Selective I/O Bypass and Load Balancing Method for Write-Through SSD Caching in Big Data Analytics

Jaehyung Kim, Hongchan Roh, and Sanghyun Park

Abstract—Fast network quality analysis in the telecom industry is an important method used to provide quality service. SK Telecom, based in South Korea, built a Hadoop-based analytical system consisting of a hundred nodes, each of which only contains hard disk drives (HDDs). Because the analysis process is a set of parallel I/O intensive jobs, adding solid state drives (SSDs) with appropriate settings is the most cost-efficient way to improve the performance, as shown in previous studies. Therefore, we decided to configure SSDs as a write-through cache instead of increasing the number of HDDs. To improve the cost-per-performance of the SSD cache, we introduced a selective I/O bypass (SIB) method, redirecting the automatically calculated number of read I/O requests from the SSD cache to idle HDDs when the SSDs are I/O over-saturated, which means the disk utilization is greater than 100 percent. To precisely calculate the disk utilization, we also introduced a combinational approach for SSDs because the current method used for HDDs cannot be applied to SSDs because of their internal parallelism. In our experiments, the proposed approach achieved a maximum 2x faster performance than other approaches.

Index Terms—I/O load balancing, SQL-on-Hadoop, SSD cache, storage hierarchies

1 Introduction

SK telecom, which is the largest wireless communications service provider in South Korea, has built an analytical system based on Hadoop clusters for the radio signal quality analysis of wireless cell towers. In this system, an extract-transform-load (ETL) process periodically loads data to a hadoop distributed file system (HDFS) [1]. For newly added data, SparkSQL [2], which is known as an SQL-on-Hadoop platform [3], executes analytical queries that cause read dominant and heavy I/O loads on the disks [4].

In our system, each node is a scaled-up server that has a vast number of cores in contrast to a small number of HDDs. The performance gap between the processing unit and storage was broad and obvious. Nevertheless, it was difficult to increase the number of HDDs because of the limited budget cost, small number of disk bays, and power consumption per rack. Moreover, the storage performance should be improved without data integration.

In the previously conducted study, adding SSDs with appropriate settings on the cluster node was found to be the most efficient method for improving the performance of MapReduce, which is the beginning of SQL-on-Hadoop, based on the price per performance when there was no capacity constraint [5]. This was because an HDD cannot efficiently tolerate multiple requests at one time, but an SSD has internal parallelism [6], which allows it to perform multiple requests concurrently without a severe performance reduction.

Therefore, we applied SSDs to the existing system. Specifically, we configured the SSDs as a write-through cache for the following reasons. By replacing the cached data, an SSD cache can resolve the issue noted in the previous study [5] that SSDs become filled with data faster than HDDs because SSDs should store more data in proportion to the theoretical bandwidth compared with HDDs to exploit the maximum bandwidth. Moreover, an SSD write-through cache can prevent data loss when an SSD is worn out [7]. Above all, an SSD cache can be integrated into an existing HDFS cluster without reconfiguration and data migration.

Most general purpose SSD caching software from both academia and industry has focused on increasing the cache hit ratio because its main concern is reducing the I/O latency. However, in parallel distributed systems that have a dataset that is too large to fit into memory, the aggregate bandwidth is a more important factor than the latency. Thus, HDDs are still useful resources. From this perspective, we propose the selective I/O bypass (SIB) method for load balancing among disks, including SSDs and HDDs. The SIB method redirects the automatically calculated number of read I/O requests (not write I/O requests) from the SSD cache to idle HDDs when a large number of I/O requests beyond the performance limit of the SSD are issued to the SSD. We define this situation as I/O over-saturation. Consequently, both the SSDs and HDDs will be fully utilized. Note that bypassing the read I/O requests is possible because the cached data must also be stored in at least one HDD in the write-through SSD cache.

However, there is no way to detect the I/O saturation of the SSD using the current measurement, because it is skewed as a result of the internal parallelism of SSD. Therefore, we propose an estimation method that uses both the bandwidth and average I/O await time. In fact, bandwidth and the average await time fluctuate due to garbage collection when the mixed read/write I/O requests are issued. We detect this fluctuation and ignore it by analyzing trend for a short period because fluctuation happens intermittently for only a brief time.

We implemented the SIB method using bcache [8], which is open-source SSD caching software, i.e., a Linux kernel block layer cache. Note that we did not modify the Linux kernel to forestall compatibility issues.

We evaluated the SIB method in comparison with various approaches to 22 TPC-H queries [9] using SparkSQL to confirm the efficiency and the scalability across various workloads generated by the query operators used in a distributed query processor, such as join and aggregation operations. Moreover, we conducted a TeraSort benchmark to demonstrate the generality of our approach.

Our experimental results showed that the maximum performance gain in the query execution time was 2.06 times faster than the original bcache, 1.68 times faster than Spark RDD persistence, 1.35 times faster than the performance-driven approach and 1.99 times faster than the archival storage of HDFS.

The remainder of this paper is organized as follows. Section 2 describes the motivation and related works. We explain how the proposed approach selectively bypasses I/O streams and controls the load balance between SSDs and HDDs in Section 3. The experimental results for the throughput and load balancing are given in Section 4. Finally, Section 5 presents the conclusions, open problem and future work.

2 Related Works and Background

In this section, we provide a brief explanation of the components employed in the proposed approach and studies related to the proposed approach.

2.1 Hybrid Storage Architecture

There have been many attempts to exploit flash SSDs as primary storage in enterprise environments; however, some limitations remain, such as the life span. In a recent study on the reliability of

- J. Kim and S. Park are with the Department of Computer Science, Yonsei University, Seoul 03722, Republic of Korea. E-mail: {jaehyungkim, sanghyun}@yonsei.ac.kr.
- H. Roh is with SK Telecom, Bundang 100-999, Republic of Korea. E-mail: hongchan.roh@sk.com.

Manuscript received 16 Feb. 2017; revised 20 Oct. 2017; accepted 31 Oct. 2017. Date of publication 7 Nov. 2017; date of current version 19 Mar. 2018.
(Corresponding author: Jaehyung Kim.)
Recommended for acceptance by R.F. DeMara.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2771491

TABLE 1
Comparison Table for Hybrid Storage Researches

	bcache	Archival storage	Spark RDD Persistence	Performance-driven Approach	Selective I/O Bypass
Management method	Caching	Tiering (exclusive cache*)	Caching	Caching	Caching
Caching granularity	Bucket (e.g. 512 KB)	HDFS block (e. a. 128 MB)	RDD	Extent (e.g. 128 KB)	Bucket (e.g. 512 KB)
Replacement method	LRU, FIFO, Random	N/A	Manual	LRU	FIFO
Limitations adapting to legacy system	HDD format and data migration are required	Data migration is needed	-	Data migration is needed	-
Load balancing	N/A	Based on HDFS block placement policy	Manual	Extent placement policy	Dynamic method

Data is guaranteed to be in at most one of the storages, never in both.

flash SSDs, it was determined that flash SSDs have a higher rate of uncorrectable errors than HDDs [10]. Thus, SSD caching can supplement the reliability of flash SSDs.

Most of all, extra data migration and reconfiguration should be avoided in our operating environment. Version management of frameworks in a cluster environment is an important issue in an operating perspective. Many studies about heterogeneous storages have tried to improve utilization by leveraging replacement policies between SSDs and HDDs in cluster environment [11], [12]. However, these approaches require modification of the existing cluster and extra migration.

One study stated that the ensuring the equality of the await time for all the requests leads to a high SSD caching throughput by distributing write I/O operations like SSD tiering and performing background data migration between SSDs and HDDs [13]. That study attempted to eliminate the HDD idle time by allocating data to be read on HDDs rather than SSDs. Similar to that study, an allocation method across heterogeneous devices consisting of multi-tiered storage for evenly-balanced concurrent clients has been proposed [14]. The main purpose of this study coincides with the purpose of our approach, which attempts to maximize storage utilization. However, their method was not valid if the workload is different when the data loading and processing such as HDFS. In that situation, data migration is likely to occur and may cause extra I/O load. In the industrial area, general purpose SSD caches exists to reduce the response time [15], [16]. Open-source groups have released many general purpose SSD cache solutions [8], [17], [18]. All these solutions have mainly focused on increasing the cache hit rate to leverage the shorter latency of SSDs. These hit ratio oriented solutions cannot balance I/O load between many HDDs and SSD cache. Our approach can alleviate this problem and can be integrated into an existing cluster without data migration and the reconfiguration of clusters. Table 1 differentiate among various approaches with respect to performance-related features.

TABLE 2
Block Layer Statistics

Name	Units	Description
I/O ticks	Milliseconds	Total time the block device has been active
Read ticks	Milliseconds	Total wait time for read requests
Write ticks	Milliseconds	Total wait time for write requests
Read I/Os	Requests	Number of processed read I/Os
Write I/Os	Requests	Number of processed write I/Os
Read sectors	Sectors	Number of sectors read
Write sectors	Sectors	Number of sectors written

TABLE 3
Disk Performance Measures

Name	Units	Definition
Utilization	Percentage	I/O ticks/Total CPU ticks
Read bandwidth	Bytes per second	Read sectors/Elapsed I/O ticks
Write bandwidth	Bytes per second	Write sectors/Elapsed I/O ticks
Read response time	Milliseconds	CPU ticks/Read I/Os
Write response time	Milliseconds	CPU Ticks/Write I/Os

2.2 Utilization Measure Based on HDD

In the Linux kernel, there are many statistical metrics for the block device, as listed in Table 2. By regularly observing these statistical metrics, the status can be measured regarding the performance and used to manually adjust the system configuration to better balance the I/O load between block devices [19].

In Table 3, we briefly describe the formulas that are used to monitor the disk status. Utilization is defined as the rate of the CPU time used for I/O processing (referred to as I/O ticks) during I/O requests issued to a given block device. According to this definition, I/O saturation occur when the utilization reaches to 100 percent. This definition applies to an HDD but not to an SSD because of the internal parallelism of an SSD. Traditionally, the CPU time can be used to measure the utilization of block devices because an HDD is unable to service concurrent requests because of its physical architecture.

Fig. 1 shows why I/O ticks cannot be used to measure the SSD utilization. Fig. 1a shows that an HDD cannot accept an I/O request before the previous request is completed. In contrast, Fig. 2b indicates that multiple I/O requests can be submitted concurrently and overlap. If only serial I/O requests from a single process are submitted sequentially to a device, the utilization goes up to 100 percent even if an SSD can perform more requests.

3 Selective I/O Bypass Method

3.1 Basic Operations

Our approach follows a general policy of write-through cache; however, the read I/O requests cannot be cached on an SSD as

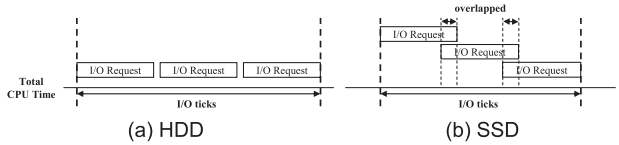


Fig. 1. Different calculation methods for HDD and SSD utilization.

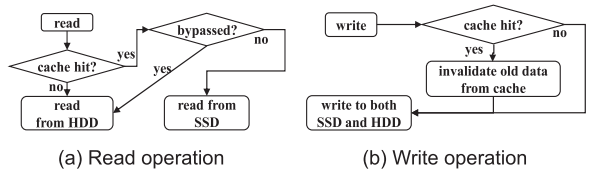


Fig. 2. Basic operations of SSD cache with SIB.

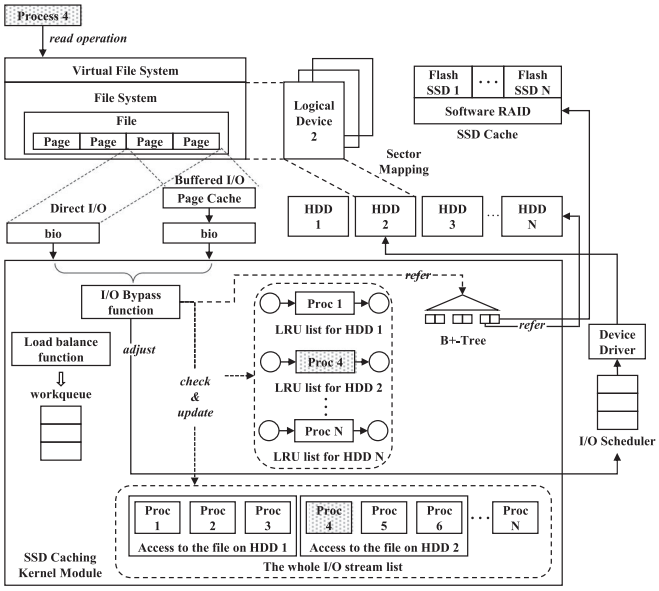


Fig. 3. SSD caching with the SIB method.

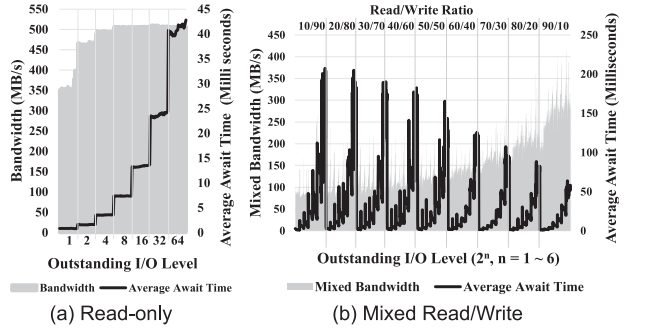


Fig. 5. Bandwidth and average await time as outstanding I/O level increases for 256 KB block size.

shown in Fig. 2a. The data can only be cached when the write I/O requests are issued as shown in Fig. 2b. The write-only cache policy prevents cache pollution from rarely accessed data in an analytics environment. If the read I/O request hits the cache, it can be bypassed by our approach as described in Section 3.2. The SIB method activates the bypassing when I/O saturation occurs on the SSD. The I/O saturation can be detected, and the temporary fluctuation is ignored as described in Section 3.3.

3.2 I/O Bypass Method

An important design consideration is that the HDFS relies on data parallelism, which involves many processes for the same task, but to the different data. The SIB method selects a process that delivers relatively large and sequential read I/O requests to an SSD and bypasses the read I/O requests from the SSD to an idle HDD. Note that a large and sequential I/O on the HDD can be served at a higher throughput, because there is a smaller extra delay compared to that for a random I/O. After the process is completed, the other processes will be selected as target for the I/O bypass.

Since the SSD cache is in the kernel block layer, the information that can be used to identify each process and to detect its termination will be restricted, if the kernel code is not modified. For compatibility reasons, we do not make any changes to the Linux kernel, which necessitates a data structure in the kernel module to obtain information on the process lifecycle. To detect the process, the SIB method should maintain a data structure such as the LRU list (as shown in Fig. 4), which stores information related to the kernel process.

Then, if the process has completed its I/O requests, another process should replace it as soon as possible, in order to minimize the idle time. If the terminated process is replaced slowly, the idle time will be sufficient to cause degradation of the performance gain.

In Fig. 3, process 4 generates a read operation for the file residing on logical device 2, which is mapped to HDD 1 sector-by-sector. If the page is not cached, the read request is performed on HDD 2. The file system transfers the read operation, which is converted into a bio structure, to/from the SSD caching kernel module. There is no

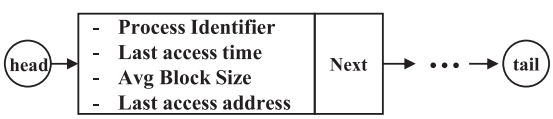


Fig. 4. I/O stream list.

difference between the direct I/O mode and the buffered I/O mode at this point. The I/O bypass function (Section 3.2) checks whether the current bio can be bypassed by searching the key in the B+-Tree. Let's assume that the current bio hits the cache.

Then, the I/O bypass function determines whether the process that generates the current bio is in the LRU list. Here, Process 4 is in the LRU list; therefore, the current bio is part of the bypassed I/O requests, and the SIB method changes the destination from logical device 2 to HDD 2. Process 4 is in the LRU list, because the I/O access pattern is sequential and larger than the others in the same group, which consists of Processes 4, 5, and 6. Similarly, the SIB method chooses I/O requests generated from Process 1 as a bypassed I/O stream. Before the request is issued to the request queue, the SIB method updates the statistics on the list structure. Finally, the SIB method issues the request to the request queue, where it is dedicated to HDD 2.

3.3 Load Balancing Method

3.3.1 Estimation Method for I/O Saturation

To estimate the I/O saturation of an SSD, we exploit the fact that the average await time of an SSD increases while the bandwidth is sustained as the number of issued I/O requests are increased in I/O saturation. We consider four factors that affect the bandwidth and the average await time. First, the maximum bandwidth varies as the block size changes. Second, if the block size is fixed, the maximum bandwidth also changes, as the read/write ratio changes. Third, sharp fluctuations exist in the I/O statistics due to garbage collection of an SSD. Lastly, the measured values fluctuate slightly even if the same number of I/O requests is issued. Therefore, the maximum bandwidth should be defined as a range.

These factors were confirmed through experimental results, as shown in Fig. 5. The bandwidth and average await time were measured for the read-only and the mixed read/write workloads with 256 KB request size as the number of outstanding I/O requests increased. The number of outstanding I/O requests indicates the number of simultaneous requests served to the device. Although fluctuations exist in both the bandwidth and average await time, these values are converged into the mean.

We only collect I/O statistics if the aggregate bandwidth increases immediately after the number of bypassed I/O streams are increased. By doing this, it is ensured that the collected values are always within the maximum range. Moreover, we update the minimum average await time at this point if it is lower than before. The minimum average await time indicates that if we bypass an I/O stream from the SSD to an idle HDD when the measured average await time is equal to or higher than the minimum average await time, then the aggregate bandwidth will be increased.

We incrementally calculate the mean, variance and standard deviation of the bandwidth values to derive the standard score (a.k.a. z-score) [20]. The z-score shows how far the observed value is from the mean. In our approach, we obtained a score higher than

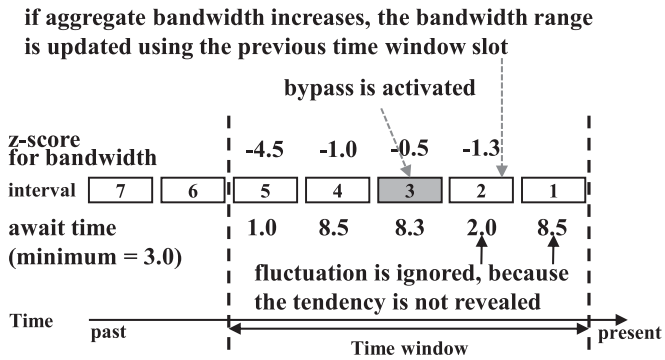


Fig. 6. Load balancing method based on I/O saturation estimation.

−4.0, as it is within the maximum bandwidth range. If the z-score of the observed bandwidth is higher than −4.0 and if the observed average await time is higher than the minimum, then the SSD is over-saturated.

3.3.2 Load Balancing Method Details

The load balancing function in Fig. 3 dynamically controls the number of bypassed I/O streams, based on the continued tendency of the I/O saturation of an SSD to eliminate noise from the fluctuation of the SSD. As shown in Fig. 6, we assume that the fluctuation happened at the second interval from the present and that the minimum average await time was 3.0. The number of bypassed I/O processes should be increased from now on; however, it is ignored because the current average await time exceeds the minimum value. Besides, the minimum await time is not updated to 2.0. If an HDD is saturated, the HDD is away from the bypassing candidates. It should be noted that because the currently used measure is still valid for an HDD, we adapt it for an HDD.

Until the size of the sample is sufficient to be used for measuring the I/O saturation, the load balancing function eagerly and randomly tries to increase the number of bypassed I/O streams. This method is called exploration. If the aggregate bandwidth is decreased immediately after increasing the number of bypassed I/O streams, the number of bypassed I/O streams will restore to the original setting. If the size of the sample is sufficient, the SIB method depends on the I/O statistics. This is called exploitation.

The SIB method attempts to select the lowest utilized HDD as the target where the I/O requests are performed. If an HDD is already saturated, this HDD will be away from bypass candidate. It should be noted that bypassing on the HDD that performs write I/O requests is disabled.

Then, it checks whether the SSD is still over-saturated. If it is over-saturated, the load balancing function attempts to find another HDD and performs the same job again. If the SSD is not saturated while the load balancing function checks the SSD utilization continuously, the entire LRU list will have zero length to be disabled bypassing.

For this, we build statistics individually for both the read-only and read/write mingled patterns. In this case, we define the saturation range by using the maximum bandwidth and the sum of the average read/write response time. If over-saturation occurs due to

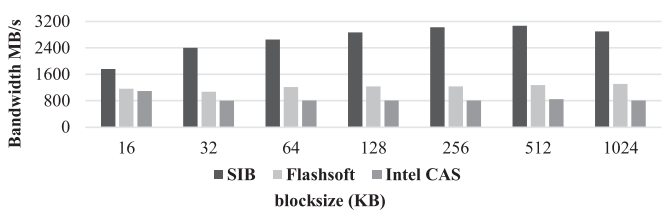


Fig. 7. Average throughput for sequential read with various block sizes.

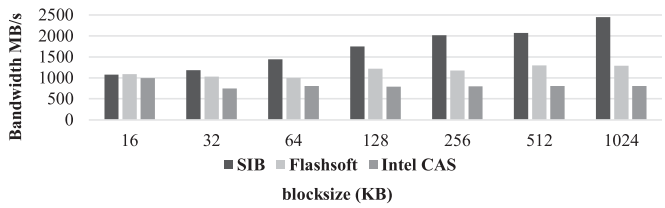


Fig. 8. Average throughput for random read with various block sizes.

massive write requests, the SIB method bypasses as many read requests as possible to prioritize the write requests.

This section presents the results of the evaluation conducted to analyze the influence of the SIB method on SQL-on-Hadoop and another application.

4 Evaluation

This section presents theresults of the evaluation con-ducted to analyze the influence of the SIB method on SQL-on-Hadoop and another application.

4.1 Customized Benchmarks

4.1.1 Throughput

We ran the FIO benchmark test [21] with multiple sequential read operations using different block sizes to compare the SIB method to commercial SSD cache such as SanDisk FlashSoft and Intel cache acceleration software (CAS), which are the representative general purpose SSD cache. For this experiment, we use eight 110 MB/s HDDs and four 550 MB/s SSDs. The caching strategy was set to write-through, and most of the data were cached on an SSD.

As seen in Fig. 7, the SIB method always showed a better performance regardless of the block size. However, we concluded that the performance of the SIB method improved as the block size increased because an HDD was appropriate for a large block size, which reduced the latency. Thus, if there were multiple I/O streams with different block sizes, bypassing the I/O requests with the larger request size rather than the smaller request size improved the efficiency of the proposed method.

Fig. 8 shows similar results for the case where the I/O pattern contained random read requests. An HDD cannot tolerate random I/O requests efficiently, and the performance gain from the SIB method was much lower than that for sequential I/O requests. These results provided a good reason for choosing the bypassed I/O stream that a sequential I/O pattern and larger request sizes. This is already reflected in our proposal.

4.1.2 Responsiveness and Accuracy

Here, we report how quickly the proposed approach could react to a change, along with its accuracy, based on the results of a simulation that was conducted using FIO benchmark tests.

We defined three workloads that were executed serially. All of the I/O requests hit the SSD cache. The characteristics of these workloads are described as follows.

1. Section (A): *Over-saturation*. The number of I/O streams was set to greater than twice the number of HDDs to cause I/O over-saturation. Each I/O stream performed sequential reads with a request size of 128k.
2. Section (B): *Under-saturation*. The bandwidth was intentionally limited to 110 MB/s for each stream to cause under-saturation. Bypassing was disabled in this situation to obtain the optimal performance.
3. Section (C): *Precise Saturation*. This section was designed to determine the accuracy of the proposed approach. We restricted the bandwidth of an I/O stream to 150 MB/s. If

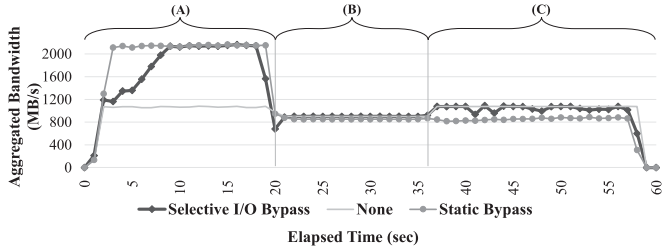


Fig. 9. Responsiveness and accuracy of SIB method compared to static bypass.

one I/O stream was bypassed, the throughput decreased to 110 MB/s due to the theoretical bandwidth of an HDD. Thus, when no I/O stream was bypassed, it was in the optimal state.

In this experiment, static bypass means that the number of bypassed I/O streams was set manually and did not change during the query processing. The none mode meant that only the SSD cache was used.

The experimental results show that the SIB method always extracted the maximum bandwidth. In section (A) in Fig. 9, the static bypass mode shows the best result, and the SIB method gradually reaches the maximum bandwidth. The automatic load balancing algorithm can eventually find the optimal aggregated bandwidth (accuracy). In section (B), the SIB method does not bypass any I/O stream because the aggregated bandwidth does not exceed the theoretical maximum bandwidth of the SSD.

The saturation range is updated continuously and gradually as the load is performed; The time required by the SIB method to find the optimal bandwidth can be reduced (responsiveness). This is confirmed in section (C) because the bandwidth fluctuation is gradually reduced.

We found that the proposed approach could efficiently react to various workloads in the FIO benchmark tests.

4.2 Performance Evaluation on SQL-on-Hadoop

4.2.1 Hardware Configuration

Our experiments were conducted on a cluster of nine state-of-the-art nodes. One of the nodes was solely used for the HDFS NameNode, SparkSQL coordinator. We used the YARN ResourceManager as the SparkSQL cluster manager. The remaining nodes were configured as HDFS DataNodes to store the data designated by SparkSQL workers for query processing. All of the nodes had two Xeon E5-2660 v3 processors, each of which had 10x physical core running at 2.6 GHz, with 128 GB of memory, one 10 gigabit Ethernet card, ten 150 MB/s SATA-III 1 TB HDDs used as the original disks of the logical devices, and one 550 MB/s SATA-III 500 GB SSD. The SSD was partitioned, with one partition for a local directory where the intermediate results were temporarily stored. The other partition was used as an SSD cache.

4.2.2 Software Configuration

We used Hadoop 2.7.3 and SparkSQL 2.2.0 in our experiments. The HDFS replication factor was set to three as a default, and the size of the HDFS block was set to 128 MB. The HDFS file was in an uncompressed and un-indexed text file format.

We conducted experiments with four different solutions used for comparison with the proposed approach. First, Archival storage [22] is a built-in SSD tiering technique in HDFS that controls the data movement between HDDs and SSDs. Archival storage has various block placement policies. Among these strategies, we chose the One_SSD policy because it is difficult for the other policies to warm up ideally to be balanced for heterogeneous devices. Second, original bcache was used as a representative hit ratio-oriented SSD cache. Third, Spark provides a caching technique called the RDD persistence. Lastly, we implemented a performance-driven approach (PDA) that focused on both the throughput and latency equilibrium between the HDDs and SSD cache using data distribution. We configured the parameters as well as possible, given the hardware and software settings.

4.2.3 Performance Evaluation

First, we investigated how the proposed approach worked with various workloads by monitoring the I/O requests and other system statistics.

In this experiment, we generated a 1 TB dataset, which was sufficiently large to utilize the aggregated bandwidth of the SSD caching. Using this dataset, we ran 22 TPC-H queries [17] five times, each of which was solely issued to SparkSQL to measure the average elapsed time for each query execution because each query had its own I/O characteristics. We used TPC-H scripts implemented as Scala[1] to use Spark RDD persistence. We configured all the tables as cached except for the lineitem table to distribute the I/O load using the theoretical bandwidth between the SSD and HDDs.

Table 4 lists the execution times (in seconds) of the TPC-H queries in SparkSQL. For each case, there is a column that presents the ratio between the execution times for each method and the SIB method. This metric shows that the SIB method outperformed the others in most cases. These results verify that other approaches cannot efficiently distribute I/O load. Interestingly, the archival storage shows slow performance. We found that the replicas were not uniformly distributed across the HDDs when using the archival storage method. This inefficiency results in the degradation of the throughput. It should be noted that the PDA showed a good performance compared to the other methods, except for the SIB method. The data loading and analytics have different workloads. Thus, the PDA, which is solely based on the average await time, could not balance I/O loads in this case.

The next section discusses how representative TPC-H queries with distinct I/O patterns were analyzed as the query operator and database tables changed.

4.2.4 TPC-H Query 6

TPC-H query 6 included four filtering predicates and a single aggregation operator was executed while scanning the lineitem table. In query 6, SparkSQL did not consume much CPU power but did consume a large I/O bandwidth, because the shuffle write phase is an only small portion of each map operation. This means that faster read I/O performance leads to a performance gain.

Fig. 10 shows the variation of the aggregate bandwidth for each case on the node consisting of the cluster for query 6 during the query processing. As seen in this figure, the SIB method outperformed the other methods with respect to the average read throughput.

The archival storage used both SSDs and HDDs but was much slower than the proposed approach because the initial data distribution on the local storages (including an SSD) was not uniform. Thus, many tasks could sometimes be skewed on a specific disk. The upper bound of the aggregate bandwidth of the archival storage was almost the same as for the SIB method because data were well distributed among storages at the specific moment

Fig. 11a shows how much load was issued to each storage for each node. If we assume that the aggregate bandwidth for the HDDs is approximately 750 MB/s with 550 MB/s for the SSD cache, the ideal theoretical load distribution would be almost 5.8/4.2. In the SIB method, the load distribution is approximately 5.1:4.9, with 6.7:3.3 for the archival storage. However, the ratio for PDA is 4:6, which is worse than the archival storage but shows a better performance. The data distribution between HDDs is not uniform in the archival storage, and the I/O requests are intensively issued to specific HDDs where the more data are stored than

1. https://github.com/ssavvides/tpch-spark

TABLE 4
TPC-H Execution Time on SparkSQL

	Original bcache		Archival Storage		Spark Cache (RDD Persistence)		Performance-driven Approach		SIB
	Execution Time (sec)	ORIGINAL bcache over SIB	Execution Time (sec)	Archival Storage over SIB	Execution Time (sec)	SparkRDD persistence over SIB	Execution Time (sec)	Performance driven Approach over SIB	Execution Time (sec)
Q1	233.52	1.95	203.92	1.70	201.74	1.68	134.38	1.12	119.81
Q2	78.68	1.24	95.16	1.50	74.28	1.17	72.24	1.14	63.3
Q3	308.33	1.68	377.17	2.06	250.47	1.37	200.2	1.09	183.21
Q4	300.9	1.68	277.27	1.54	258.44	1.44	189.36	1.05	179.62
Q5	335.83	1.59	332.51	1.57	275.28	1.30	232.24	1.10	211.85
Q6	233.31	1.99	201.3	1.71	189.27	1.61	126.5	1.08	117.4
Q7	336.48	1.59	327.73	1.55	273.32	1.29	236.09	1.12	211.43
Q8	336.5	1.53	341.64	1.55	297.54	1.35	241.81	1.10	219.77
Q9	434.23	1.17	557.79	1.51	437.85	1.18	497.89	1.35	369.58
Q10	297.07	1.64	309.53	1.71	248.34	1.37	204.09	1.12	181.49
Q11	69.55	1.38	80.74	1.60	62.67	1.24	49.08	0.97	50.38
Q12	294.56	1.76	297.31	1.77	227.95	1.36	177.23	1.06	167.68
Q13	86.39	1.20	90.5	1.26	78.64	1.09	75.02	1.04	71.89
Q14	254.59	1.96	249.91	1.92	203.15	1.56	144.57	1.11	130.05
Q15	245.79	1.78	248.62	1.80	210.6	1.53	146.6	1.06	137.76
Q16	109.06	1.13	131.86	1.36	96.25	0.99	98.1	1.01	96.93
Q17	547.07	1.55	636.8	1.81	563.44	1.60	427.98	1.22	352.12
Q18	558.96	1.64	606.4	1.78	494.01	1.45	382.09	1.12	341.23
Q19	271.85	1.78	284.18	1.86	224.27	1.47	175.99	1.15	152.55
Q20	330.67	1.64	352.26	1.75	285.47	1.42	226.46	1.13	201.03
Q21	899.4	1.44	1111.48	1.78	855.96	1.37	660.7	1.06	623.31
Q22	98.14	1.50	99.12	1.52	83.32	1.27	68.24	1.04	65.38

The right column for each case presents the ratio between the execution time for each case over the SIB method's execution time. The SIB method's execution times are normalized to one, with a lower number being better.

in the other devices, as presented in Fig. 11b. On the other hand, other methods use a round robin policy to place HDFS blocks between HDDs, which can ensure a uniform load distribution.

4.2.5 TPC-H Query 9

As shown listed in Table 4, query 9 is approximately 1.35 times faster with the SIB method than with the PDA. This query consists of multiple subqueries with join, aggregate, and sort operations. Typically, a sort operation may generate large intermediate results that do not fit into the memory. Consequently, many more write operations on the SSD partition occurred for intermediate results compared to query 6.

Fig. 12 shows the initial phase of query 9 (up to 100 s of the whole query process). In this graph, we separate the I/O requests issued on the SSD and HDDs to trace how the SIB method reacts to the change in the I/O pattern. The shuffle writes were performed simultaneously with read operations on the SSD. The shuffle writes were intermittently issued to the SSD, which caused the bandwidth to fluctuatate. As the write operations were issued to the SSD, the mixed read/write bandwidth decreased. Then, the number of bypassed read I/O requests increased. Therefore, the overall query processing performance was maintained even when mingled read/write patterns occurred.

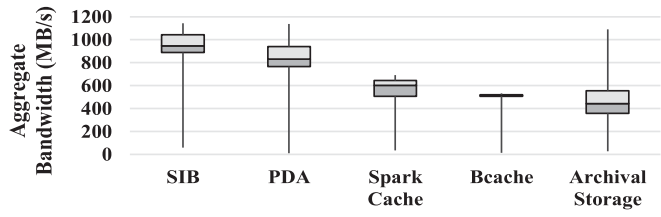
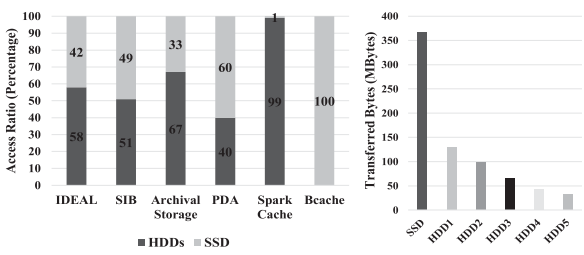


Fig. 10. Aggregate bandwidth variation for TPC-H Query 6.



(a) Transferred bytes ratio between HDDs and SSD for each case

(b) Access ratio between disks for archival storage

Fig. 11. I/O load distribution between SSD and HDDs.

4.2.6 Scalability Evaluation

We conducted an additional experiment to prove the scalability of our approach by gradually increasing the number of datanodes. For every case, namenode was always run on a separate server. We used a 1 TB dataset for 8 nodes, 500 GB for 4 nodes, and 200 GB for 2 nodes. Fig. 13 presents the arithmetic mean of the execution time for the 22 TPC-H queries for each approach. By conducting this experiment, we confirmed that the ratio of performance gain is maintained even when the number of nodes increased, because the data between nodes and between the disks in a node were still evenly distributed.

4.2.7 Effect of Changing Observation Interval

In our approach, kernel I/O statistics were periodically collected by the background kernel thread. The collection interval could be set to

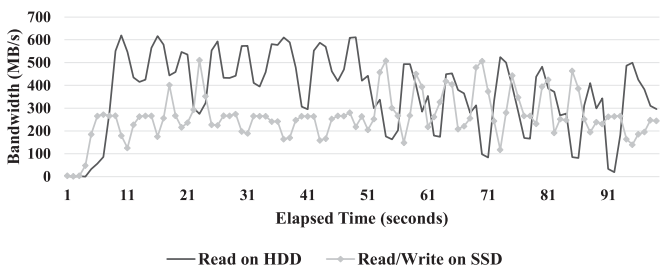


Fig. 12. Mixed read/write bandwidth variation of TPC-H Query 9 with SIB.

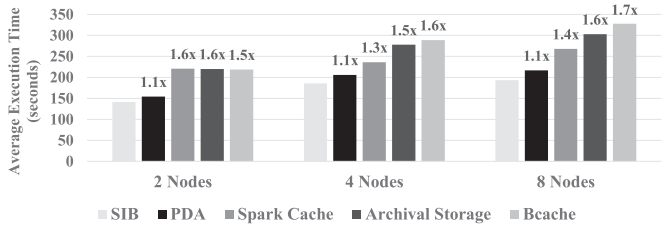


Fig. 13. Average execution time for 22 TPC-H queries. The lower is the better.

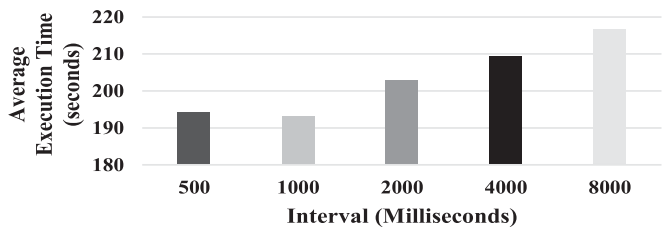


Fig. 14. Effect of various intervals on 22 TPC-H queries.

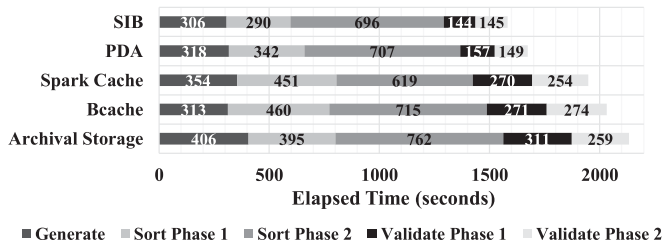


Fig. 15. Execution time of TeraSort for a 1 TB dataset.

any appropriate time. Fig. 14 shows that a longer interval prevented the SIB method from detecting a change in the workloads. When the interval was 500 milliseconds, the result was almost the same as that with a 1000 milliseconds interval. Because the basic data unit of the TPC-H benchmark on big data analytics is the HDFS block, it requires more than a second to be performed on disks. Thus, the workload may change in seconds. We empirically set the interval to a 1000 milliseconds based on the experimental results.

4.3 Performance Evaluation on TeraSort

We confirmed that our proposal could be applied to the HDFS based parallel distributed processing frameworks such as Spark. We performed TeraSort for a 1 TB dataset and confirmed that the SIB method could improve the performance of the map phase (sort and validate phase 1), as seen in Fig. 15. Note that we measured the elapsed time persisted on SSD for Spark RDD persistence. In the sort phase 2, the Spark reads the shuffle results and wrote sorted blocks on the HDFS. Because the SIB method, original bcache, and PDA are write-through SSD caches, heavy writes on the HDFS led to a worse performance. However, the SIB method was better in total performance.

5 Discussion and Conclusion

In this paper, we propose a SIB method that automatically balances the load using I/O statistics to maximize the overall performance of SSD caching. However, there is no way of measuring the flash SSD utilization using the kernel I/O statistics. Thus, we also proposed a method to estimate the utilization of a flash SSD. We examined the proposed approach using the TPC-H benchmark on SparkSQL and TeraSort on Spark. The results revealed that the proposed approach showed a well-balanced I/O load distributed among disks and outperformed other approaches.

Recently, many types of research have adopted reinforcement learning to optimization problem and showed good performance. We expect that reinforcement learning can be applied to our approach measuring I/O saturation and adjusting bypass degree.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2015R1A2A1A05001845).

References

[1] Apache Hadoop. [Online]. Available: http://hadoop.apache.org
[2] M. Armbrust, et al., "Spark SQL: Relational data processing in spark," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2015, pp. 1383–1394.
[3] Chen, Yueguo, et al., "A study of SQL-on-Hadoop systems," in *Proc. Workshop Big Data Benchmarks Performance Optim. Emerging Hardw.*, 2014, pp. 154–166.
[4] A. Floratou, U. F. Minhas, and F. Özcan, "SQL-on-Hadoop: Full circle back to shared-nothing database architectures," *Proc. VLDB Endowment*, vol. 7, pp. 1295–1306, 2014.
[5] K. Kambatla and Y. Chen, "The truth about MapReduce performance on SSDS," in *Proc. 28th Large Installation Syst. Administration Conf.*, 2014, pp. 118–126.
[6] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Ann. Techn. Conf.*, 2008, pp. 57–70.
[7] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, "Differential RAID: Rethinking RAID for SSD reliability," *ACM Trans. Storage*, vol. 6, pp. 1–22, Jul. 2010.
[8] Bcache. [Online]. Available: http://bcache.evilpiepirate.org
[9] TPC-H. [Online]. Available: http://www.tpc.org/tpch/
[10] B. Schroeder, et al., "Flash reliability in production: The expected and the unexpected," in *Proc. USENIX Conf. File Storage Technol.*, Feb. 22-25, 2016, pp. 67–80.
[11] K. R. Krish, S. Iqbal, and A. Butt, "VENU: Orchestrating SSDs in Hadoop storage," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 207–212.
[12] N. S. Islam, et al., "Triple-H: A hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 101–110.
[13] X. Wu and A. L. Reddy, "A novel approach to manage a hybrid storage system," *J. Commun.*, vol. 7, no. 7, pp. 473–483, 2012.
[14] H. Wang and P. J. Varman, "Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation," in *Proc. USENIX Conf. File Storage Technol.*, Feb. 16-19, 2014, pp. 229–242.
[15] Intel Cache Acceleration Software. [Online]. Available: http://www.intel.com/content/www/us/en/software/intel-cache-acceleration-software-performance.html
[16] Sandisk Flashsoft. [Online]. Available: https://www.sandisk.com/business/datacenter/products/flash-software/flashsoft
[17] Flashcache. [Online]. Available: https://github.com/facebook/flashcache/
[18] EnhanceIO. [Online]. Available: https://github.com/stec-inc/EnhanceIO
[19] Iostat. [Online]. Available: http://linux.die.net/man/1/iostat
[20] Finch, T., "Incremental calculation of weighted mean and variance," *Univ. Cambridge*, vol. 4, pp. 11–15, 2009.
[21] FIO benchmark. [Online]. Available: https://github.com/axboe/fio
[22] Archival Storage, SSD & Memory. [Online]. Available: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/ArchivalStorage.html