# Improving HBase Write Performance With SSD

Dan Ferguson dan.ferguson@ttu.edu
Prachita Gupta prachita.gupta@ttu.edu
Vinod Yanala vinod.yanala@ttu.edu
Joy Gloria joy.gloria@ttu.edu

## ABSTRACT
HBase is a column oriented data storage system that is built upon HDFS. It is modeled after Google's BigTable and written in Java. It began as a project by the company Powerset to process massive amounts of data for the purposes of natural language search. It is currently a top-level Apache project. Facebook uses HBase to implement its new messaging platform. These applications place very different demands on HBase, both in terms of data size and latency requirements. HBase relies on HDFS for all file read and write operation. In this paper, we propose a new technique to improve HBase performance by introducing SSD (Solid-state disks) into the HBase storage architecture.

## Keywords
HBase, Big Data, SSD, Distributed Computing

## 1. INTRODUCTION
We are in a flood of data today. Data is constantly being generated, not only by the use of Internet, but also by companies generating big amounts of information coming from sensors, computers and automated processes. Significant Internet players like Google, Amazon, Facebook and Twitter were the first facing these increasing data volumes and designed ad-hoc solutions to cope with the situation [10].

But this Big Data analysis is a challenging task. The different aspects of large data sets are:

- Volume: The most visible aspect of Big Data, referring to the fact that the amount of data generated has increased tremendously the past years. The NoSQL database approach is a response to store and query huge volumes of data heavily distributed [11].
- Velocity: Captures the growing data production rates. More and more data is produced and must be collected and utilized in shorter time periods. Millions of devices are added daily to the network, which not only increases volume but also velocity [11].
- Variety: The data ranges from structured information to free text. It is necessary to collect and analyze non- structured or semi- structured data model and query languages. This reality has been a strong incentive to create new kinds of data stores able to support flexible data models [11].
- Value: This refers to the fact that large volumes of data are recorded but not exploited. The challenge is to find a way to transform raw data into information that has value internally or for making a business out of it [11].

The language of data is SQL, so lots of tools have been developed to bring SQL to Hadoop. They range from wrappers on top of Map Reduce to full data implementations built on top of HDFS. Some of these technologies include, Apache Hive, Impala, Presto, Shark, Apache Drill and Apache HBase.

Apache HBase is a scale out database storage system that runs on top of Hadoop platform. The purpose of HBase is to provide low latency random access to data. Google's BigTable, Amazon's Dynamo, Apache HBase and Apache Cassandra are being widely used by several industries to store Big Data of the order of terabytes and petabytes. Out of these, we choose to work with HBase for multiple reasons:

- It is an open-source software, hence, easy to modify;
- Has been deployed in many enterprises; and
- Efficiently hosts very large data including sparse data [9].

A solid-state drive (SSD) is a device used to store data making use of integrated circuit assemblies as memory to store data persistently [12]. These devices are capable of producing not only exceptional bandwidth but also random I/O performance. Due to absence of mechanical parts, they are able to respond to requests much faster than rotating disks [1]. They have their own CPU to manage the storage of data and produce highest possible I/O rates. In applications where the I/O response time is crucial, SSDs are effective.

To improve HBase performance, we propose a new technique where a SSD handles the write-ahead-log (WAL) and initial memStore flushing. This SSD will be the only place memStore flushes are written to and where the WAL will be written so as to improve write speeds and HFile compaction. Forcing memStore flushes to SSD allows faster random read access and smaller block sizes, which saves decompression time and Bloom filter performance. Additionally, since HBase stores all data as immutable data types, writes to SSD disks will be continuous providing optimal wear-leveling when matching SSD block and pages sizes to WAL size and memStore.

## 2. HBASE BACKGROUND
The original distributed key-value database design came out of Google's BigTable. The approach Google took was to first relax the ACID constraints. Within BigTable (and HBase we will soon see) ACID properties are only guaranteed on a row-by-row basis. This relaxation of ACID properties allows BigTable to scale across multiple machines with greater ease. After the publish of [1], the popularity of key-value store databases grew quickly and the Apache Incubator project HBase soon became an Apache Top-level project. HBase was implemented in the same way as BigTable, but under the Apache License. HBase is built on top of Apache Hadoop and utilizes Zookeeper for communication. HBase relies on Hadoop's data replication for data distribution and reliability. HBase is designed just like BigTable and is written in Java [1].

Each HBase table is stored as a multidimensional sparse map, with rows and columns, each cell having a timestamp. A cell value at a given row and column is uniquely identified by (Table, Row, ColumnFamily:Column, Timestamp) → Value. HBase has its own Java client API. The tables in HBase can be used both as an input source and as an output target for MapReduce jobs through TableInput/TableOutputFormat.

HBase has its Pros and Cons just as any database solution, so it is not applicable to every problem. However, if an application meets these general guidelines HBase will likely be a good solution for storing application data:

- Adequate data: Since large amount of data winds up on a single node (or two), HBase should be used when there are hundreds of millions or billions of rows. For a few thousand/million rows, using a traditional RDBMS might be a better choice due to the fact that the cluster might be idle majority of the time [14].
- Adequate hardware: Even HDFS doesn't do well with anything less than 5 DataNodes (due to things such as HDFS block replication which has a default of 3), plus a NameNode [14].
- If the application has a variable schema where each row is different, HBase is used. Example: In a modeling exercise using a standard relational schema; when you can't add columns fast enough and most of them are NULL in each row, you should consider HBase [22].
- If the data is a collection of metadata, message data or binary data that is all keyed on the same value, then you should consider HBase should be considered [22].

It is important to remember, HBase is not a replacement for relational database. It does not behave like SQL, have an optimizer, support cross record transactions or joins. So, if we don't want all these features then HBase is a great fit. Also, HBase is CPU and Memory intensive with sporadic large sequential I/O access. On the other hand, MapReduce jobs are basically I/O bound with fixed memory and sporadic CPU. Combining these can lead to unpredictable latencies for HBase and CPU [22].

## 2.1 Writes to HBase

When a write request is received from a client, data is first written to a HLog, which is HBase's version of a write-ahead-log (WAL), and then to the memStore, an in memory object which stores all recent changes, before returning a successful write to the client [2]. There is a maximum size for the memStore (specified by the system admin). When this maximum is reach, a new memStore is created and the old memStore is written to a new HFile. When compaction occurs, the memStore is written to an HFile (SSTable in BigTable). To keep storage requirements as low as possible, these HFiles are combined regularly through two types of compactions, minor and major. A minor compactions combines the memStore and a few other HFiles into one HFile. This compaction removed out of date versions and deleted data and duplicate metadata. Major compactions combine all HFiles into one HFile. This has the same affect as a minor compactions, but on a larger scale and ensures that no deleted or out of date data remains. All data written to HBase is immutable and to ensure no data loss during update, data is written to the memStore with a copy-on-write protocol, which allows for parallel writes [2]. The WAL is shared between all HRegions in an HRegionServer and is written to disk as a HFile for every write. These writes are coalesced to reduce disk seek time. Upon flushing of the memStore to a new HFile, the WAL is bookmarked so that it can be discarded later once all in memory data has been flushed to disk [5]. If the WAL becomes too large a forced memStore flush is executed on all HRegions. This ensures storage is used efficiently and recovery time is minimized in the event of a failure [5].

## 2.2 Read From HBase

Reading from a table requires reading all versions of the requested row from every the HFiles that contains the requested value and the memStore (and replaying an the HLog if necessary) and returning the most recent version (in most cases). This retrieval is optimized by reading the metadata for each HFile before decompressing. This allows HBase to determine which tables contain potential matching data. This metadata is stored in memory for that tablet. Additional metadata is read from each potential HFile which indicates off-sets for each row key requested [2].

## 2.3 Compaction

HBase simply stores files in HDFS. It does so for the actual data files (HFile) as well as its log (WAL). It simply uses Filesystem.create(Path path) to create these. The updates to these files are stored in memory in a sorted buffer called a memStore. So, as write operations execute, the size of memStore increases. When the size reaches a threshold, the memStore is frozen, a new memStore is created, and the old one is converted to an HFile and stored in HDFS. This is known as minor compaction. If this continues without check, we will have arbitrary large number of HFiles. A merging compaction reads the contents of HFile and the memStore, combines them and writes a new HFile. This is known as merging compaction [1].

A major compaction produces an HFile that contains no out of data or deleted data. To achieve this, the input HFile and memStores are discarded as soon as the merging compaction has finished. As we can see, this method involves a frequent need for garbage collection.

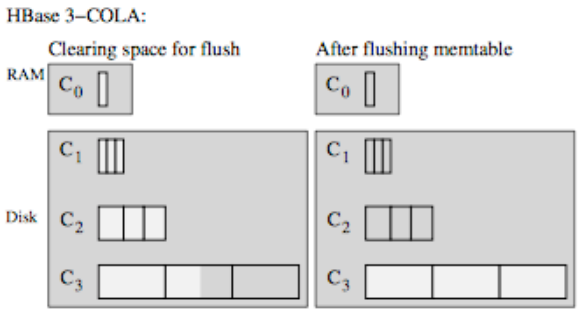



Figure 1: Data placement in a 3-COLA before and after compaction [3]

HBase's compaction algorithm is based on a Cache-Oblivious Lookahead Array (R-COLA). R-COLA can be configured with multiple levels (hence the 'R'), which favors read heavy workloads with higher values of R. HBase implements a 3-COLA configuration, which give the best performance in practice [3]. The cost of insertion in a R-COLA is dependent on the number of compaction levels R, the number of elements inserted N, and the number of key-value byte arrays transferred from memory to disk (either SDD or HDD) and visa versa B. The cost of an insertion follows $(R-1)(\log_R N)/B$ with lookup costs modeled by $\log_R N$. These equations are explained in further detail in [4]; however, looking at Figure 1 we can get an intuitive understanding of this equation's origins. In a 3-COLA configuration each level is shows as $C_0$ through $C_3$ where $C_0$ is RAM and levels $C_1$ through $C_3$ are on disk. On insertion, a check is performed on $C_0$ to determine if that level is full, if so, a compaction is performed on $C_0$ and all subsequent full levels up to and including the first non-full level.

In Figure 1 this includes $C_0$ through $C_3$. After compaction the result is written to the last level compacted, in this example $C_3$. This results in each block of data to be written *R-1* times to each $log_R N$ level; hence the equation [3].

# 3. STORAGE TYPES

## 3.1 SSD

A solid-state drive (SSD) is a device used to store data making use of integrated circuit assemblies as memory to store data persistently [16][17][18]. It is named as a drive though it does not contain an actual disk or a motor, which helps to spin a disk. Electronic interfaces, which are compatible with traditional block input/output hard disk drives, are used which enables simple replacement in common applications. To address specific requirements of the SSD technology new I/O interfaces like SATA (Serial ATA Express) [6] which supports both serial ATA as well as PCI express storage devices have been designed.

### *3.1.1 SSD Properties*

Unlike the traditional electromagnetic disks such as the hard disk drives and floppy disks, which contain moving components, the SSDs have no such mechanical moving components. The SSD components include a controller and memory to store data. The controller is the embedded processor and performs functions like Error-correcting code, read write caching, collecting garbage and encryption. It is one of the major factors for the SSD performance. SSDs perform faster compared to the traditional hard disks as they have their own CPU to manage the storage of data and produce highest possible I/O rates. In applications where the I/O response time is crucial, SSDs are effective.

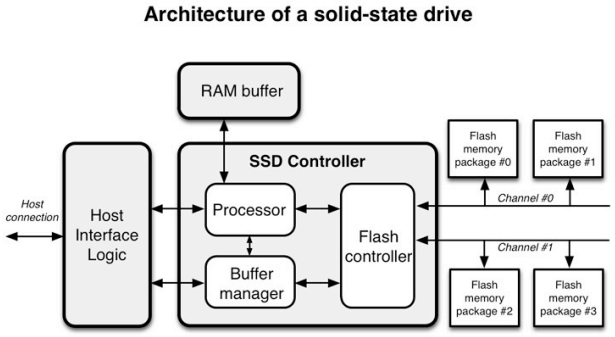



Figure 2: High level architecture of a generic SSD[1]

1: Via http://codecapsule.com/wp-content/uploads/2014/02/ssd-presentation-02.jpg

SSDs also allow parallel and random read access to non-volatile storage and offers many advantages over Hard Disk Drives (HDDs) as outline in Section 3.2. The parallel access can be seen in Figure 2, where the controller has access to each flash chip via the channels illustrated. This increase in over all performance can be seen in [3] where SSD is used as a secondary cache between the memStore and HDD; however, additional performance improvements can be gained by allocating SSD resources for specific HBase processes.

### *3.1.2 Flash Chips*

To persistently store data, solid-state disks make use of flash chips. SSDs store data in a matrix of storage cells. Multi-Level Cells (MLC) [10], can store more than one bit per cell. The process of reading from flash chips can only be done by page wise where the size of the page is about 1-4KB. The time taken per page is around 50 microseconds. When compared to reading, the writing process takes 10 times more computing cycles than reading, which is 500 microseconds and also consumes higher voltage. On the other side each cell should be erased if the new values are to be written on cell. Erasing is even costlier than reading and writing, which can only be done at the block level instead of individual pages. It consumes about 1000 microseconds, which is approximately 20 times the time taken for reading. Grouping of flash chips can be done which are called planes to achieve increase in storage capacity. These planes can be accessed in parallel to increase performance.

### *3.1.3 Flash Translation Layer (FTL)*

On the top of the flash chips or planes there is a Flash Translation Layer (FTL), which provides a block device interface to upper layers. This makes the SSD similar to a common storage disk. FTL manages garbage collection and data mapping to logical blocks and sectors, which allow it to interface with existing hardware that was designed to work with HDDs. To avoid the erasing of hot spot areas repeatedly, page mapping is used on every new write to direct the logical page accesses to various physical locations. This technique helps to improve the performance by saving erase cycles. When the device is in idle state a garbage collection method can be implemented which enables the SSD to perform erase and cleanup processes. Similar to the traditional hard disks, solid state drives have a internal DRAM cache which helps to buffer write requests and store pages which are fetched previously. The drawbacks in flash chips can be avoided by making use of FTL. The FTL is an important part of the SSD, which plays critical role to in providing performance of SSD.

## 3.2 HDD

HDD stands for hard disk drive, which is the most common storage device used to store and retrieve information. It makes use of rapidly rotating disks, which are coated with a magnetic material [19][20]. The circular disk on which the magnetic data is stored is called the hard disk drive platter [19]. A thin film of ferromagnetic material is magnetized on the disk to record data. The binary data bits are represented by sequential change in direction of magnetization. A hard disk generally consists of a spindle, which holds circular disks used to store data. The circular disks also called as platters are coated with a layer of magnetic material.

## 3.3 SSD vs. HDD

The traditional hard-disk drive (HDD) is used commonly as non-volatile storage, which allows the data stored to be retrieved even after loss of power. As mentioned above the metal platters have a magnetic coating which stores the data [8]. There is read/write head on the arm, which accesses the data when the disks spin. A solid-state drive (SSD) has the similar functionality but works in a different way. It has interconnected flash memory chips that retain data. These differ from the regular USB drives in memory type, speed, and expense when compared for the same capacity. In case of form factor the SSDs have got better form factor compared to HDDs. SDD is a more recent technology than the HDD, which was first used in 1956. HDDs are less expensive than SDDs considering the same capacities. In case of boot time a PC equipped with SSD takes seconds to boot and HDD is generally slow. I case of HDD large files can get scattered around the disk platter leading to fragmentation which is not seen in SDD as there

is no physical read head. SSD has no moving parts which gives a chance for increase in durability and also make no noise at all.

# 4. CURRENT APPROACHES

## 4.1 Hybrid HBase

Column-oriented database systems like BigTable and HBase have been developed to store large key-value databases considering HDD as storage media. Hybrid HBase [9] is an attempt to improve cost per throughput by introducing SDDs for HBase. As SSDs are expensive it is not feasible to store the entire database on flash chips. It has advantages like consuming less power, lower cost for cooling, lesser noise and have smaller form factor. The Hybrid HBase takes advantage of high throughput of SSD and high storage capacity of HDD. When compared to the complete HDD setup the hybrid setup has an improvement of 33% in performance. The components of HBase [9] are Zookeeper, Catalog tables, Write-ahead logs, and Temporary storage for compaction. The Zookeeper stores data about master server and the region server hosting the -ROOT-table. There can be performance boost if the data is stored in a flash SSD as the data is in order of kilobytes. The catalog tables are read intensive and the size of these tables are less compared to the data which therefore can be hosted using SSDs. Using SSD in case of write-ahead-log (WAL) can make the system recovery faster after a crash occurs as data on WAL can be read faster from SSDs. Using SSD in case of all components gives a performance boost but considering the cost under consideration the Hybrid HBase gives the best cost per throughput.

## 4.2 HybridStore

Another current solution is HybridStore, which uses SSDs as well and can be combined with other systems effectively. [15] talks about a new hybrid system that combines HDDs and SSDs and uses the complementary properties of the two to create an improvement on performance while staying under a certain cost budget. HybridStore has many interactions between various components and as shown in Figure 3, it is designed to provide a capacity planner called HybridPlan and a dynamic controller called HybridDyn. [15].

### *4.2.1 HybridPlan*

The HybridPlan is an improved capacity planning technique provided for administrators. It includes an overall goal of operating within cost-budget and makes long-term decisions for the given workload depending on resource conditions. The HybridPlan is able to find the most cost-effective storage configuration since it is designed to optimize the cost of equipment needed to obtain the performance and lifetime for workloads that are large in scale.

### *4.2.2 HybridDyn*

On the other hand, the dynamic controller, HybridDyn, is used during episodes of deviations from expected workloads on short time-scales. It improves performance or lifetime guarantees during such episodes and is achieved through two mechanisms: fragmentation busting and write regulation (common data management techniques within SSDs). The HybridDyn system is able to decrease the average response time for workloads that are enterprise scale as well as those that are "random-write dominant." When this is system is compared to a HDD-based system, the time is reduced by 71% [15] and the system operates at significantly finer time-scales. According to the authors of this

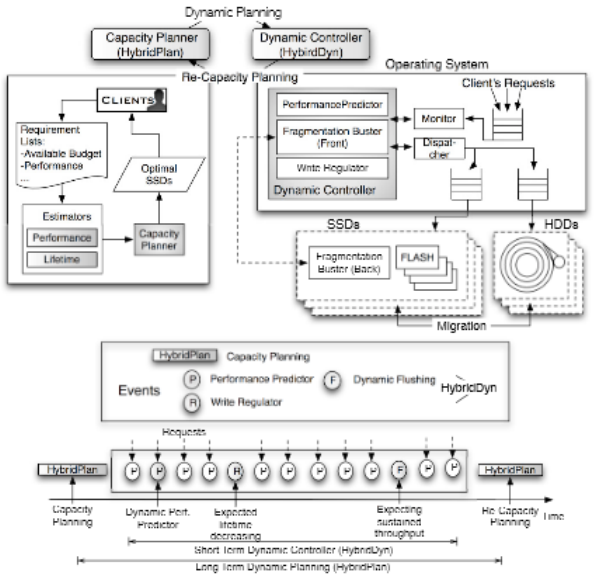



Figure 3: Depiction of various components of HybridStore and how they interact [15].

article, the increase in the time-scale is as large as going from milliseconds to hours. (This percentage is taken from the research and studies that the authors completed.)

#### *4.2.2.1 Fragmentation Busting*

As stated above, the algorithms used in HybridDyn are fragmentation busting and write regulation. The first technique, fragmentation busting, is a flushing methodology that was developed in order to prevent fragmented zones on flash. There are small and random writes on flash which increase data fragmentation and worsens the garbage collection overhead. [15] By flushing a small portion of the small random writes, variation in response times is reduced and performance is improved. Since the mapping tables are not exposed to the outer systems and are only present within the device, flushing requires a co-operation from the device. This means that only the scheduler part of the flushing technique is implemented in HybridDyn. A LRU (Least Recently Used) list of valid pages is maintained to be able to implement the mechanism without having a major impact on HybridStore's performance.

#### *4.2.2.2 Write Regulation*

The second technique used in HybridDyn is write regulation, an essential tool that is used to preserve the lifetime budget requirements. This method deals with the unpredictability in workloads by handling sudden and unanticipated amounts of requests. These extended and/or "recurring period of unanticipated random writes detrimentally impact" on the lifetime of flash. [15] The write regulator monitors the erase rate of blocks and if there are violations due to unanticipated write activities, the monitor starts to regulate the writes being sent to flash. This is done by overriding the decisions created in HybridDyn.

### *4.2.3 HybridStore Shortcomings*

In conclusion, SSDs should be used as complementary devices to HDDS in problems where there is a hybrid system that employs

both HDDs and SSDs. The use of a short-time scale SSD, HDD models, and implementing write regulation to the SSD to make HybridStore demonstrates that combining the two types of data storage is successful. This also shows that combining SSD with another system is possible.

Patterns like sequential reads or scans would not work well for this new system. Since SSD outperforms Hybrid and HDD in these types of patterns with scans and sequential reads, this access pattern in not effective. The reason behind this is because of the difference between the reads for HDD and flash SSD not being as high as random reads [9]. Random reads and random writes are faster for SSD, which in turn, shows that sequential would not be as efficient.

# 5. HBASE WITH SSD

The approach used in this paper seeks to improve write performance by introducing SSD drives into the HBase underlying file system. While HBase relies on Hadoop to handle all file writes and replication, we assume here the addition of a new HDFS feature, which allows HBase to specify which drive (or type of drive) a write should occur. Building off the observations made in the HBase background section and [3], we introduce a SSD that handles the WAL and initial memStore flushing. The memStore and WAL will flush and write only to the SSD. Forcing memStore flushes to SSD allows faster random read access and smaller block sizes, which saves decompression time and Bloom filter performance. Additionally, since HBase stores all data as immutable data types, writes to SSD disks will be continuous providing optimal wear-leveling when matching SSD block and pages sizes to WAL size and memStore.

## 5.1 Theoretical Analysis

To analyze this new layout, lets first begin with a basic model. The below equation describes the number disk accesses for each key-value written.

$$N * L_{disk}(B) * (1 + \frac{(R-1)\log_R(N))}{B})$$

Each write of *B* bytes to a table in HBase involves a write to the HLog and the memStore, each of which has a maximum size, *Lmax* and *Mmax* respectively. Writes to HLog are written to disk immediately (writes to the HLog are actually coalesced which will be accounted for later), which produces a single write. Because of how HBase merges and compacts HFiles, each write to HBase incurs an additional *R-1* writes for each level in the R-COLA system, $log_R N$. For each *N* writes, we then incur a latency *L* which varies by the number of bytes written. Finally, writes to the memStore require only writes to RAM, which are magnitudes faster than writes to disk. This allows us to safely ignore RAM access in our model. From this model we can see, that lessening the latency for any of these disk writes will improve write performance.

In R-COLA, each level ($C_0$ to $C_3$ for the case of HBase) increases in size. Lower levels ($C_3$) will involve large continuous read and writes while the higher levels ($C_1$) are smaller. The larger I/O is favored by HDD, while the smaller reads can be favored by the SSD. In addition, because SSD is ideal for random read access, the memStore dump can be partitioned into smaller HFiles to improve row retrieval. These smaller files will allow for even faster performance for row reads and could improve Bloom filter performance by giving higher granularity to each HFile's metadata.

## 5.2 Data Access Patterns

Each application produces a different data access pattern, and some application will perform better under this new architecture than others. Here we layout high-level data access patterns and their affect on this new architecture. Properties of SSD led itself to particular types of data access.

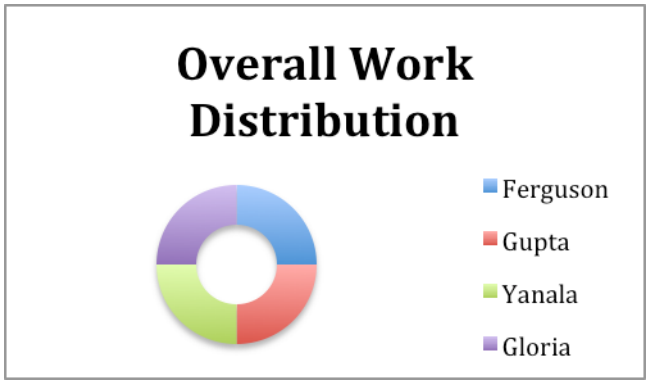
- **Reads:** Row retrieval often involves several random reads, which are faster on SSD and hence have lower latencies. Applications, which tend to read recently changed data, will have improved performance under this new architecture.
- **Writes:** Write operations are sequential for the WAL and memStore due to immutable data types. The WAL HFile can be pre-allocated and written continuously to the SSD drives (producing optimal wear leveling). Flushing of the memStore produces the same write pattern. SSD garbage collection is reduced since updates to existing HFiles never occur allowing entire pages or blocks to be flashed without rewriting data.

Just as HBase is not applicable to every application so is our proposed design.

# 6. CONCULSION

By looking at the properties of HBase and SSD as well as current approaches, it can be concluded that the combination of HBase and SSD can create a new system with possible improvements. Therefore, merging some of the properties of HBase and SSD will provide improvement in performance. Writing the memStore to a SSD disk eliminates the need for garbage collection on the SSD. This is possible since the write to disk is big and is continuous. Therefore, a full page or block can be used in the SSD, which enables progress. One of the main features that will have improvement is the ability to get faster reads from HBase, which in turn saves decompression time. With the constant need to handle big data, a new system with a design like the one mentioned in this paper can provide a possible solution.

# 7. WORK DISTRIBUTION




| Paper Breakdown | |
|---|---|
| Yanala, Vinod | Storage Types section (3) excluding SSD Properties (3.1.1) paragraph 2; and Hybrid HBase section 4.1 |
| Gupta, Prachita | Abstract, Introduction Section (1); HBase |

| | |
|---|---|
| | Background Section (2) 2nd paragraph to end of section; 1st and 2nd paragraphs of Compaction section (2.3); Access Patterns section (5.2), |
| Gloria, Joy | HybridStore Section (4.2); and Conclusion Section (6) |
| Ferguson, Dan | HBase Background section (2) excluding Background Section (2) 2nd paragraph to end of section; 1st and 2nd paragraphs of Compaction section (2.3); SSD Properties section (3.1.1) paragraph 2; HBase with SSD section (5) excluding 5.2 |

## 8. REFERENCES


[1] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4 (June 2008), 26 pages. DOI=10.1145/1365815.1365816 http://doi.acm.org/10.1145/1365815.1365816

[2] George, L. (October 12, 2009) HBase Architecture 101 - Storage [Blog Post]. Retrieved from http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

[3] Richard P. Spillane, Pradeep J. Shetty, Erez Zadok, Sagar Dixit, and Shrikar Archak. 2011. An efficient multi-tier tablet server storage architecture. In Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC '11). ACM, New York, NY, USA, , Article 1 , 14 pages. DOI=10.1145/2038916.2038917 http://doi.acm.org/10.1145/2038916.2038917

[4] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. 2007. Cache-oblivious streaming B-trees. In Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures (SPAA '07). ACM, New York, NY, USA, 81-92. DOI=10.1145/1248377.1248393 http://doi.acm.org/10.1145/1248377.1248393

[5] George, L. (January 30, 2010) HBase Architecture 101 – Write-ahead-Log [Blog Post]. Retrieved from http://www.larsgeorge.com/2010/01/hbase-architecture-101-write-ahead-log.html

[6] "Serial ATA Revision 3.2 (Gold Revision)" (PDF). *knowledgetek.com*. SATA-IO. August 7, 2013. Retrieved March 27, 2014.

[7] Hudlet, V., & Schall, D. (2011, March). SSD!= SSD-An Empirical Study to Identify Common Properties and Type-specific Behavior. In *BTW* (pp. 430-441).

[8] Wei, M. Y. C., Grupp, L. M., Spada, F. E., & Swanson, S. (2011, February). Reliably Erasing Data from Flash-Based Solid State Drives. In *FAST* (Vol. 11, pp. 8-8).

[9] Awasthi, A., Nandini, A., Bhattacharya, A., & Sehgal, P. (2012). Hybrid HBase: Leveraging Flash SSDs to Improve Cost per Throughput of HBase. In *COMAD*(pp. 68-79).

[10] Labrinidis, A., & Jagadish, H. V. (2012). Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, *5*(12), 2032-2033.

[11] Big Data A new World of Opportunities , Nessi White Paper , December 2012.

[12] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. S., & Panigrahy, R. (2008, June). Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference* (pp. 57-70).

[13] Omer. (April 20, 2011) Apache HBase Do's and Don'ts [Blog Post]. Retrieved from http://blog.cloudera.com/blog/2011/04/hbase-dos-and-donts/

[14] Apache Software Foundation (2014) The Apache HBase Reference Guide. http://hbase.apache.org/book/architecture.html#arch.overview

[15] Kim, Y., Gupta, A., Urgaonkar, B., Berman, P., & Sivasubramaniam, A. (2011, July). HybridStore: A cost-efficient, high-performance storage system combining SSDs

[16] "Texas Memory Systems: Solid State Disk Overview". *Texas Memory System Resources*. Texas Memory Systems. Retrieved 14 December 2012.

[17] Whittaker, Zack. "Solid-state disk prices falling, still more costly than hard disks".*Between the Lines*. ZDNet. Retrieved 14 December 2012.

[18] "What is solid state disk? - A Word Definition From the Webopedia Computer Dictionary". *Webopedia*. ITBusinessEdge. Retrieved 14 December 2012.

[19] Arpaci-Dusseau, Remzi H.; Arpaci-Dusseau, Andrea C. (2014). "Operating Systems: Three Easy Pieces [Chapter: Hard Disk Drives]". Arpaci-Dusseau Books.

[20] Santo Domingo, Joel (May 10, 2012). "SSD vs HDD: What's the Difference?". *PC Magazine*. Retrieved November 24, 2012.