard disk	flash SSD
# of pages read	39,703	40,787
read time	328s	21s
CPU time	3s	3s
elapsed time	351.0s	23.6s

Table 4: Undo data read performance

We measured actual performance of the last step of T_1 with a hard disk or a flash memory SSD drive being used as a storage medium for rollback segments. Table 4 summarizes the performance measurements obtained from this test. Though the numbers of pages read were slightly different between the cases of disk and flash memory SSD (presumably due to subtle difference in the way old versions were created in the rollback segments), both the numbers were close to what amounts to three full scans of the database table ($3 \times 12,500 = 37,500$ pages). Evidently, this was because all three old versions had to be fetched from rollback segments, whenever a transactionally consistent version of a data page was requested by T_1 running in the snapshot isolation mode.

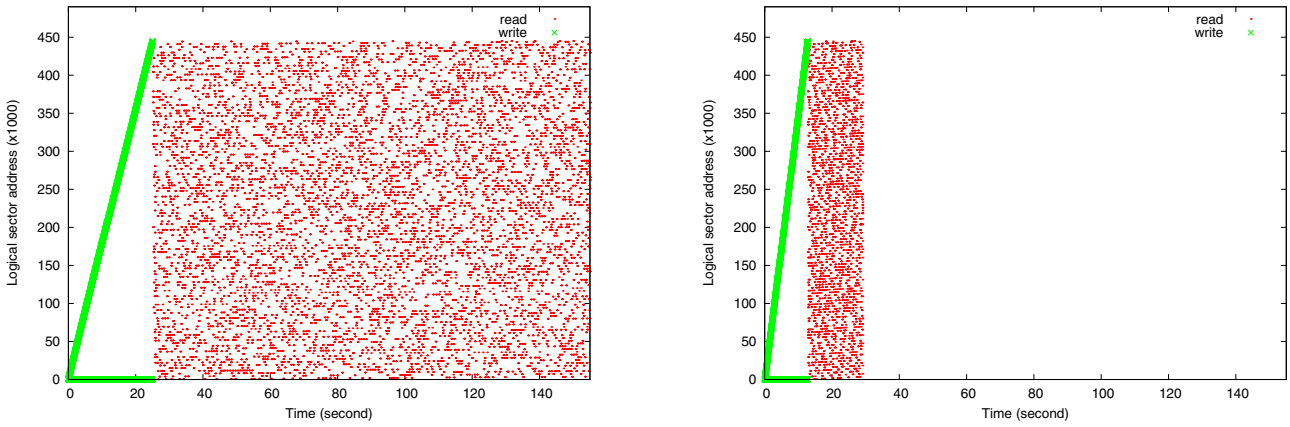
Despite a slightly larger number of page reads, flash memory SSD achieved more than an order of magnitude reduction in both read time and total elapsed time for this processing step of T_1, when compared with hard disk. The average time taken to read a page from rollback segments was approximately 8.2 msec with disk and 0.5 msec with flash memory SSD. The average read performance observed in this test was consistent with the published characteristics of the disk and the flash memory SSD we used in this experiment. The amount of CPU time remained the same in both the cases.

6. TEMPORARY TABLE SPACES

Most database servers maintain separate temporary table spaces that store temporary data required for performing operations such as sorts or joins. I/O activities requested in temporary table spaces are typically bursty in volume and are performed in the foreground. Thus, the processing time of these I/O operations on temporary tables will have direct impact on the response time of individual queries or transactions. In this section, we analyze the I/O patterns of sort-based and hash-based algorithms, and discuss the impact of flash memory SSD on the algorithms.

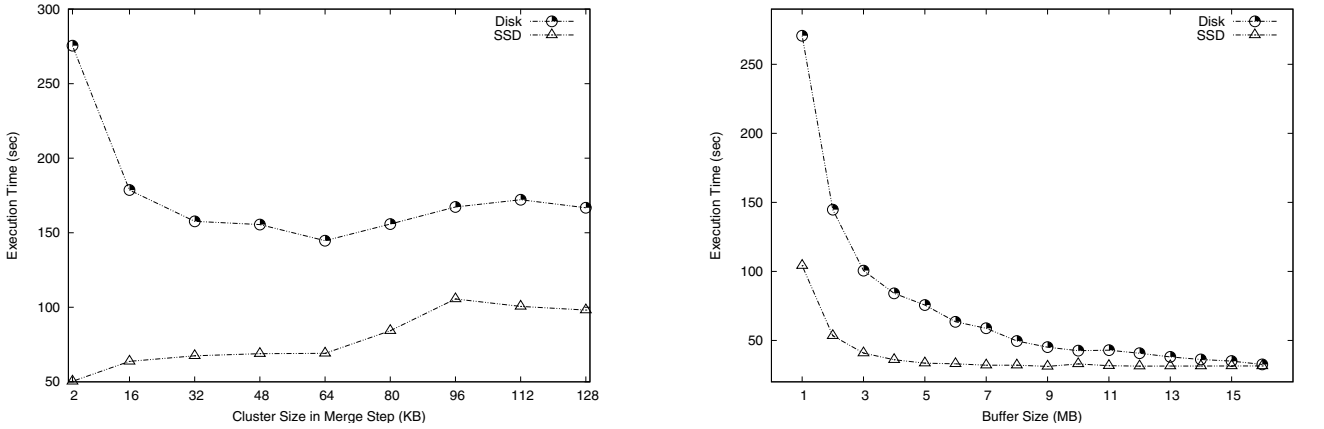
6.1 External Sort

External sort is one of the core database operations that have been extensively studied and implemented for most database servers, and many query processing algorithms rely on external sort. A sort-based algorithm typically partitions an input data set into smaller chunks, sorts the chunks (or runs) separately, and then merges them into a single sorted file. Therefore, the dominant pattern of I/O requests from a sort-based algorithm is *sequential write* (for writing sorted runs) followed by *random read* (for merging runs) [8].



(a) Hard disk (b) Flash memory SSD

Figure 6: IO pattern of External Sort (in Time×Address space)



(a) Varying cluster size
(buffer cache size fixed at 2 MB)

(b) Varying buffer size
(cluster size fixed at 64 KB for disk and at 2 KB for SSD)

Figure 7: External Sort Performance : Cluster size vs. Buffer cache size

To better understand the I/O pattern of external sort, we ran a sort query on a commercial database server, and traced all I/O requests made to its temporary table space. This query sorts a table of two million tuples (approximately 200 MBytes) using a buffer cache of 2 MBytes assigned to this session by the server. Figure 6 illustrates the I/O pattern of the sort query observed (a) from a temporary table space created on a hard disk drive and (b) from a temporary table space created on a flash memory SSD drive. A clear separation of two stages was observed in both the cases. When sorted runs were created during the first stage of sort, the runs were written sequentially to the temporary table space. In the second stage of sort, on the other hand, tuples were read from multiple runs in parallel to be merged, leading to random reads spread over the whole region of the time-address space corresponding to the runs.

Another interesting observation that can be made here is different ratios between the first and second stages of sort with respect to execution time. In the first stage of sort for run generation, a comparable amount of time was spent in each case of disk and flash memory SSD used as a storage device for temporary table spaces. In contrast, in the second stage of sort for merging runs, the amount of time spent on this stage was almost an order of magnitude shorter in the case of flash memory SSD than that in the case of disk. This is because, due to its far lower read latency, flash memory SSD can process random reads much faster than disk, while the processing speeds of these two storage media are comparable for sequential writes.

Previous studies have shown that the unit of I/O (known as *cluster*) has a significant impact on sort performance beyond the effect of read-ahead and double buffering [8]. Because of high latency of disk, larger clusters are generally expected to yield better sort performance despite the limited fan-out in run generation and the increased number of merge steps. In fact, it is claimed that the optimal size of cluster has steadily increased roughly from 16 or 32 KBytes to 128 KBytes or even larger over the past decade, as the gap between latency and bandwidth improvement has become wider [7, 9].

To evaluate the effect of cluster size on sort performance, we ran the sort query mentioned above on a commercial database server with a varying size of cluster. The buffer cache size of the database server was set to 2 MBytes for this query. The input table was read from the database table space, and sorted runs were written to or read from a temporary table space created on a hard disk drive or a flash memory SSD drive. Figure 7(a) shows the elapsed time taken to process the sort query excluding the time spent on reading the input table from the database table space. In other words, the amount of time shown in Figure 7(a) represents the cost of processing the I/O requests previously shown in Figure 6 with a different size of cluster on either disk or flash memory SSD.

The performance trend was quite different between disk and flash memory SSD. In the case of disk, the sort performance was very sensitive to the cluster size, steadily improving as cluster became larger in the range between 2 KB and 64 KB. The sort performance then became a little worse when the cluster size grew beyond 64 KB. In the case of flash memory SSD, the sort performance was not as much sensitive to the cluster size, but it deteriorated consistently as the cluster size increased, and the best performance was observed when the smallest cluster size (2 KBytes) was used.

Though it is not shown in Figure 7(a), for both disk and flash memory SSD, the amount of time spent on run generation was only a small fraction of total elapsed time and it remained almost constant irrespective of the cluster size. It was the second stage for merging runs that consumed much larger share of sort time and was responsible for the distinct trends of performance between disk and flash memory SSD. Recall that the use of a larger cluster in general improves disk bandwidth but increases the amount of I/O by reducing the fan-out for merging sorted runs. In the case of disk, when the size of cluster was increased, the negative effect of reduced fan-out was overridden by considerably improved bandwidth. In the case of flash memory SSD, however, bandwidth improvement from using a larger cluster was not enough to make up the elongated merge time caused by an increased amount of I/O due to reduced fan-out.

Apparently from this experiment, the optimal cluster size of flash memory SSD is much smaller (in the range of 2 to 4 KBytes) than that of disk (in the range of 64 to 128 KBytes). Therefore, if flash memory SSD is to be used as a storage medium for temporary table spaces, a small block should be chosen for cluster so that the number of steps for merging sorted runs is reduced. Coupled with this, the low latency of flash memory SSD will improve the performance of external sort quite significantly, and keep the upperbound of an input file size that can be externally sorted in two passes higher with a given amount of memory.

Figure 7(b) shows the elapsed time of the same external sort executed with a varying amount of buffer cache. The same experiment was repeated with a disk drive and a flash memory SSD drive as a storage device for temporary table space. The cluster size was set to 64 KBytes for disk and 2 KBytes for flash memory SSD, because these cluster sizes yielded the best performance in Figure 7(a). Evidently, in both the cases, the response time of external sort improved consistently as the size of buffer cache grew larger, until its effect became saturated. In all the cases of buffer cache size, flash memory SSD outperformed disk – by at least a factor of two when the buffer cache was no larger than 20% of the input table size.

6.2 Hash

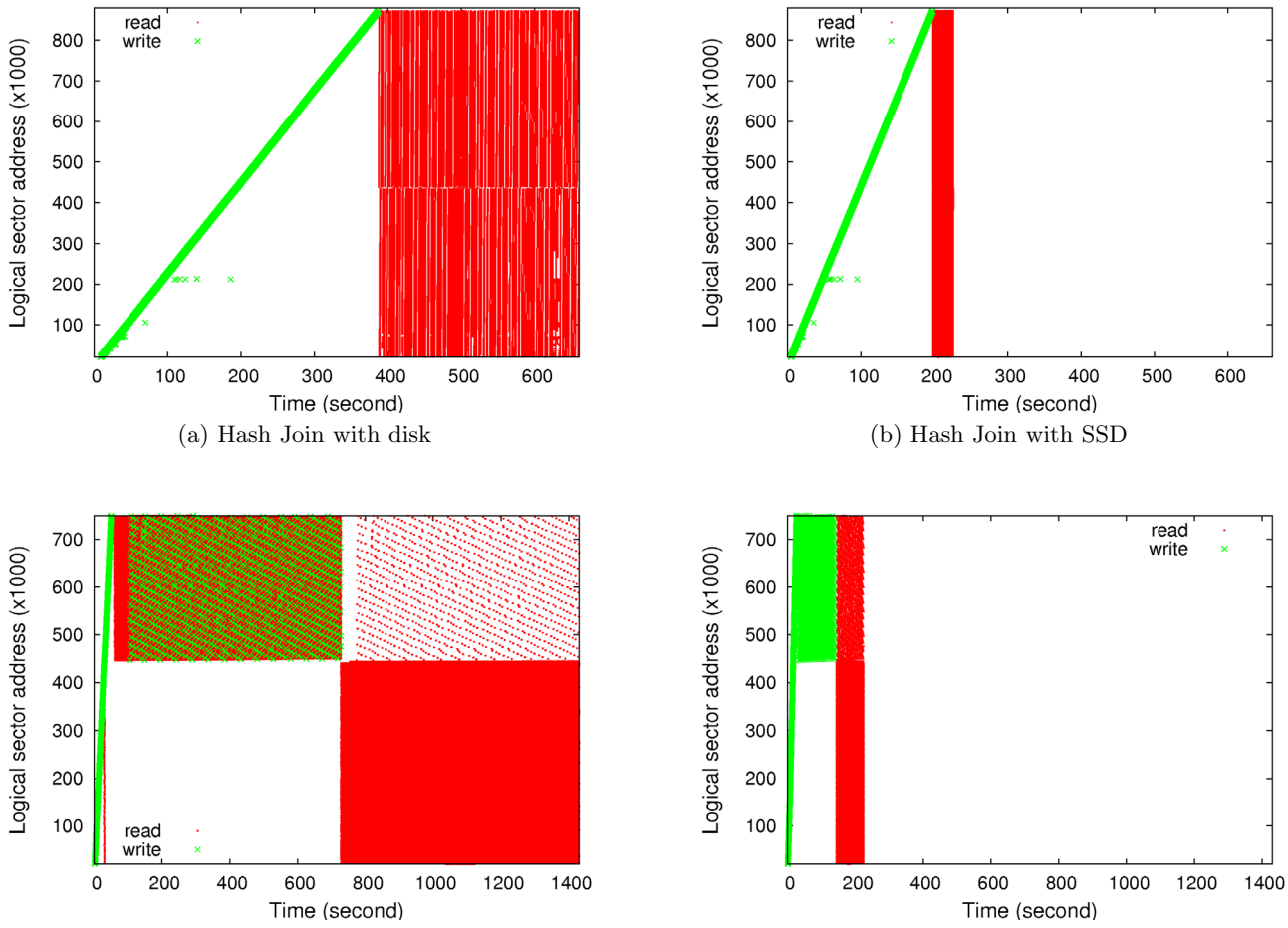
Hashing is another core database operation frequently used for query processing. A hash-based algorithm typically partitions an input data set by building a hash table in disk and processes each hash bucket in memory. For example, a hash join algorithm processes a join query by partitioning each input table into hash buckets using a common hash function and performing the join query bucket by bucket.

Both sort-based and hash-based algorithms are similar in that they divide an input data set into smaller chunks and process each chunk separately. Other than that, sort-based and hash-based algorithms are in principle quite opposite in the way an input data set is divided and accessed from secondary storage. In fact, the duality of hash and sort with respect to their I/O behaviors has been well studied in the past [8]. While the dominant I/O pattern of sort is *sequential write* (for writing sorted runs) followed by *random read* (for merging runs), the dominant I/O pattern of hash is said to be *random write* (for writing hash buckets) followed by *sequential read* (for probing hash buckets).

If this is the case in reality, the build phase of a hash-based algorithm might be potentially problematic for flash memory SSD, because the random write part of hash I/O pattern may degrade the overall performance of a hash operation with flash memory SSD. To assess the validity of this argument, we ran a hash join query on a commercial database server, and traced all I/O requests made to a temporary table space. This query joins two tables of two million tuples (approximately 200 MBytes) each using a buffer cache of 2 MBytes assigned to this session by the server. Figures 8(a) and 8(b) show the I/O patterns and response times of the hash join query performed with a hard disk drive and a flash memory SSD drive, respectively.

Surprisingly the I/O pattern we observed from this hash join was entirely opposite to what was expected as a dominant pattern suggested by the discussion about the duality of hash and sort. The most surprising and unexpected I/O pattern can be seen in the first halves of Figures 8(a) and 8(b). During the first (build) phase, both input tables were read and partitioned into multiple (logical) buckets in parallel. As shown in the figures, however, the sectors which hash blocks were written to were somehow located in a consecutive address space with only a few outliers, as if they were written in append-only fashion. What we observed from this phase of a hash join indeed was *similarity* rather than *duality* of hash and sort algorithms with respect to their I/O behaviors.

Since the internal implementation of this database system is opaque to us, we cannot explain exactly where this idiosyncratic I/O behavior comes from for processing a hash join. Our conjecture is that when a buffer page becomes full, it is flushed into a data block in the temporary table space in append-only fashion no matter which hash bucket the page belongs to, presumably because the size of each hash partition (or bucket) cannot be predicted accurately. Then, the affinity between temporary data blocks and hash buckets can be maintained via chains of links or an additional



(a) Hash Join with disk

(b) Hash Join with SSD

(c) Sort-Merge Join with disk

(d) Sort-Merge Join with SSD

Figure 8: IO Patterns of Hash Join and Sort-Merge Join

level of indirection. We believe this explains why the read requests during the second (probe) phase of hash join was randomly scattered rather than sequential.

Whatever the reason is for this implementation of hashing, the net effect is quite favorable for flash memory SSD. As Figures 8(a) and 8(b) show, the average response time of the hash join was 661.1 seconds with disk and 226.0 seconds with flash memory SSD.

Now that the dominant I/O pattern of both hash and sort is sequential write followed by random read, it is not difficult to expect a similar performance trend between disk and flash memory SSD for a sort-merge join query as well. From the same experiment repeated for a sort-merge query on the same commercial database server, the average response time of the sort-merge join was 1452.7 seconds with disk and 224.7 seconds with flash memory SSD, as shown in Figures 8(c) and 8(d).

When the hash join and sort-merge join queries were executed with disk, the response time of hash join was about twice faster than that of sort-merge join. This result concurs to some extent with the previous comparative study arguing that sort-merge join is mostly obsolete with a few exceptions [6]. On the other hand, when the hash join and sort-merge join queries were executed with flash memory SSD, their response times were almost identical. This evidence may not be strong enough to claim that sort-merge join is better suited than hash join with flash memory SSD, but it certainly warrants a fresh new look at the issue all over again.

7. CONCLUSIONS

We have witnessed a chronic imbalance in performance between processors and storage devices over a few decades in the past. Although the capacity of magnetic disk has improved quite rapidly, there still exists a significant and growing performance gap between magnetic disk and CPU. Consequently, I/O performance has become ever more critical in achieving high performance of database applications.

Most contemporary database systems are configured to have separate storage spaces for database tables and indexes, transaction log, rollback segments and temporary data. The overall performance of transaction processing cannot be improved just by optimizing I/O for tables and indexes, but also by optimizing it for the other storage spaces as well. In this paper, we demonstrate the burden of processing I/O requests for transaction log, rollback and temporary data is substantial and can become a serious bottleneck for transac-

tion processing. We then show that flash memory SSD, as a storage alternative to magnetic disk, can alleviate this bottleneck drastically, because the access patterns dominant in the storage spaces for transaction log, rollback and temporary data can best utilize the superior characteristics of flash memory such as extremely low read and write latency without being hampered by the *no in-place update* limitation of flash memory.

In our experiments, we have observed more than an order of magnitude improvement in transaction throughput and response time by replacing a magnetic disk drive with a flash memory SSD drive for transaction log or rollback segments. We have also observed more than a factor of two improvement in response time for processing a sort-merge or hash join query by adopting a flash memory SSD drive instead of a magnetic disk drive for temporary table spaces. We believe that a strong case has been made out for flash memory SSD, and due attention should be paid to it in all aspects of database system design to maximize the benefit from this new technology.

8. REFERENCES

[1] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD*, pages 1–10, San Jose, CA, May 1995.

[2] Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobbe. A Design for High-Performance Flash Disks. Technical Report MSR-TR-2005-176, Microsoft Research, December 2005.

[3] Transaction Processing Performance Council. TPC Benchmark. http://www.tpc.org/.

[4] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the ACM SIGMOD*, pages 1–8, Boston, MA, June 1984.

[5] Eran Gal and Sivan Toledo. Mapping Structures for Flash Memories: Techniques and Open Problems. In *International Conference on Software - Science, Technology & Engineering (SwSTE'05)*, Herzelia, Israel, February 2005.

[6] Goetz Graefe. Sort-Merge-Join: An Idea Whose Time Has(h) Passed? In *Proceedings of ICDE*, pages 406–417, Houston, TX, February 1994.

[7] Goetz Graefe. The Five-minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *Third International Workshop on Data Management on New Hardware (DAMON2007)*, Beijing, China, June 2007.

[8] Goetz Graefe, Ann Linville, and Leonard D. Shapiro. Sort versus Hash Revisited. *IEEE Transactions on Knowledge and Data Engineering*, 6(6):934–944, December 1994.

[9] Jim Gray. Rules of Thumb in Data Engineering. In *Proceedings of ICDE*, pages 3–12, San Diego, CA, March 2000.

[10] Jim Gray and Bob Fitzgerald. Flash Disk Opportunity for Server-Applications. http://www.research.microsoft.com/~gray, January 2007.

[11] Chang-Gyu Hwang. Nanotechnology Enables a New Memory Growth Model. *Proceedings of the IEEE*, 91(11):1765–1771, November 2003.

[12] Intel. Understanding the Flash Translation Layer (FTL) Specification. Application Note AP-684, Intel Corporation, December 1998.

[13] Edmond Lau and Samuel Madden. An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse. In *Proceedings of VLDB*, pages 703–714, Seoul, Korea, September 2006.

[14] Sang-Won Lee and Bongki Moon. Design of Flash-Based DBMS: An In-Page Logging Approach. In *Proceedings of the ACM SIGMOD*, pages 55–66, Beijing, China, June 2007.

[15] David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. Transaction Time Support Inside a Database Engine. In *Proceedings of ICDE*, pages 35–44, Atlanta, GA, April 2006.

[16] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proceedings of ICDE*, pages 535–546, Boston, MA, March 2004.

[17] Oracle. Oracle Flashback Technology. http://www.oracle.com/technology/deploy/-availability/htdocs/Flashback_Overview.htm, 2007.

[18] Chanik Park, Prakash Talawar, Daesik Won, MyungJin Jung, JungBeen Im, Suksan Kim, and Youngjoon Choi. A High Performance Controller for NAND Flash-based Solid State Disk (NSSD). In *The 21st IEEE Non-Volatile Semiconductor Memory Workshop (NVSMW)*, Monterey, CA, February 2006.

[19] David A. Patterson. Latency Lags Bandwidth. *Communications of the ACM*, 47(10):71–75, October 2004.

[20] Michael Stonebraker. The Design of the POSTGRES Storage System. In *Proceedings of VLDB*, pages 289–300, Brighton, England, September 1987.

[21] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stravros Harizopoulos, Nabil Hachem, and Pat Helland. The End of an Architectural Era (It's Time for a Complete Rewrite). In *Proceedings of VLDB*, pages 289–300, Vienna, Austria, September 2007.

[22] Theo Härder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Survey*, 15(4):287–317, December 1983.

[23] Kimberly L. Tripp and Neal Graves. SQL Server 2005 Row Versioning-based Transaction Isolation. Microsoft SQL Server Technical article, July 2006.

[24] Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. In *FAST02*, January 2002.