Towards Physical Design Management in Storage Systems

Kathryn Dahlgren[1] Jeff LeFevre[1] Ashay Shirwadkar[2] Ken Iizawa[3]

Aldrin Montana[1] Peter Alvaro[1] Carlos Maltzahn[1]

[1]*University of California, Santa Cruz*
{kmdahlgr, jlefevre, akmontan, palvaro, carlosm}@ucsc.edu

[2]*University of California, Riverside*
ashay.shirwadkar@email.ucr.edu

[3]*Fujitsu Laboratories Ltd.*
iizawa.ken@fujitsu.com

***Abstract*—In the post-Moore era, systems and devices with new architectures will arrive at a rapid rate with significant impacts on the software stack. Applications will not be able to fully benefit from new architectures unless they can delegate adapting to new devices in lower layers of the stack. In this paper we introduce physical design management which deals with the problem of identifying and executing transformations on physical designs of stored data, i.e. how data is mapped to storage abstractions like files, objects, or blocks, in order to improve performance. Physical design is traditionally placed with applications, access libraries, and databases, using hard-wired assumptions about underlying storage systems. Yet, storage systems increasingly not only contain multiple kinds of storage devices with vastly different performance profiles but also move data among those storage devices, thereby changing the benefit of a particular physical design. We advocate placing physical design management in storage, identify interesting research challenges, provide a brief description of a prototype implementation in Ceph, and discuss the results of initial experiments at scale that are replicable using Cloudlab. These experiments show performance and resource utilization trade-offs associated with choosing different physical designs and choosing to transform between physical designs.**

***Index Terms*—storage systems, data processing, data management, physical design**

I. Introduction

During the past 25 years most speed-up was due to exponential increases in clock speeds and transistor densities while power increase remained small. Little was due to architectural innovations which meant that little had to change in the software stack. Performance improvement was implemented almost entirely below the device level. Due to the demise of Dennard Scaling and the imminent end of Moore's Law, many predict an era where performance improvement will be *mainly* driven by architectural innovations [12]. The first harbingers of this era are GPUs, TPUs, flash devices, and 3D XPoint technology, all of which require significant changes in the software stack to fully utilize their potential. In the near future new devices are expected to emerge at a higher frequency than today. Software stacks that require applications to be significantly changed in order to benefit from architectural innovations will not be affordable. For stacks to become feasible, they will have to move device-specific functionality as close to devices as possible in order to isolate applications from architectural changes.

In this paper we examine one such case of device-specific functionality: *physical design*, a term used by the data management community to refer to a specification of how data is mapped to storage abstractions such as files, blocks, or objects. Currently, physical design resides in the application or middleware layer where designers encoded assumptions about storage devices, e.g. the assumption that large sequential access is better than small random access. These assumptions worked well during the long period where storage devices were dominated by spinning media and physical design was mostly dependent on workload patterns. With new storage devices, these assumptions might or might not be adequate depending on the underlying storage system. Furthermore, storage hierarchies are deepening and provide multiple tiers of storage, including spinning media, flash media, and non-volatile memory. These tiers have vastly different performance profiles. Thus, the benefit of a physical design might depend on which tier the data is stored at any one time. The heterogeneity of storage devices and the movement of data between storage tiers along with diverse workloads require physical designs to be *managed*: when workloads change, when data is moved to new kinds of devices, whether due to tiering or storage system upgrades, performance might significantly improve by transforming the physical design of the data.

The heterogeneity of devices and the rapid arrival of new kinds of devices suggests that physical designs should be managed in storage systems, since that would isolate applications and middleware from the impact of storage architecture changes. We identified two key enabling technologies: (1) emerging computational storage technologies are making it possible to carry out some data processing in the storage layer [17]; and (2) embedding fast serialization libraries such as FlatBuffers [23] and Apache Arrow [7] in a computational storage layer and using them to store structured data will allow physical design transformations that are transparent to the application.

Note that physical design management leverages computational storage but is not subsumed by it: while physical design management is executed in the computational storage layer, it is *orchestrated* by storage clients (see Figure 2). This arrangement relieves storage servers from having to keep track of related data stored on other storage servers, greatly increasing the ability of the overall storage system to scale out.

Physical design transformations are potentially long-running and costly and need to be based on careful consideration of expected workloads, overall system load, and storage device performance characteristics. We expect that their orchestration creates an intriguing space for future research. Even if the source and target physical designs are given, the design space for transformations is large, including possible answers to the following two important questions: (1) to create benefits of a transformation as quickly as possible, what intermediate steps should a transformation produce and make available to production workloads, and (2) how many resources are dedicated to achieving each of these steps relative to resources available to applications (e.g., fast start can quickly cover a working set, slow start can better hide impact on production workloads).

Moving physical design management into storage requires significant changes in the layers above, especially in data access libraries and database management systems. However, as it turns out, database management systems can be easily adapted using "external/foreign table" facilities, which are plugin infrastructures that allow the offloading of access methods to storage systems. A welcome side-effect of offloading is increased scalability because distributed storage systems are generally easier to scale out than access libraries or database management systems. The HDF5 access library [2] also has foreign table support with its Virtual Object Layer (VOL) that similarly allows the offloading of access methods. The contributions of this paper are:

1) the idea of and the case for physical design management in storage systems,
2) the identification of research challenges,
3) a prototype implementation in Ceph, and
4) initial experiments at scale that are replicable using CloudLab.

We will first introduce terminology related to physical design management and then identify important research challenges in transformation identification and execution. We then will provide a brief introduction to object storage and Ceph. Then we describe how physical designs and transformations are implemented in Ceph using its extensibility features and internal object copying infrastructure. We then show performance and resource utilization trade-offs associated with choosing different physical designs and choosing to transform between physical designs before detailing related and future work.

II. Physical Design Management

Physical design is a particular mapping of a dataset to storage devices (including secondary data such as metadata, views, and indices). The *physical design management problem* is identifying and executing a transformation of a physical design of a dataset to another physical design in order to reduce the access cost for a given workload without changing the logical structure of the dataset. For example, transforming row-structured tabular data into column-structured data can reduce the processing cost by orders of magnitude [21] but does not change the data's logical structure.

We borrow terminology from the Visualization Toolkit's model [18] to define physical design management: the physical design of a dataset consists of *topology* and *geometry*, where topology is the logical structure of the data invariant under transformations of physical designs and geometry is a particular mapping of that topology to storage devices. Using this terminology, the physical design management problem is identifying a transformation of a physical design geometry in order to reduce the access cost for a given workload. To illustrate the topology and geometry of a physical design, consider a distributed storage system that stores data with different forms and levels of redundancies (e.g. Reed-Solomon ($k = 3, m = 2$)-erasure code or 3-way primary replication) on multiple storage servers which contain multiple storage devices of different tiers (e.g. flash and spinning media). In this case, the geometry of a physical design consists of points in the space of (redundancy, server, tier, device, allocation). Actual storage systems might provide convenient abstractions to hide some of that space or add different dimensions, as we will see in Section III. Observe that physical design management is not limited to modifying the set of indices for stored data and is therefore more general than physical design *tuning*, a well-known term in the data management community.

Physical design management consists of *identifying* and *executing* transformations. Physical design management identifies needed transformations by either applying user-specified transformation policies (off-line) or actively examining workload performance (on-line). Physical design management executes transformations transparently from the perspective of the application. Application workload accesses are adapted to suit the available physical designs containing relevant datasets. Depending on physical design management policies, old designs are either deleted or kept for redundancy. Furthermore, transformations can be compositions of more elementary transformations, some of which could be executed in parallel and some of which could depend on others, i.e. a single transformation can consist of a directed acyclic graph (DAG) of transformations.

a) Transformation identification: consists of finding a transformation that will not only solve a physical design management problem but do so in an efficient way. The result of transformation identification is a *transformation plan* in the form of a DAG of transformations. For a simple example, assume a row-structured table with 100 fields per row and a

workload with column-oriented access patterns. Thus, a transformation of the row-structured table to a column-structured table has very likely a beneficial impact on performance. One could copy the table 100 times but write only the *i*th field of each row to the target table. A more efficient way is to switch the order of this “copy then filter” transformation plan to a “filter then copy” plan so only the *i*th field of each row gets copied instead of the entire row. Transformation identification requires metadata to inform cost estimates for transformation plans, e.g. if the table has only one field per row, there will be no difference between “copy then filter” and “filter then copy”.
Research challenges: what performance characteristics would signal that it would be worthwhile to transform the current physical design? How can candidate transformation plans be constructed and evaluated quickly before transforming anything?

b) Transformation execution: consists of managing the resources needed to complete the transformation and coordinating workload access between the original data and the transformed data such that there is a good trade-off between transformation time and workload performance. The transformation execution is guided by a *transformation schedule* that informs the scheduler how many resources should be allocated (e.g. how many parallel streams are available) for executing a transformation plan, and when to make them available. For example, a transformation schedule could make only a few parallel streams available at the beginning to avoid stealing too many resources from the workload, but, as the transformation progresses and more of the workload benefits from the new physical design, the transformation schedule can shift more resources to the transformation. Another example would be a transformation schedule that starts out with quickly transforming a workload’s working set and then slowing down transformations in order to minimize impact on workload performance. In both examples, the metadata necessary for creating a schedule in advance might not be available. In those cases, we expect that physical design management will determine a schedule in real time based on observed transformation progress, workload behavior, as well as parameters such as transformation duration targets or minimum workload performance.
Research challenges: What is a good trade-off between transformation time and workload performance? How can candidate transformation schedules be constructed and evaluated quickly (in case there is enough information to create a schedule in advance)? What are the factors deciding whether the schedule should capture the working set as fast as possible and then slow down, or start slowly and speed up once workloads start to benefit from the new physical design?

To provide the benefits of transformations to workloads as early as possible, physical design management transparently adapts workload accesses to across the different physical designs containing relevant data even as the old design evolves into the new design. For example, a workload accessing a range of data may be serviced by physical design management by combining the results derived from still-existing row data with results derived from data that has already been transformed to a column-structured physical design. For this, physical design management needs to maintain metadata indicating whether a particular access can be satisfied by a completed part of the new physical design or whether it has to still use the old physical design. One parameter of transformation execution is the shape of increments at which a new physical design becomes available.

III. Object Storage

To investigate the design space of physical design management we chose object storage as the underlying storage abstraction. Extensibility and scalability benefits, combined with open source availability, industry-strength performance and reliability, and wide use made Ceph an ideal basis for our prototyping work. Object storage (also known as “object-based storage”) was first proposed by Mesnier et al. in 2003 [16]. Object storage was created to overcome the bottleneck of file servers that stood between clients and the full utilization of block-based storage devices connected via a fast (and expensive) storage-area network (SAN). Object storage overcomes this bottleneck by splitting the traditional file system server into (1) a file storage service with a flat name space where files are called “objects” that never span more than one server (except for redundancy), and (2) a metadata service which overlays the flat object name space with the hierarchical name space of POSIX file systems and stripes POSIX files across multiple objects. This split allows clients to directly interact with storage servers and move the metadata service out of the bandwidth-intensive data path. While object storage was designed in the context of file systems, object storage without the file system component gained popularity in virtualized environments because flat name spaces and the simple one-to-one mapping of objects to servers made object storage easy to scale out.

The widely-used open source Ceph storage system [24], [25] (see ceph.com) introduces a reliable and self-healing distributed object service called RADOS [26]. Ceph/RADOS transparently distributes objects and their replicas over servers by hashing object names. The hash function is provided by Ceph/RADOS servers to clients so they can always calculate the location of a given object. As Ceph/RADOS clusters change due to server failures or other reasons, servers are keeping clients up-to-date so that these changes do not impact application-level access to objects. Applications can create or delete objects and can call *object methods* such as reading or writing. Objects do not have a fixed size but the ideal size of objects in Ceph/RADOS is around 8MB due to trade-offs between locality, parallelism, and locking overhead. Since typical deployments of Ceph are in the range of 10s of Petabytes, the total number of objects is in the billions. With this many objects, enumerating them can become very costly, especially in a flat name space.

The Ceph file system CephFS, the Ceph/RADOS block device `rbd`, the AWS S3-compatible interface `radosgw`, and other Ceph/RADOS applications use naming conventions that allow applications to *derive* object names instead of having to look them up. For example, data that can be accessed with a coordinate system can be partitioned into objects according to a partitioning function, using object names that represent bounding boxes within that coordinate system. The combination of the partitioning function and the object convention allows the generation of object names for any coordinates. For the rest of the paper we will focus on Ceph/RADOS as reference system and use its terminology. In principle, our contributions can be applied to any object-based storage system.

An object is created in the context of a *pool* that defines resilience properties, such as the level of replication, and placement rules for which devices to use and how to avoid placing more than one replica of an object in a single failure domain. Ceph/RADOS supports multiple pools that can vary depending on their resilience properties. Placement rules either represent different types and levels of redundancies, or different storage tiers, or a combination of these.

Object storage systems are extensible: in addition to the usual read and write methods of objects, new methods can be introduced by adding new object classes that are implemented on top of an abstract local storage interface. As Sevilla et al. show [19], Ceph/RADOS in 2016 had added to its mainline over 15 new object classes with a total of close to 150 additional methods that were introduced over the span of six years (see Figure 1). These methods have read/write access to the object's state that is locally maintained in a key/value store and a blob store within a name space private to the object (objects cannot access the state of other objects stored on the same storage server through the local key/value or blob store). To add a new method, a deployed Ceph/RADOS storage system needs to be rebooted. In many deployments, this is an unacceptable interruption of service. To avoid this interruption, the Ceph community created object classes that embed interpreters and VMs that allow extensibility without reboot. Examples are Python, Lua, and very lightweight and fast Javascript engine VMs such as V8 isolates.

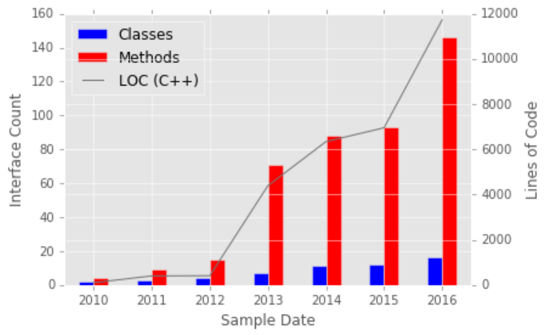


Fig. 1: The number of object classes and object methods as well as the total lines of code dedicated to new methods committed to Ceph's mainline from 2010 through 2016 [19]

The extensibility of object storage systems implements a form of computational storage and allows the introduction of highly-efficient serialization libraries like FlatBuffers or Apache Arrow and thereby enables storage systems to manage structured data and its physical design. This gives rise to the following implications: (1) Structured data management in storage systems enables offloading of work to storage via an application's external data source interface. Relational database management systems can offload relational access methods to storage via their external table interface. Similarly for the cluster computing framework Apache Spark that also offers an external data source interface that supports offloading. The widely used access library HDF5 [11] offers an external data source interface called "Virtual Object Layer" which also allows offloading. Each of these are examples of systems with interfaces that can be used to offload work to storage. (2) Since object storage systems can be scaled to a large number of servers, offloading of access methods provides a powerful scale-out mechanism. (3) Offloading of access methods removes the need to accumulate increasingly inadequate assumptions about storage systems (see the study of Trivedi et al. [22] that showed that even the assumptions in modern access libraries like Parquet can significantly diminish performance).

IV. Toward Physical Design Management with Ceph

Physical design management aims to provide a flexible means of defining and reconfiguring physical designs. Reconfigurations are done to increase the efficiency of data access for input workloads by considering the existing geometry of physical designs. When changes are driven by on-line workload analyses, changes in the workload may trigger a mechanism to compare the performance of the current workload (or sliding window over the recent workload) on a relevant subset (enumeration) of physical designs under consideration. A cost-benefit analysis is performed for executing the given workload over the current and potential designs including the cost to transform (i.e., reconfigure) to the potential new design. If a new design is selected, then a transformation plan is created and executed in a rolled-out fashion. The transformation roll-out allows the current workload to enjoy incremental benefits and future workloads with the same properties will enjoy the full benefits. Additionally, the flexibility of the physical design management infrastructure allows the ability to control the rate of transformation roll-out to decrease the impact of shifting storage resources away from other concurrent processes.

All transformations operate over serialization libraries but below the layer where applications access datasets. The strategy makes both transformation execution and results transparent to the application.

a) Serialization libraries: In this paper we use FlatBuffers, FlexBuffers, and Apache Arrow.

FlatBuffers [3] is a fast in-memory serialization library which supports arbitrary data schemas. FlatBuffers are structured as ordered contiguous sequences of bytes such that the fields of the data structure have pre-determined memory

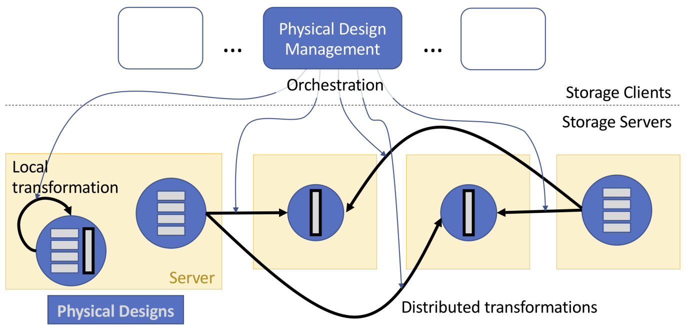


Fig. 2: Physical designs are managed via local and distributed transformations that are identified and orchestrated at the storage client layer and executed in the storage server layer.

allocations tied to the data types of fields specified in the schema. The structures are built with minimal additional storage overhead for FlatBuffer-specific metadata. FlatBuffers are similar to ProtocolBuffers, but the difference is FlatBuffers allows access of individual elements without deserializing the entire structure, which is a requirement for ProtocolBuffers.

FlexBuffers [4] is a fast serialization in-memory library which complements FlatBuffers. While a pure FlatBuffers schema requires knowing the types of data to store in advance, FlexBuffers allows storage of arbitrary types, which must be interpreted at run-time. FlexBuffers complement FlatBuffers and a FlatBuffer schema may include FlexBuffer-defined field. The latter characteristic is core to the FlatBuffer-FlexBuffer Rows format utilized in our physical design management prototype.

Apache Arrow [5] is a serialization library for flat and hierarchical data. Arrow is optimized for in-memory column-wise storage and includes a number of facilities for automatic data compression. Defining structures in Arrow has the constraint similar to FlatBuffers wherein the data types of the schema cannot be dynamically defined.

We distinguish *local* transformations within an object that are always local to a server from *distributed* transformations that usually form a DAG of data streams between source and target objects and that usually span multiple servers and therefore incur network traffic. We also distinguish between *primary* transformations that modify the physical design of the primary data from *secondary* transformations that "tune" the physical design by adding or modifying secondary data like indices while leaving the physical design of the primary data unmodified.

b) Local transformations: have the advantage of not causing any network traffic (except for redundancy updates) and the disadvantage of keeping all primary and secondary transformations within the object boundary, e.g. for tabular data row-to-column transformations the length of columns are limited by the number of rows stored in the local object, or an index only points to local data. Local *primary* transformations move part or all of the data of an object into a new physical design within the object. That means the identity of the object is preserved, the changes are persisted according to the redundancy and placement policies of the object's pool, and network traffic is only due to redundancy (i.e. no network traffic if there is no redundancy). Local *secondary* transformations create secondary data that speeds up access under certain workloads. Creating local indices or statistics are both examples of a local secondary transformation. This transformation also preserves object identity and maintains redundancy and placement of the object.

c) Distributed transformations: are a DAG of streams creating the new physical design in a new set of objects from the data stored in a set of objects with the original physical design. Distributed transformations have the advantage of parallel execution across multiple servers and can produce global transformations, e.g. for tabular data row-to-column transformations the length of columns is only limited by the total number of rows across all objects of the dataset, or an index can be defined over the entire dataset. Distributed *primary* transformations create a sequence of streams that pull filtered data from each source object in parallel for each target object. These streams are implemented with a generalized version of Ceph's `copy_from` command that in its original version allows a Ceph client to delegate to Ceph/RADOS the copying of the entire content from a source object to a target object, replacing all content at the target object. The generalized version allows calling serialization library access methods at the source object and adding the result at the target object without necessarily replacing all content that is already there, also using serialization library access methods.

V. Evaluation

In this section, we present an experimental evaluation of our Physical Design Management framework in Ceph for both data transformations and pushdown computation within the storage layer. Section V-A describes the experimental methodology. Section V-B presents a comparison of formatted dataset sizes to highlight the size trade-offs among our serialization formats. Section V-C compares query performance with and without pushing down processing (i.e., select, project, etc.) into storage for our structured row format (FlatBuffers) and our structured column format (Arrow). We then further drill down into this experiment in Section V-D to show the resources consumed by a client and server machine with and without pushdown processing to highlight the resource trade-offs for offloading processing to storage. In Section V-E we compare approaches for transforming a dataset from row to column formats via *local transformation* (no data movement), *distributed transformation* (data movement stays entirely within the storage layer), and a representative approach for *client-side* transformations that performs transformations outside the storage layer. Lastly we examine the benefits transformations that organize columns differently on disk by grouping all columns together or independently.

A. Methodology

Datasets and workloads. We consider two datasets. One is the LINEITEM table from the standard TPC-H benchmark. The LINEITEM table schema consists of 15 cols including ints, floats, chars, dates, and strings/text schema. This dataset was randomly generated using the TPCHDATAGEN program and has 750 million rows. Our second dataset is inspired by IoT/sensor datasets and consists of 100 columns of identical types (32-bit integers). This dataset was randomly generated with uniform distribution of values $1 - 10000$ and has 250 million rows.

Our workloads consist of simple selection and projection queries on each dataset with selectivities 1%, 10%, and 100%. For LINEITEM, we use the float column `extended_price`, and for 100COLS dataset, we use the first integer column. The selection query on the LINEITEM dataset is given as `SELECT * FROM lineitem WHERE extended_price` $> X$, where X results in 1,10, and 100% selectivity. The selection query on the 100COLS dataset is given as `SELECT * FROM 100cols WHERE col0` $> Y$, where Y results in 1,10, and 100% selectivity. All selections and projections are applied using the corresponding FlatBuffer or Arrow APIs with standard comparison operators on each value.

Execution environment. All of our experiments are performed on Cloudlab [10], which is a bare-metal-as-a-service environment for scientific research. We create clusters of a single machine type, "c220g5", with the following configuration: two Intel Xeon Silver 4114 10-core 2.20 GHz CPUs, 192GB ECC DDR4-2666 RAM, one 1TB 7200 RPM 6G SAS HDD, one Intel DC S3500 480 GB 6G SATA SSD, dual-port Intel X520-DA2 10Gb NIC (PCIe v3.0, 8 lanes), and onboard Intel i350 1Gb per node. All of our object storage data resides on the HDD of these machines. Each of our configuration profiles is available publicly (Appendix A). Each profile consists of a single client machine and a number of storage servers – Ceph Object Storage Devices (OSDs).

The client machine is where the client driver application (in this case our query driver, or similarly a database application) is installed and executes. This machine has no data storage. All storage-side query processing and transformations are performed on the storage servers only. All client-side query processing and transformations are performed only the client machine within the driver, representing a system where the storage does no extra work for either physical design or processing.

Systems. We use Ceph (Luminous version) with our Skyhook Data Management [15] extensions that can perform transformations and basic pushdown processing over structured data using the FlatBuffers and Arrow serialization libraries. All of our extensions are implemented as RADOS library object classes.

B. Dataset sizes

Table I shows the dataset sizes for each schema and data format that we evaluate. Datasets consist of 10,000 objects based on 10MB raw data in each object, resulting in 100GB of raw data in each dataset. When each object is converted to FlatBuffer or Arrow format, the final sizes change as indicated in Table I.

Schema	Format	Size in GB	Number of rows
LINEITEM	flatbuffer	210	750 million
LINEITEM	arrow	103	750 million
LINEITEM	raw	100	750 million
100COLS	flatbuffer	85	250 million
100COLS	arrow	188	250 million
100COLS	raw	100	250 million

TABLE I: Dataset sizes for our two datasets in each format. Each dataset consists of 10,000 uniformly-sized objects.

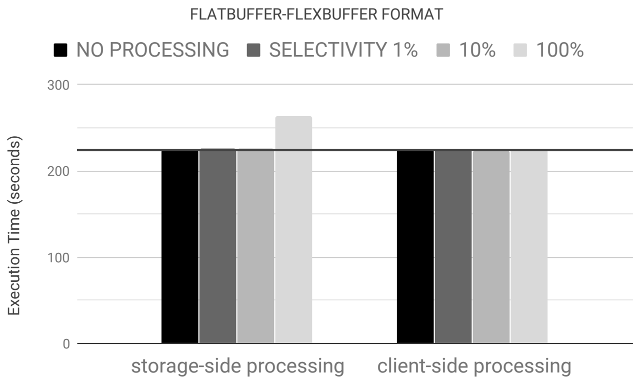
Table I shows that our FlatBuffer format increases the LINEITEM data size by $2.1\times$ whereas Arrow format is opnly slightly larger than raw size. Interestingly, almost the inverse is true for the 100COLS dataset where FlatBuffer format is nearer to raw size (in fact slightly smaller). One possible reason is due to the way FlexBuffers (each row in our FlatBuffer format is actually stored as a FlexBuffer) store integers, using only the minimum number of bits for the given value. Because our integers in 100COLS dataset range from $1 - 10000$ the FlexBuffer may be using 14 only bits per integer instead of 32 as would be done with Arrow. This is one benefit of FlexBuffers, although the drawback is that updating in place is not possible unless the new value fits squarely within the previous space.

C. Impact of transformations on query performance

In this section we evaluate the performance of our physical design management transformations in order to show the benefits and tradeoffs our approach provides for query processing, such as for computational storage. We also evaluate resources consumed for pushdown query processing (performed entirely within the storage servers) versus all processing done by the client driver application. When all processing is performed by the client application, the storage system functions in a traditional way simply returning the entire object as a standard read using LIBRADOS (no custom object class processing is applied).

Figure 3 shows query performance with 8 storage servers for the LINEITEM dataset at various selectivities. The NO PROCESSING value indicates the standard LIBRADOS read where data is returned directly to the client. This represents traditional storage system usage, so we have added a horizontal line as a baseline. The top figure represents FlatBuffer format and bottom figure Arrow format.

For each format, the various selectivities have nearly the same performance as the baseline, indicating that the overhead of the corresponding APIs to apply filters is minimal. As we show next in Section V-D, this is likely due to the disk speed (HDD) or network being the bottleneck since the processing overhead in terms of CPU is quite small in this case. Comparing FlatBuffer format with Arrow format shows that Arrow is faster to process by a large margin, due in part to the LINEITEM dataset size for Arrow as shown in Table I. In every case, the 1%, 10%, 100% selectivity, storage-side



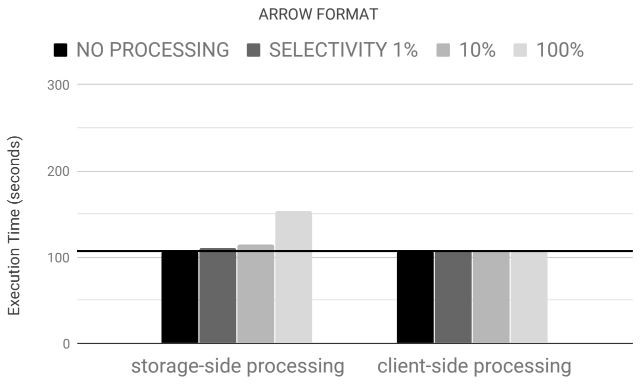


Fig. 3: Execution time with 8 Storage Servers (OSDs) row format (top) and column format (bottom).

(pushdown) processing performs slightly worse than client-side processing due to the need to reassemble the matching results into the corresponding format (FlatBuffer or Arrow) to be returned to the client. The 100% selectivity is the worst since the storage must apply the predicate filter to every row in the dataset, and then since all rows pass, it must reassemble the *exact same* data to be returned to the client. Another option to reduce the extra cost for queries that select most of the data in a given partition is to simply return the original data structure as-is but mark the rows that do not match as “deleted” or similar. This overhead shows the need for a cost model based on statistics that can decide whether or not to pushdown the selection processing into the storage layer, an area of future work for us.

D. Machine resources consumed during query processing

Figures 4 and 5 show the CPU and network resources consumed by a client and server machine during the first 60 seconds of query processing at the 1% selectivity shown in Figure 3 using the Arrow format. Each figure reports the resources used on the client and one of the 8 storage servers, without pushdown processing (top row) and with pushdown processing (bottom row). The x-axis of both figures denotes 1 second intervals for the first 60 seconds of the query. The y-axis in Figure 4 indicates CPU as percent usage, while in Figure 5 it indicates network send and receive in MBps, both as reported by the dstat utility.

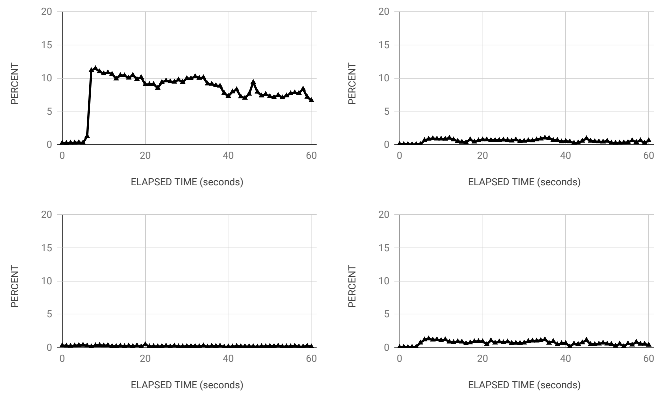


Fig. 4: (a) client machine (b) one storage server CPU usage during first 60 seconds of 1% selectivity query *without* pushdown processing (top), and *with* pushdown processing (bottom).

Figure 4 (top row) shows the CPU consumed by the client machine (a) without pushdown processing is much higher than the storage server (b). This is because the client is both receiving all of the data from the storage servers as well as applying the processing. With pushdown processing, Figure 4 (bottom row) shows a dramatic reduction in client CPU usage (a) , with very small corresponding increase in storage server CPU usage (b). This in part due to fewer data packets being sent to the client since only 1% of the data passes the filter, as well as distributing the processing from the client to all 8 of the storage servers. Figure 5 confirms that without pushdown processing (top row), the client (a) is receiving much more data from the storage server (b) than with pushdown processing (bottom row). This result indicates that pushdown processing using our structured formats and their corresponding APIs has very little overhead on the storage servers in this example, for a simple selection filter applied to every row of a given column.

E. Transformations

Here we compare several approaches to transform data within our physical design management framework. Because row-oriented storage is write-optimal while column-oriented storage is known to be beneficial for certain workloads such as analysis queries, here we highlight the capability of our system to perform these transformations dynamically from row to column orientation within storage. We compare local transformations (no network traffic), distributed transformations (within storage traffic only), and client-side transformations (network traffic back and forth between storage and client). In each approach, we only transform the first 1,000 out of 10,000 objects representing a possible schedule of a transformation plan that acts on subsets of the objects in turn. This is because non-local transforms are required to read-many-write-one object, for *each column*, which injects a very high amount of work and traffic into the storage system and thus indicates

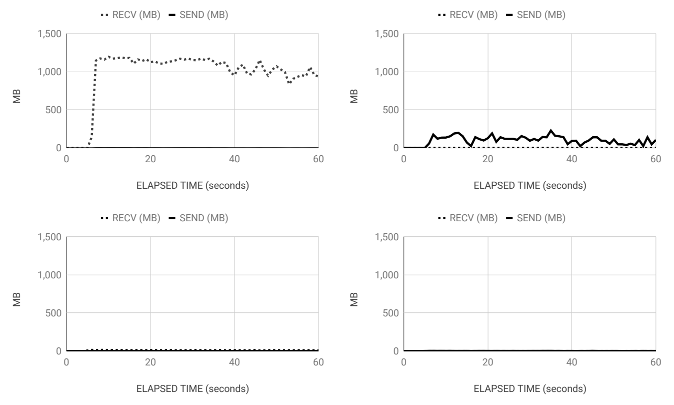


Fig. 5: (a) client machine (b) one storage server Network usage during first 60 seconds of 1% selectivity query *without* pushdown processing (top), and *with* pushdown processing (bottom).

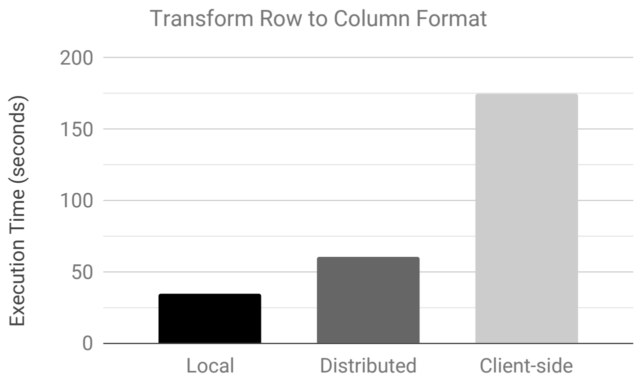


Fig. 6: Time to transform from FlatBuffer-FlexBuff (row) to Arrow (col) on an 8 OSD cluster using the 100COLS dataset.

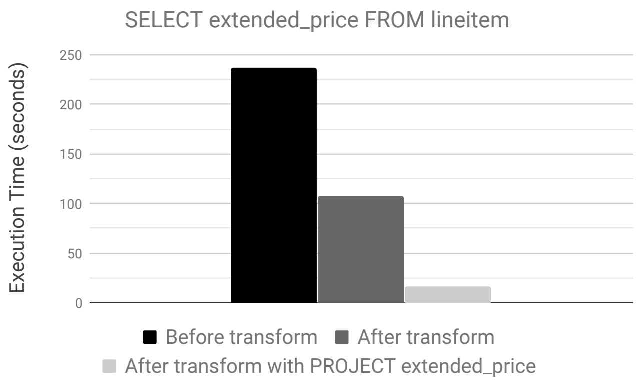


Fig. 7: SELECT single column before and after transform all cols, and after transform with PROJECT only a single column, on an 8 OSD cluster using the LINEITEM dataset.

the need for a transformation plan schedule that can be rolled out over time as we mention in the "research challenges" of Section II. We report results for the 8 OSDs cluster, the same trends were observed with the 4 OSDs cluster and so are omitted for brevity.

Figure 6 reports the time to transform from row to column with three approaches. The first result "Local" represents a local transformation (Section IVb) on each object from row to column format, and no data travels over the network. At the end of this transformation phase, all transformed objects contain column formatted data.

The second result "Distributed" is a distributed transformation (Section IVc) which is done in two phases. First a local transformation is performed, and then one column (`col0`) is collected into a single target object. We use our modified Ceph `copy_from` command where a single target object collects data from multiple source objects (e.g., `col0` from objects 1-m) instead of a single source object as is otherwise done when creating replica objects. Here there is network traffic between objects within the storage layer only, however copying data to replicas is a typical function of distributed storage system for replication tasks. As future work, these two phases could also be combined into a single transform *inline* during `copy_from`. At the end of this transformation phase, one object (target) contains all the data from a single column.

The third result "Client side" represents a more traditional way to transform data, where the transformation work is performed outside of the storage system. In this case, all of the data is read by the client, the transformation is performed, and one column of data (`col0`) is written back to a single target object. The increased network traffic of this approach has a large impact on execution time.

Lastly we present results when columns are organized differently within each object, for the local transform case. The physical design transformation plan may store all columns together within the same data structure on disk, or subsets of columns independently or grouped together (i.e., co-located) on disk, for instance when certain columns are accessed more frequently by the workload. Here we again use the 8 OSDs cluster size as in our previous row and column comparisons.

Figure 7 reports the execution for a query that selects only one column from the original row format (FlatBuffer-FlexBuffer), from the transformed to column format (Arrow) with all columns grouped together, and from the transformed to column format with transform PROJECTing each column into its own data structure (Arrow) on disk. The time to SELECT the `extended_price` column is highest before transform from row format at 237 seconds, then after transform to column format the time is reduced to 107 seconds, similar to our previous row versus column results in Figure 3. After transforming the `extended_price` column into its own data structure on disk, the execution time to select that single column drops to just 16 seconds. This experiment highlights the potential benefit that physical design management in storage can provide for query processing such as with computational storage techniques.

VI. Related Work

As noted in Section II, physical design management draws similarities with physical design tuning. There has been a long line of research in both offline [28] and online [8] physical design tuning for databases, including reorganizing data differently across two systems [14] for workload co-processing. However, physical design management subsumes traditional database physical design tuning by introducing an expanded space of design elements as well as enabling the storage layer to directly implement physical designs. Transforming between row and column orientations and the idea of supporting both row and column access are not new. For example, Teradata [6] and Greenplum [1], can shift data between column and row storage and allow both row and column access. While our system can shift and access data between row and column formats, we focus on more general physical design capabilities, performed directly within the storage layer rather than a custom file system or database application.

Efforts toward hardware-embedded database functionality [9], [27] fall within the computational storage paradigm and are similar to ours in terms of pushing down higher-level processes below the application layer, but address query processing tasks rather than physical design. In our work, the physical design can be informed by query cost models and the storage can be reconfigured dynamically to adjust to changing workloads, with a goal of improving performance of pushdown computations. Semantically-aware storage [20] offers an approach to software-based storage processing that focuses on using database application semantics to govern mapping out bytes in storage to optimize data access and availability. In contrast, physical design management operates at a higher level focusing on selecting beneficial structured data formats, auxiliary data structures, local organization of formatted data within an object, and global dataset partitioning across potentially many objects and devices.

Very recent work [13] has proposed a continuum of learned data structure designs for key-value stores including automatically synthesized data structures, highlighting a large design space that considers data access and layout options, among other choices. Their work is more specific to data structure choices based on low-level primitives such as block size and memory budgets for candidate structures whereas we focus on higher level physical design choices. However, provided a good cost model for candidate data structures, such methods could possibly inform physical design reconfigurations within storage in our work as well.

VII. Conclusion and Future Work

We have presented our initial work toward the concept of physical design management directly within the storage system, along with several research challenges. Our approach considers the typical elements of database physical design, but expands these to include additional elements relevant to the storage system. We described how physical design transformations may be done in a local or distributed fashion, synchronously or asynchronously, or via a scheduled roll-out of a computed transformation plan, though we leave such a plan for future work. Although our work includes aspects of computational storage, one explicit goal of physical design management in storage is to enhance both the performance and type of computations that can be performed within storage by reorganizing the physical data configuration of a dataset across the entire storage system.

A key aspect of our design includes utilizing fast in-memory serialization libraries such as FlatBuffers and Arrow to store formatted data structures within objects that retain data semantics and are aware of their current format. Thus objects can use the corresponding APIs available for processing or reorganizing themselves as directed by a physical design management orchestration layer. Crucially this object-local knowledge helps to avoid updating the current state (e.g., format, organization on disk, etc.) of an object to the application layer each time a physical design change occurs. This also enables interesting scheduling options for rolling out transformation plans.

Using an object storage system, we evaluated physical design changes between row and column oriented data formats using fast in-memory serialization libraries. We also showed the benefits of different formats for query processing as well as the tradeoffs among resources consumed by the client application machine and storage servers during processing and transformations. Our evaluation showed the CPU resources used by the library APIs during query processing in storage was small, and the network traffic was greatly reduced for highly selective queries. We also showed the flexibility of our approach for the given test queries performing select and project over different datasets, different data formats, and various selectivities.

An area of future work is to examine the resource impact of design transformations and processing on storage servers with many concurrent workloads. Additional future work includes developing cost models to help explore how different storage media such as HDDs, SSDs and NVMe can affect the tradeoffs among physical designs and the resulting impacts on query processing.

Acknowledgements. This work was supported in part by NSF grants OAC-1836650, CNS-1764102, and CNS-1705021, and by the Center for Research in Open Source Software (cross.ucsc.edu).

References

[1] *Pivotal Greenplum (6.0)*. Greenplum documentation. Accessed: Aug. 17, 2019. [Online]. Available: https://gpdb.docs.pivotal.io/6-0/admin_guide/ddl/ddl-storage.html.

[2] Hierarchical Data Format. Web Page. www.hdfgroup.org/HDF5/.

[3] Google FlatBuffers documentation. https://google.github.io/flatbuffers/, Sept 2015. Accessed: 2019-08-01.

[4] Google FlexBuffers documentation. https://google.github.io/flatbuffers/flexbuffers.html, Sept 2015. Accessed: 2019-09-08.

[5] Apache Arrow documentation. https://arrow.apache.org/, Feb 2016. Accessed: 2019-08-01.

[6] D. Abadi. Hybrid row/column-stores: A general and flexible approach. [Online] https://downloads.teradata.com/database/articles/hybrid-rowcolumn-stores-a-general-and-flexible-approach, Mar 2015. Accessed: 2019-09-08.

[7] Apache Software Foundation. The Apache® Software Foundation announces Apache Arrow™ as a top-level project. February 17 2016.

[8] N. Bruno and S. Chaudhuri. An online approach to physical design tuning. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 826–835. IEEE, 2007.

[9] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230, 2013.

[10] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[11] M. Folk, A. Cheng, and K. Yates. HDF5: A file format and I/O library for high performance computing applications. In *Proceedings of SC'99*, Portland, OR, November 13-19 1999.

[12] B. Hendrickson. The day after tomorrow: The looming post-exascale crisis. In *Keynote at IPDPS 2018*, Vancouver, BC, Canada, May 21-25 2018.

[13] S. Idreos, N. Dayan, W. Qin, M. Akmanalp, S. Hilgard, A. Ross, J. Lennon, V. Jain, H. Gupta, D. Li, et al. Design continuums and the path toward self-designing key-value stores that know and learn. In *CIDR*, 2019.

[14] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, and M. J. Carey. MISO: souping up big data query processing with a multistore system. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1591–1602. ACM, 2014.

[15] J. LeFevre and N. Watkins. Skyhook: Programmable storage for databases. In *Vault'19*, Boston, MA, Feb. 2019. USENIX Association.

[16] M. Mesnier, G. R. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8), Aug. 2003.

[17] D. Robinson. What's up with computational storage. Web Page. blocksandfiles.com/2019/06/14/computational-storage-market-research/, June 14 2019.

[18] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit*. Kitware, 2006.

[19] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn. Malacology: A programmable storage system. In *EuroSys '17*, Belgrade, Serbia, April 23-26 2017.

[20] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *FAST*, volume 5, page 18, 2005.

[21] M. Stonebraker, C. Bear, U. Çetintemel, M. Cherniack, T. Ge, N. Hachem, S. Harizopoulos, J. Lifter, J. Rogers, and S. Zdonik. One size fits all?—part 2: Benchmarking results. In *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, Jan. 2007.

[22] A. Trivedi, P. Stuedi, J. Pfefferle, A. Schuepbach, and B. Metzler. Albis: High-performance file format for big data systems. In *USENIX ATC '18*, 2018.

[23] W. van Oortmerssen. Flatbuffers: A memory-efficient serialization library. Web Page. android-developers.googleblog.com/2014/06/flatbuffers-memory-efficient.html, June 17 2014.

[24] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI'06*, Seattle, WA, November 2006.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *SC '06*, Tampa, FL, November 2006. ACM.

[26] S. A. Weil, A. Leung, S. A. Brandt, and C. Maltzahn. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2007 ACM Petascale Data Storage Workshop (PDSW 07)*, Reno, NV, November 2007.

[27] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos. Beyond the wall: Near-data processing for databases. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 2:1–2:10, 2015.

[28] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 Design Advisor: integrated automatic physical database design. In *Proceedings of the Thirtieth international conference on Very Large Data Bases-Volume 30*, pages 1087–1097. VLDB Endowment, 2004.

Appendix

A. Reproducibility

All experimental results included in the evaluation are scripted for reproducibility. Running the experiments requires obtaining a Cloudlab [10] account and obtaining access to a Cloudlab-supported project. See instructions for requesting an account on the Cloudlab website:

`https://cloudlab.us/`

Note reproducing experiments will require adding an SSH key from your machine to your list of Managed Keys in the Cloudlab.

Reproducing evaluation experiments breaks down into three major steps:

1) Spin up a Cloudlab cluster instance using one of the project profiles
2) Run the setup and install scripts associated with the kind of experiments you wish to run.
3) Choose and run the experiment scripts.

Detailed instructions for reproducing the evaluation experiments are located here:

`https://github.com/KDahlgren/pdsw19-reprod/wiki`