# Self-Organizing Data Containers

Samuel Madden
madden@csail.mit.edu
MIT CSAIL

Jialin Ding
jialind@mit.edu
MIT CSAIL

Tim Kraska
kraska@mit.edu
MIT CSAIL

Sivaprasad Sudhir
siva@csail.mit.edu
MIT CSAIL

David Cohen
david.e.cohen@intel.com
Intel

Timothy Mattson
timothy.g.mattson@intel.com
Intel Labs

Nesime Tatbul
tatbul@csail.mit.edu
MIT CSAIL and Intel Labs

## ABSTRACT

We propose a new self-organizing, self-optimizing, meta-data rich storage format for the cloud, called a self-organizing data container (SDC), that enables order-of-magnitude performance improvements in data-intensive applications through instance-optimization, i.e., the adaptation of data representation to exploit both the distribution of the data and the workload operating on it. Unlike existing low-level cloud storage formats like Apache Arrow and Parquet, SDCs capture both data and metadata, like access histories and distributional statistics, and are designed to be flexible enough to encompass a variety of modern high-performance representations for data analytics, including partitioning, replication, indexing, and materialization. We present a preliminary design for SDCs, some motivating experiments, and discuss new challenges they present.

## 1 INTRODUCTION

Historically, DBMSes have adopted a monolithic architecture, with the system under control of all aspects of data management, including data placement, layout, scheduling, allocation of computation and memory to queries, as well as query optimization and execution. In the cloud, data services are increasingly disaggregated, with a reliable storage layer (e.g., Amazon S3) managed by the cloud provider, file-based data structured data (e.g., Parquet files), often produced by applications outside the data management layer, and a variety of high-level interfaces to the data (e.g., SQL, machine learning applications, data visualization engines). This enables a separation of concerns, where each layer is managed and scaled independently, unlike the "shared nothing" designs of conventional DBMSes where physical nodes that store data are created for each processing node that operates on this data. The rise in popularity of data processing systems like Spark [22] is driven by their successful adoption of this new style of cloud architecture.

While the success of these new systems demonstrates the advantages of this new way of architecting data-driven applications, disaggregation leaves a great deal of performance on the table. For example, when running TPC-H, the disaggregated version of Amazon Redshift, called Redshift Spectrum is at least 8× slower than Redshift itself [20]. This inefficiency is partly due to the layered design of Spectrum, which requires it to load data from underlying S3 files, preventing it from sharing information about data structure and representation between the storage and query execution layers, and partly due to the fact that S3 is lower bandwidth and higher latency to local storage. Because important metadata is not preserved by low-level cloud storage formats like Parquet, data processing systems operating on such data often lack performance-related metadata such as histograms, and other statistics that are key to good performance, preventing them from optimizing layouts for efficient storage, especially on lower-performance cloud storage where optimizing access is even more critical.

What is needed is a way to build efficient data systems on the cloud while maintaining the advantages of disaggregation. Our key insight is that cloud data systems can be much more efficient if they have a storage layer rich enough to support modern data storage optimizations that are at the heart of high performance data analytics systems, including indexing, flexible multi-dimensional partitioning, compression, and an ability to adapt to the workloads that run on them. To this end, we propose a cloud-optimized storage format, called self-organizing data containers (SDCs). By *self-organizing* we mean that the container has the flexibility to take on many possible layouts, and that it adapts to the clients' workload as they interact with it. Unlike raw Arrow [1] or Parquet [4] files, SDCs are metadata rich and capture data distributions and access patterns, and unlike systems that implement transactional operations on cloud object stores like Delta Lake [5], Hudi [2], and Iceberg [3], SDCs can represent complex storage layouts encompassing different partitioning and replication strategies.

Specifically, by tightly coupling meta-data, including distributional statistics, indexes, and access patterns, with the data itself, SDCs naturally contain the information needed to support efficient data access. Most cloud storage systems view data objects as immutable; however, they confound immutability of the logical contents of blocks (i.e., the set of records stored in each file) with the physical layout of data (i.e., whether records are column-oriented, compressed, etc). In SDCs, different physical representations, including summaries, aggregates, different columnar and row-oriented layouts, and data layout optimizations for modern hardware (GPU/CPU) can be represented in a single data object, and those layouts can be transformed and adapted over time as the access patterns shift. Hence, SDCs physically mutate over time, "self-organizing" into the optimal layout, even if the data itself remains immutable. Our vision towards such self-organization is influenced by our work on instance-optimized systems over the past several years [8–10, 13, 15, 18], where we have shown that by building data layouts that adapt to both the data and the queries that run on them, dramatic performance gains are possible.

Once we have SDCs, many existing systems, including conventional relational databases, parallel data processing frameworks, and ML systems can be easily adapted to use our new data format. By building on our prior work on cross-layer optimization [7, 16, 19] and instance-optimized storage systems, we believe we can construct next generation cloud-processing systems on SDCs that preserve the flexibility of conventional systems while offering order of magnitude performance gains that rival the performance of monolithic systems running on bare-metal hardware.

In this paper, we describe a preliminary design for a cloud-based SDC and its performance advantages. We also highlight a number of interesting research directions that SDCs present. Specific contributions include:

- We describe how SDCs will enable cloud-based storage systems to take advantage of modern instance-optimization techniques for data storage and layout.
- We describe a preliminary implementation of SDCs, designed to be stored on immutable cloud storage. This implementation is "serverless", in that it requires only client-side libraries to mediate access, while still allowing SDCs to adapt their layout over time. The key idea is to engage clients in the incremental transformation of data as they access it, in a way that ensures clients see a consistent view of the data.
- We provide motivating experiments showing the advantage of instance-optimization.
- We describe a number of research directions for SDCs.

## 2 SDC OVERVIEW

At its core, an SDC is a physical representation of a relational data model. Clients access the SDC through a client-side library that exposes typical operations that "data frame" APIs like Pandas and Spark [22] implement: filters, projections, grouping and aggregation, and joins. SDCs are designed to be optimized for particular access paths — for example, they may be partitioned or sorted on a particular combination of attributes — which are exposed through metadata, so clients are able to easily identify and take advantage of those optimized access paths. In particular, clients do not need to configure these optimized access paths themselves; SDCs are designed to automatically create and modify these access paths over time based on the client's interactions with the SDC through the client-side library.

One key goal of SDCs is that access to the SDC should not be mediated by a cloud service: i.e., the client-side library contains all of the logic to read the SDC. Beyond simple read access, we also envision that each client library will publish both metadata about the SDC, such as (a sample of) the history of operations it posted over it, as well as make optimizations to the structure of the SDC designed to improve its performance, such as creating materialized views, replicating portions of the SDC that correspond to frequently queried subsets of the data, creating a (multi-)dimensional index and storage layout or replicating the data within the SDC file. This is done incrementally as a part of the operations clients perform on the data (similar to database cracking [12]).

An SDC comes in two flavors, one optimized for in-memory storage and one optimized for disk/cloud storage (see Figure 1). This is in contrast to Apache Arrow, which is focused on in-memory analytics, or Parquet, which is primarily a format for serializing tabular data to disk. Both in-memory and cloud-storage SDCs can be used simultaneously by several clients. Most commonly, an in-memory SDC is "backed" by a cloud storage SDC, but it doesn't

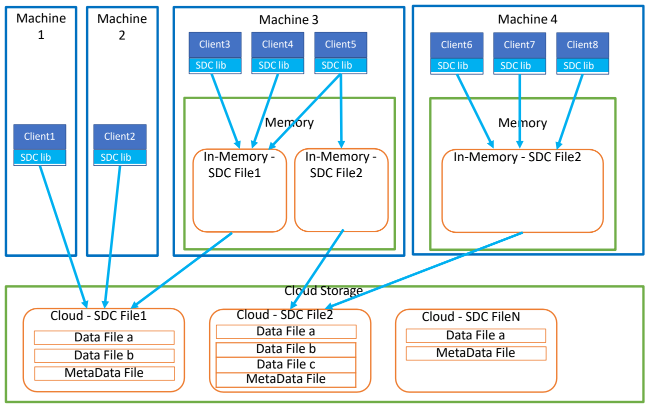


**Figure 1: SDC Architecture: Client 1 and 2 access the same SDC file on cloud storage. Client 3 and 4 work on the same in-memory SDC file 2, which is backed by a cloud SDC file.**

have to be. In the case where multiple in-memory SDCs are backed by the same cloud-storage SDC (such as File 2 in Figure 1), each in-memory SDC can be organized differently in a way that most suits their respective clients' access patterns.

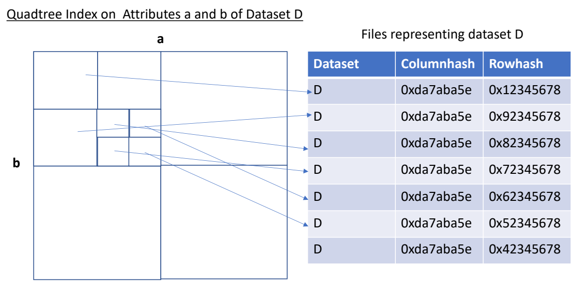
There are several possible physical representations for a given SDC. For example, an SDC can be a single mutable file (to support metadata additions and transformations) or a collection of files (some immutable, like the base data, some mutable for metadata and reorganizations), or even a collection of immutable files for storage on S3. However, in all cases the client/application will perceive it as a single file (like in Arrow). We describe how an implementation of SDCs on top of purely immutable storage would work in Section 3 below. Then, Section 4 provides some preliminary numbers show that SDCs can offer significant performance gains. Finally, Section 5 describes several open problems in SDCs, including different possible re-organizations and how clients coordinate to perform dataset optimization without a centralized coordinator.

## 3 STORING SDCS ON IMMUTABLE STORAGE

Figure 1 shows how several clients access a collection of SDCs. Note that there are no centralized services nor communication between them. Hence, all instance-optimizations have to be synchronized through the file itself.

In this section we describe how this is achieved. Namely, we describe how SDCs can be stored as a collection of objects on an immutable cloud-storage system like S3 (which only supports file upload, rename, and delete – i.e., no updates or appends). Our design is focused on providing the ability to evolve the physical layout of the SDC over time without requiring any intermediate service to mediate client access (i.e., libraries running on clients directly access the SDC files in cloud storage). This introduces challenges related to ensuring that clients see a consistent view of the data, and that they don't perform concurrent conflicting operations.

This design assumes that all clients have write access to the directory in which the SDC resides; clients that lack write access can still read SDCs but cannot participate in the adaptation protocol. While this certainly incurs additional overhead, previous work [6] has shown that it is possible to do even on S3. In this design, an


Quadtree Index on Attributes a and b of Dataset D
Files representing dataset D
a
b

| Dataset | Columnhash | Rowhash |
|---|---|---|
| D | 0xda7aba5e | 0x12345678 |
| D | 0xda7aba5e | 0x92345678 |
| D | 0xda7aba5e | 0x82345678 |
| D | 0xda7aba5e | 0x72345678 |
| D | 0xda7aba5e | 0x62345678 |
| D | 0xda7aba5e | 0x52345678 |
| D | 0xda7aba5e | 0x42345678 |

**Figure 2: SDC Index Example**

SDC consists of a number of different block types, including data blocks, index blocks, meta-data blocks, and lock files.

## 3.1 Data Blocks

Each SDC is represented as a collection of data blocks in storage, each stored as a different file. Blocks are read-only, variable sized, and can contain different subsets of columns. They may overlap (i.e., contain some of the same rows and columns) with other blocks.

Internally a data block is column-oriented. Each row has a specific row-identifier (rowid), and each block contains a rowid column and one or more data columns. Data columns can support standard types, including numeric and variable length objects such as string. Variable length objects are dictionary encoded, and dictionaries are separately stored in each data block. Data blocks are identified by the dataset id, a 128 bit hash of the concatenation of the column names (the column hash), and a 128 bit hash of the concatenation of the rows and rows in the block (the row hash), respectively.

Although not strictly required, most SDCs will contain a *primary replica*, which is a set of data blocks horizontally partitioned on rowid that contain all columns in the SDC.

## 3.2 Index blocks

Index blocks contain the structure of indexes that make it possible to find blocks of an SDC. All live SDC data blocks are referenced by an index block. An index block simply maps from logical ranges of the dataset to data files on the storage system. For example, Figure 2 shows how a quad-tree index might be implemented. The index block contains the extents of each cell in the quad tree (in terms of the attributes a and b), along with a pointer (filename) to the data block containing those records on storage.

We discuss several different types of indexes that SDCs can support below, including multi-dimensional space partitioning indexes like qd-tree [21] and Tsunami [10], as well as partial replication. One common index type is a sparse row index on the primary replica, which maps from contiguous rowid sets to the blocks of the replica. We call this the primary replica index.

## 3.3 Discovering the current state of an SDC

When a client wants to read a part of the SDC, it first finds all of the index blocks for the SDC, and then chooses the index it prefers to access the rows it wants to read.

Index blocks are stored in a well-known subdirectory of the SDC, making it easy to discover all of the indexes. Since indexes are expected to be mostly sparse (i.e., pointers to large contiguous data blocks, rather than pointers to individual records), they will be small, quick to read, and easy to keep in client memory.

Since most SDCs will have a primary replica index, a default access method is to use this index to discover all of the primary replica blocks and perform a sequential scan of the data through those blocks.

## 3.4 Meta-data blocks

Clients add meta-data blocks summarizing access patterns and distributional statistics of the blocks as they read and write them. Metadata blocks are essentially annotations that indicate the parts of the SDC that clients read, or derived statistics they have calculated over the SDC. For example, a metadata block might specify that a client C scanned a range $A_1[0] \rightarrow A_1[150]$, and might include a histogram for the attribute $A_1$. Statistics metadata might include histograms for a particular column or columns. We discuss in Section 3.6 how several metadata blocks can be consolidated into a single metadata block.

## 3.5 Evolving the contents of an SDC

The most common way to evolve an SDC is to create a new index on it. To create a new index, a client simply uploads any data blocks for the index, and then finally uploads the index blocks.

We expect that a common way to evolve an SDC will be to create new indexes that slightly modify existing indexes. For example, in the quadtree shown in Figure 2 at some point a client might decide to split the lower-left quadrant into four blocks, because, for example, it finds that it is often performing highly selective reads of small portions of this region. To do this, it simply creates four new data blocks for the four quadrants, and creates a new index referencing all of the old blocks from the three unmodified quadrants as well as the new four blocks for the lower-left quadrant.

The key challenge is that over time data blocks will accumulate and storage use will grow, so we need a way to reclaim space by removing unused indexes and the data blocks they reference.

## 3.6 Reclaiming Space

To reclaim storage, a client needs to first determine that it wants to remove some data, e.g., by using the meta-data blocks from the SDC to observe the recent accesses to the data set, and then make changes to the structure itself. Specifically a common operation is to delete an index and all of the data blocks that are accessed only by that index. Because other clients may be using the index at the time a client decides to delete it, a protocol is needed to prevent this from happening. Another common operation is to consolidate several meta-data blocks or histograms into a single block, so that clients have to read fewer files to access metadata.

A key observation in this protocol is that for most data sets, especially on cloud storage, space reclamation does not need to be done particularly quickly, because storage is essentially unlimited and wasted space is only paid for by the hour. Hence, an SDC defines a parameter, the maximum index cache duration (MICD), which is the maximum amount of time a client will retain a reference to an index and its data blocks in memory before re-loading from storage. Given the MICD, deletion work as follows.

First, a client $C_D$ that intends to delete an index creates a *deletion intent* file for the index, marked with a deletion timestamp when the deletion will happen. The existence of such a file indicates that an index is scheduled for deletion; the timestamp is set to the current time plus the MICD.

Then, when other clients read from the SDC, they will ignore any indexes for which a deletion intent exists. Clients which have the index in memory or are currently reading from it can continue to do so as for up to MICD time after they first read it. After the MICD timeout elapses they will have to refresh the indexes from that SDC. If an index they were using is scheduled for deletion they will have to choose a different access method.

Finally, when a client (not necessarily $C_D$) discovers that a deletion intent exists for an index with a timestamp greater than the deletion timestamp on the intent it can safely remove the index and the deletion intent.

## 4 PRELIMINARY EVALUATION OF SDCS

To demonstrate the potential of SDCs, we built a prototype implementation of a SDC[1]. Our prototype lacks many of the features described in this paper, but provides the ability to encode a dataset as a set of blocks laid out using either range partitioning or qd-trees [21]. Blocks are stored as Apache Parquet files, either on local disk or on cloud storage. We have built a simple API that allows clients to read a subset of the SDC data by applying a projection over the columns and a filter over the rows, and that records metadata about the read operations as they happen. Clients can periodically reorganize the SDC by calling an explicit layout optimize operation.

To evaluate SDCs, we used four public datasets and associated real-world query logs from an analytics dashboard. The datasets are a financial **contributions** dataset, a **flights** dataset, a geo-spatial dataset of NYC **taxi** rides, and a dataset of geo-coded **tweets**. The queries consist of single-table range queries that perform a filter and projection over multiple attributes (for example, selecting a date range and a latitude/longitude bounding box in the taxi or tweets data.) We believe these datasets are representative of the types of data and queries SDCs will be used for.

### 4.1 SDC on local disk

We first evaluate the performance of SDCs when all data is stored on local disk. We ran on a c5.9xlarge EC2 machine with 72GB RAM. For each dataset, we compare SDCs in three states: (1) with no layout optimizations, so the entire data is stored in a single Parquet file, (2) with range partitioning over the optimal column for each workload, so that data is stored in multiple Parquet files, and (3) with the layout optimized using qd-tree after observing a fraction of the workload, so that each qd-tree block is stored in its own Parquet file. We furthermore store partitioning metadata (e.g., the qd-tree index) and a query log on disk. Query execution is single-threaded, and for each query, we read the Parquet files (and possibly also the partitioning metadata) from disk and materialize the desired data as an Arrow table. Figure 3a shows the total time to run all queries in each workload. The optimized version of SDCs achieves up to 10× faster query times than SDCs with range partitioning.

[1]The source code will be made available at http://dsg.csail.mit.edu/mlforsystems#open-source.

### 4.2 Delta Lake

To compare the speedup achieved by SDC's layout optimization against optimized layouts in production systems, we also ran experiments over the same data using Delta Lake [5], a popular storage layout that adds support for updates and z-ordering on top of Parquet. We used Databricks Delta Lake version 9.1 LTS on a single-node cluster with a c5d.4xlarge EC2 machine. Delta Lake automatically stores a table as multiple Parquet files on S3. To evaluate Delta Lake for data on disk, we used Delta cache to pin the tables to local disk. Figure 3b shows the performance of Delta Lake with data stored on disk in three states: (1) with no layout optimization, (2) with data sorted over the optimal column for each workload, and (3) using the Delta Lake `OPTIMIZE ZORDER BY` command, which sorts the data according to a z-order over manually selected columns.

These results show that Delta Lake's layout optimization only outperforms a single-column sort layout by up to 1.24×, which is much less than the performance benefit of SDC's layout optimization. The reason for this difference is that although SDC's qd-tree index and Delta Lake's z-order use roughly the same columns (qd-tree automatically determines the important columns, and we manually selected the optimal columns for Delta Lake's z-order), the qd-tree's layout is more precise, since it creates blocks with the specific goal of minimizing data access for the observed workload.

### 4.3 SDC on cloud storage

We have also performed preliminary evaluations of SDCs stored as objects on AWS S3, which are read into the memory of the EC2 machine during query processing. In this setting, SDCs with optimized layout achieve up to 3× faster query times than SDCs with range partitioning. The smaller performance benefit of layout optimization on S3 when compared to disk is due to the higher overhead of accessing data from cloud storage—as a result, SDC is forced to use larger block sizes when storing data on S3, which means that for the same dataset, SDC's optimized layout is coarser-grained on S3 than it is on disk. We believe that with further implementation optimizations and parallelism support, we can efficiently support finer-grained layouts and therefore further improve SDC's performance on cloud storage.

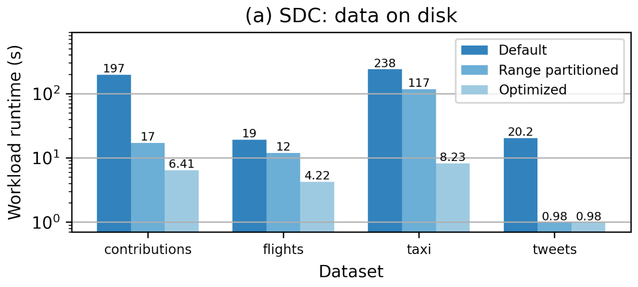
## 5 RESEARCH DIRECTIONS FOR SDCS

So far, we described the high level idea of SDCs and showed the advantages of optimized storage layouts. In this section, we focus on some of the new problems and opportunities that SDCs present.

### 5.1 Variety of Physical Layouts

A principal motivation for SDCs is that they can encompass a variety of physical layouts, several of which we described above and have explored in our previous work [8, 10]. However is a large additional space of physical layouts to explore, including partial materialization (e.g., of common queries or subqueries), or replication of certain data elements to support access to blocks of data without needing to read unnecessary subsets of data.

We believe replication offers a potential for a significant generalization of previous work on efficient layouts [10]. Consider, for example, Figure 4a. Here, different queries that predicate on attributes X and Y are shown as rectangles over the domain of the



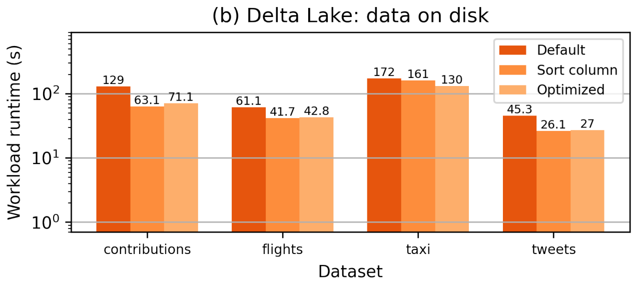
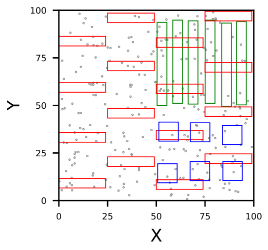
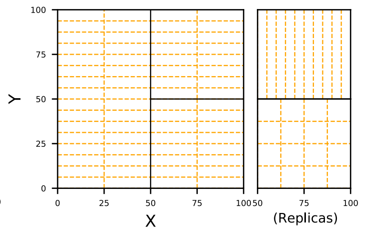


**Figure 3: SDC's layout optimization achieves up to 10× speedup over optimal range partitioning. On the other hand, layout optimizations for production system such as Databricks Delta Lake, which uses z-ordering, are not as effective.**





**(a) Dataset and Workload** **(b) Layout w/ 0.5X replication budget**

**Figure 4: Figure 4a shows a multi-dimensional dataset and workload with 2 columns X and Y. Gray dots represent tuples; colored rectangles represent queries. Figure 4b shows a layout with enough budget to replicate half the table. The right half of the data space is replicated.**

variables. From the figure, it's clear that different queries select different ranges of each attribute and cover different parts of the data space. Without replication, the storage layout would need to account for the average query; in the upper- and lower-right quadrants this means that the layout would not be optimal for the red, green, or blue queries, since they have such different shapes. However, by replicating 50% of the data, each type of query could have access to an optimal layout. Figure 4b show such an optimal design; here the yellow dashed boxes correspond to physical partitions of the SDC. The left (full) replica contains partitions optimized for the red queries, while the upper-right replica contains optimal partitions for the green queries, and the lower-right contains optimal partitions for the blue queries. Designing algorithms that are able to find such optimal partial partitioning layouts presents an intriguing challenge, since the space of all possible replicas is exponential in several dimensions of the problem.

## 5.2 Hybrid Layouts & Novel Hardware

As noted above, we envision that SDCs will have both a persistent (on disk/cloud storage) and in memory component, and that many clients will load data into memory and then further optimize their layouts for efficient access. One interesting direction for SDCs is to explore optimized layouts for novel hardware, such as GPUs and NVRAM, such as Intel Optane®. These devices have different random access latencies and are optimized for different sized blocks than either RAM, S3, or Flash drives. Further, it may be helpful to have SDCs encompass replicas that are optimized for different types of hardware, if clients with such specialized hardware are frequently accessing data.

## 5.3 Incremental changes

It's possible that SDCs may be created with optimal layouts at the outset. However, since the workload may not be known in advance, it's useful to be able to adapt the storage of the SDCs over time, using algorithms like those described in Section 3. However, it's unlikely that one would re-write an entire dataset in one pass, particularly if we rely on clients to make incremental changes to the dataset. Hence, a key challenge to is identify algorithms that incrementally re-organize data as clients access it, similar to database cracking [12], but focused more on re-partitioning the actual data storage rather than optimizing the structure of indexes over time. Although this idea has been partially explored in prior work [9, 18], developing incremental transformation schemes for multi-dimensional indexing and partial replication settings, and in a way that does not place undue burden on any one client, is an important area for investigation.

In addition to developing algorithms that describe *how* to perform incremental re-organization, another key challenge is developing policies to determine *when* to reorganize. A straightforward policy is to react to the recent history of access patterns (e.g., reorganize if the past day's access patterns are significantly different from the average pattern over the preceding week), but this policy may react too slowly to sudden access pattern changes or may falsely react to random noise. A proactive policy could schedule incremental changes in anticipation of future changes in the workload, but this requires accurate forecasting of the future workload, which is an active area of research [14].

## 5.4 Auto-optimization

SDCs can encompass multiple optimized physical layouts and access methods, including coarse indexes for partitioning schemes and several different replicas. Client libraries need to embed logic for choosing what scheme to use when answering a query, which requires loading the metadata and index blocks and using traditional query optimization methods. A key challenge here is that the metadata may be spread across many blocks, and it may not be practical for a client to read all of them prior to processing any requests. Hence, clients will need to asynchronously fetch metadata, and will likely have a partial and possibly out of date view of it, which they will need to use to answer queries as best as possible.

In addition, because clients are active participants in the reorganization process, they need to select which transformations to make to the data next. Since many transformations are possible, and will be done incrementally by different clients, methods to coordinate the behavior of several clients so they converge towards an optimal design are needed. One possible way to achieve this is via additional metadata blocks that express partial re-organizations that are in flight, so other clients can continue them. A natural way to express such optimization objects is through reinforcement learning, but spreading this across multiple clients raises the interesting prospect of a shared, distributed RL framework.

One critical part of any auto-optimization scheme is the optimization objective. Prior work on self-organizing data layouts often optimize to minimize average query time, but is also possible to optimize for objectives such as space usage (when maintaining a fixed SLA for query latency) and monetary cost (especially when storing on the cloud). Each optimization objective likely requires a different optimization algorithm.

### 5.5 Encryption and Access Control

Finally, we envision that SDCs can encompass a variety of other cloud-storage use cases. We are particularly interested in using SDCs to public as a well private data, where only particular authorized clients can read or modify the SDC. The typical way to achieve this would be to encrypt the SDC use a symmetric-key encryption scheme, and then embed a list of authorized users (e.g., as a set of public keys) with the SDC. Users who want to read the SDC would submit a signed request to some security service, which would authenticate the user and then securely send the encryption key to the client to allow it to read the data. Unfortunately, this requires a server, which is something we would like to avoid.

In cryptography, allowing a single publisher to deliver encrypted data to a large number of clients with whom it wants to share data is known as broadcast encryption [11], and was originally developed, e.g., to allow media companies to share encrypted data with only authorized subscribers or devices over one-way channels like cable TV. However, SDCs as we have described them are not a pure one-way communication channel, as different clients will need to modify the data over time. One possible way to resolve this is to use broadcast-encryption to encode a symmetric key that can be used to encrypt/decrypt the SDC, so that only authorized users can read the dataset. However, this introduces the possibility that this single key could leak and risk the forward secrecy of the dataset. Research is needed to identify schemes that provide the best tradeoffs between data protection, forward secrecy, and flexible access control.

We also plan to investigate whether it is possible to extend this model to encryption schemes that support direct operations, like the schemes used in CryptDB [17], for example, order-preserving encryption which supports subsetting on encrypted data without reading or decrypting the entire dataset.

## 6 CONCLUSION AND DISCUSSION

We presented our vision for self-organizing data containers (SDCs), a storage format designed to allow disaggregated, cloud-based data processing systems to take advantage a number of storage techniques, such as materialization, replication, and multi-dimensional partitioning and indexing. We presented an approach that allows SDCs to evolve their physical layout via client-side interactions with the data, even when data is stored on immutable storage like Amazon S3. We also presented a number interesting challenges that the SDC design presents, including new challenges around storage layouts, incremental data optimization, and encryption and access control without a centralized key management system.

**Applicability of SDCs**. Given SDCs, we believe a number of data-intensive applications could benefit – any application that currently uses storage formats like Parquet or Arrow could easily be reconfigured to used SDCs, including database systems like Spectrum, cloud data processing frameworks like Spark, or a variety of data and ML platforms that are increasingly relying on these data formats for data exchange and serialization.

**Acknowledgements.** This research is supported by the MIT Data Systems and AI Lab (DSAIL) and by NSF IIS 1900933.

## REFERENCES

[1] 2021. Apache Arrow. https://arrow.apache.org/docs/.

[2] 2021. Apache Hudi. https://hudi.apache.org/.

[3] 2021. Apache Iceberg. https://iceberg.apache.org/.

[4] 2021. Apache Parquet. https://parquet.apache.org/documentation/latest/.

[5] 2021. Delta Lake. https://delta.io/.

[6] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *ACM SIGMOD*. 251–264.

[7] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *Proc. CIDR*.

[8] Philippe Cudré-Mauroux, Eugene Wu, and Samuel Madden. 2009. The Case for RodentStore: An Adaptive, Declarative Storage System. In *Proc. CIDR*.

[9] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *Proc. SIGMOD*. 418–431.

[10] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proc. VLDB Endow.* 14, 2 (Oct. 2020), 74–86.

[11] Amos Fiat and Moni Naor. 1994. Broadcast Encryption. In *Advances in Cryptology — CRYPTO' 93*. Berlin, Heidelberg, 480–491.

[12] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In *Proc. CIDR*. 68–78.

[13] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *Proc. CIDR*.

[14] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-Based Workload Forecasting for Self-Driving Database Management Systems. In *Proc. ICDE*. 631–645.

[15] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. 2020. Learning Multi-Dimensional Indexes. In *Proc. SIGMOD*. 985–1000.

[16] Shoumik Palkar, James J. Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2017. Weld: Rethinking the Interface Between Data-Intensive Applications. *CoRR* abs/1709.06416 (2017). arXiv:1709.06416

[17] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. SOSP*. 85–100.

[18] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge Quiane, and Aaron J. Elmore. 2017. A Robust Partitioning Scheme for Ad-Hoc Query Workloads. In *Proc. SOCC*. 229–241.

[19] Leonhard F. Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *ACM SIGMOD*. ACM, 1718–1731.

[20] Junjay Tan, Thanaa Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing a Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2170–2182.

[21] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. 2020. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *Proc SIGMOD*. 193–208.

[22] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. SOCC*. 10.