ApproxSSD: Fast Data Sampling on SSD Arrays

Jian Zhou, Xunchao Chen, Jun Wang
University of Central Florida, Orlando, FL, USA
{jzhou, xchen, jwang}@eecs.ucf.edu

***Abstract*—With the explosive growth in data volume and the increasing demand for real-time analysis, running analytical frameworks on a subset of the myriad input data has been trending. Such a data sampling technique computes based on a combination of sub-datasets and delivers the results with an acceptable "error bars" at an interactive speed. Furthermore, data sampling is often performed at the application level by selecting data randomly without any knowledge of the lower level data placement. However, for today's widely deployed primary storage - Solid State Disk (SSD), its I/O performance is highly dependent on the data access pattern. Random workloads will result in sever performance degradation for SSDs. In this paper, we propose ApproxSSD, which is a framework that leverages the tolerance of data selection in many applications to perform data-layout aware sampling on the SSD array. Aiming to minimize the read latency, ApproxSSD not only uses data-layout aware sampling to balance workloads on SSD array, but also utilizes delay reflection to avoid occasional contentions. We have developed a prototype system for the ApproxSSD in Scala. Evaluation results show that our prototype system can achieve up to 2.7 speedup compared to Spark and maintain the high output accuracy simultaneously.**

I. Introduction

A. Approximate Data Processing

Data intensive computing frameworks usually handle large amounts of data and require fast data retrieval to support applications, such as intelligent machine learning [21], graph processing [9], [10], [18], [20]. To cope with the explosive growth in data volume and facilitate near-real-time analysis, many works have been done to reduce the input data size and deliver approximate queries. Data movement reduction techniques can be classified into two categories: data transformation and sampling. The data transformation technique utilizes a compressed format to represent the original data, such as using random projection to reduce the dimensionality of matrix [22] and using multi-probe locality sensitive hashing [19] in similarity search [13]. Data sampling, on the other hand, selects a portion of input data (i.e., sub datasets) to process and typically has a lower overhead [4], [16], [29].

An in-memory computing engine [30] has been proposed to reduce the intermediate data access delay in many iterative algorithms (e.g., machine learning and graph processing). However, there is still an I/O bottleneck for big data analysis due to slow external data access and limited DRAM capacity. To be more specific, first, the datasets for big data applications are often too large to fit into memory. To process the data as a whole, applications have to repeatedly switch the data between memory and external disk array [6]. Second, large iterative algorithms generate a vast of intermediate data which will exhaust memory space. DRAM memory capacity, on the other hand, is limited by various constraints, such as power consumption and price per gigabyte. As DRAM is insufficient, data partitions are evicted to the disk based on the replacement policy and reload to the memory as needed [30]. However, accessing lower level disks will introduce significant performance penalty. As a result, data sampling is becoming a promising technique to shrink the data size and enable the in memory processing.

The motivation of data sampling is diminishing data access and computation time via taking sub datasets to present the entire input data. Due to its simplicity, the data sampling has increased the deployment of approximate applications. The accuracy of data sampling depends on the data features and the sampling function. For the majority of sampling applications, uniform and independent random data choice [2], [4], [27] is used, while others, such as sparse datasets in a specific field [16], [29], require conditional sampling for better accuracy. Since data sampling can often provide an order of magnitude performance improvement, most of the data sampling research focus on enhancing the output accuracy or the estimation of error bar. However, without the knowledge of low level data layout, data sampling will impose random I/O to the disks, resulting in suboptimal resource utilization, such as CPU and disk waste.

B. The Pitfalls of Deploying SSD

NAND Flash-based Solid State Disks (SSDs) [3] have recently been widely employed in data centers. Without any moving parts, SSDs are expected to provide fast random read accesses [25]. As a result, some literature consider SSDs for random reads dominated applications [15], [31]. Data sampling, which mainly performs random data access by definition, is expected to provide faster approximate results with low energy consumption on SSDs [23]. Unfortunately, in contrast to these intuitive expectations, performance characteristics of the SSD have dramatically changed as its internal architecture becomes increasingly complex, and the firmware employs advanced strategies for address mapping.

As reported, the performance of random read accesses on SSDs is worse than that of other access patterns and operations, including random write accesses [15]. There are two main reasons for this. First, since random read accesses exhibit very low locality, the cache hit ratio in the Flash Translation Layer (FTL) is also very low. Thus, the FTL needs to read many translation pages in order to retrieve the requested address mapping table. This problem only gets even worse when the firmware uses advanced fine-grained address mapping strategies such as page-level FTL [12]. To cope with this problem,

we alleviate the random read access in data sampling and increase the spacial locality by performing semi-random data sampling. Second, random access results in the underutilization of internal parallelism. While sequential accesses can be easily striped over multiple channels in a round-robin fashion to exploit the internal parallelism of SSDs, random read accesses can cause severe resource conflicts when requesting data from the same internal resources (e.g., channel, package, die, plane). The foregoing problem also exists in the SSD based storage array. To cope with this one, we expose the internal parallelism to the sampling process and perform parallelism aware data sampling.

The other problem of introducing SSDs is their unpredictable performance characteristics. The performance of an SSD, even when the workload is sequential read, can degrade with NAND flash cell aging and the increasing amount of I/O requests. This problem is caused by the complex SSD firmware system [3], which consists of multiple modules such as address translation, wear leveling, garbage collection and error block management. The complex firmware stack prevents the run-time SSD performance from being forecasted by the OS. Note here the worst-case latency of fully-utilized SSDs is much worse than that of HDDs due to GC invocations in SSDs [15]. As the overall performance of a flash array is determined by the slowest SSD in the array, we design the dynamic task drop based on the run-time performance of SSDs.

C. Contributions

In this paper, we have made the following contributions:

- A parallel data storage and processing engine is implemented for data-layout aware sampling on SSD array.
- An intelligent task scheduling scheme is proposed to fully release the performance potential of SSDs.
- Preliminary results shows that our prototype can achieve upto 2.7 speedup compared to the state-of-art data analysis engine - Spark.

II. Motivation and Analysis

A. Workloads Balance in Data Sampling

Sampling-based approximate query has been widely used in modern data intensive analytic applications. Sampling obtains a much smaller data set with the same data structure and feature. As the sampled dataset typically can be fit into the main memory, sampling significantly reduce the latency and resource usage. The basic technique used to do data sampling is random data choosing on the entire dataset. Unfortunately, random data access results in imbalanced workloads across SSDs in an array. The overall execution time of data sampling is determined by the SSD with the maximum load. This has been discussed in the classical "balls into bins" problem [5], [26]. We sample m sub dataset on n SSDs. When the data is sampled uniformly, the maximum load on a single SSD might be as large as m. However, the chance to have all the sampled dataset stored on a single SSD is low. For the case $m = n$, the probability of the maximum load $\frac{logn}{loglogn}(1+o(1))$ is $1-o(1)$. For example, while we select 8 sub data from 8 SSD array and ignore other factors such as data placement and fragmentation, the expected best outcome is load 1 sub data from each SSDs. However, the maximum load on a single SSD is around 2.5 with high probability. This is due to the fact that data sampling is performed by the upper level approximate applications or computing frameworks [8], [27], while physical data layout is masked by the storage software such as filesystems or RAID controller. Motivated by this, we propose to expose the internal parallelism inside storage architecture to the data sampling framework. The data layout aware sampling can effectively balance the maximum workload of each SSD in a storage array.

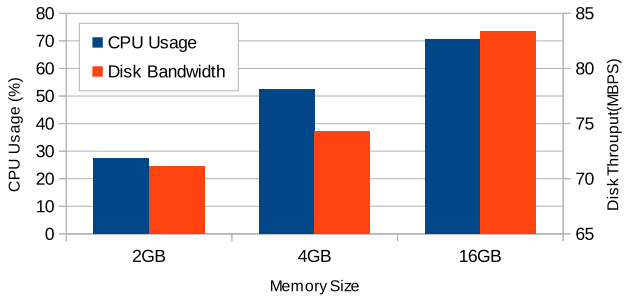

Fig. 1: SSD bandwidth utilization under different sizes of DRAM.

B. SSD Performance Characteristics

In memory computing has been gaining tremendous popularity in the era of big data. It can substantially reduce the disk I/O because the data are stored in DRAM by definition. However, when dataset is large enough, the in-memory computation framework Spark will be outperformed by its counterpart such as Hadoop [11]. For some extremely large dataset, the Spark even crashes with JVM heap exceptions. The challenge of allocating memory space is that the runtime memory usage for applications are often unpredictable. To study the Spark behavior when memory is insufficient, we run "WordCount" under different size of physical memory. 49GB Wikimedia database dumps [1] were utilized as the dataset in this experiment. Fig. 1 shows the experimental results under different size configuration of DRAM. We can see that when memory is insufficient, both disk bandwidth usage and CPU usage significantly decreased. This is because when the DRAM size is too small to load the whole datasets into memory, Spark needs to frequently switch data partitions between disks and the main memory. The "WordCount" application is supposed to has very simple I/O access pattern, which is dominated by sequential reads and writes. Ideally, the disk will work very efficiently under these type of workloads, and the traffic to disk will increase as we reduce the DRAM size. However, as shown in the results, the disk bandwidth usage decreased under smaller memory configuration. This is due to the frequent data switch has introduced a large amount of random access to SSDs. Apart from that, the CPU usage is very low since there is not enough data to saturate the CPU.

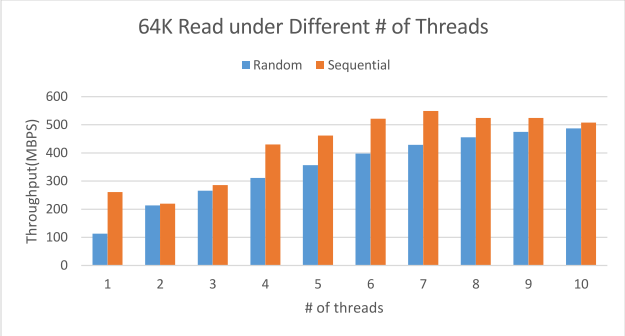

Fig. 2: SSD bandwidth utilization under different number of read threads.

The low performance associated with random access to SSDs has been well studied in literatures [15]. SSDs have multiple channels working independently. Multiple I/O operations can be served on different block windows simultaneously. The block level parallelism of SSDs has been hidden to the data parallel processing system because of the deep I/O stack, such as the file-system and RAID controllers. As a result, the random data access issued by job execution engine can not fully exploit the parallelization and thus causes contention on channels. Another reason for low disk bandwidth usage in parallel data processing engine is the concurrent read and write. Data processing engines exploit the chip multi-core by launching multiple concurrent analysis jobs for each partition to enhance the performance. This results in concurrent read and write on SSDs. The parallel I/O can increase the storage performance. However, concurrent read and write will impair the performance as shown in our experiment [24]. To study the concurrent I/O behavior, we benchmark SSDs with Iometer [14]. In Fig. 2, we can see that with one I/O thread, the random access performance is significantly lower than the sequential access performance. However, if we use multiple threads to perform random access concurrently, random and sequential I/O performance become very close. This is due to that with concurrent read threads, there is a higher chance of saturating multiple parallel storage units (e.g. channel). However, as discussed previously, increasing concurrent read threads of a given disk will unbalance the maximum amount of workload need to be served by each SSD.

C. Error-bar Estimation

We adopt the standard multi-stage sampling theory [8], [17] to compute error bounds for approximate applications. The set of supported aggregation functions includes SUM, COUNT, AVG, and PROPORTION. In particular, we describe the approximation of SUM as an example. Approximation for other aggregation functions share similar process. Depending on the dataset and computation, it may be necessary to use bootstrap methods [2], [7], [17] to generate a lower error bar (i.e., more accurate) by introducing additional re-sampling overhead. Supposing we divide the entire datasets into multiple

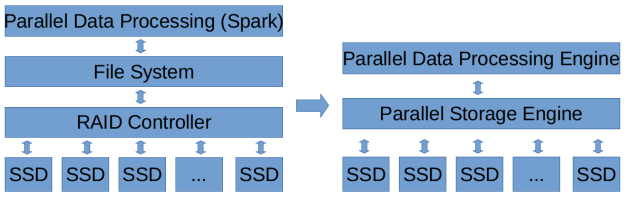

Fig. 3: Reducing I/O Stack.

data partitions, we have a total of T items which are divided into N partitions, and each partition has M_i items so that $T = \sum_{i=1}^{N} M_i$. Each unit in partition i has an associated value v_{ij}, and the sum of these values can be obtained by $\tau = \sum_{i=1}^{N} \sum_{j=1}^{M_i} v_{ij}$. To estimate the sum τ, we sample a list of n partitions which are randomly selected based on their inclusion probability π_i.

$$\hat{\tau} = \frac{N}{n} \sum_{i=1}^{n} (\frac{M_i}{m_i} \sum_{j=1}^{m_i} v_{ij}) \pm \epsilon \tag{1}$$

where the error bound ϵ is defined as:

$$\epsilon = t_{n-1,1-\alpha/2} \sqrt{\widehat{V}(\hat{\tau})} \tag{2}$$

$$\widehat{V}(\hat{\tau}) = N(N-n)\frac{s_u^2}{n} + \frac{N}{n} \sum_{i=1} nM_i(M - i - m_i)\frac{s_i^2}{m_i} \tag{3}$$

where, (s_u^2) is the variance between partitions, and (s_i^2) is the variance within the partition i.

To perform the multi-stage sampling, our data analysis engine divide the job into multiple Map and Reduce tasks. For example, in a program that counts the occurrence of a specific word, we count the word occurrence in each partition in Map tasks and sums the counts in Reduce tasks. The approximation can be performed by executing map tasks on only a subset of partitions. Several data sampling engine has been introduced to MapReduce frameworks [8], [27]. However, the data placement on SSD array is hidden to these data sampling engine. Thus, none of them leverage the relationship between data layout and access pattern to further improve the data sampling performance on the SSD array.

III. Design and Implementation of ApproxSSD

The design details are listed as follows:

A. Simplifying the I/O Stack

Parallel data analysis applications storing large datasets consume significant I/O bandwidth. Clusters of storage servers managed by parallel file systems (e.g., HDFS) promise to provide scalable I/O performance for high-end applications. In these systems, there are many interactions across application, system software and hardware components layers. Current

data-intensive computing frameworks [28], [30] mainly focus on providing an easy to use programming interface and scalable computing power for applications. The interaction between local storage is based on the local filesystem interface. Despite both the computing framework and the underline storage system process data in a parallel manner, there is an isolation between their internal parallelism.

Listing 1: Selected Interface Design of ApproxSSD

```
abstract class Partition[T:ClassTag] extends Serializable {
  var drive:Int
  var offset:Int
  var size:Int
  private var _persist = false

  def persist(implicit drive:Int = 0)
  def cache
  def compute
  def content: Array[T]
}

abstract class LDD[T: ClassTag] extends Serializable {
  def partitions: Array[Partition[T]]

  def map[V: ClassTag](v: T => V): MapPartitionsLDD[V, T]

  def flatMap[V:ClassTag](v:(T => GenTraversableOnce[V])):
      MapPartitionsLDD[V, T]

  def sample(n:Int):MapPartitionsLDD[T, T]

  def samplePartitions(n:Int):LDD[T]

  def groupByPartitions[V:ClassTag](v:(T => V)):
      GroupByPartitionsLDD[V, T]

  def sortWithPartitions(v:((T, T) => Boolean)):
      MapPartitionsLDD[T, T]

  def filter(v:(T => Boolean)):MapPartitionsLDD[T, T]

  def toMapLDD[U:ClassTag, V:ClassTag](f: T => (U, V)):
      MapLDD[U, V]
}

abstract class MapLDD[T: ClassTag, U: ClassTag] extends LDD
    [(T, U)] {
  def reduceByKeyPartitions(f:(U, U) => U):MapLDD[T, U]

  def reduceByKey(f: (U, U) => U):ReduceByKeyLDD[T, U]
}
```

In this paper, we directly use external SSDs as an array and bypass multiple complex storage layers (e.g., filesystem and RAID controller). As shown in Fig. 3, we developed a standalone storage manager for the array. Each disk is managed by a log-structured storage engine. The storage engine has one log head to serve the write request and can support multiple read threads to load data partitions concurrently. The interface of dataset is defined as LDD in Listing 1, which is constructed by a number of data partitions. Each partition contains it's disk drive ID, the offset of the data on disk and the size of the partition. The content of a partition can either stored on the disk, cached in memory or derived from it's parent partitions.

B. Parallel data storage and processing engine

The existing parallel data processing frameworks does not consider the data layout placed on the local storage, thus they are unable to perform optimal tasks scheduling. On the contrary, the ApproxSSD has its built-in storage engine. In the storage engine, a data partition is stored in a continuous address space on the disk. The partition to physical storage mapping information is always stored in the memory, so that there is no metadata related issue which many filesystem faces. As shown in Fig. 4, the LDD to be stored on the disk submits all its partitions to the persist router. The persist router then deliver the persist task to individual disk driver in a balanced manner. For simplicity, the disk driver has only one data store thread. All the incoming partition persist task will be served in a log-structured manner, so that all the write to the SSD are sequential. To boost the performance, the number of threads that can read partitions from one disk concurrently is not limited.

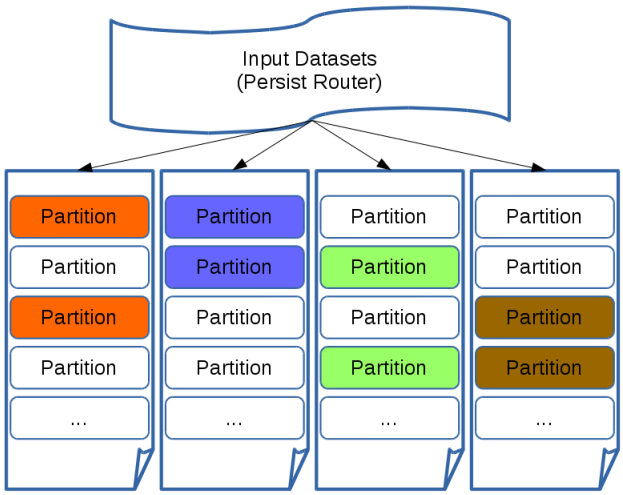

Fig. 4: Parallel Data Storage Engine.

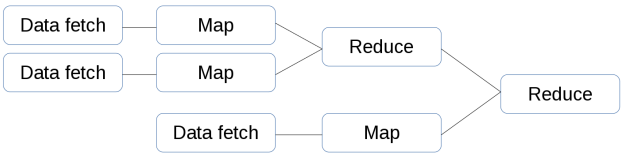

Fig. 5: Approximate Pipeline.

1) Data Layout Aware Sampling on SSD Array: When the user perform approximate sampling, the input datasets are usually randomly selected without any knowledge of its physical placement on the disk. This can result in 1) unbalanced amount of data to be selected on disks and 2) I/O contention in the data reads process. In ApproxSSD, we can get the corresponding drive of a given partition in the task scheduler and planner. The sampling algorithm first groups the partitions based on their drive ID, then it initiates multiple sampling threads in each group. As shown in Fig. 4, multiple sampling threads, distinguished by the color, almost select the same amount of data from each disk.

2) Approximate Pipeline: As discussed in the previous section, SSD can deliver better performance when the number

of concurrent read threads increases and saturate its internal hardware capability. However, this is not true for the computing engine as increasing the number of computing threads will introduce significant OS scheduling and memory usage overhead. To better exploit the SSD performance, we pipeline the I/O and the computation workloads. Upon submission, the LDD job is compiled into a partition dependency tree. There are three distinct partitions: the one having the content stored on the disk; the intermediate transactional partition that need to be computed; and the partition with critical data need to be persist to the disk. There are two types of dependency defined in ApproxSSD. The map dependency has one input partition and one output partition. The reduce dependency defines the multi in multi out (MIMO) data transforming. The logical job is expressed by a partition dependency tree as a DAG of tasks. These tasks execute in parallel, and operate on a set of data samples that are distributed across multiple disks. The storage engine is responsible for the partitions with it's data stored on disk e.g. input partitions. It selectively sample these partitions and fetch the content from the disk. Once this is done, the partition then be submitted to Map and Reduce executor. As shown in Fig. 5, the data sampling and task computing is overlapped so that we can both increase the disk and CPU usage.

3) Straggler Dropping: Because of the unpredictable performance of SSDs, balance the amount of data to be sampled from each disk can not avoid the runtime straggler. In order to reduce the probability of a straggling SSD slowing down the entire application, we always launch 10% more tasks on random samples of underlying data on multiple disks and, as a result, do not wait for the last 10% tasks to finish. With a fine grained control over the sampling ratio of multiple disks, avoiding straggling SSDs during data loading phase results in further improvements in approximation runtime. Straggler dropping does increase the chance of dropping some critical data partitions in cases when data is sparse [16], [29]. However, in our experiments, there is no significant error-bar changes while approximation runtime has been noticeably reduced.

IV. Preliminary Experiments

We conducted a preliminary performance evaluation of the ApproxSSD system by comparing against to one of the most popular parallel data analysis platform Spark [30] version 1.6.1. We evaluated Spark and ApproxSSD by timing the execution of the Wordcount algorithm on the 49GB Wikipedia database dump [1]. All experiments were conducted on a single node machine which has 8 virtual cores, 16GB of memory, 8 SSDs, and runs 64-bit Linux 3.16.0. The Spark runs under standalone mode with software RAID 0 and ext4 file system.

As illustrated in Fig. 6, our results show that ApproxSSD is 2.7 times faster than the in memory data parallel platform Spark while using 0.01% sampling ratio. It is worth noticing that ApproxSSD can even outperform Spark by 1.3 times faster while doing whole data processing. This is because of the reduced I/O stack and parallel data prefetch engine can reduce the data load time. The results show an increase of the overall performance speedup as we reduce the sampling ratio. The extra performance gain while doing approximative query is because of the balanced data choices among the SSDs and the runtime straggler dropping. The error-bar of ApproxSSD, as shown in Fig. 7, is very comparable to that of Spark. ApproxSSD is designed to be a general data parallel engine that supports a wide range of operations and applications including graph processing and machine learning, while leveraging the performance advances developed in data layout aware task scheduler, by integrate both the storage and computing engine. We have already identified a number of candidates for performance improvement in our prototype. We belief that we can further improve the performance of ApproxSSD in the near future, while providing a highly usable and low cost system for complex data analysis.

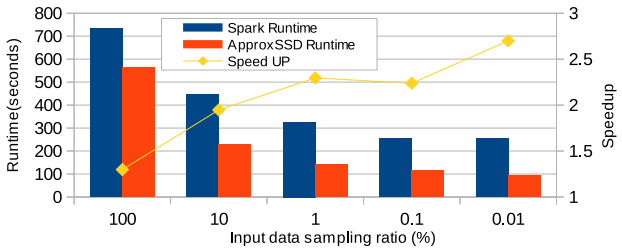

Fig. 6: **Wordcount runtime comparison** between ApproxSSD and Spark. The reported runtime includes the time to load the data and then run the target analysis algorithms.

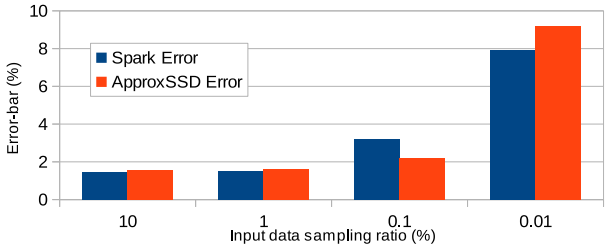

Fig. 7: **Error-bar comparison** between ApproxSSD and Spark. There is no noticeable error-bar changes in the ApproxSSD.

V. Conclusions and Future Work

We have presented ApproxSSD, an approximate data processing engine that combines the parallel storage systems and parallel data processing systems. It leverages the data layout aware sampling in approximate query to reduce the possible workload imbalance and I/O contentions among SSDs thus speed up the overall performance. Rather than do the data choice at the application level, ApproxSSD provides a sampling ratio to the parallel storage engine. The parallel storage engine then launch multiple data sampling threads for each disks to maximize the disk throughput. Besides, we also developed straggler drop which capture the runtime disk performance and balance the amount of data been served by each disk. We have implement the prototype system in

Scala. Primarily results shows that out prototype system can outperform Spark by upto 2.7.

We have identified a number of possible future optimization. First, we will continue improving the performance of ApproxSSD. Second, we would like to develop complex iterative approximative applications based on ApproxSSD. and implement a semi-external memory MapReduce system abstraction. Last but not least, we plan to investigate how data parallel system can leverage emerging non volatile memory.

ACKNOWLEDGMENT

This work is supported in part by the US NSF Grant CCF-1527249, CCF-1337244 and NSF CAREER Award 0953946.

REFERENCES

[1] Wikimedia database dumps. https://dumps.wikimedia.org/.

[2] AGARWAL, S., MILNER, H., KLEINER, A., TALWALKAR, A., JORDAN, M., MADDEN, S., MOZAFARI, B., AND STOICA, I. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 481–492.

[3] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference* (2008), pp. 57–70.

[4] COHEN, M. B., LEE, Y. T., MUSCO, C., MUSCO, C., PENG, R., AND SIDFORD, A. Uniform sampling for matrix approximation. In *Proceedings of the 2015 Conference on Innovations in Theoretical Computer Science* (2015), ACM, pp. 181–190.

[5] CZUMAJ, A., AND STEMANN, V. Randomized allocation processes. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on* (1997), IEEE, pp. 194–203.

[6] DA ZHENG, D. M., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. Flashgraph: processing billion-node graphs on an array of commodity ssds. In *FAST* (2012).

[7] EFRON, B. *Bootstrap methods: another look at the jackknife*. Springer, 1992.

[8] GOIRI, Í., BIANCHINI, R., NAGARAKATTE, S., AND NGUYEN, T. D. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), ACM, pp. 383–397.

[9] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 17–30.

[10] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 599–613.

[11] GU, L., AND LI, H. Memory or time: Performance evaluation for iterative operation on hadoop and spark. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on* (2013), IEEE, pp. 721–727.

[12] GUPTA, A., KIM, Y., AND URGAONKAR, B. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, vol. 44. ACM, 2009.

[13] HUA, Y., JIANG, H., AND FENG, D. Fast: near real-time searchable data analytics for the cloud. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), IEEE Press, pp. 754–765.

[14] IOMETER, T. Iometer: I/o subsystem measurement and characterization tool. *Open source code distribution: http://www. iometer. org* (1997).

[15] JUNG, M., AND KANDEMIR, M. Revisiting widely held ssd expectations and rethinking system-level implications. In *ACM SIGMETRICS Performance Evaluation Review* (2013), vol. 41, ACM, pp. 203–216.

[16] LI, P., CHURCH, K. W., AND HASTIE, T. J. Conditional random sampling: A sketch-based sampling technique for sparse data. In *Advances in neural information processing systems* (2006), pp. 873–880.

[17] LOHR, S. *Sampling: Design and Analysis*. Cengage Learning, 2009.

[18] LOW, Y., GONZALEZ, J. E., KYROLA, A., BICKSON, D., GUESTRIN, C. E., AND HELLERSTEIN, J. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1408.2041* (2014).

[19] LV, Q., JOSEPHSON, W., WANG, Z., CHARIKAR, M., AND LI, K. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases* (2007), VLDB Endowment, pp. 950–961.

[20] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[21] MENG, X., BRADLEY, J., YAVUZ, B., SPARKS, E., VENKATARAMAN, S., LIU, D., FREEMAN, J., TSAI, D., AMDE, M., OWEN, S., ET AL. Mllib: Machine learning in apache spark. *arXiv preprint arXiv:1505.06807* (2015).

[22] MENG, X., AND MAHONEY, M. W. Low-distortion subspace embeddings in input-sparsity time and applications to robust linear regression. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing* (2013), ACM, pp. 91–100.

[23] NATH, S., AND GIBBONS, P. B. Online maintenance of very large random samples on flash storage. *Proceedings of the VLDB Endowment 1*, 1 (2008), 970–983.

[24] PARK, S., AND SHEN, K. Fios: a fair, efficient flash i/o scheduler. In *FAST* (2012), p. 13.

[25] PARK, S.-H., HA, S.-H., BANG, K., AND CHUNG, E.-Y. Design and analysis of flash translation layers for multi-channel nand flash-based storage devices. *Consumer Electronics, IEEE Transactions on 55*, 3 (2009), 1392–1400.

[26] RAAB, M., AND STEGER, A. balls into binsa simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*. Springer, 1998, pp. 159–170.

[27] VENKATARAMAN, S., PANDA, A., ANANTHANARAYANAN, G., FRANKLIN, M. J., AND STOICA, I. The power of choice in data-aware cluster scheduling. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 301–316.

[28] WHITE, T. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[29] YAROSLAVSKY, L. P., SHABAT, G., SALOMON, B. G., IDESES, I. A., AND FISHBAIN, B. Nonuniform sampling, image recovery from sparse data and the discrete sampling theorem. *JOSA A 26*, 3 (2009), 566–575.

[30] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.

[31] ZHENG, D., MHEMBERE, D., BURNS, R., AND SZALAY, A. S. Flashgraph: Processing billion-node graphs on an array of commodity ssds. *arXiv preprint arXiv:1408.0500* (2014).