Towards Batch-Processing on Cold Storage Devices

Ali Hadian, Thomas Heinis

Imperial College London
{hadian,t.heinis}@imperial.ac.uk

***Abstract*—Large amounts of data in storage systems is cold, i.e., Written Once and Read Occasionally (WORO). The rapid growth of massive-scale archival and historical data increases the demand for petabyte-scale cheap storage for such cold data. A Cold Storage Device (CSD) is a disk-based storage system which is designed to trade off performance for cost and power efficiency. Inevitably, the design restrictions used in CSD's results in performance limitations. These limitations are not a concern for WORO workloads, however, the very low price/performance characteristics of CSDs makes them interesting for other applications, e.g., batch processes, too. Applications, however, can be very slow on CSD's if they do not take their characteristics into account.**

In this paper we design two strategies for data partitioning in CSDs — a crucial operation in many batch analytics tasks like hash-join, near-duplicate detection, and data localization. We show that our strategies can efficiently use CSDs for batch processing of terabyte-scale data by accelerating data partitioning by 3.5x in our experiments.

I. Introduction

Data is growing at an unprecedented rate but at the same time, not all data is accessed at the same rate. A substantial amount of data is so cold or infrequently accessed that the cost of keeping it on an always-on HDD is unaffordable, making new storage solutions imperative. Cold Storage Devices, for example, are rack-scale hard disk storage with petabytes of capacity specifically designed for cold workloads. CSD services include Microsoft Pelican [1], Amazon Glacier[1], OpenVault's Knox storage [2], and Facebook's Blu-ray storage [3]. The key restriction that distinguishes CSDs from other storage systems is that only a few disks of a CSD cluster are spun up at the same time, due to power and cooling limitations. For example, only 8% of disks can be simultaneously spun up in Pelican, and switching between disks takes 10s [1].

While the design-level restrictions of CSD, such as limiting the number of accessible disks, greatly reduces cost of archival storage, such restrictions make CSD an unsuitable infrastructure for *processing* data per se. The straightforward approach for analyzing an archival dataset on a CSD is to copy and run the analytic tasks outside on a cluster. Archival datasets, however, are so big that data transfer is expensive and the compute clusters have to be of unreasonable size to make the data fit.

Data partitioning is a common and crucial building block for many data system operations such as on-disk hash joins, creating inverted index/graphs, and near duplicate removal in data systems (e.g., MapReduce [4]) and database engines. In data partitioning methods, the input data is read to a buffer (e.g. main memory or local disk), mapped to partitions and then flushed to the target partition. When the data is located on disk, partitioning methods assume uniform access to the partitions, i.e., reading and writing to any partition takes the same time regardless of where it is located on storage. In a CSD, however, only a part of the storage (one disk group) and thus partitions can be accessed at any time, as switching between disk groups requires 1) spinning down the active disk group and 2) spinning up the target disk group which takes multiple seconds. If the partitioning method is unaware of the CSD characteristics, it will spend substantial time waiting for disk groups to spin up to flush or read data, rendering batch processing on a CSD impractical.

In this paper, we study the problem of performing partitioning on a CSD. Our contribution is to develop two partitioning strategies — *BuffPack* and *OffPack* — for efficient partitioning of CSD-resident datasets. BuffPack targets at minimizing the number of disk group switches, while OffPack leverages *write offloading*, a data consolidation technique designed for power management in data centers. The latter allows flushing data to the 'wrong' disk group (i.e., the active disk group) to avoid a switch but eventually moves all data to its correct destination/partition.

We additionally define analytical models to estimate the execution time for both strategies so that the best suited strategy can be chosen for a given CSD configuration. Results show that both strategies are orders of magnitude faster than a CSD-oblivious baseline. On a CSD with similar characteristics as Pelican (1GB/s bandwidth and 12 disk groups) our strategies partition a 100TB dataset in two days while it takes two weeks using the baseline.

II. Background and Related Work

A. Cold Storage Devices

Pelican. Pelican is a cold storage developed at Microsoft [1]. Multiple hardware capabilities are limited to reduce the cost of CSD deployment, cooling, and power usage. For example, while Pelican contains 1152 disks, only 96 can be cooled and only 144 can be powered simultaneously. Disk groups thus need to be spun up and down, introducing a group switching latency of 10s.

OpenVault cold storage: OpenVault's Knox storage is a cold storage cluster by the Open Compute Project [2]. It consists of multiple 2U chassis where only one disk can be accessed

[1]https://aws.amazon.com/glacier/

at a time. A similar device is Facebook's BluRay cold storage where loading each BluRay disk using a mechanical arm takes 30 seconds.

B. Data Processing on Cold Storage

Research on data processing and management on cold storage hardware has only just started. Skipper [5] is a framework for query processing on CSDs with disk grouping. It combines adaptive query processing techniques with customized I/O scheduling and page caching to execute queries on PostgreSQL database on CSD [5]. Further, a data layout [6] for cold storage hardware (similar to Pelican but with a switching latency of 30s) has been proposed. Based on the access patterns, groups of objects that are accessed together are stored on same disk group. Nakshatra [7] is a data processing framework for batch data analytics on tapes using pre-fetching.

C. Data Processing on Tertiary Storage

Work has also been done on tape-based tertiary storage, i.e., tape libraries where tapes are mounted and dismounted by a mechanical arm. Holtman et al. suggest a caching algorithm for query processing over massive scientific data residing in tertiary storage, where frequently accessed data files are cached in the hard disk drive. [8]. An analytial performance model for tertiary storage is provided in [9]. Li et al. suggest various methods for aligning objects in tertiary storage based on features like timestamp and object relations, in order to minimize data access latencies [10]. Myllymaki et al. suggest that a parallel adaptive Nested Block Join is efficient for relational databases in tertiary environment [11]. Other work on efficient management and access ordering of multi-dimentional data, such as large matrix data, on tertiary storage [12], [13], [14], [15] has also been carried out.

Previous work shows that using in-memory buffering and the data placement strategies play a crucial role in efficient data processing on cold storage devices such as tertiary storage. However, CSDs have different performance characteristics compared to earlier generations of tertiary storage. For example, while inter-drive and intra-drive seek time in tape libraries are in order of several minutes, seek time on the active disk group in a CSD takes only a few milliseconds.

III. Data Partitioning on a CSD

Given an input data set D containing data records, the aim of general partitioning is to assign the records in D to N partitions $P_1, P_2, \cdots, P_N$ using a partitioner function. The data is initially stored on the CSD as will the final partitions. On a CSD, the aim further is to store each partition entirely in one disk group. Without loss of generality we assume the number of disk groups K to be the number of partitions N, i.e., $K = N$.

Figure 1 illustrates data partitioning on a CSD. Data records are read from each disk group and the target partition for each record is computed using a partitioner function running on the computing node. The buffer (RAM) accumulates records, grouping records belonging to the same partition P_j in the same group B_j. Once the buffer is full, at least data belonging

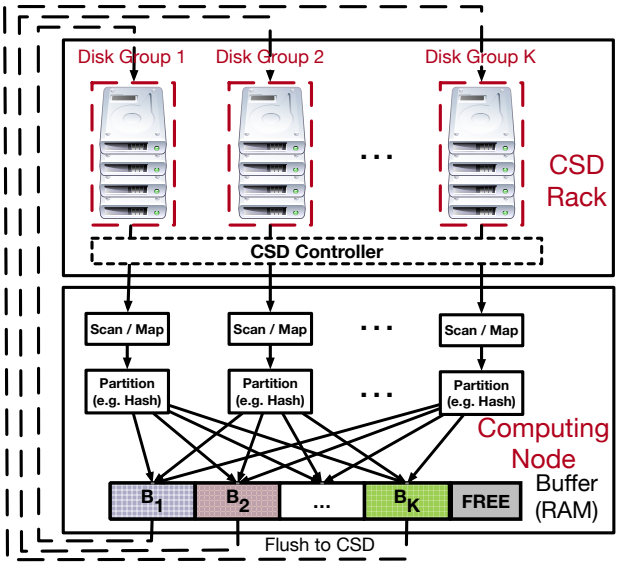


Fig. 1: Data partitioning on CSD

to one partition is flushed to make room for reading more input data.

A. Applications

Data partitioning is a ubiquitous operation in large-scale data processing. Table I gives examples of tasks that involve a partitioning/shuffling phase where data points are relocated between machines or disks.

The methods we propose can be used in different CSD-resident batch processing systems that rely on data decomposition using partitioning, including massive-scale duplicate detection, database sharding, and inverted index creation. They can also be used for changing the layout of a database, e.g., re-sorting the data records based on a specific column. This is a common scenario in archival systems where, for example, a massive set of satellite images that are originally ordered by data is to be re-sorted based on location, so that images from the same location are co-located.

The methods we propose can also be used to adapt the MapReduce engine for efficient execution on CSD devices. In this case, our approach handles the 'shuffling' mechanism of MapReduce, so that the `(reducerID = partitioner(map(k,v).key(), K)` is treated as the target disk group ID.

B. Assumptions & Parameters

In our work we assume that the data size is larger than the capacity of a single disk group(Figure 2). Also, the partitioning function is lightweight, e.g., a simple hash function with little overhead is used. Further, we assume that the internal HDD of the computing node is not used as a buffer since it is much slower than both memory and the CSD disk group (unless high-bandwidth storage such as NVMe or SSD-RAID is used).

The parameters for our partitioning methods and models are presented in Table II. S_M and S_D are the size of main memory and the dataset. α is the ratio of output data to raw input data,

TABLE I: Data partitioning tasks in different batch tasks

Task	Operations on Input	Partitioner	Operations on Output
MapReduce [4]	Load, `map()`, Combine	Hash, User-defined	Sort, Merge, `reduce()`
N-gram Extraction / Text Indexing [16], [17]	Tokenize	Hash	Count, Create Index
DB Sharding, Data Localization [18], [19]	Scan, Project	Hash, Range	Create local indexes / files
Near-duplicate Detection [20], [21]	Tokenize / Split	Locality-sensitive Hash	Compute similarities, Filter

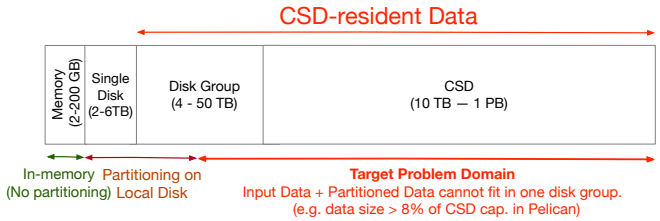


Fig. 2: Application domain of this paper

i.e., for X bytes of input data, $\alpha \cdot X$ is written as output. In many applications $\alpha = 1$, but α can be smaller than 1 if some input data is filtered or bigger than 1 if, for example, data is decompressed. T_{sw} is the group switch time (˜10-20 sec), T_{sk} is the disk seek time (˜5-10 ms), and $T_r(S), T_w(S)$ are the times to read/write S bytes sequentially from/to a disk group. α is typically known a priori but can also be computed when partitioning (comparing the bytes read and written from/to CSD). When flushing data to the j'th disk group, $|B_j|$ bytes of space become available in the buffer. The system can thus now read $|B_j|/\alpha$ bytes from the CSD.

C. Baseline Method

Data partitioning starts with reading the data into the buffer and assigning records to partitions with the partitioner. Once the buffer is full, we switch to a disk group and flush to it to free up memory, e.g., we switch to disk group g_j and flush all data in B_j to disk. The switch takes T_{sw}s (or 0s if the previous and next disk groups are the same). Reading and partitioning then resumes for the remaining input data on g_j until the buffer is again full or the input data on g_j is entirely read. Finally, the remaining data in the buffer is flushed to each disk group.

TABLE II: Parameters of the model

Parameter	Description
S_D	Input data size
S_M	Memory (bytes)
S_M^{free}	Free memory (bytes)
K	Number of disk groups
T_{sk}	Time of random disk seek
$T_{r/w}(S)$	Time to Read/Write S bytes from/to CSD
T_{sw}	Groups Switch time
α	Output/input size ratio
B_i	Buffered data for the i'th partition

The choice of the next disk group for flushing data affects performance considerably. The baseline method is oblivious to the underlying storage, i.e. the high latency of disk group switching, and may use an excessive number of switches. The baseline flushes the entire buffer when full, thus requiring a switch to each disk group. As a result, the CSD will spend excessive time in switching between disk groups.

Each of the K switches and subsequent flush frees the entire memory (S_M bytes). The total number of flushes is $K\lceil \alpha S_D/S_M \rceil$ and the total time for flushing the buffer is $T_w(S_M) + K \cdot (T_{sw} + T_{sk})$, making the execution time for the baseline method:

$$T_{\text{Baseline}} = T_r(S_D) + T_w(\alpha S_D) + T_{sw} \cdot K \cdot \lceil \alpha S_D/S_M \rceil \quad (1)$$

IV. Efficient Data Partitioning

A. BuffPack: The Greedy Approach

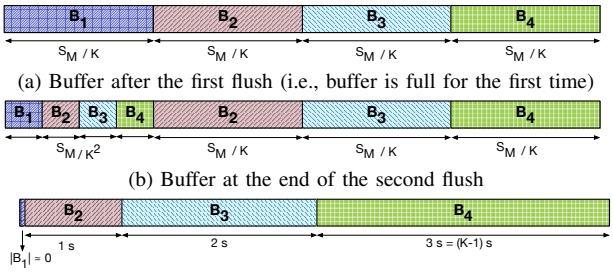
Unlike the baseline, the greedy approach does not flush the entire buffer but instead picks the $|B_j|$ that leads to the best performance. The objective of BuffPack is to greedily minimize 'overhead per GB of data' between any two flushes. If we switch from g_i to g_j, then we can (1) flush $|B_j|$ bytes to g_j, taking $T_w(|B_j|)$ and (2) read $|B_j|/\alpha$ bytes from g_j so that the buffer is again full ($T_r(|B_j|)$. This process has two overheads: two seek overheads for writing and reading ($2{*}T_{sk}$) as well as group switch time from g_i to g_j (T_{sw}). We should thus choose the disk group with the least 'overhead per GB' overhead, i.e., B_j with the best *efficiency*, η_j:

$$\eta_j = \frac{T_w(|B_j|) + T_r(|B_j|/\alpha)}{T_w(|B_j|) + T_r(|B_j|/\alpha) + \underbrace{2 \cdot T_{sk} + Tsw^{i \to j}}_{\text{overhead}}} \quad (2)$$

Since flushing to the active disk group requires no disk group switching ($Tsw^{active} = 0$), BuffPack will flush to the active disk group, unless if $|B_{active}|$ is so small that $\exists j : \eta_j > \eta_{active}$. For example, for the Pelican configuration ($T_{r/w}(1GB) = 1sec, T_{sw} = 10s, T_{sk} \approx 0.01s$) and $\alpha = 1$, the efficiency of flushing 100MB of data into the active disk group is $\eta_{active} = (2 \cdot .1)/(2 \cdot .1 + 2 \cdot 0.01) = 91\%$. In this case, BuffPack will only flush to a non-active disk group if $\exists j : \eta_j > 91\% \rightarrow |B_j| > 51GB$. It is trivial to show that BuffPack always chooses the disk group with largest buffer (which has the highest efficiency).

Assuming a uniform partitioner, records will be evenly distributed to all partitions and before the first flush, as shown in Figure 3(a), the buffers nearly have the same size. BuffPack therefore flushes to the active disk group to avoid disk group switching overhead. The flush frees $1/K$ of the memory buffer for reading more data. Once the buffer is again full, data for g_1 only fills $1/K^2$ of the entire buffer, as shown in Figure 3(b). More flushes can follow until the size of the remaining flushable buffer is insignificant.

We refer to the flushes within each disk group as a *super-step*, i.e., the period between two disk group switches. With S_M^{free} bytes available at the start of each super-step, the total



(a) Buffer after the first flush (i.e., buffer is full for the first time)

(b) Buffer at the end of the second flush

(c) The steady buffer state in BuffBack at the end of each flush (last four switches: $4 \rightarrow 3 \rightarrow 2 \rightarrow 1$)

Fig. 3: Buffer status illustration

output data that can be produced without switching to another disk group is:

$$S^{\text{output}}_{\text{super-step}} = S_M^{free}(1 + 1/K + 1/K^2 + ...) \simeq \frac{K}{K-1} S_M^{free} \tag{3}$$

Let $\{|P|\} = \{|B_1|, \cdots, |B_K|\}$ be the size of each buffer. Without loss of generality, suppose that the partitions indices with higher indices are larger, i.e. $|B_1| \leq |B_2| \leq \cdots \leq |B_K|$. It is trivial to show that for any two partitions B_i and B_j, if the last super-step (i.e., last flush) in g_i has been earlier than the last flush in g_j, then $|B_i| > |B_j|$. Considering a uniform partitioner, after K super-steps the system reaches a steady-state. During a super-step, each partition grows by s bytes. With 4 partitions, for example, the size of partitions will be $\{0, s, 2s, 3s\}$ (Figure 3(c)). The size of partitions is an arithmetic sequence with common difference of s, i.e., $\{|P|\} = \{0, s, 2s, \cdots, (K-1) \cdot s\}$. Normalizing the values with respect to the total buffer size ($\sum |B_i| = S_M$), yields:

$$s = \frac{2 \cdot S_M}{K(K-1)} \tag{4}$$

After the steady state is reached, the size of the biggest partition in the buffer at the end of a super-step is $S_M^{free} = (K-1) \cdot s = 2 \cdot S_M/K$. The data flushed in a super-step thus is $S^{\text{output}}_{\text{super-step}} = 2 \cdot S_M/(K-1)$. The estimated number of disk group switches becomes:

$$\text{Switches (BuffPack)} = \frac{\alpha S_D}{S^{\text{output}}_{\text{super-step}}} = \frac{\alpha S_D \cdot (K-1)}{2 \cdot S_M} \tag{5}$$

Hence, the total execution time is:

$$T_{\text{BuffPack}} = T_r(S_D) + T_w(\alpha S_D) + \frac{T_{sw} \cdot \alpha S_D \cdot (K-1)}{2 \cdot S_M} \tag{6}$$

B. OffPack: Write Offloading

Write offloading temporarily writes data to the 'wrong' place to consolidate data transfers [22] and save power in data centers. We base our strategy to reduce the number of disk group switches (and thus reduce execution time) using this approach. In our setting, when the buffer is full, we allow the entire buffer to be temporarily flushed (i.e., 'offloaded') into the active disk group instead of the target disk group. Once all input is processed, the partitions flushed to the 'wrong' disk group are transferred to the correct one. The output data records for each disk group are stored together, so that data of each partition can be read separately in the transfer phase, which results in fewer disks groups switches

In the transfer phase, OffPack chooses the target disk group with the most offloaded data, so that the offloaded data on different partitions finish up together. Otherwise, the data on some disk groups may finish far earlier than others, so that every time OffPack carries data to the partitions with no offloaded data, it will switch back to other disk group with an optimal buffer. The choice of the next partition is not crucial since there are only few disk groups in a CSD and for most transfers, the entire buffer is filled up, hence there is not much room for improvement.

In the partitioning phase of OffPack, assuming a uniform partitioner, $1/K$ of the output is written to the correct disk group, while $(K-1)/K$ is stored in the 'wrong' disk group. This requires reading the input data (S_D), and writing αS_D bytes.

The second phase transfers output data not stored in the correct disk group. The size of data to be transferred is $S_{\text{offload}} = \frac{K-1}{K} \cdot \alpha S_D$. Assuming $S_D >> S_M$ the number of switches required for moving the data is $\alpha S_D/S_M$. As a result, the total time for data partitioning using write-offloading is:

$$T_{\text{offload}} = T_r\left((\frac{K-1}{K}\alpha + 1)S_D\right) + T_w\left(\frac{2K-1}{K}\alpha S_D\right) + T_{sw}\left(\frac{\alpha S_D}{S_M}\right) \tag{7}$$

V. Experimental Validation

In this section, we evaluate the efficiency of the methods, along with a HDD cluster (which has no disk switch overhead). We simulate a CSD based on Microsoft's Pelican [1], [23]. The main difference between the CSD and other disk-based rack-scale storage hardware is a ~10-second latency for switching between the disk groups. Whenever a switch between two different disk groups occurs, the switching delay is added to the execution time. Also, the time taken for read/write operations is simulated using the configuration and expected performance of the SMR disk on Pelican. The parameters for the experiments are listed in Table III. We assume that each disk group stores the same amount of input data (according to Pelican's middleware policies). Except for the skewness experiment (Figure 8), we also assume that input records are uniformly distributed between partitions.

TABLE III: Default parameters for simulation

Parameter	Value
Dataset size	100 TB
Number of disk groups	12
Buffer Size	8 GB
Read/Write rate	1 GB/sec
Disk spin-up latency	10 sec

A. Sensitivity to Data Size

Figure 4(left) compares execution times, which grow almost linear for all methods. As the experiment shows, the size of data does not determine which method is fastest.

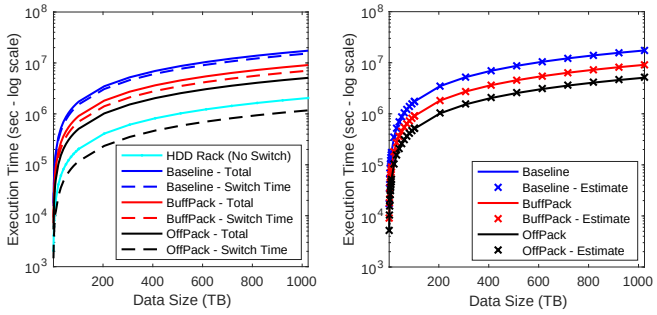


Fig. 4: Effect of data size

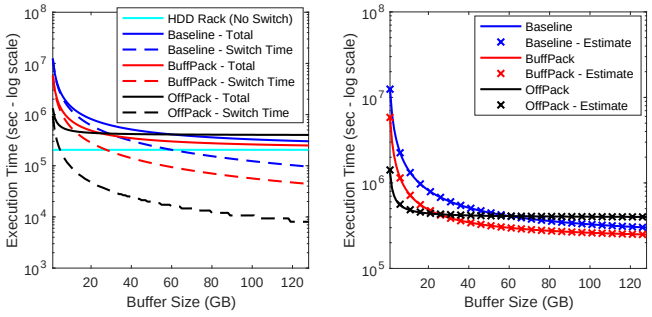


Fig. 5: Effect of buffer size

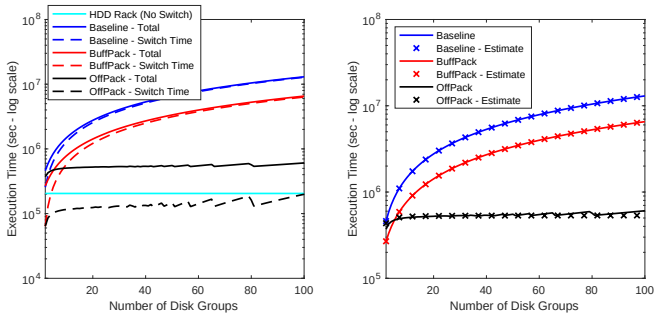


Fig. 6: Effect of the number of disk groups

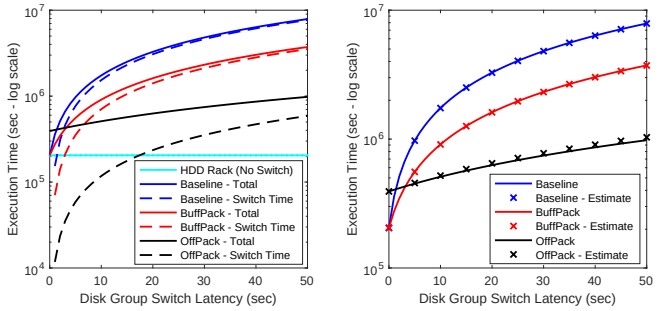


Fig. 7: Effect of disk group switch latency

B. Sensitivity to Buffer Size

According to the analytical models, we expect the number of disk group switches in all the three methods to be inversely proportional to buffer size. Figure 5(left) shows the switch time and the total execution time. For smaller buffers, e.g., $<$ 25GB, OffPack performs better as disk group switching is the main overhead. That is because OffPack requires fewer switches as it can use the entire buffer and can thus maximize the data moved to the correct partition for most transfers. For larger buffers, however, OffPack is slower than BuffPack, due to the extra cost of transferring (re-reading and re-writing) the offloaded data. The baseline method is inferior to BuffPack, and is also slower that OffPack for small buffers.

C. Sensitivity to the Number of Disk Groups

The number of disk groups has a key effect, as more disk groups lead to more switching overhead. However, as shown in Figure 6, the performance of OffPack is not affected as much as the baseline and BuffPack. This shows that there is a great opportunity for many applications that rely on data partitioning to be run on CSD in which 1% (or fewer) of disks are active at any time.

D. Sensitivity to the Group Switch Latency

The latency of switching between disk groups varies between 10 to 30 second in current CSDs. Figure 7 shows the execution time for disk group switch latencies. The results from disk switch latency are similar to our analysis of the number of disk groups, as both the number of switches and latency of switches add up to the total disk switch latency, i.e., total switching time = number of switches $\times$ switch latency. OffPack performs better when the disk group switch latency is high compared to other approaches.

E. Model Accuracy

As the analysis of our analytical model in Figures 4, 5, 6 and 7 (right) shows, the model accurately identifies the execution time of all approaches. The fastest method can be chosen using the hardware specification and data size, before execution.

More interestingly, both the experimental results and the model show that the choice of the fastest method does not depend on the data size. The fastest approach can be identified using the CSD configuration and the capabilities of the computing node. This facilitates designing data analytics applications on CSDs.

F. Uniform versus Skewed Workloads

The analytical model assumes that the partitioner is uniform. This is typically the case but it is important to understand the performance of a non-uniform partitioner. A non-uniform workload can be the result of different effects:

1) **Skewness of input data**: This occurs when the input data is not equally distributed between the disk groups. While this may be common for small datasets, e.g., size$<$1TB, the focus of our approach is on large datasets that need to be stored on multiple disk groups (as shown in Figure 2).
2) **Skewness of output data**: This type of skewness is caused by the partitioner function, i.e., more data is assigned to some partitions while less is assigned to others. For example, hashing data records based on 'year' when most data belongs to the last few years will assign most data records to a few partitions.

To model a non-uniform distribution in both input and output data, we use a Zipfian distribution where the odds for assigning one of N to group i of K groups is:

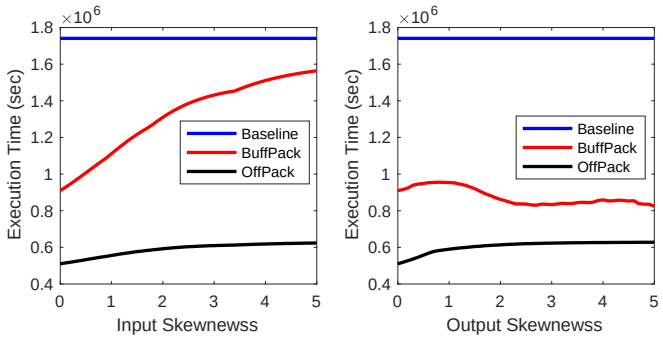


Fig. 8: Effect of skewness on performance

$$f(i; s, N) = (1/i^s)/\sum_{n=1}^{N}(1/n^s) \tag{8}$$

s is the skewness of the distribution. When $s = 0$, the Zipfian distribution is uniform and with increasing s, each item is more likely assigned to a disk group with smaller indices.

Figure 8 shows the effect of skewness in both input (left) and output (right) data.

When input data is skewed and concentrated on a few disk groups, the gain of applying BuffPack and OffPack reduces. This is inevitable, because the system has to distribute the data from few partitions to all disk groups, so when the system switches from a disk group with lots of records to another one with few input data for flushing the buffer, it cannot find enough input data on the disk groups with few data. This imbalance in transfer demands results in more disk group switches.

When the skewness occurs in output data, BuffPack performs slightly better compared to uniform partitioning, because the share of the biggest partition in the buffer is more than the expected share in the uniform case ($2 \cdot S_M/K$). However, higher skewness degrades OffPack's performance, as the buffer is more likely empty when it switches from g_i to g_j for $i < j$.

VI. Conclusions

Our work demonstrates how CSDs can be exploited for batch data analytics on archival data. We focus on data partitioning — a crucial part of many batch applications such as MapReduce, aggregation, and indexing — which is indeed the main bottleneck when running such batch-based applications on CSD-resident data. The suggested methods for data partitioning on CSDs, namely BuffPack and OffPack, reduce the disk switch overhead and in our experiments lead to a comparable performance to data partitioning on HDD clusters (which has no disk switch overhead). BuffPack is based on scheduling the order of flush operations to reduce the disk switch time, and is very effective when the computing node has a relatively large buffer. OffPack, however, leverages write offloading to aggregate data transfers, and is efficient when the buffer is small. Also, OffPack can be used for CSDs with extreme power-efficiency constraints, such that it can keep its performance even if only 1% (or less) of the disk groups are active at any time.

Acknowledgement

The support of the EPSRC Centre for Doctoral Training in High Performance Embedded and Distributed Systems (HiPEDS, Grant Reference EP/L016796/1) is gratefully acknowledged.

References

[1] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron, "Pelican: A Building Block for Exascale Cold Data Storage," in *OSDI*, 2014.

[2] M. Yan and L. Song, "Cold Storage Hardware v0.7," Open Compute Project, Tech. Rep., Apr. 2014. [Online]. Available: http://opencompute.org/projects/storage/

[3] K. Bandaru and K. Patiejunas, "Under the Hood: Facebook's Cold Storage System," Facebook, Tech. Rep., May 2015. [Online]. Available: https://code.facebook.com/posts/1433093613662262/-under-the-hood-facebook-s-cold-storage-system-/

[4] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.

[5] R. Borovica-Gajic, R. Appuswamy, and A. Ailamaki, "Cheap Data Analytics Using Cold Storage Devices," *VLDB Endow.*, vol. 9, no. 12, 2016.

[6] R. Reddy, A. Kathpal, J. Basak, and R. Katz, "Data Layout for Power Efficient Archival Storage Systems," in *HotPower*, 2015.

[7] A. Kathpal and G. A. N. Yasa, "Nakshatra: Towards Running Batch Analytics on an Archive," in *MASCOTS*. IEEE, 2014.

[8] K. Holtman, P. Van Der Stok, and I. Willers, "A cache filtering optimisation for queries to massive datasets on tertiary storage," in *DOLAP*. ACM, 1999.

[9] "an analytical performance model of robotic storage libraries."

[10] J. Li and S. Prabhakar, "Data Placement for Tertiary Storage," in *Goddard Conference on Mass Storage Systems and Technologies*. NASA, 2002.

[11] J. Myllymaki and M. Livny, "Relational Joins for Data on Tertiary Storage," in *ICDE*. IEEE, 1997.

[12] S. Sarawagi and M. Stonebraker, "Efficient Organization of Large Multidimensional Arrays," in *Proceedings of the 10th International Conference on Data Engineering*. IEEE, 1994.

[13] L. T. Chen, R. Drach, M. Keating, S. Louis, D. Rotem, and A. Shoshani, "Efficient Organization and Access of Multi-Dimensional Datasets on Tertiary Storage Systems," *Information Systems*, vol. 20, no. 2, 1995.

[14] A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim, "Multidimensional Indexing and Query Coordination for Tertiary Storage Management," in *SSDBM*. IEEE, 1999.

[15] B. Reiner and K. Hahn, "Optimized management of large-scale data sets stored on tertiary storage systems," *IEEE Distributed Systems Online*, vol. 5, no. 5, 2004.

[16] J. Uszkoreit, J. M. Ponte, A. C. Popat, and M. Dubiner, "Large Scale Parallel Document Mining for Machine Translation," in *COLING*. Stroudsburg, PA, USA: ACL, 2010.

[17] S. Huston, A. Moffat, and W. B. Croft, "Efficient Indexing of Repeated N-grams," in *WSDM*. New York, USA: ACM, 2011.

[18] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli, "Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning," *The VLDB Journal*, vol. 25, no. 3, 2016.

[19] R. Cattell, "Scalable SQL and NoSQL Data Stores," *ACM SIGMOD Record*, vol. 39, no. 4, 2011.

[20] Q. Zhang, Y. Zhang, H. Yu, and X. Huang, "Efficient Partial-duplicate Detection based on Sequence Matching," *SIGIR*, 2010.

[21] M. Najork, "Detecting Quilted Web Pages at Scale," in *SIGIR*. ACM, 2012.

[22] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-loading: Practical Power Management for Enterprise Storage," *Transactions On Storage*, vol. 4, no. 3, 2008.

[23] R. Black, A. Donnelly, D. Harper, A. Ogus, and A. Rowstron, "Feeding the Pelican: Using Archival Hard Drives for Cold Storage Racks," in *HotStorage*, 2016.