ArrayMorph: Optimizing Hyperslab Queries on the Cloud for Machine Learning Pipelines

Ruochen Jiang
jiang.2091@osu.edu
The Ohio State University
Columbus, OH, USA

Spyros Blanas
blanas.2@osu.edu
The Ohio State University
Columbus, OH, USA

ABSTRACT

Cloud storage services such as Amazon S3, Azure Blob Storage, and Google Cloud Storage are widely used to store raw data for machine learning applications. When the data is later processed, the analysis predominantly focuses on regions of interest (such as a small bounding box in a larger image) and discards uninteresting regions. Machine learning applications can significantly accelerate their I/O if they push this data filtering step to the cloud. Prior work has proposed different methods to partially read array (tensor) objects, such as chunking, reading a contiguous byte range, and evaluating a lambda function. No method is optimal; estimating the total time and cost of a data retrieval requires an understanding of the data serialization order, the chunk size and platform-specific properties. This paper introduces ArrayMorph, a cloud-based array data storage system that automatically determines which is the best method to use to retrieve regions of interest from data on the cloud. ArrayMorph formulates data accesses as hyperslab queries, and optimizes them using a multi-phase cost-based approach. ArrayMorph seamlessly integrates with Python/PyTorch-based ML applications, and is experimentally shown to transfer up to 9.8X less data than existing systems. This makes ML applications run up to 1.7X faster and 9X cheaper than prior solutions.

PVLDB Reference Format:
Ruochen Jiang and Spyros Blanas. ArrayMorph: Optimizing Hyperslab Queries on the Cloud for Machine Learning Pipelines. PVLDB, 18(9): 3189 - 3202, 2025.
doi:10.14778/3746405.3746437

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ruochenj123/ArrayMorph.

1 INTRODUCTION

Large multi-dimensional array (tensor) datasets are ubiquitous in machine learning applications like image recognition [20], natural language processing [12] and video analytics [16, 26, 39, 41]. In these applications, data is often collected at storage-constrained edge devices, is uploaded to cloud object stores such as Amazon S3 or Azure Blob Storage, and is analyzed further with machine learning frameworks such as PyTorch. The first step in many machine learning pipelines is accessing data from regions of interest, such as retrieving data between two points in time from a sequence of observations, or pixels within a bounding box from an image, or particular frames from a video. We refer to accesses that retrieve data from regions of interest as *hyperslab queries*.

A naive way to obtain the region of interest is to retrieve the entire object to the client in a single request, and throw away unneeded data in the client. (Surprisingly, this naive approach to data filtering is extremely common in machine learning applications!) Prior work has explored more efficient methods to filter data in the cloud that do not retrieve the entire object. Array-based systems split each object into *chunks* and retrieve all chunks that overlap the region of interest using separate requests. TileDB [38] and HSDS [17] follow this approach. HSLAMBDA [51] filters array data in a lambda function and returns the result to the client. Another option is to convert the data to a tabular representation and use services such as S3 Select to perform filter pushdown using SQL, as implemented in PushdownDB [56] and FlexPushdownDB [55].

No data retrieval method is optimal. Retrieving the entire object in one request minimizes the request cost, but transfers redundant data and hence incurs high data transfer time and charges. Retrieving the overlapping chunks requires issuing multiple retrieval requests, which greatly increases the request cost. Invoking a lambda function has a high initialization cost which is charged as CPU cycles. Using S3 Select requires expensive schema conversion (from multi-dimensional to relational) and data format conversion (from dense array to Parquet).

This paper introduces ArrayMorph, a cloud-based array data storage system that addresses the challenge of efficiently retrieving hyperslab-shaped regions of data, which are common in scientific and ML workflows, from cloud object stores. ArrayMorph minimizes response time and cost through adaptive query planning: ArrayMorph formulates data accesses as queries, considers multiple possible access methods including lambdas, and uses multi-phase, cost-based optimization that is inspired by access method selection in relational database systems. ArrayMorph supports queries against multiple cloud vendors, specifically Amazon S3, Azure Blob Storage, and Google Cloud Storage. ArrayMorph seamlessly integrates with Python/PyTorch-based ML applications through the h5py interface that returns a dense NumPy array / PyTorch tensor. Under the covers, ArrayMorph is implemented as a dynamically-loaded HDF5 VOL plugin, which allows a drop-in binary install without recompiling any C++-based HDF5 applications. The main contributions of this paper are as follows:

(1) We explore the limitations of existing systems for processing hyperslab queries. They use a single method to retrieve data, which is not always optimal in either time or cost.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746437

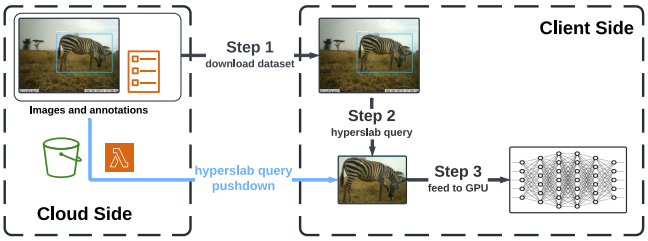


Figure 1: Species classification in the CameraTrap pipeline.

(2) We formulate data retrievals as a cost-based optimization problem and develop an optimization heuristic that chooses better access methods than the state of the art.
(3) We design and implement a cloud-based array data storage system, ArrayMorph, using HDF5 VOL features. This makes integration with Python/PyTorch-based ML applications easy.
(4) We evaluate ArrayMorph on synthetic and real-world workloads. The evaluation shows ArrayMorph transfers up to 9.8X less data than prior solutions. This makes ML applications run up to 1.7X faster and 9X cheaper.

The paper is structured as follows. Section 2 introduces how an ML application today can access data in cloud object stores. Section 3 shows the architecture of ArrayMorph. Section 4 formulates the hyperslab query processing optimization problem. Section 5 experimentally evaluates ArrayMorph. Section 6 presents related work and Section 7 concludes.

2 BACKGROUND AND MOTIVATION

2.1 Application example

Camera traps are cameras that are triggered by motion. They are used for wildlife monitoring (such as estimating the population size of endangered species) and ecological research (such as studying the impact of climate change on bird migration patterns) [54]. One large research project [30, 41] performs species classification as follows: The on-field team places camera traps, collects raw data, discards unusable images, and uploads to the cloud. A data post-processing team performs object detection using MegaDetector [31], and stores pixel coordinates of objects of interest (bounding boxes) in JSON format [9].

The data inside the bounding box is then processed using a custom species classification model as shown in Figure 1: The PyTorch script first downloads the requested images and the associated annotations from the cloud into memory (step 1). The bounding box is used to crop the original image (step 2), before transmitting the region of interest (a tensor) to the species classifier running on the GPU (step 3). In this pipeline, the data download step commonly results in significant I/O waste, as only a small subset of the raw image (i.e., the cropped part) is used. Instead of downloading the entire image, data retrieval can be accelerated by issuing a *hyperslab query* to the cloud storage service to directly retrieve the needed data. Modern cloud storage platforms support various methods to access only a part of an object, as we describe next.

2.2 Data retrieval methods

ArrayMorph considers three methods for processing hyperslab queries. All methods are supported by leading cloud providers, including Amazon Web Services, Microsoft Azure, and Google Cloud. Table 1 summarizes the trade-offs associated with each.

Get is the most commonly used data retrieval method. A Get request accepts a bucket and an object (key) name, and returns the entire payload (value) associated with this object. Cloud providers charge a per-request fee and a data transfer (egress) fee for each byte transferred out of the storage service. The data transfer fee dominates the total cost with Get requests.

Range retrieval is another data access method. A Range request is a Get request that only retrieves a specific byte range from the payload. To issue a Range request, users specify a byte range (expressed as a start and end offset) and add it to the HTTP header of the Get request. Range retrievals do not transfer data outside the retrieval range. The request fee of a Range request is the same as for a Get request, but returning less data reduces latency and data transfer (egress) costs. The limitation of the Range method is that it only fetches a *contiguous* region. When the requested data is not stored contiguously, one has to either issue multiple requests (which means higher service charges), or expand the byte range to span all regions of interest (which increases the data transfer cost).

Lambda functions are the most flexible data retrieval method, as they allow the execution of arbitrary code to process data on the cloud before returning to the client. Users invoke a lambda function through an HTTP request, and get charged for the compute cost (running time and memory usage) used to run the function, in addition to an egress fee charged for each byte transferred out. Compared to the Get and Range methods, Lambda incurs the lowest data transfer fee but introduces more server-side overhead: the processing time of a lambda function includes the actual execution time plus a function initialization time. For low-selectivity queries, the lambda execution fee is the major contributor to the total cost.

Object Lambda functions are a special form of lambda functions. Instead of invoking a lambda through an HTTP request, some providers (notably Amazon S3) can invoke a lambda function automatically when a Get request completes. Lambda functions that are invoked from Get requests are referred to as Object Lambdas. Object Lambdas intercept the output of the Get request and can return arbitrary data back to the requester.

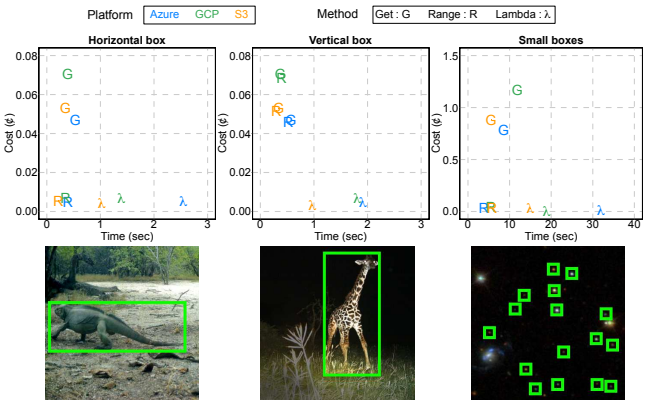
Performance and cost profile. Different data retrieval methods have significantly different performance and cost. We demonstrate these differences on a dataset of 2D arrays (images) with dimensions of 4096x4096 cells (pixels), that is chunked in 1MB square chunks. (The size is similar to the image data used for species classification in Section 2.1.) We evaluate the time and cost for processing three representative hyperslab queries: (a) Horizontal box query, which

Method	Invocation overhead	Service charge	Egress fees
Get	Negligible	Request fee, single	High
Range	Negligible	Request fee, multiple	Moderate
Lambda	High	Compute cost	Low

Table 1: Cloud data retrieval methods.

	HSDS [17]	HSLAMBDA [51]	TileDB [38]	PushDownDB [55, 56]	ArrayMorph
Supported platforms	S3, Azure	S3	S3, Azure, GCS	S3	S3, Azure, GCS
Data model	Array-based	Array-based	Array-based	Relational	Array-based
Get	✓	–	–	–	✓
Range	–	–	✓	–	✓
Lambda	–	✓	–	–	✓
Object Lambda	–	–	–	–	✓
S3 Select	–	–	–	✓	–

Table 2: Design choices of prior work.



(a) Horizontal box **(b) Vertical box** **(c) Small boxes**

Figure 2: Performance and cost profile of different retrieval methods across cloud platforms.

reads contiguous rows of data chosen at random, (b) Vertical box query, which reads contiguous columns chosen at random, and (c) Small box queries, which retrieves random 21x21 small regions. (Workload details are presented in Section 5.1.) We ran experiments on three cloud platforms and used different data retrieval methods.

Figure 2 shows the results and the motivating application patterns from species classification [30, 41] and supernovae detection [21, 27], respectively. Get (*G*) is not the best choice, as it reads redundant data which impacts performance and cost. Range (*R*) always performs better than Get. Lambda is a low-cost data retrieval method, as data transfers to a lambda are not charged, but the lambda initialization and processing time is significantly higher than the other methods. No single method is always optimal, as the performance and cost us influenced by the query shape and the cloud platform choice. To efficiently access data on the cloud, one should consider multiple data retrieval methods and their trade-offs.

2.3 Data retrieval methods in prior work

Table 2 summarizes the design choices of systems in prior work. Despite the fact that the performance and cost profiles of each data retrieval method are significantly different, prior work uses a single method to access data. HSDS [17] and HSLAMBDA [51] store each chunk as a single object. HSDS issues Get requests to each object and fetches the entire object to the client side for further processing. HSLAMBDA invokes an AWS lambda function and executes the query completely on the cloud. TileDB [38] stores the entire array as one object, with each logical chunk serialized as a contiguous block within the object. To process hyperslab queries, TileDB uses the Range method to fetch the accessed chunks. If the accessed chunks are stored contiguously, TileDB merges individual requests into a single large Range request.

Another approach is to use existing relational cloud management systems. PushdownDB [55, 56] uses the AWS S3 Select service to process SQL queries on the cloud. To process hyperslab-based queries using SQL, one needs to convert array data into a relational form by adding array indices as additional columns. This is known to be inefficient, because storing array indices has significant storage overhead [42]. Our experiments (see Section 5.2) corroborate that converting to a relational form can be up to 1,000X slower than directly using an array-based storage representation.

3 SYSTEM ARCHITECTURE

Figure 3 shows the architecture of ArrayMorph. It is implemented using the Virtual Object Layer (VOL) feature of HDF5 and serves as an I/O interceptor between the HDF5 library and cloud storage services. Users can seamlessly connect existing applications to the cloud by using ArrayMorph without code changes and recompilation, as ArrayMorph exposes the API of HDF5/h5py and is loaded as a dynamic plugin. Machine learning pipelines load data on demand through ArrayMorph's PyTorch DataLoader.

ArrayMorph consists of four components: The Micro-profiler estimates constant variables and tunes the cost model by benchmarking small workloads. The I/O Interceptor uses the HDF5 VOL API to intercept I/O requests from relevant HDF5/h5py calls (e.g. H5Dread) in client applications. It forwards these accesses to the

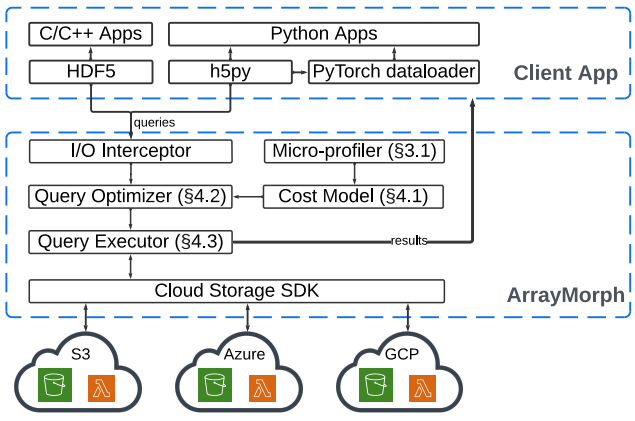


Figure 3: ArrayMorph system architecture.

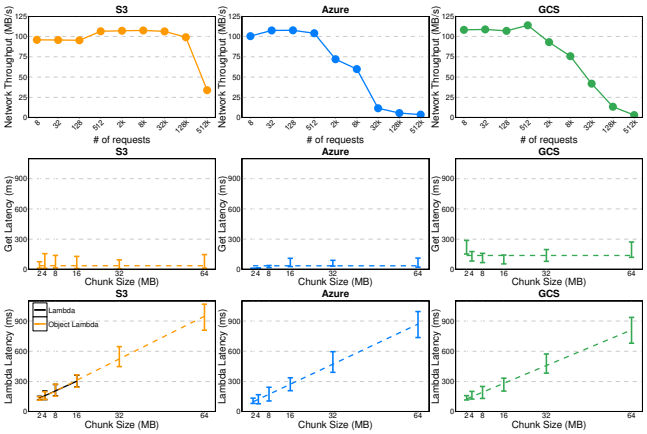


Figure 4: Micro-profiling results.

Query Optimizer, which selects an appropriate access method for each access. The Query Executor then issues the appropriate I/O requests to the cloud platforms.

ArrayMorph organizes data in the HDF5 hierarchical structure where each file can contain multiple datasets, and each dataset can contain multiple chunks. ArrayMorph adopts the standard HDF5 chunking mechanism, where each chunk in ArrayMorph is serialized in row order. Each chunk is stored as an object on cloud storage and the object is named according to its index position.

3.1 Micro-profiler

The Micro-profiler estimates the value of performance-sensitive constants in the cost models by profiling a simple synthetic workload. The Micro-profiler is configured once for each cloud provider, and does not need to be re-configured unless the network or compute capabilities change. Specifically, for each cloud provider $P \in \{S3, Azure, GCS\}$, the Micro-profiler estimates (1) the network bandwidth BW for different numbers of concurrent requests N, (2) the request concurrency range N_{min} to N_{max} where peak BW can be attained, and (3) the server-side processing time for regular requests T_R and for lambda functions T_λ.

Network bandwidth estimator. Estimating data transfer throughput is important to accurately determine the total data retrieval time. The bandwidth estimator first generates a randomly-filled 16GB, 4-byte integer array. The estimator then selects a level of concurrency N, and issues N Get requests of appropriate size to retrieve 16GB of data in total. (For example, $N = 256$ retrieves 64MB per request.) It then computes the average data transfer throughput for concurrency level N by dividing the data size (16GB) with the time to complete processing all N requests. Through these experiments, the Micro-profiler estimates the network bandwidth at points N_1, N_2, ... for cloud platform P. The function $BW(P, N)$ finds the nearest points N_i, N_j such that $N_i < N < N_j$ for which bandwidth measurements have been collected for cloud provider P, and performs linear interpolation.

Request bounds N_{min} and N_{max}. The top row of Figure 4 shows the results for three cloud platforms. We observe that the request rate significantly affects the network throughput. For Amazon S3, too few requests cannot fully utilize the network, while other cloud vendors fully utilize the network with as few as 8 requests. Too many requests lead to excessive overhead due to resource contention for all cloud vendors. In order to maintain high throughput for data transfers, for each cloud platform P, the Micro-Profiler computes two variables N^P_{min} and N^P_{max} which represent the minimum and maximum request rates that can sustain peak throughput for cloud platform P. N^P_{min} is the smallest N_i such that $BW(P, N_i)$ reaches at least 90% of the maximum observed bandwidth. Similarly, N^P_{max} is the largest N_i such that $BW(P, N_i)$ reaches at least 90% of the maximum observed bandwidth.

Server-side processing time T_λ and T_R. ArrayMorph considers whether to use lambda functions or issue regular data retrieval requests to process queries. Estimating the server-side processing time of either is crucial to making a good choice. To do this, the micro-profiler picks different request sizes and generates 32 data retrieval requests on a single chunk. It observes the server-side execution time for processing requests as lambdas (T_λ) or as regular retrievals (T_R). (The Google Cloud platform reports server-side processing time only for lambda requests, so we approximate server-side processing time of regular requests as the total request execution time minus the data transfer time.) The two bottom rows of Figure 4 show the results, where the whiskers are the minimum and the maximum time for different request sizes.

The first observation is that the lambda processing time (Fig. 4, bottom row) linearly correlates with the request size, which is attributed to the time to transfer the object from storage to the function. In contrast, the processing time for regular requests (Fig. 4, middle row) is not sensitive to request size. Therefore, the Micro-profiler estimates the processing time of regular requests (T_R) as a fixed cost, and the processing time of lambda functions (T_λ) through linear regression. Specifically, the micro-profiler produces parameters α_P, β_P and L_P for each cloud provider P such that:

$$T_R(P) = L_P$$
$$T_\lambda(P, \text{Object Size}) = \alpha_P \times \text{Object Size} + \beta_P$$

The second observation is that HTTP-triggered lambda functions fail in AWS when chunk sizes are larger than 16MB, due to restrictions on the response payload size. S3 Object Lambda has the same performance but can handle significantly larger chunk sizes. In terms of cost, Object Lambda follows the pricing policy of HTTP-triggered lambda, with a negligible extra cost ($0.4 per million requests). Based on these observations, ArrayMorph uses Object Lambdas for Amazon S3, and HTTP-triggered lambda functions for other cloud platforms.

4 QUERY PROCESSING

In this section, we describe how the Query Optimizer chooses how to access data in each chunk to process hyperslab queries. For a d-dimensional array, a hyperslab region is a continuous d-dimensional range of cells. A range s_i:e_i defines the starting s_i and ending e_i index values along each dimension i, with $s_i \leq e_i$. A hyperslab query Q is the d-dimensional selection range [s_1:e_1, ..., s_d:e_d]. Figure 5 colors the hyperslab query Q_1=[3:4, 1:2] in grey.

		Access method		
Meaning	**Symbol**	Get	Range $(r_1, r_2, ..., r_n)$	Lambda (Q)
Number of total requests	N_R	1	n	1
Number of lambda requests	N_λ	0	0	1
Transferred data (bytes)	S_T	S_C	$\sum_{i=1}^{n}(r_i.e - r_i.s + 1)$	$\prod_{i=1}^{d}(Q.e_i - Q.s_i + 1)$
Cloud service fee ($)	F	$F_R(\text{Get}) + F_T \cdot S_T$	$F_R(\text{Range}) \cdot N_R + F_T \cdot S_T$	$F_R(\text{Lambda}) + F_T \cdot S_T + F_\lambda \cdot M_\lambda \cdot T_\lambda(P, S_C)$

Table 3: Fee estimation for accessing one chunk with different access methods.

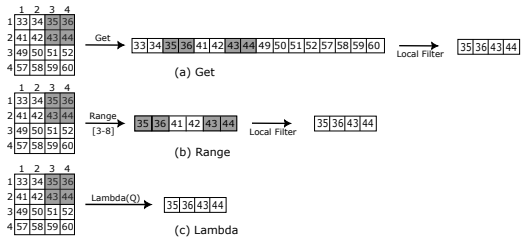


Figure 5: Different ways to process a hyperslab query.

4.1 Cost estimation

In the cloud, there is often a trade-off between the response time $T(Q)$ of a hyperslab query Q and the service fee $F(Q)$ to process it. To balance the trade-off between response time and monetary cost, ArrayMorph combines these two metrics by introducing a user-defined parameter ϕ, i.e.

$$Cost(Q) = T(Q) + \phi \times F(Q)$$

where $T(Q)$ represents the response time and $F(Q)$ represents the cloud service fee paid for processing query Q. In other words, ϕ conveys price sensitivity, or how many seconds may a query be delayed to save one dollar (if such an opportunity arises). Applications can always express a strict preference for time or cost: setting $\phi = 0$ minimizes time; setting $\phi = +\infty$ minimizes cost.

Fee estimation model. Let us first assume that a hyperslab query Q only accesses one chunk C of the array. That is, assume the range $(Q.s_i, Q.e_i)$ indicated by Q for each dimension i is within the boundaries of C in all dimensions, or $Q \cap C = Q$. Let S_C be the size of chunk C in bytes. The role of this model is to estimate the total service fee $F(Q)$ for different access methods by calculating the number of requests N_R and the transferred data size S_T for each method. Table 3 summarizes the estimations for each method.

Symbol	Meaning
$F_R(i)$	Request fee, where $i \in$ {Get, Range, Lambda}
F_T	Data transfer (egress) fee per byte
M_λ	Memory (GB) used in lambda function execution
F_λ	Lambda fee per GB-second for lambda functions
F	Total service fee
N_R	Number of requests in total
N_λ	Number of lambda requests
S_T	Data transferred (bytes)
S_C	Chunk size (bytes)

Table 4: Notation for cost estimation and query processing.

Get reads the entire chunk in one request ($N_R = 1$). The transferred data size is the chunk size ($S_T = S_C$). The service fee F is the data transfer fee plus the request fee.

Range accepts ranges $r_1, r_2, ..., r_n$ for retrieval. Each range r_i indicates a starting byte offset $r_i.s$ and an ending byte offset $r_i.e$. Each range is retrieved as a separate request ($N_R = n$) where each request retrieves $r_i.e - r_i.s + 1$ bytes. The service fee F is the data transfer fee plus the request fee for all n requests.

Lambda passes the hyperslab query Q to the cloud function, which reads chunk C, filters unneeded data, and only returns the query result to the client. The data transfer size can be computed in advance from the hyperslab ranges $(Q.s_i, Q.e_i)$ for every dimension i. Lambda functions are charged a request fee and a data transfer fee, similar to regular requests. In addition, there is a lambda function execution fee F_λ which is proportional to the execution time T_λ for processing a chunk with size S_C (as estimated by the Micro-profiler in Sec. 3.1) and the requested memory $M_\lambda > S_C$ to run the function. When a hyperslab query Q accesses multiple chunks $C_1, C_2, ..., C_n$, the fee estimation model computes the cost per chunk C_i, and then adds the per-chunk costs together:

$$F(Q) = \sum F(Q \cap C_i)$$

Response time estimation model. Assume hyperslab query Q accesses chunks $C_1, C_2, ..., C_n$. The response time estimation model partitions Q into per-chunk queries $Q_i = Q \cap C_i$. If t represents the number of threads issuing concurrent requests, on the cloud platform P, the time $T(Q)$ is estimated as the sum of the data transfer time, request handling time and lambda execution time:

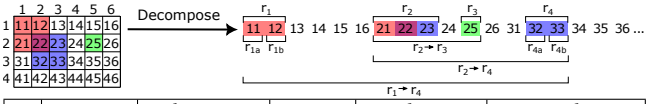
$$T(Q) = \underbrace{\frac{\sum_i Q_i.S_T}{BW(\sum_i Q_i.N_R)}}_{\text{Data transfer}} + \underbrace{T_R\left\lceil\frac{\sum_i Q_i.N_R}{t}\right\rceil}_{\text{Request handling}} + \underbrace{T_\lambda(S_C)\left\lceil\frac{\sum_i Q_i.N_\lambda}{t}\right\rceil}_{\text{Lambda execution}}$$

where BW, T_R, and T_λ are the functions obtained from profiling the cloud platform, as discussed in Section 3.1.

4.2 Query optimization

The role of query optimization is to assign a data retrieval strategy to each chunk that needs to be retrieved to answer a hyperslab query, among the following choices: (1) Get, where one request will transfer the entire chunk from the cloud and then filter unneeded data locally, (2) Range, which issues one or more partial reads for contiguous byte ranges within the chunk, (3) Lambda, which invokes a lambda function on the cloud to filter the data before returning them to the client.

Interestingly, it is not sufficient to minimize the number of bytes transferred. The Range method can return exactly the cells that the query needs, but if this creates too many requests it is better to



#	Current	Split groups	Action	Pending	Assigned
1	$r_1 \rightarrow r_4$	$(r_1,\ r_2 \rightarrow r_4)$	Split	$[r_1, r_2 \rightarrow r_4]$	[ ]
2	r_1	$(r_{1a},\ r_{1b})$	Stop	$[r_2 \rightarrow r_4]$	$[r_1]$
3	$r_2 \rightarrow r_4$	$(r_2 \rightarrow r_3,\ r_4)$	Split	$[r_2 \rightarrow r_3, r_4]$	$[r_1]$
4	$r_2 \rightarrow r_3$	$(r_2,\ r_3)$	Stop	$[r_4]$	$[r_1, r_2 \rightarrow r_3]$
5	r_4	$(r_{4a},\ r_{4b})$	Stop	[ ]	$[r_1, r_2 \rightarrow r_3, r_4]$

Figure 6: Decomposition and range refinement example. The algorithm iteratively splits groups when it reduces estimated cost, and stops when there is no further benefit.

merge requests and send fewer messages. (In the example in Figure 5, the query can be answered using Range(3–8) in one request, and also with Range(3–4, 7–8) and two requests.) Conversely, even when all cells in a chunk are accessed, Get may not be the fastest method because it does not benefit from the processing concurrency that many smaller Range requests would enjoy. The query optimization procedure needs to carefully consider how to merge and split requests to balance these trade-offs.

For simplicity, the presentation has assumed that a single hyperslab query Q is processed. Technically, the query optimization algorithm processes a set of queries at once. This is because the query abstraction of HDF5 is a dataspace, which is defined as a union of hyperslab queries [52]. This allows HDF5 to support sampling and scatter-gather I/O patterns in one read call.

Decomposition. The first phase is the decomposition phase. The decomposition phase first converts a multi-dimensional hyperslab query (technically, an HDF5 dataspace) to a list of one-dimensional byte ranges that must be accessed for each chunk. This is a simple calculation based on the query shape, the chunk shape and the data serialization order. The decomposition phase then sorts the ranges and merges overlapping and contiguous byte ranges. An example of the decomposition phase is shown in Figure 6 for an array that is serialized in row-major order and the queries Q_1=[1:2, 1:2], Q_2=[2:3, 2:3] and Q_3=[2, 5]. After sorting and merging overlapping ranges, the resulting ranges are r_1, r_2, r_3, and r_4. Note how a single hyperslab query can produce multiple byte ranges for further optimization.

Range refinement. Next is the range refinement phase. The goal of range refinement is to find how many Range requests should be transmitted to cover all ranges $r_1, ..., r_n$ that were produced in the decomposition phase. Algorithm 1 shows the range refinement algorithm. It starts by considering grouping all ranges together in one Range request (line 2). It then splits the group into two (lines 8-9), based on which range i has starting point $r_i.start$ with the greatest distance from $r_{i-1}.end$. (If multiple ranges have the same distance to their predecessor, the one that produces the most even split is selected.) The algorithm then compares the cost of processing the range as a single request versus processing the two split groups as separate requests (line 10). If splitting the request lowers the total cost, the two split groups will be added to the end of the pending queue (line 11), else the request will not be split further (line 13). Eventually, some split group will only consist of a single range. The algorithm attempts to split the single range

Algorithm 1: Range refinement algorithm

```
Input: [r_1, ..., r_n], vector of sorted, non-overlapping ranges
1  Pending = []; Assigned = [];
2  Pending.enqueue(r_1 → r_n)
3  while Pending is not empty do
4      cur = Pending.dequeue()
5      if cur is a single range request then
6          (sg1, sg2) = BreakInHalf(cur)
7      else
8          idx = FindLargestGap(cur)
9          (sg1, sg2) = (r_cur.start → r_idx−1, r_idx → r_cur.end)
10     if Cost(cur) > Cost(sg1) + Cost(sg2) then
11         Pending.enqueue([sg1, sg2])          ▷ Split into two
12     else
13         Assigned.enqueue(cur)                ▷ Stop splitting further
14 return Assigned
```

into two halves by breaking the range in the middle (line 6). This continues until the cost model advises to stop splitting, or the range points to a single cell and cannot be split further.

Figure 6 shows how the range refinement algorithm would process the decomposition shown in Figure 6. In the first iteration, the entire range $r_1 \rightarrow r_4$ is considered versus the split groups r_1 and $r_2 \rightarrow r_4$. The cost model indicates that a split has lower total cost, so both split groups are added to the end of the pending queue. In the next iteration, r_1 is considered for splitting into r_{1a} and r_{1b}. The cost model indicates that splitting does not lower the cost, so r_1 will be processed as one request. The processing continues until the pending queue is empty.

Final selection. The cost of the issuing a Range request for each group produced by the range refinement algorithm is compared with the cost of retrieving the entire chunk (Get method) and the cost of retrieving all ranges through a Lambda function invocation. The method that results in the lowest cost will be selected.

Analysis. Selecting the optimal access method requires considering all combinations of methods for every possible range grouping. A complete exploration of the search space is impractical: the difficulty lies in the fact that the retrieval method for each chunk depends on the total number of requests (N_R) and the number of Lambda requests (N_λ) across all chunks, and whether they can fully utilize the bandwidth (expressed as the non-linear BW function). This interdependence makes dynamic programming infeasible: the selected plans for other chunks may no longer be the best if there are changes in the request distribution in this chunk.

The optimization process makes simplifying assumptions to reduce the search space. The heuristic does not consider if a combination of lambda and range retrievals further lowers cost, does not consider splitting at other points except the largest gap, and will never produce overlapping range groups (such as $r_1 \rightarrow r_3$ and $r_2 \rightarrow r_4$). The worst case scenario for Algorithm 1 is to keep splitting the input ranges until there are as many requests as cells in the array, for $O(nlogn)$ worst case complexity. Experimental results corroborate that the search stops quickly: it is rare for the refinement algorithm to create 3 or more range groups (see Section 5.1.2).

4.3 Query execution

The Query Executor (cf. Figure 3) issues the requests the Optimizer selected for each chunk. It issues requests in batches, and the default batch size is 256 requests. Batches are processed sequentially. Requests within a batch are executed in parallel, and execution blocks until all requests in the batch return. While the requests in a batch are issued asynchronously, a read in HDF5 is a synchronous call: the application is blocked until H5Dread() returns with a full data buffer (or an error).

Get and Range requests are issued against the chunk URLs, while Lambda requests are issued against an endpoint that accepts the chunk URL and the hyperslab query as parameters. The Lambda function retrieves the chunk at the pointed URL using a Get request, discards unneeded data, and returns the data directly in the H5Dread return buffer. Range and Get requests may require additional filtering on the client. This filtering step is implemented as a callback function which is called asynchronously after a response is received. After filtering, the callback function will place the data in the appropriate location in the return buffer.

Error handling. ArrayMorph has limited fault-tolerance capabilities to mitigate against networking errors. When a failure or timeout occurs in a Lambda, ArrayMorph falls back to the Get-based method. GET requests are retried three times. If they are not successful, the error code is propagated back to the HDF5 library, which returns the code for I/O error on H5Dread to the application.

Extending ArrayMorph. ArrayMorph supports AWS S3, Azure and Google Cloud Storage. It can be extended to platforms that support data retrievals via HTTP GET (such as OpenStack Swift [14] or MinIO [32]) and serverless functions (such as Illuvatar [15]).

5 EXPERIMENTAL EVALUATION

This section evaluates the performance of ArrayMorph. In the first subsection, we use synthetic workloads to evaluate the performance and robustness of ArrayMorph. In the second subsection, we use real-world applications to compare the end-to-end performance of ArrayMorph with other systems.

Implementation. We build ArrayMorph as an HDF5 VOL plugin in C++ using HDF5 1.14.2. The source code of ArrayMorph is publicly available [24].

Hardware. Unless otherwise noted, we use the us-east-2 region for AWS S3, Azure and Google Cloud. The experiments run on an on-premises 6-core Xeon E-2246G server with 60GB of memory.

Micro-profiling. We micro-profiled each cloud vendor once before running experiments. Micro-profiling completes in about 70-80 minutes, depending on the cloud platform. Bandwidth estimation finishes in about 25 minutes, latency estimation finishes in about 5 minutes, vendors, and the remainder of the time is data generation. A single profiling run is generally sufficient: although the cost modeling is not as accurate when bandwidth or latency fluctuate, the chosen plans remain superior. (We further investigate sensitivity to bandwidth and latency in Section 5.1.4.)

System baselines:

- **HSDS** [17] is an open-source software package that provides a RESTful web interface for accessing datasets in HDF5 format.
- **HSLAMBDA** [51] encapsulates the entire HSDS system in an AWS Lambda function that users can send HTTP requests to.
- **TileDB** [38] is an open-source data management system for efficient management of complex, multi-dimensional data.
- **PushDownDB** [55] uses S3 Select to process SQL queries for CSV and Parquet tables on AWS S3. This baseline evaluates how state-of-the-art relational cloud-based processing performs with array data. To run experiments, we convert all array data to relational tables and add array indices as additional columns (one column per dimension). We convert hyperslab queries to SQL queries which filter on predicates on the array indices. We use the **S3-Side Filter** configuration in PushDownDB as it had the best performance in our experiments.

ArrayMorph configurations. We evaluate ArrayMorph's plan quality by forcing the optimizer to select a particular access method.

- **Get** always issues one Get request for each accessed chunk.
- **Lambda** always invokes a Lambda function for each chunk.
- **Range(Merge)** sends one Range request per chunk that groups together all requests for this chunk. In the example in Figure 6, this configuration would issue the request Range($r_1 \rightarrow r_4$).
- **Range(Fetch)** issues as many Range requests as the ranges produced by the decomposition phase. In the example in Figure 6, the Range(Fetch) configuration would issue four separate Range requests for r_1, r_2, r_3, and r_4.

Methodology and metrics. We run each experiment at least three times at various times of the day. We report the following metrics:

- **Queries per minute** measures performance, and is calculated as the total number of queries issued divided by the median total execution time (in minutes). The total execution time includes creating the hyperslab queries, optimizing and executing them, and transferring the results from the cloud to the client.
- **Queries per dollar** measures cost efficiency, and is calculated as the total number of queries issued divided by the cost. Service providers calculate charges daily, which is too coarse for our experiments. We thus compute the cost manually using the published charging policy and pertinent information from server-side logs. Our service fee calculation is equal to the cost reported from the cloud vendors down to the cent (which is the highest resolution data we can access from the cost dashboard).

We often plot these two metrics together in a two-dimensional plot, where cost efficiency is shown vertically and performance is shown horizontally. This shows a comprehensive view of system performance and cost, where the top right corner indicates the highest performance and cost efficiency. However, because the number and type of queries that are issued is different for each application, comparisons are only meaningful when comparing different systems within the same experiment. Comparisons across experiments are not meaningful as they involve different queries.

5.1 Synthetic Workload Evaluation

In this section, we use a synthetic dataset to evaluate ArrayMorph. The evaluation focuses on the following questions:

- **System performance and cost.** Does ArrayMorph outperform existing cloud-based baselines, and by how much?

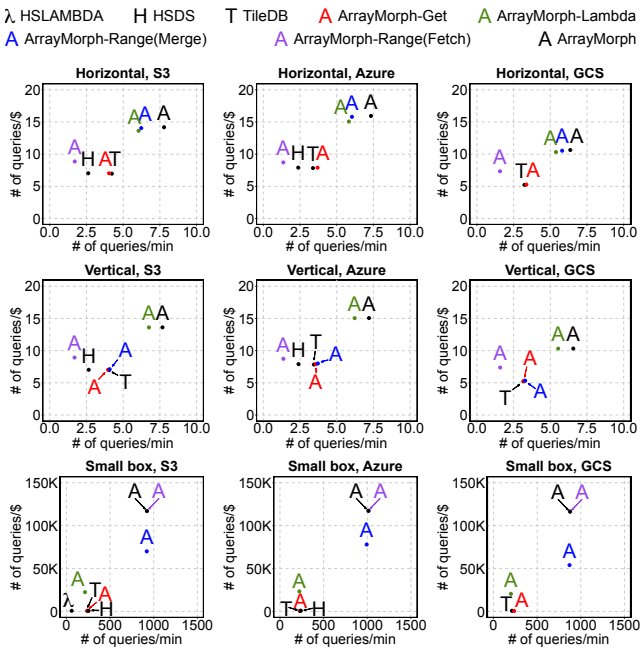


Figure 7: Performance on synthetic workloads.

- **Query plan quality.** How does the mixed strategy chosen by ArrayMorph compare with single-choice alternatives? Does the user-defined ϕ parameter impact query plan selection?
- **Impact of chunking.** Is the performance of ArrayMorph sensitive to the chunking size? If we assume perfect knowledge of the query workload, does optimal chunking trivially produce the optimal data access plan?
- **Big datasets, unstable network and complex queries.** How robust is ArrayMorph? Can it scale to larger datasets, adapt to network fluctuations, and handle complex query patterns?

Synthetic dataset and query workload. We generate a two-dimensional (2D) array with 128K rows and 128K columns of random integers, with a total size of 64GB. Unless otherwise noted in specific experiments, the dataset is stored in 16MB square chunks. We issue three synthetic query workloads on this array:

- The **horizontal box** and **vertical box** workloads issue 10 queries that read contiguous rows or columns, respectively. The starting point is chosen randomly, and the selectivity is fixed to 1% of the array. This workload was used to evaluate the dicing performance of ArrayStore [47] in prior work, and mimics the species classification workload in CameraTrap (see Figures 2a-2b).
- The **small box** workload issues 100 queries that read a 21x21 pixel region from the image, where the starting point is chosen at random. This access pattern is encountered in supernovae detection (see Figure 2c) when focusing on specific objects of interest, such as supernovae, in a much larger image [21, 27].

5.1.1 *System performance and cost.*

Figure 7 plots the performance and cost of ArrayMorph (A) versus the system baselines HSDS (H), TileDB (T), and HSLAMBDA (λ) that always use a single data retrieval method. The upper right corner indicates the best performance and cost efficiency. Looking at the black marks in Figure 7, we find that ArrayMorph consistently outperforms the baseline systems HSDS and TileDB by about 2X, in both performance and cost. This is because both HSDS and TileDB return a lot of redundant data: they transfer about 16GB for the Horizontal and Vertical Box workloads, which is 2X more than the query requests. (Recall that both HSDS and TileDB read every accessed chunk; HSDS uses multiple Get requests, while TileDB uses multiple Range requests.) HSLAMBDA invokes one lambda function to evaluate the entire query. This has poor performance for the small box workload (20X slower than ArrayMorph) due to the very high initialization overhead of lambda functions. HSLAMBDA crashes for the horizontal and vertical box workload on AWS. (Interestingly, although HSLAMBDA crashes before returning any data, it still incurs costs for the lambda executions!) Overall, ArrayMorph achieves the best performance and cost efficiency in all workloads.

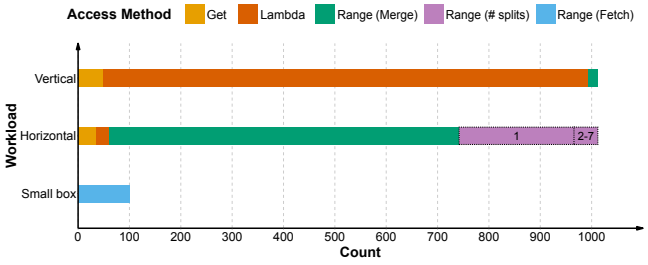
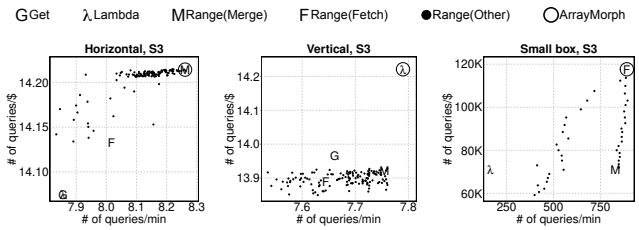


Figure 8: Breakdown of ArrayMorph access method choices for one random query from each workload. Note how ArrayMorph adapts the retrieval strategy based on the hypeslab query and often selects a mix of different strategies.

5.1.2 *ArrayMorph plan quality.*

We now turn our attention to the quality of the plans produced by ArrayMorph. For this analysis, we force ArrayMorph to always pick one plan, and we indicate the different choices in Figure 7 with different colors. We see that no method is always best for all query workloads. ArrayMorph-Get behaves most similarly to TileDB and HSDS, which is reflected in its near-identical performance, cost and data transfer volume. ArrayMorph-Range(Merge) performs much better for the horizontal box workload but does not benefit the vertical box workload: it transfers only about 8GB for the horizontal workload, but twice as much (16GB) for the vertical workload. This is because of how data is serialized: ArrayMorph-Range(Merge) issues a few large requests with the horizontal box workload, but as many as Get when there is no benefit in merging in the vertical box workload. ArrayMorph-Range(Fetch) and ArrayMorph-Lambda minimize the data transferred for all query workloads, transferring 8GB for the horizontal and vertical workloads, and just 170KB for the small box workload, but they have limitations. Range(Fetch) issues too many requests (about 150k requests per query) for the Horizontal/Vertical workloads, which has significant overhead. On the other hand, using lambda functions indiscriminately has significant lambda execution overhead: for the Small Box workload ArrayMorph-Lambda has lower throughput than ArrayMorph-Get.



(a) Horizontal box **(b) Vertical box** **(c) Small box**

Figure 9: Alternative plans ArrayMorph considered for each workload on AWS S3. The selected plan (indicated by a circle) has competitive performance and cost efficiency.

ArrayMorph further benefits from selecting a mixed execution strategy. Figure 8 shows which access strategy ArrayMorph selected for each chunk for one run from each query workload. Note how ArrayMorph adapts, preferring the Range(Merge) method for Horizontal box queries that align with the row-major storage order, and relying more on the Lambda method for Vertical box queries to avoid strided accesses that other methods would incur. Additionally, for some chunks in the Horizontal Box workload, ArrayMorph applies a small number of splits (ranging from 1 to 7), striking a balance between Range(Merge) and Range(Fetch). This result confirms that the range refinement algorithm generally finishes after very few iterations. For the Small box workload, Range(Fetch) is selected because it is estimated to achieve similar throughput to Range(Merge), but lower cost due to transferring less data.

We now look more closely into how well the minimum cost plan that ArrayMorph selected performs in practice, compared to alternatives that the optimizer considered but did not select. To assess this, we perform the following experiment. We start with the same plan that is used in the previous figures, we select one accessed chunk at random, and we force the optimizer to produce all possible plans by making the stop condition (Alg. 1, line 10) to evaluate to false after k iterations. By changing k from 1 to as many cells in the query, we obtain thousands of plans, which we execute and report their performance and cost.

Figure 9 shows the results. We label the single-plan ArrayMorph configurations with the corresponding letter. The dots represent plans that issue a different number of Range requests. The circled plan indicates the final choice made by the ArrayMorph optimizer to process the chunk. (We omit the Get(G) plan from the Small Box result as its performance is over 100X away from all other plans.) The results show that for all workloads the plan selected by ArrayMorph is the best or near-best in terms of performance and cost efficiency. In addition, the results suggest that the parameter ϕ has limited impact in practice, as it is often the case that a choice is better than the others in both the time and cost dimensions. Overall, the results corroborate that ArrayMorph derives its efficiency gains from careful selection among different access plans for different access patterns.

We now evaluate how query plans selected by ArrayMorph are affected by the user-defined ϕ, which expresses a preference between saving time or saving costs. We vary the value of ϕ from

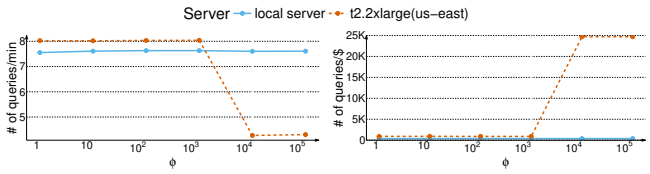


Figure 10: Impact of ϕ parameter on performance and cost.

1 to 10^5 when processing the Vertical box workload. We run experiments on both a local server and an EC2 t2.2xlarge instance located in us-east-2. Figure 10 shows the results. On the local server, both time and service fees are bottlenecked by data transfer, so the Query Optimizer consistently minimizes transfer size regardless of ϕ, resulting in similar query plans and performance. On the EC2 node data transfer from/to S3 is free, so Lambda costs dominate. In this setting, ϕ balances response time and cost: smaller ϕ values favor Lambda for faster execution, while larger ϕ values shift the preference to Get and Range methods to lower cost.

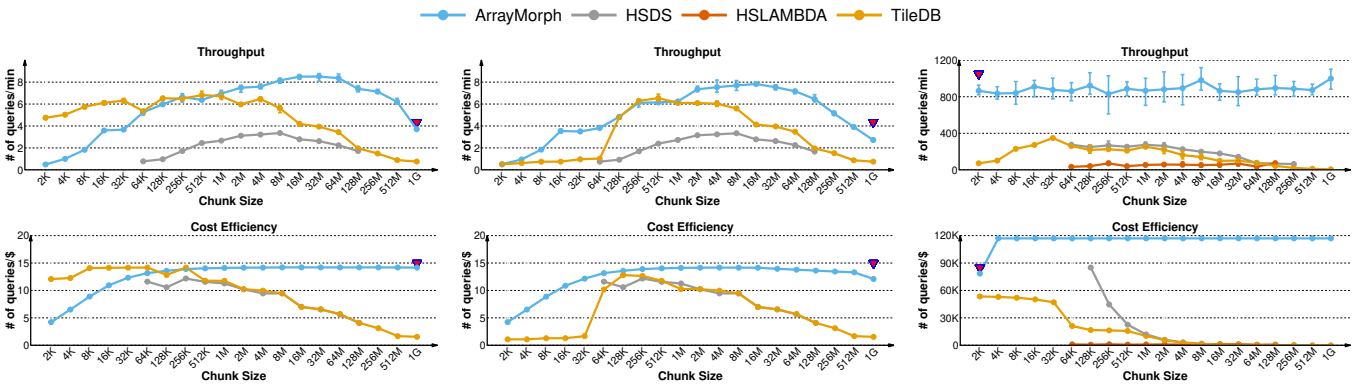
5.1.3 Sensitivity to chunk size.

This section evaluates the robustness of ArrayMorph over different chunk sizes. Of particular interest is whether baselines can achieve comparable performance with a chunking layout generated by optimal chunking algorithms in prior work. We use the QS (Query Shape) algorithm [43] as a representative optimal chunking algorithm. QS takes as input a set of hyperslab queries and generates a chunking layout by analyzing the distribution of historical query shapes, aiming to minimize the number of chunks retrieved for a given workload. The optimal chunking shape generated for the Horizontal, Vertical and Small box workload are [2048, 131072], [131072, 2048], and [16, 32], with chunk sizes of 1GB, 1GB, and 2KB respectively.

In this experiment we vary the chunk size from 2KB to 1GB and fix the chunk shapes to what the QS algorithm recommended. HSDS fails with errors for datasets with chunk sizes smaller than 64KB or larger than 256MB, so we only show results from this smaller range. We report the average, and show variability using error bars, where the bars show the minimum to the maximum value observed. Figure 11 shows the results on AWS S3, where the triangle marks the chunking recommendation of the QS algorithm.

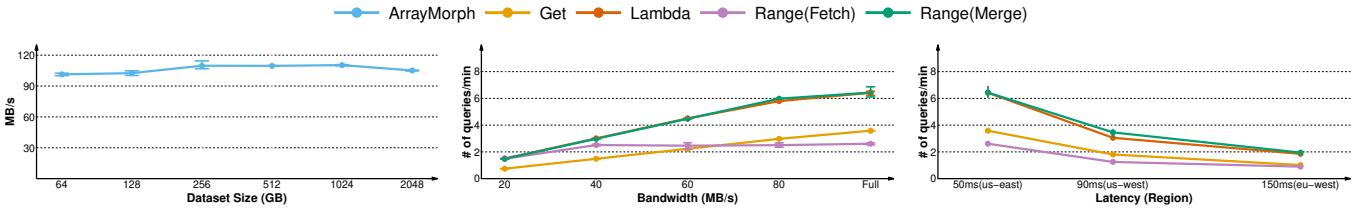
ArrayMorph is the fastest and most cost-effective system for all workloads and chunk sizes, except for the horizontal box workload with TileDB for small chunk sizes (<64KB). Importantly, ArrayMorph achieves the highest throughput for this workload with more typical chunk sizes (in the MBs range). TileDB outperforms ArrayMorph for this particular workload and chunk size due to its unique design choice to store all data in a single object and perform logical chunking. This means that for the horizontal box workload TileDB will only issue a few large requests to one S3 object. In comparison, ArrayMorph stores chunks in separate S3 objects, and hence needs to send separate requests to each object, which impacts performance. This specific design choice results in poor TileDB performance when the requests cannot be merged in the Vertical box workload for small chunk sizes.

The chunking choice of the QS algorithm is suboptimal, because although the chunking layout perfectly matches the query pattern



(a) Horizontal box. (b) Vertical box. (c) Small box.

Figure 11: Sensitivity to chunking size on AWS S3. For efficient hyperslab query processing in the cloud, choosing the access method carefully as ArrayMorph does is as important as selecting the perfect chunk size.



(a) Varying dataset size. (b) Varying bandwitdh. (c) Varying latency.

Figure 12: Scalability and robustness of ArrayMorph on the Horizontal box workload on AWS S3.

for the Horizontal and Vertical Box workloads, the chunk boundaries never perfectly align with the hyperslab queries, requiring accesses to multiple very large chunks for a single hyperslab query. Because of this, ArrayMorph is up to 4.7X faster and 8.8X cheaper than the baselines with 1GB chunk size. For the Small box workload, ArrayMorph runs 11.5X faster and 1.5X cheaper than TileDB with 2KB chunking. This demonstrates that tuning the chunking layout is insufficient for hyperslab query processing—choosing the access method carefully is just as important.

5.1.4 Scalability and system-level robustness.
This section evaluates the scalability and system-level robustness of ArrayMorph on AWS S3.

Scalability. To evaluate how ArrayMorph performs on larger datasets, we change the dataset size from 64GB to 2TB and execute the Horizontal Box workload. The selectivity of the queries is fixed (1% on the 64GB dataset) and does not change with the dataset size. Figure 12(a) shows the average data transfer throughput (in MB/s) of ArrayMorph, along with the throughput variability indicated by error bars. The results show that ArrayMorph has consistent throughput as the dataset size grows, consistently achieving approximately 100-110 MB/s, which is the network speed for this system. This demonstrates that ArrayMorph is capable of handling much larger TB-sized datasets.

Robustness under network fluctuations. The estimated network bandwidth and latency are crucial factors for the cost model. In this experiment we investigate how sensitive the plan selection of ArrayMorph is to changes in network bandwidth and latency that may occur. We focus on four query plans (Get, Range(Merge), Lambda, and Range(Fetch)), which reflect different choices made by ArrayMorph. By comparing their relative performance under different network conditions, we assess whether the same plan remains competitive—confirming if ArrayMorph needs frequent re-profiling as network conditions change.

We control the available bandwidth using the `readRateLimiter` option provided by the AWS S3 SDK, and vary it from 20 MB/s up to the full available bandwidth. We control the latency by deploying EC2 instances in different AWS regions with different round-trip latencies to where the S3 data are stored: local/us-east (50 ms), us-west (90 ms), and eu-west (150 ms). Figure 12(b) and 12(c) show the results. In this experiment Merge consistently achieves the best performance. Although all methods show reduced throughput as bandwidth decreases and latency increases, the relative performance order remains unchanged. The fact that relative performance remains consistent under network fluctuations indicates that ArrayMorph will not benefit from frequent re-profiling.

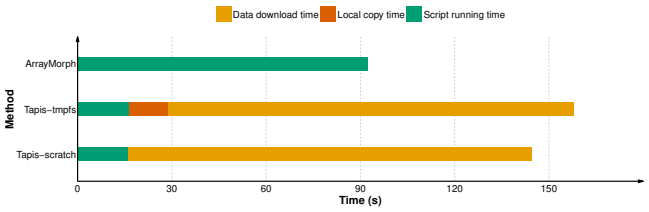


Figure 13: ArrayMorph performance on the DLIO benchmark versus file-based data transfer.

Complex query patterns. We also evaluate ArrayMorph using the Deep Learning I/O (DLIO) benchmark [5, 11]. DLIO is designed to emulate diverse I/O patterns commonly observed in deep learning workloads, and has been used to study and categorize I/O behaviors in machine learning applications [29]. We integrate ArrayMorph into the existing DLIO benchmarking scripts [3], replacing its default h5py-based data reader. We compare the end-to-end running time of using ArrayMorph against the conventional pipeline, which first downloads the entire dataset from cloud storage to local storage before running ML tasks. For dataset download, we use Tapis [50], a high-performance data transfer service optimized for moving large-scale scientific datasets to HPC environments. Specifically, we compare two Tapis-based methods: (1) *Tapis-scratch*, which downloads the dataset to the distributed file system and starts the DLIO benchmark, and (2) *Tapis-tmpfs*, which copies the downloaded dataset into a RAM-based temporary file system (tmpfs) to eliminate disk-based I/O during the execution of the DLIO benchmark, as ArrayMorph does.

Figure 13 shows the results. ArrayMorph runs about 1.7X faster than Tapis-based methods. The key performance gain comes from avoiding redundant data transfer and I/O. ArrayMorph fetches only the necessary data directly from the cloud during the DLIO benchmark execution, avoiding redundant data transfer and bypassing local disk I/O. In contrast, Tapis-based methods incur the overhead of downloading the entire dataset before computation. The results demonstrate that ArrayMorph efficiently handles and remains robust against the complex query patterns commonly encountered in ML applications.

5.2 Real-world Workload Evaluation

This section evaluates the end-to-end performance of ArrayMorph compared to existing cloud-based data management systems on real-world machine learning workloads. All systems are evaluated across their supported cloud platforms to load data using hyperslab queries, with all ML tasks executing on a single NVIDIA A100 GPU. We use the following real-world ML applications in our evaluation:

- **BERT pretrained model evaluation [35]** follows the PyTorch implementation by NVidia for evaluating the pretrained BERT model. The dataset consists of 250 HDF5 files (totaling 250GB) with five 2-D and one 1-D array per file, all in 4-byte integer format. Each file is chunked and uses 16MB chunks. We convert and upload the dataset to the cloud using ArrayMorph and all baselines. We run the PyTorch evaluation script while directly fetching data from the cloud using h5py interface. The script issues about 1500 hyperslab queries to randomly sample 1% of the dataset. Throughput (queries per minute) is computed by dividing the number of queries by the total running time. For PushDownDB, the total running time includes the overhead of converting query results from CSV to NumPy arrays.

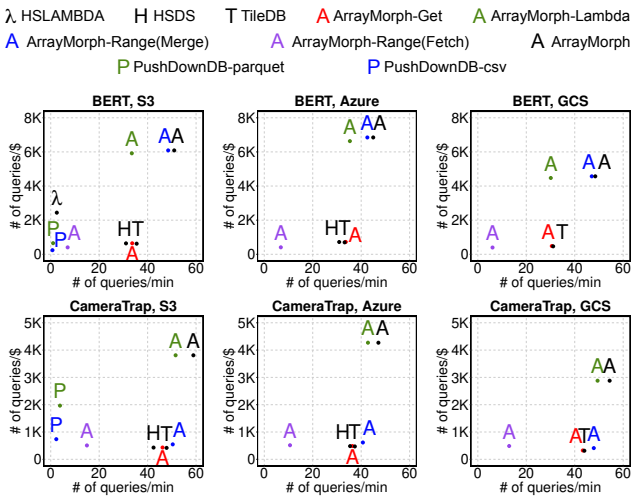


Figure 14: Performance of Real-world Workloads.

- **BioCLIP few-shot classification [49]** evaluates the BioCLIP vision model on wildlife images using the SWG CameraTrap dataset [30], which contains 10K images with bounding box annotations and class labels. Each image is stored as a (3, 2880, 3840) 3-D array of 1-byte unsigned integers. The total size is about 300GB, and we use a chunk size of 16MB. We execute the BioCLIP repository's PyTorch evaluation script [48], directly retrieving bounding boxes from the cloud via hyperslab queries (see Figure 1). During evaluation, 10K hyperslab queries are issued to fetch region-of-interest data.

5.2.1 *BERT Results.*

Figure 14 shows the results from evaluating ArrayMorph when sampling BERT training data, Compared with existing systems, ArrayMorph is up to 1.66X faster and up to 9X cheaper than HSDS and TileDB, mainly by transferring less data. Specifically, ArrayMorph transfers only 2.7GB of data, whereas HSDS and TileDB each transfer around 26.8GB. PushDownDB and HSLAMBDA avoid redundant data transfer but exhibit low performance. For PushdownDB, the converted relational tables are much larger than the original data size, which S3 Select takes longer to process. Although S3 Select returns no redundant data, it incurs the extra overhead of data conversion before running ML tasks from CSV or Parquet formats into a dense tensor representation. These drawbacks make it up to 71.X slower than ArrayMorph. HSLAMBDA exacerbates the long cold start times of executing Lambda functions, making it 20.3X slower than ArrayMorph.

Among the single-operator methods implemented by ArrayMorph, Range(Merge) transfers no redundant data due to the alignment of hyperslab queries with the row-major storage order. However, it uses a single request to fetch data, which limits network throughput. In contrast, ArrayMorph improves performance by

splitting large segments into multiple smaller requests, making better use of the network and achieving 1.06X speedup over the Range(Merge) method. Range(Fetch) generates an excessive number of small (512-byte) requests, leading to significant overhead in both time and cost, resulting in low throughput and cost efficiency. For Lambda, the execution time of the lambda function becomes a bottleneck when queries access only a single 16MB chunk, making it even slower than Get. Although its cost efficiency is comparable to ArrayMorph, the throughput is 1.52X lower. In conclusion, the results show ArrayMorph outperforms existing systems in terms of both time and cost efficiency. This is because ArrayMorph considers factors such as serialization order and lambda execution overheads, instead of always picking a single data retrieval method.

5.2.2 CameraTrap Results.
We now focus on the CameraTrap workload of Figure 14. CameraTrap fails to run on HSLAMBDA. Compared with existing systems, ArrayMorph is up to 1.39X faster and 9.25X cheaper than other array-based systems, and 25.6X faster and 5.24X cheaper than PushDownDB. It transfers only 28.3GB of data while TileDB and HSDS transfer more than 265GB. Among the single-operator methods implemented internally, ArrayMorph runs up to 1.17X faster and up to 7.0X cheaper than Range(Merge) and Lambda. This is because hyperslab queries generated by annotations show diverse access patterns—some align with the row-major storage order, others do not, and some may even span an entire chunk. As a result, no single-operator method performs optimally across all cases. In contrast, ArrayMorph adapts to different access patterns by selecting suitable plans through the split-based algorithm guided by the cost model. In conclusion, ArrayMorph achieves the highest throughput in terms of both time and cost by adapting its data access methods for different queries.

6 RELATED WORK

Data processing on the cloud. Cloud storage services have gained popularity in recent years due to their convenience, flexibility and scalability. Many data processing platforms integrate with cloud services, including Presto [44], Snowflake [10], Hive [53], Redshift [19], and Spark [4] for relational data; TileDB [38], and HSDS [17] for array data. Prior research has explored efficient cloud data processing. PushdownDB [56] and FlexPushdownDB [55] leverage computation pushdown via AWS S3Select to execute SQL operations closer to the storage. Recent work also utilizes serverless cloud functions. Lambada [33], Starling [40], and Flint [28] build query engines on AWS Lambda to reduce latency. HSLAMBDA [51] extends this approach to hyperslab queries, while LambdaML [23] applies it to ML training.

Unlike relational cloud databases, efficient array data management remains underexplored. Existing systems [17, 38] often fetch entire objects, which is inefficient for applications requiring partial data, such as ML sampling [12] and video analytics [16, 26, 39, 41, 49]. In contrast, ArrayMorph optimizes query execution across multiple data retrieval metods, selecting the most efficient strategy.

Cloud Data Placement Optimization. Data placement in cloud storage significantly impacts query latency and cost. Prior work has explored intelligent placement strategies to optimize performance. Many systems [1, 6, 36] automate data distribution across multiple cloud data centers to reduce cross-region transfer costs and improve locality for frequently accessed data. Some systems [22, 34, 55, 58] leverage edge storage or caching to minimize redundant data transfers. Other works [18, 45] study how edge computing (e.g., on sensors or drones) can preprocess data before uploading to the cloud, reducing transfer latency. While ArrayMorph optimizes hyperslab retrievals rather than data placement, integrating adaptive placement strategies could further enhance efficiency.

Array data processing. Array data processing is essential for high-performance applications in fields like machine learning [12, 16, 20, 26, 39, 41] and scientific computing [25, 46], where vast volumes of multidimensional data are common. Prior research has extensively studied high-performance array data management systems, covering areas such as storage and efficient data retrieval [8, 38, 47], as well as advanced array operators, including join [13, 59], multiplication [7], and others. The comprehensive survey by Rusu [42] concisely summarizes recent research in array data management.

The chunking strategy plays a crucial in array data management, as well-tuned chunk sizing can significantly reduce I/O operations for array data retrieval and processing. Previous work [37, 42, 43] has extensively studied the implications of chunking and proposed different chunking strategies. Some array systems [2, 57] support fast rechunking to adapt to dynamic workloads. While ArrayMorph does not support automatic rechunking, it mitigates the impact of suboptimal layouts by changing the data access strategy based on the data layout.

7 CONCLUSION

This paper explores the opportunity of accelerating the I/O part of machine learning applications by pushing the evaluation of hyperslab queries to the cloud. Prior work considers three methods to selectively retrieve parts of objects in cloud storage services, namely chunking, reading a contiguous byte range, and evaluating a lambda function. We find that existing solutions have limitations in that they are only implementing one of these methods, which is not always optimal in time or cost. To overcome these limitations, we propose ArrayMorph, a cloud-based array data storage system that automatically determines which is the best method to use to retrieve regions of interest from data on the cloud. ArrayMorph formulates data accesses as queries and optimizes them using a multi-phase cost-based optimization. ArrayMorph is designed to be seamlessly integrated with Python/PyTorch-based ML applications, and supports queries against leading object store services, namely Amazon S3, Azure Blob Storage, and Google Cloud Storage. We evaluate ArrayMorph on synthetic workloads and real-world workloads. The experimental results show that ArrayMorph can transfer up to 9.8X less data than existing systems, which makes it run up to 1.7X faster and 9X cheaper than existing systems.

8 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments that helped us improve the paper. We thank Donghe Kang for participating in discussions and giving feedback during the initial development of this work. This work was supported by the U.S. National Science Foundation grants OAC-2112606 and OAC-2018627.

REFERENCES

[1] Sharad Agarwal, John Dunagan, Navendu Jain, Stefan Saroiu, Alec Wolman, and Harbinder Bhogan. 2010. Volley: automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 2.

[2] Md Mahbub Alam, Luis Torgo, and Albert Bifet. 2022. A Survey on Spatio-temporal Data Analytics Systems. *ACM Comput. Surv.* 54, 10s, Article 219 (Nov. 2022), 38 pages.

[3] Argonne Leadership Computing Facility. 2025. Deep Learning I/O (DLIO) Benchmark. https://github.com/argonne-lcf/dlio_benchmark/tree/main. Accessed: 2025-06-21.

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394.

[5] Oana Balmau. 2022. Characterizing I/O in Machine Learning with MLPerf Storage. *SIGMOD Rec.* 51, 3 (Nov. 2022), 47–48.

[6] Tiemo Bang, Chris Douglas, Natacha Crooks, and Joseph M. Hellerstein. 2024. SkyPIE: A Fast & Accurate Oracle for Object Placement. *Proc. ACM Manag. Data* 2, 1, Article 55 (March 2024), 27 pages.

[7] Charles R. Baugh and Bruce A. Wooley. 1973. A Two's Complement Parallel Array Multiplication Algorithm. *IEEE Trans. Comput.* 22, 12 (Dec. 1973), 1045–1047.

[8] 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* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 963–968.

[9] Microsoft Corporation. 2024. Image Detection Demo with Pytorch-Wildlife. https://cameratraps.readthedocs.io/en/latest/demo/image_detection_demo.html#3.-JSON-Format Accessed: 2025-06-21.

[10] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226.

[11] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath. 2021. DLIO: A Data-Centric Benchmark for Scientific Deep Learning Applications. 81–91 (2021).

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint* arXiv:1810.04805 (2018). https://arxiv.org/abs/1810.04805 Accessed: 2025-06-21.

[13] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. 2015. Skew-Aware Join Optimization for Array Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 123–135.

[14] OpenStack Foundation. 2024. OpenStack: Open Source Cloud Computing Platform. https://www.openstack.org/ Accessed: 2025-06-21.

[15] Alexander Fuerst, Abdul Rehman, and Prateek Sharma. 2023. Ilúvatar: A Fast Control Plane for Serverless Computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 267–280.

[16] Asaf Gal, Jonathan Saragosti, and Daniel JC Kronauer. 2020. anTraX, a software package for high-throughput video tracking of color-tagged insects. *Elife* 9 (2020), e58145.

[17] HDF Group. 2023. Highly Scalable Data Service. https://github.com/HDFGroup/hsds Accessed: 2025-06-21.

[18] Piotr Grzesik and Dariusz Mrozek. 2022. Accelerating edge metagenomic analysis with serverless-based cloud offloading. In *International Conference on Computational Science*. Springer, 481–492.

[19] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[21] 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.

[22] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph Seffi Naor, and Sriram Rao. 2018. Netco: Cache and I/O Management for Analytics over Disaggregated Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 186–198.

[23] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 857–871.

[24] Ruochen Jiang. 2024. ArrayMorph. https://github.com/ruochenj123/ArrayMorph. Accessed: 2025-06-21.

[25] CO Justice, JRG Townshend, EF Vermote, E Masuoka, RE Wolfe, Nazmi Saleous, DP Roy, and JT Morisette. 2002. An overview of MODIS Land data processing and product status. *Remote sensing of Environment* 83, 1-2 (2002), 3–15.

[26] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1586–1597.

[27] Donghe Kang, Vedang Patel, Ashwati Nair, Spyros Blanas, Yang Wang, and Srinivasan Parthasarathy. 2019. Henosis: workload-driven small array consolidation and placement for HDF5 applications on heterogeneous data stores. In *Proceedings of the ACM International Conference on Supercomputing* (Phoenix, Arizona) *(ICS '19)*. Association for Computing Machinery, New York, NY, USA, 392–402.

[28] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 451–455.

[29] Noah Lewis, Jean Luca Bez, and Suren Byna. 2025. I/O in Machine Learning Applications on HPC Systems: A 360-degree Survey. *ACM Comput. Surv.* 57, 10, Article 256 (May 2025), 41 pages.

[30] LILA: The Living Image of Animals. 2023. SWG Camera Traps 2018-2022 Dataset. https://lila.science/datasets/swg-camera-traps. Accessed: 2025-06-21.

[31] Microsoft. 2023. Microsoft MegaDetector. https://github.com/microsoft/CameraTraps. Accessed: 2025-06-21.

[32] MinIO, Inc. 2025. MinIO Object Storage. https://github.com/minio/minio. Accessed: 2025-06-21.

[33] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 115–130.

[34] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. 2021. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 228–244.

[35] NVIDIA. 2023. BERT for PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT. Accessed: 2025-06-21.

[36] Kwangsung Oh, Abhishek Chandra, and Jon Weissman. 2017. TripS: automated multi-tiered data placement in a geo-distributed cloud environment. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '17)*. Association for Computing Machinery, New York, NY, USA, Article 12, 11 pages.

[37] E. J. Otoo, Doron Rotem, and Sridhar Seshadri. 2007. Optimal chunking of large multidimensional arrays for data warehousing. In *Proceedings of the ACM Tenth International Workshop on Data Warehousing and OLAP* (Lisbon, Portugal) *(DOLAP '07)*. Association for Computing Machinery, New York, NY, USA, 25–32.

[38] 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.

[39] Talmo D Pereira, Nathaniel Tabris, Arie Matsliah, David M Turner, Junyu Li, Shruthi Ravindranath, Eleni S Papadoyannis, Edna Normand, David S Deutsch, Z. Yan Wang, Grace C McKenzie-Smith, Catalin C Mitelut, Marielisa Diez Castro, John D'Uva, Mikhail Kislin, Dan H Sanes, Sarah D Kocher, Samuel S-H, Annegret L Falkner, Joshua W Shaevitz, and Mala Murthy. 2022. SLEAP: A deep learning system for multi-animal pose tracking. *Nature Methods* 19, 4 (2022).

[40] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 131–141.

[41] Tapis Project. 2022. Camera Traps: A simulator and edge device application for classifying images. https://github.com/tapis-project/camera-traps Accessed: 2025-06-21.

[42] Florin Rusu. 2023. Multidimensional Array Data Management. *Foundations and Trends® in Databases* 12, 2-3 (2023), 69–220.

[43] Sunita Sarawagi and Michael Stonebraker. 1994. Efficient Organization of Large Multidimensional Arrays. In *Proceedings of the Tenth International Conference on Data Engineering*. IEEE Computer Society, USA, 328–336.

[44] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813.

[45] Sachchidanand Singh. 2017. Optimize cloud computations using edge computing. In *2017 International Conference on Big Data, IoT and Data Science (BID)*. 49–53.

[46] William C Skamarock, Joseph B Klemp, Jimy Dudhia, David O Gill, Dale M Barker, Michael G Duda, Xiang-Yu Huang, Wei Wang, Jordan G Powers, et al. 2008. A description of the advanced research WRF version 3. *NCAR technical note* 475, 125 (2008), 10–5065.

[47] 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* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 253–264.

[48] Samuel Stevens, Jiaman Wu, Matthew J. Thompson, Elizabeth G. Campolongo, Chan Hee Song, and David Edward Carlyn. 2024. *BioCLIP*. Accessed: 2025-06-21.

[49] Samuel Stevens, Jiaman Wu, Matthew J Thompson, Elizabeth G Campolongo, Chan Hee Song, David Edward Carlyn, Li Dong, Wasila M Dahdul, Charles Stewart, Tanya Berger-Wolf, Wei-Lun Chao, and Yu Su. 2024. BioCLIP: A Vision Foundation Model for the Tree of Life. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 19412–19424.

[50] Joe Stubbs, Richard Cardone, Mike Packard, Anagha Jamthe, Smruti Padhy, Steve Terry, Julia Looney, Joseph Meiring, Steve Black, Maytal Dahan, Sean Cleveland, and Gwen Jacobs. 2021. Tapis: An API Platform for Reproducible, Distributed Computational Research. In *Advances in Information and Communication*, Kohei Arai (Ed.). Springer International Publishing, Cham, 878–900.

[51] The HDF Group. 2021. HSDS New Features: AWS Lambda and Direct Access. https://www.hdfgroup.org/wp-content/uploads/2021/10/HSDS-New-Features-AWS-Lambda-and-Direct-Access.pdf. Accessed: 2025-06-21.

[52] The HDF Group. 2024. *HDF5 Documentation: Dataspace Transfer Property List: Selection Conversion Mode.* https://support.hdfgroup.org/documentation/hdf5/latest/_h5_s__u_g.html#subsubsec_dataspace_transfer_select Accessed: 2025-06-21.

[53] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1626–1629.

[54] Devis Tuia, Benjamin Kellenberger, Sara Beery, Blair R. Costelloe, Silvia Zuffi, Benjamin Risse, Alexander Mathis, Mackenzie W. Mathis, Frank van Langevelde, Tilo Burghardt, Roland Kays, Holger Klinck, Martin Wikelski, Iain D. Couzin, Grant van Horn, Margaret C. Crofoot, Charles V. Stewart, and Tanya Berger-Wolf. 2022. Perspectives in machine learning for wildlife conservation. *Nature Communications* 13 (2022), 792.

[55] Yifei Yang, Matt Youill, Matthew Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: hybrid pushdown and caching in a cloud DBMS. *Proc. VLDB Endow.* 14, 11 (July 2021), 2101–2113.

[56] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1802–1805.

[57] Ramon Antonio Rodriges Zalipynis. 2018. ChronosDB: distributed, file based, geospatial array DBMS. *Proc. VLDB Endow.* 11, 10 (June 2018), 1247–1261.

[58] Qizhen Zhang, Philip A Bernstein, Daniel S Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. Compucache: Remote computable caching using spot vms. In *Annual Conference on Innovative Data Systems Research (CIDR'22)*.

[59] Weijie Zhao, Florin Rusu, Bin Dong, and Kesheng Wu. 2016. Similarity Join over Array Data. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 2007–2022.