Skyway: Connecting Managed Heaps in Distributed Big Data Systems

Khanh Nguyen
University of California, Irvine
khanhtn1@uci.edu

Lu Fang
University of California, Irvine
lfang3@uci.edu

Christian Navasca
University of California, Irvine
cnavasca@uci.edu

Guoqing Xu
University of California, Irvine
harry.g.xu@uci.edu

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

Shan Lu
University of Chicago
shanlu@uchicago.edu

Abstract

Managed languages such as Java and Scala are prevalently used in development of large-scale distributed systems. Under the managed runtime, when performing data transfer across machines, a task frequently conducted in a Big Data system, the system needs to *serialize* a sea of objects into a byte sequence before sending them over the network. The remote node receiving the bytes then *deserializes* them back into objects. This process is both performance-inefficient and labor-intensive: (1) object serialization/deserialization makes heavy use of *reflection*, an expensive runtime operation and/or (2) serialization/deserialization functions need to be hand-written and are error-prone. This paper presents Skyway, a JVM-based technique that can directly connect managed heaps of different (local or remote) JVM processes. Under Skyway, objects in the source heap can be directly written into a remote heap without changing their formats. Skyway provides performance benefits to any JVM-based system by *completely eliminating* the need (1) of invoking serialization/deserialization functions, thus saving CPU time, and (2) of requiring developers to hand-write serialization functions.

CCS Concepts • **Information systems → Data management systems**; • **Software and its engineering → Memory management**;

Keywords Big data, distributed systems, data transfer, serialization and deserialization

ASPLOS '18, March 24–28, 2018, Williamsburg, VA, USA

ACM ISBN 978-1-4503-4911-6/18/03…$15.00
https://doi.org/10.1145/3173162.3173200

ACM Reference Format:
Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of ASPLOS '18*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3173162.3173200

1 Introduction

Modern Big Data systems need to frequently shuffle data in the cluster – a map/reduce framework such as Hadoop shuffles the results of each map worker before performing reduction on them; a dataflow system such as Spark supports many RDD transformations that need to shuffle data across nodes. As most of these systems are written in managed languages such as Java and Scala, data is represented as objects in a *managed heap*. Transferring an object *o* across nodes is complicated, involving three procedures shown in Figure 1. (1) A *serialization* procedure turns the whole object graph reachable from *o* into a binary sequence. This procedure reformats each object — among other things, it extracts the object data, strips the object header, removes all references stored in an object, and changes the representation of certain meta data. (2) This byte sequence is transferred to a receiver machine. (3) A *deserialization* procedure reads out the byte sequence, creates objects accordingly, and eventually rebuilds the object graph in the managed heap of the receiver machine.

Problems While many serialization/deserialization (S/D) libraries [3, 22, 32] have been developed, large inefficiencies exist in their implementations. Both our own experience (§2) and evidence from previous work [27] show that S/D accounts for 30% of the execution time in Spark. To explain why S/D is so costly, we discuss the handling of three key pieces of information these procedures have to extract, transfer, and reconstruct for every object reachable from *o*: (1) object data (*i.e.*, primitive-type fields), (2) object references (*i.e.*, reference-type fields), and (3) object type.

(1) Object-data access: An S/D library needs to invoke *reflective functions* such as `Reflection.getField` and `Reflection.setField` to enumerate and access every field to extract, on the sender side, and then write-back, on the

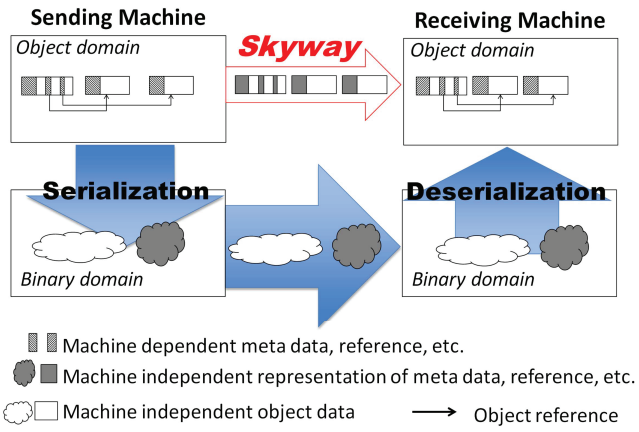

Figure 1. A graphical illustration of data transfer.

receiver side, each primitive object field *individually*. In a Big Data system, each data transfer involves many millions of objects, which would invoke these functions for millions of times or more. Reflection is a very expensive runtime operation. It allows the program to dynamically inspect or invoke classes, methods, fields, or properties without type information available statically at the cost of time-consuming string lookups, and is undesirable in performance-critical tasks.

(2) Type representation: Each type is represented by a special (meta) object in a managed runtime, and is referenced by the headers of the objects of the type. However, type references cannot be used to represent types in a byte sequence, because the meta objects representing the same type may have different addresses in different runtimes. The Java serializer represents every type by a string that contains the name of a class and all its super classes. This design causes meta data (*i.e.*, type strings) to consume a huge portion of the byte sequence transferred across the network. Furthermore, *reflection* must be used to resolve the type from each string during object re-creation on the receiver node.

(3) Reference adjustment: References contained in reference-type fields of transferred objects need to be adjusted, since those objects will be placed in different addresses on the receiver node. The Java serializer uses reflection to obtain and inline the contents of referenced objects into the binary representation of the referencing object. It constructs all objects reachable from o on the receiver machine using reflection, and then sets reference fields with the addresses of the just created referenced objects through reflection.

Recent Progresses Many third-party libraries have been developed. In particular, Kryo [22] is the library recommended in Spark. Kryo asks developers (1) to manually define S/D functions for types involved in data transfer, which speeds up object-data access, and (2) to manually register these types in a consistent order across all nodes, which makes it possible to use integers to represent types. Other libraries [3, 11, 32] follow similar principles.

However, the fundamental inefficiencies in data transfer still remain in Kryo – the user-defined functions need to be invoked for every transferred object at both the sender side and the receiver side. Due to the extremely large number of invocations of these S/D functions during sending and receiving, serialization and deserialization still takes a large portion of a data processing task's run time.

Furthermore, tremendous burden is put on developers who use Kryo. It is difficult for developers to understand how many and what types are involved, let alone consistently registering these types and developing correct and efficient S/D functions for each type. For instance, consider a `HashMap` object. Its serialization involves its key-value array, all the key/value pairs, and every key/value object. Its deserialization needs to recreate key and value objects, pair them, and additionally reshuffle key/value pairs to correctly recreate the key-value array because the hash values of keys may have changed.

Our Solution – Skyway The key problem with existing S/D libraries is that, with an existing JVM, there are no alternative routes to transfer objects other than first disassembling and pushing them down to a (different) binary format, and then reassembling and pulling them back up into a remote heap. In this paper, we advocate to build a "skyway" between managed heaps (shown in Figure 1) so that data objects no longer need to be pushed down to a lower level for transfer.

Skyway enhances the JVM, and enables object graphs to be moved *as is* from heap to heap and used on a remote node right after the move. Specifically, given a root object o specified by the application (*e.g.*, the RDD object in Spark), the Skyway-enhanced JVM performs a GC-like heap traversal starting from o, copies every reachable object into an output buffer, and conducts lightweight adjustment to machine-dependent meta data stored in an object without changing the object format. This output buffer can then be copied *as a whole* directly into the remote heap and used almost immediately after the transfer. This provides the following benefits to existing and future Big Data systems: (1) Skyway completely eliminates the cost of accessing fields and types, saving computation costs; and (2) the developer does not need to hand-write any S/D functions.

To achieve these goals, Skyway addresses the aforementioned three issues much more efficiently than all the existing S/D libraries, as discussed below.

First, Skyway, by changing the JVM, transfers every object as a whole, which completely eliminates the need of accessing individual data fields. Furthermore, since the hashcode of an object is cached in the header of the object, transferring the entirety of each object preserves the original hashcode of the object, so that hash-based data structures can be used on

the receiver node without rehashing — a process that takes a great amount of time in traditional S/D.

Second, Skyway represents types by employing an automated global type-numbering procedure – the master node maintains a registry of all types and their IDs, and each worker node communicates with the master to obtain IDs for its classes upon class loading. This process enables all classes across the cluster to be globally numbered without any developer intervention and thus each ID can be used to uniquely identify the same class on different nodes.

Third, Skyway employs an efficient "relativization" technique to adjust references. As objects are copied into the output buffer, pointers stored in them are relativized in linear time — they are changed from *absolute addresses* to *relative addresses*. Upon receiving the buffer, the Skyway client on the receiver node performs another linear scan of the input buffer to *absolutize* the relative information in the buffer.

Skyway may push more bytes over the network than S/D libraries, because it transfers the entirety of each object yet S/D libraries do not transfer object headers. However, much evidence [44] shows that bottlenecks in real systems are shifting from I/O to computing, and hence, we believe this design strikes the right design tradeoff — the savings on the computation cost significantly outweigh the extra network I/O cost incurred by the extra bytes transferred on a modern network. Our empirical results show that, even on a 1000Mb/s Ethernet (*e.g.*, most data centers use networks with higher bandwidth), transferring 50% of more data (about 100GB in total) in Spark for a real graph dataset increases the execution by only 4% (on network and read I/O) whereas the savings achieved by eliminating the S/D invocations are beyond 20%.

Why Does It Work? It is important to note that Skyway is *not* a general-purpose serializer. Our insight why Skyway would work well for Big Data processing is two-fold. First, data processing applications frequently shuffle many millions of objects and do so in strongly delimited phases. Hence, sending objects *in batch* without changing their formats provides significant execution efficiency. Second, the use of modern network technology enables extra bytes to be quickly transferred without incurring much overhead.

We have implemented Skyway in OpenJDK 8. Our evaluation on a Java serializer benchmark set JSBS [34], Spark [45], and Flink [2] shows that (1) Skyway outperforms *all the 90 existing S/D libraries* on JSBS, which uses a media-content based dataset – for example, it is **2.2**× faster than Kryo and **67.3**× faster than the Java serializer; (2) compared with Kryo and the Java serializer, Skyway improves the overall Spark performance by **16%** and **36%** for four representative analytical tasks over four real-world datasets; (3) for another real-world system Flink, Skyway improves its overall performance by **19%** compared against Flink's highly-optimized built-in serializers.

```
1  class Date extends Serializable{
2    private Year4D year;
3    private Month2D month;
4    private Day2D day;
5    public Date(String year, String month, String day) {
6      this.year = Year4D.parse(year);
7      this.month = Month2D.parse(month);
8      this.day = Day2D.parse(day);
9    }
10   public String toString() {
11     return "Date [year=" + year + " month=" + month +
           " day=" + day + "]";
12   }
13 }
14
15 class Year4D extends Serializable{...}
16 class Month2D extends Serializable{...}
17 class Day2D extends Serializable{...}
18 class DateParser extends Serializable {
19   /* Turn a string into a Date object*/
20   Date parse(String s) {...}
21 }
22
23 class SimpleSparkJob {
24   void main(String[] args) {
25     StreamingContext ssc = new StreamingContext(args
           [0], new Duration(1000));
26     DateParser parser = new DateParser();
27     JavaRDD<String> lines = ssc.textFileStream("dates.
           txt");
28     JavaRDD<Date> mapRes = lines.map(line -> parser.
           parse(line));
29     List<Date> result = mapRes.collect();
30   }
31 }
```

Figure 2. A simple Spark program that parses strings into `Date` objects.

2 Background and Motivation

This section gives a closer examination of S/D and its cost using Spark as an example.

2.1 Background

When Does S/D Happen? Spark conducts S/D throughout the execution. There are two categories of S/D tasks: *closure serialization* and *data serialization*. Closure S/D occurs between the driver and a worker. Since a Spark program is launched by the driver, the driver needs to execute portions of it on remote workers.

Figure 2 shows a Spark program that reads a sequence of strings, each of which represents a date, from a text file (Line 27). It next parses these strings by invoking a `map` function on the RDD (Line 28). The map transformation takes a lambda expression (*i.e.*, a *closure*) as input, which parses each string by invoking the `parse` function that turns a string into a `Date` object. Finally, the RDD action `collect` is invoked to bring all `Date` objects to the driver.

While this program is executed by the driver, Spark schedules the execution of the closure (*i.e.*, the lambda expression passed to `map`) on the worker nodes. Closure serialization is thus needed to transfer the closure and everything it needs from the driver to each worker node. In this example, the

closure refers to the object `parser` created outside its scope. Hence, `parser` also needs to be serialized during closure serialization. This explains why the `DateParser` class needs to implement the Java `Serializable` interface.

The second type of S/D is data serialization that occurs between different workers or a worker and the driver. For example, action `collect` would cause all `Date` objects on the worker nodes to be transferred back to the driver. When each `Date` object is serialized, all the (`Year4D`, `Month2D`, and `Day2D`) objects directly or transitively reachable from it are serialized as well. To shuffle data across nodes, Spark serializes data objects on each node (*e.g.*, the result of a `map` operation) into disk files with a shuffling algorithm (*e.g.*, sort-based or hash-based). These files are then sent to different remote nodes where data objects are deserialized.

How Does S/D Work? *The Kryo serializer* requires the developer to register classes using the following code snippet:

```
SparkConf conf = new SparkConf();
conf.set("spark.kryo.registrator", "org.apache.spark.
    examples.MyRegistrator");
...
public class MyRegistrator implements KryoRegistrator
    {
    public void registerClasses(Kryo kryo) {
        kryo.register(Date.class);
        kryo.register(Year4D.class);
        kryo.register(Month2D.class);
        kryo.register(Day2D.class);
    }
}
```

The order in which these classes are registered defines an integer ID for each class. Using these integer class identifiers, the bytes generated by Kryo do not contain strings, leading to significant space savings during data transfer. Furthermore, Kryo deserializer can now resolve types without using reflection — Kryo automatically generates code like

```
switch(id) {
 case 0: return new Date();
 case 1: return new Year4D();
 ...
}
```

that uses regular `new` instructions to create objects on the receiving node.

However, in any real-world application, there can be a large number of user classes defined (including many classes from different libraries). Fully understanding what classes are referenced (directly or transitively) is a very labor-intensive process. Moreover, the developer has to manually develop S/D functions for each of these types; without these functions, the standard Java serializer would be used instead.

In both Kryo and the standard Java serializer, the number of times S/D functions are invoked is proportional to the dataset cardinality; every data transfer can easily require several millions of S/D invocations, taking a significant fraction of the execution time.

2.2 Motivation

To understand the S/D costs in the real world, we have performed a set of experiments on Spark. We execute Spark on a small cluster of 3 worker nodes, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 SSD, running CentOS 6.8. These three nodes are part of a large cluster connected via InfiniBand. We ran a TriangleCounting algorithm over the LiveJournal graph [4] that counts the number of triangles induced by graph edges. It is widely used in social network analysis for analyzing the graph connectivity properties [38]. We used Oracle JDK 8 (build 25.71) and let each slave run one single executor – the single-thread execution on each slave made it easy for us to measure the breakdown of performance. The size of the input graph was around 1.2GB and we gave each JVM a 20GB heap – a large enough heap to perform in-memory computation – as is the recommended practice in Spark. Tungsten sort was used to shuffle data.

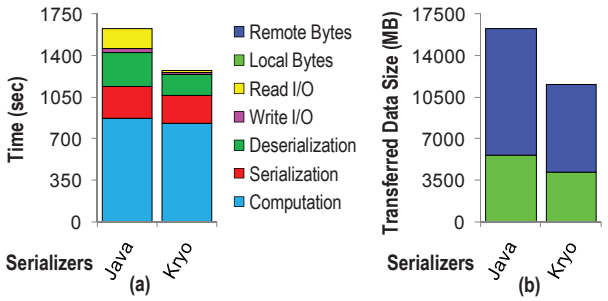

Figure 3. Spark S/D costs: (a) performance breakdown when running TriangleCounting over the LiveJournal graph on three nodes; (b) bytes shuffled under the two serializers; Local Bytes and Remote Bytes show the number of bytes fetched from the local and remote RDD partitions.

Figure 3(a) shows Spark's performance under the Kryo and Java serializers. Before transferring data over the network, Spark shuffles and sorts records, and saves the sorted records as disk files. The cost is thus broken down into five components: computation time, serialization time (measured as time spent on turning RDD records into byte sequences), write I/O (measured as the time writing bytes onto disk), deserialization time (measured as time spent on reconstructing RDD record objects from bytes), and read I/O (measured as time reading bytes). Since each JVM has a large heap compared to the amount of data processed, the garbage collection cost is less than 2% and thus not shown on the figure. The network cost is negligible and included in the read I/O.

One observation is that the invocation of S/D functions takes a huge portion (more than 30%) of the total execution time under both Kryo and the Java serializer. Under Kryo, the invocations of the serialization and deserialization take 18.2% and 14.1% of the total execution time, respectively; under the Java serializer, these two take 16.3% and 17.8%. The actual write and read I/O time is much shorter in comparison, taking 1.4% and 1.1% under Kryo, and 2.3% and

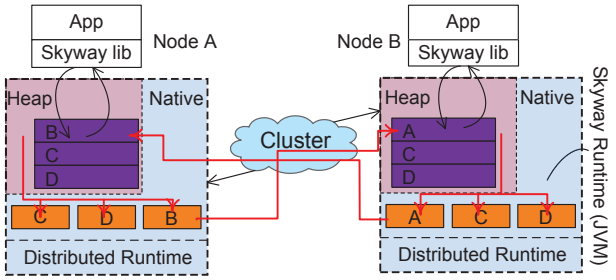

Figure 4. Skyway's system architecture. Purple and orange rectangles represent input (in-heap) buffers and output (native) buffers, respectively; objects flow along red arrows.

9.9% under the Java serializer. The read I/O is significantly increased under the Java serializer primarily because the Java serializer needs to read many type strings. For example, serializing an object containing a 1-byte data field can generate a 50-byte sequence [40] – in addition to its own field and the fields in its superclasses, the serializer needs to (1) write out the class name and (2) recursively write out the description of the superclasses of the object's class until it reaches `java.lang.Object` (*i.e.*, the root of all classes). This is validated by the "Remote Bytes" results in Figure 3(b).

Another observation is that the S/D process is a bottleneck that cannot be easily removed by upgrading hardware. Unlike other bottlenecks such as GC (that can be eliminated almost entirely by using a large heap) or I/O (that can be significantly reduced by using fast SSDs and InfiniBand networks), S/D is a memory- and compute-intensive process that turns heap objects into bytes and vice versa. The inefficiencies inherent in the process strongly call for system-level optimizations.

3 Design Overview

This section provides an overview of Skyway, explaining how Skyway is designed towards three goals — correctness, efficiency, and ease of integration.

Figure 4 shows the system architecture of Skyway, including three major parts. First, to achieve *correct* data transfer, Skyway modifies the JVM to conduct object traversal, object cloning, and adjustment within each cloned object. Second, to achieve *efficient* data transfer, Skyway carefully maintains input and output buffers, and streams buffer content across machines. Third, to make Skyway *easy* to use, Skyway library provides a set of easy-to-use and backward-compatible APIs for application developers.

3.1 Correctness

Skyway adjusts machine-specific parts of each transferred object to guarantee execution correctness. First, Skyway fills the type field of an object header with an automatically maintained global type-ID during sending, and later replaces it with the correct type representation on the receiving node. The details are presented in §4.1. Second, Skyway replaces the references stored in all non-primitive fields of an object with relativized references during sending, and turns them back to the correct absolute references during receiving. The details are presented in §4.2. Finally, certain meta data such as GC bits and lock bits need to be reset when objects are moved to another machine. Skyway resets these flags at sending, and does *not* need to access them at receiving.

Skyway also provides support for heterogeneous clusters where JVMs on different machines may support different object formats. If the sender and receiver nodes have different JVM specifications, Skyway adjusts the format of each object (*e.g.*, header size, pointer size, or header format) when copying it into the output buffer. This incurs an extra cost only on the sender node while the receiver node pays *no* extra cost for using the transferred objects. For homogeneous clusters, such platform-adjustment cost is not incurred on any nodes. The only assumption Skyway uses is that the sender and the receiver use the same version of each transfer-related class – if two versions of the same class have different fields, object reading would fail. However, this assumption is not unique for Skyway; it needs to hold for all other serializers as well.

3.2 Efficiency

Skyway uses a GC-like traversal to discover the object graph reachable from a set of root objects. To improve efficiency, Skyway uses buffering — Skyway copies every object encountered during the traversal into a buffer on the sending node (*i.e.*, output buffer) and streams the buffer content to the corresponding buffer(s) on the receiving node (*i.e.*, input buffer). Both output and input buffers are carefully designed for efficiency concerns. Multi-threaded data transfer is also supported (*cf.* §4).

Skyway output buffers are segregated by receivers — objects with the same destination are put into the same output buffer. Only one such output buffer exists for each destination. The output buffer can be safely cleared after its objects are sent. Skyway input buffers are segregated by senders, so that data objects coming from different senders can be written simultaneously without synchronization. Note that the heap of a receiver node may actually contain multiple input buffers for each sender, each holding objects sent in a different round of shuffling from the sender. Skyway does not reuse an old input buffer unless the developer explicitly frees the buffer using an API – frameworks such as Spark cache all RDDs in memory and thus Skyway keeps all input buffers.

Output buffers are located in *off-the-heap native memory* – they will not interfere with the GC, which could reclaim data objects before they are sent if these buffers were in the managed heap. Input buffers are allocated from the managed heap so that data coming from a remote node is directly written into the heap and can be used right away. Furthermore, while each input buffer is shown as consuming contiguous

heap space in Figure 4, we allow it to span multiple small memory chunks for two reasons. First, due to streaming, the receiver may not have the knowledge of the number of sent bytes, and hence, determining the input-buffer size is difficult. Second, allocating large contiguous space can quickly lead to memory fragmentation, which can be effectively mitigated by using smaller memory chunks (§4.3).

Streaming is an important feature Skyway provides for these buffers: for an output buffer, it is both time-inefficient and space-consuming if we do not send data until all objects are in; for an input buffer, streaming would allow the computation to be performed in parallel with data transfer. Supporting streaming creates many challenges, *e.g.*, how to adapt pointers without multiple scans and how to manage memory on the receiver node (§4.2).

3.3 Ease of Integration

Skyway aims to provide a simple interface for application developers. Skyway should support not only the development of brand new systems but also easy S/D library integration for existing systems such as Spark. To this end, Skyway provides a set of high-level Java APIs that are directly compatible with the standard Java serializer.

Skyway provides `SkywayObjectOutputStream` and `SkywayObjectInputStream` classes that are subclasses of the standard `ObjectOutputStream` and `ObjectInputStream`. These two classes create an interface for Skyway's (native) implementation of the `readObject` and `writeObject` methods. A `SkywayObjectOutputStream`/`SkywayObjectInputStream` object is associated with an output/input buffer. We have also created our `SkywayFileOutputStream`/`SkywayFileInputStream` and `SkywaySocketOutputStream`/`SkywaySocketInputStream` classes – one can easily program with Skyway in the same way as programming with the Java serializer.

Switching a program from using its original library to using Skyway requires light code modifications. For example, we do not need to change object-writing/reading calls such as `stream.writeObject(o)` at all. The only modification is to (1) instantiate `stream` to be a `SkywayFileOutputStream` object instead of any other type of `ObjectOutputStream` objects and (2) identify a shuffling phase with an API function `shuffleStart`. Since all of our output buffers need to be cleared before the next shuffling phase starts (§4), Skyway needs a mark from the developer to know when to clear the buffers. Identifying shuffling phases is often simple – in many systems, a shuffling phase is implemented by a `shuffle` function and the developer can simply place a call to `shuffleStart` in the beginning of the function. Also note that, user programs written to run on Big Data systems, such as the one in Figure 2, mostly do not directly use S/D libraries and hence can benefit from Skyway without changes.

Finally, Skyway provides an interface that allows developers to easily update some object fields after the transfer, such as re-initializing some fields for semantic reasons. For example, the code snippet below updates field `timestamp` in the class `Record` with the value returned by the user-defined function `updateTimeStamp` when a `Record` object is transferred. Of course, we expect this interface to be used rarely — the need to update object data content after a transfer never occurs in our experiments.

```
1 /*Register the update function*/
2 registerUpdate(Record.class, Record.class.getField("
      timeStamp"), SkywayFieldUpdateFunctions.
      getFunction(SkywayUpdate.class, "updateTimeStamp"
      , "()[B");
3 ...
4 class SkywayUpdate{
5   /*The actual update function*/
6   public byte[] updateTimeStamp(){
7     return new byte[]{0};
8   }
9 }
```

4 Implementation

We implemented Skyway in Oracle's production JVM OpenJDK 1.8.0 (build 25.71). In addition to implementing our object transfer technique, we have modified the classloader subsystem, the object/heap layout, and the Parallel Scavenge garbage collector, which is the default GC in OpenJDK 8. We have also provided a Skyway library for developers.

4.1 Global Class Numbering

Skyway develops a distributed type-registration system that automatically allows different representations of the same class on different JVM instances to share the same integer ID. This system completely eliminates the need of using strings to represent types during data transfer (as in the standard Java serializer) or the involvement of human developers to understand and register classes (as in Kryo).

Skyway type registration runs inside every JVM and maintains a *type registry*, which maps every type string to its unique integer ID. The driver JVM assigns IDs to all classes; it maintains a complete type registry covering *all* the classes that have been loaded in the cluster and made known to the driver since the computation starts. Every worker JVM has a *registry view*, which is a subset of the type registry on the driver; it checks with the driver to obtain the ID for every class that it loads and does not yet exist in the local registry view. An example of these registries is shown in Figure 5.

Algorithm 1 describes the algorithms running on the driver and worker JVMs. The selection of the driver is done by the user through an API call inserted in the client code. For example, for Spark, one can naturally specify the JVM running the Spark driver as the Skyway driver, and all the Spark worker nodes run Skyway workers. Fault tolerance is provided by the application – *e.g.*, upon a crash, Spark restarts the system on the Skyway-equipped JVMs; Skyway's driver JVM will be launched on the node that hosts Spark's driver.

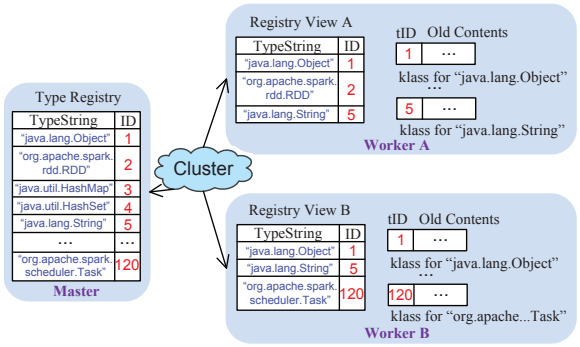

Figure 5. Type registries used for global class numbering.

At the beginning, the driver populates the registry by scanning its own loaded classes after the JVM finishes its startup logic (Lines 4 – 8). Next, the driver switches to background by running a daemon thread that listens on a port to process lookup requests from the workers (Lines 10 – 19).

Skyway uses a *pull*-based communication between the driver and workers. Upon launching a worker JVM, it first requests (Line 22) and obtains (Line 12) the current complete type registry from the driver through a "REQUEST_VIEW" message. This provides each worker JVM with a *view* of all classes loaded so far in the cluster at its startup. The rationale behind this design is that most classes that will be needed by this worker JVM are likely already registered by the driver or other workers. Hence, getting their IDs in a *batch* is much more efficient than making individual remote-fetch requests.

We modify the class loader on each worker JVM so that during the loading of a class, the loader obtains the ID for the class. The loader first consults the registry view in its own JVM. If it cannot find the class, it goes on to communicate with the driver (Lines 29 – 34) by a "LOOKUP" message with the class name string. The driver returns the ID if the string exists in its own registry or creates a new ID and registers it with the class name (Line 18). Once the worker receives this ID, it updates its registry view (Line 34). Finally, the worker JVM writes this ID into the meta object of the class (Line 35). In the JVM terminology, a meta object is called a "klass" (as shown in Figure 5). We add an extra field in each klass to accommodate its ID.

During deserialization, if we encounter an unloaded class on the worker JVM, Skyway instructs the class loader to load the missing class since the type registry knows the full class name. While other options (*e.g.*, low-collision hash functions such as the MD and SHA families) can achieve the same goal of assigning each class a unique ID, Skyway cannot use them as they cannot be used to recover class names.

Comparing with the standard Java serializer that sends a type string over the network together with every *object*, Skyway sends a type string at most once for every *class* on each machine during the whole computation. Naturally, the number of strings communicated under Skyway is several orders-of-magnitude smaller. Comparing with Kryo, Skyway

Algorithm 1: Driver and worker algorithms for global class numbering.

```
1  /* Driver Program */
2  /*Part 1: right after the JVM starts up*/
3  JVMStartUp() /*Normal JVM startup logic*/
4  /*Initialize the type registry*/
5  globalID ← 0
6  registry ← EMPTY_MAP
7  foreach class k loaded in the driver JVM do
8      registry ← registry ∪ {(Name(k), globalID++)}
9  /*Part 2: a daemon thread that constantly listens*/
10 while Message m = ListenToWorkers() do
11     if m.type == "REQUEST_VIEW" then
12         SendMsg(m.workerAddr, registry)
13     else if m.type == "LOOKUP" then
14         /*The content of a "LOOKUP" message from worker to driver is a
           class string*/
15         id ← LookUp(registry, m.content)
16         if id == Null then
17             id ← globalID++
18             registry ← registry ∪ {(m.content, id)}
19         SendMsg(m.workerAddr, id)
20 /* Worker Program*/
21 /* Part 1: inside the JVM startup logic*/
22 SendMsg(driverAddr, ComposeMsg("REQUEST_VIEW", Null, myAddr))
23 Message m = ListenToDriver()
24 /*The content of a "LOOKUP" message is the registry map*/
   registryView ← m.content
25 /* Part 2: after the class loading routine*/
26 clsName ← GetClassName()
27 metaObj ← LoadClass(clsName)
28 id ← LookUp(registryView, clsName)
29 if id == Null then
30     SendMsg(driverAddr, ComposeMsg("LOOKUP", clsName, myAddr))
31     Message m = ListenToDriver()
32     /*The content of a message from driver to worker is an ID*/
33     id ← m.content
34     registryView ← registryView ∪ {(clsName, id)}
35 WriteTID(metaObj, id)
```

automatically registers all classes, and eliminates the need for developers to understand what classes will be involved in data transfer, leading to significantly reduced human effort.

4.2 Sending Object Graph

Overview When `writeObject(root)` is invoked on a `SkywayObjectOutputStream` object, Skyway starts to traverse and send the object graph reachable from `root`. Algorithm 2 describes the single-threaded logic of copying the object graph reachable from a user-specified *root*, and we discuss the multi-threaded extension later in this section.

At a high level, Skyway mimics a BFS-based GC traversal. It maintains a queue `gray` holding records of every object that has been visited but not yet processed, as well as the location `addr` at which this object will be placed in the output buffer `ob`. Every iteration of the main loop (Line 8) processes the top record in `gray` and conducts three tasks.

First, based on the object-address pair (*s*, *addr*) retrieved from `gray`, an object *s* is cloned into buffer *ob* at a location calculated from *addr* (Line 10). CloneInBuffer would also

Algorithm 2: Copying the object graph reachable from object *root* and relativizing pointers for a single thread.

```
   Input: Shuffling phase ID sID, a top object root, output buffer ob
1  ob.allocableAddr ← 0
2  Word w ← Read(root, OFFSET_BADDR)
3  pID ← HighestByte(w)
4  /*root has not been visited in the current phase*/
5  if pID < sID then
6      /*gray is a list of pairs of objects and their buffer addresses*/
7      gray ← {(root, ob.allocableAddr)}
8      while gray ≠ ∅ do
9          Object-Address pair (s, addr) ← RemoveTop(gray)
10         CloneInBuffer(s, ob, addr − ob.flushedBytes)
11         /*Update the clone of s in the buffer*/
12         Write(addr, OFFSET_BADDR, 0)
13         ResetMarkBits(addr)
14         Write(addr, OFFSET_KLASS, s.klass.tID)
15         foreach Reference-typed field f of s do
16             Object o ← s.f
17             if o ≠Null then
18                 Word v ← Read(o, OFFSET_BADDR)
19                 phaseID ← HighestByte(v)
20                 if phaseID < sID then
21                     /* o has not been copied yet*/
                       newAddr ← ob.allocableAddr
22                     Write(o, OFFSET_BADDR, Compose(sID,
                       newAddr))
23                     PushToQueue(gray, {(o, newAddr)})
24                     ob.allocableAddr += GetSize(o)
25                 else
26                     newAddr ← Lowest7Bytes(v)
27                 Write(addr, OFFSET(f), newAddr)
28 else
29     oldAddr ← Lowest7Bytes(w)
30     WriteBackwardReference(oldAddr)
31 SetTopMark()
```

adjust the format of the clone if Skyway detects that the receiver JVM has a different specification from the sender JVM, following a user-provided configuration file that specifies the object formats in different JVMs. Second, the header of the clone is updated (Lines 12 – 22). Third, for every reference-typed field f of s, Skyway pushes the referenced object o into the working queue `gray` if o has not been visited yet and then updates f with a relativized address (*i.e.*, o's position in output buffer), which will enable a fast reference adjustment on the receiver machine (Lines 15 – 27).

As objects are copied into the buffer, which is in native memory, the buffer may be flushed (*i.e.*, the streaming process). A flushing is triggered by an allocation at Line 10 — the allocation first checks whether the buffer still has space for the object s; if not, the buffer *ob* is flushed and the value of *ob.flushedBytes* is increased by the size of the buffer.

Reference Relativization Imagine that a reference field f of an object s points to an object o. Skyway needs to adjust f in the output buffer, as o may be put at a different address on the receiver node. Skyway replaces the cloned field f with the relative address in *ob* where o will be cloned to. This will allow the receiver node to easily calculate the correct absolute value for every reference in an input buffer, once the input buffer's starting address is determined.

We first describe the overall relativization algorithm, and then discuss how Skyway addresses the three challenges caused by streaming and multi-phase data shuffling.

As shown on Lines 15 – 27 of Algorithm 2, for each reference-type field $s.f$, Skyway follows the reference to find the object (o). Skyway determines whether o has been visited in the current data-shuffling phase; details are discussed shortly. If not (Line 20), we know o will be cloned to the end of the output buffer at location *ob.allocableAddr*. We use this location to fill the `baddr` field of o (Line 22), and bump up *ob.allocableAddr* by the size of o to keep tracking the starting address of the next cloned object in *ob*. If o has been visited (Line 26), we retrieve its location in the output buffer from the lowest seven bytes of the `baddr` field in its object header, which we will explain more later. We then update the clone of f with this buffer location *newAddr* at which the clone of o will be or has already been placed (Line 27).

The first challenge is related to streaming. When Skyway tries to update f with the output-buffer location of o's clone (f points to o), this clone may have been streamed out and no longer exists in the physical output buffer. Therefore, Skyway has to carefully store such buffer-location information, making it available throughout a data-shuffling phase. Skyway saves the buffer location in the header of the original object, not the clone, using an extra field `baddr`. The modified object layout is shown in Figure 6(a). When o is reached again via a reference from another object o', the `baddr` in o will be used to update the reference in the clone of o'.

The second challenge is also related to streaming. The buffer location stored in `baddr` of an object s and in its record in `gray`-queue both represent the *accumulative* bytes that have been committed to other objects in output buffer before s. However, when Skyway clones o into the buffer, it needs to account for the streaming effect that the physical buffer may have been flushed multiple times. Therefore, Skyway subtracts the number of bytes previously flushed *ob.flushedBytes* from *addr* when computing the actual address in the buffer to which s should be copied (Line 10).

The third challenge is due to multi-phase data shuffling. Since one object may be involved in multiple phases of shuffling, we need to separate the use of its `baddr` field for different shuffling phases. Skyway employs an *sID* to uniquely identify a shuffling phase. Whenever Skyway updates the `baddr` field, the current *sID* is written to as a prefix to the highest byte of `baddr`. Thus, Skyway can easily check whether the content in a `baddr` field is computed during the same phase of data shuffling (*i.e.*, valid) or an earlier phase (*i.e.*, invalid). Examples are on Lines 2 – 5 and Lines 19 – 20 of Algorithm 2. In the former case, if *root* has already been copied in the same shuffling phase (due to a copy procedure initiated by another root object), Skyway simply creates

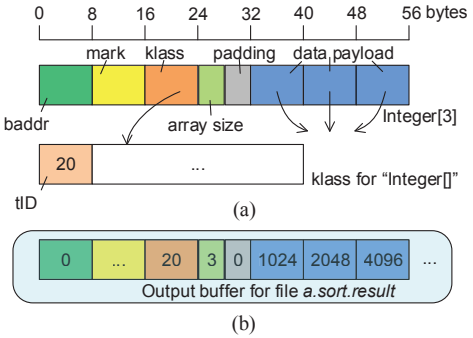

Figure 6. Skyway object layout in the heap (a) and an output buffer (b). This is an `Integer` array of three elements on a 64-bit HotSpot JVM. `mark` contains object locks, hash code of the object, and GC bits. `klass` points to the meta object representing the object's class. What follows is the data payload – three references to `Integer` objects. `baddr` and `tID` are both added by Skyway.

a *backward reference* pointing to its location in the buffer (Line 30). Skyway provides an API function `shuffleStart` that can be used by developers to mark a shuffling phase. *sID* is incremented when `shuffleStart` is invoked.

Header Update Lines 12 – 14 update the header of the cloned object in buffer. Following Figure 6, Skyway first clears the `baddr` field of the cloned object; this field will be used later to restore the object on the receiver side. Second, Skyway processes the `mark` word in the header, resetting the GC and lock bits while preserving the object hashcode. Since hashcodes are used to determine the layout of a hash-based data structure (*e.g.*, HashMap or HashSet), reusing them on the receiver side enables the immediate reuse of the data structure layout without rehashing. Third, Skyway replaces the `klass` pointer with the type ID stored in the `klass` meta object (Line 14).

Root Object Recognition After copying all objects reachable from *root* into the buffer, we set a *top mark*, which is a special byte indicating the starting point of the next top-level object. The reason for setting this mark is the following. For the original implementation of `writeObject`, an invocation of the function on a top object would in turn invoke the function itself recursively on the fields of the object to serialize the referenced objects. The deserialization process is exactly a reverse process – each invocation of `readObject` in the deserialization processes the bytes written in by its corresponding invocation of `writeObject` in serialization. However, Skyway's implementation of `writeObject` works in a different way – one invocation of the function on a top object triggers a system-level graph traversal that finds and copies all of its reachable objects. Similarly, Skyway's `readObject` also reads one object from the byte sequence instead of recursively reading out all reachable objects.

Although on the receiver side we can still compute all reachable objects for a root, this computation also needs a graph traversal and is time-consuming. As an optimization, we let the sender explicitly mark the root objects so that the receiver-side computation can be avoided. This is achieved by top marks. With these top marks, Skyway can easily skip the lower-level objects in the middle and find the next top object. Note that this treatment does not affect the semantics of the program – all the data structures reachable from top objects are recovered by the system, not by the application APIs.

Support for Threads Algorithm 2 does not work in cases that multiple threads on one node try to transfer the same object concurrently (*i.e.*, shared objects). Since each data-transfer thread has its own output buffer and the `baddr` field of a shared object can only store the relative buffer address for one thread t at a time, when other threads visit the object later, they would mistakenly use this address that is specific to t. To solve the problem, we let the lower seven bytes of `baddr` store both stream/thread ID (with the two highest bytes) and relative address (with the five lowest bytes).

When an object is first visited by t, t's thread ID is written into `baddr` together with the address specific to t's buffer. When the object is visited again, Skyway first checks whether the ID of the visiting thread matches the thread ID stored in its `baddr`. If it does, `baddr` of the object is used; otherwise, Skyway switches to a *hash table-based* approach – each thread maintains a thread-local hash table; the object and its buffer address for the thread are added into the hash table as a key and a value. Compare-and-swap (CAS) is used to provide thread safety when updating each `baddr`.

This approach prevents a thread from mistakenly using the object's buffer address for another thread. An object will have distinct copies in multiple output buffers when visited by different threads; these copies will become separate objects after delivered to a remote node. This semantics is consistent with that of the existing serializers.

4.3 Receiving Object Graph

With the careful design on sending, the receiving logic is much simpler. To receive objects from a sender, the receiver JVM first prepares an input buffer, whose size is user-tunable, for the sender in its managed heap to store the transferred objects. A subtle issue here is that a sender node may use multiple streams (in multiple threads) to send data to the same receiver node simultaneously. To avoid race conditions, the receiver node creates an input buffer for each stream of each sender so that different streams/threads can transfer data without synchronizations. We create oversized buffers to fit objects whose size exceeds that of a regular buffer.

After the input buffer is filled, Skyway performs a linear scan of the buffer to absolutize types and pointers. For the `klass` field of each object, Skyway queries the local registry view to get the correct `klass` pointer based on the type ID and writes the pointer into the field. For a relative address

a stored in a reference field, Skyway replaces it with $a + s$ where s is the starting address of this input buffer.

There is one challenge related to streaming. Since Skyway may not know the total size of the incoming data while allocating the buffer, one buffer of a fixed length may not be large enough. Skyway solves this by supporting linked chunks – a new chunk can be created and linked to the old chunk when the old one runs out of space. Skyway does not allow an object to span multiple chunks for efficiency. Furthermore, when a buffer contains multiple chunks, the address translation discussed above needs to be changed. We first need to calculate which chunk i a relative address a would fall in. Then, because previous chunks might not be fully filled, we need to calculate the *offset* of a in the i-th chunk. Suppose s_i is the starting address of chunk i and hence, s_i + *offset* is the final absolute address for a. This address will be used to replace a in each pointer field.

As each input buffer corresponds to a distinct sender, we can safely start the computation to process objects in each buffer for which streaming is finished. This would not create safety issues because objects that come from different nodes cannot reference each other. However, we do need to block the computation on buffers into which data is being streamed until the absolutization pass is done.

Interaction with GC After receiving the objects, it is important for the Skyway client on the receiver JVM to make these objects reachable in the garbage collection. Skyway allocates all input buffers in the old generation (tenured) of the managed heap. In Skyway, we use the Parallel Scavenge GC (*i.e.*, the default GC in OpenJDK 8), which employs a *card table* that groups objects into fixed-sized buckets and tracks which buckets contain objects with young pointers. Therefore, we add support in Skyway that updates the card table appropriately to represent new pointers generated from each data transfer.

5 Evaluation

To thoroughly evaluate Skyway, we have conducted three sets of experiments, one on a widely-used suite of benchmarks and the other two on widely-deployed systems Spark and Flink. The first set of experiments focuses on comparing Skyway with *all* existing S/D libraries – since most of these libraries **cannot** be directly plugged into a real system, we used the Java serializer benchmark set (JSBS) [34], which was designed specifically to evaluate Java/Scala serializers, to understand where Skyway stands among existing S/D libraries. JSBS was initially designed to assess single-machine S/D. We modified this program to make it work in a distributed setting; details are discussed shortly.

In the second and third set of experiments, we modified the Spark and Flink code to replace the use of Kryo and the Java serializer (in Spark) and built-in serializers (in Flink) with Skyway in order to assess the benefit of Skyway to real-world distributed systems. All of our experiments were run on a cluster with 11 nodes, each with 2 Xeon(R) CPU E5-2640 v3 processors, 32GB memory, 1 100GB SSD, running CentOS 6.8 and connected by a 1000Mb/s Ethernet. Each node ran 8 job instances. The JVM on each node was configured to have a 30GB heap.

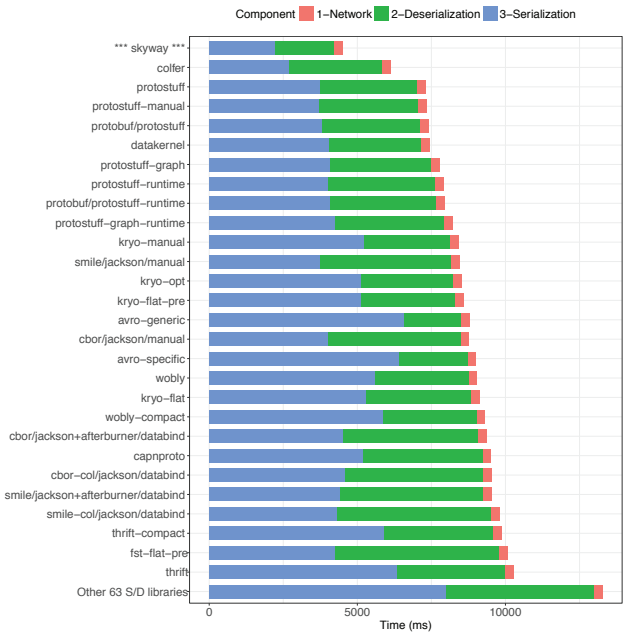

Figure 7. Serialization, deserialization, and network performance of different S/D libraries. Although we have compared Skyway with 90 existing libraries, we include in this table Skyway and 27 fastest-performing libraries. The last bar is a placeholder of the 63 libraries that perform slowly.

5.1 Java Serializer Benchmark Set

The JSBS contains several workloads under which each serializer and deserializer is repeatedly executed. Each workload contains several media content objects which consist of primitive `int` and `long` fields as well as reference-type fields. The driver program creates millions of such objects, each of which is around 1KB in JSON format. These objects are serialized into in-memory byte arrays, which are then deserialized back to heap objects. To understand the cost of transferring the byte sequences generated by different serializers, we modified the benchmark, turning it into a distributed program – each node serializes these objects, broadcasts the generated bytes to all the other nodes, and deserializes the received bytes back into objects. To execute this program, we involved five nodes and executed this process 1000 times repeatedly. The average S/D time for each object and the network cost are reported.

We have compared Skyway exhaustively with 90 existing S/D libraries. Due to space constraints, we excluded from the

Graphs	#Edges	#Vertices	Description
LiveJournal [5]	69M	4.8M	Social network
Orkut [18]	117M	3M	Social network
UK-2005 [6]	936M	39.5M	Web graph
Twitter-2010[23]	1.5B	41.6M	Social network

Table 1. Graph inputs for Spark.

paper 63 slower libraries whose total S/D time exceeds 10 seconds. The performance of the fastest 28 libraries is shown in Figure 7. Skyway, without needing any user-defined S/D functions, is the fastest of all of them. For example, it is 2.2× faster than Kryo-manual, which requires manual development of S/D functions. It is more than 67× faster than the Java serializer, which is not shown in the figure.

Colfer [11] is the only serializer whose performance is close to (but still 1.5× slower than) that of Skyway. It employs a compiler colf(1) to generate serialization source code from schema definitions to marshal and unmarshal data structures. Hence, the use of colf(1) requires user-defined schema of data formats, which, again, creates a practicality obstacle if data structures are complicated and understanding their layouts is a daunting task.

Skyway's faster S/D speed is achieved at the cost of greater numbers of bytes serialized. For example, Skyway generates, on average, 50% more bytes than the existing serializers. The details of the numbers of bytes are omitted from the paper due to space constraints. Note that the increased data amount does not cause the network cost to change much, whereas the computation cost in S/D is significantly reduced.

5.2 Improving Spark with Skyway

Experience We have modified Spark version 2.1.0 (released December 2016) to replace the use of Kryo-manual with the Skyway library. Spark was executed under Hadoop version 2.6.5 and Scala version 2.11. Our experience shows that the library replacement was rather straightforward – to use Skyway, we created a Skyway serializer that wraps the existing Input/OutputStream with our `SkywayInput/OuputStream` objects. We modified the Spark configuration (*spark.serializer*) to invoke the Skyway serializer instead of Kryo. Since data serialization in Spark shuffles orders of magnitude more data than closure serialization, we only used Skyway for data serialization. The Java serializer was still used for closure serialization. The entire `SkywaySerializer` class contains less than 100 lines of code, most of which was adapted directly from the existing `JavaSerializer` class. The number of lines of new code we wrote ourselves was only 10: 2 lines to wrap the I/O stream parameters, 3 lines to modify calls to `readObject`, and 5 lines to specify tuning parameters (*e.g.*, buffer size).

We ran Spark with four representative programs: WordCount (WC), PageRank (PR), ConnectedComponents (CC), and TriangleCounting (TC). WordCount is a simple MapReduce application that needs only one round of data shuffling.

Sys	Overall	Ser	Write	Des	Read	Size
Kryo	0.39 ~ 0.94 (0.76)	0.33 ~ 0.89 (0.59)	0.12 ~ 0.83 (0.61)	0.11 ~ 0.55 (0.26)	0.01 ~ 0.03 (0.02)	0.31 ~ 1.09 (0.52)
Skyway	0.27 ~ 0.92 (0.64)	0.19 ~ 1.29 (0.62)	0.12 ~ 1.61 (0.97)	0.04 ~ 0.43 (0.16)	0.01 ~ 0.05 (0.02)	0.91 ~ 3.13 (1.15)

Table 2. Performance summary of Skyway and Kryo on Spark: normalized to baseline (Java serializer) in terms of **Overall** running time, **Ser**ialization time, **Write** I/O time, and **Des**erialization time, **Read** I/O time (including the network cost), and the **Size** of byte sequence generated. A lower value indicates better performance. Each cell shows a percentage range and its geometric mean.

The other three programs are iterative graph applications that need to shuffle data in each iteration. We used four real-world graphs as input – LiveJournal (LJ) [4], Orkut (OR) [18], UK-2005 (UK) [6], and Twitter-2010 (TW) [23]; Table 1 lists their details.

For PageRank over Twitter-2010, Spark could not reach convergence in a reasonable amount of time (*i.e.*, 10 hours) for all configurations. We had to terminate Spark at the end of the 10th iteration and thus the performance we report is *w.r.t.* the first 10 iterations. All the other iterative applications ran to complete convergence. We have experimented with three serializers: the Java serializer, Kryo, and Skyway.

Spark Performance Figure 8(a) reports the running time comparisons among three serializers over the four input graphs. Since different programs have very different performance numbers, we plot them separately on different scales. For each dataset, WordCount and ConnectedComponents finished much more quickly than PageRank and TriangleCounting. This is primarily due to the nature of the application – WordCount has one single iteration and one single round of shuffling; it is much easier for ConnectedComponents (*i.e.*, a label propagation application, which finishes in 3-5 iterations) to reach convergence than the other two applications that often need many more iterations.

It is the same reason that explains why Skyway performs better for PageRank and TriangleCounting – since they perform many rounds of data shuffling, a large portion of their execution time is taken by S/D and thus the savings in data transfer achieved by Skyway are much more significant for these two applications than the other two.

A detailed summary of each run-time component is provided in Table 2. Network time is included in **Read**. On average, Skyway makes Spark run 36% and 16% faster than the Java serializer and Kryo. Compared to the Java serializer, Kryo achieves most of its savings from avoiding reading/writing type strings since Kryo relies on developers to register classes. As a result, the I/O in network and local reads has been significantly reduced. Skyway, on the contrary, benefits most from the reduced deserialization cost. Since the transferred objects can be immediately used, the process of recreating millions of objects and calling their constructors

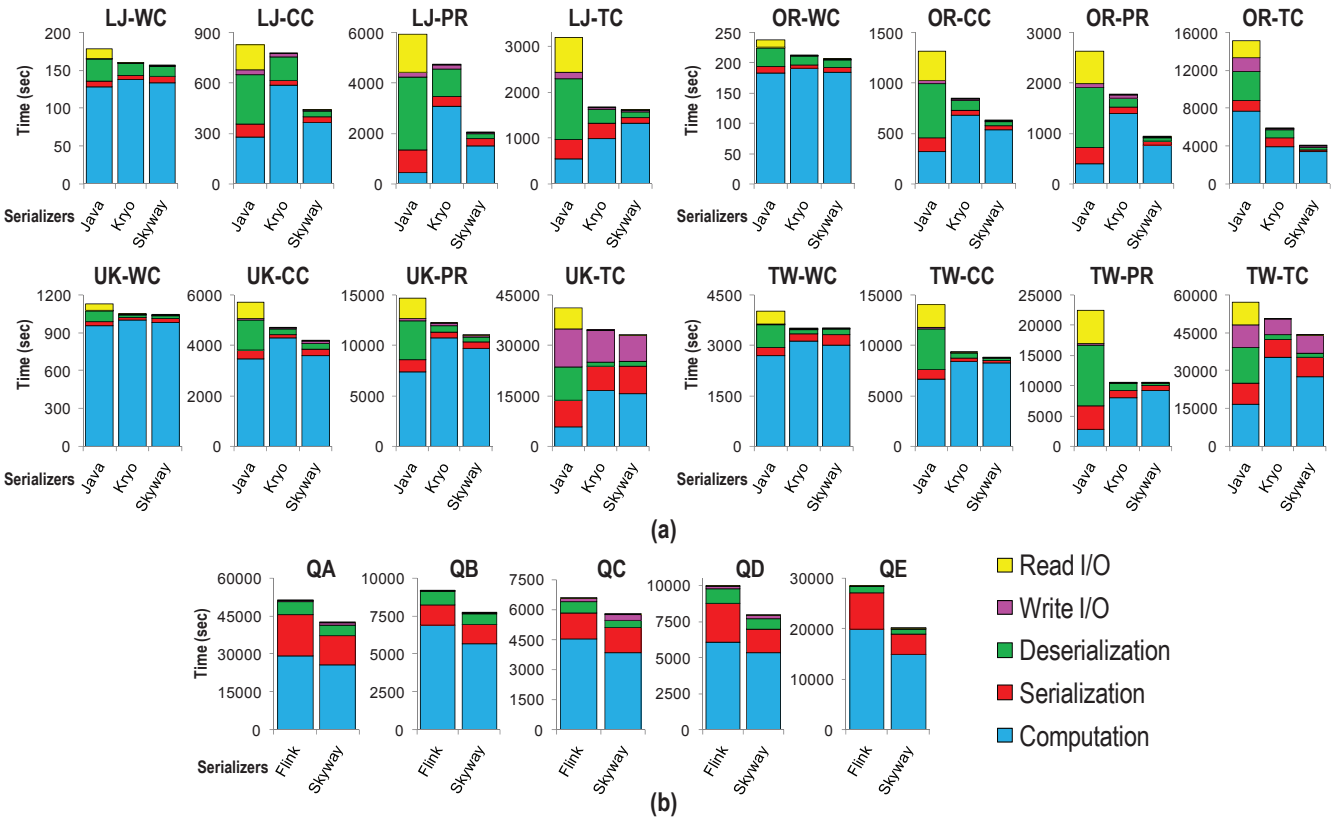

Figure 8. (a) Spark with Java serializer, Kryo, and Skyway; (b) Flink with Skyway and Flink's built-in serializer.

is completely eliminated. Furthermore, it is worth noting, again, that Kryo achieves its benefit via heavyweight manual development – there is a package of more than 20 classes (several thousands of lines of code) in Spark developed to use Kryo, while Skyway completely eliminates this manual burden and simultaneously achieves even higher gains.

The number of bytes transferred under Skyway is about the same as the Java serializer, but 77% more than Kryo due to the transferring of the entirety of each object. The increased data size is also reflected in the increased write I/O. Skyway's read I/O time is shorter than that of the Java serializer. This is primarily due to the elimination of object creation – we only need one single scan of each buffer instead of reading in individual bytes to create objects as done in Kryo. Skyway's read I/O is longer than that of Kryo because Kryo transfers much less bytes.

To understand what constitutes the extra bytes produced by Skyway, we analyzed these bytes for our Spark applications. Our results show that, on average, object headers take 51%, object paddings take 34%, and the remaining 15% are taken by pointers. Since headers and paddings dominate these extra bytes, future work could focus on compressing headers and paddings during sending.

Memory Overhead To understand the overhead of the extra word field *baddr* in each object header, we ran the Spark programs with the unmodified HotSpot and compared peak heap consumption with that of Skyway (by periodically running `pmap`). We found that the difference (*i.e.*,, the overhead) is relatively small. Across our four programs, this overhead varies from 2.1% to 21.8%, with an average of 15.4%.

5.3 Improving Flink with Skyway

We evaluated Skyway with the latest version of Flink (1.3.2, released August 2017) executing under Hadoop version 2.6.5. Flink has both streaming and batch processing models. Here we focus on the batch-processing model, and particularly, query answering applications.

Flink reads input data into a set of tuples (*e.g.*, rows in relational database); the type of each field in a tuple must be known at compile time. Flink can thus select a built-in serializer for each field to use when creating tuples from the input. Flink falls back to the Kryo serializer when encountering a type with neither a Flink-customized nor a user-defined serializer available. Since the read/write interface is clearly defined, we could easily integrate Skyway into Flink.

We used the TPC-H [37] data generator to generate a 100GB dataset as our input. Next, we transformed 5 representative SQL queries generated by TPC-H into Flink applications. The description of these queries can be found in Table 3. They were selected due to the diverse operations they perform and database tables they access.

	Description
QA	Report pricing details for all items shipped within the last 120 days.
QB	List the minimum cost supplier for each region for each item in the database.
QC	Retrieve the shipping priority and potential revenue of all pending orders.
QD	Count the number of late orders in each quarter of a given year.
QE	Report all items returned by customers sorted by the lost revenue.

Table 3. Descriptions of the queries used in Flink.

Overall	Ser	Write	Des	Read	Size
0.71 ~ 0.88	0.56 ~ 1.06	0.51 ~ 1.76	0.58 ~ 0.82	0.49 ~ 1.13	1.23 ~ 2.03
(0.81)	(0.77)	(0.96)	(0.75)	(0.61)	(1.68)

Table 4. Performance improvement summary of Skyway on Flink: normalized to Flink's built-in serializer.

Figure 8(b) shows Flink's performance improvement using Skyway. Performance summary is also shown in Table 4.

In Flink, the amount of time in deserialization (8.7%) is much less than that in serialization (23.5% on average). This is because Flink does not deserialize all fields of a row upon receiving it – only those involved in the transformation are deserialized. Despite this lazy mechanism, Skyway could improve Flink's performance by, an overall of 19%, compared to Flink's built-in serializer. The total number of bytes written by Skyway is also higher than the baseline – on average, Skyway emits 68% more bytes. It is worth noting that Skyway is compared with Flink's highly optimized built-in serializer; it is statically chosen and optimized specifically for the data types involved in the queries, and has been shown to outperform generic serializers such as Kryo.

6 Related Work

Object Sharing in the OS The idea of sharing memory segments across processes has been studied in the OS design [10, 15, 19, 24, 33]. An object can exist in different address spaces, allowing the system to share memory across simultaneously executing processes. Mach [33] introduces the concept of a memory object mappable by various processes. The idea was later adopted in the Opal [10] and Nemesis [19] operating systems to describe memory segments characterized by fixed virtual offsets. Lindstrom [24] expands these notions to shareable containers that contain code segments and private memory, leveraging a capability model to enforce protection. Although most contemporary OSes allow one process to be associated with a single virtual address space (SVAS), there exist systems that support multiple virtual address space (MVAS) abstractions.

The idea of multiple address spaces has mainly been applied to achieve protection in a shared environment [10, 15, 35]. More recently, to support the vast physical memory whose capacity may soon exceed the virtual address space size supported by today's CPUs, SpaceJMP [15] provides a new operating system design that promotes virtual address spaces to first-class citizens, which enables process threads to attach to, detach from, and switch between multiple virtual address spaces. Although this line of work is not directly related to Skyway, they share a similar goal of achieving memory efficiency when objects are needed by multiple processes. XMem [39] is a JVM-based technique that shares heap space across JVM instances. None of these techniques target object transfer in distributed systems.

Memory Management in Big Data Systems A variety of data computation models and processing systems have been developed in the past decade [1, 7, 9, 12, 13, 21, 30, 31, 36, 41–43, 45]. MapReduce [14] has inspired much research on distributed data-parallel computation, including Hyracks [20], Hadoop [1], Spark [45], and Dryad [21]. It has been extended [41] with Merge to support joins and adapted [12] to support pipelining. Yu et al. propose a programming model [42] for distributed aggregation for data-parallel systems. A number of high-level declarative languages for data-parallel computation have been proposed, including Sawzall [31], Pig Latin [30], SCOPE [9], Hive [36], and DryadLINQ [43]. These frameworks are all developed in managed languages and perform their computations on top of the managed runtime. Hence, data shuffling in these systems can benefit immediately from Skyway, as demonstrated in our evaluation (§5).

Recently, there has been much interest in optimizing memory management in language runtimes for efficient data processing [8, 16, 17, 25, 26, 28, 29]. These works are largely orthogonal to Skyway, although Skyway also fits in the category of language runtime optimizations. ITask [16] provides a library-based programming model for developing interruptible tasks in data-parallel systems. ITask solves the memory management problem using an orthogonal approach that interrupts tasks and dumps live data to disk. In addition, it is designed specifically for data-parallel programs and does not work for general (managed) systems.

7 Conclusion

This paper presents Skyway, the first JVM-based system that provides efficient data transfer among managed heaps. Our evaluation shows that Skyway outperforms all existing S/D libraries and improves widely-deployed systems such as Spark and Flink.

Acknowledgments

We thank the anonymous reviewers for their valuable and thorough comments. We are also grateful to Kathryn McKinley who pointed us to important related works. This material is based upon work supported by the National Science Foundation under grants CCF-1319786, CNS-1321179, CCF-1409829, IIS-1546543, CNS-1514256, CNS-1613023, CNS-1703598, and by the Office of Naval Research under grants N00014-14-1-0549 and N00014-16-1-2913.

References

[1] Apache 2017. Hadoop: Open-source implementation of MapReduce. http://hadoop.apache.org. (2017).

[2] Apache Flink 2017. Apache Flink. http://flink.apache.org/. (2017).

[3] Apache Thrift 2017. Apache Thrift. http://thrift.apache.org/. (2017).

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *KDD*. 44–54.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group Formation in Large Social Networks: Membership, Growth, and Evolution. In *KDD*. 44–54.

[6] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *WWW*. 595–601.

[7] Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *ICDE*. 1151–1162.

[8] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A Bloat-Aware Design for Big Data Applications. In *ISMM*. 119–130.

[9] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (2008), 1265–1276.

[10] Jeff Chase, Miche Baker-Harvey, Hank Levy, and Ed Lazowska. 1992. Opal: A Single Address Space System for 64-bit Architectures. *SIGOPS Oper. Syst. Rev.* 26, 2 (1992), 9.

[11] Colfer. 2017. The Colfer Serializer. https://go.libhunt.com/project/colfer. (2017).

[12] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online. In *NSDI*. 21–21.

[13] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*. 137–150.

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[15] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. 2016. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *ASPLOS*. 353–368.

[16] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-Parallel Programs. In *SOSP*. 394–409.

[17] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping Out Garbage Collection from Big Data Systems. In *HotOS*.

[18] Google. 2017. Orkut social network. http://snap.stanford.edu/data/com-Orkut.html. (2017).

[19] Steven M. Hand. 1999. Self-paging in the Nemesis Operating System. In *OSDI*. 73–86.

[20] UC Irvine. 2014. Hyracks: A data parallel platform. http://code.google.com/p/hyracks/. (2014).

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*. 59–72.

[22] Kryo 2017. The Kryo serializer. https://github.com/EsotericSoftware/kryo. (2017).

[23] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *WWW*. 591–600.

[24] A. Lindstrom, J. Rosenberg, and A. Dearle. 1995. The Grand Unified Theory of Address Spaces. In *HotOS*. 66–71.

[25] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *HotOS*.

[26] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *ASPLOS*. 457–471.

[27] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2015. Latency-tolerant Software Distributed Shared Memory. In *USENIX ATC*. 291–305.

[28] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *OSDI*. 349–365.

[29] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *ASPLOS*. 675–690.

[30] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig Latin: a not-so-foreign language for data processing. In *SIGMOD*. 1099–1110.

[31] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.* 13, 4 (2005), 277–298.

[32] Protocol Buffers 2017. Protocol Buffers. https://developers.google.com/protocol-buffers/docs/javatutorial. (2017).

[33] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. 1987. Machine-independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. In *ASPLOS*. 31–39.

[34] Eishay Smith. 2017. The Java Serialization Benchmark Set. https://github.com/eishay/jvm-serializers. (2017).

[35] Masahiko Takahashi, Kenji Kono, and Takashi Masuda. 1999. Efficient Kernel Support of Fine-Grained Protection Domains for Mobile Code. In *ICDCS*. 64–73.

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

[37] TPC. 2014. The standard data warehousing benchmark. http://www.tpc.org/tpch. (2014).

[38] Duncan J. Watts and Steven H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393, 6684 (1998), 440–442.

[39] Michal Wegiel and Chandra Krintz. 2008. XMem: Type-safe, Transparent, Shared Memory for Cross-runtime Communication and Coordination. In *PLDI*. 327–338.

[40] Java World. 2017. The Java serialization algorithm revealed. http://www.javaworld.com/article/2072752/the-java-serialization-algorithm-revealed.html. (2017).

[41] Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 2007. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD*. 1029–1040.

[42] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. 2009. Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations. In *SOSP*. 247–260.

[43] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*. 1–14.

[44] Matei Zaharia. 2016. What is changing in Big Data? https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/Zaharia_Matei_Big_Data.pdf. (2016). MSR Faculty Summit.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *HotCloud*.