Vineyard: Optimizing Data Sharing in Data-Intensive Analytics

WENYUAN YU, Alibaba Group, China
TAO HE, Alibaba Group, China
LEI WANG, Alibaba Group, China
KE MENG, Alibaba Group, China
YE CAO, Alibaba Group, China
DIWEN ZHU, Alibaba Group, China
SANHONG LI, Alibaba Group, China
JINGREN ZHOU, Alibaba Group, China

Modern data analytics and AI jobs become increasingly complex and involve multiple tasks performed on specialized systems. Sharing of intermediate data between different systems is often a significant bottleneck in such jobs. When the intermediate data is large, it is mostly exchanged through *files* in standard formats (*e.g.,* CSV and ORC), causing high I/O and (de)serialization overheads. To solve these problems, we develop Vineyard, a high-performance, extensible, and cloud-native object store, trying to provide an intuitive experience for users to share data across systems in complex real-life workflows. Since different systems usually work on data structures (*e.g.,* dataframes, graphs, hashmaps) with similar interfaces, and their computation logic is often loosely-coupled with how such interfaces are implemented over specific memory layouts, it enables Vineyard to conduct data sharing efficiently at a high level via memory mapping and method sharing. Vineyard provides an IDL named VCDL to facilitate users to register their own intermediate data types into Vineyard such that objects of the registered types can then be efficiently shared across systems in a polyglot workflow. As a cloud-native system, Vineyard is designed to work closely with Kubernetes, as well as achieve fault-tolerance and high performance in production environments. Evaluations on real-life datasets and data analytics jobs show that the above optimizations of Vineyard can significantly improve the end-to-end performance of data analytics jobs, by reducing their data-sharing time up to 68.4×.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Theory of computation → Data exchange**; • **Information systems → Key-value stores**.

Additional Key Words and Phrases: data sharing, in-memory object store

ACM Reference Format:
Wenyuan Yu, Tao He, Lei Wang, Ke Meng, Ye Cao, Diwen Zhu, Sanhong Li, and Jingren Zhou. 2023. Vineyard: Optimizing Data Sharing in Data-Intensive Analytics. *Proc. ACM Manag. Data* 1, 2, Article 200 (June 2023), 27 pages. https://doi.org/10.1145/3589780

Authors' addresses: Wenyuan Yu, Alibaba Group, China, Beijing; Tao He, Alibaba Group, China, Beijing; Lei Wang, Alibaba Group, China, Beijing; Ke Meng, Alibaba Group, China, Beijing; Ye Cao, Alibaba Group, China, Beijing; Diwen Zhu, Alibaba Group, China, Shanghai; Sanhong Li, Alibaba Group, China, Shanghai; Jingren Zhou, Alibaba Group, China, Hangzhou, vineyard@alibaba-inc.com.

2836-6573/2023/6-ART200 $15.00
https://doi.org/10.1145/3589780

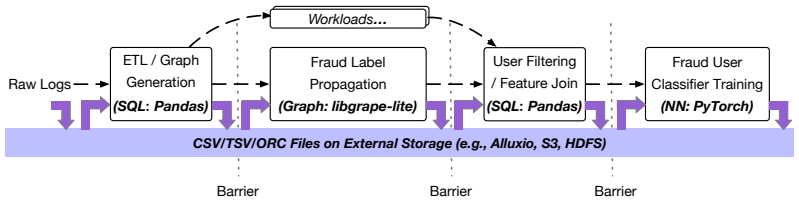


Fig. 1. A Real-life Fraud Detection Job.

1 INTRODUCTION

Data-intensive computing is a typical class of big data analytics applications, and most of their processing time is devoted to I/O and data movement and manipulation [4, 48]. Recent studies [25, 36, 51] have reported that many jobs submitted to cloud platforms fall into this category. In addition, there is an increasingly new trend where multiple tasks belonging to different types of workloads are fused together to form a single *complex* job (workflow) [41, 61, 64]. Figure 1 depicts a real-life fraud detection job from Alibaba [23], in which diverse kinds of workloads (*e.g.,* SQL, graph processing, and deep learning) are involved. Consequently, handling such complex data-intensive jobs efficiently is highly desired.

In response, two practical solutions have been proposed, namely SINGLE-SYSTEM and MULTI-SYSTEM, but both of them still leave large room for improvement. The SINGLE-SYSTEM copes with diverse workloads in a single *general-purpose* system like Spark [63, 64]. Unfortunately, a workload implemented in general-purpose systems often performs worse than that in *workload-specific* systems, which are tailored for a particular type of workload. Workload-specific systems (*e.g.,* Gemini [67] for graph processing) adopt specialized data structures (*e.g.,* CSR/CSC for graphs) and optimizations in their execution engines to offer superior performance. Sometimes, the performance gap can reach up to several orders of magnitudes [47, 53].

Thus, a more widely-adopted solution to handling complex data analytics is MULTI-SYSTEM, where users employ multiple workload-specific systems to handle different types of workloads. To exchange intermediate results between these systems, *files* in standard formats (*e.g.,* CSV and ORC) on external storage (*e.g.,* HDFS [57], Amazon S3 [10], and Alluxio [40]) are commonly used, as shown in Figure 1. In this way, different systems can be nicely bridged, but the cost of *data sharing* across systems becomes higher than that in SINGLE-SYSTEM (*e.g.,* Spark). The root cause is that the interfaces for files (*i.e.,* read and write) are primitive and unstructured. Hence, dumping/loading data as files can incur unnecessary copying, (de)serializations and/or I/Os. Moreover, instances of high-level data structures (*i.e.,* objects) lose their rich semantics and behaviors when flattened to a sequence of bytes. It requires systems to repeatedly implement their own in-memory formats and methods, as well as the (de)serialization logic. Such loss of semantics also makes it difficult to apply cross-system optimizations (*e.g.,* pipelined execution and computation-data co-locating).

To tackle the above issues, we present Vineyard, a distributed object store for data sharing across workload-specific data processing systems. We observe that various workload-specific systems are built on top of some common data structures (*e.g.,* dataframes and graphs). Although data structures of the same type often have different memory layouts and implementations in different systems, their high-level interfaces keep almost the same [33]. This observation inspires us to offer customizable common memory layouts and implementations for intermediate data structures shared by multiple systems, and enable the efficient in-memory sharing of these data structures and their associated interfaces and methods among different systems written in different languages. In this fashion, different systems can access *objects* in Vineyard just like the high-level native objects of their own. Vineyard supports zero-copy data sharing by decoupling an object into

metadata and a group of payloads, allowing users to reconstruct a complex object from memory-mapped payloads in different processes with metadata. Vineyard organizes large objects in chunk granularity, allowing each chunk spilled to an external storage or distributed in remote hosts to support big data. Like Protobuf [35], Vineyard provides an IDL (intermediate description languages) named VCDL for users to define new formats to alleviate the integration burden with Vineyard. Besides, Vineyard is designed to be cloud-native and work robustly and efficiently in real-world workflows. Vineyard is open-sourced[1] and under active development. It is already integrated or being integrated with more than a dozen data processing systems including PyTorch [52], TensorFlow [8], PowerGraph [34], GraphScope (distributed graph computation systems) [23] and Mars (large-scale scientific computation engine) [45].

Contributions & organization. We make the following contributions to facilitate and speed up cross-system data sharing:

(1) Motivation (§2). We analyze the current solutions for data sharing from the aspects of data storage medium and data format to reveal the reasons why current solutions fail in complex workflows.

(2) System (§3). We first describe the opportunity that Vineyard tries to seize. Then we detail the system architecture of Vineyard, and its components, and discuss the challenges to achieve our design goal.

(3) Implementation (§4). We detail the implementation of Vineyard from three aspects: (i) A distributed in-memory object store for composable data structures (§4.1). (ii) The VCDL IDL and code generator to facilitate the type registration and integration (§4.2). (iii) Locality awareness on Kubernetes clusters [7] and fault tolerance in cloud environments (§4.3).

(4) Use cases (§5). We provide a set of Vineyard integration use cases with six data processing systems, which demonstrates the integration friendliness of Vineyard.

(5) Evaluation (§6). An extensive evaluation of Vineyard. We find the following: (i) Vineyard can boost the end-to-end performance by 3× times on average and reduce the data-sharing time by 28.8× on average compared with its best competitors. (ii) The overhead of Vineyard is negligible, which only takes about 40 milliseconds to share hundreds of Gigabytes of objects. (iii) The integration effort is low which only requires about 100 lines of changes for a system.

Limitations and non-goals. Vineyard has a number of restrictions and non-goals. First, Vineyard targets at optimizing data sharing across systems for data-intensive workflows. As for computation-dense workflows such as model training, the end-to-end performance improvement is quite limited since most of their execution time is devoted to computational kernels. Second, Vineyard does not provide a global workflow optimizer and always assumes users have chosen an appropriate workload-specific system for each task, while some prior studies (*e.g.,* Musketeer [33]) aim to map tasks to suitable back-end execution systems. Third, Vineyard aims at the sharing of large immutable intermediate results, while the caching of frequently updated data like Memcached [20] or Redis [44] is not our goal.

2 MOTIVATION

To enable data sharing among different systems, diverse solutions have been proposed. Their fundamental differences mainly come from *where* the intermediate data is stored (*i.e.,* storage medium), and *how* the data is represented (*i.e.,* format).

[1]Vineyard is available at https://github.com/v6d-io/v6d

Storage medium. In general, existing solutions utilize either *main memory* or *external storage* to share data across systems. However, many challenging issues still exist in handling large and distributed data under diverse running environments.

Memory. Memory can be used as a medium for sharing data among different libraries and systems on a single machine. For example, in the PyData ecosystem, libraries, such as Numpy [15] and PyTorch, can exchange a large tensor as a variable directly within a single Python process with zero copying. Some multi-process parallel computation systems use shared memory for exchanging intermediate data across processes. An object store Plasma developed for Apache Arrow [31] and Ray [49] also allows sharing of immutable data (objects) between processes with simple `PUT/GET` operations in shared memory. Memory provides an efficient way to exchange data across systems on a single machine avoiding unnecessary copying and/or I/Os with external storage. However, memory-based solutions are difficult to scale. First, intermediate data is required to locate on a single machine and fit in the main memory, while existing solutions lack the support of distributed or outsized data, both are pivotal to real-life big data applications. Second, memory-based solutions sometimes require ad-hoc and specialized integrations between pairs of involved systems/libraries.

External storage. Different from memory, external storage (*e.g.,* local disks, HDFS and Amazon S3) often provides primitive yet unified file system-like interfaces (*e.g.,* `open`, `read` and `write`). Furthermore, its capacity can be considered unlimited in cloud environments. As a result, external storage-based solutions can bridge different systems without intrusive changes to the data processing system itself, and are appropriate to exchange extremely large-scale data. However, compared with memory-based solutions, the performance of external storage-based solutions is relatively poor due to the following reasons. First, intermediate data is stored as *files* in file systems (*e.g.,* ORC and Parquet files), while dumping/loading data as files will incur expensive copying and I/O overheads. Second, each system needs to access files (*i.e.,* a sequence of bytes) through primitive file interfaces, while high-level data structures contain rich semantics and methods (*e.g.,* the `find(key)` method for a hashmap). To keep rich semantics of high-level data structures, each system needs to build its internal data structures from a sequence of bytes, causing (de)serialization costs. For example, to execute the complex job shown in Figure 1 over external storage, the costs for data sharing across systems take over 40% of end-to-end execution time (see §6).

More recently, some work, such as Alluxio and JuiceFS [39], tries to alleviate the high I/O cost in external storage-based solutions, by caching frequently accessed data in memory and local SSDs while keeping the less used data in external storage. Unfortunately, its interfaces are based on files, and still suffer from excessive data copying and (de)serialization overheads. For example, to share the 200GB neuraltalk2 dataset [46] from Numpy to PyTorch, conducting it over Alluxio in `.npy` files requires 656 seconds while the cost is nearly zero with in-memory sharing within the Python process.

Format. To share intermediate data, systems must agree with the data format, *i.e.,* how the data should be organized and represented. To this end, much effort has been devoted to defining some standard data formats to represent commonly used data structures, such as columnar format and ORC for table-like data, and multi-dimensional array for tensors. With these standard data formats in place, data processing systems only need to implement *adaptors* to read/write standard formats. However, there still exist some circumstances where standard data formats cannot work well. First, in many cases, there are often some differences between a standard data format and the internal data structure defined in a specific system. As a result, it takes some transformation and (de)serialization costs for the data processing system to convert data between the standard formats and its internal data structures. For example, Apache Dremio [30] uses Apache Arrow as its internal columnar data format. When facing ORC files, it requires an extra (de)serialization process to write

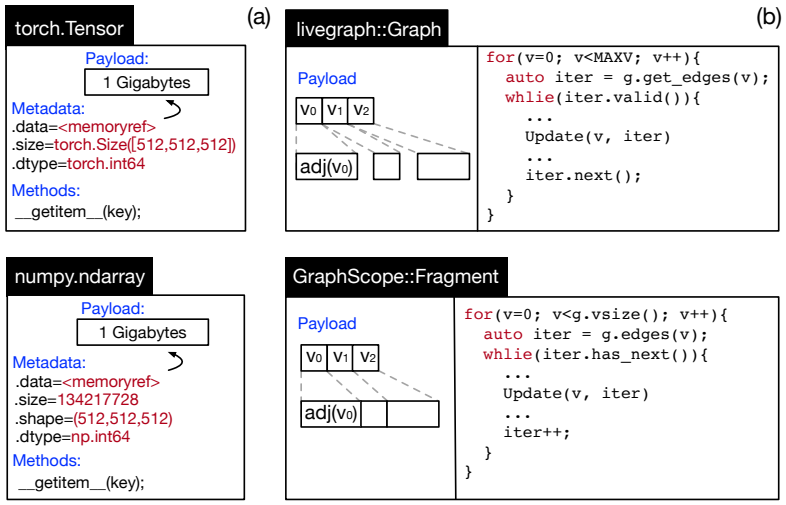


Fig. 2. The data structure commonality when (a) the layouts of tensor payloads are the same while only differ in metadata, (b) the layouts of graph differ between adj list and CSR, while the interfaces defined on them are similar.

to (read from) ORC files even though ORC is also organized in columnar formats. Second, there is still a lack of standard formats for many commonly-used data structures (*e.g.,* hashmaps and graphs). As a result, their implementations vary a lot in different systems. Under such a situation, an internal data structure has to be serialized into primitive and *flat* data formats. For example, hashmaps are usually shared as a sequence of key-value pairs, and then rebuilt from scratch by the receiver. During this process, high-level data structures lost their semantics (*e.g.,* key lookups for a hashmap), amplifying the unnecessary transformation and (de)serialization overheads.

Insights. Based on the above analysis, an underlying framework that can make data sharing across systems more efficient and flexible has to simultaneously satisfy the following requirements.

(1) On the data medium side, it should be able to provide an efficient in-memory data-sharing mechanism while supporting distributed and outsized data with external storage, and minimizing data copying, I/O and network overheads.

(2) On the data format side, it should allow developers to easily define and implement new and complex data structures, and make the data format used for intermediate data consistent with the internal data structures in data processing systems when possible, to avoid unnecessary overheads such as data transformation and (de)serialization.

3 APPROACH AND CHALLENGES

Opportunity: many intermediate data structures share some commonalities across different systems. In general, a data structure can be conceptually broken down into *payloads*, *metadata* and *methods*, as shown in Figure 2 (a). The main parts that hold the actual data are *payloads*, which correspond to a continuous memory space (*e.g.,* the 1GB buffer in `Tensor`), and *metadata* provide necessary attributes (*e.g.,* `.size` and `.dtype` for `Tensor`) to interpret the payloads. Besides, *methods* provide high-level and opaque interfaces (*e.g.,* `__getitem__(key)` for `Tensor`) for users to manipulate the data. We found the data structures in diverse data processing systems do share some degree of commonalities. They can be divided into the following categories:

(1) **Payload commonality:** The memory layouts of the payloads of the upstream and downstream systems are exactly the same, but only differ in the organization of metadata in the data structure. For example, DataFusion [28] and Polars [17] have payload commonality, since their underlying data structures both are based on Apache Arrow. As another example, shown in Figure 2 (a), both PyTorch and NumPy process tensors. The layout of payload buffers of the tensor type in these two systems are exactly the same but only differ in metadata. Compared to the payload (1GB), the space to keep their metadata is negligible. To leverage such commonalities for data sharing, one can re-create the metadata of the shared object in another system with little overhead, as long as the (de)serialization, copying and I/O over the payloads between two systems can be avoided (*e.g.,* via memory mapping). And such sharing is called *payload sharing*.

(2) **Interface commonality:** The layout of payloads of data structures of upstream and downstream systems differ from each other, but the provided interfaces follow the same logic and semantics. We can find many interface commonalities in big-data ecology. (i) The index data structures, *e.g.,* trees, hashmaps, or filters, can be implemented in different ways, while their exposed interfaces are similar, *e.g.,* `exist(key)`, `size()`. (ii) The main abstraction of Spark is a resilient distributed dataset (RDD), a collection of elements partitioned across hosts of a cluster that can be operated in parallel. Data structures that implement interfaces over RDD can be processed in Apache Spark without changing the physical memory layout. (iii) GraphScope and LiveGraph [68] represent graphs in different ways. However, they both provide adjacency-list-like interfaces on their data structures, as shown in Figure 2 (b). To leverage this kind of commonality for data sharing, one can write a wrapper around the methods of objects in the upstream system to provide the interfaces for the downstream system, instead of a thorough conversion, and such sharing is called *method sharing*.

(3) In other situations, data structures of upstream and downstream systems are totally irrelevant or systems do engine-specific optimizations such as manipulating the raw data. For example, ClickHouse [19] and Apache Doris [29] enable engine-specific optimization such as vectorization and data compression. It is intractable to share intermediate data between them without a physical conversion or non-intrusive modification. This kind of conversion often requires time-consuming data scans and is inevitable in this case.

To seize the opportunity, we propose Vineyard, an off-the-shelf distributed object store for efficient cross-system data sharing, exploiting both payload and interface commonalities for in-memory data sharing in cloud environments. At the same time, it insulates users from cumbersome data alignment boilerplate code by providing simple `put()` and `get()` APIs, allowing users to put objects in a system and get them back in another system without suffering the pain of I/Os, (de)serialization, annoying glue code, and inefficient cross-language interaction.

3.1 System Overview

Architecture of Vineyard. Figure 3 depicts the architectural overview of Vineyard. Compared with file system-based data-sharing solutions, Vineyard stores intermediate data as *objects*, and introduces a novel data-sharing mechanism to achieve high performance and flexibility while keeping the cost of integration as low as possible. Overall, Vineyard consists of seven modules:

Vineyard client (labeled by ❶). Vineyard provides a SDK supporting multiple programming languages (including C/C++, Python, Java and Rust) to integrate with data processing systems. With the SDK, an object can be retrieved from Vineyard by its ID via `get()`, and new objects can be constructed via `put()` for other systems to consume later. The object returned from the local Vineyard is a language-native object (*e.g.,* `numpy.ndarray` in Numpy) while the payloads of the object are still kept in the shared memory managed by Vineyard daemon without copy. To achieve

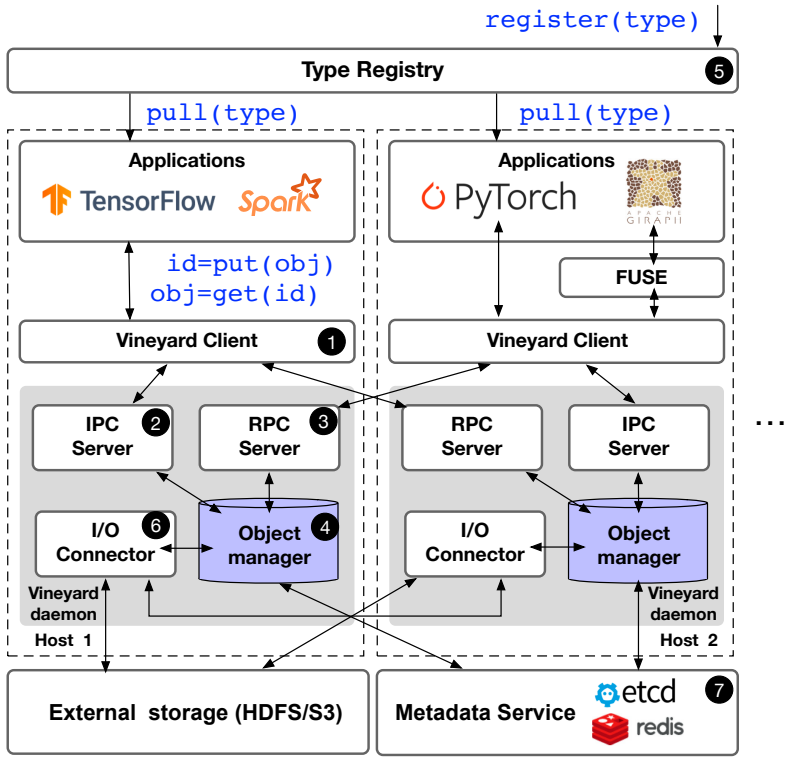


Fig. 3. System Overview.

these, the SDK includes an IPC client to Vineyard and communicates with the Vineyard daemon on the same host via the UNIX-domain socket. Besides the IPC client, the Vineyard SDK includes a RPC client as well to communicate with the Vineyard daemon on remote hosts via the TCP socket. The RPC client is used to send requests to the Vineyard daemon on remote hosts to complete the requests sent from peers such as retrieving metadata and migrating an object from/to the local Vineyard daemon.

IPC/RPC server (labeled by ❷ and ❸). An IPC server listens to the requests from the clients in the same host. It then interacts with the object manager to complete the requested tasks such as object creation, accessing and deletion. Requests are organized into queues and consumed asynchronously and concurrently. A completion reply is sent to the client when a request is done. For example, a client that sends a `GET_OBJECT` request will receive a reply with metadata dictionary and a set of file descriptors, then it can create memory-mapped blobs first and construct the object with the metadata and the blobs. An RPC server listens requests from clients in remote hosts to complete the requests sent from peers such as retrieving metadata and migrating an object from the remote Vineyard daemon.

Object manager (labeled by ❹). The payloads (blobs) of an object are stored in the in-memory Vineyard object store, which lives in a Vineyard daemon process on every host, thus different computing processes running on the same host can share data through the object store. The object manager takes charge of orchestrating these payloads such as allocation, movement, seal and deletion. Note that an object is mutable after creation and invisible to other processes until it is sealed. Once sealed, the object becomes *immutable, i.e.,* cannot be modified anymore, and

visible to other processes. There are two main reasons: (i) immutable objects suffice for most data analytics [40]; and (ii) they reduce the complexity of concurrent data accesses.

Type registry (labeled by ❺). In order to support cross-system object accessing, Vineyard provides a type registry for users to register user-defined data structures, *a.k.a.* how the payloads and metadata constitute an object, and how to reinterpret an object. Like Protobuf, Vineyard provides an IDL (intermediate description languages) named VCDL (detailed in §4.2) for users to define their customized data structures. When a system tries to `get`/`put` an object from/to the object store, it will lookup the registry to transform the registered type in Vineyard into its own internal type or vice versa. The type registry is implemented as a key-value map, with the name of the type as the key, and the concrete description of this type as the value. For example, the key may be "`foo::bar::Graph::1.10.0`", and the value is a set of VCDL files.

I/O connector (labeled by ❻). The interfaces of the I/O connector are provided to enable pluggable adapters, which allows Vineyard to load/store data from/to external storage such as local filesystem, HDFS and S3, in common formats like Parquet, ORC, HDF5 and CSV, and migrate data from remote Vineyard daemon over networks. Built on the I/O connector, Vineyard supports to spill cold objects under high memory pressure, and load from external storage back when those spilled objects been requested, as well checkpoint/reload to/from external storage for fault-tolerance. In addition, cache and prefetch are adopted to alleviate the I/O overheads in these cases.

Metadata service (labeled by ❼). Payloads of objects are stored as blobs in local shared memory while the metadata of objects are stored in a distributed, consistent key-value store (*e.g.,* etcd [22] and Redis) for object resolution. Metadata is essentially key-value pairs that describe how a group of blobs constitute an object. The metadata service guarantees the synchronization and consistency of metadata across the cluster, and metadata connects correct blobs when the objects are created, deleted, or migrated from a remote host. The consistency also allows Vineyard to support distributed objects called "*collection*", which is composed of objects on multiple hosts across the cluster as well as external storage (see details in §4.1).

Vineyard is designed to be extensible, and its functionality is divided into several loosely-coupled modules for users to customize the integration. (i) Vineyard provides an IDL named VCDL to enable users to register customized data structures succinctly. (ii) APIs for I/O connectors to exchange Vineyard objects with other storage media, file formats and networking stacks. (iii) Vineyard also provides useful modules to facilitate the integration. For example, Vineyard provides format converters for conversions of common data formats (*e.g.,* row-based tables to/from column-based ones), and FUSE (Filesystem in userspace) drivers [43], to provide file system APIs to project Vineyard objects to/from common file formats (*e.g.,* Parquet, ORC and CSV). Incorporating these modules in a workflow, Vineyard can liberate users from the tedious task of implementing and integrating such logic with individual data processing systems.

3.2 Challenges

C#1: How to share complex objects efficiently. At the center of Vineyard, the way it organizes and serves ***objects*** is crucial to its efficacy. It is challenging and non-trivial as well. There are three key criteria to meet: (i) *Support big data.* Many intermediate data is larger than the memory capacity of a single machine or even the aggregated memory of a cluster. Vineyard should support large objects, making it able to take advantage of distributed computing and external storage. (ii) *Object composability.* Many data structures, *e.g.,* graphs, are *complex*, and composed of several other objects (*e.g.,* arrays and hash indices are used in graphs). In addition, it is very common that a task incrementally creates a new object from an existing one with small changes. These two

characteristics require to express an object in a *composable* fashion. (iii) *High efficiency.* As we discussed, the performance of data sharing is largely affected by (de)serialization and extra memory movements. The object store of Vineyard needs to be designed to minimize such costs.

C#2: How to reduce integration effort. For cross-system data sharing, a downstream task does not understand the data type of the object put by its upstream tasks, unless Vineyard provides a mechanism to register the type. To this end, there are three key requirements to satisfy: (i) *Avoid intrusive integration.* Vineyard should avoid hard-coding the type into the downstream task, by modifying its source code or re-compilation. (ii) *Serve the polyglot workflow.* As a language-agnostic framework, Vineyard should ensure cross-language interoperability to achieve high flexibility and performance in ubiquitous polyglot workloads. It is challenging even with standard FFI (Foreign Function Interface), as language boundaries often add performance overheads because the cross-language interface has to marshal foreign objects. (iii) *Less boilerplate code.* Vineyard needs to insulate users from wrapping guest-language functions with annoying glue code by automatically generating the boilerplate code to achieve high flexibility.

C#3: How to work in cloud-native environments. Vineyard is designed as a daemon to serve diverse applications in cloud-native environments. Therefore, it is important for Vineyard to (i) help the workflow scheduler to consider data affinity when launching a new task and automatically prefetching the required data into memory when the downstream tasks are scheduled to a group of new hosts; (ii) handle exceptions such as shutdown, operation abort, and memory exhaustion to work robustly in production environments; and (iii) provide stable performance in a variety of scenarios such as burst objects pouring, and storing extremely small/large objects. The object store of Vineyard should hide the system complexity from users and handle the above cases gracefully.

4 DESIGN AND IMPLEMENTATIONS

In this section, we introduce key designs and implementations to overcome the above challenges.

4.1 Sharing of Complex Objects Efficiently

As discussed in **C#1** of §3.2, sharing complex objects needs to simultaneously meet the requirements of *big-data*, *composability*, and *efficiency*. First, we introduce the data model of Vineyard objects. Next, we introduce how these goals are achieved.

Data model. Vineyard objects are chunk-based, distributed, and immutable. From common practices of big data processing, we observed that large-scale data is often chunked and distributed processed in chunk-granularity. Chunk-granularity brings flexibility and efficiency, and the data model of Vineyard aligns with such a design. More specifically, the objects in Vineyard can be divided into three categories:

(1) *Blob.* A blob is a big amorphous binary data structure stored as a single entity. Blobs are typically buffers that consume consecutive memory. In Vineyard, blobs are basic units to constitute the payloads of a complex object.

(2) *Local object.* A local object is a set of blobs with metadata to describe how these blobs constitute a complex object. The metadata is a dict-like structure, with field names as keys and primitive types (*e.g.,* int, double and String), blobs, or other objects as values. Values of the same type can be repeated zero or more times for a given key, and their order is preserved (like Protobuf). Local objects are conceptually similar to the aforementioned chunks in data processing systems. For example, a local object can be a partition of a RDD in Spark [65], a part of a table in ClickHouse, or a partition of a graph in PowerGraph. “Local" means the object can fit in the memory of the

```
template <typename T>
class Iterator{
  // Get the next object in the collection.
  iterator next();
  // Get the next local object in the collection.
  iterator next_local();
};

template <typename T>
class Collection{
  // Return the reference of a specific local object.
  iterator at(int idx);
  // Return the number of objects of the collection.
  size_t  size();
  // Return the iterator of the first object.
  iterator begin();
  // Retrun the iterator of the first local object.
  iterator local_begin();
  // Return the iterator of the past-the-end object.
  iterator end();
};

class CollectionBuilder{
  // Put the object of given id to the collection.
  void put(size_t idx, ObjectID id);
  // Seal the collection to prevent mutation.
  void seal();
};
```

Fig. 4. The APIs of Collection.

local host, *i.e.,* once a client tries to `GET` a local object, Vineyard will first make sure all its blobs present in the host memory before sharing them through memory mapping.

(3) *Collection.* A collection is a set of local objects of the same type with metadata to describe how these local objects logically constitute a global object and which hosts these local objects locate. Parts of a collection's objects may be stored in external storage or on remote hosts. Vineyard provides APIs for users to pick a specific local object from a collection or just iterate on all interior objects or local ones as shown in Figure 4.

Support big data scenario. To store large objects as collections in either external storage or distributed memory, Vineyard provides corresponding mechanisms to move data between external storage and memory or between different hosts.

Moving blobs between external storage and memory. It is common that data cannot fit in memory in big data applications for users who want to persist the data as checkpoints. (i) Vineyard provides a spilling mechanism that can swap some blobs from memory to external storage or remote object store when memory pressure is high. The spilling process can be triggered automatically when either Vineyard fails to allocate new shared memory blobs or the memory usage reaches a specified threshold. With I/O connectors, Vineyard can leverage various external storage systems, *e.g.,* local filesystem, HDFS, and S3 for spilling. By default, the least recently used (LRU) policy is adopted for automatic spilling. Users can keep required objects in memory during computing with the `pin()` API to prevent object from being spilled. Users can also rely on the auto-spilling by utilizing basic `put()`/`get()` APIs, or they can fully manage the spilling process to control the location of each

blob at any time with the `evict()` and `load()` APIs to lest auto-spilling introduces unpredictable overheads and deliver guaranteed performance. Further, users are allowed to give hints when getting `Collection` to indicate the expected access patterns, *e.g.,* for a sequential scan, the I/O workers will pre-reload the spilled blobs to overlap the computation and communication and reduce the I/O overhead. (ii) For checkpointing, Vineyard provides a `save` method for users to copy the entire local object or collection into external storage. A `load` method is provided to restore an object from a checkpoint. When Vineyard clients send requests to save or load an object, the I/O worker will block other requests until the process completes to avoid data race conditions.

Fetching blobs from remote hosts. Since objects of a collection are scattered in multiple hosts, applications may access a non-local object. To support such remote object accessing, the Vineyard object manager first queries the location of the object in metadata service, then it requests the I/O connector with the object identifier and the location, and finally, the I/O connector will communicate with the data holder to access the object. The fetched objects will be stored in the local object manager and this object will be marked as local. The fetching can be triggered automatically or manually in either asynchronous or synchronous mode. After receiving the fetching request, the I/O connector will fully handle the data movement. Since the I/O connector allows developers to integrate diverse network libraries such as verbs for RDMA, gRPC [37], or DPDK [26], it is easy to achieve high performance in diverse scenarios.

Object composability To achieve composability, Vineyard adopts a *decoupled design*, where the *metadata* of *objects* are managed separately from the *payloads*. The metadata of each object (identified by an object ID) is represented as a set of key-value pairs. Taking the `Collection<DataFrame>` (a simplified dataframe) shown in Figure 5 as an example, it is represented by its unique ID "`0001`", names of its two member objects ("`0012`" and "`0013`") located on different hosts, and each object has two columns (namely "`UserID`" and "`ItemID`") with two objects IDs associated with two `Array` objects. Each `Array` object corresponds to a column of the dataframe, and is linked to a blob that represents the payload of the `Array` object. The `Array` objects can be accessed either independently or as a part of the `DataFrame` object or as an indirect part of the `Collection<DataFrame>` object. Recall that objects are immutable in Vineyard, to add a new column `Amount` to the `DataFrame` object `0012`, Vineyard simply create a new object "`0014`" to replace the old object "`0013`" with an extra column "`0120`" and reuse the columns "`1005`" and "`1006`", without copying the entire `DataFrame` object, saving memory and improving performance.

This decoupled design brings two benefits to Vineyard: (i) referring to the same blob by different objects is allowed, without worrying about data race and consistency issues since objects are immutable in Vineyard, and metadata is managed separately. (ii) It is very common that a data analytics job creates a new object O' from an existing object O with small changes and keeps both O and O' in Vineyard for later processing. Compared with duplicating the same payloads twice, incrementally creating a new object O' from an existing object O with small changes is more time and space-efficient.

Vineyard provides out-of-the-box efficient implementation for de-facto standard data structures, *e.g.,* `Vector`, `HashMap`, `Tensor`, `DataFrame` and `Graph` in the SDK, and integrates those data types with widely-adopted data processing systems, *e.g.,* `Numpy`, `Pandas`, `PyTorch`. Thanks to the decoupled design, these data structures can be directly used as the basic building blocks when users attempt to construct and share more complex system-specific data structures in their applications.

Efficiency. With the decoupled design, the overheads of creating and accessing an object in Vineyard can come from two sources, namely dealing with the payloads and metadata. For most common data types, the vast majority of the overheads are the payload part. For example, the `DataFrame` objects shown in Figure 5 can represent a table with tens of billions of records in only a handful

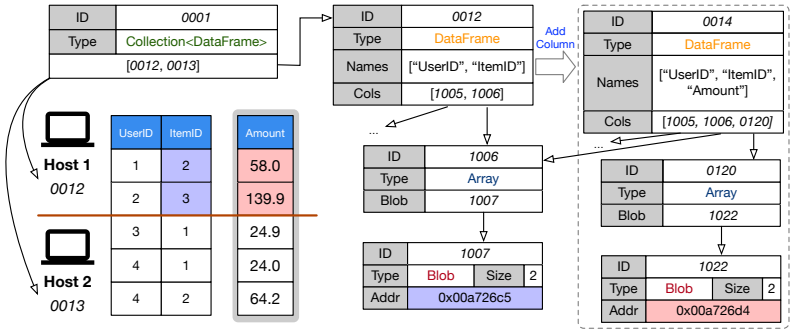


Fig. 5. Object Storage and Data Sharing.

of key-value pairs as metadata (less than 1KB) while the size of blobs is greater than tens of GBs. To achieve high efficiency, the blobs are directly memory-mapped to the clients by Vineyard for accessing payloads with *zero-copy*. Generally, the payload data are managed by individual systems before handing over to Vineyard, therefore at least one copying is required to move the data into the Vineyard object manager for the first task in each workflow. Fortunately, it is still possible to optimize out such copying. Vineyard provides a special *memory allocator* to allow dynamic memory allocation directly on its shared memory pools from processes of the data processing systems. The memory allocator serves as a drop-in replacement for the default `malloc`/`free` functions using the `LD_PRELOAD` [3], if the target systems allow the replacement of their own memory allocators. When `put()` native objects to Vineyard, the client will skip copying the blobs that already reside on Vineyard's shared memory and link metadata to the existing blobs.

4.2 Minimize high integration effort

In §4.1, we have described how objects are organized in Vineyard, and explained how Vineyard shapes the performance with the help of object composability. However, sharing objects between systems via memory mapping in real-life workflows is still intricate. (i) To exploit payload commonality, users have to construct from the metadata and payloads manually since there is no standard way to cast a type from one system to another. For example, in Figure 2 (a), users store a PyTorch Tensor as a blob into Vineyard, then `get` and reinterpret it as a NumPy `ndarray`. Such manual integration is ad-hoc and users have to write many error-prone and boilerplate codes. Worse still, such integration is not extensible: to share data between M upstream tasks and N downstream tasks, repeatedly conducting $M \times N$ integrations case by case can be daunting enough to deter the usage of Vineyard for larger workflow. (ii) To exploit interface commonality, users should first introduce the metadata and methods of the intermediate data structure into a downstream task, then implement a wrapper to enable method sharing. For example in Figure 2 (b), to share a `livegraph::Graph` to GraphScope, users first need to link LiveGraph as a library dependency during building, then forward methods of `GraphScope::Fragment` like `g.edges(v)` to methods of `livegraph::Graph` like `g.get_edges(v)`. Similarly, such integration is also not extensible. Enabling method sharing in polyglot workflows requires further effort. For example, although `Griaph.GraphType` (Java class) and `GraphScope::Fragment` (C++ class) both represent graphs in their engines respectively and have similar interfaces, nevertheless, these systems still have gaps that need users to bridge manually, *i.e.,* different memory management models in different programming languages. One way to achieve it is to re-implement the logic or implement an FFI (Foreign Function Interface) wrapper of the methods of `GraphScope::Fragment` in Java, then wrap the methods of Java-version `GraphScope::Fragment` to imitate behaviors of methods of `Giraph.GraphType`, which is hard to

```
// type.h
template <typename T>
[[shared]] class Array {
  private:
    [[shared]] Blob b;
    size_t len; // non-shared property
  public:
    [[shared]] T getItem(size_t idx) {...}
};

template <typename T>
[[shared]] class DataFrame {
  private:
    [[shared]] Repeated<String> names;
    [[shared]] Repeated<Array<T>> cols;
  public:
    [[shared]] Array<T> getCol(size_t idx){
        return cols.get(idx);
    }
};
```

Fig. 6. An example of VCDL.

maintain and optimize. Once the methods change in one version, users have to re-implement them again to keep the consistency. To be more specific, the integration burden stems from the following two aspects:

(1) No common formats for some data structures between upstream and downstream systems. Without a common format, users have to manually implement low-level memory assignment case by case.

(2) Re-implement methods for method sharing. Since upstream and downstream systems reside in different processes (or even implemented with different languages), users require to re-implement the logic of shared complex objects to enable method sharing.

Vineyard Class Description Language. To liberate users from such an interminable integration burden, Vineyard provides an IDL (intermediate description languages) named Vineyard Class Description Language (VCDL). Inspired by Protobuf, VCDL allows users to describe the shared *objects* and the *methods* over objects once, then generates the boilerplate codes automatically for different programming languages. With VCDL, users can define and implement new types of data structures succinctly. To ensure the expressivity of VCDL, we design VCDL as a C++ dialect, which keeps a majority of the features of C++, as shown in Figure 6. Like C++ class definition, a data structure defined with VCDL (*i.e.,* class) contains members and methods; objects are organized in a *composable* fashion: they can be other user-defined data structures. To enable Vineyard code generation, we only add several built-in types and directive *annotations* into the dialect.

Built-in data types. VCDL recognizes primitive types such as `int`, `double` and `String` as metadata entries. Besides primitive types, VCDL predefines object-related types such as `Blob` and `Repeated` to facilitate users to describe the components of their objects. A `Blob` represents a large binary object with a given size, and is used to describe the actual payload of a shared object. A `Repeated` is a sequence container that repeats its field any number of times (including zero), and the order of the repeated values will be preserved. For example, column names of a dataframe (line 6 in Figure 7). This concept is from Protobuf, and it is helpful to group a bunch of objects together to achieve composability. With the primitive types, predefined types for objects and the composable

```
// builders.h
template <typename T>
[[builder(DataFrame)]] ColWiseDataFrameBuilder
              : public DataFrameBuilder {
  private:
    Repeated<String> names;
    Repeated<ArrayBuilder<T>> col_builders;
  protect:
    DataFrame seal() {
      DataFrameBuilder::set_names(names);
      DataFrameBuilder::set_cols(col_builders);
      return DataFrameBuilder::seal();
    }
  public:
    void addCol(ArrayBuilder<T> col, String col_name) {
      names.push_back(col_name);
      // Enable zero-copy if possible.
      col_builders.emplace_back(col);
    }
};

template <typename T>
[[builder(DataFrame)]] RowWiseDataFrameBuilder
              : public DataFrameBuilder {
  private:
    Repeated<String> names;
    Repeated<ArrayBuilder<T>> col_builders;
  protect:
    DataFrame seal() {
      DataFrameBuilder::set_names(names);
      DataFrameBuilder::set_cols(col_builders);
      return DataFrameBuilder::seal();
    }
  public:
    void setSchema(Repeated<String> schema) {
      names = schema;
      cols_builders.resize(names.size())
    }
    void addRow(Repeated<T> row) {
      for (size_t i=0; i<row.size(); ++i) {
        cols_builders[i].push_back(row[i])
      }
    }
};
```

Fig. 7. An example of VCDL builders.

design, complex objects in data processing systems can be hierarchically mapped to Vineyard's object types and make it convenient to integrate and share with Vineyard.

Annotation `shared`. In VCDL, classes that annotated with `shared` annotation mean that systems can `PUT/GET` objects of these types to/from Vineyard for sharing. All of their members with `shared` annotation will be kept as metadata in Vineyard and shared across systems and hosts. Other members without annotations can be reconstructed from members with `shared` annotation, *e.g.*, `len` (line 6) can be re-calculated by dividing the size of Blob `b` by the size of type `T` without needing

to keep its value in Vineyard. All object types that are annotated with `shared` in VCDL must be immutable, since objects in Vineyard cannot be modified once been sealed.

Annotation `builder`. To construct an instance of the type (*e.g.,* `DataFrame`) defined by VCDL, users can implement specific classes tagged with `builder` annotations (*e.g.,* `builder(DataFrame)`), and provide methods for an upstream task to build the data structure. As shown in Figure 7, users can write different builders to support a variety of scenarios. A `seal()` method must be provided in each builder to return the final immutable object from it. Base builders that directly interacts with the Vineyard client are automatically provided by Vineyard (*e.g.,* ArrayBuilder, DataFrameBuilder). As mentioned in §4.1, Vineyard provides a memory allocator to allow memory allocation on its shared memory pools during the builder processes, to achieve zero-copy data sharing. VCDL code generation can leverage such zero-copy ability. For example, in the line 18 of Figure 7, the `ArrayBuilder` will check if the column is already in the Vineyard's shared memory poll, and avoid copying when possible. Builders can be nested, since Vineyard objects are composable (*e.g.,* member `col_builders` of class `ColWiseDataFrameBuilder<T>` in Figure 7).

Code generation. With VCDL, Vineyard can generate most of the glue code required by integration. Thus, users can focus on the data structure they want to share. Specifically, the benefits of Vineyard code generation are three-fold: (i) produce common formats; (ii) generate boilerplate code; (iii) enable cross-language optimizations.

Producing common formats. To get an object put by an upstream system, the downstream system should understand the data type of the object. For systems that adopt C++ as their programming language, they can directly invoke the methods of data structures defined in VCDL, as VCDL is a C++ dialect. To work with systems written in other languages, Vineyard will generate a new class definition in their languages to access the methods and implementations defined in VCDL. VCDL wraps classes described in VCDL as native classes in multiple guest languages instead of re-implementing them in another language. It first leverages libclang [42] to get the ASTs (Abstract Syntax Trees) of VCDL classes to figure out the classes, members, and methods that need to be exposed to the guest languages for data access and creation. It then maps each class annotated with `[[shared]]` to a native data type (*e.g.,* `interface` for Java, `struct` for Rust) in the guest languages, and generates an FFI wrapper for each method that is annotated with `[[shared]]` in VCDL. Currently, VCDL supports C/C++, Java, Python and Rust as guest languages. When adding support for a new language, VCDL only requires to develop a code generator that handles the primitive type mappings and can generate classes and method wrappers in the guest language from the ASTs generated by libclang.

Generating boilerplate code. As discussed above, without Vineyard's code generation, users who want to share intermediate data between systems have to write a lot of boilerplate code like field getter and setter, which is common in cross-language wrapper generators *e.g.,* SWIG [18]. With the VCDL code generator in place, Vineyard first generates a common type of the class with annotation `shared`, then generates getters and setters for members tagged with annotation `shared`. Given a common type between upstream and downstream systems, users can always get the type defined in VCDL without re-implementing them manually. Moreover, the VCDL code generator can also handle the generic types (*i.e.,* the `template` in C++ and `Generics` in Java) across languages. Generics is an essential language feature for the implementation of data processing systems. However, the programming principles of generic may vary a lot in different programming languages (*e.g.,* C++ and Java), and it is non-trivial to generate safe and efficient cross-language interfaces. Instead of simply mapping C++ template instantiations to parameterized types in Java, the VCDL code generator generates a unique `class` in Java for each instantiation of the same C++ template to avoid type errors in native code.

Enabling cross-language optimizations. To ensure the performance, the VCDL code generator enables optimizations to be applied across the boundaries between languages. The VCDL code generator reads annotated VCDL files and creates wrapper code (glue code) to make the corresponding C/C++ libraries available to other guest languages or to extend C/C++ programs with a scripting language. Code generated for LLVM-based languages (*e.g.,* C/C++, Rust) can be compiled into LLVM bitcode (IR), and can be optimized and linked at the IR level with the help of the LLVM LTO (link-time optimizations). For JVM-based languages, translated functions can be optimized (*e.g.,* inlining) with the code of the target data processing system by JVM JIT. In addition, it also allows a large proportion of simple yet performance-critical routines (*e.g.,* iterators) can be translated to efficient JVM bytecode. Fortunately, with the `sun.misc.Unsafe` mechanism of Java, large payloads/blobs can be also memory mapped off-heap and efficiently accessed from JVM. If functions cannot be translated, Vineyard will fall back to JNI (Java Native Interface) calls.

Discussion. With VCDL, integration is intuitive. Users first define required data structures (types) in VCDL, then they can directly implement their applications on generated data structures, or convert the defined data structures to the existing native types in custom wrappers. With payload sharing, users just need to "cast" the generated types to the native types. Such casting is zero-copy and only requires to manipulate their metadata. With method sharing, users need to wrap methods of native types with methods of generated types, which is also zero-penalty. If users fail to provide VCDL files for some reasons, Vineyard provides a FUSE driver that encapsulates the Vineyard client to provide file system APIs, as shown in Figure 3. Since the FUSE driver provides filesystem interfaces, objects have to be serialized as buffered bytes when users read/write objects from/to Vineyard, while buffered bytes will be automatically deserialized as objects when the file handle are closed.

4.3 Working in cloud-native environments.

In the cloud-native era, big-data analytics jobs are usually deployed as containerized applications, which are orchestrated by workflow engines (*e.g.,* Apache Airflow [27], Dagster [21], Kedro [9]), and managed by Kubernetes. Vineyard is deployed as a `Deployment` on Kubernetes and managed by the Kubernetes Operator [14] `vineyard-operator`. After applications have been submitted to a Kubernetes cluster, Vineyard will place the application pods near where its required inputs are located. Besides, by integrating with the workflow orchestration engine, Vineyard archives application-level fault tolerance. In this subsection, we will discuss how Vineyard addresses challenges proposed in **C#3** in §3.2 in cloud-native environments.

Locality awareness on Kubernetes. Vineyard archives alignment between application workers and their required inputs by integrating with the scheduler of underlying cloud-native infrastructure, *i.e.,* Kubernetes. The collections of Vineyard objects that will be shared across hosts are abstracted and organized as Custom Resource Definitions (CRDs) [13] in Kubernetes, making them observable and accessible from the scheduler component of the Kubernetes cluster. Developers can specify the required input objects for a task in the specification by the `k8s.vineyard.io/required`, which indicates the prerequisite tasks that generated the required inputs for this task. Once the upstream tasks have created outputs as CRDs, the current task itself will be ready for being scheduled. As shown in Figure 8, the task *B* requires a CRD of type `DistDataFrame<int>` generated by task *A*. Once the CRD generated by task *A* is available, task *B* will be ready for being scheduled..

Vineyard implements a data-aware scheduling policy in a scheduler plugin for Kubernetes [5]. Specifically, given a task, Vineyard partitions required collections into local object collections and assigns collections of local objects to workers of this task. The scheduler plugin then inspects the location metadata from CRDs of those local objects and assigns the highest priority to the host

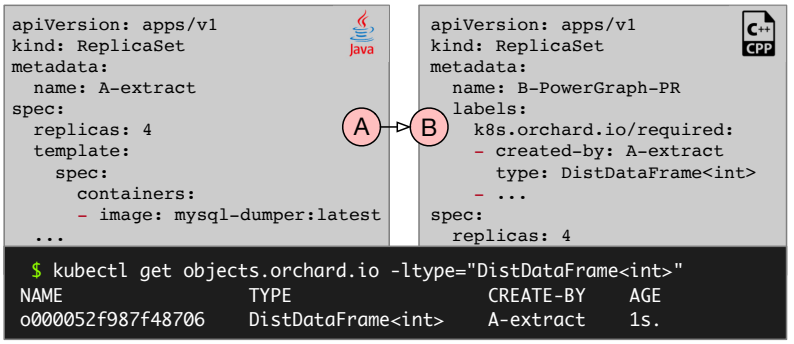


Fig. 8. Program a Workflow DAG on Vineyard

where the required local objects are located for each worker of this task. In this way, the scheduler can respect the locality of the required inputs and reduce the data transfer costs. As the cluster resources are dynamically changing and tasks may have their own access pattern on distributed collections, remote data accessing is unavoidable. Vineyard's `Collection` abstraction in §4.1 fits such scenarios, and required objects will be migrated from remote instances by Vineyard when needed. In environments where external Kubernetes scheduler plugin is not allowed to be deployed, Vineyard provides a command line tool `vineyardctl` which accepts the workflow specification in YAML format and injects the node affinity annotations into the specification to route tasks as near to where their required inputs been placed as possible.

Vineyard implements workflow isolation with `Sessions`, which is aligned with the Namespace mechanism in Kubernetes. Multiple sessions can be created in the same Vineyard cluster and each session can be connected via its own UNIX-domain socket. Vineyard clients can only see and manipulate objects in the connected session. When a workflow finishes and intermediate data can be dropped, removing a session will clean up all objects in it.

Fault tolerance and data consistency. Failures are inevitable for big data analytics in a cloud-native environment. Designed as an object store for intermediate data, Vineyard does not replicate objects but provides the `save(ObjectID,path)` API and users can insert checkpoint tasks into their workflows. Vineyard has been integrated with the failover mechanism of workflow orchestration engines. When application failure happens, the results produced by the last succeeded step are still kept in Vineyard and the workflow scheduler decides whether to reload data from checkpoints with the `load(path)` API and rerun from the failed steps or restart from scratch.

Vineyard maintains an object dependency tree in metadata service and keeps a periodic heartbeat between instances. When Vineyard instance failure happens the heartbeat connection will be lost, the failure would be detected by other instances and all objects that depend on objects resides on the failed instance will be dropped recursively across the cluster. Tasks that get involved with the garbage-collected objects will be marked as failed and the workflow scheduler will decide whether to reload data from the last checkpoint and rerun affected tasks or propagate the error to users. Vineyard uses external key-value storage (*e.g.,* etcd [22], Redis, or Kubernetes CRDs) as the metadata service backend, which is ACID-compliant and supports high-availability deployment. Thus, the consistency and availability of metadata will not be affected by Vineyard instance failures.

Usage patterns of big data systems. Big data analytics usually involves two kinds of objects: several large objects and many small objects. For extremely large objects, Vineyard provides the `Collection` abstraction mentioned in §4.1 where large objects are organized as a sequence of local objects and can be handled by Vineyard as long as the local object can be fit into the memory of

a single machine. Local objects that are not accessed by the data processing tasks can be spilled to external storage and will be automatically reloaded when been requested. Vineyard uses memory mapping to share a local object between clients and the server, thus the runtime overhead of `get()` is a constant no matter how large the local object is. For extremely large volumes of small objects, *e.g.,* millions of scalars and short bytes, Vineyard inlines their payloads into metadata to mitigate shared memory fragmentation, and Vineyard metadata service can process many metadata requests simultaneously. Moreover, in Vineyard only objects that need to be shared between tasks will be persisted to the metadata service backend to reduce the overhead of creating too many entries. Objects that are only used by a single task will be kept in the local Vineyard server and will not be synchronized across the cluster.

5 USE CASES

This section describes some use cases for integrating Vineyard into data processing systems. Recall that we introduce three types of commonalities for intermediate data across systems in §3. We first choose three representative examples, each for one type of commonality, to show how Vineyard can be easily and non-intrusively integrated to bridge different data processing systems.

Dask and PyTorch. Dask is a widely used package for distributed scientific computing. PyTorch is an efficient distributed machine-learning framework that operates on Numpy-like tensors with built-in GPU and `autograd` support. It is common to use Dask for data preprocessing and PyTorch for model training in machine learning applications. To share data from Dask to PyTorch, Dask generates `Collection<VYTensor>` in Vineyard, and PyTorch consumes the `Collection` batch-by-batch for training. Specifically, Dask represents each local object in the `Collection` as NumPy's `ndarray`, which shares the same memory layout with PyTorch's `Tensor`. Therefore, the integration is straightforward. (i) We implemented a wrapper for `numpy.ndarray` to build `VYTensor` by coping the blobs in `numpy.ndarray` to Vineyard's shared memory as blobs, and construct `VYTensor`'s metadata using properties of the `numpy.ndarray` object. If allocator hooks are enabled, the copy of blobs can be further eliminated as the `numpy.ndarray` already resides in Vineyard's shared memory. (ii) We implemented a wrapper to construct `torch.Tensor` from `VYTensor` using the blobs and metadata, without deserialization, data transformation and memory coping, as `VYTensor` and `torch.Tensor` share payload commonality. (iii) Further, we implemented two wrappers to handle collections, where the former builds a Dask's distributed tensor as `Collection` while the latter resolves a `Collection` to a `torch.utils.data.Dataset`. The integration takes 30 lines of Python code for Dask and 58 lines of Python code for PyTorch.

GraphScope and GraphX. Both GraphScope and GraphX are distributed graph processing systems. GraphScope provides an efficient graph data structure (`Fragment`) implementation and a set of standard built-in algorithms with superior performance, whereas GraphX supports various user-defined algorithms and has been deployed earlier and widely deployed. It is common in our organization where GraphScope is used to execute some standard algorithms and GraphX is deployed for user-defined algorithms in a single workflow. The graph data structure in GraphScope and GraphX has completely different memory layouts but roughly the same APIs (interfaces), *e.g.,* `Fragment.Vertices()` and `Graph.vertices()`. These two graph processing systems are implemented in different programming languages, in C++ and Scala, respectively. As described in §4.2, we implemented a thin C++ wrapper over `VYFragment` for GraphScope, and a wrapper over the generated JNI bindings by VCDL to align with GraphX's APIs. Thanks to optimizations described in §4.2 that ensure the efficiency of the JNI bindings, the integration finally removes the cost of graph data transformation and archives 2∼10× speedup in running GraphX algorithms directly on GraphScope's graph structure. To align the graph data structure in these two systems

with Vineyard, it takes 108 lines of C++ code for GraphScope and 208 lines of code for GraphX for implementing the straightforward wrappers over `VYFragment` defined using VCDL.

Mars and ClickHouse. Mars is an open-source framework that scales Numpy and Pandas computation engines to large clusters. ClickHouse is a distributed SQL engine for running interactive queries over big data. The former executes analytics tasks described operations over `VYDataFrame` using Python, and the latter is utilized for ad-hoc SQL queries. Without Vineyard, to bridge these two systems, users usually need to save the result from Mars to external storage, then load it back to ClickHouse for mixed computations, incurring large I/O and data transformation costs. Mars has been integrated with Vineyard and each chunk of its distributed dataframe is a `VYDataFrame` object, composing a `Collection` across the cluster finally. However, due to the highly customized data layout in ClickHouse's `MergeTreeTableEngine`, it is non-trivial to integrate a plain columnar data structure into ClickHouse. Vineyard provides a FUSE driver which provides a filesystem view for objects stored in Vineyard, where a `VYDataFrame` can be read as a Parquet or ORC file using the standard POSIX file system interfaces. Upon the FUSE driver, ClickHouse can directly consume those `VYDataFrame` objects as Parquet files using its built-in Parquet reader, avoiding the cost of expensive I/O between external storage, without any modification to existing data processing engines. Further, ClickHouse can pass chunk access patterns (*e.g.,* sequential scan) as hints to Vineyard using the standard `ioctl()` API [2] to enable Vineyard to preload chunks that will be accessed shortly from external storage back to memory when spilling happens, improving the overlapping of computation and I/O time to archive better performance.

6 EVALUATION

In this section, we report the performance of Vineyard over real-life complex data analytics jobs, micro benchmarks about optimizations, the Vineyard integration with various data processing systems, as well as our experience and observations of deploying Vineyard in a production environment. The test bed is a Kubernetes cluster with over 1000 hosts. Each host is equipped with 2 Intel 8269CY CPUs, 768GB RAM, 1TB SSD, and 50G NIC with RoCE support.

Data-intensive analytics jobs. We choose three real-life workflows to evaluate Vineyard:

(1) *A node classification job* on citation network. Given the *ogbn-mag* data [38], we build a heterogeneous citation network from Microsoft Academic Graph. To predict the class of each paper, we building a machine learning pipeline and apply both the attribute and structural information of graph data. The workflow involves graph analytics, graph neural network inference and subgraph extraction, and consists of the following steps: (i) defining graph schema and loading the graph; (ii) running graph algorithms (K-core and triangles counting) in libgrape-lite [24] to generate more features for each vertex in the graph; (iii) executing a GNN model for vertex classification in GraphLearn [66]; and (iv) querying the concerned subgraph structure in GAIA [53].

(2) *A customer revenue prediction job.* Based on user visiting behaviors from Google Play Store [1], this job leverages a random forest model after several data cleaning steps to predict the per-user revenues. The workflow contains: (i) cleaning the dataset (*e.g.,* dropping missing values) using Presto SQL and combining necessary feature columns to a feature table; (ii) predicting the per-user revenues using a pre-trained random forest model; and (iii) adopting the prediction results with further data analysis like the correlation between revenues and user devices. This workflow consists of 16 tasks which involve Presto [56], Pandas [16] and scikit-learn [55].

(3) *A fraudulent user detection job.* As shown in Figure 1. Given a set of transaction records (*i.e.,* user-item pairs), the attributes of users as well as some users marked with *known fraud* labels, this job aims to detect more users involved in fraud. The workflow consists of: (i) creating a bipartite

Table 1. Results for End-to-end time & Data sharing time& Memory footprints.

Jobs	Statistics	End-to-end time (s)			Data sharing time (s)			Memory footprint (GB)		
	# Size (GB)	S3-like	Alluxio	Vineyard	S3-like	Alluxio	Vineyard	S3-like	Alluxio	Vineyard
Graph Processing	255.5	5473.9	3978.4	**1403.1**	1797.5	411.4	**6**	1474.7	1684.9	**1219.2**
Revenue Prediction	170.6	4472.4	1088.5	**439.1**	4151.2	767.3	**117.9**	311.8	526.9	**290.1**
Fraud Detection	617.9	16669.2	4401.6	**1767.7**	15137.6	2869.2	**235.3**	1120.8	1738.7	**785.4**

graph from the transaction records, where vertices are users and items, and edges represent transaction relationships; (ii) running graph algorithms such as PageRank/SimRank in libgrape-lite [24] as new vertex attributes; (iii) selecting influential users and tailoring attribute tables in Pandas [16]; and (iv) training a deep learning model to predict more fraudulent users in PyTorch.

Datasets. For job (1), we use a heterogeneous network *ogbn-mag* [6]: it contains 4 types of entities, as well as four types of directed relations connecting two entities. For job (2), we apply the dataset from the Kaggle contest "Google Analytics Customer Revenue Prediction" [1] which consists of a set of visiting records including visitID, location, device, time, visiting count, and other extra attributes. For job (3), we employ three real-life datasets from production which consist of all transaction records in a period of 15 days from Alibaba. Table 1 summarizes the statistics of the dataset in jobs (1) to (3). All datasets are stored as compressed CSV files.

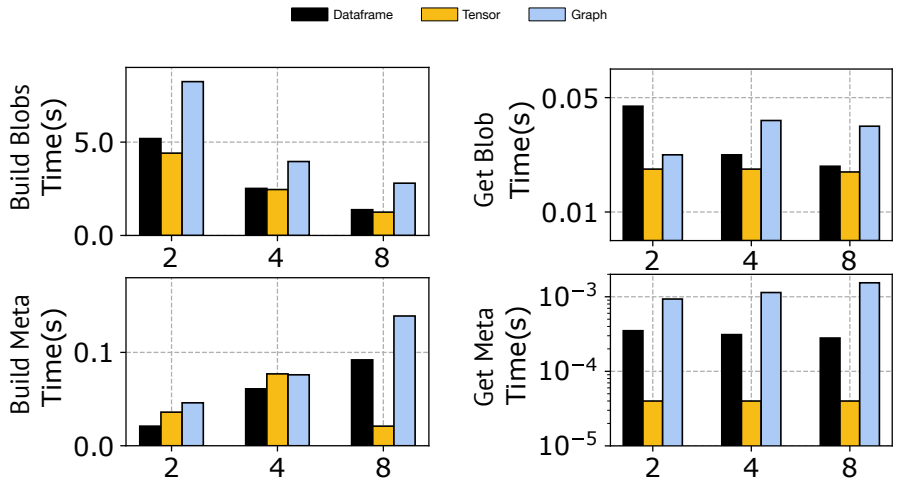
Baselines. We compared Vineyard with the following baselines: (i) *S3-like* object store service while all intermediate data is directly stored as files, and (ii) *Alluxio*, which works like a memory cache for a file system which transparently caches frequently accessed data in memory, to improve throughput and reduces I/O costs.

Exp-1: End-to-end and data-sharing performance. We first evaluate the end-to-end performance and data-sharing costs of Vineyard, and compare with its competitors on the cluster when there were no other data analytics jobs running. Table 1 reports the end-to-end performance of the three jobs. The end-to-end time means the runtime of the whole workflow, starting from loading data from files to outputting the final results. Here data-sharing costs include those for data (de)serialization, I/Os, and data migration across hosts when the data processing system is not co-located with its inputs. In this experiment, we used 8 workers for each job.

(1) Overall, on the end-to-end performance side, Vineyard achieves 3.9∼10× speedups compared with S3-like, 2.5∼2.8× speedups compared with Alluxio. This is mostly due to the effectiveness of Vineyard in reducing data-sharing costs for complex and nested data structures. Note that job (1) shares the single graph in the whole workflow, rendering data-sharing time small with Vineyard. Compared with its competitors, Vineyard achieves a 28.8× speedup on average, up to 68.4×, on cross-system data sharing.

(2) Vineyard requires less memory footprint in all jobs. Vineyard uses 70%∼90% memory compared with S3-like, and 45%∼72% memory compared with Alluxio. This benefit mainly comes from the memory mapping of Vineyard, which enables the zero-copy fashion for data sharing. Alluxio requires more memory due to its file-cache mechanism, while copying inevitably incurs (de)serialization costs.

Exp-2: Put and get objects. We further evaluated the data-sharing efficiency of Vineyard with three widely used data structures: (1) a dataframe containing 6 columns and 351 million rows, (2) a tensor with 2.1 billion elements, and (3) a graph of 60 million vertices and 167 million edges. Each element of the dataframe object and the tensor object is stored as an `int64`. A graph has two main data structures: a `HashMap` to index its vertices and a sparse matrix in CSR (compressed sparse row) format for its edges. Each object takes around 16GB space when loaded into main memory. We partitioned these objects into 2, 4, and 8 chunks and evaluated the time of building a collection to Vineyard



(a) Building Objects (b) Getting Objects

Fig. 9. Efficiency of data sharing

and getting it back from Vineyard. Note that we turn off the allocator when building the blobs in this experience, otherwise the building blob time would be nearly zero. This is because there is no data-copy cost when the allocator is enabled, and the only extra cost of malloc is less than $1\mu s$ on average which is comparable to the state-of-the-art malloc libraries. More specifically, the malloc cost is *constant* to either the size of a blob or the structure of data. Therefore, the building blob time is generally ignored when the allocator is on. The results are reported in Figure 9. We find the following:

(1) We observe that on average it takes over 99% of building time to save the blobs to the shared memory of Vineyard. The time of saving metadata to the object store is quite small, *i.e.,* less than 0.13s in all cases. Since there is no serialization cost when building these objects, storing objects in Vineyard only involves memory copying and is very efficient (see §4.1).

(2) The building time of objects scales very well. It takes 1.4s, 1.3s, and 2.9s to build build a `Collection` of 8 local objects distributed evenly across hosts for the tested dataframes, tensors, and graphs, respectively. The graph building time is larger than others, as each fragment of the graph object needs to build the same global vertex map, which alone accounts for around 5GB of memory space.

(3) Compared with the building time, the time of getting an object from Vineyard is *negligible*, *i.e.,* less than 0.9% for all cases. On average, it takes 0.034s, 0.024s, and 0.038s to get the dataframe, tensor and graph objects, respectively. This is because getting objects is conducted in a zero-copy fashion via memory mapping, thanks to the decoupled design of objects.

(4) The extra overhead by Vineyard for fetching objects from remote hosts is small as well. As discussed in §4.3, in many cases, fetching remote objects are minimized with locality-aware scheduler plugin. When such fetches are necessary, we measured the cost of `fetch()` of $100K$ tensor objects sizes ranging from 8MB to 10GB over UDP, TCP and RDMA I/O connectors. Our evaluation shows that Vineyard can fully utilize the networking with little overhead (more than 94.61% utilization of the network bandwidth reported by the respective `iperf3` (`iperf-rdma` for RDMA) tests).

Exp-3: Incremental object creation. We next evaluated the time and space efficiency of incremental object creation. Varying the number of rows from 88 million to 352 million of an existing dataframe with five columns, we built a new dataframe by inserting one new column into existing

one. Varying the number of vertices from 148 million to 2 billion of a simple graph with an average degree 5, we added 2 properties for each vertex and convert it to a property graph [53]. Each dataframe or graph object has 8 chunks. We compared incremental object creation IncBuild with a baseline ReBuild that re-builds new objects starting from scratch.

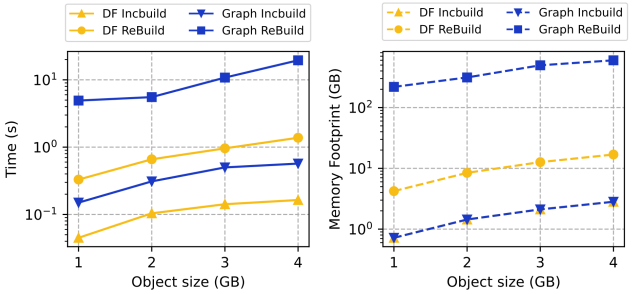


Fig. 10. Effectiveness of Incremental Object Creation.

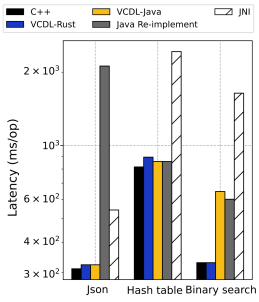


Fig. 11. Performance of VCDL code generation for different languages.

As shown in Figure 10, IncBuild outperforms ReBuild in time efficiency by 7.2× and 26.3× on average, up to 8.4× and 33.9× for dataframe and graph, respectively. To build the new dataframe and graph objects, IncBuild on average saves 83.3% and 99.4% memory space usage since Vineyard objects is composable and it can safely reuse and share existing blobs with the old dataframe and graph objects (see §4.1). Moreover, the saved time and space get larger as the size of the original data volume scales, indicating the necessity of composability and incremental object creation when facing large-scale data.

Table 2. Comparison of integration cost & performance with different systems.

		S3-Like		Alluxio		Vineyard			Vineyard FUSE	
VCDL Type	Systems	C_{in}	C_{out}	C_{in}	C_{out}	C_{in}	C_{out}	LOCs	C_{in}	C_{out}
`VYTensor`	NumPy	1.0	0.5627	0.1057	0.2794	**0.0026**	**0.0019**	32 *	0.0949	0.1559
	PyTorch	1.0	0.3859	0.0669	0.1854	**0.0029**	**0.0025**	58 *	0.0746	0.1290
	XTensor	1.0	0.6881	0.0908	0.3195	**0.0045**	**0.0588**	46 +	0.0734	0.1782
`VYDataFrame`	Arrow	1.0	0.6360	0.5964	0.6345	**0.015**	**0.033**	148*	0.1279	0.3610
	Pandas	1.0	0.7708	0.7516	0.7671	**0.0139**	**0.023**	68 *	0.5288	0.4434
	HDataFrame	1.0	0.5495	0.2282	0.2569	**0.0035**	**0.1399**	121+	0.2187	0.3206
`VYFragment`	GraphScope	1.0	1.9529	0.4002	1.8365	**0.0038**	**0.0323**	108*	0.3721	1.7891
	PowerGraph	1.0	0.6442	0.8128	0.6359	**0.0058**	**0.2334**	144+	0.8323	0.6010
	GraphX	1.0	0.3440	0.8049	0.1985	**0.0039**	**0.1148**	208+	0.6700	0.1783

[1] When benchmarking Alluxio, we preload data to `MEMORY` cache for read and use `MUST_CACHE` as write policy for its best performance.

[2] Data processing systems are implemented in different programming languages, including Python, C++ and Java.

[3] Vineyard is integrated into data processing systems by either payload sharing (annotated with ∗) or method sharing (annotated with +).

Exp-4: Integration cost and efficiency. To evaluate the cost of integration and performance of Vineyard, we compared the integration cost (measured by lines of source code change [50]) and performance (measured by execution time) between Vineyard and other solutions. We chose three types of data structures, *i.e.,* tensors, dataframes, and graphs, and three data processing systems for each data type to evaluate the performance of exchanging intermediate data between those

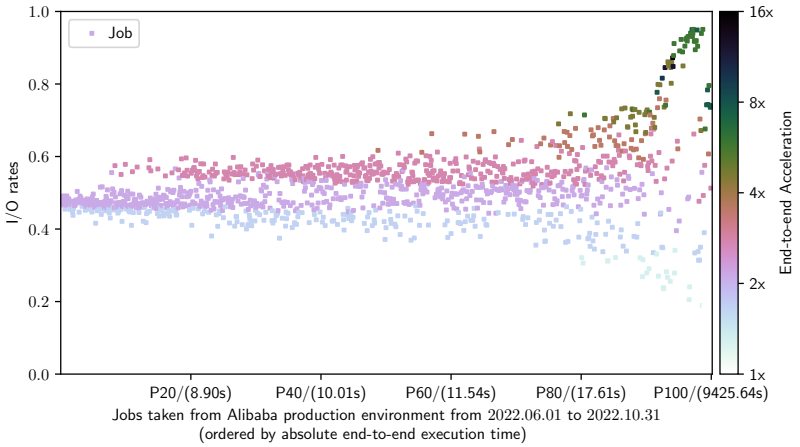


Fig. 12. Deploying Vineyard in production environments.

systems over different data storage mediums. For our baseline *S3-like* object store and *Alluxio*, we use formats that are already supported by those data processing systems as the intermediate data formats, *i.e.,* `HDF5`, `Parquet`, and `CSV`, respectively. When integrating with Vineyard, we use VCDL to define `VYTensor`, `VYDataFrame`, and `VYFragment` as the intermediate data formats.

We deployed a micro-benchmark with around 10GB intermediate data for each data type. Table 2 shows the normalized execution time of data sharing as well as lines of code needed for integrating the above data types with data processing systems. C_{in} represents the cost of `get()` from Vineyard and C_{out} represents the cost of `put()` into Vineyard. We observed the following: (i) Vineyard outperforms baselines in both `get()` and `put()`. With payload sharing, both `get()` and `put()` only involve metadata manipulation without memory copy for blobs. While with only method sharing, the `get()` is still fast, as no data (de)serialization or transformation is needed. However, the `put()` needs extra costs to build system-specific data structure into Vineyard. (ii) Even without extra integration, the FUSE driver can still help reduce the cost of data sharing, compared to exchanging intermediate data over external storage or file system-oriented caching systems, as the filesystem views are carefully maintained in memory by Vineyard.

Further, we take Java as the target to evaluate the performance of cross-language optimizations in method sharing mentioned in §4.2. Figure 11 shows: (i) the overhead of cross-language method sharing is low compared to naive FFI calls (*e.g.,* VCDL-Java vs JNI); and (ii) calling into a foreign language is more efficient, compared with implementation in native languages (*e.g.,* VCDL-java vs. Java).

Exp-5: Vineyard in production environments. We have deployed Vineyard in production environments and obtained a huge gain for optimizing intermediate data-sharing time in various data-intensive workflows. Among them, many consist of steps including similarity search, graph analytics, SQL and deep learning, and Hive-like data-warehouse tables are used as external storage. Those workflows are scheduled to a large Kubernetes cluster as around 40, 000 jobs within a day. The scale of steps in these jobs ranged from 1 to 4000 workers, and the size of intermediate data varies from several MBs to hundreds of GBs.

Figure 12 demonstrates the statistics of the I/O cost ratio in end-to-end execution time for intermediate data sharing uniformly sampled from our daily production environments. With data-warehouse tables, loading input data from tables and saving results into tables usually consume over 40% of the total execution time, and the number can be up to 95.06% in some cases when the end-to-end execution time gets longer. The I/O cost is high for both short-lived and long-running jobs and sharing intermediate data is a common concern in big-data analytical applications. We

then measured the acceleration after introducing Vineyard into intermediate data sharing. As shown in Figure 12, there is a stable acceleration effect for both short-lived and long-running jobs, and the end-to-end execution time can be accelerated up to 9.48×.

7 RELATED WORK

single-system vs. multi-system. Existing systems like Ray [49] and Spark [65] aim to provide simple and universal APIs for diverse workloads. However, previous work [67] has shown that workload-specific systems may outperform these general systems over 100× for applications like graph computation. Thus, multi-system is a necessity for handling read-life complex data analytics workflows.

Cross-system data sharing. Various distributed storage systems, *e.g.,* HDFS [57], S3 [10] and Alluxio [40], are often applied to share immediate data in multi-system. However, they suffer from huge (de)serialization and/or I/O overheads [11, 32, 60]. Thus, some recent work, *e.g.,* Apache Arrow Plasma [31], enables zero-copy data sharing for some data structures. However, it cannot cover diverse data structures and failed to scale out to share distributed data that cannot be fit into the memory of single machine.

Multi-language interfaces. The idea of providing multiple language interfaces by generating boilerplate code from a description language has been widely adopted in many projects like Protobuf and Thrift [58]. There are also attempts showing the benefits of translating LLVM bitcode to JVM bytecode [54, 59]. Another research direction in polyglot programming is compiling programs in different programming languages to the same IR and executing on the same runtime [62]. However, the compilation barrier across languages still exists. Compared with existing work, Vineyard proposed a novel way to handle *generics* when generating multiple-language interfaces and made the integration easier. Instead of compiling all LLVM bitcode to JVM bytecode, Vineyard only translates certain instructions that can be properly handled by the JIT compiler and gains performance improvement.

Data-aware scheduling. Data-intensive analytics workflows (jobs) suffer from expensive data shuffle costs. Especially on Kubernetes, the scheduler routes tasks to nodes only based on the computing resources consideration, lacking information about data exchanging between tasks. To solve this problem, some recent work, *e.g.,* Fluid [12], binds pods to nodes based on the volumes information that mounts the required inputs, in order to ensure co-locating between computation and data. However, it is ill-suited for dynamic resources on multi-tenant clusters where co-locating cannot always be fulfilled. Instead, Vineyard applies an adaptive switch mechanism to optimize the task scheduling *on-the-fly* to minimize the data shuffle costs.

8 CONCLUSION

Specialized data processing systems targeted to specific workloads often provide high performance. However, sharing intermediate data from one to another becomes a major bottleneck. To alleviate high cross-system data-sharing costs, we present Vineyard, a high-performance, extensible, and cloud-native object store. With Vineyard, data sharing can be efficiently conducted via payload sharing and method sharing. It also provides an IDL named VCDL to facilitate the integration. As a cloud-native system, Vineyard is designed to be native in interacting with container orchestration systems, as well as achieving fault tolerance and high performance in production environments. Vineyard is open-source and under active development. It is already integrated or being integrated with over 20 data processing systems. We hope Vineyard can be used as a common component for data-intensive jobs and connect diverse big-data engines.

REFERENCES

[1] 2019. Google Analytics Customer Revenue Prediction. https://www.kaggle.com/c/ga-customer-revenue-prediction.

[2] 2023. `ioctl(2)` — Linux manual page. https://man7.org/linux/man-pages/man2/ioctl.2.html.

[3] 2023. `LD_PRELOAD` — Linux manual page. https://man7.org/linux/man-pages/man8/ld.so.8.html.

[4] 2023. Data-intensive computing. https://en.wikipedia.org/wiki/Data-intensive_computing.

[5] 2023. Kubernets Scheduling Framework. https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework.

[6] 2023. Node Property Prediction. https://ogb.stanford.edu/docs/nodeprop/.

[7] 2023. Production-Grade Container Orchestration. https://kubernetes.io.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[9] Sajid Alam, Nok Lam Chan, Gabriel Comym, Yetunde Dada, Ivan Danov, Deepyaman Datta, Tynan DeBold, Jannic Holzer, Rashida Kanchwala, Ankita Katiyar, Amanda Koh, Andrew Mackay, Ahdra Merali, Antony Milne, Huong Nguyen, Nero Okwa, Juan Luis Cano Rodríguez, Joel Schwarzmann, Jo Stichbury, and Merel Theisen. 2023. *Kedro*. https://github.com/kedro-org/kedro

[10] Inc. Amazon Web Service. 2022. Amazon Simple Storage Service: Object Storage built to retrieve any amount of data from anywhere. https://aws.amazon.com/s3/.

[11] Ganesh Ananthanarayanan, Ali Ghodsi, Andrew Warfield, Dhruba Borthakur, Srikanth Kandula, Scott Shenker, and Ion Stoica. 2012. Pacman: Coordinated memory caching for parallel jobs. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 267–280.

[12] Fluid Authors. 2021. Fluid: elastic data abstraction and acceleration for BigData/AI applications in cloud. https://fluid-cloudnative.github.io.

[13] Kubernetes Authors. 2022. Kubernets Custom Resources. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources.

[14] Kubernetes Authors. 2022. Kubernets Operator Pattern. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[15] NumPy Authors. 2022. NumPy: The fundamental package for scientific computing with Python. https://www.numpy.org/.

[16] Pandas authors. 2022. Pandas: Python Data Analysis Library. https://pandas.pydata.org/.

[17] Polars Authors. 2022. Polars: Fast multi-threaded, hybrid-streaming DataFrame library. https://www.pola.rs.

[18] SWIG Authors. 2019. SWIG: Simplified Wrapper and Interface Generator. https://github.com/swig/swig.

[19] Inc. ClickHouse. 2022. ClickHouse: Fast Open-Source OLAP DBMS. https://clickhouse.com/.

[20] Dormando. 2022. memcached: a distributed memory object caching system. https://memcached.org/.

[21] Inc. Elementl. 2023. Dagster: An orchestration platform for the development, production, and observation of data assets. https://github.com/dagster-io/dagster.

[22] etcd Authors. 2022. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

[23] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine For Big Graph Processing. *Proc. VLDB Endow.* 14, 12 (2021), 2879–2892.

[24] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, Xiaojian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing sequential graph computations. *ACM Transactions on Database Systems (TODS)* 43, 4 (2018), 1–39.

[25] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. 2021. Scaling Large Production Clusters with Partitioned Synchronization. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 81–97.

[26] Linux Foundation. 2015. Data Plane Development Kit (DPDK). http://www.dpdk.org.

[27] The Apache Software Foundation. 2022. Apache Airflow: A platform to programmatically author, schedule, and monitor workflows. https://airflow.apache.org/.

[28] The Apache Software Foundation. 2022. Apache Data Fusion SQL Query Engine. https://arrow.apache.org/datafusion/.

[29] The Apache Software Foundation. 2022. Apache Doris: An easy-to-use, high-performance and unified analytical database. https://doris.apache.org/.

[30] The Apache Software Foundation. 2022. Apache Dremio: The Easy and Open Data Lakehouse. https://www.dremio.com/.
[31] The Apache Software Foundation. 2022. Arrow: A cross-language development platform for in-memory analytics. https://github.com/apache/arrow.
[32] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.
[33] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P Grosvenor, Allen Clement, and Steven Hand. 2015. Musketeer: all for one, one for all in data processing systems. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.
[34] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 17–30.
[35] Inc. Google. 2022. Protocol Buffers: A language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers.
[36] Robert Grandl, Arjun Singhvi, Raajay Viswanathan, and Aditya Akella. 2021. Whiz: Data-Driven Analytics Execution. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.
[37] gRPC Authors. 2022. gRPC: A high performance, open source universal RPC framework. https://grpc.io.
[38] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430* (2021).
[39] Inc. Juicedata. 2022. JuiceFS: A POSIX, HDFS and S3 compatible distributed file system for cloud. https://juicefs.com/en/.
[40] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–15.
[41] Jingdong Li, Zhao Li, Jiaming Huang, Ji Zhang, Xiaoling Wang, Xingjian Lu, and Jingren Zhou. 2021. Large-scale Fake Click Detection for E-commerce Recommendation Systems. In *ICDE*.
[42] libclang Authors. 2022. libclang: C interface to Clang. https://clang.llvm.org/doxygen/group__CINDEX.html.
[43] libfuse authors. 2022. libfuse: The reference implementation of the Linux FUSE (Filesystem in Userspace) interface. https://github.com/libfuse/libfuse.
[44] Redis Ltd. 2022. Redis: The open source, in-memory data store. https://redis.io/.
[45] The Alibaba Group Holding Ltd. 2022. Mars: a tensor-based unified framework for large-scale data computation. https://github.com/mars-project/mars.
[46] Ruotian Luo. 2017. An Image Captioning codebase in PyTorch. https://github.com/ruotianluo/ImageCaptioning.pytorch.
[47] Frank McSherry, Michael Isard, and Derek G Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.
[48] Anthony M Middleton. 2010. Data-intensive technologies for cloud computing. In *Handbook of cloud computing*. Springer, 83–136.
[49] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.
[50] Vu Nguyen, Sophia Deeds-Rubin, Thomas Tan, and Barry Boehm. 2007. A SLOC counting standard. In *Cocomo ii forum*, Vol. 2007. Citeseer, 1–16.
[51] Shoumik Palkar and Matei Zaharia. 2019. Optimizing Data-Intensive Computations in Existing Libraries with Split Annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 291–305. https://doi.org/10.1145/3341301.3359652
[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[53] Zhengping Qian, Chenqiang Min, Longbin Lai, Yong Fang, Gaofeng Li, Youyang Yao, Bingqing Lyu, Xiaoli Zhou, Zhimin Chen, and Jingren Zhou. 2021. GAIA: A System for Interactive Analysis on Distributed Graphs Using a High-Level Language. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*.
[54] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. 2016. Sulong - Execution of LLVM-Based Languages on the JVM: Position Paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Rome, Italy) *(ICOOOLPS '16)*. Association for Computing Machinery,

New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/3012408.3012416

[55] scikit-learn Authors. 2022. scikit-learn: Machine-Learning in Python. https://scikit-learn.org/.

[56] 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. https://doi.org/10.1109/ICDE.2019.00196

[57] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[58] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable cross-language services implementation. *Facebook white paper* 5, 8 (2007), 127.

[59] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. 2005. Inlining Java Native Calls at Runtime. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. Association for Computing Machinery, New York, NY, USA, 121–131. https://doi.org/10.1145/1064979.1064997

[60] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive-a petabyte scale data warehouse using hadoop. In *2010 IEEE 26th international conference on data engineering (ICDE 2010)*. IEEE, 996–1005.

[61] Hongwei Wang, Fuzheng Zhang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. 2019. Multi-task feature learning for knowledge graph enhanced recommendation. In *The World Wide Web Conference*. 2000–2010.

[62] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, Indiana, USA) *(Onward! 2013)*. Association for Computing Machinery, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581

[63] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.

[64] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. 2016. Apache spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

[65] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

[66] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.

[67] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 301–316.

[68] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

Received November 2022; revised February 2023; accepted March 2023