Henosis: Workload-driven small array consolidation and placement for HDF5 applications on heterogeneous data stores

Donghe Kang, Vedang Patel, Ashwati Nair, Spyros Blanas, Yang Wang, Srinivasan Parthasarathy
{kang.1002,patel.3140,nair.131,blanas.2,wang.7564,parthasarathy.2}@osu.edu
The Ohio State University

ABSTRACT

Scientific data analysis pipelines face scalability bottlenecks when processing massive datasets that consist of millions of small files. Such datasets commonly arise in domains as diverse as detecting supernovae and post-processing computational fluid dynamics simulations. Furthermore, applications often use inference frameworks such as TensorFlow and PyTorch whose naive I/O methods exacerbate I/O bottlenecks. One solution is to use scientific file formats, such as HDF5 and FITS, to organize small arrays in one big file. However, storing everything in one file does not fully leverage the heterogeneous data storage capabilities of modern clusters.

This paper presents Henosis, a system that intercepts data accesses inside the HDF5 library and transparently redirects I/O to the in-memory Redis object store or the disk-based TileDB array store. During this process, Henosis consolidates small arrays into bigger chunks and intelligently places them in data stores. A critical research aspect of Henosis is that it formulates object consolidation and data placement as a single optimization problem. Henosis carefully constructs a graph to capture the I/O activity of a workload and produces an initial solution to the optimization problem using graph partitioning. Henosis then refines the solution using a hill-climbing algorithm which migrates arrays between data stores to minimize I/O cost. The evaluation on two real scientific data analysis pipelines shows that consolidation with Henosis makes I/O 300× faster than directly reading small arrays from TileDB and 3.5× faster than workload-oblivious consolidation methods. Moreover, jointly optimizing consolidation and placement in Henosis makes I/O 1.7× faster than strategies that perform consolidation and placement independently.

1 INTRODUCTION

The data volume processed by scientific pipelines is increasing rapidly. Scientific pipelines in various domains, such as plasma simulation [9], climate modeling [16], transient detection [19], and computational fluid dynamics, analyze massive amounts of array data that range up to petabytes. For example, the Large Hadron Collider produces approximately 15 petabytes of data annually, and the Sloan Digital Sky Survey (SDSS) [7] archives terabytes of data for hundreds of millions of astronomical objects.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '19, June 26–28, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330380

Although the aggregate data volume of many datasets is enormous, individual observations are often stored as independent small files for downstream processing. The supernovae detection pipeline in the ASAS-SN sky survey stores detected transient stars in separate 1.7 KB files [19]; a vortex prediction pipeline on the simulation of a Mach 1.3 jet produces vortices with average size 8 KB [36]; sequencing the human genome produces nearly 30 million files averaging 190 KB each [8]; and 20 million images served by SDSS are less than 1 MB on average [7]. These files are small compared with the typical block sizes of modern file systems. For example, the default block size of the Hadoop Distributed File System (HadoopFS) is 64 MB and the default stripe size of the Lustre parallel file system is 1 MB. Achieving respectable I/O performance when accessing a large number of small files is particularly challenging for a distributed file system. In addition, it is cumbersome and error-prone for users to organize large collections of small files manually.

One solution that has been embraced by scientists is storing such datasets in array-centric file formats like HDF5, netCDF and FITS. These file formats arrange collections of objects in an internal hierarchy, offer richer metadata support than file systems, and store entire collections of objects as a single file. However, these file format libraries adopt a monolithic design that tightly couples an array-centric API with a particular physical data layout. This monolithic design is not well-suited to the heterogeneous I/O capabilities of modern clusters. Storing everything in one file in the parallel file system does not fully utilize nodes with large memory or nodes with locally attached flash-based storage. A large memory node would be an ideal deployment setting for an in-memory key/value store such as Redis, while fast locally-attached storage can be utilized with a locality-conscious file system in user space, such as HadoopFS. Unfortunately, established array-centric file format libraries do not support multiple storage backends with heterogeneous I/O capabilities.

This paper describes Henosis, an I/O library that allows HDF5-based programs to consolidate, place and read small arrays on heterogeneous data stores. The prototype implementation of Henosis that we describe in this paper supports two storage backends: (1) Redis [25], an in-memory object store with a key/value interface, and (2) TileDB [28], an array management system for distributed file systems such as HadoopFS. By leveraging the recent virtual object (VOL) interface of the HDF5 library, existing HDF5 applications can benefit from the I/O optimizations of Henosis without recompilation, as Henosis makes no modifications to the public HDF5 API and can be dynamically loaded at runtime.

The research question that naturally arises when supporting multiple storage backends is how one should physically lay out array data across data stores with heterogeneous I/O capabilities. Consider, for example, the Redis and TileDB data stores mentioned

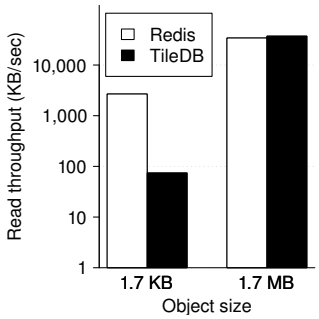


Figure 1: Read throughput of Redis, an in-memory object store, and TileDB, a disk-based array store, when accessing small (1.7 KB) and large (1.7 MB) objects. Redis is nearly 100× faster than TileDB with small objects, but read throughput is statistically indistinguishable with large objects.

earlier. If one stores small objects that are 1.7 KB each, the read throughput of Redis outperforms TileDB by nearly 100×, as shown in Figure 1. This is an instance of the data placement problem on heterogeneous storage systems, a topic which has been well-studied in prior literature [11–13, 24, 31, 38]. Yet, if one stores objects that are 1.7 MB each, the read throughputs of TileDB and Redis become statistically indistinguishable. Although consolidation is less explored in prior research, the results clearly show that it has the potential to equalize I/O performance as long as the requested small objects can be consolidated into sufficiently big chunks. However, consolidation and placement are not orthogonal optimizations: A consolidation-oblivious placement algorithm misses opportunities to place data in a manner that benefits from sequential I/O. Conversely, a placement-oblivious consolidation algorithm cannot differentiate between fast and slow I/O devices that often co-exist in modern HPC systems.

This paper describes how Henosis formulates consolidation and placement as a single optimization problem. Henosis carefully constructs a graph to capture the I/O activity and produces an initial solution to the optimization problem using graph partitioning. Henosis then uses a hill-climbing algorithm to iteratively refine the solution by migrating arrays between data stores to minimize an I/O cost metric. Experimental results from Henosis on two real scientific pipelines, transient star detection and vortices prediction, show that Henosis speeds up I/O by 300× compared with native TileDB (which doesn't support consolidation) and 3.5× compared with workload-oblivious consolidation methods. The iterative refinement procedure in Henosis speeds up I/O performance by nearly 3× compared with the initial solution from graph partitioning and 1.7× compared with two strategies that perform consolidation and placement independently.

The main contributions of this paper are:

(1) We formulate array consolidation and placement as a single optimization problem that allows both techniques to be considered simultaneously.
(2) We design a heuristic method to optimize the array storage plan that consists of two steps. The first step devises an initial storage plan by creating and partitioning a *query-weighted* graph. The plan is then iteratively refined by moving arrays between TileDB and Redis based on an I/O cost metric.
(3) We design and implement an I/O library prototype, Henosis, to transparently consolidate, place and read small arrays for HDF5 applications. The evaluation, based on two real workloads, shows that Henosis accelerates I/O by 300× compared with native TileDB and 1.7× over workload-oblivious consolidation and placement strategies.

The remainder of the paper is structured as follows. Section 2 presents necessary background on the HDF5 library. Section 3 describes two application drivers that process small arrays. Section 4 describes the Henosis architecture. Section 5 formulates consolidation and placement as a single optimization problem, and proposes a heuristic method to optimize the array storage plan based on graph partitioning and the hill climbing technique. Section 6 follows with more details about the implementation. Section 7 describes the experimental setup and presents the performance evaluation, Section 8 presents related work, and Section 9 concludes.

2 BACKGROUND

This section first introduces the array model which is prevalent in scientific computing. It then describes two new features of the HDF5 array library, namely the virtual dataset (VDS) and the virtual object layer (VOL), which allow Henosis to transparently store and consolidate arrays on different storage backends. Although this paper focuses on the HDF5 library, the consolidation and placement techniques are also applicable to other array formats.

Scientific datasets in various domains such as astronomy, physics, and medicine can be represented by arrays. Arrays are said to be dense when every cell has an associated value, or sparse when the majority of the cells are empty. Sparse arrays are sometimes stored as dense arrays after filling all empty cells with a *null* value. Dense arrays are commonly stored in a *chunked* layout. A chunk is a subarray bounded by a (hyper-)rectangle that covers adjacent cells. Chunks commonly have a fixed user-defined size, although research prototypes like ArrayStore support irregular chunks that cover a different volume of the coordinate space [34].

HDF5 is a prominent scientific data format used to manage arrays and is the one we have developed Henosis on. The two HDF5 operations of interest are read and write. The read operation returns the values of any subset of the cells in an array. Inversely, the write operation updates a subset of the cells in an array. In both operations, users define the accessed cells either as a set of (hyper-)rectangles or as a list of points.

Virtual object layer: The virtual object layer (VOL) is a new abstraction layer in HDF5 that allows one to intercept and inject I/O operations without modifying the application-facing public HDF5 interface. VOL intercepts all the function calls that manipulate data and routes them to a custom, user-defined virtual object driver. The driver is a C program that performs user-defined operations, such as storing data in another representation. Henosis implements a VOL driver that redirects I/O operations to other storage backends. This allows existing HDF5 applications to leverage the Henosis functionality without modification.

Virtual datasets: The virtual dataset (VDS) is a recent feature of the HDF5 library that allows one to construct non-materialized array views on HDF5 datasets. A virtual dataset defines a mapping

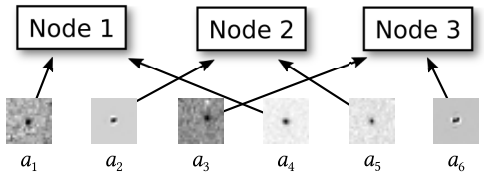


Figure 2: Transient star detection in ASAS-SN.

from a set of (hyper-)rectangles in one or more source datasets to a contiguous target address space. After a virtual dataset has been created, applications access the target address space using the standard HDF5 programming interface. When the target address space is accessed, the data corresponding to this access will be retrieved from the source locations defined in the mapping. The virtual dataset feature allows Henosis to physically store data contiguously in a large array but present the logical view of many small objects to existing HDF5 applications. By defining these virtual datasets, reads or writes to small arrays will be transparently redirected by the HDF5 library to a large chunk that consolidates many small arrays.

3 APPLICATION DRIVERS

Henosis is motivated by the complex I/O patterns of modern applications that use data mining and machine learning techniques on large complex datasets. Many ML-centric data processing pipelines operate on a large number of small arrays that are processed and managed individually. These small arrays are typically produced once, often by a classification and segmentation procedure from a much larger dataset. Then the small arrays are analyzed multiple times. In-place updates to individual small arrays are rare and are usually done manually by scientists. We observed this pattern in two scientific application drivers that analyze observational and simulation data, respectively.

Supernovae detection in large-scale astronomy sky surveys. Large-scale systematic astronomy sky surveys search for transient, variable stars at multiple optical frequencies. One such project is ASAS-SN [19]. During transient detection in ASAS-SN, the sky survey image data are segmented into many small 21×21 pixel images, each 1.7 KB in size, shown as $a_1, a_2, \ldots, a_6$ in Figure 2. The pipeline analyzes these arrays by a Convolutional Neural Network-based classifier in TensorFlow to detect supernova. Because the images are classified independently, the pipeline can be executed on several nodes concurrently to process disjoint subsets of the sky survey images. Figure 2 shows three nodes evaluating the detection pipeline independently, each of which processes two images.

Vortices prediction in computational fluid dynamics simulations. The need to manage small arrays also arises during the analysis of simulations of complex phenomena, such as computational fluid dynamics. One such analysis focuses on vortex prediction in turbulent flows. Turbulent flows are characterized by a broad range of spatial and temporal fluctuations. Vortices are fluctuations that occur across larger scales, and retain their signature over prolonged durations in space and time. Vortices prediction isolates coherent structures from the fluid flow and predicts their dynamics [41]. In this pipeline, density-based spatial clustering (DBSCAN) and machine learning (LSTM) are used to predict acoustic emissions

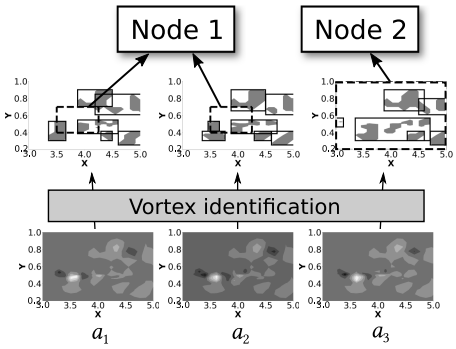


Figure 3: Vortices prediction in a Mach 1.3 jet.

of a supersonic jet. This pipeline accesses a large number of vortices that are stored as independent small arrays when identifying vortices in each timestamp. Figure 3 shows arrays a_1, a_2, a_3 from simulation timestamp $t = 1, 2, 3$, respectively. The vortices that were identified in the three simulation timestamps are bounded by rectangles with solid lines. There are two types of queries in this pipeline which are indicated with a dashed box in Figure 3. The first query type tracks vortices between timestamps and accesses files that intersect a spatio-temporal box. For example, node 1 in Figure 3 reads vortices where $3.5 \leq X \leq 4.25$, $0.4 \leq Y \leq 0.7$ and $1 \leq t \leq 2$. The second query type accesses all vortices in a given timestamp for visualization. For example, node 2 in Figure 3 reads all vortices for timestamp $t = 3$.

4 SYSTEM OVERVIEW

Henosis facilitates small array management by consolidating and placing small arrays on data stores with heterogeneous I/O capabilities. Henosis stores arrays in two data stores, TileDB and Redis. Henosis stores an array either as a single key/value item in Redis or it consolidates multiple small arrays into a chunk in a TileDB array. Henosis does not currently replicate arrays across data stores.

One required parameter to create a TileDB array is the chunk size. When the TileDB chunk size is small, more I/O requests are sent to the underlying file system. When the chunk size is large, more data are transferred if users only request a subset of small arrays in a chunk. The impact of this configuration parameter is evaluated in Section 7.2.3.

Figure 4 shows the Henosis optimization workflow. Henosis first monitors and logs the access patterns of HDF5 applications to produce a *workload specification*. A workload specification tracks

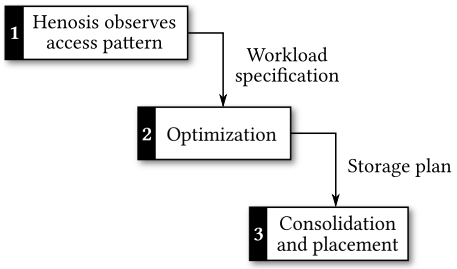


Figure 4: The Henosis optimization workflow.

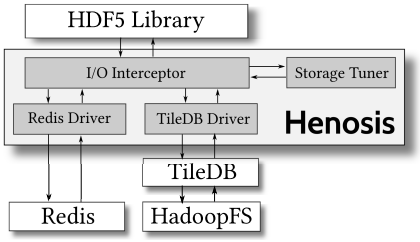


Figure 5: The Henosis system architecture.

which small arrays are requested by each process of the observed application. Sections 5.1 formally defines the workload specification. Henosis then generates a *storage plan* through the optimization process described in Sections 5.2 and 5.3. The storage plan determines how small arrays will be consolidated and placed on TileDB and Redis. An offline process performs consolidation and placement of the data based on the storage plan.

Henosis intercepts I/O to small arrays and redirects them to the appropriate data store at runtime. Figure 5 shows the architecture of the Henosis system. If one looks at the software stack of HDF5 applications, Henosis sits below the HDF5 library and above the Redis and TileDB data stores. Henosis is composed of four main components, namely the *I/O Interceptor*, the *Storage Tuner*, and two backend drivers for TileDB and Redis. The *I/O Interceptor* is virtual object layer (VOL) driver for the HDF5 library that intercepts I/O-related function calls, such as the read method (`H5Dread`) of the HDF5 API. When a process requests to read a small array, Henosis transparently forwards the I/O request to the appropriate data store and logs the access pattern. The *Storage Tuner* finds a storage plan based on the observed access pattern. It then consolidates and places small arrays based on the storage plan. The *TileDB Driver* and the *Redis Driver* read data from TileDB and Redis respectively.

Henosis is a first step towards *data independence* for scientific data analysis. With Henosis, applications can remain agnostic to the underlying data storage layout and program against the standard array-centric HDF5 interface. The main advantage of designing Henosis to work behind an array file format library interface is easy adoption by existing applications. This way Henosis can optimize data placement and accelerate I/O without additional effort from application developers.

5 THE STORAGE OPTIMIZATION PROBLEM

5.1 Preliminary Definitions

Capacity constraints. Redis and TileDB cannot store an unlimited number of small arrays due to the finite space of disk and memory. We denote C_{keys} as the maximal number of small arrays that Redis can store. TileDB stores up to C_{chunks} chunks. The consolidation factor C_f is the number of small arrays that can be consolidated into a TileDB chunk. Hence, Henosis stores at most $C_{chunks} \times C_f$ small arrays in TileDB.

Storage plan. Given a set of small arrays $A = \{a_1, a_2, \cdots, a_N\}$, a storage plan is a partitioning over A. One partition, referred to as the *Redis partition*, contains small arrays stored in Redis, and all other partitions contain small arrays that are consolidated in TileDB chunks. Due to the capacity constraints, the size of the *Redis partition* cannot be greater than C_{keys} and the sizes of the other partitions cannot be greater than C_f. A storage plan $\langle S, L\rangle$ is represented as two 0/1 variables, a matrix S and a vector L, which are defined as follows:

$$S_{i,j} = \begin{cases} 1, & \text{if array } a_j \text{ is stored in TileDB chunk } i \\ 0, & \text{otherwise} \end{cases}$$
$$L_j = \begin{cases} 1, & \text{if array } a_j \text{ is stored in Redis} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Workload specification. Assume Henosis observes P processes accessing N small arrays. The workload specification W is a $P \times N$ matrix that is constructed as such:

$$W_{i,j} = \begin{cases} 1, & \text{if process } p_i \text{ reads array } a_j \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

Example. Figure 6 shows an example of the I/O optimization in Henosis. Figure 6a shows seven processes, $p_1, ..., p_7$, accessing eight small arrays $a_1, ..., a_8$. The workload specification W is shown in Figure 6b. Suppose that the I/O optimization procedure determines that a_4 and a_5 will be stored in Redis, and two TileDB chunks will consolidate $\{a_1, a_2, a_3\}$ and $\{a_6, a_7, a_8\}$ respectively. Figure 6c shows the storage plan $\langle S, L\rangle$ for this configuration.

5.2 Problem Definition

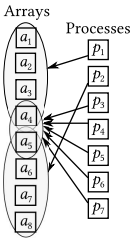
Let T_{chunk} and T_{key} be the time to access a TileDB chunk and a Redis key/value item. Given a a storage plan $\langle S, L\rangle$ and a workload specification W, which represents P processes accessing N arrays, we define the cost function $cost(S, L, W)$ as:

$$cost(S, L, W) = T_{chunk} \times count(SW^{\mathsf{T}}) + T_{key} \times sum(WL) \quad (3)$$

The $count(\cdot)$ function counts the number of non-zero elements in the input matrix, and the $sum(\cdot)$ function returns the sum of all elements in the input matrix. Cell (i, j) of the matrix SW^{T} is the number of arrays process p_j reads from TileDB chunk i. Element j of the vector WL is the number of arrays that process p_j will read from Redis. For TileDB, it suffices to count non-zero elements in SW^{T} because all arrays in a chunk are retrieved in a single I/O operation regardless of how many are actually accessed. For Redis, it is necessary to sum all accesses because arrays are stored as separate key/value items which require multiple I/O requests.

Consider the workload specification W and the storage plan $\langle S, L\rangle$ shown in Figure 6. The model estimates a total of two TileDB chunk accesses, as $count(SW^{\mathsf{T}}) = 2$, and twelve Redis key/value accesses, as $sum(WL) = 12$. Therefore from Formula 3, $cost(S, L, W) = 2\,T_{chunk} + 12\,T_{key}$.

There are three constraints a valid solution must meet. Due to limited memory for Redis, the number of non-zero elements in L, $sum(L)$, cannot be larger than C_{keys}. Similarly for TileDB, the number of non-zero elements in each row i in S, $sum(S_{i,:})$, cannot exceed C_f. Given that Henosis does not replicate data, array a_j must be stored exactly once, hence either $L_j = 1$ or $sum(S_{:,j}) = 1$. We now define the *Storage Optimization* problem:



(a) Access pattern.

Requested arrays	$W =$ $a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8$
$p_1 = \{a_1, a_2, a_3, a_4\}$	1 1 1 1 0 0 0 0
$p_2 = \{a_5, a_6, a_7, a_8\}$	0 0 0 0 1 1 1 1
$p_3 = \{a_4, a_5\}$	0 0 0 1 1 0 0 0
$p_4 = \{a_4, a_5\}$	0 0 0 1 1 0 0 0
$p_5 = \{a_4, a_5\}$	0 0 0 1 1 0 0 0
$p_6 = \{a_4, a_5\}$	0 0 0 1 1 0 0 0
$p_7 = \{a_4, a_5\}$	0 0 0 1 1 0 0 0

(b) Workload specification W.

TileDB

	Stored arrays	$S =$ $a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8$
Chunk 1	$\{a_1, a_2, a_3\}$	1 1 1 0 0 0 0 0
Chunk 2	$\{a_6, a_7, a_8\}$	0 0 0 0 0 1 1 1

Redis

Stored arrays	$L^\mathsf{T} =$ $a_1\,a_2\,a_3\,a_4\,a_5\,a_6\,a_7\,a_8$
$\{a_4, a_5\}$	$\langle 0, 0, 0, 1, 1, 0, 0, 0\rangle$

(c) Storage plan $\langle S, L\rangle$.

Figure 6: Example of the storage optimization in Henosis.

Storage Optimization Problem. *Given a workload specification W find a storage plan $\langle S, L\rangle$ that minimizes $cost(S, L, W)$:*

$$\begin{aligned} & \underset{\langle S,L\rangle}{\operatorname{argmin}}\ cost(S, L, W) \\ \textit{such that}\quad & sum(S_{i,:}) \le C_f \qquad \forall i \in [1, C_{chunks}] \\ & sum(L) \le C_{keys} \\ & sum(S_{:,j}) + L_j = 1 \quad \forall j \in [1, N] \end{aligned} \tag{4}$$

5.3 Optimization Procedure

Finding the optimal storage plan is not easy. The number of 0/1 variables is $N \times (C_{chunks} + 1)$ and both N and C_{chunks} can be in the millions even for modestly-sized datasets. In addition, the objective function of the *Storage Optimization Problem* is non-polynomial.

Henosis uses a heuristic optimization procedure, shown in Algorithm 1, to obtain an initial solution and then iteratively refine the solution to lower the total cost. The algorithm proceeds in two phases. The algorithm first obtains an initial storage plan $\langle S^0, L^0\rangle$ by graph partitioning, as described in Section 5.3.1. This initial storage plan is then iteratively refined to reduce its cost by migrating arrays between Redis and TileDB, as described in Section 5.3.2. The iterative refinement stops when no migrations are possible.

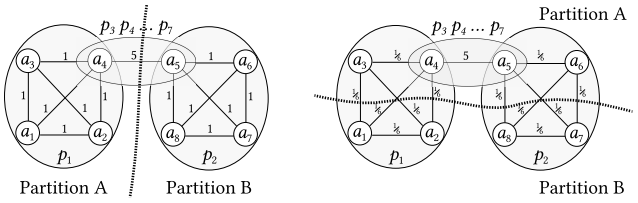
Algorithm 1: Henosis I/O optimization algorithm

```
⟨S^0, L^0⟩ ← initialize a storage plan by graph partitioning;
repeat
    u ← a set of TileDB arrays in S^{i-1} with the largest
        benefit that take less than the free space in Redis;
    if benefit(u) > 0 then
        ⟨S^i, L^i⟩ ← migrate all arrays in u to Redis;
    foreach Redis array k in L^{i-1} do
        c ← the TileDB chunk that can store k and achieves
            the largest cost reduction from migration;
        if cost reduction > 0 then
            ⟨S^i, L^i⟩ ← migrate k to TileDB chunk c;
until ⟨S^i, L^i⟩ = ⟨S^{i-1}, L^{i-1}⟩;
```

5.3.1 Initial storage plan. Intuitively, the cost is minimized when small arrays with high co-access frequency are stored in the same TileDB chunk, because these arrays can be retrieved using a single I/O operation. In the example shown in Figure 6, process p_1 needs to only issue a single I/O request to read small arrays a_1, a_2 and a_3, because they are consolidated into one chunk. In contrast, p_1 would issue multiple I/O requests if the three small arrays were stored in different chunks or in Redis. Based on this empirical observation, Henosis obtains the initial storage plan by solving a *partitioning problem* to group co-accessed arrays together.

Graph partitioning has been used in prior work such as Schism [15] to minimize cross-partition transactions in transaction processing workloads. Schism builds an *object-weighted graph* to represent small arrays and workloads. Nodes in the graph are objects and an edge connects two nodes if the two objects are co-accessed by the same processes. The weight of the edge is the number of processes that co-access the two objects. Figure 7a shows the object-weighted graph that will be constructed for the example shown in Figure 6a. The graph is partitioned such that the partition sizes are balanced and the sum of the weights of the edges that are cut is minimized. The problem with partitioning an object-weighted graph is that it is agnostic to the number of accesses a process performs. Hence, it is often prone to scattering co-accessed arrays to different chunks. Consider how to create two equally-sized partitions in Figure 7a. The minimum cut splits a_4 and a_5 to different partitions, although they are co-accessed by all but two processes.

Henosis uses a *query-weighted graph* to guide partitioning. The query-weighted graph represents small arrays as nodes and connects two nodes if the arrays are co-accessed by the same process. However, the edge weight calculation is different: edge weights are initialized to zero and increase as processes are added to the graph. Assume process p_i accesses $|p_i|$ arrays and is added to the graph. Edge e connects two arrays which are both accessed by p_i. In the query-weighted graph, the weight of e increases by $\frac{2}{|p_i| \times (|p_i|-1)}$. Thus, the sum of edge weights increases by 1 for each process regardless of how many arrays the process accesses. Figure 7b shows how the query-weighted graph is partitioned. Any minimum cut that produces two equal partitions keeps arrays a_4 and a_5 in the same partition, as the query-weighted graph has reduced the edge weights for all other node pairs from 1 to $\frac{1}{6}$.



(a) Object-weighted graph. (b) Query-weighted graph.

Figure 7: Graph representation of accesses in Henosis.

5.3.2 Iterative refinement. Once the initial storage plan has been obtained from graph partitioning, Henosis iteratively refines the storage plan to reduce the anticipated I/O cost by migrating arrays between Redis and TileDB chunks. Migrating arrays from TileDB to Redis reduces the access frequency to TileDB chunks, but increases the access frequency to key/value items. Migrating arrays from Redis to TileDB conversely impacts the frequencies. The trade-off is that although accessing an individual array is much faster in Redis than in TileDB, TileDB can retrieve multiple arrays more efficiently than Redis. Henosis balances this trade-off through the cost model and migrates in both directions, from TileDB to Redis and from Redis to TileDB, in each iteration. The iterative refinement procedure reaches the final, optimize storage plan when no arrays have migrated on either direction during an iteration.

In each iteration Henosis moves a set of arrays from TileDB to Redis with the aim of reducing the TileDB chunk access frequency. Let $u_{i,j}$ be the set of arrays stored in TileDB chunk i that is accessed by process p_j. Moving all arrays in $u_{i,j}$ to Redis reduces the access frequency of TileDB chunk i at least by 1. If the storage plan before and after migration is $\langle S, L\rangle$ and $\langle S', L'\rangle$, the *benefit* of moving $u_{i,j}$ to Redis is defined as:

$$benefit(u_{i,j}) = \frac{cost(S, L, W) - cost(S', L', W)}{|u_{i,j}|} \quad (5)$$

Henosis decides which arrays to migrate to Redis by computing the benefit score for the set $u_{i,j}$ for every TileDB chunk i and process p_j. A set $u_{i,j}$ will be migrated to Redis if (1) no other set has a higher benefit score, and (2) it fits in the Redis free space, and (3) its benefit score is positive. Henosis may not move any array to Redis in an iteration if no array satisfies all conditions.

For example, consider the initial storage plan shown in Figure 7b. Initially, all partitions are stored in TileDB. For partition A, $u_{A,1} = \{a_3, a_4\}$, $u_{A,2} = \{a_5, a_6\}$ and $u_{A,3} = \cdots = u_{A,7} = \{a_4, a_5\}$. For partition B, $u_{B,1} = \{a_1, a_2\}$ and $u_{B,2} = \{a_6, a_7\}$. In the first iteration, the set $\{a_4, a_5\}$ will be migrated to Redis because it has the highest positive benefit score.

Henosis also migrates arrays from Redis to TileDB in each iteration. Henosis considers each Redis array a_j individually. For each a_j array, Henosis computes the reduction of the estimated cost if a_j is moved to TileDB chunk i. TileDB chunks that are full (that is, already contain C_f arrays) are ignored during this pass. Henosis migrates a_j to the TileDB chunk with the highest positive reduction. Henosis may not move any array to TileDB in an iteration if no migration results in a cost reduction.

5.3.3 Algorithm comparison. The Henosis optimization procedure jointly considers array consolidation and placement. Performing consolidation and placement in two independent steps is also a viable strategy that deserves additional consideration.

One strategy is to consolidate first to chunks and then optimize the chunk placement. This *consolidate-then-place* strategy may store arrays with low access frequencies in Redis. The consolidation algorithm only considers the co-access frequency between small arrays. Consequently, small arrays with low access frequencies may be stored in Redis if they are often co-accessed with small arrays which have high access frequencies. Assume there are 4 arrays $\{a_1, a_2, a_3, a_4\}$ and 8 processes. Let 3 processes read a_2, 3 processes read a_4, 1 process read $\{a_1, a_2\}$ and 1 process read $\{a_3, a_4\}$. Let $\{a_1, a_2\}$ be consolidated into one chunk and $\{a_3, a_4\}$ into a different chunk. Then, assume $\{a_1, a_2\}$ is placed in Redis. This means that array a_1 has been placed in Redis although a_4 is accessed more frequently. In contrast, the Henosis optimization algorithm places a_2 and a_4 in Redis that are accessed by 8 processes. 2 processes will access the $\{a_1, a_3\}$ chunk in TileDB.

Another strategy is optimizing placement first and then consolidating. This *place-then-consolidate* strategy spreads arrays which are frequently co-accessed across data stores, because co-access frequencies are not considered during data placement. Assume, again, that there are 4 arrays and 8 processes. Let 3 processes read $\{a_1, a_2\}$, 3 processes read $\{a_3, a_4\}$, 1 process read a_2 and 1 process read a_4. Placing first would store arrays a_2 and a_4 in Redis. Consolidation would then bring a_1 and a_3 into one TileDB chunk. This strategy breaks the frequently co-accessed array pairs $\{a_1, a_2\}$ and $\{a_3, a_4\}$ and results in 6 accesses to the TileDB chunk. In contrast, the Henosis optimization algorithm could place a_1 and a_2 in Redis, and require 4 accesses to the $\{a_3, a_4\}$ chunk in TileDB.

6 THE HENOSIS PROTOTYPE

6.1 Access Pattern Monitoring

Henosis first generates the *workload specification* based on the monitored access pattern of HDF5 applications. Henosis logs the set of arrays read by each of an HDF5 application. An array is uniquely identified by its path. A process in a large cluster is uniquely identified by its node identifier (hostname) and the process identifier (PID). Hence, for each access Henosis collects the node identifier, process identifier and the paths of the arrays that were requested by the process.

A process first registers the three drivers that were introduced in Section 4, namely the *I/O Interceptor*, the *Redis Driver* and the *TileDB Driver*. Array accesses are tracked by the *I/O interceptor* when calling the HDF5 functions `H5Fopen` and `H5Dopen` to open files and datasets, respectively, before reading. When the process is terminated, the three Henosis drivers are unregistered. When the *I/O interceptor* is unregistered, it flushes the access log to the parallel file system such as Lustre, if available, or else to HadoopFS.

Two straightforward ways to log workload specification information have drawbacks. One way is to append all data to a single log file. However, the log file becomes the point of contention when multiple processes concurrently write to the file. Another way is writing information about each process to a different file. Although there is no contention, managing many small files is inefficient as it entails substantial metadata operations. Henosis uses a hybrid strategy where a process logs to a temporary file and, periodically, a spooler process merges temporary log files and appends them to a permanent log. This hybrid strategy allows full I/O concurrency during the active phase and constructs a single log offline.

6.2 Array Storage

Henosis stores arrays in two data stores, TileDB and Redis. In Redis, an array is stored as a key/value item, where the key is the array path and the value is the array content. In TileDB, Henosis creates a merged array by placing multi-dimensional arrays sequentially along their first dimension, as shown in Figure 8. Let N_f be the maximum number of small arrays consolidated in a TileDB chunk.

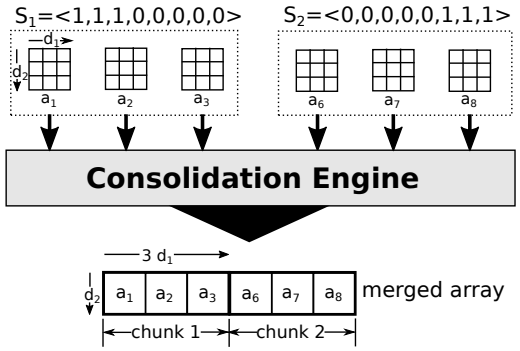


Figure 8: Consolidation in TileDB.

The length of the first dimension of the merged TileDB chunk is $N_f \times d_1$, where d_1 is the first-dimension length of the individual arrays. Let N_{chunks} be the number of rows in S which have at least one non-zero element. The length of the first dimension of the merged array is $N_{chunks} \times N_f \times d_1$. For each row in S, Henosis stores the arrays whose corresponding elements are non-zero in one chunk. Figure 8 shows how the example from Figure 6 would be stored in TileDB.

In addition to the raw data, Henosis stores additional metadata in Redis. Specifically, Henosis records where a small array is stored. Henosis creates a tuple *<small array path, data store, storage array path, offset>* for each small array. The *data store* indicates which data store the small array is located in. If an array is stored in Redis, the *storage array path* is the key of the corresponding key/value item, and the *offset* is 0. If an array is stored in TileDB, the *storage array path* is the merged array path and the *offset* is the first-dimension offset of the small array in the merged array. For example, the *offset* of array a_6 in Figure 8 is $3d_1$.

6.3 Read procedure

When a process reads an array through the HDF5 function `H5Dread`, Henosis intercepts the function and forwards the request to the corresponding data store. Henosis stores small arrays which have been loaded by the process in a cache. Henosis directly returns the requested small array if the array is in the cache. Otherwise, Henosis retrieves the metadata tuple of the small array and reads from the corresponding data store. When the array is in Redis, Henosis caches the loaded array. When the array in in TileDB, Henosis reads a TileDB chunk which contains the requested small array and caches all the arrays contained in the chunk. The workflow of the Henosis read procedure is shown in Figure 9. Assume process p_1 in Figure 6 requests the arrays a_1, a_2, a_3, a_4 sequentially and the arrays are consolidated as shown in Figure 8. Henosis first reads TileDB chunk 1, caches a_1, a_2 and a_3, and returns a_1. When a_2 and a_3 are then requested, Henosis directly returns arrays from the cache. When a_4 is requested, Henosis reads a_4 from Redis, stores the array in the cache and returns the array.

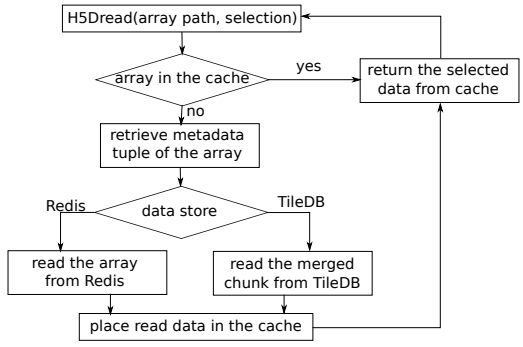


Figure 9: The Henosis read procedure.

7 EXPERIMENTAL EVALUATION

This section experimentally evaluates the optimization algorithm in Henosis. We perform the experiments in a shared cluster where each node is equipped with a 14-core 2.4 GHz Intel Xeon E5-2680 CPU, 0.5 TB DRAM, and three 2 TB HDDs. We use nine nodes in our evaluation, unless otherwise noted. The two underlying data stores are TileDB 1.1.0 and Redis 5.0.3. TileDB has been configured to store data in the Hadoop distributed file system (HadoopFS). Our cluster deploys Hadoop 2.8.3. The replication factor for both HadoopFS and Redis has been fixed to three. We repeat each experiment ten times and report the average of the measured values. We also report the standard deviation as error bars when it is distinguishable. We consider the following questions:

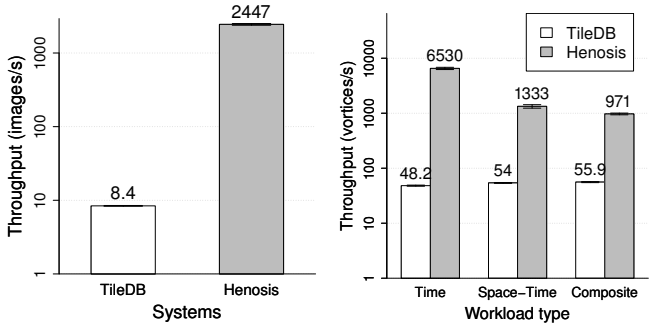
(1) What is the I/O improvement from consolidation over directly reading small arrays for real scientific pipelines? Does generating the initial storage plan by partitioning a query-weighted graph outperform other partitioning methods?
(2) Does the iterative refinement method improve the initial storage plan? What is the improvement compared to performing consolidation and placement independently?

7.1 Datasets

All experiments use the two I/O-bound scientific application drivers, transient detection and vortices prediction, that were described in Section 3. Both pipelines analyze a large number of small arrays (astronomy images and vortices, respectively). Henosis stores small arrays in TileDB on HadoopFS and Redis.

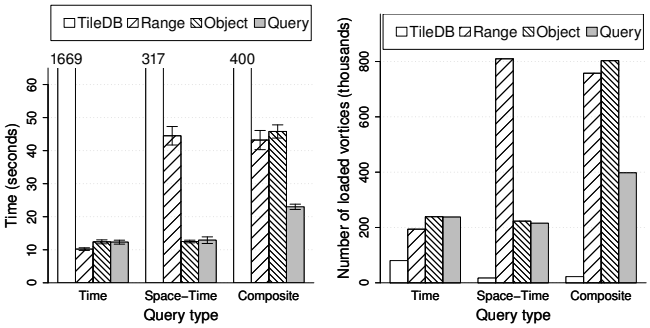
There are 11,889 astronomy images in the transient detection pipeline. Each image contains 5 arrays, and the size of each array is 21×21. The analytical operation is classification using a convolutional neural network (CNN) in TensorFlow. Each image is analyzed independently.

The vortices prediction pipeline contains 164,599 vortices, which are identified by the DBSCAN algorithm in 2040 fluid flow snapshots. A vortex is represented as a small 2D array, with size 8.4 KB and 42×50 cells. Each vortex is represented by a timestamp and a bounding box showing its temporal and spatial locations respectively. Queries in the vortices prediction pipeline retrieve a subset of all vortices. There are three types of query workloads in this pipeline: (1) a *Time* workload consists of queries that read all vortices at a given timestamp chosen at random; (2) a *Space-Time* workload consists of queries that access all vortices that intersect with a spatiotemporal query box; (3) a *Composite* query workload is a mixture of the previous two workloads: 90% of the queries are Space-Time queries and 10% of the queries are Time queries.



(a) Transient detection (b) Vortices prediction

Figure 10: Throughput with and without consolidation in the two pipelines. Henosis, reading from merged TileDB array, is almost 300× faster than directly reading small arrays in TileDB.



(a) Time vs. query types (b) Data volume vs. query types

Figure 11: Time and data volume with different queries. The query-weighted graph is 3.5× faster than range partitioning, 2× faster than object-weighted graph, and 136× faster than TileDB.

7.2 Effectiveness of object consolidation

This section evaluates the impact of consolidation on performance without considering the effect of the iterative refinement technique. Thus, only the *query-weighted* graph partitioning algorithm (the initial storage plan) is evaluated in this section. In these experiments Henosis only consolidates small arrays and stores data exclusively in the TileDB data store.

We compare Henosis with native TileDB, which stores small arrays independently. The initial storage plan is also compared with the other two storage plans, generated by *object-weighted* graph partitioning and *range* partitioning. These two storage plans also store all the data in the TileDB data store. We use METIS [21] to partition the *query-weighted* and *object-weighted* graphs.

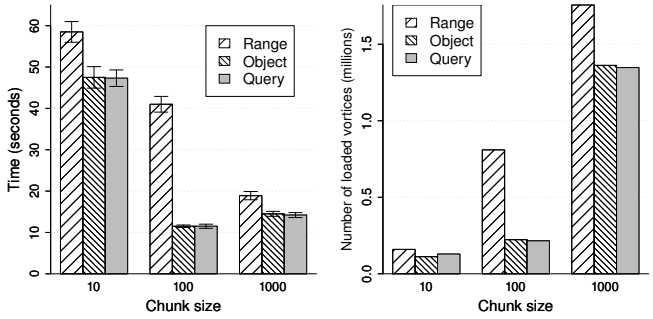
7.2.1 Comparison with native TileDB. We first compare Henosis with native TileDB in the transient detection pipeline. This experiment uses a single node. In Henosis, a chunk in the merged array contains 1024 images, or 5120 small arrays. Figure 10a shows the number of inferred images per second in the transient detection pipeline, reading from native TileDB and Henosis respectively. Henosis is almost 300 times faster than TileDB.

We now compare Henosis with native TileDB in the vortices prediction pipeline. In Henosis, 164,599 vortices are consolidated into 1646 chunks, hence each chunk contains roughly 100 vortices. Figure 10b shows the number of small arrays read per second when vortices are consolidated based on the initial storage plan. We compare Henosis with TileDB when users execute the three types of queries. Henosis improves the throughput by as much as 135× for *Time* queries, and by at least 17× for *Composite* queries. We conclude from the two experiments that Henosis, when consolidating small arrays only based on the initial storage plan, improves the performance by as much as 300× over directly reading small arrays from TileDB.

7.2.2 Comparison with alternative consolidation algorithms. Henosis constructs and partitions a *query-weighted* graph to create the initial storage plan. We compare the effectiveness of this algorithm in the vortices prediction pipeline with two baselines: *range* partitioning and *object-weighted* graph partitioning. Range partitioning consolidates on a single dimension. For this experiment, we consolidate vortices with the same timestamp into one chunk.

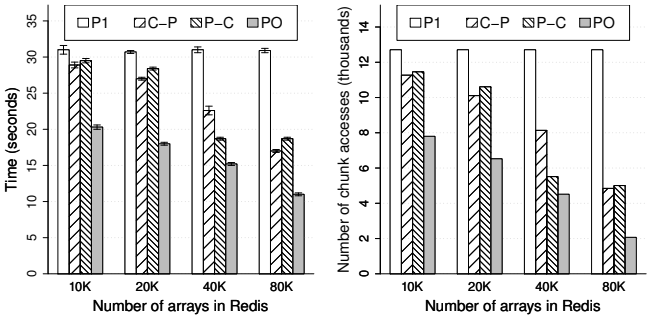
Figure 11a and 11b shows the impact of different query types. We report the read performance when evaluating 1000 queries of each query type. With Time queries, range partitioning performs best because the consolidation dimension is the query dimension. However, object-weighted graph partitioning and query-weighted graph partitioning (denoted as *Object* and *Query*, respectively) are only slightly slower than range partitioning. With Space-Time queries, range partitioning reads 3.6× more vortices than the two graph partitioning algorithms. This is because Space-Time queries also filter on spatial locations, while range partitioning only splits on the timestamp dimension and thus returns redundant data in each I/O operation. With the Composite query, the query-weighted graph outperforms the other two partitioning algorithms. Time queries on average access 80 vortices, which is much more than the 20 vortices that Space-Time queries roughly access. As a result, partitioning the object-weighted graph splits vortices mostly based on their timestamps, impeding the Space-Time type queries which are 90% of the Composite query mix. Native TileDB never reads redundant data for any query. However, all three partitioning algorithms are faster than reading small arrays in native TileDB. This suggests that picking a sub-optimal storage plan can still be orders of magnitude faster than reading from native TileDB.

7.2.3 Impact of chunk sizing. The size of the merged array is another factor impacting performance. We evaluate the three partitioning algorithms with different chunk sizes. We use the Space-Time queries and vary the number of vortices that are stored per chunk to 10, 100 and 1000. Figure 12a shows the query response time and Figure 12b plots the total number of vortices accessed. First, we observe that the range partitioning strategy reads more vortices and takes more time than the other two alternatives. Second, the number of loaded vortices decreases with smaller chunks. However, the chunk size is not the only factor impacting I/O performance. The chunk size 100 gives the lowest query response time but reads more vortices than chunk size 10 and fewer vortices than chunk size 1000. Selecting the optimal chunk size for an entire workload is a promising avenue for future work.



(a) Time vs. chunk size (b) I/O volume vs. chunk size

Figure 12: Time and I/O volume for different chunk sizes with Space-Time queries in the vortices prediction pipeline. Query-weighted graph partitioning is over 3.5× faster than range partitioning, and statistically indistinguishable from object-weighted graph partitioning.



(a) Time vs. Redis capacity (b) TileDB accesses vs. Redis capacity

Figure 13: Time and number of TileDB accesses with different Redis capacity limits. Reading from the optimized storage plan (PO) is 2.8× faster than the initial storage plan (P1), and 1.7× faster than the consolidate-then-place and place-then-consolidate strategies (C-P and P-C, respectively).

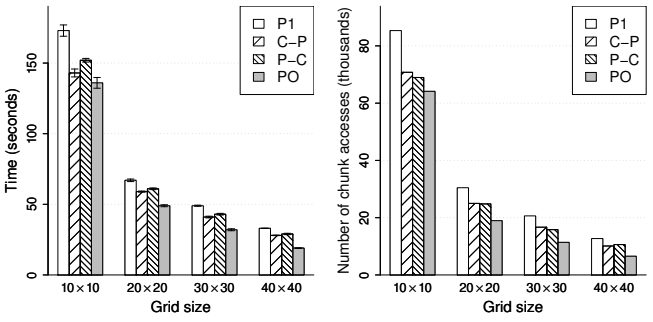
7.3 Effectiveness of placement optimization

This section evaluates the effectiveness of the iterative refinement algorithm. We first compare with performing consolidation and placement independently. Specifically, we compare with the *consolidate-then-place* and *place-then-consolidate* strategies which perform consolidation first and placement first, respectively. We also evaluate the optimization effectiveness by comparing the final storage plan after iterative refinement with the initial storage plan from *query-weighted* graph partitioning. In the following experiments, **P1** denotes the initial storage plan from the query-weighted graph partitioning algorithm, **C-P** denotes consolidation-then-placement, **P-C** denotes placement-then-consolidation, and **PO** denotes the optimized storage plan from the iterative refinement algorithm.

We evaluate the data placement optimization on the vortices prediction pipeline. Space-Time queries are created on a grid. Each query is a box with dimension lengths l_t, l_x and l_y corresponding to the time, x, and y dimensions respectively. A Space-Time query is created by picking a random grid cell and extending the grid cell in all directions by l_t to accommodate vortex movement between timestamps. For example, if grid cells are 40×40 big and $l_t = 3$, a space-time query will have the same center as a randomly chosen grid cell, but $l_x = l_y = 46$.

7.3.1 Impact of limited memory. This experiment varies the Redis capacity limit C_{keys} which controls the number of objects that can be stored in Redis. The grid size is 40×40 and Henosis consolidates 45 arrays into a TileDB chunk. In this experiment, we limit Redis to 10K, 20K, 40K and 80K small arrays. (We stop at 80K because the vortices prediction pipeline has 165K small arrays in total and we desire Redis to store no more than 50% of the dataset.)

Figure 13a shows the time it takes to complete the query workload. Storing arrays in Redis improves I/O performance. Reading from the optimized storage plan after iterative refinement (PO) is at most 2.8× faster than the initial storage plan (P1). The optimized storage plan is also up to 1.7× faster than the consolidate-then-place (C-P) and the place-then-consolidate (P-C) strategies, even with very limited memory for Redis (10K objects). To understand where the speedup comes from, Figure 13b shows the number of accesses to TileDB. Reading from the optimized storage plan PO has fewer accesses than the other strategies: PO has up to 6.2× fewer accesses than P1 and up to 2.4× fewer accesses than performing consolidation and placement independently (P-C and C-P).



(a) Time vs. grid size (b) TileDB accesses vs. grid size

Figure 14: Time and number of TileDB accesses with different grid sizes. The final, optimized storage plan (PO) is up to 1.75× faster than the initial storage plan (P1), and up to 1.5× faster than the consolidate-then-place (C-P) and the place-then-consolidate (P-C) strategies.

7.3.2 Impact of query size. In this experiment, we limit Redis storage to 20K vortices and vary the size of the uniform grid which is used to build queries. The grid cell length in spatial dimensions ranges from 10×10 to 40×40. As a result, the number of vortices read in a query increases as the cell size increases. Figure 14a shows the query response time with the four storage plans. Reading from the optimized storage plan (PO) outperforms other storage plans. Compared with the initial storage plan (P1), the optimized storage plan PO improves the read performance by up to 1.75× when the grid cell size is 40×40. Reading from the optimized storage plan is 1.5× faster than the consolidate-then-place (C-P) and the place-then-consolidate (P-C) strategies. Figure 14b shows the number of accesses to TileDB, which aligns with the query response time observations.

7.3.3 Runtime overhead of optimization. Although the optimization is performed once and is amortized over multiple queries, it is fair to ask how long does the optimization procedure take. The optimization time is proportional to the Redis capacity and inversely proportional to the query size. For the experiments shown in Figure 13, optimization takes 36 seconds when Redis stores up to 10K vortices, and 342 seconds when Redis stores 80K vortices. The time decreases from 109 seconds to 68 seconds when the grid size increases from 10×10 to 40×40 in Figure 14. It is important to note that the current prototype performs optimization serially. Parallelizing the optimization process would reduce its runtime overhead, and is an interesting direction for future work.

8 RELATED WORK

This section presents related work in distributed array management systems, object consolidation and data placement.

Distributed array management. Increasing data volumes pose a big challenge for scientific data management. Many scientific pipelines usually analyze arrays stored in file formats such as HDF5 and netCDF in high-performance computers with a shared parallel file system. However, shared-nothing commodity clusters are getting more popular. Several array database systems have been developed to facilitate large array management in shared-nothing clusters. ArrayStore [34] and SciDB [10] are distributed systems, storing arrays across the local file system of multiple nodes. Other systems, such as TileDB [28], take advantage of a distributed file system to enable large scale array storage.

Prior research has identified drawbacks when using these systems in existing scientific pipelines. First, these systems are inefficient in managing small arrays as they issue a large number of small I/O requests. Speculative loading [4, 14] speeds up analysis by overlapping the I/O time and the compute time. Moreover, existing systems either store data in a single file system or request users to explicitly manage the placement of arrays in the storage hierarchy. Manual data placement is onerous for users and often underutilizes heterogeneous clusters. In addition, these systems cannot be seamlessly integrated with existing scientific pipelines. In situ array processing [6, 22, 40] allows applications to process array data without loading, but does not optimize for manipulating small arrays in heterogeneous data stores.

Small object consolidation. Recent work aims to accelerate I/O to small objects. A common solution is consolidating multiple small objects into larger ones. Systems and file formats such as HAR [18], MapFile [39], Haystack [5] and Ambry [26] consolidate small objects into few files. However, the consolidation in these systems does not exploit access correlations between small objects.

Henosis consolidates small arrays which are frequently accessed together into a chunk to eliminate small I/O requests. Data management research has utilized similar strategies to enhance query performance by data partitioning for OLTP [29] or OLAP [27, 33] workloads. Many advanced consolidation techniques [2, 32, 42] assume knowledge of the database schema and offer limited support for complex access patterns. Schema-free partitioning algorithms [1, 3, 15, 35] partition a database based on workload monitoring. These solutions observe the similarity between tuples directly from the issued queries and build partitions according to the co-access frequency of tuples. Another example of a system that mines correlation patterns in I/O activity to optimize I/O performance with consolidation is Pacaca [20]. The consolidation algorithm in Henosis is based on graph partitioning but is aware of queries with different sizes. While Henosis partitions at a single granularity, prior work has investigated more complex partitioning methods that shard at multiple granularities [17, 37].

Data placement. Intelligent data placement in heterogeneous storage systems has been investigated for years as a mechanism to optimize I/O performance. The key idea is to automatically place data items based on the data access pattern. Some works [12, 24, 31] dynamically tune the data placement at runtime by placing the hot data in and evicting cold data out of the fast data store. Other work models the data placement problem as a static optimization problem. CAST [13] formulates data placement as a non-linear optimization problem that maximizes the tenant utility for heterogeneous cloud storage services. ProfDP [38] calculates a *moving factor* for each data object based on its latency sensitivity and bandwidth sensitivity, and determines data placement by ranking all objects based on their moving factor score. Canim [11] models data placement between SSD and HDD devices as the 0-1 knapsack problem and investigates dynamic programming and greedy techniques to obtain good placement strategies. These prior solutions are agnostic to the opportunity to consolidate objects into larger groups. Systems like SPAR [30] and related efforts [17, 23] rely on payload-aware data aggregation and placement strategies (in some cases with statistical guarantees [37]). Henosis entails a much simpler design that is agnostic of payload characteristics.

9 CONCLUSION

This paper highlights the problem of accessing small arrays in scientific data analysis workloads. Although scientific file formats such as HDF5 can consolidate small objects in one big file, their monolithic design is oblivious to the heterogeneous I/O capabilities of modern clusters. This paper introduces Henosis, an I/O library that allows HDF5-based programs to consolidate, place and read small arrays on different storage backends, specifically TileDB and Redis. Henosis accelerates I/O by performing consolidation and placement optimization simultaneously. We evaluate Henosis in two real-world scientific pipelines. Henosis speeds up the I/O performance by 300× compared with directly reading small arrays from TileDB. By simultaneously performing consolidation and placement, Henosis produces a storage plan that is 1.7× faster than workload-oblivious strategies which perform consolidation and placement independently.

Henosis does not handle highly dynamic workloads where the historical access pattern is not reflective of future access patterns. Henosis also cannot efficiently grow existing arrays to append new data. As part of ongoing work, we will enhance Henosis to alleviate these issues.

ACKNOWLEDGMENTS

This material is based upon work supported in part by the National Science Foundation under grants SMA-1738502, CCF-1816577, CCF-1747447, CCF-1629548 and Office of Naval Research grant #N00014-17-1-2584.

REFERENCES

[1] Sanjay Agrawal, Eric Chu, and Vivek Narasayya. 2006. Automatic Physical Design Tuning: Workload As a Sequence. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD '06)*. ACM, New York, NY, USA, 683–694.

[2] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. 2004. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 359–370.

[3] Ahmed M. Aly, Ahmed R. Mahmood, Mohamed S. Hassan, Walid G. Aref, Mourad Ouzzani, Hazem Elmeleegy, and Thamir Qadah. 2015. AQWA: Adaptive Query Workload Aware Partitioning of Big Spatial Data. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 2062–2073.

[4] Leilani Battle, Remco Chang, and Michael Stonebraker. 2016. Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 1363–1375.

[5] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 47–60. http://dl.acm.org/citation.cfm?id=1924943.1924947

[6] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. 2014. Parallel Data Analysis Directly on Scientific File Formats. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 385–396.

[7] Michael R Blanton, Matthew A Bershady, Bela Abolfathi, Franco D Albareti, Carlos Allende Prieto, Andres Almeida, Javier Alonso-García, Friedrich Anders, Scott F Anderson, Brett Andrews, et al. 2017. Sloan digital sky survey IV: Mapping the Milky Way, nearby galaxies, and the distant universe. *The Astronomical Journal* 154, 1 (2017), 28.

[8] James K Bonfield and Rodger Staden. 2002. ZTR: a new format for DNA sequence trace data. *Bioinformatics* 18, 1 (2002), 3–10.

[9] Kevin J Bowers, BJ Albright, L Yin, B Bergen, and TJT Kwan. 2008. Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation. *Physics of Plasmas* 15, 5 (2008), 055703.

[10] Paul G. Brown. 2010. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 963–968.

[11] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2009. An Object Placement Advisor for DB2 Using Solid State Storage. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1318–1329. https://doi.org/10.14778/1687553.1687557

[12] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 22–32. https://doi.org/10.1145/1995896.1995902

[13] Yue Cheng, M. Safdar Iqbal, Aayush Gupta, and Ali R. Butt. 2015. CAST: Tiering Storage for Data Analytics in the Cloud. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*. ACM, New York, NY, USA, 45–56. https://doi.org/10.1145/2749246.2749252

[14] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 1287–1298.

[15] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 48–57.

[16] Simon Driscoll, Alessio Bozzo, Lesley J Gray, Alan Robock, and Georgiy Stenchikov. 2012. Coupled Model Intercomparison Project 5 (CMIP5) simulations of climate following volcanic eruptions. *Journal of Geophysical Research: Atmospheres* 117, D17 (2012).

[17] Quang Duong, Sharad Goel, Jake Hofman, and Sergei Vassilvitskii. 2013. Sharding Social Networks. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining (WSDM '13)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/2433396.2433424

[18] Apache Hadoop. 2013 (accessed May 23, 2019). *Hadoop Archives Guide*. https://hadoop.apache.org/docs/r1.2.1/hadoop_archives.html.

[19] TW-S Holoien, JS Brown, KZ Stanek, CS Kochanek, BJ Shappee, JL Prieto, Subo Dong, J Brimacombe, DW Bishop, S Bose, et al. 2017. The ASAS-SN bright supernova catalogue–III. 2016. *Monthly Notices of the Royal Astronomical Society* 471, 4 (2017), 4966–4981.

[20] Binbing Hou and Feng Chen. 2018. Pacaca: Mining Object Correlations and Parallelism for Enhancing User Experience with Cloud Storage. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 293–305.

[21] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[22] Kalyan Khandrika. 2018. *ASHWHIN: Array Storage system on HadoopFS With HDF5 Interface*. Master's thesis. The Ohio State University.

[23] Feilong Liu, Ario Salmasi, Spyros Blanas, and Anastasios Sidiropoulos. 2018. Chasing Similarity: Distribution-aware Aggregation Scheduling. *PVLDB* 12, 3 (2018), 292–306. http://www.vldb.org/pvldb/vol12/p292-liu.pdf

[24] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, Article 26, 10 pages. https://doi.org/10.1145/3079079.3079089

[25] Tiago Macedo and Fred Oliveira. 2011. *Redis Cookbook: Practical Techniques for Fast Data Manipulation*. O'Reilly Media, Inc.

[26] Shadi A. Noghabi, Sriram Subramanian, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Indranil Gupta, and Roy H. Campbell. 2016. Ambry: LinkedIn's Scalable Geo-Distributed Object Store. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 253–265.

[27] S. Papadomanolakis and A. Ailamaki. 2004. AutoPart: automating schema design for large scientific databases using data partitioning. In *Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004.* 383–392.

[28] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. 2016. The TileDB Array Data Storage Manager. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 349–360.

[29] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 61–72.

[30] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. 2010. The Little Engine(s) That Could: Scaling Online Social Networks. In *Proceedings of the ACM SIGCOMM 2010 Conference (SIGCOMM '10)*. ACM, New York, NY, USA, 375–386. https://doi.org/10.1145/1851182.1851227

[31] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 85–95. https://doi.org/10.1145/1995896.1995911

[32] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. Automating Physical Database Design in a Parallel Database. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. ACM, New York, NY, USA, 558–569.

[33] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. 1998. Data Partitioning and Load Balancing in Parallel Disk Systems. *The VLDB Journal* 7, 1 (Feb. 1998), 48–66.

[34] Emad Soroush, Magdalena Balazinska, and Daniel Wang. 2011. ArrayStore: A Storage Manager for Complex Parallel Array Processing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 253–264.

[35] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained Partitioning for Aggressive Data Skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 1115–1126.

[36] S. Unnikrishnan and Datta V. Gaitonde. 2016. A high-fidelity method to analyze perturbation evolution in turbulent flows. *J. Comput. Phys.* 310 (2016), 45 – 62.

[37] Y. Wang, S. Parthasarathy, and P. Sadayappan. 2013. Stratification driven placement of complex data: A framework for distributed data analytics. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 709–720. https://doi.org/10.1109/ICDE.2013.6544868

[38] Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. 2018. ProfDP: A Lightweight Profiler to Guide Data Placement in Heterogeneous Memory Systems. In *Proceedings of the 2018 International Conference on Supercomputing (ICS '18)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/3205289.3205320

[39] Tom White. 2012. *Hadoop: The definitive guide*. O'Reilly Media, Inc.

[40] H. Xing, S. Floratos, S. Blanas, S. Byna, Prabhat, K. Wu, and P. Brown. 2018. ArrayBridge: Interweaving declarative array processing in SciDB with imperative HDF5-based programs. In *2018 IEEE 34th International Conference on Data Engineering*. 1–12.

[41] Hui Yang, Srinivasan Parthasarathy, and Sameep Mehta. 2005. A Generalized Framework for Mining Spatio-temporal Patterns in Scientific Data. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD '05)*. ACM, New York, NY, USA, 716–721.

[42] Jingren Zhou, Nicolas Bruno, and Wei Lin. 2012. Advanced Partitioning Techniques for Massively Distributed Computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 13–24.