Efficient Big-Data Access: Taxonomy and a Comprehensive Survey

Anis Alazzawe, Amitangshu Pal, Krishna Kant
Computer and Information Sciences, Temple University, Philadelphia, PA 19122
[aalazzawe, amitangshu.pal, kkant]@temple.edu

***Abstract*—The emerging systems are not only generating huge amounts of data but also expect this data to be analyzed expeditiously to drive online decision-making and control. Thus, identifying the most relevant data and making it available close to the computation becomes a central challenge in driving the big data revolution. Storage systems play a crucial role in enabling efficient access to the stored data and intelligent storage management techniques are thus central to addressing the problem. Generally, as the data volume increases, the marginal utility of an "average" data item tends to decline, which requires greater effort in identifying the most valuable data items and making them available with minimal overhead and latency. Data driven mechanisms have a big role to play in solving this needle-in-the-haystack problem. In this paper we propose a taxonomy to provide a structure for understanding the common issues surrounding these techniques. We discuss these techniques and articulate many research challenges and opportunities.**

***Index Terms*—Locality Exploitation, Proximity Optimization, Data Reduction, Redundancy Removal, Data Filtering**

I. Introduction

The world is awash in data, the rate at which data is generated in the future will increase as the information technologies permeate deeper into every aspect of society including energy, transportation, logistics, medical care, etc. Furthermore, there is an increasing need to analyze the data in an online or real-time manner to drive intelligent decision making for society's interactions with the cyber and cyber-physical systems. This brings in the key challenge of proactively determining the data needs of an application and providing it most efficiently and in a form that is most useful to the application. With the overall data volume growing much faster than the useful or required data, the problem is becoming increasingly more urgent.

Big data analysis and mining thus have drawn a significant amount of attention recently, and a large volume of studies and surveys exist in this area. Tsai et al. [1] looked at scaling data analytics in terms of the platform and data analytics algorithms. Ghani et al. [2] surveyed big data issues in the context of social media. In [3], experimental study of big data analytics deployment practices is performed. In [4] highlights research and challenges in big data in the context of deep learning. Different big data methodologies and storage technologies have also been surveyed in [5]–[7]. Surveys on different application areas of big data analysis including Internet of things (IoT), mobile applications, traffic management, healthcare, big data in urban environments and smart cities, and application to utilities etc. are also studied in [8]–[17]. In contrast to these surveys on big data analysis and technologies, in this paper we provide a survey of the techniques used for learning about and predicting data needs and the corresponding methods to access the required data efficiently. We propose a taxonomy to provide a context for understanding the issues involved in achieving efficient data access and discuss the opportunities and open problems. The fundamental difficulty in doing this is that there are multiple views of data, and the connection between them can be rather weak. In particular, the traditional *block storage* system, which is quite prevalent, provides the following 3 views of data:

(a) Application's view that concerns data "items" that an application works with (e.g., files, database records, etc.). These items reside in a "storage volume" presented to the application by the host system, which knows how to access it but may not know about its structure or even the location. The host views the logical volume as simply a sequence of logical block addresses (LBAs), starting at 0.[1]

(b) Storage system's view, that concerns the mapping of the logical volume on to the storage system. The logical volume may consist of multiple segments, possibly mapped to different devices as determined by the *logical volume manager* (LVM) running on the storage server. The LVM or an associated module translates the LBA number into the device ID and offset. Since a device can also be seen as a sequence of (device-level) LBAs, the offset is in terms of LBA as well.

(c) Physical device's view, that concerns the precise mapping of device-level LBAs into device specific structure such as cylinder, platter and sector for hard disk drives (HDDs), or package, die, plane, block and page number in a solid-state drive (SSD).

The more powerful "Object Storage" model, discussed in section II-C, can also be thought in terms of the above 3 views: host's, storage system's and device's. Each of these views has important implications for efficient data access, and the corresponding mechanisms must be well coordinated to ensure the desired behavior. Unfortunately, there are no standard ways of conveying the application/host level information to the storage system thereby making the connection between (a) and (b) difficult. Furthermore, the storage system typically serves a large number of hosts and applications, and so the traffic seen by the storage system becomes difficult to identify with individual applications. Although object storage systems (see Section II-C) provide a better connection between host and storage systems, many storage system specific functions such as virtualization, deduplication, low level device management, etc. still make the connection quite difficult. These aspects

[1] Typical LBA size is 4KB but could be configured differently as well.

TABLE I: Common Abbreviations

SNIA	Storage Networking Industry Association
HDD/SSD	Hard Disk/Solid State Drive
FTL	Flash Translation Layer
SLC/MLC	Single/Multi (two) Level Cell Flash technology
TLC/QLC	Triple/Quad Level Cell Flash technology
CSG	Cloud Storage Gateway
RAID	Redundant Array of Inexpensive Disks
HDFS	Hadoop Distributed File System
LBA	Logical Block Address
LRU/LFU	Least Recently/Frequently Used
(S)ARC	(Sequential) Adaptive Replacement Cache
NVMe	Nonvolatile Memory Express Interface
NVMeoF	NVMe over Fabric Interface
NVRAM	Nonvolatile Random Access Memory
OSD	Object Storage Device (for object storage)
MDS	Metadata Server (for object storage)
QoS	Quality of Service
CAP	Consistency, Availability and Partition tolerance

further obscure the connection between (b) and (c).

In view of the above, a certain access pattern from application's perspective (e.g., access to a sequential range of database records) has nothing to do where the data is actually located or accessed from the device(s). Thus much of the challenge and complexity in dealing with access to big data lies in integrating the management across different layers based on specified or learned behaviors and connections. Data driven analytics techniques can be very useful in this learning as we shall point out later in the paper.

The outline of the paper is as follows: We start with a short review of some relevant concepts of modern storage systems in section II. Section III then discusses the fundamental drivers for efficient big-data access. Section IV introduces a taxonomy of data driven access methods and details behavior and semantics driven methods. Subsequent sections elaborate on the key taxonomic elements, namely, data access locality (Section V), data-computation proximity (Section VI), and data reduction (Section VII). Then in section VIII, we discuss the tradeoff between performance and other aspects. Finally, Section IX concludes the paper. Table I includes the key abbreviations used in this paper. We have also included supplementary material on the details of storage technologies and systems that the reader may wish to consult. However, we believe that the overview of basic concepts in section II should suffice to read this article.

II. Overview of Modern Storage Systems

A. Basic Storage System Concepts

Storage technologies continue to make rapid strides in many dimensions, and it is important to understand their key characteristics relative to data access performance. Solid State Drives (SSDs) are becoming ubiquitous because of their far lower access latency, higher IO bandwidth, smaller physical size and lower energy consumption as compared to the Hard Disk Drives (HDDs). SSDs are based on the underlying NAND "Flash" technology that represents different logic levels by the number of trapped electrons in each "cell" (or transistor) [18]. The technology started with only two levels (1 bit/cell), popularly known as SLC (single level cell), and has progressed to 2 bits/cell (mischaracterized as MLC or multi-level cell), 3 bits/cell (TLC), and 4 bits/cell (QLC). The flash technology allows the writing (or programming) of trapped electrons only once, after which the cell needs to be erased before rewriting. The technology also has strict limits on the number of program-erase (PE) cycles that it can take, known as *endurance*. It also has retention issues since the cells that are not read for a long time tend to leak out some trapped electrons. Both endurance and retention tend to deteriorate rapidly with more bits/cell. For example, a cheap QLC SSD may only allow a few hundred PE cycles before it wears out.

In addition to flash, there are many other upcoming technologies that are even faster than flash, allow overwrites, and have much better endurance/retention characteristics [19]. One specific technology that is already commercially available and that we will mention later is the Intel Optane technology [20]. It is available in both persistent memory and storage form, the key difference being that the CPU waits for an in-line response from a memory interface, but storage interfaces typically provide later asynchronous completions.

The lack of in-place writing and endurance issues in SSDs cause a lot of management complexity and require a sophisticated software layer called Flash Translation Layer (FTL) that hides all of the underlying operations such as keeping track of blocks/pages for out-of-place updates, allocating and consolidating blocks to control ”write amplification” (i.e., number of internal writes done for a single user issued write), garbage collection, wear leveling (ensuring that the writes are spread evenly throughout all the blocks), special handling of blocks with high read/write errors, etc. Further complexities arise because the SSD has a complicated internal structure consisting of multiple of packages, chips per package, dies per chip, planes per die, blocks per plane, and pages per block. The relevance of these aspects on data access are discussed later in the article.

Beyond the storage technologies, there are continuing improvements in storage interfaces; for example, the so called *NVMe interface* is becoming universal for high speed technologies and can provide protocol latencies as low as 10μs or less [21]. Also, much of the storage is accessed over the network regardless of whether it is concentrated in a few storage servers or hosted by the compute servers themselves. Thus, with rapidly decreasing latency of storage devices and interfaces, the latency of the network is becoming more important. Similarly, with rapidly increasing bandwidth of storage devices, the network bandwidth can already become a bottleneck. For example, a few inexpensive SSDs can easily saturate a 100 Gb/sec Ethernet link [22].

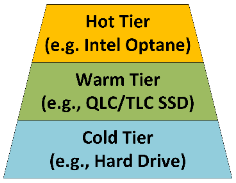

Fig. 1: Typical storage hierarchy

Modern storage systems provide a tremendous amount of flexibility in terms of how "storage volumes" are defined, allocated, and dynamically resized. All of these features are supported by "storage virtualization", which separates the logical address of the data (e.g., LBA number from a logical storage volume) from the physical device address. The require mapping can be quite complex and can add to the storage access latency in a large system.

Given storage technologies with a wide range of latency, bandwidth, and size characteristics, it is natural to organize them into a hierarchy as shown in Fig. 1. The "hottest" (or most active) data would go to the fastest devices which are limited in size due to high cost. The appropriate technologies here could include SLC/MLC SSDs or other higher speed technologies. The middle tier may consist of slower TLC/QLC SSDs, and the cold tier may consist of HDDs. A good management of storage hierarchy using intelligent "tiering", or dynamic movement of each data item to the most appropriate place in the hierarchy, is crucial to achieve high performance at a relatively moderate cost.

B. MetaData Issues in Storage Systems

The properties of the data such as location, size, time of last update, etc. are described by "metadata", often stored separately from the data. Metadata is crucial not only to access the relevant data items but also to determine which data items, if any, are relevant. The implication of the latter is that the metadata is accessed far more frequently than the data. Thus high performance access to metadata is even more crucial than the data itself. Since the size of the metadata is generally much smaller than data, it is easily cached in the highest storage tier, and may even largely reside in the memory. However, metadata may be updated even if the data is not (e.g., updating time of last read), and persisting the metadata may result in the problem of "small writes" to storage devices, which is both inefficient and may wear out the SSDs quickly. Yet, the metadata must be persisted (logged) properly to avoid data corruption in case of power failure.

Distributing the metadata can alleviate performance bottlenecks but at the cost of further complexity in grouping related metadata together, avoiding load-imbalance as the workload changes, and yet ensuring consistency. Singh and Bawa [23] provide a survey of such techniques. Xu et al. [24] present a dynamic ring-based metadata management mechanism that attempts to preserve locality using locality-preserving hashing while also maintaining consistency and load balance.

At a minimum, the metadata includes enough information to use the data (e.g., structure, representation, and location of the data), but in general what is considered as metadata can be essentially open-ended. For example, the metadata could include:

1) Provenance of data, i.e., lineage of data including its origination and update history (e.g., who, when, what regarding the creation/update, access information, etc.)
2) Context, i.e., relationships across data items such as dependencies, relative popularities, purpose and usage of data, etc.
3) Hints on how data might be accessed (e.g., largely sequentially, random, certain patterns, etc.).

The key issue in maintaining extended metadata is its accuracy and overhead/scalability both in human and computing terms vis a vis usefulness. For example, keeping track of all accesses to "hot" data could easily make the metadata size much bigger than the data itself. Liu, et al. [25] address the problem of keeping the metadata indexing scalable by exploiting the provenance information.

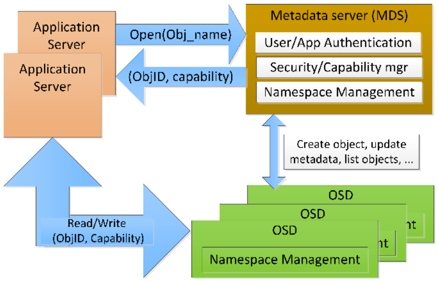

Fig. 2: Object Storage Infrastructure [26]

C. Raising the Storage Abstraction Level

With the traditional block storage discussed above, provides a very simple storage interface, but suffers from many disadvantages including poor performance, lack of security, and lack of connection between application's view of data and storage systems'. In fact, most storage systems still operate this way, with much of the intelligent data management handled above the storage layer. Such a view of storage is inherent in the "block storage", where the storage system merely deals with a sequence of "blocks", each identified by the logical block address (LBA), typically of 4KB size. The file system, built on top of block storage, does have some useful metadata such as the type of file, and could easily collect information about how the file is used, but none of this information is propagated to the block layer and beyond. With the simplistic situation of a local storage device managed by the host that runs the file system, the connection between the file access and block access is easy to make since the mapping of the file to the blocks is known. However, this is not true with most real systems where the devices may be managed by a separate storage server, and the communication between the host and the storage server is in terms of LBA numbers. Furthermore, many hosts and applications will typically share the storage server. Thus, the storage system only sees block level accesses and cannot relate them to the files, applications, or users. This state of affairs precludes any meaningful storage QoS provision since we do not know whether some blocks are more important than others.

The "object storage" model helps make storage more intelligent by raising the level of abstraction in client-storage interaction from blocks to objects. An object could be a file, file-segment, one or more database records, etc. but generally expected to be much larger than a block (e.g., 4MB) to make it efficient. The metadata associated with the object is stored separately in a metadata server (MDS) which has a few other functions as shown in Fig. 2. The MDS can authenticate the users/applications to prevent access by unauthorized parties. It also provides finer-grain access control to individual objects (e.g., read, update, append, etc.) by handing out a "capability" to the requester, which must be presented for data access. The MDS also does management of Object namespace. The data is stored separately on storage servers, each hosting one or more object storage devices (OSDs). The storage servers are usually distributed throughout the network. Multiple clients could be interacting with multiple storage servers (and OSDs) in parallel for the IO provided they have obtained the required capabilities to present to OSDs.

Each OSD does its own allocation of storage space to the objects and provides support for carrying out the metadata operations and providing any metadata updates to the MDS. The separate MDS makes it easier to maintain more varied metadata, which can include hints for OSDs regarding how data should be accessed. Migration of objects from one OSD to another becomes easier and transparent, since the Object IDs remain unchanged. The object storage model was proposed in early 2000's and because of its many advantages, it has become quite popular in large cloud and HPC data centers since it naturally supports distributed storage. Many distributed storage systems are currently in existence and widely used, including Lustre [27], Ceph [28], Gluster [29], HDFS [30].

D. Storage vs. Databases

Traditionally, storage and file systems formed the basis on which database management systems (DBMS) were built. For many decades relational database management systems (RDBMS) reigned over the database market and supported features such as well defined schemas, normalization to remove redundancies, complex SQL queries, and often ACID (Atomicity, (strict) Consistency, Isolation, and Durability) properties. This sophistication is out of step with the current trend of collecting all relevant data continuously and transforming it into value online or even in real-time. This means that the data is increasingly unstructured or semi-structured where a relational model is not appropriate. Also, the ACID properties are often unnecessary for such data, and too expensive to ensure due to huge size. Similarly, complex manipulations such as joins are both unnecessary and expensive. This has led to alternative database models, popularly known as NoSQL databases.

In contrast to RDBMS, NoSQL databases tended towards what became known as BASE (Basic availability, Soft state, Eventual Consistency) properties. These databases encompass a variety of data models such as the document, key-value, column and even graph models. NoSQL databases that use the document model don't require a fixed schema and allow the attributes in data records to change according to an application's need. The records can be hierarchical and may have a varying number of attributes. Well known databases with document models include CouchDB [31] and MongoDB [32]. Key-value stores, where a key (which can internally have a complex structure), keeps track of a blob of data, with a possible internal structure, but not necessarily a fixed schema.

Key-Value systems are becoming very popular, with significant examples being Google's LevelDB [33], Facebook's RocksDB [34], and our recent proposal called FlashKey [35]. On a high level, the column model gives the database flexibility in how records are defined and stored. The data records do not have a fixed number of columns, and in column-based database systems, each column can belong to a column family. This allows the database to store these elements together, with the expectation that they will be used together. Popular example include Cassandra [36] and HBase [37], [38]. Graph-oriented databases are usually included under the NoSQL umbrella, though some the driver behind their usage and innovation differs from other NoSQL approaches. They are useful for data that have relationships among its objects. This allows applications to traverses through these objects more naturally than what could accomplished with a relational model. Interestingly, some graph databases use other NoSQL databases as backends and some may include properties found in RDBMS such as ACID properties. Popular examples include Neo4j [39] and JanusGraph [40].

III. Key Issues in Efficient Big-Data Access

A. Role of Big Data

In the past, data rich computing was largely limited to scientific computing and deriving offline business intelligence. While the data requirements of these applications continue to increase, many other applications have emerged within the last decade. A prominent use case is the specialization of e-commerce to the needs of individual users using both the directly captured behavioral characteristics of the users and whenever possible, using their social network profiles and activities. Such data is being used for sentiment analysis, purchasing behavior, electronic media consumption behavior, search intent determination, etc. These, in turn, drive context specific advertising and bidding for that advertising, purchase recommendation systems, search result presentation, self-adaptive systems, etc., which need to work in an online manner with imperceptible additional delays. For example, to successfully enable context-specific advertising in e-commerce browsing, the system needs to determine the context of the user query, allow advertisers to bid, select the winning bids, retrieve and display the corresponding advertisements – all in the same amount of time as it takes to process the main query.

As the physical systems ranging from building to entire cities are infused with intelligence, they not only generate huge amounts of data, but also require that the data be analyzed online or even in real-time. For example, anomalies in power flows or traffic flows can be used to predict an impending problem and take corrective action. The emerging concept of edge and fog computing will involve continuous video monitoring and thus streaming data from large number of cameras that must be analyzed in real time at the edge nodes for anomalies/events and archived in the cloud for deeper offline analysis. Edge devices within a region may collaborate for situational understanding and the system would require effective means of dealing with the large amounts of data in real-time.

In most of these situations, the computation can be accelerated by using more computing power (e.g., fast multicore CPUs), although energy and space considerations may hamper that in some scenarios, particularly in the IoT environment. However, with significant amount of data needed for processing, the key difficulty is in proactively identifying the data required for processing and fetching it from storage devices into the memory so as to minimize stalls to the computation. Thus, data access latencies often dominate the performance. It is also important to note that it is no longer adequate to consider only the average latency of data access; increasingly, the focus is on *tail latency*, for example, the latency bounds for 90%, 99% and 99.9% of the cases. Effective control over tail latencies is much more difficult than targeting the average latency, but is increasingly being demanded by the industry.

Since accessibility of low-cost cloud computing eliminates the high initial investment in storage infrastructure and maintenance costs to ensure uptime, not all data may be stored on-premise. Cloud storage gateways (CSGs) allow the transparent access to cloud data as if it was in local storage. Similar mechanisms apply to managing data across different storage infrastructures such as decentralized storage towers in a data center, local storage on the server, storage on edge or micro data centers, etc. However, efficient methods to manage and deliver data with specified QoS (average and tail latency, throughput) and without errors or timeouts is essential to take advantage of the remote storage.

This has led to many innovations. To deal with large amounts of data, it is necessary to employ "scale-out" (i.e., storing data on multiple nodes), since the "scale-up" model (i.e., a bigger node) becomes quite expensive. Scaling out a system necessitates that the path data takes through the system becomes more complex. However, data partitioning must deal with the CAP theorem [41] in that a choice must be made between ensuring consistency of the data and its availability. Many features in consumer facing applications are able to get by with only approximate values in the data they present. For example, an application can display an approximate number of views for some multimedia artifact. This has led many NoSQL databases to prefer availability and eventual consistency over strict consistency in their design, while others, such as Cassandra is tuneable to be either more consistent or available depending on the usage scenario.

B. Role of Data Analytics

The volume of stored data continues to go up exponentially; however, the amount of data relevant for the computations at hand is often small and varies with time. This makes the identification of the currently relevant data increasingly challenging. At the same time, the advances in the technology have made computation increasingly efficient, to the extent that "feeding-the-monster" is becoming the central issue. In particular, the speed and energy consumption of data movement is becoming a limiting factor at all levels from on-chip caches all the ways to the lowest level in the storage hierarchy. *Given this state of affairs, spending extra computing power to identify the most relevant data and thereby reducing the overall data move volume and latency may be a winning strategy.* This is particularly true for the following three scenarios: (a) a slow device (e.g., a HDD), (b) high latency/low throughput path to the storage (e.g., access to remote cloud storage), and (c) large data transfers (and hence high latencies). However, with very fast emerging devices such as Intel Optane [20], the computational overhead could still be substantial and simpler schemes are better as shown in [42].

Data access in modern computing infrastructures goes through extremely tortuous path. For example, a user file is often chopped into *chunks*, with chunk size chosen as a compromise between the ills of small chunk size (high overhead of tracking the chunks) and that of large chunks (wasted space for small files and/or false sharing among files). Typical chunk size may range from 64KB to 4MB, but with multi-terabyte disks, large sizes become more attractive. The chunks are virtualized (so the chunks can be stored anywhere freely), and may be deduplicated to save storage, striped over multiple devices for parallel access, and eventually mapped to the devices, possibly using erasure coding for protection. The resulting 3 views of data, discussed above, become very difficult to connect together.

It is tempting to consider efficient data access techniques purely in logical terms, that is in terms of access to "data items" that an application uses, without regard to how and where those "items" are stored. For example, the application independently decides how it should request or prefetch data items that it needs. We note, that such an approach has multiple downsides:

1) It puts substantial additional burden on the application programmer in managing its IO, even if this tracking is done by making calls to some middleware routines.

2) Typically, the application interacts with the OS for its IO, and the latter often manages the IO autonomously. For example, unless unbuffered IO is requested specifically, the OS will use the buffer cache to prefetch data from the device, read data at different granularity than requested by the application, batch writebacks, etc. Thus a careful IO management by the application may not have the desired effect unless there is a robust mechanism for the application to indicate its preferences to the OS. Usually such mechanisms do not exist.

3) Since the application's actions are purely local, they have little influence on how the rest of the IO system operates. For example, the requests from different applications will be mixed up even at the file system level. Generally, there is no comprehensive mechanism to inform the OS about the application or application class so it can do appropriate QoS scheduling.

4) Generally, these front end actions, even by the file system, may not have much bearing on the storage system per se. Typically, the IO requests from applications running on a large number of servers will be interleaved and served by the storage system according to its polices. Thus, for example, a carefully timed prefetch request by the application, or even the local file system, may not be scheduled on the backend storage server as expected.

The most direct way to influence the storage system is to track the IO directly on the storage system side. In the traditional block storage, one could only analyze block accesses as a whole, and accordingly decide such things as migration of blocks across the storage hierarchy, prefetching of blocks in higher storage tier, optimal batching of writebacks of modified data, etc. Note that the overall traffic seen by the block storage devices may be generated by hundreds of applications, and the applications usually come and go. Thus, while intelligent management of blocks can be done easily, it is generally not possible to take application specific actions.

Given these challenges, the ability to discern useful characteristics using suitable analytics techniques becomes important. The success of such learning necessarily depends on the ability to observe the behavior over a sufficiently long period, patterns in the behavior that can be learnt, and relative stability of the behavior. This may not be true in some environments, and sophisticated analytics may not be useful or desirable. Thus, the key question in the data driven access is to understand the points where data analytics can be done, and what it would reveal.

Although the analytics can be used in many ways; some prominent ones are as follows: (a) Determine when certain types of applications or workloads may have started or ended, (b) predict accesses to each region of LBA space over the next (or next few) time slots, (c) correlate front-end (e.g., file system) actions with back-end (i.e., storage system), (d) discern and predict how the blocks (or chunks) move across devices, etc. Note that all of these tasks are made much harder by deduplication and virtualization.

C. Role of Optimal Configuration

Nearly every entity in a cyber or cyber-physical system has some set of "configuration parameters" that determine its behavior and performance. Not surprisingly, enabling high performance data access is crucially dependent on setting the relevant parameters correctly and optimally. Unfortunately, configuration management is one of the most difficult problems to tackle in real systems because of (a) very complex and poorly understood impact of configuration parameters on the performance and other characteristics of the system, (b) unknown or poorly understood dependencies between parameters, and (c) a constantly evolving environment where a "correct" configuration is both difficult to characterize and changing [43].

There are two key problems of interest relative to configuration parameters of a system: the forward problem (i.e., predict performance for a given configuration) or the backward problem (i.e., predict some configuration parameter(s) for a given performance target). An accurate analytic or simulation model for either of them would need to represent the interdependencies, which is difficult since the relationship is poorly understood and sometimes not even known. Nevertheless, such models can be very useful in evaluating influence of the most dominant parameters provided that we can adequately capture important interdependencies. For example, dependencies that are merely a result of queuing or resource contention can often be captured by standard queuing theoretic models [44].

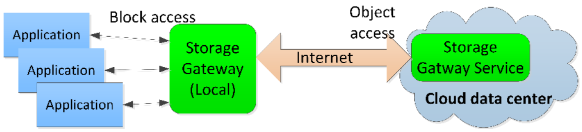

Fig. 3: Cloud Storage Gateway Architecture

Data driven techniques can be useful for capturing the dependencies indirectly and thereby yielding a better characterization. Of course, the downside is the need for large amounts of data and lack of insight into the results or the dependencies. So long as the workload shows substantial changes only occasionally, it is possible to train and retrain the model in the background without any impact on the system. Note that identifying optimal configuration parameters for a given performance target requires multi-label classification, which is generally harder and more error prone than single label classification (e.g., determining performance for a given configuration). Furthermore, since many configurations could potentially yield the same or similar performance but only a few of these may be desirable when seen from a domain knowledge perspective.

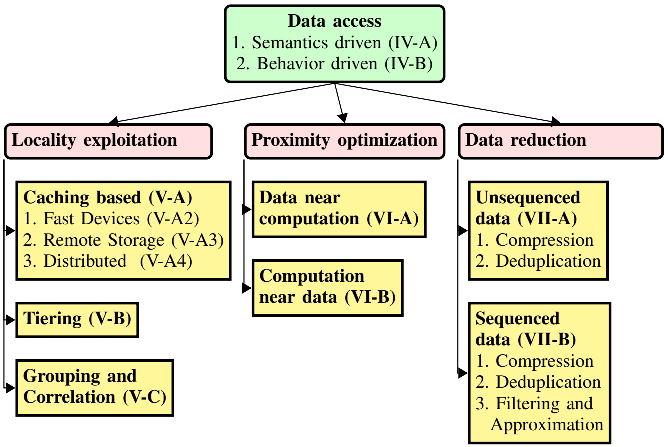

Fig. 4: Proposed Data Access Taxonomy

We illustrate this using the example of Cloud Storage Gateways (CSG) illustrated in Fig. 3. CSGs are increasingly popular since they create the impression of huge amounts of fast, local storage by providing an intelligent portal to the cloud storage. Typically, each client is allocated some amount of local storage space for data, metadata, and logging, and accesses like an ordinary "block" storage device, whereas the backend accesses the cloud object storage. Many configuration parameters relating to the amount of local storage space, local CPU resources, backend network characteristics, cache management policies, etc. are involved in meeting the client *service level agreement*. We have examined data driven characterization of CSG configuration using deep learning techniques in [45]. For configuration, we attempt to predict three labels, namely, data-cache size, meta-data cache size, and log size. It was found that the Extra Tree classifier was well-suited for multi label classification [46]–[48].

IV. A Taxonomy For Efficient Data Access Techniques

Fig. 4 shows our proposed taxonomy based on our view of many techniques that have be studied over the years for data access. For easy reference, it also shows the section numbers where they are discussed. First, there are 3 generic ways of making access more efficient:

1) *Locality Exploitation*, which decides what data to store in which part of the storage hierarchy based on the spatial and temporal locality of data accesses. This manifests into 3 types of techniques that we show at the next level in the tree, namely, caching, tiering, and grouping/correlation of data.
2) *Proximity Optimization*, where the goal is to bring the data and the computation using it closer together via

intelligent movement of data close to CPU or filtering out unnecessary data in/near the storage system itself.[2]

3) *Data Reduction*, or lossy/lossless compression, which brings in a trade-off between data size, access efficiency, and possibly the loss of information. The numerous techniques for doing so can viewed as belonging to two different classes: sequences, where the ordering of data items is important to retain (e.g., a time series) and where it is not (e.g., a complex 3-D object).

All of these techniques require exploring certain attributes of the data access and may have certain settable parameters that influence the behavior. For example, efficient caching requires discerning which data blocks will be needed soon, and which of them will be accessed close together in time. At the same time, there may be some parameters limiting how many blocks can be moved at a time. Similarly, data reduction needs to decide which part or aspect of the data is most important and thus must be preserved and this may be controlled by how much representational error is tolerable.

The key question then is to determine these attributes and the influence of input parameters. This can be done in two broad ways: (a) *Behavioral*, that depends on the observed behavior of the accesses on the same or similar systems, and (b) *Semantic*, which takes advantage of the deeper knowledge of the semantics and/or the context of the operations. We list both of these at the root of the tree in Fig. 4 since they apply to almost any technique. Semantic knowledge, when available, can invariably do a better job than simply observing behavior, but may not be known/observable, too expensive to discern, or will make the mechanism too inflexible or ad hoc. Thus a practical solution often is one based on behavioral observation, aided by some semantics related “hints” conveyed through various layers of abstraction.

In the following we describe the details of various mechanisms under the taxonomy and also point out the research challenges in each area. It is important to note that real systems may use combination of techniques that fall under different branches of our taxonomy tree, and may take advantage of semantic aspects to varying degrees. This combination itself may bring in new challenges, but our focus will be on individual techniques. As an example, if the storage system filters the data which is then cached close to the host, there are optimization issues if the filtering leaves out some data which must be later accessed by directly going to the storage.

A. Semantics Driven Optimizations

The general ways of exploiting the semantics include the following types of knowledge: (a) knowing how the data will be accessed (e.g., order of item access and size of access), (b) knowing about data “life-time” (i.e., for how long or how many times the data read before being updated), (c) knowing about different representations of the data that may be present in the system (e.g., original, curated, compressed, aggregated, filtered, etc.), (d) knowing about the data granularity or precision (i.e., to what extent data can undergo a lossy compression or reduction in resolution and still be useful), and (e) knowing the “value” of data, i.e., could the loss/corruption of a few data items in a large dataset make the entire dataset questionable or unusable?

Many of these semantic attributes are obviously known to the application designer, but there is no mechanism to convey this information to the OS or middleware. To be useful, the information needs to be conveyed using a standardized language, possibly through a configuration file. In the absence of such a mechanism, such attributes must be learned by observing the application behavior on the host side. However, in view of the semantic gap, the value of learning the application behavior is dubious unless we can also learn how the application will affect the traffic on the storage side. The analytics could be useful for learning the impact on long-lived applications with a core set of behaviors. In such a case, long term observation coupled with analytics techniques (e.g., a semi-supervised classification) could do a good enough job to be useful for grouping of data on the storage device or providing suitable QoS for storage accesses. A more direct approach can be taken, by providing direct hinting of the application class to the storage system.

Semantics driven optimizations have been explored in the literature for specific applications, but have not been dealt with in a more general context. Braun [49] proposes a semantics driven optimistic data replication to manage concurrency control overhead. Jauhar [50] offers an overview of a relation-centric view of semantic representation learning, while Noy [51] presents a survey of semantic integration. In the database context, semantics driven optimizations are used more extensively. More generally, a flexible and standard way of hinting about the contents, structure, and normal usage mode can go a long way in making data accesses efficient. However, this needs to be done in a way that is efficient to interpret and does not take up too much space. With object storage such information can be easily provided as metadata of the object and a few extra bytes to do it are perfectly acceptable. However, with block storage, there is no such provision. The block storage system such as disk array only has the intelligence to manage storage volumes and blocks within a volume (including space management, RAID operation, authentication, recovery, etc.). Generally, each major application will create and use one or more volumes exclusively, which itself may be virtualized.

The structure of the data often itself provides some information about how the data may be accessed. For example, access to a database will follow the key-based ordering of records. More generally, ontologies may be used to describe the structure, relationships among entities, and their semantics. Ideally, such description is provided by the application designer, but it may be possible to group applications based on their access characteristics and thus specify the structure and relationships more generically. These could then be translated into specific mechanisms for intelligent placement and access. Because of the difficulty of passing relevant information from host to the storage, this is very difficult to do in block storage; however, object storage can support such features. The key storage systems issue is then to encode metadata in MDS that would direct suitable storage allocation/placement to serve

[2] Although the filtering reduces the data size, it is still different from the “data reduction” aspect in that it does not alter the original data.

access requests efficiently. The efficient access could involve tuning the data fetch size from storage upon a user request (usually larger than the requested data amount) and prefetching of data that is likely to be accessed next.

B. Behavior Driven Optimizations

Behavior driven optimization relies on information that is inferred from behavior of data, e.g. read from access history. This can be obtained by observing data accesses, recording traces of workloads (so as to understand pattern i.e. behavior exhibited by the data) and by learning these patterns so as to make inferences about the data. Given the number of layers of abstraction and mapping from user's view to device's view, optimization of any given layer based purely on its observed behavior is an attractive approach, even if it is rather limited in its capabilities.

One way to exploit this behavior is to find repeating or regular patterns in the spatio-temporal characteristics of storage accesses. For example, consider the following *(offset, size)* sequence: (0, 200), (250, 400), (700, 300), (1100, 400). Here the offset depicts the starting byte offset from which the process is reading the file while the size denotes the length read from that offset. It is evident that in spite of slight irregularity in the starting address and access size, the file is essentially being read sequentially with few hundred bytes accessed at a time. This is a valuable information to learn as it can be used to prefetch the desired data. More complex patterns may also be observed – for example, there may be many such pseudo-sequential patterns in different offset ranges that repeat with perhaps random jumps between them. Such patterns can be exploited in three ways: (a) identification of pseudo-sequential streams when they occur the first time, (b) early recognition of streams when they repeat either exactly or approximately, and (c) prediction of which stream is likely to be encountered next based on relationships between them.

Such behavioral information can be useful both on the host side (e.g., to cache data in Operating System's buffer cache), and on the storage system side. In the latter case, the request stream is a mixture of requests coming from multiple applications and hosts and the requests are likely for LBA or chunks, rather than bytes from files. The storage system can exploit the patterns for various purposes including (1) simple prefetching mechanism in device's cache, (2) more sophisticated fetch/prefetch of LBAs in the storage system cache, (3) moving of data across tiers in the storage system, (4) proactive deduplication and staging of data, and (5) proactive transfer of data across the network closer to the host.

The patterns can also exploit temporal correlations unlike the locality aspect discussed previously. He et al. [52] talk about these patterns and distinguish between local and global patterns in different workloads. Global patterns are a mixture of the mentioned local patterns. Tran and Reed [53] elucidate more on the temporal correlations between accesses by discussing about bursty I/O traffic. These bursty patterns, which may occur due to nested loops, can cause buffer overflow and other network issues. Prefetching carried out on the basis of this behavior can actually reduce I/O stalls. They use standard time series analysis model to make calculations and predictions about the interarrival time of subsequent requests.

V. Locality Exploitation Techniques

Locality of access is key to efficient access to stored data at any level from CPU caches to the storage. In storage, locality drives not only caching but also placement of data both across storage tiers and inside a tier (i.e. where is the data located relative to other data). In the following we discuss these.

A. Caching Based Locality Exploitation

Caching is a vast topic and we do not intend to survey caching techniques in general but only discuss some specific ways in which caching exploits locality. A very recent survey of caching techniques may be found in [54].

1) Traditional Caching Mechanisms: Simple forms of locality such as frequent access to the same data or access to sequentially adjacent data are well known and exploited extensively at all levels. For example, LRU caching policy exploits recency of access whereas LFU caching policy exploits the frequency. Combinations of the two are exploited by the well known ARC (adaptive replacement caching) algorithm [55]. Similarly, identification of "streams" or strided access pattern is routine even in CPU caches and combined with ARC caching in the another well known SARC (Sequential ARC) algorithm [56]. SARC dynamically partitions the cache space in between sequential and random streams; the prefetching is performed only for accesses in the sequential streams to reduce the cache-miss rate.

More complex forms of locality identification concern relationships across different accesses or access patterns. For example, the well known AMP (adaptive multistream prefetching) algorithm [57] tracks the precedence relationships between accesses to drive prefetching. Dai et. al. [58] focused on locality properties of sequential blocks as they are more likely to be accessed together. He et. al. [52] go even further and claim that there are exploitable global and local patterns across storage data chunks in the wide variety of workloads they have explored. Patterson et al [59] and Kaplan [60] developed integrated policies for caching and prefetching.

Tombolo [61] discusses a way to adapt to different workloads by combining SARC and AMP [57] with a graph based algorithm called GRAPH. Tombolo uses AMP to prefetch for sequential access streams and GRAPH for random ones. The latter keeps track of what block ranges are likely to follow a given block range.

2) Caching From Fast Storage Devices: Traditional caching mechanisms for storage devices are characterized by a large difference in average request latency between the storage device (a HDD or SSD) and DRAM memory. Emerging nonvolatile memory technologies offer latencies that are much lower, but common caching algorithms do not take advantage of this aspect. We have addressed this by introducing "Fussy-Cache" (FC) [42] that does not blindly cache everything that's accessed, but instead caches only the popular items, while others are accessed directly from the device. FC also tries to keep things as simple as possible, since complex manipulations hurt the overall access latency when the devices are fast.

In addition to the regular data cache (DC), the key element in FC is a *Dynamic Metadata Cache* (DMC), that has two functions. First, it tracks the popularity of the requested items and thus determines which items should live in DC (rather than being accessed from the device directly). Second, it tracks if the mechanism is not doing well or not, and accordingly decides to use it or switch over to the normal (cache-everything) approach. The separation of cache into DC and DMC allows the two components to be executed by different threads thereby further reducing the latency.

For evaluating FC, we use Intel Optane as the storage device and 3 different traffic sources with very different characteristics, namely, Systor '17 traces [62], FIU Web Research trace [63], and the SPECSFS 2014 benchmark suite [64]. We consider the traffic from 2 different logical unit numbers (LUN0 and LUN1) for Systor trace. We also consider two important SPECSFS workload components, namely, database workload (DBTABLE) and SWBUILD (software build).

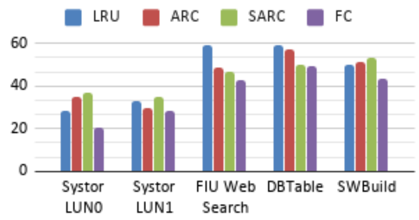

Fig. 5: Average latency with 2.5% Cache Size

We compare FussyCache against 3 highly popular caching algorithms, namely, LRU, ARC [55], and SARC [56]. Figs 5 and 6 show the comparison of average read latency for both Intel Optane with 2.5% and 10% cache size. It is seen that in both cases FussyCache does better than other algorithms.

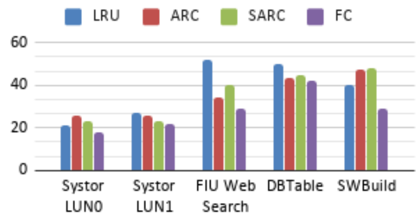

Fig. 6: Average latency with 10% Cache Size

3) Caching of Remote Storage: When the storage device is located remotely with a high access latency, it may be useful to achieve better hit rates at the cost of considerable computational overhead. A prominent example of this situation arises in the cloud storage gateway discussed in section III-C where the access latency is largely governed by the propagation delay (due to long physical distance to the cloud) and rather low WAN bandwidth to access the cloud. This is the motivation for our work on "BeliefCache" [65], [66] that exploits the notion of "belief" in determining what to cache or evict. Consider two entities X and Y and a time window W. Then the belief of X regarding Y relative to window W, denoted $B_W(X,Y)$, can be defined as a conditional probability. $B_W(X,Y)$ is the probability that a request to Y will occur within the time window W given that X was requested. BeliefCache computes these probabilities and based on those continuously decides which items should be prefetched into the cache and which cached items should be evicted.

Estimating such a belief would require maintaining and constantly updating the number of times in window W, an access to X was followed by access to Y. Note that for $Y = X$, such a count provides the access frequency of the entity X. Although in theory this means maintaining an $N \times N$ matrix of counts for N entities, this is extremely unlikely in real workloads [67]. In practice, many optimizations are possible to significantly reduce the space/time overhead and obviate explicit probability calculation in many cases, as discussed in [65], [66]. Such an approach is expected to be well suited for workloads that exhibit complex access patterns and long-term correlations.

BeliefCache's prefetching approach is similar to several methods [68], [69] that use a weighted graph to keep track of successor load requests. This also allows these methods to predict several requests using the weighted edges as a predictor of which objects will be requested next. A possible disadvantage is that rare requests may not be a good enough indicator of what follows next. In BeliefCache this issue is solved by using a window size to allow several requests to vote on what is a likely successor. That way even if the current request is rare, the previous requests can be used to vote for prefetching candidates. Moreover, these methods use a fixed prefetching degree. BeliefCache does not use a fixed prefetching degree and therefore it adapts better to the workload changes that other comparable algorithms.

If the workload changes substantially, BeliefCache will need to reset its hyperparameters, which are originally learnt in the training phase. This is accomplished by using the concept of virtual cache. A virtual cache is one that observers all accesses, determines the prefetch and eviction candidates, and gives them to the real cache as an "advice". The real cache deals with the actual job of prefetching/eviction and could decide to ignore the advice if desirable (e.g., eviction of pinned items). BeliefCache actually uses two virtual caches, one that continues to train the background and the other that currently provides the advice. These can be switched if a phase change is detected.

4) Distributed caching: Traditional caching mechanisms cannot support the needs of the largest systems in production. The issue is that the cost of resources such as memory and storage space does not increase linearly with capacity. Furthermore, a single cache would be a bottleneck as the number of requests increases. One way to deal with this is to provide caches on or near the application servers. This has the added benefit of reducing the network latency of any request. As user load is distributed among different application servers, the working set that would ideally be stored in cache, may be located on a different application service. A solution is to use a global service dedicated to caching, which adds some network latency but allows caching service to be scaled independently of the other components of the system. This is a common problem and there are many popular solutions developed such as memcached [70], [71] which gives distributed in-memory cache and more recently redis [72] provided many additional features such as the ability to persist the cache to handle failures.

One technique to allow the capacity of distributed caching to grow online by adding extra nodes without much disruption is to shard the data based on consistent hashing [73]. This technique, which is not restricted to caching, allows the addition and removal of nodes as needed with minimal disruption to the cache data store on other servers. Since some of the popular data may be used frequently from multiple places, replicated caching can be beneficial if the workload is read-dominated, but otherwise keeping the caches consistent can be make it very expensive.

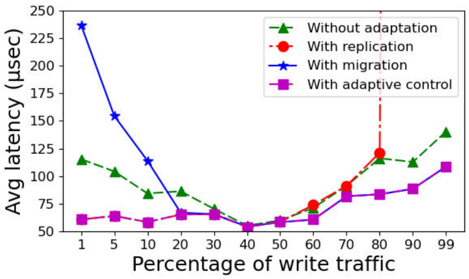

Fig. 7: Latency Reduction with Adaptive Copy and Migration

In [22], we have studied a mechanism to handle increase in demand by activating additional copies of data chunks that are pre-distributed and synchronized in the background. Fig. 7 shows how the overall IO latency varies with read/write ratio under various situations. Initially we place the chunks at different storage nodes by considering the IO demand on the node and proximity of the chunks to the applications using them. We then study how the latency is affected by the increase in overall demand. The case "without adaptation" does not take any dynamic action. The "replication" case creates additional copies of the heavily demanded chunks at nodes in the vicinity that have a relatively low load, and then splits new traffic between all the copies by modifying the virtualization map. The "migration" case still makes a copy of the busy chunk at a relatively quiet node, directs all new traffic to it, and eventually deactivates the old copy when all ongoing requests to it complete. When the traffic subsides, the original situation is reverted so as to avoid scattering copies throughout the network.

As expected, the replication works well for low write ratio, but provides poor latency for write dominated traffic due to synchronization traffic and delays. In contrast, migration works well with high write ratio but is poor for read dominated traffic. Fig. 7 also shows the adaptive policy that dynamically decides whether to use the replication or migration policy (based on current read/write activity). It is thus able to exploit the best behavior regardless of the read/write ratio.

B. Locality Exploitation for Tiering

Conceptually, the locality considerations for tiering are the same as for caching, the significant difference being that the tiering concerns coarser time granularity (or longer term decisions). For example, it would not be prudent to move a data item to a higher tier (and delete it from the lower tier) on the first reference, since the cost of a wrong decision can be rather high (i.e., need to re-allocate and writeback the data to the lower tier and in the process hurt the endurance of the device). Thus, the locality most relevant for tiering is not the recency but frequency and relationships across items in terms of frequency. Traditionally, cross-tier migration of data is triggered very infrequently (e.g., once a day or opportunistically during low traffic hours), but with very fast devices and virtualized storage, a much more agile tiering (e.g., one operating at the granularity of minutes or tens of minutes) may be appropriate.

A popular concept in tiering is "heat", which is another way to describe the popularity (or frequency of access) of the data chunks over a specified time granularity. Since the popularity can vary over time, the challenge is to predict the "heat" both in spatial and temporal terms. The more contextual information provided to a heat prediction mechanism, the more tractable heat prediction becomes. At the highest level, it is possible to develop standard mechanisms by which the applications can specify what parts of their accessed data are likely to be popular or unpopular. Such a mechanism is workable as a static information but much harder to convey dynamically. There is also the problem of burdening the application in identifying the "hot" data, which may require some "profiling" runs. An additional problem is that the concept of heat is most relevant for the storage system which may be serving many applications, each with different notion of what is hot.

The file system can take up the burden from the application and do profiling automatically to build relationships between the objects [74], [75] that can be used to predict future accesses. The key problem is that this requires the file system to know the detailed mapping of files to physical chunks and their detailed statistics. While each file-system can do this for its "virtual volume" that hosts that file system, a unified storage device level view will be missing. An example of this approach is Online OS-level Data Tiering [76] which sorts data chunks for each tier based on their degree of randomness, read ratio, and request type using a weighted priority function.

In view of these issues, it is useful to run heat prediction directly on the storage side based on the observed behavior. This would correctly provide the "heat" of all the chunks on a tier; however, it will be unable to distinguish between the applications corresponding to the chunks; therefore, no application based QoS is possible without passing some explicit end to end hints (the issue of hints is discussed in section VIII-D). Also, the mixture of chunks from numerous applications, many of which may come and go over the period of minutes or longer could make the heat prediction quite challenging.

In [77], we have attempted to do heat prediction based on machine learning. For scalability, we examined the popularity of rather large size chunks (8MB) for the entire storage system over time windows of size 2.5 minutes. We then group these windows into a small number of groups based on some "signals" that indicate similar behavior. These include read/write ratio, read intensity, and dispersion (or range of chunks accessed). The grouping is necessary both to obtain enough data for subsequent machine learning model and to reduce the overall complexity. Subsequently, we build a Random Forest (RF) prediction model for each group to make predictions. The results show that:

1) The low end of activity of chunks can be predicted highly accurately. This is useful in tiering since such chunks can stay in the lower tier.

2) On the high end of the activity, the prediction is feasible but the accuracy drops to 60-80%.

3) The prediction includes the range of the expected number of requests which could allow the prioritization of actions.

The heat prediction could be exploited for tiering, where the goal is to dynamically direct each entity to its most appropriate or "optimal" tier subject to the following constraints:

(a) Constraint on the fraction of IO capacity of each device used for tiering.
(b) Constraints on IO latency and/or IO throughput requirements specified at a suitable level (overall or types of entities, whenever possible.)
(c) Endurance and write amplification related constraints for SSDs or other NVRAM technologies.

Much of the work on tiering in the literature is based on static on semi-static profiling. For example, Cloud Analytics Storage Tiering [78] uses offline workload profiling for each tenant in the cloud to build a data placement and storage provisioning plan. An example of a dynamic mechanism is AutoTiering [79], which dynamically migrates virtual disk files (VMDK) across tiers based on the sampling of the IO rates. It tests the latency sensitivity of a VMDK by injecting latency and examining its effect on IOPS. This allows the more sensitive VMDKs to be migrated to higher tiers and less sensitive to lower tiers.

C. Grouping and Correlations

Locality can also be used for grouping items together whether physically (i.e., contiguous or nearby placement on a disk or a node) or logically (i.e., identified for prefetching together). Grouping could be used as a way to address scalability of caching algorithms since groups can be thought of as pseudo-applications, so that we only need to apply resource intensive learning techniques within these pseudo-applications. Grouping could be structural (e.g. Block, Object/File, Groups of files, etc.), or based on detailed entity access history [80]

Applications may introduce temporal correlations among blocks that are not physically close to each other. For example, the B-tree or B*-tree data structures in database systems will introduce continuous accesses between blocks that store parent nodes and child nodes. The Ext3 or Ext4 Linux file systems also introduce temporal correlations between blocks where the file data is stored and the blocks corresponding to inode is stored. Multiple applications that interact with each other also introduce block correlations. Such applications commonly exist in the form of workflow in scientific computing. Wildani and Miler [80] explored a wide variety of workloads and identified consistency of groupings in the data despite the sparsity of the traces. Kroeger and Long [81] provide strong evidence that some workloads have consistent and exploitable relationships between file accesses.

The notion of provenance is useful as another aspect of context in data driven mechanisms for grouping. Provenance is the metadata that represents the history or lineage of a data entity, its use, derivation and updates. This provides important semantic information about the data [25], [82]. Provenance information can improve belief calculation, potentially reduces the analytics overhead and improve decision making; however, collecting, maintaining, and using the provenance information itself involves substantial overhead. Thus a difficult and open problem is to characterize what provenance information should be collected and maintained so that it can lead to overall benefit in analytics.

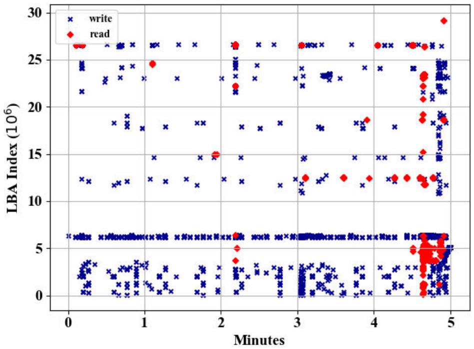

Fig. 8: A scatter plot of LBA chunks getting accessed over a five minute period extracted from the hardware monitoring workload. The red diamonds represent read requests and the blue xs represent write request.

Additional aspects of context that could be used for grouping and caching come from what can be called an *IO slice*, i.e., an executable subset of the entire program that does only those IOs that are needed to decide all other IOs and only that part of program logic that is relevant for IOs. The IO slice can be run in parallel with the main program and because it is potentially much smaller, it can thus aid IO prediction. IO slice is useful if it does much less IOs as compared to the actual program. The challenge is in analyzing, learning and deciding which parts of program logic are relevant for IOs. Techniques [83], [84] use IO slicing in parallel applications and show promising results in reducing IO access latency.

In Fig. 8 we see both read and write requests from a public trace [85] over a five minute period. It shows the LBA chunk in which the requests happens but does not capture the number requests made to that chunk. One of the challenges of predicting heat on the block level is the size of the prediction space. The combination of chunks that may be requested is extremely large and processing previous requests must be fast for the prediction to be useful. One approach to dealing with the large space is to focus prediction on some regions while ignoring others. A possible way to do this would be to focus prediction efforts on regions of high level of read and write requests. Another path that might be taken is to focus the prediction on regions that are reliable to predict and avoid those that have more random access patterns. For example regions with steady accesses may be more reliably predicted than regions that usually see little activity and sudden large and irregular bursts of read/write requests.

A well-performing heat prediction mechanism could be used for guiding caching decisions, grouping of the entities and as a part of data movement control mechanisms.

D. Open Issues in Locality Exploitation

Locality exploitation typically relates to discovering data access relationships over short periods of time; however, many workloads exhibit behavioral patterns over multiple, longer time frames, often reflecting the schedules (e.g., incremental backup) or workflow characteristics. Yet, capturing such long term relationships is challenging because of the overhead of storing and analyzing such data in an online manner. In particular, the combinations that can be considered a ”pattern” increases very rapidly with the time horizon. Furthermore, the pattern is often perturbed by ”noise”. Also, the process that controls data requests is likely a non-stationary phenomena, thus decreasing or removing any predictive power that a captured pattern may give.

While exploiting locality is important, it may need to be balanced with other considerations as well. For example, with TLC/QLC SSDs, the endurance considerations may require foregoing movement of certain data even if it is desirable from the perspective of exploiting locality. Also since much of the storage access is remote, network traffic, and increasingly the network latency, becomes an important consideration in data movement directed by locality of access considerations. For example, even if we know that next 100 LBAs will be needed soon, it may not be desirable to prefetch them all in a single IO because of its impact on overall network traffic or the latency experienced by other traffic, particularly if the other traffic demands a better QoS. Accounting for such conflicting aspects becomes quite challenging, particularly in a dynamic environment with changing traffic types.

VI. Proximity Optimization Techniques

Over the last two decades, the speed and energy efficiency of computing has progressed more than that of data storage, which increasingly makes the data movement the primary concern in terms of both performance and energy consumption. Thus, **proximity optimization** could help in placing the required data close to computation (by proactively putting it in the upper levels of the memory and storage hierarchy) and even bringing computation close to data (by embedding processing in the storage and memory devices).

Although computing technology has achieved tremendous gain in speed over the last few decades, the enhancements have been uneven in that computation speed has increased much more than the memory or communications capacity. Fig. 9 shows the increasing energy cost of data movement at all levels. This along with consistently growing data volume implies that minimizing data movement is key to high performance and energy efficiency in data intensive applications. This goal can be achieved by employing proximity optimization in two ways: (a) fetching the most needed data closest to CPU (or the highest level of the storage hierarchy) just in time for its use, and (b) embedding intelligence into the storage infrastructure itself to minimize fetching of unnecessary data.

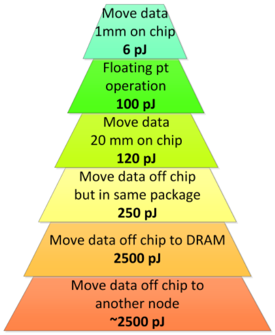

Fig. 9: Energy cost of data movement

A lot of work in storage systems relating to data caching, tiering, and placement is geared towards (a) and direct or indirect attempts to segregate data items by their popularity. Given the large amounts of data required by data intensive applications, it is usually not possible to move the data around significantly in its lowest tier after the initial placement. Therefore, initial placement of data in the lowest tier is crucial. Tiering will generally move a small subset of this data to higher tiers. We discuss initial placement and tiering in the next two subsections. The idea of intelligent storage, though very old, is less developed but is lately getting much attention, and is discussed subsequently.

A. Data Near Computation

Large scale storage systems may use multiple levels in the storage hierarchy, but the lowest layer in the primary storage is likely to consist either HDDs or, going forward, perhaps QLC SSDs). A new data set, especially a large one, will almost inevitably be created on this layer initially. Furthermore, the physical location of the data (i.e., the drives or storage servers hosting it) is likely to be handled transparently by the storage system with little regard to where or how the data will be processed. Moving an entire data set subsequently closer to computation is not only impractical but also disruptive (i.e., may take too long or require eviction of other data). Automated tiering is obviously one way to deal with this as discussed above – the data that is proven to be active (or is predicted to have “high heat”) is brought into the higher tiers dynamically.

An alternate method is to take advantage of data/application semantics so that the most essential part of the data can be identified in advance of running the application and brought into the higher tier. This requires the knowledge of the structure of the application and data. A partial automation is possible here by using a machine readable description of the program and data structure, along with indication of what portions of data are needed when. The last part could be specified or learned from prior runs, but the latter works only for environments where the data usage pattern remains similar across runs (e.g., HPC applications).

A special case of the above arises when even the data has two or more representations with adequate semantic information to decide which data is requested and used. For example, it may be useful to create both native (or high) resolution and low resolution versions of image data. Similarly, for database records, it may be useful to have native version and a leaner version where the rarely needed records and/or rarely needed fields of records are removed. This filtering can be described easily for relational databases, e.g., maintaining one or more alternate materialized views based on specific selection, projection and even join properties. Graph databases can also be similarly thinned out and so can more general ones so long a clear structure and semantics information exists. In all of these cases, a knowledge of the representations used by different applications could allow for their intelligent scheduling coupled with prefetching of the correct data versions.

With computation becoming rather inexpensive and data volumes growing, a proper consideration of computation-storage trade-off can be very useful. In particular, some of the intermediate results need not be stored at all, and instead recreated if needed. Woodman et al. [86] explore this possibility and present an approach which is able, via a collection of past performance and provenance data, to make decisions based on the underlying storage and computation costs as to which intermediate data to retain and which to regenerate on demand.

B. Computation Near Data

Traditionally, storage devices have been considered passive elements that simply store/retrieve the requested data. However, today storage devices are much more complex and have computational power and intelligence in managing IO on the device [87]. For example, SSDs need a fairly capable processor to run FTL functions. Since nearly all of the emerging storage technologies are "solid state", it is easy to integrate computing with storage. In fact the 3D architectures have been proposed that integrate computing, memory and flash in different layers. Even without such chip-level integration, processing embedded in the device electronics can allow for a whole range of selective data retrieval, filtering, and even data reduction options.

Li [88] provides an overview of the next generation storage systems that will do processing in storage. While the notion of intelligent disk is somewhat old [89], a considerable amount of research exists on the subject under "active-storage" [90]. More recent works have shown how "computational storage" can be exploited in the context of distributed processing [91] and big-data analytics [92].

Embedding intelligence in the storage necessarily requires richer metadata that can be used to decide how to process the data. In this context, the object storage model lends itself well to embedding of intelligence. In particular, a rich object metadata can be used to encode the semantics of the objects and hence their efficient access. Two recent examples of embedding processing in object storage are the Oasis system at UC/Santa-Cruz [90] and at U/Conn [93]. Similarly, the kinetic drives introduced by Seagate [94] can do some key related processing. Reddy [95] presents another form of intelligent storage that presents different "views" much like a database does. Such an approach can be used to reduce the data according to the application's needs on the fly. For example, the stored structured data may have many fields, many of which are unnecessary for the running application. Thus instead of creating another version of the data with the unneeded fields removed, the intelligent storage device or server can do the filtering on the fly. Similarly, data records with certain attributes may be unnecessary and can be discarded by the storage device/system rather than bringing them into memory and then not using them.

When the amount of data is extremely large and distributed on multiple nodes, specific frameworks are required to process the data largely in-place. The Hadoop framework [96] is an ecosystem that relies on distributed file-system (HDFS) [97], [98] to ensure that data is durable. It is designed for large scale batch processing without moving the data around. This is done by performing the computation where the data is and only moving reduced form of the data to consolidate towards a result. The development of Hadoop programs is done within the map-reduce programming paradigm [99] where computation is done on data that is local to CPU.

C. Open Issues in Proximity Optimization

Traditionally, active storage techniques have been proposed to move computation tasks to storage nodes in order to exploit data locality. However, this introduces the side effect that the computing can only scale out with the number of storage nodes. Also, sharing the compute power of the storage device/server can produce interference in the storage system. An alternate technique offered in [100] is to allow users to create small, stateless functions that intercept and operate on data flows. This disaggregated processing is similar to the network function virtualization (NFV) [101], and can be useful when the functions are placed on the path of natural data movement, or when the amount of data involved is small so that the data can be specifically sent through the function. In general, however, this alters the original vision of the storage system doing the processing.

There are several other challenges as well. Foremost among these are interlinked issues of flexibility and security. Allowing arbitrary code to be loaded into storage devices brings in many security vulnerabilities and makes it difficult to manage the functionality. A device that can do arbitrary computation on the data before returning or storing it no longer has the simple read/write interface and complicates consistency, recovery, space management, metadata management, etc. In contrast, well-defined and possibly standardized set of data operations implemented inside the storage system are easier to verify, secure and manage, but limit flexibility in processing potentially resulting in additional overhead in matching programmer's intent with storage provided functions. An even more difficult issue concerns what can be termed as "completeness"; if certain functionality is not available inside the storage system and must be implemented by bringing large portions of the stored data to the compute server, performing other functions inside the storage system may only serve to increase the overall energy budget. In other words, finding a suitable set of abstractions that are broadly useful, efficiently implementable, and complete is a grand challenge for intelligent storage systems.

VII. Data Reduction Techniques

An important attribute for data reduction in storage systems is the mode of data access. *Unsequenced data* can be accessed in any order (e.g., files and database records), and (b) *Sequenced data* where the access is inherently sequential (e.g., a time-ordered access trace from the storage system). These two require very different techniques as discussed in the following.

A. Redundancy Removal in Unsequenced Data

We start with data compression, which can be either *lossless* and *lossy*. Lossless compression has wide applicability and is commonly used since it does not lose any aspect of the data. A recent survey of various computation techniques, both lossy and lossless, can be found in [102]. The key benefit of compression in the storage context is the reduced data movement which can reduce the storage related IO, access latency, and the network traffic. Compression at the level of chunks of suitable size provides much greater flexibility than

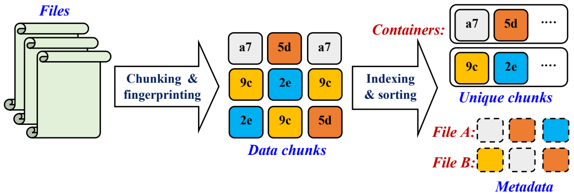

Fig. 10: Illustration of the deduplication process [104]

file compression since only the required chunks need to be decompressed. Note that while smaller chunks provide more flexibility, the corresponding lower compressibility and higher management overhead may not provide adequate reduction in access latency. Also, if the compression is done internally by the storage system or the device, it does not reduce the network traffic, though it may still reduce the access latency. It is worth noting that the compression with the granularity of blocks or very small chunks is usually not desirable for two reasons: (a) low compressibility and (b) data transfer time being only a small fraction of the overall access latency. In any case, compression does save storage space, which is important at higher levels of storage due to the cost issues.

The key problem with compression is the unpredictability of compression ratio which complicates the storage management. Since the compression ratio highly depends on the contents, it is quite possible that 4 LBAs worth of data initially compresses to one LBA, but a slight update to the data makes the compression much poorer. This would make the contiguous allocation of compressed LBAs very difficult, particularly for technologies that allow overwrite of data which includes HDDs and most emerging technologies. However, for SSDs, this is less of a problem since any updates would have to be placed elsewhere. Compression is actually used internally by the SSD firmware and the strategy is to accumulate compressed LBAs in a NVRAM buffer before writing to the SSD.

Lossy compression is routinely used for audio, video, and image data where the small degradation in the quality does not materially affect the usefulness of the data or the results obtained from using such data. For example, the jpg image format allows for various levels of compression depending on the size of the resulting image. Image compression methods are surveyed in [103] and more general discussion of lossy compression is contained in [102]. Depending on the type of data and its usage, several forms of lossy compressions can be used. For example, with data organized as a set of records, lossy compression may amount to either thinning the data out based on a sampling technique or removing records that are rarely requested. Both techniques carry risks not only in terms of removing important data but also in terms of potential misbehavior of the programs that consume them.

Data deduplication finds duplicates in the data and stores only one copy. It can be applied at the file-level [105] or the chunk level [106] where the "chunk" size is usually bigger than the LBA size. While simple, file level deduplication is not very effective since it requires completely identical files, and thus chunk level deduplication is normally used. For efficiency, the chunks are not compared directly to find duplicates, instead, the comparison is based on the "fingerprint" of the chunk. The fingerprint is computed using a hash scheme such as SHA1 or SHA256 which has an extremely low probability of collision. Fig. 10 illustrates the fixed chunk sized based deduplication where a file or object is divided into chunks of a fixed size. Only unique chunks are stored; duplicate chunks simply point to the stored chunks. A problem with fixed size chunking of files/objects is that small inserts at the beginning of the file/object would change all of the chunks. Variable size chunking, where the chunks are identified by Rabin fingerprinting over the contents. Variable size chunking is much more expensive and also complicates storage management because of variable size of chunks produced. Variable sized chunking is largely used for backup storage where the stored data is never modified. Kaur et al. [107] provide a systematic review of data deduplication techniques for efficient cloud storage management.

Given the proliferation of multiple versions of the same file, deduplication is usually quite effective and can reduce the storage requirements by 60-80%. The main problem with deduplication is that it tends to scatter the chunks of a file since a chunk N of file X could match the stored chunk M of some arbitrary file Y and thus requires accessing this chunk. While the resulting randomization is a huge issue with HDDs due to their poor random access performance, it is not much of a problem with SSDs. Comprehensive studies of deduplication techniques are conducted in [104], [107]–[111].

Current approaches do not take into account semantic deduplication, such as avoiding different representations of the same object. For example an object and a losslessly compressed version of that object represent the same underlying entity and both need not be stored. However, determining such relationships can only be done by the file system, not by the storage system. As discussed in section VI-A, it may be desirable to create smaller versions of a dataset (e.g., by reducing resolution or eliminating some information). If these relationships are available to the system it is possible to exploit them not only for tiering but also for reducing redundancy when necessary.

B. Redundancy Removal in Sequenced Data

In many situations, the stored data consists of a sequence of data accesses (with or without explicit time of access). Some examples of such data include the data generated by IoT devices that monitor a cyber-physical system, incoming user queries and responses generated by a computer system, social media logs, outputs from scientific experiments, etc. In fact, so much of the stored data can be characterized as a time series, that it is crucial to look for special techniques for reducing such data. Although, one could treat such data in the same way as data in general and apply deduplication and lossless compression; the time series nature of the data can be exploited to keep the behaviors or data access patterns that are of interest.

Generalized filtering of time-series can substantially reduce the data volume and yet retain the desired information to satisfy the given data use objectives. Such objectives necessarily relate to the data semantics and depend on the applications that operate on the data. While it is possible to specify the needs of an application and use them to create a data "view" for that application, this only addresses the problem of efficient access to data by that application, as mentioned in section VI-A. This would still not allow deletion of the original data and thus does not solve the storage problem. To solve the storage problem, it is necessary to define all potential needs and yet be able to produce a representation that satisfies these needs and is still much smaller than the original data set to justify transforming the data. This presents a huge challenge in both affirming a usage model that can anticipate all reasonable needs and making it work in terms of data reduction potential. The specific issues to consider in data filtering are:

1) What are the primary use cases and what information each use case needs to preserve?
2) What information is collectively sufficient to satisfy all specified use cases in step 1?
3) What are the potential ways of compressing the data to preserve the desired information in step 2?
4) How can we characterize the minimal representation of the data to preserve the desired information in step 2?

These questions are rather open-ended and thus not possible to tackle in general. For example, the primary use cases for the data are obviously dependent on the data, the applications, the context, and to an extent on the state of the technology. Take for example the increasing amount of data generated by Twitter. In recent years, Twitter data has received an intense interest in understanding the topics being discussed by its users. But the interest understanding also extends to sentiments and mood analysis, understanding the ongoing situation or event (e.g., disaster, accident, etc.), the relevance of the tweet to the event, etc. Even if the use cases can be defined explicitly, the question of minimal representation of data to preserve the desired information is an exceedingly difficult problem. Therefore, here we only discuss some techniques that can be useful in data filtering.

To start with, let us consider a slight extension of generic compression schemes. We would like to divide the time-series into segments with the aim of identifying a suitable (lossy) compression technique to use based on how much compression we can tolerate. Such techniques are discussed in the literature under the area of time series approximation and representation [115]. Several techniques like Fourier transform [116], wavelet transform [117], piecewise polynomials [118], singular value decomposition [119] fall in this category. Fig. 11 illustrates an example of a piecewise linear approximation of "koskiecg.dat" dataset [112]. The amount of information needed to represent data is much smaller using the approximation and it is able to faithfully reconstruct the original data.

These techniques require a method for varying compression over successive data segments. For example, each segment may represent one year's worth of data, and as we go into the past, we use more compression. In such techniques, the compression is not homogeneous, rather heterogeneous across time. In [114], [120] the authors have modeled such aspects as "amnesic" functions, that represent recent data with better precision than the past. The key motivation of using amnesic function is that the usefulness of data fades over time. This is true for many real-life applications, including finance, meteorology, network management etc [121]. Thus given a limited memory budget a practical representation of data storage is to approximate the recent data with higher accuracy, whereas accuracy can be gracefully degraded over time. Fig. 12 illustrates an example of such piecewise linear online amnesic function, where the approximation of the recent points are finer compared to the older ones.

In our recent work [122], we considered pattern mining based compression of sequenced data, such as the data coming out of IoT devices. The method preserves the information about the sequence of events. We call this as Approximate Vector Stream Compression (AVSC). AVSC starts with a sequence compression method proposed in [123] called SQS (Summarizing event seQuenceS). SQS efficiently discovers high-quality patterns that summarize the data well and correctly identify key patterns. SQS is approximate in that it does inexact matching, i.e. patterns are allowed to have gaps, but the objective function heavily penalizes long gaps.

C. Open Issues in Data Reduction

A key challenge in data reduction is simultaneously satisfying the needs of multiple applications which may conflict. Consider a data stream D, and the optimally reduced streams $R_i = f_i(D)$, $i = 1..K$ for the K applications of interest that apply the reduction function f_i for ith application. Ideally, we seek a reduction function f such that the reduced stream $R = f(D)$ provides a significant compression over the original stream D and yet it retains all the information necessary to generate all R_i's. This is quite difficult, in general, since each f_i's may preserve different properties of the data stream, and requiring them simultaneously may be feasible only by keeping several (or even all) R_i's, which may be no better than keeping D itself.

In most activity monitoring situations involving both cyber-systems (e.g., storage IO, network or processing activity in data center) and cyber-physical systems (e.g., building surveillance, road traffic monitoring, etc.), the collected data may be of interest only when some abnormal or unexpected events occur. For example, the cameras watching traffic at a road intersection may need to send the video stream only when there is congestion, accident or other problems. However, determining if these conditions must be either done by the processing associated with the cameras or the reduced data must still allow for this determination. In the latter case, if the criteria changes dynamically, it is necessary to automatically adapt the data reduction technique for it, but this can be very challenging.

With ever increasing rate of data generation, there is the emerging serious problem of retaining the data over long periods of time. Retaining data for long periods involves many challenges [124]. First and foremost, it requires a mechanism to increasingly reduce the data volume, possibly including reduction in its metadata and provenance data, as it ages. This requires a framework for expressing the use cases for the data

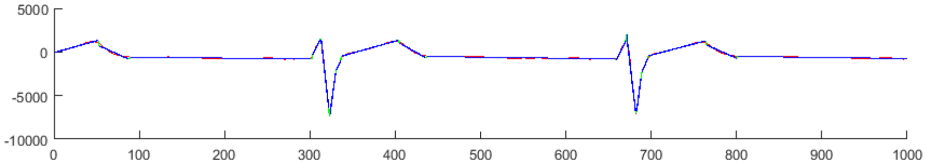

Fig. 11: Piecewise linear approximation of koskiecg.dat dataset [112]. The code obtained from this linear approximation is obtained from [113].

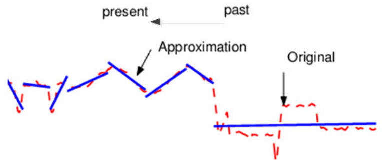

Fig. 12: Illustration of an online amnesic approximation [114].

and how those use cases change with data age. Unfortunately, not all possible uses of the data may be known in advance, and even with a few disjoint use cases put together, the total volume of reduced data may be no smaller than the original raw data. Other challenges include long-term tracking of data, its representation, devices it is stored on, retaining software (e.g., drivers) that will continue to work [125], and dealing with technology specific "bit rot" problem. The last issue is getting worse as the feature size continues to go down; for example, current low end SSDs may start to have retention problems beyond a year [126].

VIII. Other Considerations in Efficient Data Access

Although the focus of this paper is on data access performance, it is essential to balance it against other important needs of the application domain. This section provides a discussion of some of the relevant tradeoffs.

A. *Performance vs. Data Consistency and Device Life*

Most applications update the stored data dynamically – whether the original dataset or the intermediate data created during processing. Since the hottest data is often in the DRAM, a machine crash or power failure would not only lose the data in DRAM, but could also result in the stored data being inconsistent since only parts of the update to the data and corresponding metadata made it to the device.

The data loss itself can be minimized by immediately persisting any updates in memory to nonvolatile storage. However, this may result in a lot of random, small writes to the storage, which are usually very inefficient. Furthermore, if the media has limited endurance (e.g., QLC SSD), they could wear out the media quickly. Both of these problems can be addressed by introducing a fast, high-endurance NVRAM cache in between the storage and DRAM. For example, we have shown in [127] that by introducing a small amount of high endurance technology buffer (e.g., Intel Optane) between DRAM and QLC SSD, both the performance and the overall endurance can be enhanced substantially [127]. However, there is a limit to such a scheme. If the user issued IOs are very small (e.g., few bytes written or updated at a time), persisting each update/write into the NVRAM could be detrimental to NVRAM's lifetime. Thus, in reality, there is a always a trade-off between risk of data loss and performance/endurance aspects.

The inconsistency problem is more serious than small data loss as it may make the results of further computations incorrect. Inconsistency can arise at multiple levels; the lowest one being due to lack of data and corresponding metadata making it to the disk fully before a crash happens. The most basic reason for this is that all data writes involve some form of metadata update as well. This aspect is traditionally handled through "journaling", or essentially a write-ahead logging of metadata [128]. It can have a substantial performance impact, particularly, if it goes to great lengths to avoid mishaps due to multiple points of buffering, batching, and reordering that are common in modern storage systems. The problem becomes more complex with distributed storage systems where multiple clients may be simultaneously updating a shared file or database. Some systems handle this by relaxing the consistency constraints (e.g., the popular network file system or NFS), while others prohibit local data caching (e.g., PVFS2/OrangeFS) or resolve it using complicated distributed lock managers (e.g., Lustre) [129]. In particular, NFS provides no guarantee of consistency except for a very weak open-to-close consistent writeback of modified data by clients [130].

The key reason for inconsistency is lack of atomic updates. Atomicity in the emerging persistent memory can be achieved via "transactional memory", i.e., ensuring an all or nothing semantics for user level operations, which can be efficiently implemented in hardware [131]. Apache Spark provides another approach to atomicity via its *resilient distributed dataset* data structure [132]. It divides data into partitions that are distributed across nodes for parallelism, and a RDD can only be transformed from one immutable version to the next via computations that are remembered and repeated in case of crashes.

B. *Performance vs. Resilience*

In addition to the volatility, data loss/corruption can occur due to storage media issues such as device failures, uncorrectable IO errors, and a host of errors that limit durability of data stored in flash and other emerging technologies [18]. Such errors are expected to become more serious with increasing storage density and media capacities. There are two generic techniques for handling these: (a) erasure coding of data to allow for complete loss (or "erasure") of one or more data items out of a "stripe" of N items, and (b) maintaining multiple copies of data items and keeping them updated. Both of these create potential for data inconsistency due to machine crashes/power failures, that must be handled. Erasure coding is popularly used in RAIDed disks, but can be used in a distributed storage environment as well [133] by selecting a "stripe" of n data items (e.g., blocks or chunks) over which the parity items are constructed. For example, a RAID6-like arrangement can tolerate two failures/corruptions by adding two parity items to the stripe. The key challenge is in selecting how to form the stripes [134] since any update to a stripe item would require update to the parity items.

If a machine crash occurs during update of a erasure coded stripe, the stripe may no longer be consistent. Similarly, in case of multiple copies, a crash during update may leave some copies un-updated and therefore inconsistent. Note that in case of both erasure coding and multiple copies, journaling will likely only ensure that that individual data chunks have a all or nothing semantics, and thus additional mechanisms are needed to achieve consistency at the higher level.

As storage costs decline, maintaining multiple copies is becoming quite popular. The presence of multiple copies can be exploited to enhance the read bandwidth and reduce the network traffic. However, this requires that the copies are located close to the points of greatest demand, else the movement of copies will incur additional network traffic and latency. Furthermore, if these copies are updated, the synchronization overhead of maintaining strict consistency can negate any advantages. If the read/write ratio changes dynamically, an active adjustment of the number of copies may be needed as discussed in section V-A.

Although a significant data loss or corruption is always unacceptable, certain applications (e.g., those doing statistical processing) may not be affected by the loss/corruption of a few items. However, the corresponding metadata should still be handled correctly since certain metadata problems (e.g., misdirected pointer) can lead to large scale data corruption. Thus, an application specific use of fault-tolerance mechanisms can be useful but can be quite difficult to implement unless the different application classes are segregated on different devices.

C. Performance vs. Energy Consumption

The actions taken to enhance performance invariably affect energy consumption. For example, data filtering, compression, deduplication all enhance performance and hence reduce per-operation energy consumption. However, this is usually done to squeeze in more operations, and thus the overall energy consumption may not be lowered. The same applies to minimizing data movement through tiering, caching, and data placement techniques. These techniques, can, however, be also exploited to maintain a given energy budget without reducing performance. Since more active copies means higher energy consumption, it may be important to control the number of copies based on the energy limitations. In [135] we describe such a mechanism for the replicas of deduplicated storage. This work concerns the deduplication and replication of virtual machine (VM) images. The VM images are divided into an optimal number of groups based on a variety of factors such as deduplication potential of a group (i.e., overlaps in the chunks used by the VMs), performance/resilience requirements for the VMs, placement restrictions for VM images, etc. The number of active copies (or replicas) of each group could then be varied dynamically based on the limitations on how much total power can be consumed (possibly limited by cooling/thermal issues or power draw itself).

D. Differentiated Services in Data Access

In most situations, there is a trade-off lurking even within the scope of efficient data access – this is the need to provide better service (e.g., lower latency and/or higher throughput) to some workloads at the expense of others. Another way to state this requirement is the need to support quality of service (QoS) features across different classes of applications. QoS becomes relevant when the available resources are limited. In the case of distributed storage, this involves the host side, the network and the target side, and can be quite complex.

With object storage, providing simple QoS features is rather straightforward. When a user or application makes a request for the object to the MDS, it provides its credentials and the operation. Based on this the MDS can assign the request to a specific QoS class, which will be served accordingly by the OSD. The simplicity comes from the fact that both the application/host and the storage side speak the same language, that is of objects, which have globally unique IDs. Nevertheless, a complex mapping of objects can make the QoS more challenging.

With block storage, the situation is far more complicated and end-to-end QoS only has some proprietary implementations [136]. The main difficulty, as stated in section I, is the semantic gap and lack of connection between application and storage sides. In particular, the storage side does not know to which application a specific block belongs to or what kind of QoS should be provided to it. Recently, there have been some efforts to convey some "hints" from the application and/or the file-system to the storage system concerning the treatment of IO using a small number of unused bits in the IO command headers [137].

Because of the rather slow nature of storage systems in the past, much of the focus, as in this paper, has been on simply making the data access as fast as possible *on the average.* However, with the emergence of very low latency storage technologies, the goalpost has shifted further – we want quick access not just on the average, but we want it with very high probability! In particular, currently there is an ongoing effort to define "deterministic period" features in NVMe standards which allows an SSD to declare periods where its FTL will not do background activities such as wear leveling, garbage collection, etc. and thereby provide deterministic latency. Achieving deterministic latency for all latency sensitive QoS classes end-to-end is still a very challenging and open problem.

E. Performance vs. Security

There are a variety of issues that fall under this realm. They include tradeoffs that are made to attain different levels of data protection and privacy. The tradeoffs made when attaining the desired level of protections must not be made in isolation, rather the threat model [138], [139] must be defined. This is because raising the protections has an impact on complexity, cost, and performance and without foresight may not achieve the desired objective. For example, great effort may be made to protect data in motion or securing the underlying infrastructure [140], but if insider threat is a possible vector for having the data exposed and is not addressed, putting all the investment in infrastructure may be a poor choice. In addition to securing data in motion and at rest, big data brings a host of privacy [141] issues [142]. In essence, privacy leakage can take the form of a direct exposure of data. This could be through misconfiguration, poor design, security breaches, etc. It can also be indirectly, through processed or aggregated data. In the former case, design decisions may be made to make to protect privacy information at increased performance cost in retrieving the relevant data when needed. In the latter case, extra processing of the data may be needed, whether the data is in motion or at rest, to prevent leakage. With the vast amounts of user information being collected, privacy leakage can affect a large number of users.

IX. Conclusions

In this work we have provided a comprehensive survey of various facets of efficient big data access, and devised a taxonomy for numerous techniques that have been considered in the literature. Owing to the rather complex relationships between the three key views of data, namely user's view, storage system's view, and device's view, and the need to coordinate between them, achieving high performance requires sophisticated methods including the big-data analytics techniques themselves.

The proposed taxonomy considers three key aspects of efficient data access, namely exploitation of spatio-temporal locality of data access, enhancing the proximity of the data and the computation working on it, and the reduction (or compression) of data depending on which features of the data are important for the computation. By viewing a data access problem through the lens of this taxonomy, we can understand the relevant issues and possible solutions related to that problem. This allows us to address the needs of the problem by looking at techniques, tools, and technologies in other domains that may transfer to a particular problem. It also prompts us to consider the representation of data and its movement across the system, factoring in a range of considerations, from low level issues such as behavior of individual bits of data close to the hardware and abstracting out to consider emergent behavior of the system. Furthermore the need to trade off performance against other aspects mentioned in section VIII make the overall problem very challenging.

A fundamental issue in approaching the problem of efficient data access or the tradeoffs with other requirements is what we know about the semantics of the situation, i.e., the nature and representation of data items of interest, how they will be used, relationships/dependencies between different data items, how are the three different views of data connected, etc. In most cases, we do not possess this knowledge, and instead such knowledge must be gleaned from the behavior and behavioral relationships, perhaps via big data analytics techniques. Even when such knowledge is available, its expression and representation in a usable form is a huge challenge. We believe that in the foreseeable future, the issues of expressing, representing, and exploiting the semantics aspects of data and relating them to behavioral aspect (including the understanding of semantics through behavioral exploration) will assume increasing significance and would both exploit and contribute to efficient big data analytics techniques. We hope that the structured treatment of the subject of big data access, the taxonomy, and the research challenges discussed in this paper will spur researchers to further examine those issues in the space of rapidly emerging data intensive applications.

Acknowledgments

This paper includes the work of many current and former students including Dusan Ramaljak (developer of BeliefCache and data summarization ideas), Sanjeev Sondur (cloud storage gateway and configuration management issues), Lu Pang (heat prediction issues), Tanaya Roy (storage hierarchy issues), Jit Gupta (FussyCache), and Madhurima Ray (network congestion issues). We would also like to thank Ayman Abouelwafa (HPE), Tony Floeder (General Dynamics), and Jeremy Swift (Dell) for many stimulating discussions, and NSF for supporting this work through the grant IIP-1439672.

References

[1] C.-W. Tsai *et al.*, "Big data analytics: a survey," *Journal of Big Data*, vol. 2, pp. 1–32, 2015.

[2] N. A. Ghani *et al.*, "Social media big data analytics: A survey," *Comput. Hum. Behav.*, vol. 101, pp. 417–428, 2019.

[3] C. Castellanos *et al.*, "A survey on big data analytics solutions deployment," in *ECSA*, ser. Lecture Notes in Computer Science, T. Bures *et al.*, Eds., vol. 11681. Springer, 2019, pp. 195–210.

[4] X. Chen *et al.*, "Big data deep learning: Challenges and perspectives," *IEEE Access*, vol. 2, pp. 514–525, 2014.

[5] A. Oussous *et al.*, "Big data technologies: A survey," *Journal of King Saud University - Computer and Information Sciences*, vol. 30, no. 4, pp. 431 – 448, 2018.

[6] S. Mazumdar *et al.*, "A survey on data storage and placement methodologies for cloud-big data ecosystem," *Journal of Big Data*, vol. 6, no. 1, feb 2019.

[7] N. Khan *et al.*, "Big data: Survey, technologies, opportunities, and challenges," *The Scientific World Journal*, 2014.

[8] Y. Lv *et al.*, "Traffic flow prediction with big data: A deep learning approach," *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 865–873, 2015.

[9] G. Aceto *et al.*, "Know your big data trade-offs when classifying encrypted mobile traffic with deep learning," in *TMA*, S. Secci *et al.*, Eds., 2019, pp. 121–128.

[10] M. Pandey *et al.*, "Mobile applications in context of big data: A survey," in *CDAN*, 2016.

[11] M. Mohammadi *et al.*, "Deep learning for iot big data and streaming analytics: A survey," *IEEE Commun. Surv. Tutorials*, vol. 20, no. 4, pp. 2923–2960, 2018.

[12] J. Archenaa *et al.*, "A survey of big data analytics in healthcare and government," *Procedia Computer Science*, vol. 50, pp. 408–413, 2015.

[13] G. Harerimana *et al.*, "Health big data analytics: A technology survey," *IEEE Access*, vol. 6, pp. 65 661–65 678, 2018.

[14] R. A. Alshawish *et al.*, "Big data applications in smart cities," in *ICEMIS*, 2016, pp. 1–7.

[15] Y. Liang *et al.*, "Urbanfm: Inferring fine-grained urban flows," in *ACM SIGKDD*, 2019, pp. 3132–3142.

[16] L.-M. Ang *et al.*, "Big sensor data applications in urban environments," *Big Data Research*, vol. 4, pp. 1–12, 2016.

[17] Y. Liu *et al.*, "Predicting urban water quality with ubiquitous data," *arXiv preprint arXiv:1610.09462*, 2016.

[18] Y. Cai *et al.*, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1666–1704, 2017.

[19] S. Mittal *et al.*, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2016.

[20] J. Izraelevitz *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[21] H. Strass, "An introduction to nvme," https://www.seagate.com/files/www-content/product-content/ssd-fam/nvme-ssd/nytro-xf1440-ssd/_shared/docs/an-introduction-to-nvme-tp690-1-1605us.pdf.

[22] M. . Ray *et al.*, "Adaptive data center network traffic management for distributed high speed storage," *IEEE LCN*, 2019.

[23] H. J. Singh *et al.*, "Scalable metadata management techniques for ultra-large distributed storage systems – a systematic review," 2018, p. 82:1–82:37, ACM Comput. Surv.

[24] Q. Xu *et al.*, “Efficient and scalable metadata management in eb-scale file systems,” 2014, p. 2840–2850, IEEE Transactions on Parallel and Distributed Systems.

[25] J. Liu *et al.*, “Using provenance to efficiently improve metadata searching performance in storage systems,” *Future Gener. Comput. Syst.*, vol. 50, pp. 99–110, 2015.

[26] “SNIA Object Store: Tutorial: Everything You wanted to Know About Storage,” https://www.snia.org, 2018.

[27] P. Schwan *et al.*, “Lustre: Building a file system for 1000-node clusters,” in *Linux symposium*, 2003, pp. 380–386.

[28] S. A. Weil *et al.*, “Ceph: A scalable, high-performance distributed file system,” in *USENIX OSDI*, 2006, pp. 307–320.

[29] R. H. Inc., “Product documentation for red hat gluster storage 3.5,” 2019. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_gluster_storage/3.5/

[30] Apache, “Hdfs architecture guide,” 2019. [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[31] J. C. Anderson *et al.*, *CouchDB: the definitive guide: time to relax*. ”O’Reilly Media, Inc.”, 2010.

[32] K. Chodorow, *MongoDB: the definitive guide: powerful and scalable data storage*. ”O’Reilly Media, Inc.”, 2013.

[33] M. Minglani *et al.*, “Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers,” in *IEEE ICPADS*, 2017.

[34] S. Dong *et al.*, “Optimizing space amplification in rocksdb,” 2017.

[35] M. Ray *et al.*, “Flashkey: A high-performance flash friendly key-value store,” *IPDPS*, 2020.

[36] A. Lakshman *et al.*, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[37] A. Khetrapal *et al.*, “Hbase and hypertable for large scale distributed storage systems,” *Dept. of Computer Science, Purdue University*, vol. 10, no. 1376616.1376726, 2006.

[38] L. George, *HBase: the definitive guide: random access to your planet-size data*. ”O’Reilly Media, Inc.”, 2011.

[39] J. Webber, “A programmatic introduction to neo4j,” in *SPLASH*, 2012, pp. 217–218.

[40] “Janusgraph. distributed, open source, massively scalable graph database.” [Online]. Available: https://janusgraph.org/

[41] E. Brewer, “Cap twelve years later: How the” rules” have changed,” *Computer*, vol. 45, no. 2, pp. 23–29, 2012.

[42] J. Gupta *et al.*, “Fussycache: A caching mechanism for emerging storage hierarchies,” *Submitted for publication*, Oct 2020.

[43] T. Xu *et al.*, “Systems approaches to tackling configuration errors: A survey,” 2015.

[44] K. Kant, *Introduction to computer system performance evaluation*. McGraw-Hill, 1992.

[45] S. Sondur *et al.*, “Towards automated configuration of cloud storage gateways: A data driven approach,” *Proc. of Cloud 2019, San Diego, CA*, June 2019.

[46] S. R. Safavian *et al.*, “A survey of decision tree classifier methodology,” *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.

[47] M. S. Sorower, “A literature survey on algorithms for multi-label learning,” *Oregon State University, Corvallis*, vol. 18, 2010.

[48] G. Tsoumakas *et al.*, “Multi-label classification: An overview,” *IJDWM*, vol. 3, no. 3, pp. 1–13, 2007.

[49] S. Braun, “Semantics-driven optimistic data replication: Towards a framework supporting software architects and developers,” 2017, IEEE ICSAW.

[50] S. K. Jauhar, “A relation-centric view of semantic representation learning,” 2017, ph. D. thesis, Carnegie Mellon University.

[51] N. F. Noy, “Semantic integration: A survey of ontology-based approaches.” *SIGMOD Record*, vol. 33, no. 4, pp. 65–70, 2004.

[52] J. He *et al.*, “I/o acceleration with pattern detection,” 2013, HPDC.

[53] N. Tran *et al.*, “Automatic arima time series modeling for adaptive i/o prefetching,” 2004.

[54] M. Hoseinzadeh, “A survey on tiering and caching in high-performance storage systems,” *arXiv preprint arXiv:1904.11560*, 2019.

[55] N. Megiddo *et al.*, “Arc: A self-tuning, low overhead replacement cache,” in *USENIX FAST*, 2003, pp. 115–130.

[56] B. S. Gill *et al.*, “Sarc: Sequential prefetching in adaptive replacement cache.” Washington, D.C., 2005, p. 293–308, Proceedings of USENIX ATC.

[57] B. S. Gill *et al.*, “Amp: Adaptive multi-stream prefetching in a shared cache.” Washington, D.C., 2007, p. 185–198, Proceedings of FAST.

[58] D. Dai *et al.*, “Vectorizing disks blocks for efficient storage system via deep learning,” 2018, parallel Computing.

[59] R. H. Patterson *et al.*, “Informed prefetching and caching,” in *ACM SOSP*. New York, NY, USA: ACM, 1995, pp. 79–95. [Online]. Available: http://doi.acm.org/10.1145/224056.224064

[60] S. F. Kaplan *et al.*, “Adaptive caching for demand prepaging,” in *ACM SIGPLAN Notices*, vol. 38, no. 2 supplement. ACM, 2002, pp. 114–126.

[61] S. Yang *et al.*, “Tombolo: Performance enhancements for cloud storage gateways,” Hoes Lane Piscataway, NJ, 2016, p. 1–14, Proceedings of MSST.

[62] S. I. T. Repository, “Systor’17 fujitsu laboratory traces.” [Online]. Available: http://iotta.snia.org/traces/4964

[63] “Snia iotta trace repository, fiu srcmap trace.” [Online]. Available: http://iotta.snia.org/traces/414

[64] S. SPEC, “Benchmark, 2014.”

[65] D. Ramljak *et al.*, “Belief-based data prefetching and replacement in storage systems,” 2017, USENIX FAST.

[66] D. Ramljak *et al.*, “Modular framework for data prefetching and replacement at the edge,” Cham, 2018, p. 18–33, Proceedings of EDGE.

[67] J. Oly *et al.*, “Markov model prediction of i/o requests for scientific applications,” New York, NY, 2002.

[68] L. Lin *et al.*, “Amp: an affinity-based metadata prefetching scheme in large-scale distributed storage systems,” in *IEEE CCGRID*, 2008, pp. 459–466.

[69] P. Gu *et al.*, “Nexus: a novel weighted-graph-based prefetching algorithm for metadata servers in petabyte-scale storage systems,” in *IEEE CCGRID*, 2006, pp. 409–416.

[70] B. Fitzpatrick, “Distributed caching with memcached,” in *Linux Journal*, Aug. 2004.

[71] R. Nishtala *et al.*, “Scaling memcache at facebook,” in *USENIX NSDI*, N. Feamster *et al.*, Eds., 2013, pp. 385–398.

[72] M. Paksula, “Persisting objects in redis key-value database,” *University of Helsinki, Department of Computer Science*, 2010.

[73] D. Karger *et al.*, “Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web,” in *ACM symposium on Theory of computing*, 1997, pp. 654–663.

[74] P. Pipada *et al.*, “Loadiq: Learning to identify workload phases from a live storage trace,” 2012, USENIX HotStorage.

[75] N. J. Yadwadkar *et al.*, “Discovery of application workloads from network file traces.” 2010, p. 183–196, FAST.

[76] R. Salkhordeh *et al.*, “Operating system level data tiering using online workload characterization,” *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1534–1562, 2015.

[77] L. Pang *et al.*, “Data heat prediction in storage systems using behavior specific prediction models,” *IEEE IPCCC*, 2019.

[78] Y. Cheng *et al.*, “Cast: Tiering storage for data analytics in the cloud,” in *ACM HPDC*, 2015, pp. 45–56.

[79] Z. Yang *et al.*, “Autotiering: automatic data placement manager in multi-tier all-flash datacenter,” in *IEEE IPCCC*, 2017, pp. 1–8.

[80] A. Wildani *et al.*, “Can we group storage? statistical techniques to identify predictive groupings in storage system accesses,” 2016, ACM Transactions on Storage (TOS).

[81] T. M. Kroeger *et al.*, “Design and implementation of a predictive file prefetching algorithm.” 2001, p. 105–118, USENIX Annual Technical Conference, General Track.

[82] E. Bertino *et al.*, “A roadmap for privacy-enhanced secure data provenance,” 2014, p. 481–501, Journal of Intelligent Information Systems.

[83] Y. Chen *et al.*, “Exploring parallel I/O concurrency with speculative prefetching,” in *ICPP*, 2008, pp. 422–429.

[84] C. S. Smowton, “I/o optimisation and elimination via partial evaluation,” University of Cambridge, Computer Laboratory, Tech. Rep., 2014.

[85] D. Narayanan *et al.*, “Write off-loading: Practical power management for enterprise storage,” 2008, v. 4, n. 3, p. 10, ACM Transactions on Storage (TOS).

[86] S. Woodman *et al.*, “Workflow provenance: an analysis of long term storage costs,” in *ACM WORKS*, J. Montagnat *et al.*, Eds., 2015, pp. 9:1–9:9.

[87] R. Kaplan *et al.*, “Prins: Processing-in-storage acceleration of machine learning,” 2018, v. 17, n. 5, p. 889–896, IEEE Transactions on Nanotechnology.

[88] D. Li, “Processing in storage, the next generation of storage system,” 2019.

[89] K. Keeton *et al.*, “A case for intelligent disks (idisks),” 1998, p. 42–52, SIGMOD Rec.

[90] Y. X *et al.*, “Oasis: An active storage framework for object storage platform,” 2016, p. 746–758, Future Generation Computer Systems.

[91] M. Torabzadehkashi *et al.*, "Compstor: An in-storage computation platform for scalable distributed processing," in *IEEE IPDPS Workshops*, 2018, pp. 1260–1267.

[92] M. Torabzadehkashi *et al.*, "Catalina: In-storage processing acceleration for scalable big data analytics," 2019, euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP).

[93] M. T. Runde *et al.*, "An active storage framework for object storage devices," Hoes Lane Piscataway, NJ, 2012, p. 1–12, Proceedings of MSST.

[94] S. Inc., "Seagate kinetic hdd," https://www.seagate.com/www-content/product-content/hdd-fam/kinetic-hdd/en-us/docs/100764174b.pdf, 2015.

[95] X. Ma *et al.*, "Mvss: an active storage architecture," 2003, p. 993–1005, IEEE Transactions on Parallel and Distributed Systems.

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

[97] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, no. 2007, p. 21, 2007.

[98] K. Shvachko *et al.*, "The hadoop distributed file system," in *IEEE MSST*, 2010, pp. 1–10.

[99] J. Ekanayake *et al.*, "Mapreduce for data intensive scientific analyses," in *IEEE International Conference on eScience*, 2008, pp. 277–284.

[100] J. Sampé *et al.*, "Data-driven serverless functions for object storage," New York, NY, USA, 2017, p. 121–133, ACM/IFIP/USENIX Middleware Conference.

[101] B. Han *et al.*, "Network function virtualization: Challenges and opportunities for innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[102] J. Uthayakumar *et al.*, "A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications," *Journal of King Saud University-Computer and Information Sciences*, 2018.

[103] G. Vijayvargiya *et al.*, "A survey: various techniques of image compression," *arXiv preprint arXiv:1311.6877*, 2013.

[104] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," 2016, p. 1681–1710, Proceedings of the IEEE.

[105] H. Hovhannisyan *et al.*, "Whispers in the cloud storage: A novel cross-user deduplication-based covert channel design," 2018, p. 277–286, Peer-to-Peer Networking and Applications.

[106] C. L. P. Chen *et al.*, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," 2014, p. 314–347, Inf. Sci.

[107] R. Kaur *et al.*, "Data deduplication techniques for efficient cloud storage management: a systematic review," 2018, p. 2035–2085, The Journal of Supercomputing.

[108] N. Mandagere *et al.*, "Demystifying data deduplication," 2008, p. 12–17, ACM/IFIP/USENIX Middleware.

[109] J. Paulo *et al.*, "A survey and classification of storage deduplication systems," 2014, p. 11:1–11:30, ACM Comput. Surv.

[110] M. Fu *et al.*, "Design tradeoffs for data deduplication performance in backup workloads," 2015, p. 331–344, USENIX FAST.

[111] D. T. Meyer *et al.*, "A study of practical deduplication," 2012, v 7, n 4, p. 14:1–14:20, TOS.

[112] Keogh *et al.*, "iSAX: Indexing and mining terabyte sized time series ecg data," 1996. [Online]. Available: https://www.cs.ucr.edu/~eamonn/iSAX/koski_ecg.dat

[113] Keogh *et al.*, "Segmenting time series," 2008. [Online]. Available: http://bearcave.com/software/market_trading/intraday_trading/SegmentingTimeSeries.html

[114] T. Palpanas *et al.*, "Online amnesic approximation of streaming time series," Hoes Lane Piscataway, NJ, 2004, p. 339–349, IEEE ICDE.

[115] S. Imani *et al.*, "Matrix profile xiii: Time series snippets: A new primitive for time series data mining," 2018, IEEE ICBK.

[116] C. Faloutsos *et al.*, "Fast subsequence matching in time-series databases," 1994, p. 419–429, ACM SIGMOD.

[117] I. Popivanov *et al.*, "Similarity search over time-series data using wavelets," 2002, p. 212–221, IEEE ICDE.

[118] B.-K. Yi *et al.*, "Fast time sequence indexing for arbitrary lp norms," 2000, p. 385–394, VLDB.

[119] K. Chakrabarti *et al.*, "Locally adaptive dimensionality reduction for indexing large time series databases," *ACM Trans. Database Syst.*, vol. 27, no. 2, pp. 188–228, 2002.

[120] S. Gandhi *et al.*, "Space-efficient online approximation of time series data: Streams, amnesia, and out-of-order," Hoes Lane Piscataway, NJ, 2010.

[121] P. Missier *et al.*, "Preserving the value of large scale data analytics over time through selective re-computation," 2017, p. 65–77, Data Analytics.

[122] D. Ramljak *et al.*, "Pattern mining based compression of iot data," New York, NY, USA, 2018, p. 12:1–12:6, Proceedings of the Workshops ICDCN.

[123] N. Tatti *et al.*, "The long and the short of it: summarising event sequences with serial episodes," in *ACM SIGKDD*, 2012, pp. 462–470.

[124] F. Luan *et al.*, "A survey of digital preservation strategies," *World Digital Libraries-An international journal*, vol. 3, no. 2, pp. 133–150, 2010.

[125] B. El Idrissi, "Long-term digital preservation: A preliminary study on software and format obsolescence," in *ACM ArabWIC*, 2019.

[126] A. Cox, "Jedec ssd specifications explained." [Online]. Available: https://www.jedec.org/sites/default/files/Alvin_Cox[CompatibilityMode]_0.pdf

[127] T. Roy *et al.*, "Enhancing endurance of ssd based high-performance storage systems using emerging nvm technologies," *High-Performance Storage Workshop, IPDPS*, May 2020.

[128] V. Prabhakaran *et al.*, "Analysis and evolution of journaling file systems." in *USENIX ATC*, vol. 194, 2005, pp. 196–215.

[129] J. Valerio *et al.*, "Evaluating the price of consistency in distributed file storage services," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer, 2013, pp. 141–154.

[130] O. Kirch, "Why nfs sucks," in *Linux symposium*, 2006.

[131] A. Shahid *et al.*, "Hardware transactional memories: A survey," in *Innovative Research and Applications in Next-Generation High Performance Computing*. IGI Global, 2016, pp. 47–65.

[132] M. Zaharia *et al.*, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX NSDI*, S. D. Gribble *et al.*, Eds., 2012, pp. 15–28.

[133] S. Balaji *et al.*, "Erasure coding for distributed storage: an overview," 2018, science China Information Sciences.

[134] A. L. N. Reddy *et al.*, "Design and evaluation of gracefully degradable disk arrays," 1993, p. 28–40, Journal of Parallel and Distributed Computing.

[135] M. Murugan *et al.*, "Software defined deduplicated replica management in scale-out storage systems," *Future Generation Computer Systems*, vol. 97, pp. 340–354, 2019.

[136] C. Naddeo, "End-to-end quality of service: Cisco, vmware, and netapp team to enhance multi-tenant environments," https://www.netapp.com/fr/communities/tech-ontap/tot-smt-qos-201001-hk.aspx, 2019.

[137] M. Mesnier *et al.*, "Differentiated storage services," in *ACM SOSP*, 2011, pp. 57–70.

[138] A. Shostack, *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[139] S. Myagmar *et al.*, "Threat modeling as a basis for security requirements," in *SREIS*, vol. 2005, 2005, pp. 1–8.

[140] Y. Demchenko *et al.*, "Big security for big data: Addressing security challenges for the big data infrastructure," in *Workshop on Secure Data Management*. Springer, 2013, pp. 76–94.

[141] E. Bertino *et al.*, "Big data security and privacy," in *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*. Springer, 2018, pp. 425–439.

[142] D. S. Terzi *et al.*, "A survey on security and privacy issues in big data," in *IEEE ICITST*, 2015, pp. 202–207.

Anis Alazzawe has received both a B.S. in computer science and M.S. in software engineering from George Mason University. He has received his Ph.D. from Temple University in 2019. His interests include energy and resilience issues in HPC and machine learning.

Amitangshu Pal received the B.E. degree in computer science and engineering from Jadavpur University, in 2008, and the Ph.D. degree in electrical and computer engineering from The University of North Carolina at Charlotte, in 2013. He is currently an Assistant Professor of Computer and Information Science Department at Temple University. His current research interests include wireless sensor networks, reconfigurable optical networks, smart health-care, cyber-physical systems, mobile and pervasive computing, and cellular networks.

Krishna Kant is a Professor with Temple University, Philadelphia, PA, USA. His current areas of research include sustainability and energy efficiency in data centers, configuration robustness and security, and application of computing technologies to larger sustainability problems. He has published in a wide variety of areas in computer science, has authored a graduate textbook on performance modeling of computer systems, and has co-edited two books on infrastructure and cloud computing security. He is a Fellow of the IEEE.

Efficient Big-Data Access: Taxonomy and a Comprehensive Survey (Supplemental Materials)

Anis Alazzawe, Amitangshu Pal, Krishna Kant
Computer and Information Sciences, Temple University, Philadelphia, PA 19122
[aalazzawe, amitangshu.pal, kkant]@temple.edu

I. Emerging Storage Technologies

In this section, we briefly survey the essential characteristics of the emerging storage technologies, particularly as they relate to efficient data access issues.

A. Magnetic Recording Storage Technologies

The most widely deployed storage technology currently is the traditional hard disk drive (HDD); however, this technology is by no means static and is itself undergoing rapid advancements. Much of the action is in advanced magnetic recording techniques which includes many techniques in different stages of development. For example, perpendicular recording (i.e., magnetization perpendicular to the disk surface) is already well established. Other techniques include 3D recording (i.e., being able to select one of several layers and reading/writing data on it), heat assisted magnetic recording (HAMR), microwave assisted magnetic recording (MAMR), heated dot magnetic recording (HDMR), etc. The HAMR technology, already demonstrated to product 16TB drives, heats the point where the magnetic pole is to be flipped to a high but sub-curie temperature to make the flipping easy and then cools it down rapidly. MAMR technology accomplishes similar goal (i.e., reduction of coercivity) using microwaves. These technologies have and will likely continue to keep HDDs an essential part of the storage hierarchy on the high capacity end, in spite of rapid advancements in solid state drives (SSDs). In fact, variants of the HDD technology such as SMR [1] (shingled magnetic recording) or IMR [2] (interlaced magnetic recording) further increase the track density but at the cost much more difficult random data updates. The idea behind SMR technology is to record successive tracks so closely that while they can be read individually, writing a single track disturbs the nearby tracks. The recording is done in a series of "zones" with either gaps between them (in SMR) or normal spacing zones (in IMR). Thus it is possible to write an entire SMR zone sequentially without losing any data but random writes will not work. Consequently, these technologies are initially targeted towards backups and archiving, but can also find other uses where the IO is largely sequential.

B. Flash Storage Technology

The increasingly popular storage media is the solid-state drive (SSD) based on the flash technology. It is rapidly augmenting or replacing HDDs [3]–[5] in "higher" tiers (i.e., defined as those logically closer to CPU) of the storage hierarchy. The flash technology is based on the "floating gate transistor", i.e., a MOS transistor with an extra gate encapsulated entirely inside the SiO_2 layers but not directly connected to anything. By applying a high positive voltage to the normal gate which sits above the floating gate, electrons can be forced into the floating gate. If the gate voltage is now withdrawn, the electrons get trapped inside floating gate and this forms the nonvolatile memory. The only way to force the electrons out is by applying a large negative voltage to the regular gate, a process known as "erasure". Unfortunately, the erasure is very energy intensive and is designed to be done one entire "block" at a time. The "block" here refers to flash block and could be rather large – at least 256KB and continuing to go up due to technological considerations and could already be as high as 8MB. A block is divided into "pages" that are 2KB or larger. While the reads and writes are in the units of pages, because of the erasure process, it is not possible to overwrite a page once written. Instead, any intended overwrite must be recorded elsewhere along with suitable metadata to ensure that the latest copy of the page is always read. A written page becomes "clean" (or writeable) only after the entire block containing the page is erased. The erasures are usually done according to some delayed policy (e.g., periodically or when number of clean blocks dips below a threshold) by a process known as *Garbage Collection* (GC). The extent of GC is related to how much extra space is available to manage out-of-place writes (often known as *Space Amplification* or *Overprovisioning*) and how many page writes does the SSD perform internally for single page write issued by the OS (often known as *Write Amplification*). Given the complexity of these management tasks, SSDs have inbuit computer which runs a layer called "File Translation Layer" (FTL) to make the SSD look like a normal HDD.

The trapped electron based operation leads to several other aspects that need to be managed. First, the electrons do tend to leak out eventually, thereby leading to limited "Retention". That is, the number of times a written page can be read continuously is not unlimited. Similarly, an SSD left alone for a very long time may not retain the information, and thus not suitable for long term storage (10's of years or more). Also, when reading or writing a "cell" (or a floating gate transistor), nearby cells may be affected – known as "read disturb" and "write disturb" problems respectively. However, the most important limitation is the "endurance", or the number of times a block can erased (and then reprogrammed). This is commonly specified in terms of PE (program-erase) cycles that a cell can endure before it can no longer be programmed reliably.

The standard solution to deal with certain cells being heavily written and thereby wearing out quickly is "wear leveling". Internally, the FTL implements a virtual to physical address translation at the block level, so that it can transparently ensure that all blocks are programmed roughly at the same rate. Individual blocks could have significant variations in their lifetimes, and increasingly the wear leveling schemes take this variability into account.

The flash technology started with binary cells, i.e., cells with trapped electrons representing a "1" and one without representing a "0". This is known as SLC (single level cell) recording. It is the simplest, fastest, and can provide about 100K PE cycles of endurance per block, which is adequate in most cases. Later versions of the technology have tried to stuff multiple bits per cell by varying the number of trapped electrons to represent different levels of programming. As an illustration, we could have 4 levels as follows: 0: no electrons trapped, 1: 10 electrons trapped, 2: 20 electrons trapped, and 3: 30 electrons trapped. This then represents 2 bits/cell, incorrectly known as MLC (multi-level cell). MLC can provide about 20K PE cycles. More recently, the fastest growing technologies are TLC (3 bits/cell) with endurance in the range of a few thousand PE cycles, and QLC (quatro level cell or 4 bits/cell) with endurance measured in less than a thousand PE cycles only. The last two are clearly deficient in terms of endurance; however, with increasing size of the drives and a good wear leveling, the deficiency can be made less problematic. For example, a 1TB TLC drive with endurance of 2000 PE cycles, can, in theory, take 1PB of writes before it is exhausted. Given a rated 5 year life-time of the SSD, this amounts to 400 TB/year or 1.08 GB/day. This may be workable for data that's updated only infrequently; however, there is no doubt that the problem of endurance becomes more serious with more bits/cell. Unfortunately, all of the other problems (including retention, read disturb, write disturb, unreliable reads, unreliable writes, etc.) become more pronounced with shrinking feature size of the flash technology (which reduces the overall number of trapped electrons) and with increasing bits/cell.

C. Other Emerging Storage Technologies

Beyond the Flash technology, numerous nonvolatile memory technologies have been in development, including Phase Change Memory (PCM), Spin-torque transfer RAM (STTRAM), Toggle MRAM (TMRAM), etc. Generally, these technologies are byte addressable (unlike SSD or HDD), are much faster than flash, are re-writable, and have high endurance and retention [6]. Their current downside is the high cost and various technology maturity issues.

The technologies operate using very different principles. For example, the PCM technology works much like writeable CD technology; for writing it quickly melts a phase-change material cell and let's it cool down. A slow cool results in crystalline structure (say a "1") and a fast cool results an amorphous structure (say a "0"). (This actually means that writing a 1 takes longer than writing a 0, which has implications for suitable low-level coding mechanisms). Obviously, this makes writing much slower than reading, but the read speed is only 2-4X slower than DRAM.

Spin-transfer Torque MRAM, or STT-RAM, records data by manipulating the spin of the electrons. Electrons inherently have a small angular momentum (called spin). Electric current is generally unpolarized (consisting of 50% spin-up and 50% spin-down electrons); but by passing the current through a thick magnetic layer (called the "fixed layer"), one can produce a spin-polarized current. If this spin-polarized current is directed into a second, thinner magnetic layer (the "free layer"), the angular momentum can be transferred to this layer, and even flipped. STT-RAM has the potential to simultaneously provide nonvolatility, DRAM-like memory latencies, and extremely high retention and endurance. The technology is still very new but Everspin Electronics has recently released 1Gb chip that operates like a DDR3 memory.

Toggle MRAM is a related technology that uses a magnetoresistive toggle cell. It uses a transistor coupled with a magnetic tunnel junction (MTJ) memory cell consisting of a fixed magnetic layer, a thin dielectric tunnel barrier and a free magnetic layer. When a bias is applied to the MTJ, electrons that are spin polarized by the magnetic layers tunnel through the dielectric barrier and change the resistance of the MTJ device. The MTJ device has a low resistance when the magnetic moment of the free layer is parallel to the fixed layer and a high resistance when the free layer moment is oriented anti-parallel to the fixed layer moment. During a read, the pass transistor is activated and data is read by comparing the resistance of the cell to a reference device. During writes, the magnetic field from two Write Lines writes the cell at the intersection of the two lines but does not disturb other cells on either line.

One technology that is already available commercially and gaining rapid popularity is the Intel "Optane" [7], [8]. It is organized as a 3D cross-point structure and believed to be based on the PCM technology, although Intel has not revealed the underlying technology. It is 10X faster and has 30X higher endurance than Flash (QLC SSD), but 20X more expensive; therefore, a suitable use for it in the current storage hierarchy is in form a nonvolatile cache for enhancing both the overall storage performance and endurance (by judiciously deciding whether to accumulate writes in Optane or directly in the SSD).

With DRAM-like latencies and byte-addressability, the emerging technologies are challenging the traditional distinction between memory and storage. The CPU accesses memory in small chunks (i.e., "cachelines", typically 64B in size) and waits until the access is complete. In contrast, the storage access usually is in large chunks (e.g., 4KB or larger) and asynchronous, i.e., the CPU issues the storage access request, and then does something else until informed of the IO completion. Technologies such as STTRAM can also be used as "persistent memory" when used like a memory. In fact, Intel has already released Optane based persistent memory modules that plug into regular DDR3 memory slots. The developments lead to the notion of so called "Storage Class Memory" (SCM), which provides unified view of memory and storage, and the data using the volatile or nonvolatile storage or different types of nonvolatile storage depending on where it is best to keep the data. Also, whether to wait for a transaction inline or not can decided dynamically based on the latencies, as proposed in [9]. Persistent memory does bring additional challenges precisely due to its data retention capability, since the data in the persistent memory at the point of machine crash may not be consistent or in a known state and thus not usable following a crash. One instead needs a transactional approach to persistent memory use and suitable journaling mechanism to enable restart from a consistent state following the crash.

There are also other storage technologies using new materials or paradigms. For example, Anderson et al. [10] discusses using glass for storage, Organick et al. [11] discusses DNA storage for random data, and Caferty et al. [12] discusses storage using small organic molecules. DNA based archival storage is discussed in [13].

II. Characteristics of Modern Storage Systems

The most crucial requirement for efficient access to "big data" is the storage system that can manage large volumes of data and distribute them across devices and storage servers for low latency access without stressing individual devices, interfaces or the network. In the following we discuss some of these aspects and point out their role providing low data access latency and high throughput.

A. Storage Interfacing

Traditional HDD technologies are limited by the device capabilities, and thus the speed of the hardware/software interface between the device (or "target") and the host is not that important. However, with the emergence of SSD and other modern storage technologies, both the latency and throughput capability of the interface has become crucial. This has led to new developments interfacing technologies as well. In particular, the traditional SATA (Serial ATA) and SAS (Serial Attached SCSI) technologies are rapidly being replaced by the NVMe interface operating directly over the PCI-Express (PCI-E) bus and using its lightweight "queue-pair" mechanisms. NVMe provides not only a low-overhead/low-latency transfer mechanism, but also a number of other features including: (a) Quality of service (QoS) in data transfer with multiple queues along with priority and weighted round-robin disciplines, (b) NVMe sets, that can effectively carve out multiple virtual devices on a single real device, (c) Endurance classes, which can guide out-of-place writes and garbage collection on SSDs. Data transfer to/from a fast NVMe SSD can be done in about 100μs and to/from Optane in under 50μs. The peak data rates supported can be tremendous; with several current SSDs capable of doing sequential reads in excess of 25 Gb/sec. An Optane drive can go even faster.

B. Storage Networking

Since the data storage needs have always outpaced the storage that can be deployed on one server, much of the storage access in a large system is typically over the local network. This is true regardless of whether the storage is concentrated in a central place or distributed over many servers. Traditionally, with HDD based storage, the network bandwidth required or the network latency incurred in the remote storage access was negligible; therefore, storage was largely concentrated into a set of "storage towers", and accessed over either the regular network (e.g., Ethernet) or specialized network such as Fiber channel. Distributed storage using the object model is becoming quite popular and allows for objects to reside anywhere. However, with SSDs and newer technologies capable of driving tremendous bandwidths and able to provide data with very low latency, both the network bandwidth and latency become important. Thus, specialized protocols such as NVMe over Fabric (NVMe-OF) are emerging to seamlessly support remote NVMe storage [14]; such protocols allow NVMe based access to defined for devices located remotely across the network. However, network issues such as bandwidth allocation, congestion notification/management, and network level QoS become crucial for supporting efficient storage access. In particular, storage access bursts from even a few current SSDs can exceed the bandwidth of a 100 Gb/sec Ethernet link. Thus mechanisms to monitor congestion and take corrective actions become essential. For example, some of our recent work has shown how dynamic data migration, copy creation, and copy inactivation can be used to deal with network congestion [15].

C. Storage Virtualization

The need to migrate data across multiple devices or storage servers requires a dissociation between an application's addressing of data from its physical location. For example, the data that is currently hot may be brought in from a lower tier to the highest tier for more efficient access without affecting application's ability to reference it. This requires the application to refer to the data in a device and location independent way, such as a unique data chunk id, which is mapped to a physical address (e.g., a combination of device, volume, and physical chunk number) transparently by the "storage virtualization layer" which is ubiquitous in current large storage systems. Since logical to physical address translation is required for every access, it is usually managed via an appliance that can provide very low lookup latency and high lookup throughput. However, the downside of the appliance is the funneling of all requests through a single node which may be unscalable for large systems. Other techniques are also used, such as built-in translation capability in the local switch that the host connects to, or the host software itself. These all have their pros and cons. For example, switch based translation is the fastest and least likely to face congestion, but such a solution is expensive. Host based virtualization can be slow and inelegant since the host would need to track data mapped to all storage servers.

Note that the latency of translation and the extra network traffic caused by it become important considerations primarily because of the emerging high throughput and low latency storage technologies; this would not at all an issue if the storage system consisted entirely of HDDs. Large storage systems could even use more than one level of translation for scalability or flexibility purposes. For example, a single appliance may be inadequate in a large storage system, and multiple such appliances may be used. In such a case, two lookups may be used: the first one decides the storage partition ID (and hence the appliance to use) while the second one decides the precise location using that appliance. While such an approach relieves congestion, it adds to the latency. It also requires careful partitioning so that no appliance is overloaded. Data driven techniques could again be useful in this optimization.

The appliance load can also be reduced by caching the translation mapping in the hosts. However, this too becomes quite complex in a large system. In particular, when a data chunk is migrated, we need to invalidate all of the caches that hold relevant translation entry. If not done carefully, the overhead of tracking this information and the network traffic caused by it could become overwhelming to the extent of making the caching undesirable. Even without caching, the data migration inherently involves some synchronization as discussed in the context of tiering in section II-D.

D. Storage Tiering

Given the extremely wide range of read/write speeds offered by different technologies, a natural way to use them is in the form of a storage hierarchy as shown in Figure 1. Although the figure shows only 3 layers, others are possible and often used in real systems. For example, SMR/IMR or slow HDD drives may form the archival storage layer ("lowest" tier), whereas the regular HDDs may form the "near-line" and "primary" storage sublayers. Similarly, a large sublayer of QLC SSDs may back a much smaller layer of fast enterprise MLC or SLC drives ("highest" tier). The DRAM can be thought of as the highest layer of "storage" in the sense that DRAM buffer cache usually stores the most active part of the data. However, since this layer is volatile, it needs to be handled differently.

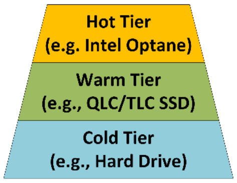

Fig. 1: Typical storage hierarchy

The key issue in designing a tiered system is to deduce from the access history the required data migration across the tiers. The tiering problem is even more complex than caching since any data moved to a higher tier is deleted from the lower tier, which means that upon eviction from the higher tier the data must be written back to the lower tier. Since the lower tier devices likely offer smaller IO bandwidth, the writeback could interfere with the reads from the lower tier. There are several other important aspects to tiering as well including placement, endurance and access disruption. When data is migrated to a different level, we need to find a suitable place for it on

Traditional caching has a simple semantics – data update in the highest level cache, lookup from highest layer to lower layers, and writeback to next lower level – which makes it easy

the device to minimize fragmentation and slowdowns due to random access to the data. Also, the greater write activity for tiering (as compared to caching) has endurance implications for SSDs that need to be considered carefully.

to ensure consistency.[1] With tiering, ensuring the consistency between the chunk location and the map maintained by the virtualization mechanism can become a bit tricky. There are several issues to take care of here: (a) there may be several requests that have obtained translation (to go to the old place) when the movement is desired; all these requests must be allowed to complete first, (b) while we are waiting for old requests to complete, it is desirable not to allow any new translations to go through, but this causes delays to these requests, (c) copy the chunk over after all the pending updates are handled and then update the map, and (d) eventually delete the old copy. For some applications, this may cause serious disruptions, and even timeouts. These issues are not that important for moving colder data to the lower tier, but for moving hotter data to higher tier, this could be problematic. For this reason, moving of hot data may need to wait until the current high activity on the chunk has subsided.

References

[1] T. Feldman *et al.*, "Shingled magnetic recording: Areal density increase requires new data management," 2013, p. 22–30, USENIX; login: Magazine.

[2] E. Hwang *et al.*, "Interlaced magnetic recording," 2017, p. 1–7, IEEE Transactions on Magnetics.

[3] L. Norman, "Ramsan-20 pcie flash ssd: Expanding the role of flash in the enterprise," 2009, texas Memory Systems White Paper.

[4] D. Cobb *et al.*, "Nvm express and the pci express ssd revolution," 2012, intel Developer Forum.

[5] Y. Cai *et al.*, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1666–1704, 2017.

[6] S. Mittal *et al.*, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2016.

[7] I. Newsroom, "Intel and micron produce breakthrough memory technology," 2015.

[8] Intel, "Intel optane technology," 2018, retrieved November, 02, 2018. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html

[9] T. H. J. Gim, Y. Won *et al.*, "Smartcon: Smart context switching for fast block devices," *ACM transactions on Storage, Vol 11, Issue 2*, March 2015.

[10] P. Anderson *et al.*, "Glass: A new media for a new era?" 2018, USENIX HotStorage.

[11] L. Organick *et al.*, "Random access in large-scale dna data storage," 2018, v. 36, n 3., p. 242, Nature biotechnology.

[12] B. J. Cafferty *et al.*, "Storage of information using small organic molecules," 2019, aCS central science.

[13] J. Bornholt *et al.*, "A dna-based archival storage system," 2016.

[14] D. Minturn *et al.*, "Under the hood with nvme over fabrics," in *Ethernet Storage Forum. SNIA*, 2015.

[15] M. . Ray *et al.*, "Adaptive data center network traffic management for distributed high speed storage," *IEEE LCN*, 2019.

[1] Write-through also works, but perhaps not useful in storage context.