Hybrid HBase: Leveraging Flash SSDs to Improve Cost per Throughput of HBase

Anurag Awasthi
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kanpur, India
anuraga@cse.iitk.ac.in

Avani Nandini
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kanpur, India
nadini@cse.iitk.ac.in

Arnab Bhattacharya
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Kanpur, India
arnabb@iitk.ac.in

Priya Sehgal
NetApp Corporation, India
priya.sehgal@netapp.com

ABSTRACT

Column-oriented data stores, such as BigTable and HBase, have successfully paved the way for managing large key-value datasets with random accesses. At the same time, the declining cost of flash SSDs have enabled their use in several applications including large databases. In this paper, we explore the feasibility of introducing flash SSDs for HBase. Since storing the entire user data is infeasible due to impractically large costs, we perform a qualitative and supporting quantitative assessment of the implications of storing the system components of HBase in flash SSDs. Our proposed HYBRID HBASE system performs 1.5-2 times better than a complete disk-based system on the YCSB benchmark workloads. This increase in performance comes at a relatively low cost overhead. Consequently, Hybrid HBase exhibits the best performance in terms of cost per throughput when compared to either a complete HDD-based or a complete flash SSD-based system.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Processing and Optimization*

Keywords

HBase, Flash SSD, Big Data, Cost per Throughput

1. INTRODUCTION

Column-oriented databases have been proven to be well-suited for large database applications including data warehouses and sparse data [1]. Recently, there is a substantial interest in distributed data stores for large chunks of data, specially in the NoSQL domain, such as Google's BigTable [3], Amazon's Dynamo [6], Apache HBase [8] and Apache Cassandra [13]. These are being widely used by several companies and industrial users to store "big data" of the order of terabytes and petabytes on a daily basis. These systems are of *key-value store* type that utilize the column-oriented architecture.

Out of these, we choose to work with HBase for multiple reasons: (i) it is an open-source software and, therefore, easy to modify, (ii) it has been successfully deployed in many enterprises, (iii) it is capable of efficiently hosting very large data with tables having billions of rows and millions of columns including sparse data, and (iv) it has become increasingly popular in recent years and has a significantly large community following.

Traditionally, the column-oriented database systems have been designed considering disk (HDD) as the underlying storage media. This means that generally only random seeks have been attempted to be minimized. The lower latency involved in random reads in comparison to HDDs has drawn attention, and coupled with the reducing cost of flash drives and increasing capacity per drive, several successful attempts have been made for improving query performance by introducing flash SSDs (some well known examples are [7, 16, 20]). The use of flash SSDs as a substitute as well as a complementary storage media for hard disks is also increasing due to their lower power consumption, lower cooling cost, lesser noise and smaller sizes.

However, flash has certain disadvantages as well. While exhibiting good performance for random reads, it suffers in case of random writes. Flash SSDs do not allow in-place updates and requires subsequent garbage collection which results in write amplification and erasures overhead, thereby impacting random write performance. Frequent erase operations also shorten the lifetime of SSDs as flash devices can typically sustain only 10,000 to 100,000 erase cycles. This adversely affects the overall reliability of the SSD drive. Further, the cost per unit capacity of flash SSDs is approximately 10 times that of HDDs.

With such high costs, low density, and low reliability compared to hard drives, it is impractical to completely replace HDDs with flash SSDs in large deployments like databases (100s of terabytes to petabytes of capacity requirement). Instead, practitioners have adopted hybrid solutions consisting of a mix of SSD and HDD with different media serving different purposes – SSDs offering high throughput (measured in terms of I/O operations per second) while HDDs offering high storage capacity. Such hybrid solutions provide good performance at better costs compared to pure HDD or pure SSD systems [12].

The 18th International Conference on Management of Data (COMAD), 14th-16th Dec, 2012 at Pune, India.

In this work, we leverage this hybrid approach to come up with a better cost per throughput solution for HBase columnar database systems. As HBase has a lot of metadata or system components, we try to figure out the relevant items that should be placed in flash as opposed to HDD to yield an attractive cost per throughput. We call this modified HBase as HYBRID HBASE.

Hybrid SSD and HDD solutions come in two forms with SSD used as either as (i) a read-write cache for HDD [10, 24], or as (ii) a permanent store at the same level as HDD [4, 12, 15]. While the first case of using SSD as an intermediate tier between DRAM and HDD seems very simple to use and deploy, it leads to caching problems like redundancy, cache coherency (in case of shared HDD infrastructure), etc. Further, as flash is limited in its erase and program cycles, using it as a cache hurts its lifetime much more, thereby increasing the overall cost per unit capacity of the hybrid solution. Hence, we propose to use SSD as a permanent store at the *same* memory hierarchy as the HDD for our Hybrid HBase.

Any column-oriented database system has two main components residing on storage media: (i) user data components that store the actual data, and (ii) system components needed for user data management that include catalog tables, logs, temporary storage or other components storing information about current state of system, etc. While flash can be used to host both the components, for industrial strength data stores where data sizes are in the order of terabytes and petabytes, it may be infeasible to host the user data components due to impractically large costs. Further, the gain in throughput will depend heavily on access patterns, etc.

Thus, we focus only on hosting the system components of a large key-store data store on flash. In addition to being much smaller in size, system components do not change significantly with different database sizes and access patterns. For example, since write-ahead log is designed to have sequential I/O, it will be accessed sequentially irrespective of whether the update operation is a random update or a sequential update. Also, the size of the write-ahead log remains of the order of gigabytes even under heavy load. Additionally, system components must reside on a *persistent* media so that they can be retrieved after a system crash. This rules out the possibility of hosting them on main memory.

In this paper, we estimate which system components to host in the flash to improve the cost per throughput of the system. We identify the system components for a HBase system and analyze the effects of migrating them to flash both analytically as well as empirically (by performing a thorough benchmarking using the YCSB workloads). Since flash is used to host only a small amount of data, the increase in cost is low, although the improvement in throughput is quite high. Overall, this improves the cost per throughput of the system considerably as compared to a complete HDD-based setup or a complete flash SSD-based setup.

The focus of our proposed system is three-fold: (i) better cost per throughput, (ii) performance independent of access pattern, hit ratio, and size of data, (iii) easy to setup, i.e., easy deployment and migration from standard HBase system. Further, the approach presented is generic and can be applied to other column store architectures after similar analyses.

Specifically, the contributions of this paper are:

- We analyze the significance of the storage media in the performance of HBase. We assess disk and flash as storage media, and compare changes in performance with changes in system cost.
- We propose Hybrid HBase, which uses a combination of HDD (for data components) and flash SSD (for system components), and analyze its performance gain and system cost. The generic analysis can be extended to other column stores for improving the cost per unit throughput.

Parameter	Disk	Flash
Model	Western Digital wd10EARS	Kingston SV100S2
Capacity (GB)	1024	128
Cost/GB	$0.15	$2.00
Random Seeks (/s)	151	1460
Reads (MB/s)	161	307.5
Sequential writes (MB/s)	128	182.5
Random re-writes (MB/s)	63.2	81.63

Table 1: Different parameters of the two storage media.

The rest of the paper is organized as follows. Section 2 presents the required background information needed to understand the idiosyncrasies of flash SSD as a storage media and HBase as a data store. It also briefly describes the related work. Section 3 discusses the feasibility of using flash SSD for hosting the system components of HBase and proposes the Hybrid HBase system. Section 4 describes the experimental setup along with performance comparison of the Hybrid HBase system against a complete HDD-based setup and a complete flash SSD-based setup. Finally, Section 5 concludes and outlines some possible future work.

2. BACKGROUND AND RELATED WORK

2.1 Flash as Storage Media

Hard disk drives (HDDs) are electromagnetic devices that have moving heads that read/write data using rotation of spindles. This enforces a mechanical bottleneck for I/O operations. In contrast, flash solid state devices (SSDs) does not contain any moving parts and provide instant reads. Consequently, flash provides good latencies for random reads in comparison to disks (up to 100 times for enterprise SSDs). Re-writes are slower in comparison to reads due to the *erase-before-write* mechanism where re-writing requires erasing a complete block after persisting all its data to a new location, leading to *write amplification*. This, therefore, results in asymmetric read and write performance. Further, each block can be erased only a finite number of times before it turns into a bad block (non-usable). Due to this erase-before-write mechanism, efficient wear leveling mechanism and garbage collection need to be supported on flash, else some blocks become unusable much earlier than others. Flash can, however, offer good performance for sequential writes. Also, it requires less power consumption. Performance comparison of HDDs versus flash SSDs have been done in [21, 22]. As illustrated in Table 1, the comparison of actual runtime parameters between disk and flash for the models used in our experiments shows the same trends.

2.2 HBase

Apache HBase[1] is an open-source implementation of Google's BigTable [3]. It is a distributed column-based key-value storage system that leverages existing open-source systems such as Zookeeper[2] and Hadoop's Distributed File System (HDFS)[3].

HBase *cluster* has one *master server*, multiple *region servers* and the client API. Zookeeper assists the master server in coordinating with the region servers.

Tables are generally sparse and contains multiple rows containing several columns, grouped together into *column-families*. All

[1]http://hbase.apache.org/
[2]http://zookeeper.apache.org/
[3]http://hadoop.apache.org/

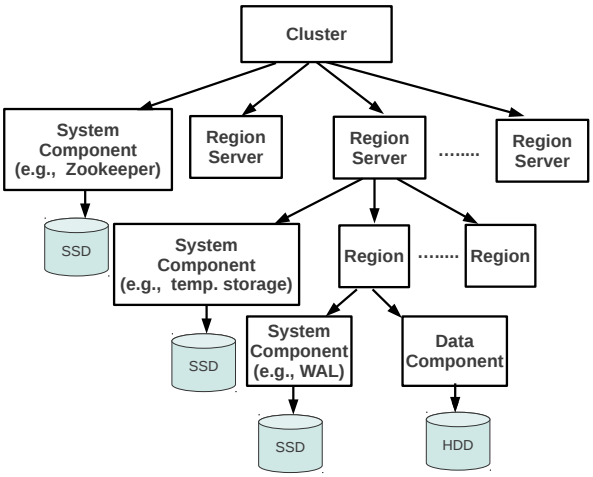


Figure 1: Hybrid HBase setup.

columns of a column family are stored together in sorted *key-value* (ordered by key) format in *store files*. Each store file stores key-value pair corresponding to only one column-family.

Each region server can host several *regions*. A region is a horizontal division of a table and contains store files corresponding to all column-families of that division. A region splits horizontally (based on row key) into two daughter regions if its size grows above a threshold. Therefore, a table is comprised of multiple regions distributed over different region servers.

Each region server also has a *write-ahead log (WAL)* file shared by all its regions. When a write request from a client reaches a region server, data is first written persistently to the WAL and then to the in-memory *memstore*. The write-ahead log is used to retrieve the data after a server crash. After each flush, the write-ahead log can be discarded up to the last persisted modification.

The memstore stores data in a sorted manner, and its size can grow to the order of gigabytes. Once the size of memstore crosses a threshold, it is flushed to disk as a *store file* in a rolling fashion, i.e., HBase stores data residing on disk in a fashion similar to *log-structured merge (LSM) trees* [19], more specifically in "log-structured sort-and-merge-map" form as explained in [8]. According to [8], background compaction of store files in HBase corresponds to the merges in LSM trees and happens on a store file level instead of the partial tree updates. Therefore, HBase uses a *write-behind* mechanism and internally converts multiple random writes to a sequential write for large chunks of data.

To read a key-value pair, first the region server hosting the corresponding region is identified using *catalog tables*. At the region server, first the memstore is searched to see if the required value is present there. If not, then the next level of LSM tree stored persistently needs to be examined. This process continues until either all the levels of LSM trees have been examined or the key is found.

Write involves inserting the updated or new key-value pair in memstore and writing it sequentially to a WAL. Compaction, memstore flush and other such operations happen in background.

Therefore, in HBase, read latencies are higher than write latencies as a read requires first searching the memstore, followed by searching on-disk LSM-trees from the top most level to the bottom level in a merging fashion.

On the administrative side, all the information about regions and region servers are hosted in two *catalog tables* called *.META.* and *-ROOT-*. Zookeeper, which stores information about the region server, hosts the *-ROOT-* table. The *-ROOT-* table gives the address of the server hosting the *.META.* table which, in turn, contains the list of region servers and regions that they are hosting.

A new client first contacts Zookeeper to retrieve the server name hosting the *-ROOT-* table. Afterwards, these catalog tables are queried by the clients to reach the region server directly.

Only when catalog tables are changed due to system crash, region splitting, region merging or load balancing, does the client need to re-establish the connection. It is important to note that for most workloads such events are not too frequent. Thus, the catalog tables are mostly read-intensive entities. Further, although Zookeeper is extremely I/O intensive, it needs only a small amount of persistent data.

2.3 Related Work

Flash SSDs have been successfully used as storage media in many embedded systems and are ubiquitous in devices such as cell phones and digital cameras. Hybrid database systems using both types of storage media (i.e., HDDs and flash SSDs) have also been proposed [12, 24]. In [12], capacity planning technique was proposed to minimize the cost of a hybrid storage media. It uses flash SSDs as a complementary device for HDDs rather than a replacement. Further, in [24], a novel multi-tier compaction algorithm was designed. An efficient tablet server storage architecture that extends the Cassandra SAMT structure was proposed. It was shown to be capable of exploiting any layered mix of storage devices.

In [2], a flash-friendly data layout was proposed that used flash to boost the performance for DRAM-resident, flash-resident and HDD-resident data stores. Flash has also been used as part of a memory hierarchy (in between RAM and HDD) for query processing. In [9, 25], a general pipelined join algorithm was introduced that used a column-based page layout for flash. In [10] flash was used as a streaming buffer between DRAM and disk to save energy.

In order for applications to work transparently to the idiosyncrasies of the flash SSD media, various flash specific file systems have been developed. YAFFS [18] and JFFS[4] are among the most popular ones and are part of the log-structured file system (LFS) [23] class. LFS file systems has an advantage on flash as they log the changes made to the data instead of overwriting it, thereby trading the costly erase operations with increased number of read operations. LGeDBMS [11] used the design principle of LFS further and introduced log structure to flash-based DBMS.

In OLTP systems, significance of flash becomes evident due to the work of [15]. An order of magnitude improvement was observed in transaction throughput by shifting the transactional logs and roll back segments to flash SSD. An improvement by an order of two was also observed in sort-merge algorithms by using flash SSD for temporary tables storage. Further, in [14], it has been shown that flash SSDs can help reduce the gap between the increasing processor bandwidth and I/O bandwidth.

The work presented here is different from others due to multiple reasons. Firstly, there have been attempts to introduce flash in the memory hierarchy between RAM and disk as in [24], but to the best of our knowledge there is no work done for benchmarking the performance of column stores such as HBase with respect to flash SSD as storage media. Secondly, we focus on and explore the feasibility of using flash SSDs at the *same* memory hierarchy as disk for hosting system components. Thirdly, our approach can be generalized for any distributed key-value column-oriented storage system, in particular the NoSQL domain.

[4] http://sourceware.org/jffs2/jffs2-html/

3. THE HYBRID HBASE SYSTEM

In this section, we describe our Hybrid HBase system. The analyses of flash SSDs and HBase done in Section 2.1 and Section 2.2 respectively suggest that it is beneficial to leverage flash SSDs for setting up a HBase system. However, when storage requirements are high, it is not feasible to replace the entire storage capacity of HDDs by flash SSDs. Hence, we focus *only* on the system components of HBase.

3.1 System Components

The major system components of HBase are:

- Zookeeper data
- Catalog tables (*-ROOT-* and *.META.*)
- Write-ahead logs (WAL)
- Temporary storage for compaction and other such operations

In the following sections, for each of the above mentioned system components, we discuss analytically whether hosting it on flash SSD can give any performance boost. Section 4.2 analyzes the empirical effects of putting them on a flash SSD as opposed to a HDD.

3.1.1 Zookeeper

The Zookeeper data component stores information about the master server as well as the region server hosting the *-ROOT-* table, in addition to a list of alive region servers. The client contacts the Zookeeper to retrieve the server hosting the *-ROOT-* table while the master contacts it to know about the available region servers. The region servers report to Zookeeper periodically to confirm their availability. This is similar to a heartbeat keep-alive mechanism and a region server would be declared unavailable if it fails to report. This, thus, makes the Zookeeper very I/O intensive.

The storage requirements for Zookeeper is essentially proportional to the number of systems in the HBase cluster. For most cases, it is very low and is in the order of kilobytes only per system. Hence, it should be beneficial to host it in a flash SSD. However, it cannot be hosted on main memory due to persistency requirements.

3.1.2 Catalog Tables

The catalog tables (*-ROOT-* and *.META.*) are mostly read intensive and are not updated as frequently as the data tables. While the *-ROOT-* table has almost a fixed size, the size of the *.META.* table grows with the total number of regions in the cluster. Nevertheless, their sizes are much less (almost insignificant) in comparison to the data. Thus, these tables are also good candidates for being hosting on flash SSDs. Again, although the sizes of these tables can fit into main memory, they cannot be hosted there as persistency needs to be maintained across system crashes.

3.1.3 Write-ahead-log (WAL)

The write-ahead-log (WAL) is used to simulate sequential writes. Any write is first done on the WAL and it is later committed to the disk in a rolling fashion. The WAL itself is written in a sequential manner as well.

The size of the WAL, unlike the other system components, is not small. The size grows proportionately with the following three parameters: (i) the time after which the WAL is committed to disk, (ii) the rate at which writes happen, and (iii) the size of each key-value pair. Thus, depending on the workload, the size can become as large as gigabytes. This, therefore, rules out the possibility of using main memory. Also, if the WAL resides on a flash SSD, system recovery would be faster after a system crash as data written in WAL could be read faster from SSDs. Hence, it would be productive to host it on flash SSDs.

3.1.4 Temporary Storage

Temporary storage space is used when a region is split or merged. The rows are generally written sequentially in the temporary storage and then later read in a sequential manner again. The size is not expected to be large unless there are many region splits and merges. Combined with the sequential nature of access, introducing flash for temporary storage should, thus, improve the performance.

The above analyses thus suggest that shifting all the four system components of HBase to flash SSDs can yield a better performance at a marginal cost overhead. (Section 4.2 shows the gain in throughput for each system component individually.) This forms the basis of our proposed HYBRID HBASE system. The setup is shown schematically in Figure 1. We next estimate the additional cost of such a hybrid system.

3.2 Additional Cost of Hybrid HBase

The overhead of catalog tables is directly related to the size of the database. If the maximum number of keys per region (as configured by the HBase administrator) is R, then the number of entries in *.META.* is $m = N/R$, where N is the total number of records in the database in a stable *major compacted* state. The *-ROOT-* in turn contains only m/R entries. Thus, we need extra space in the order of $1/R + 1/R^2$ times the user data space. For typical values of R, e.g., when $R = 1000$, this translates to an overhead of only $\approx 0.1\%$.

The space overheads for the Zookeeper and the temporary directory are proportional to the number of systems in cluster and are insignificant in comparison to the total size of the database.

The write-ahead-log (WAL), however, can grow to a significant size, and a flash SSD needs to be installed on each region server. To get an upper estimate of the size of WAL, we observe that in the worst case all the memstores will remain uncommitted and the WAL will keep on growing. Usually there is an upper limit on the size of the memstores and is always less than the heap size allocated to HBase. However, in the extreme case, the entire heap may be used for this purpose (although not recommended), thereby starving other processes. This allows us to estimate the upper limit by the size of the heap allocated for HBase. For our experiments, we used 4 GB of heap and a maximum of 2 GB of memstores before flushing is forced. Even in higher end server machines having 32 GB RAM, if 16 GB is devoted for WAL (which is a high estimate)[5], we only need a flash SSD partition of size 16 GB on each region server. The user data hosted on these machines can be very high (say up to 2-4 TB) without increasing the risk of over-running WAL. Thus, this constitutes the largest system cost requirement. Assuming a 1 TB database and a 8 GB WAL space, the cost overhead is $8/1024 \approx 0.8\%$.

Adding all the system components together, the space overhead grows to at most 1% of the total database size. At an estimate of flash SSDs being 10 times more expensive than HDDs, the extra cost overhead of our proposed Hybrid HBase system for installing flash SSD drives is 10%. Thus, if the gain in throughput becomes more than 10%, then the cost per unit throughput of the hybrid system would be better.

Section 4 extensively discusses the gain in throughput by using flash SSDs. However, before we present the experimental results on how the hybrid system fares vis-à-vis a completely HDD based system or a completely flash SSD based system, we describe our

[5]It is better to flush WAL when the size is small as then the system rollback and recovery are faster after a system crash.

Workload Name	Operations	Access Pattern
A−Update heavy	Read: 50% Update: 50%	Zipfian
B−Read heavy	Read: 95% Update: 5%	Zipfian
C−Read only	Read: 100%	Zipfian
D−Read latest	Read: 95% Insert: 5%	Latest
E−Short ranges	Scan: 95% Insert: 5%	Zipfian/ Uniform
F−Read-modify-write	Read: 50% Read-Modify-Write: 50%	Zipfian

Table 2: YCSB workloads, as published in [5].

model of how the systems are compared according to the cost and the cost per unit throughput measures.

3.3 Metrics for Comparing Systems

We compare the cost of storage media only as this is the sole component varying across different system setups. In addition to a fixed installation cost, there is a maintenance cost associated with each storage media that includes power usage, cooling cost and other such recurring costs. However, since it is harder to estimate them and manage them, in this paper, we only consider the installation cost, information about which is readily available.

To calculate the system cost for a storage media over a given workload, we first estimate the maximum amount of data stored in the device while the workload is running. We also set the device utilization ratio to 80% for HDDs and 50% for flash SSDs as suggested in [12]. The device utilization ratio is important as when the data size grows above it, the performance of the media decreases due to various factors including garbage collection.

Assume that a system setup S uses n storage media. The maximum capacity and the utilization ratio for each of them are $\{D_1, D_2, \ldots, D_n\}$ and $\{R_1, R_2, \ldots, R_n\}$ respectively. Hence, the amount of data that can be stored in a device i is only D_i/R_i. If the price for unit capacity of each storage media is $\{P_1, P_2, \ldots, P_n\}$, the system cost C for the entire setup S is

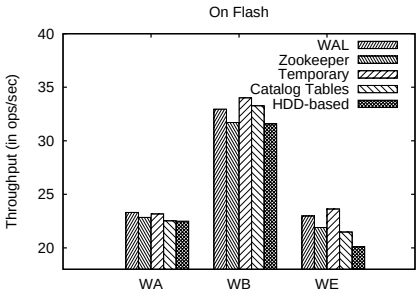
$$C = \sum_{i=1}^{n} (P_i.D_i/R_i)$$

However, due to significant differences in latencies and cost of the three systems (the hybrid one and the two using only one type of storage media), we use the *cost per unit throughput* metric for a fair comparison. If a system having a cost of C achieves a throughput of T IOPS (I/O operations per sec), the cost per unit throughput is C/T.

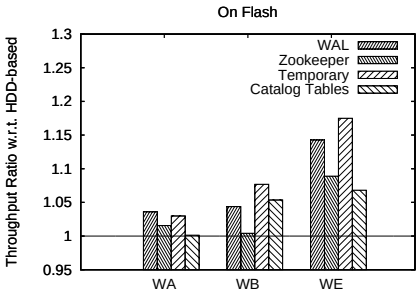
4. EXPERIMENTAL EVALUATION

In this section, we present the experimental analysis and benchmarking of our proposed hybrid system vis-à-vis a complete flash-based system and a complete disk-based system. We conduct the experiments on a standalone instance of HBase (similar to [24]) to completely eliminate the network related latencies. This enables us to better understand the performance and design implications of Hybrid HBase. Since the idea is to analyze performance improvement with respect to storage media, we can expect gain in performance in similar proportions for a distributed environment.

The results are reported for experiments on a system running on an Intel i5-2320 LGA1155 processor (4 cores and 4 threads at 3 GHz) with a total of 8 GB of RAM (4 GB as heap), Western Digital 1 TB HDD, Kingston SV100S2 128 GB Flash SSD, with 64-bit Ubuntu-Server 11.10 as the operating system and ext4 as the underlying file system. We used HBase version 0.90.5 from the Apache repository as the base system. For all analysis and performance evaluations, we used Yahoo! Cloud Serving Benchmarking (YCSB) [5] version 0.1.4. Table 2 shows the six standard workloads (A to F) as identified in [5].



(a) Raw throughputs



(b) Throughput as a ratio with HDD

Figure 2: Throughputs when single system components are hosted on flash SSD.

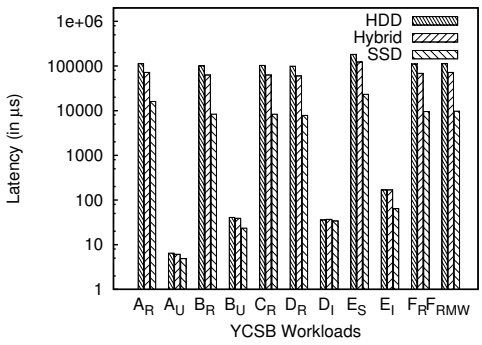
The workloads are composed of Q number of queries (or operations) on R records, and the key generation pattern is decided by three models, namely, latest, uniform and Zipfian. For a workload following a uniform distribution, all records in the database are equally likely to be chosen for the next query. For a Zipfian distribution, some randomly selected keys are hot (more frequently accessed) while most records are rarely accessed for queries. Latest distribution, as the name implicates, reads or writes the most recently accessed key-value pairs with a higher probability.

For our analysis, we used $Q = 10^6$ queries on a database with $R = 6 \times 10^7$ records. Each record is of size 1 KB and the total number of regions in a compact state was found to be 72 (with a maximum region size of $\approx$ 1 GB). We next discuss a few important parameters of the system and the HBase configuration.

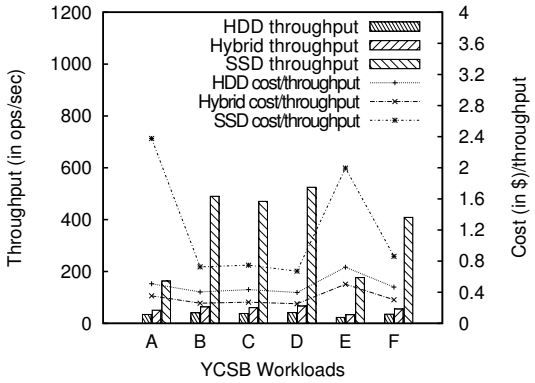
4.1 System Tuning

The benchmarking of any given system involves several variables which must be taken care of appropriately to get the true effect of the desired variable, which in our case, is the storage media. By considering a standalone system, we have removed all external network related issues. We run HBase on a dedicated partition which is different from the operating system's (O/S) partition. The O/S runs on an ext4 HDD partition. Out of 8 GB RAM available, 4 GB had been allocated as heap for HBase and 4 GB had been used by O/S. We also set the swappiness[6] parameter to zero to enable using the entire available RAM. For the ext4 file system, we

[6]Swappiness is the tendency to use swap area in place of RAM in order to reserve some RAM for future processes.



(a) Average Latency[8]



(b) Throughput

Figure 3: Performance over different YCSB workloads.

deactivated the maintenance of file access times done by kernel to further reduce the administrative overheads not needed by HBase.

On flash SSD, we additionally enable TRIM[7] support to reset all flash SSD wear-leveling tables prior to evaluation and maintain a 50% utilization ratio. This minimizes the internal flash SSD firmware interference due to physical media degradation and caching and enhances the flash performance. An unused flash performs very well for the initial read and writes, before reaching a stable lower performance. Hence, we completely fill and empty the flash several times to eliminate this effect. Further, before starting the experiments, we fill SSD completely with some random data so that each query has the same state of flash for garbage collection.

For HBase, automatic major compaction was disabled. We perform major compaction manually and also empty the cache before each experiment to provide the same *data locality*, i.e., the same initial state for both cache and the data layout on disk. The MSLAB [17] feature has been enabled to facilitate garbage collection as well as to avoid lengthy pauses and memory fragmentation due to write heavy workloads. We set the maximum regions per server to 200 and extended the session timeout limit (after which a server is declared dead) to avoid possible server crashes due to delay in responses when the system is subjected to an overload.

[7]The TRIM command specifies which blocks of data in an SSD are no longer used and can be erased.

[8]$X_R \equiv$ Read operation of workload X; $X_U \equiv$ Update operation of workload X; $X_S \equiv$ Scan operation of workload X; $X_I \equiv$ Insert operation of workload X; $X_{RMW} \equiv$ Read-modify-write operation of workload X.

4.2 Single Component Migration

Before we benchmark the proposed Hybrid HBase system, we first assess the effect of migrating one system component at a time. These experiments, thus, measure the effects of hosting each system component *individually* on a flash SSD while the rest three *remain* on the HDD.

We ran half a million (5×10^5) queries on a database having 60 million (6×10^7) records over the workloads WA, WB and WE, i.e., update-heavy, read-heavy and short-ranges. The characteristics of the other workloads are similar to these (WC and WD are both read-heavy and are similar to WB while WF has 50% read and 50% write, similar to what WA also has).

Figure 2 shows the throughputs of the setups (both raw and as a ratio with a completely HDD-based system). The gains in throughput are more pronounced for WAL and temporary storage. Hence, hosting these components on flash SSD is likely to improve the cost per throughput ratio. However, since the space (and therefore, cost) overheads of the catalog tables and Zookeeper are almost insignificant, it is beneficial to host them on flash SSDs as well. These conclusions, therefore, agree with the analyses done in Section 3.1.

4.3 Performance over the YCSB Workloads

Figure 3 depicts the performance of the Hybrid HBase setup vis-à-vis the completely HDD-based system and the completely flash SSD-based system for the different operations on the six YCSB workloads. (As mentioned earlier, for all subsequent experiments, the database consists of 6×10^7 keys and results reported are averages over 3 runs, each having 10^6 queries. Moreover, all the four system components are hosted on a flash SSD.)

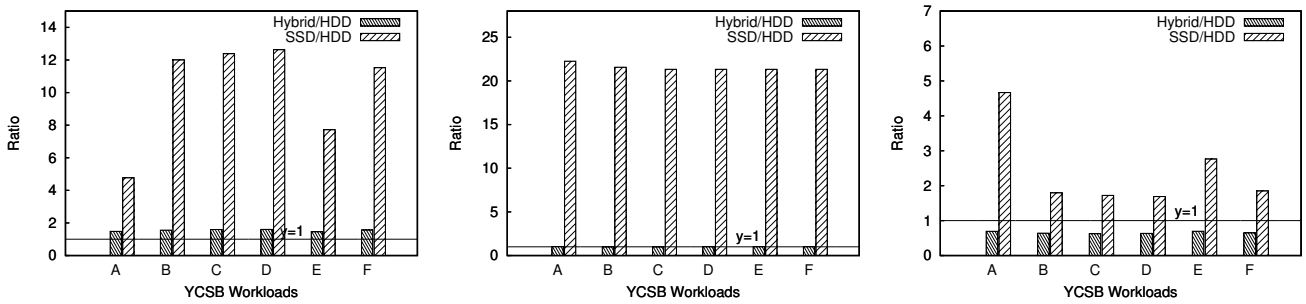
Read latencies of SSD-based setup are significantly lower (approximately 13 times) than both Hybrid and HDD-based setups. These read operations are random reads which are significantly faster for a flash SSD and, hence, the lower latencies. Since the catalog tables (*-ROOT-* and *.META.*) and also the Zookeeper data is stored on SSD in the hybrid setup, read latencies are lower than HDD (approximately 1.6 times). The user data remains on the disk, and therefore, latencies are not as low as SSD.

Average latency for update operation is the lowest for SSD followed by Hybrid and is the highest for HDD. The update operation is similar to a random write, and thus, involves writing to the write-ahead-log (WAL) persistently and storing the updates in memstores to be flushed later. Since WAL is on flash SSD in a Hybrid setup, average update latencies for Hybrid and SSD should be similar. However, due to other background processes (e.g., major compaction and JVM garbage collection) that run faster in SSD, the update latencies for SSD setup are lower.

SSD outperforms Hybrid and HDD setup in scans (sequential reads) moderately as the difference between sequential reads for HDD and flash SSD is not as high as random reads (see Table 1). Average insert latency for HDD, Hybrid and SSDs are also almost similar. Insert operation differs from update operation as during inserts, the size of a region grows and may lead to a region split. A region split also requires updating the *.META.* table. Thus, average insert latency is higher than average update latency over different workloads.

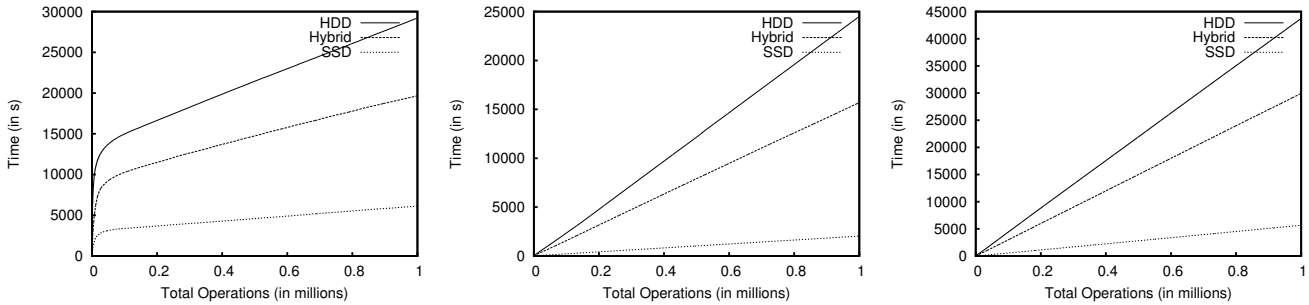
Overall, therefore, as expected, the throughputs of a completely SSD-based system is higher than that of the Hybrid one, which in turn is better than a completely HDD-based setup.

Workload A is an update heavy workload and, hence, the throughputs are lower in comparison to the other workloads. This high variance in overall throughput is in accordance with the asymmetric read/write performance of flash SSDs. Throughputs for workloads having higher percentage of reads are larger in comparison to



(a) Throughput ratio (b) Cost ratio (c) Cost per throughput ratio

Figure 4: Relative comparison for different setups.



(a) Workload A (b) Workload B (c) Workload E

Figure 5: Total time taken for YCSB workloads.

workloads having no random reads (WE) or higher percentage of random writes (WA).

4.4 Performance Ratios with respect to HDD

Figure 4 shows the different performance ratios of the Hybrid and the completely flash SSD-based systems as compared to the completely HDD-based setup. The performance metrics are throughput, cost and cost per throughput. Even if the SSD-based setup gives the highest throughput for all the workloads, the cost per throughput is worse as compared to a Hybrid setup. In fact, due to the high costs of flash SSDs, it is worse than even a fully HDD-based setup. The $y = 1$ line is shown in Figure 4 to mark the base HDD-based setup.

The throughput ratio between Hybrid and HDD setups is around 1.75 for all workloads. This leads to a lower cost per throughput ratio for the Hybrid setup. The cost per throughput ratio for Hybrid setup is below 1 (approximately 0.66 for all workloads).

The difference between cost per throughput of HDD-based and SSD-based setups is even larger for workloads A and E, thereby indicating that flash SSDs are not so suitable for update heavy workloads or workloads having no random reads. Our proposed Hybrid HBase setup exhibits the lowest cost per throughput ratio for all the workloads and can, therefore, be considered the best on this criterion.

4.5 Progressive Running Time

Figure 5 shows the progressive running time for the different workloads as more queries arrive (workloads C, D and F are not shown as they exhibit similar effects). The SSD setup always performs better than the Hybrid one which in turn outperforms the HDD setup consistently.

We next measure the effect of introducing flash SSDs for garbage collection and the CPU performance.

4.6 Garbage Collection

Figure 6 shows the behavior of Java garbage collector over the three different experimental setups. The freed memory per minute is the highest for SSD setup followed by Hybrid. However, accumulative pauses are also the highest for SSD setup. Accumulative pauses are significantly larger for workloads involving updates/inserts. Thus, memory fragmentation is highest for SSD setup which further increases if an update heavy workload or an insert heavy workload is applied. Accumulative pauses due to garbage collector are higher for HDD setup in comparison to Hybrid setup. This is due to the fact that system components on flash in a Hybrid setup requires very less frequent random writes, and hence, there is less memory fragmentation and less garbage collection time.

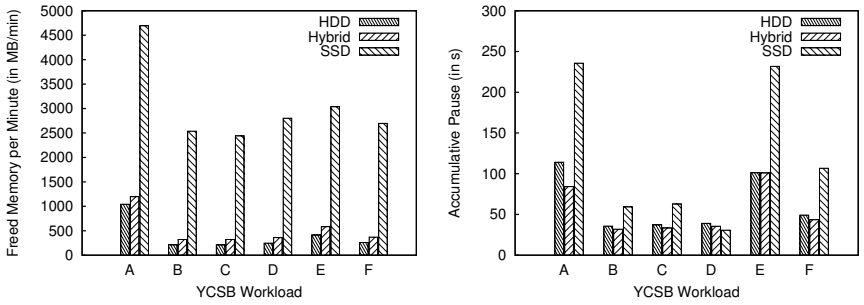
4.7 CPU Performance

Figure 7 shows the CPU utilization over the three different setups for the workloads A, B and F (others are similar to WB). CPU utilization for Hybrid setup is slightly larger than HDD setup. The CPU utilization is highest for the SSD-based setup as flash SSDs narrow the gap between I/O bandwidth and processor bandwidth. Variation of CPU utilization in WA for SSD is high as it is an update heavy workload and requires running garbage collector more frequently, thereby increasing the CPU utilization significantly.

4.8 Effect of Database Size

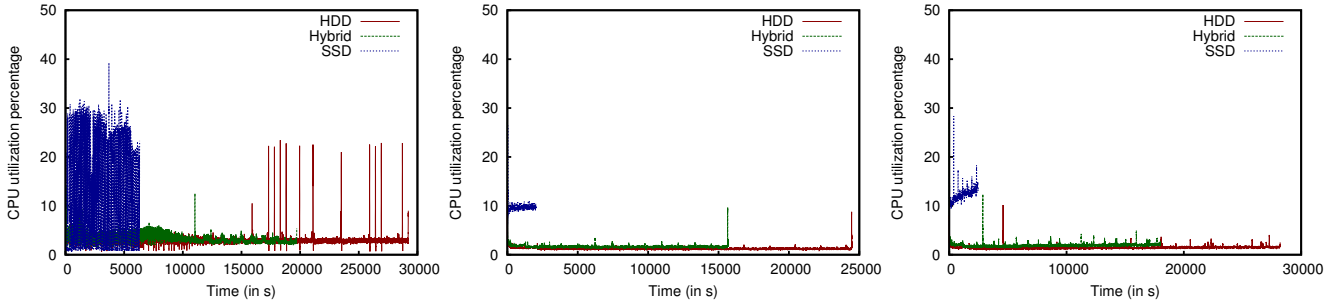
The next set of experiments assess the impact of database size on the storage layer in the standalone system. We vary the number of records in the database, R, for $R = \{2, 4, 6, 8, 10\} \times 10^7$. Due to space limitations, we proceed only up to 6×10^7 records for a completely flash SSD-based setup. Figure 8 to Figure 13 show average latencies for all operations and overall throughputs for the six workloads A to F.

For workload A, with the increase in number of records, read latency also increases for all the setups. However, as shown in Figure



(a) Freed memory per minute (b) Accumulative pauses

Figure 6: Effect on garbage collector over YCSB workloads.



(a) Workload A (b) Workload B (c) Workload F

Figure 7: Effect on CPU utilization over YCSB workloads. (Please see the soft copy version for better visualization of colors.)

8a latency increases faster for HDD setup in comparison to Hybrid setup. As number of records increase, number of regions increases as well. This leads to more accesses to *-ROOT-* and *.META.* tables which are hosted on flash SSD in a Hybrid setup. Hence, although initially with 2×10^7 records, read latencies of Hybrid and HDD setup are comparable, for larger sizes, there is a significant difference between them. Read latency of SSD is very small in comparison to other two setups as random reads are much faster on SSDs. The same behavior is shown for read latencies in workloads B, C, D and F and scan latencies in workload E.

To compare update latencies, it should be noted that while updates to a single region are sequential, those to multiple regions are random. Hence, if incoming updates/inserts are distributed across multiple regions, the random write characteristic aggravates. Update latency for workload A and B increases moderately with increasing number of records as shown in Figure 8b and Figure 9b. The update latencies for workload B is higher for all the three setups as there are only 5% update operations as compared to 50% in workload A. Since the update operations are distributed over all the regions, and the number of regions remain approximately equal for both workloads, this results in more random writes corresponding to each region for workload A. Thus, in an update heavy workload (WA), update latency for all database sizes is comparable owing to the larger sequential write characteristics.

Throughput decreases as number of records increase in all three setups for all workloads. However, as shown in Figure 8c, the change in throughput is maximum for SSD setup for workloads A, E and F. As number of regions increases, writes get more distributed. This results in smaller chunks of sequential write (random writes converted to sequential write for each region when written to new store files) for each region and larger number of such random chunks. Workload E includes insert operations and leads to many region splits. Consequently, garbage collection requirements become higher as well. Thus, the throughput drops rapidly for SSD setup for workloads A and E. The drop in throughput for the workloads B, C and D are less sharper as they are more read-heavy (Figure 9c, Figure 10b and Figure 11c).

For workloads A, E and F, the cost per throughput is the highest for SSD. With increase in number of records, it increases faster than the other two setups as shown by slope of the lines. This happens since the increase in cost is not proportional to the increase in throughput. For workloads B, C and D as well, SSD has the highest cost per throughput, but the difference with SSD is smaller as they are more read-intensive.

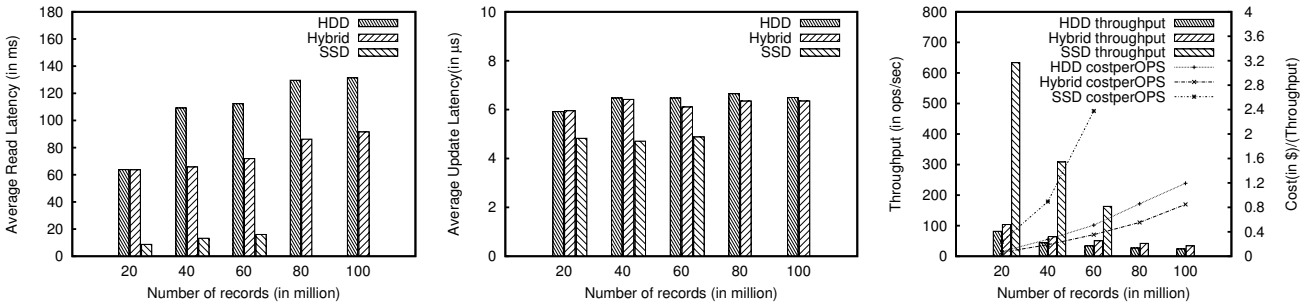
For all database sizes and all workloads, Hybrid HBase has the lowest cost per throughput. This establishes the benefits of our proposed system.

4.9 Effect of Access Pattern

We next evaluate effect of access pattern for workloads A to F. The results are reported in Figure 14 to Figure 19.

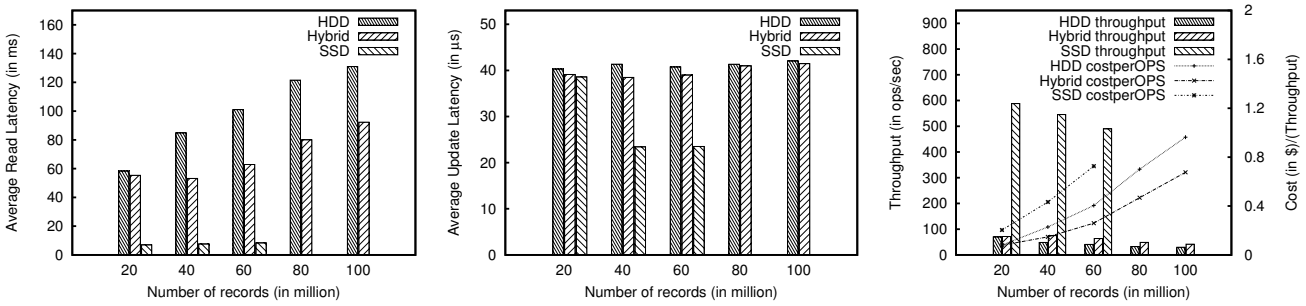
Update latencies for the uniform access pattern are higher as compared to the other access patterns since they are distributed to a larger number of regions. To understand this better, consider the scenario where there are 5000 write operations. If these are distributed over 10 regions, then there are 10 chunks of sequential writes each containing 500 write operations. However, if these operations are distributed over 100 regions (as is more likely for a uniform access pattern), then there are 100 chunks of sequential writes each containing 50 write operations. The first will always be favorable for both HDDs and flash SSDs.

In a uniform access pattern, insert operations lead to lower number of region splits as all regions grow equally. However, in a Zipfian or latest access pattern, insert operations will happen more frequently on a few regions, thereby resulting in more frequent region splitting. Thus, in spite of having a more random write effect in



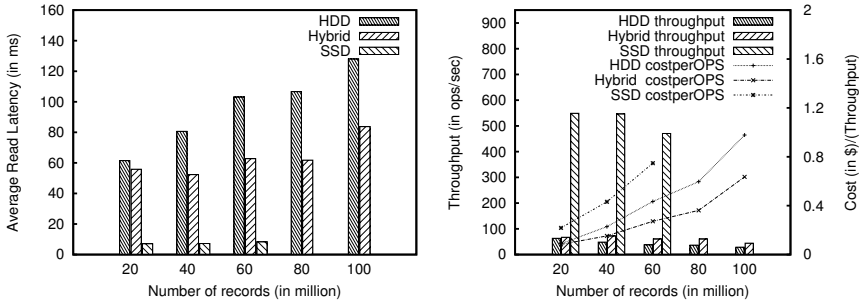
(a) Average Read Latency (b) Average Update Latency (c) Throughput

Figure 8: Effect of database size on YCSB workload A.



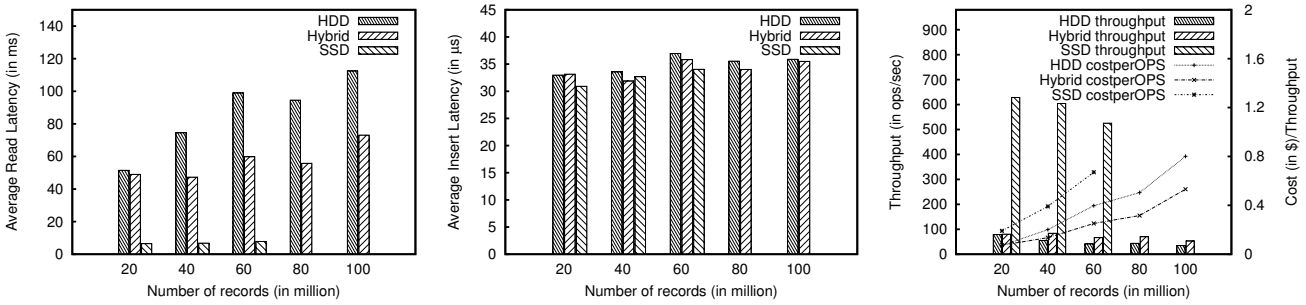
(a) Average Read Latency (b) Average Update Latency (c) Throughput

Figure 9: Effect of database size on YCSB workload B.



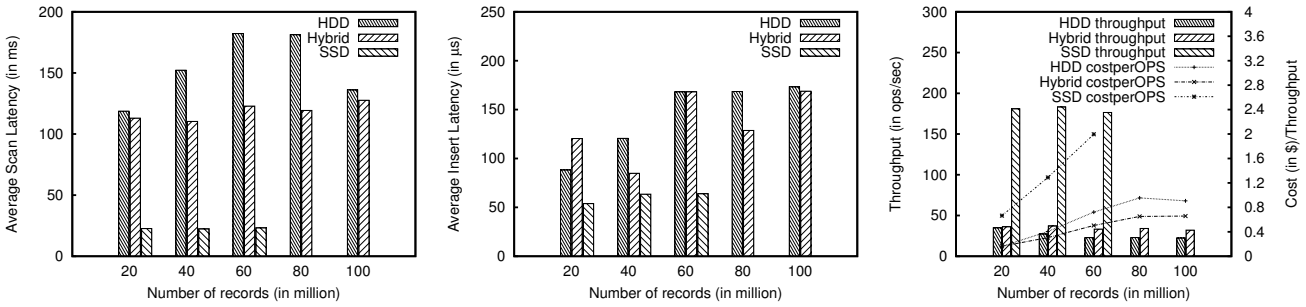
(a) Average Read Latency (b) Throughput

Figure 10: Effect of database size on YCSB workload C.



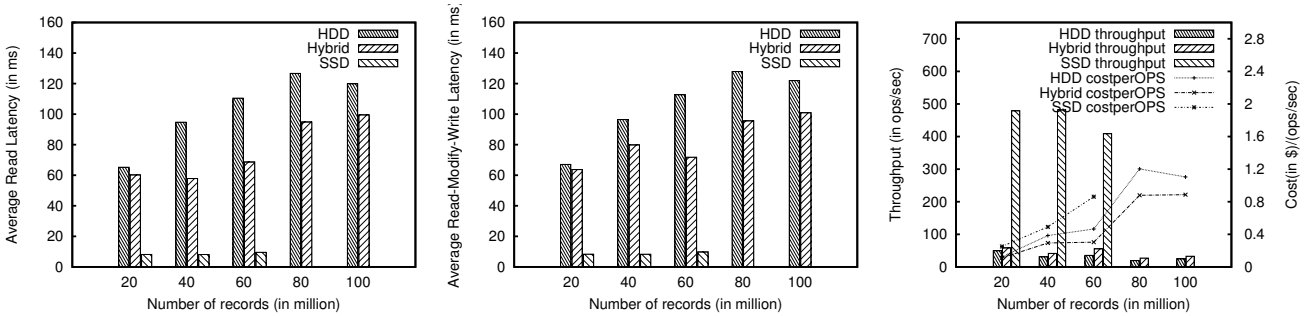
(a) Average Read Latency (b) Average Insert Latency (c) Throughput

Figure 11: Effect of database size on YCSB workload D.



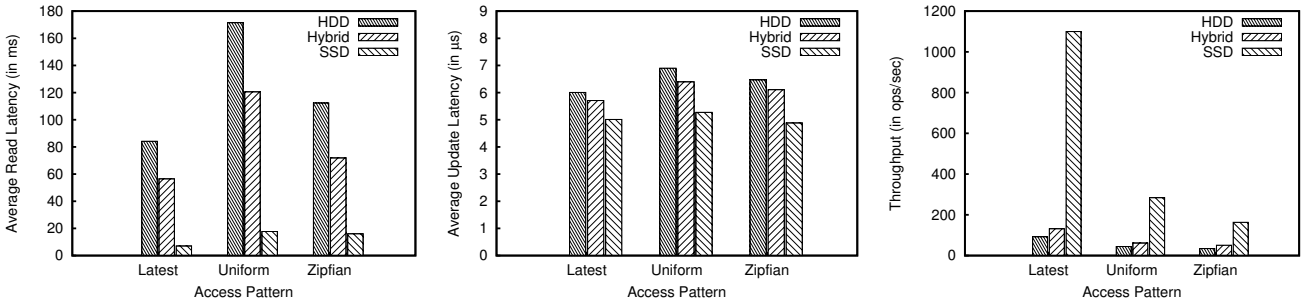
(a) Average Scan Latency (b) Average Insert Latency (c) Throughput

Figure 12: Effect of database size on YCSB workload E.



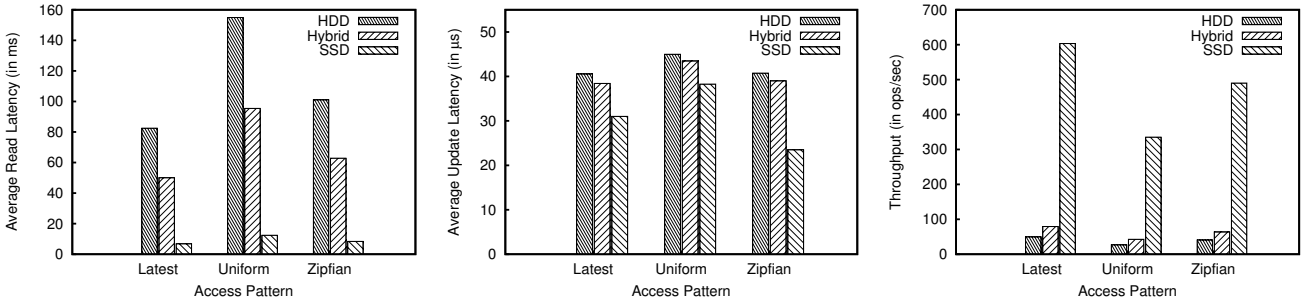
(a) Average Read Latency (b) Average Read-Modify-Write Latency (c) Throughput

Figure 13: Effect of database size on YCSB workload F.



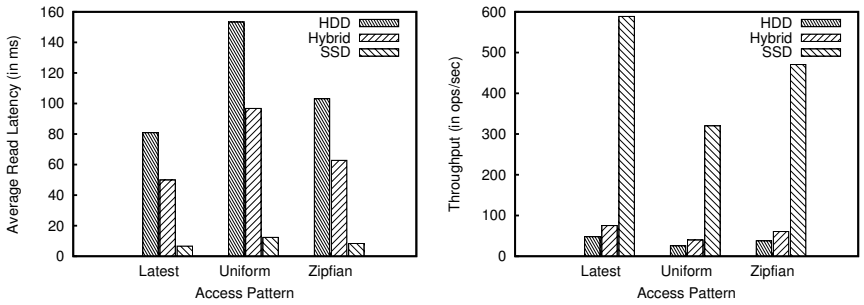
(a) Average Read Latency (b) Average Update Latency (c) Throughput

Figure 14: Effect of access pattern on operation combination of YCSB workload A.



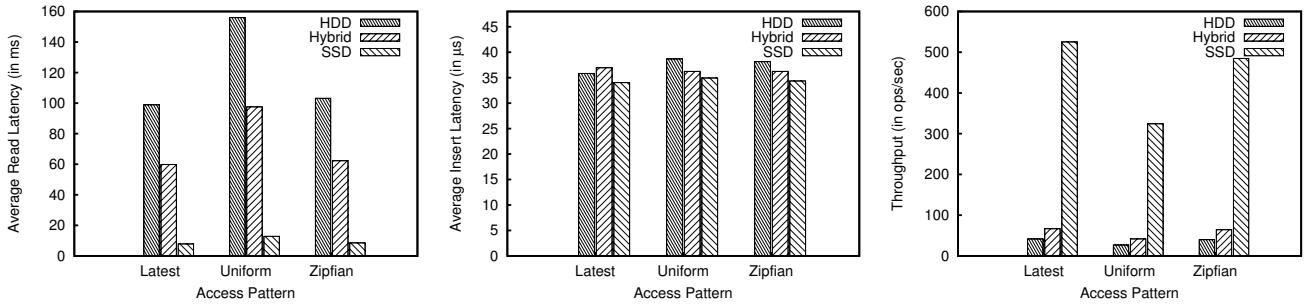
(a) Average Read Latency (b) Average Update Latency (c) Throughput

Figure 15: Effect of access pattern on operation combination of YCSB workload B.



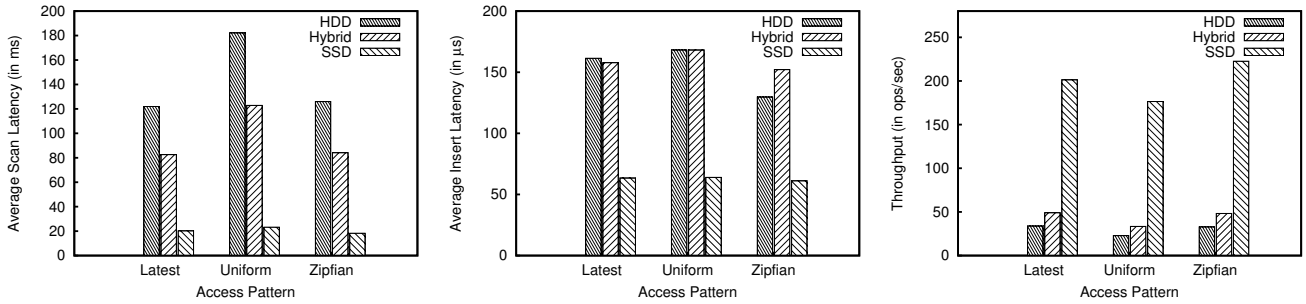
(a) Average Read Latency (b) Throughput

Figure 16: Effect of access pattern on operation combination of YCSB workload C.



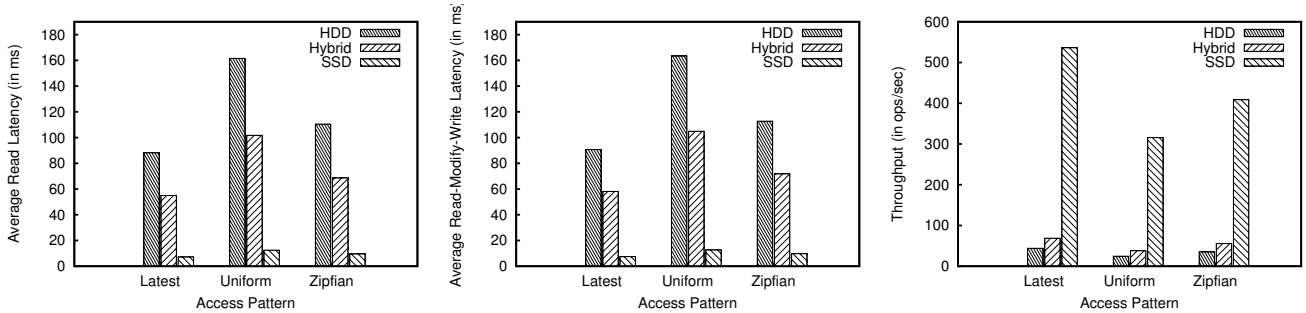
(a) Average Read Latency (b) Average Insert Latency (c) Throughput

Figure 17: Effect of access pattern on operation combination of YCSB workload D.



(a) Average Scan Latency (b) Average Insert Latency (c) Throughput

Figure 18: Effect of access pattern on operation combination of YCSB workload E.



(a) Average Read Latency (b) Average Read-Modify-Write Latency (c) Throughput

Figure 19: Effect of access pattern on operation combination of YCSB workload F.

update access pattern, insert latencies for all access patterns are almost similar, with only slightly higher values for uniform access pattern for all the three setups. If fewer regions are accessed more frequently for read operations, then read latency decreases due to lower cache miss. So, both read (in workloads A, B, C, D and F) and scan (in workload E) latencies are lower for Zipfian and latest access patterns in comparison to uniform. Read-modify-write operation in workload F involves a read and an update operation, and hence, has a higher latency for uniform access pattern.

Hence, throughput of uniform access pattern is the lowest for all workloads. Also, similar to previous analyses, SSD setup provides maximum throughput followed by Hybrid setup for all the tested access patterns and workloads.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we analyzed the feasibility of introducing flash SSD drives for large column store systems such as HBase. Since hosting the entire database on flash SSDs is infeasible due to its large costs, we chose only the system components. We did a thorough qualitative and quantitative assessment (by using the standard YCSB benchmark workloads) of the effects of hosting the four major system components of HBase on flash SSDs.

While a complete SSD-based solution exhibited the best throughput, and a complete HDD-based setup had the least cost, our proposed Hybrid HBase achieved the best performance in terms of cost per throughput. It was shown to be better by almost 33% than the complete HDD setup.

In future, it would be useful to assess the effects of flash specific file systems, if any. Also, we plan to extend our system to a truly distributed setup where network latencies can play an important role. Finally, it needs to be explored whether storing some data components on the flash SSD instead of the HDD can improve the cost per throughput ratio even further, and whether such a setup can be tuned automatically according to the workload.

ACKNOWLEDGMENTS

We thank NetApp Corporation, India for partly supporting this work through grant number NETAPP/CS/20110061.

6. REFERENCES

[1] D. J. Abadi. Columnstores for wide and sparse data. In *CIDR*, pages 292–297, 2007.

[2] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. Flash in a DBMS: Where and how? *IEEE Data Engg. Bull.*, 33(4):28–34, 2010.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Grube. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.

[4] S. Chen. FlashLogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD*, pages 73–86, 2009.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramkrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[7] M. Du, Y. Zaho, and J. Le. Using flash memory as storage for read-intensive database. In *First Int. Workshop on Database Technology and Applications*, 2009.

[8] L. George, editor. *HBase – The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly, 2011.

[9] G. Graefe, S. Harizopoulos, H. A. Kuno, M. A. Shah, D. Tsirogiannis, and J. L. Wiener. Designing database operators for flash-enabled memory hierarchies. *IEEE Data Engg. Bull.*, 33(4):21–27, 2010.

[10] M. G. Khatib, B.-J. van der Zwaag, P. H. Hartel, and G. J. M. Smit. Interposing flash between disk and dram to save energy for streaming workloads. In *ESTImedia*, pages 7–12, 2007.

[11] G. J. Kim, S. C. Baek, H. S. Lee, H. D. Lee, , and M. J. Joe. LGeDBMS: A small DBMS for embedded systems. In *VLDB*, pages 1255–1258, 2006.

[12] Y. Kim, A. Gupta, B. Urgaonkar, P. Berman, and A. Sivasubramaniam. HybridStore: A cost-efficient, high-performance storage system combining SSDs and HDDs. In *MASCOTS*, pages 227–236, 2011.

[13] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *Operating Systems Review*, 44(2):35–40, 2010.

[14] S. W. Lee, B. Moon, and C. Park. Advances in flash memory SSD technology for enterprise database applications. In *SIGMOD*, pages 863–870, 2009.

[15] S. W. Lee, B. Moon, C. Park, J. M. Kim, and S. W. Kim. A case for flash memory SSD in enterprise database applications. In *SIGMOD*, pages 1075–1086, 2008.

[16] Y. Li, S. T. On, J. Xu, B. Choi, and H. Hu. DigestJoin: Exploiting fast random reads for flash-based joins. In *Mobile Data Management*, pages 152–161, 2009.

[17] T. Lipcon. Avoiding full GCs in HBase with memstore-local allocation buffers. http://www.cloudera.com/blog, February 2011.

[18] A. One. YAFFS: Yet Another Flash File System. http://www.yaffs.net/.

[19] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, 1996.

[20] S. Pelley, T. F. Wenisch, and K. LeFevre. Do query optimizers need to be SSD-aware? In *Second Int. Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures*, 2011.

[21] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *3rd Petascale Data Storage Workshop, Supercomputer*, 2008.

[22] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, March 2009.

[23] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log structured file system. *ACM Trans. on Comp. Sys.*, 10(1):26–52, 1992.

[24] R. P. Spillane, P. J. Shetty, E. Zadok, S. Dixit, , and S. Archak. An eficient multi-tier tablet server storage architecture. In *SoCC*, pages 1–14, 2011.

[25] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe. Query processing techniques for solid state drives. In *SIGMOD*, pages 59–72, 2009.