An Experimental Comparison of Complex Object Implementations for Big Data Systems

Sourav Sikdar
Rice University
ss107@rice.edu

Kia Teymourian
Rice University
kiat@rice.edu

Chris Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

Many cloud-based data management and analytics systems support complex objects. Dataflow platforms such as Spark and Flink allow programmers to manipulate sets consisting of objects from a host programming language (often Java). Document databases such as MongoDB make use of hierarchical interchange formats—most popularly JSON—which embody a data model where individual records can themselves contain sets of records. Systems such as Dremel and AsterixDB allow complex nesting of data structures.

Clearly, no system designer would expect a system that stores JSON objects as text to perform at the same level as a system based upon a custom-built physical data model. The question we ask is: How significant is the performance hit associated with choosing a particular physical implementation? Is the choice going to result in a negligible performance cost, or one that is debilitating? Unfortunately, there does not exist a scientific study of the effect of physical complex model implementation on system performance in the literature. Hence it is difficult for a system designer to fully understand performance implications of such choices. This paper is an attempt to remedy that.

CCS CONCEPTS

• **Information systems** **Data management systems**; **Data structures**;

KEYWORDS

Experimental Comparison, Complex Objects Implementation, Big Data Management, Data Serialization

ACM Reference Format:
Sourav Sikdar, Kia Teymourian, and Chris Jermaine. 2017. An Experimental Comparison of Complex Object Implementations for Big Data Systems. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 13 pages.
https://doi.org/10.1145/3127479.3129248

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3129248

1 INTRODUCTION

Classically, relational data management systems supported records made of flat types. Using Java-like syntax, relational [10] and pre-relational systems [22] generally stored records like:

```
class Part {
 public long partKey;
 public String name;
 public String brand;
 public int size;
 ... }
```

Objects might have links between them, either implicitly (as in the relational model) or explicitly (in hierarchical database system [20]) but they were essentially flat.

Over time, various efforts were aimed at extending this simple model—most notably object-oriented [6] (or object-relational [19]) systems in the 1980's. But, no doubt following the slow integration of generic programming into mainstream object-oriented languages (C++ did not offer templates until 1991 [21] and Java lacked generics until 2004 [15]), such systems arguably did not offer much in the way of additional functionality compared to the flat relational model, besides pointers and some support for inheritance, and as such, the efforts never gained widespread attraction.

Storing Complex Objects. Today, however, the data management landscape is awash in systems supporting many complex data types [3, 4, 9, 14, 24, 25]. One might conjecture that as programmers have gotten used to modern programming languages with extensive support for features such as generic programming, they have become less accepting of the limitations associated with flat relations.

Objects In Modern Systems. In response, some modern cloud-based data processing systems (such as Hadoop [24], Spark [25] or Flink [3] and others) allow programmers to define objects in a host language, such as Java, taking full advantage of features such as generics. The objects are then managed in RAM by the language and its runtime (the Java Virtual Machine in the case of Java or Scala, the .NET framework in the case of C#). When objects must be moved over the network or onto (or off of) disk, they go through a separate serialization/deserialization process.

At What Cost? Clearly, parsing, objectification, allocation, garbage collection, etc., all cost something, and few system designers would expect a system using Java serialization or storing JSON objects as text to perform at the same level as a system based upon a custom-built physical data model. That said, self-describing interchange formats such as JSON are

attractive for many different reasons, and many popular systems *are* built upon such technologies. The question we ask in this paper is: what is the *extent* of the performance hit a system designer can expect when choosing one physical implementation instead of another? Is choosing to implement a system using JSON going to result in a negligible performance hit, or one that is debilitating? The literature lacks a comprehensive, scientific study of the effect of physical complex model implementation on system performance, and without such a study, it is difficult for a system designer to fully understand the performance implications of such choices.

Our Contributions. We experimentally compare different models for complex objects (host language objects and interchange formats such as JSON) as well as different methods for moving between RAM and a wire (host language serialization, BSON, Protocol buffers, and the classical database method of avoiding de/serialization entirely) and different language runtimes (managed and garbage-collected vs. not). We offer a detailed account of the pros and cons of the various approaches, with a special emphasis on tasks in a networked environment.

Some specific contributions of the paper are:

- We compare ten different complex object implementations across four different, representative tasks, from simple object reads, to an external sort for duplicate removal, to networked computations.
- We consider a spectrum of object complexities, from an object type that looks a lot like a classical relational record, to a complex object nested four levels deep, requiring up to 30KB to store on disk. We also consider the task of implementing a sparse vector for a statistical computation.
- We find that there is tremendous variance in the speed at which the tasks considered can be completed, depending upon the object implementation. For some tasks, there can be a 50× difference between the fastest and slowest implementations.

The massive variance in the performance of different physical object implementations demonstrates that—surprisingly—object implementation *may be the most important choice made by a system designer*. The cost of choosing the wrong implementation is so high that the wrong choice can make it impossible to build a performant system, no matter the quality of the rest of the system.

2 COMPLEX OBJECT MODELS

As described in the introduction to the paper, there are three different categories of data models commonly used for complex objects in modern systems; each has one or more possible, physical implementations, which we consider subsequently. They are:

- Host programming language objects;
- Self-describing documents (XML or JSON);
- System-specific, nested data models.

2.1 Host Language Objects

Host language objects are commonly supported by Big Data analytics engines whose main purpose is distributed, data-oriented computation (Hadoop, Spark, Flink, etc.), as opposed to the more traditional data storage and access tasks associated with database systems. Host language objects provide a very natural model for more computationally-oriented tasks because it is efficient (both in terms of programmer time and execution time) to write host code in a standard programming language code that manipulates standard language objects. Programmers have found this data model to be very natural and quite high performant. One the most obvious benefits is that it obviates the need for a foreign-function interface to call out to external libraries for mathematical or statistical computing, which have for decades vexed programmers writing code in a domain-specific language such as SQL.

2.2 Self-Describing Documents

An alternative to host language objects is to use self-describing documents to store and process data. In this data model, data are logically (though not necessarily physically) stored using a self-describing document language. The two most common languages are XML [8] and JSON [12], with the latter arguably becoming more popular than the former as a modern, NoSQL database data model, due to its compactness and relative simplicity.

Notable document-based database systems such as MongoDB [9] and CouchDB [5] use JSON. Microsoft Azure supports DocumentDB [16] (which is a JSON-based NoSQL database), and Amazon DynamoDB [17] supports JSON.

2.3 Nested Data Models

Several modern Big Data management and data analytics systems provide custom, nested data models. For example, AsterixDB [4] supports ADM (which stands for "Asterix Data Model"), and Dremel [14] defines a nested, column-oriented model.

As data models, these are related to interchange formats such as JSON, allowing for data fields that are collections of sub-objects (which in turn may contain collections of sub-objects). The key difference is that in systems such as AsterixDB and Dremel, the physical implementation is decoupled from the data model. This allows for some important optimizations. For example, such systems will typically not store the schema with the data, and the schema is instead managed externally - such custom, nested data models are generally not self-describing.

3 OBJECT IMPLEMENTATIONS

In this section, we enumerate the possible implementations and implementation choices available for each of these logical models that we evaluate in this paper.

3.1 Host Language Objects

Managed Or Not? There are many degrees of freedom available to a system-designer when choosing to support host-language objects. The first crucial choice is whether to use a managed language (such as Java or Scala) or not (using a language such as C++ that is compiled into machine code, with no separate runtime). Managed languages utilize a runtime such as the Java Virtual Machine that confers many advantages; portability and automatic memory management are among the most important. But managed languages are generally slower for data processing, since automatic memory management can be expensive.

Serialization for Managed Objects. Beyond the choice of a managed language or not, there is the question of how the objects are moved between RAM and the network and/or secondary storage. Generally, host language objects are not flat, and include pointers and linked structures. However, network and secondary storage require that objects be stored as a sequence of bytes. Moving between these representations is known as *serialization/deserialization*. Many different methods exist for implementing object serialization/deserialization. For Java—undoubtedly the most popular managed language — the most obvious choice is to use the serialization/deserialization capabilities natively built into the Java object model. However, using native Java object serialization/deserialization can be slow, and so other libraries are commonly used. One of the most popular is Kryo [1], which is used by Spark.

Another, somewhat different option for Java is to use Google's Protocol Buffers (PBs) [23] framework. PBs require that the user use a domain-specific language to describe the data, and then the PB framework automatically generates code that serializes and deserializes the data into Java classes. The generated classes then serve as the host-language objects. PBs represent a general class of automatically generated serialization containers.

Another option is to hand-code object serialization. In the case of Java, we might write code to serialize the object directly into a Java ByteBuffer, the contents of which can then be written to disk or sent across a network.

In-Place Objects: The "Database-ey" Solution. One aspect that all of the aforementioned solutions have in common is that they require serialization and deserialization of objects, which necessarily uses CPU cycles often many of them as memory is allocated and deallocated, and data are copied around. This contrasts with the classical, database approach, which avoids serialization and deserialization altogether. In a classical database system, records are never extracted from pages. Rather, pages are moved into memory in their entirety, and the records come along for the ride, so to speak. The contents of the page are interpreted directly, without preprocessing the page's contents into a separate, in-memory representation. Records are written directly to pages so deserialization is not required.

Mimicking this in a system supporting host language objects is challenging, but it can be done. The main difficulty is that programmers aim to store and manipulate complicated, nested structures that in turn contain complicated, nested structures that include pointers. The main difficulty is that programmers are going to want to store and manipulate complicated, nested structures that have complicated, nested structures that include pointers. In C++, one way to do this is to override the `new ()` method so that all allocations happen directly to a memory page managed by the data management or data processing system. Thus, objects (and arrays, and linked structures) are created right on a page and need not be serialized. A difficulty, however, is following pointers or references: if a linked structure is housed on a page, what happens when the page is sent across the network to a different process with a different address space? A standard way to handle this is to create a "pointer-like" object, such as an `offset_ptr`. An `offset_ptr` always contains the absolute number of bytes from the pointer to its target, and so a dereference of an `offset_ptr` in an address-space independent way involves simply adding the contents of the pointer to the address of the pointer.

A variant of the idea of in-place objects is implemented as part of the BOOST shared-memory library, where in-place objects are used to facilitate inter-process communication.

3.2 Self-Describing Documents

A self-describing document format such as JSON tends to blur the line between a data model and a storage model. Since, by definition, JSON (and also XML) are a self-describing document formats used for data interchange, they are really meant for both purposes.

3.3 Nested Data Models

As data models, nested data models are not very different from an interchange format such as JSON, allowing for data fields that are collections of sub-objects (which in turn may contain collections of sub-objects) but in systems such as AsterixDB and Dremel, storage and data transfer is decoupled from the logical data model. It is generally assumed that the schema is not stored with the data, and is instead managed externally.

Designers of nested data models are, in theory, free to choose any implementation for their data model. In practice, there are really two main categories of implementations, both of which are functionally equivalent to one of the implementations for supporting host language objects. First, a system can implement a custom nested data model using a standard serialization/deserialization library, such as Protocol Buffers. This is the path chosen by Dremel. Second, a system can take the "database-ey" path, and eschew serialization and deserialization altogether, writing and reading objects directly from a page. There are, of course, many ways that this can be accomplished, but the performance is likely to be similar to (though probably somewhat inferior to) the in-place objects implementation described previously.

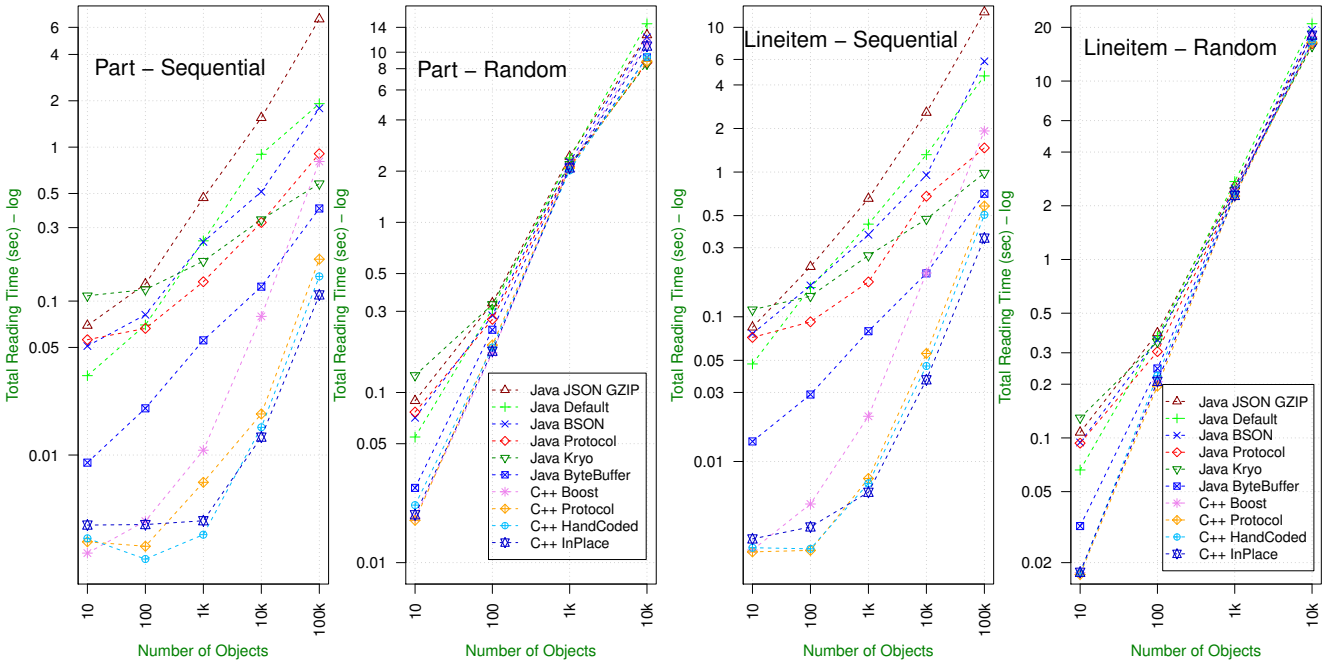

Figure 1: Reading times for `part` objects (left) and `lineitem` objects (right).

3.4 Methods Tested

Given all of these considerations, we now list the set of implementations for complex objects that we evaluated in the paper.

1. Java objects with Java de/serialization. This method simply uses built-in Java de/serialization, and serves as the straw man to which we compare the other Java-based methods.

2. Java objects with Kryo, version 3.0.3. Kryo is a common serialization package, used, for example, by Spark. Java Kryo requires the instantiation of a Kryo instance (`Kryo myKryo= new Kryo();`) and then the various types need to be registered with Kryo (`kryo.register(Part.class);`).

3. Java objects with Protocol Buffers, version 3.0.0-alpha-3.1. PBs are quite different from the other two methods, in that they require a text description of the object, which is then compiled into a set of PB classes that can then be used by the programmer (or by the data management/analytics system). In practice, the resulting PB classes are typically used directly, without conversion to/from a "regular" Java class.

4. Java objects with hand-coded using Java `ByteBuffer`. We also hand-coded our own Java Object serialization. All hand-coded serialization directly encodes the various data structures by writing bytes, ints, and doubles into a Java `ByteBuffer`.

5. Java JSON objects compressed using GZip. JSON is a common interchange format, often associated with document databases. In our implementation, we use standard Java as the in-memory representation, and then use JSON purely as an interchange and storage format. This is required since it is not possible to perform the computations required by our experiments directly on `javax.json.JsonObject` objects.

We used gzip to compress the plain text JSON objects to achieve a high compression ratio, incurring additional CPU overhead of decompression. However, for big data workloads, the choice of gzip as compression technique might not always be optimal and often a trade-off needs to be made between CPU efficiency and compression rate [2, 13].

6. Java objects with BSON, `bson4jackson` version 2.7.0 and the `FasterXML/jackson` packages.

7. C++ objects with BOOST serialization/deserialization, version 1.59. We also consider various C++ implementations of complex objects. BOOST is a classical C++ serialization package.

8. C++ objects with hand-coded serialization/deserialization. Again, this should be considered to provide an upper-bound on the performance of any C++-based method that requires serialization/deserialization.

9. C++ objects with Protocol Buffers, ver. 2.6.1. This is a C++ version of PBs, described above.

10. In-Place C++ objects. This is an implementation of in-place C++ objects, as described previously.

We have selected these implementations as being representative; many others exist (Apache Colfer, Apache Avro, Protostuff, Google Flatbuffers, DSL-JSON, Fast Serialization, Apache Thrift, and others).

4 EXPERIMENTAL OVERVIEW

In the next few sections of the paper, we will give detailed explanations of the experimental tasks we consider. As a preview, the tasks we consider are:

(1) A set of serialized objects stored externally on an SSD; the task is to read the objects into memory and deserialize them to their in-memory representation.
(2) A set of objects are stored in a large file (larger than the available RAM). The task is to perform an external sort of the file in order to perform a duplicate removal.
(3) A set of objects are partitioned across a number of machines in a network; the task is to send requests to the machines. Each machine answers the request by serializing the objects, then sending them over the network to the requesting machine.
(4) Finally, a set of sparse vectors are stored across various machines on a network. The task is to perform a tree aggregation where the vectors are aggregated over $\log(n)$ hops.

4.1 TPC-H Data Sets

For the various experiments, we use four different data sets, implemented using each of the ten different physical implementations from Section 3.4. The first three data sets are based on the TPC-H data set [11], and the fourth consists of a set of sparse vectors.

For the TPC-H-based data sets, in the order of complexity, we have `part`, `lineitem` and `customer`. Since our goal is testing implementations of complex objects, not surprisingly, `part`, `lineitem`, `customer` are large sets consisting of individual `Part`, `Lineitem` and `Customer` objects, respectively.

While these objects types are based on the TPC-H benchmark, they do not all correspond precisely to the schemas for the `part`, `lineitem` and `customer` tables in the TPC-H schema; rather, these sets contain nested object that are used to implement de-normalized, nested versions of the corresponding tables.

Our `Part` objects are the simplest, and they correspond to the `part` schema in the TPC-H benchmark:[1]
To define `Lineitem` objects, we first create a `Supplier` class:

```
class Supplier {
 int supplierKey;
 String name;
 String address;
 /* four more members... */}
```

Then `Lineitem` is:

```
class LineItem {
 Supplier supplier;
 Part part;
 int orderKey;
 int lineNumber;
 /* twelve more members... */}
```

[1]Note that we describe the objects using Java syntax; the precise definition relies to a certain extent upon the representation.

Note that rather than having foreign keys – the part sold and the supplier, we have actually nested `Supplier` and `Part` objects within the `LineItem` objects.

`Customer` objects are the most complex, as they comprise all of the `Order` objects attributes to a single customer:

```
class Order {
 List<LineItem> lineItems;
 int orderkey;
 int custkey;
 /* seven more members... */}
class Customer {
 List<Order> orders;
 int custkey;
 String name;
 /* seven more members... */}
```

Objects are nested four levels deep within the `Customer` class.

4.2 Encoding sizes

The ten different complex object implementations that we considered have very different encoding densities when the objects are serialized for storage or transmission across the network. The average, per-object sizes are given in Table 1. Interestingly, there is a lot of variance in average object size. For the smaller `Lineitem` and `Order` objects, Kryo produces the smallest storage size, whereas gzipped JSON is the best option for the larger `Customer` objects. In the case of `Customer` objects, there is nearly a $4\times$ difference in terms of average object size comparing gzipped JSON (the best option) to BSON (the worst option).

4.3 Sparse Vector Data Set

For the distributed aggregation data set, we encode the sparse vector using the following objects:

```
class Entry {
 int position;
 double value;
}
class SparseVector {
 List <Entry>;
}
```

We store one `SparseVector` object on each machine. Each object implements a 100-million dimensional vector, though before aggregation, 90% of the slots in each `SparseVector` object are unused (unused slots encode zero values). The empty slots are chosen randomly. During aggregation, the number of empty slots decrease as vectors are added together. This represents the realistic case where we have a distributed machine learning computation (such as a stochastic gradient descent) [7] over sparse data.

4.4 Experimental Details

We run our experiments on Amazon EC2 `c3.4xlarge` instances which have 16 vCPU cores, 30 GB RAM and two 160

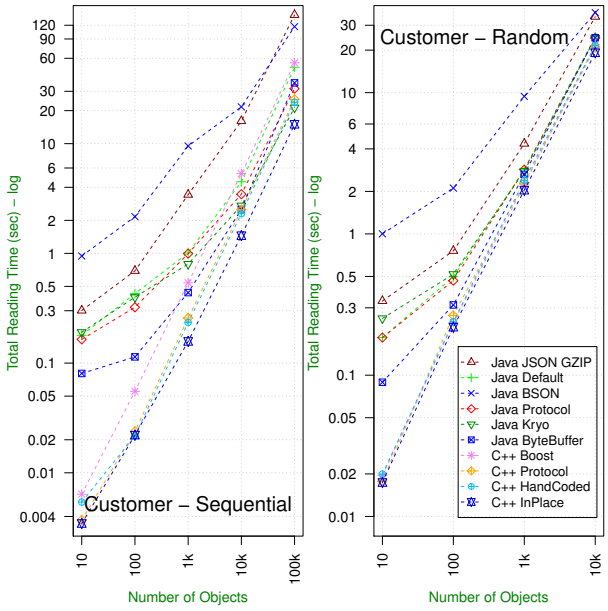

Figure 2: Reading times for `customer` objects.

Serialization Methods	Part	LineItem	Customer
Java Default	298	1037	19556
Java BSON	209	701	33879
Java JSON GZIP	196	504	8508
Java Protocol Buffer	122	398	17305
Java Kryo	114	375	16176
Java Hand Coded ByteBuffer	136	443	19478
C++ Boost	180	528	21004
C++ Protocol Buffer	126	416	17931
C++ Hand Coded	136	445	19275
C++ InPlace	174	580	25127

Table 1: Comparison of average object sizes (in bytes)

GB SSD hard disks (AWS Instance Store) running with Ubuntu Linux 14.04.4 LTS 3.13.0-74-generic x86_64. Before running each experiment task, we "warmed up" the Java Garbage Collector (GC) by creating a large number of objects. We do not include this warm-up-time in our performance time calculations. We used two Java GC flags `-XX:-UseGCOverheadLimit` and `-XX:+UseConcMarkSweepGC`. The first flag is used to avoid OutOfMemoryError exceptions while using the complete RAM size for data processing and the second flag is for running concurrent garbage collection.

We run all of our experiments 5 times and observed that the results have low variance. In this paper we present the average of those runs. Before running each experiment, we deleted th OS cache using the Linux command:
`echo 3 > /proc/sys/vm/drop_caches`.

Our Java implementation is written using Java 8 with the Oracle JDK version "1.8.0_101" and for our C++ implementation we use the C++11, compiled using GCC (version 5.4.0). The source codes of our implementation and a brief description of technical details can be found on the Github Repository [2].

4.5 Garbage Collection

All programs must use CPU cycles to perform memory management. In C++, `malloc/free` consume most of the memory-management cycles, and both run on the same thread as the user code. In Java, memory management is performed by a garbage collector, which runs on one or more separate threads within the JVM.

Java garbage collectors typically use all available user CPU cycles to reduce the wall-clock time, without regard to the overall CPU time. Hence measuring total CPU time would be measuring a quantity that the JVM was simply not optimizing for. In a resource constrained environment, Java could obtain the same (or similar) wall-clock time using a much lower CPU time.

After much thought, we decided to run all Java codes pinned to a fixed number of cores, where the number of cores is equal to the number of worker threads in the particular experiment. This is done using the Unix/Linux `taskset` command. In this way, the garbage collector and user code run on the same core, and the garbage collector is forced to either take cycles away from the user code, or to take cycles that would have been wasted while the user code is waiting for I/O, for example. In one sense, it still gives Java an unfair advantage, as Java is able to perform memory management during user I/O requests, whereas our C++ codes could not. But in another sense, Java is at a disadvantage. In a realistic Big Data scenario where many threads are running in the same JVM, all threads will share the same garbage collector, and there is some evidence that memory management for n threads is likely not n times as expensive as memory management for one thread. Given these balancing factors, it is probably the correct way to conduct the experimentation.

5 EXP. 1: I/O FROM LOCAL DISK

The first set of experiments are a set of simple, single-machine experiments. The goal is to examine how the various complex object implementations compare for a simple from-disk retrieval task. In this set of experiments, the `part`, `lineitem`, and `customer` data sets are first loaded onto the SSD drive of a single machine—one version for each of the ten complex object implementations tested–where they are organized into 256KB pages. The objects are then indexed, using a dense index.

Two experiments are run. In the first, a particular object is looked up in the index, and then enough pages are read from disk to access that object, as well as the following $n-1$ objects. As the pages are loaded into RAM, all n objects are

[2]The source code of our Implementation is available at https://github.com/kiat/serialization

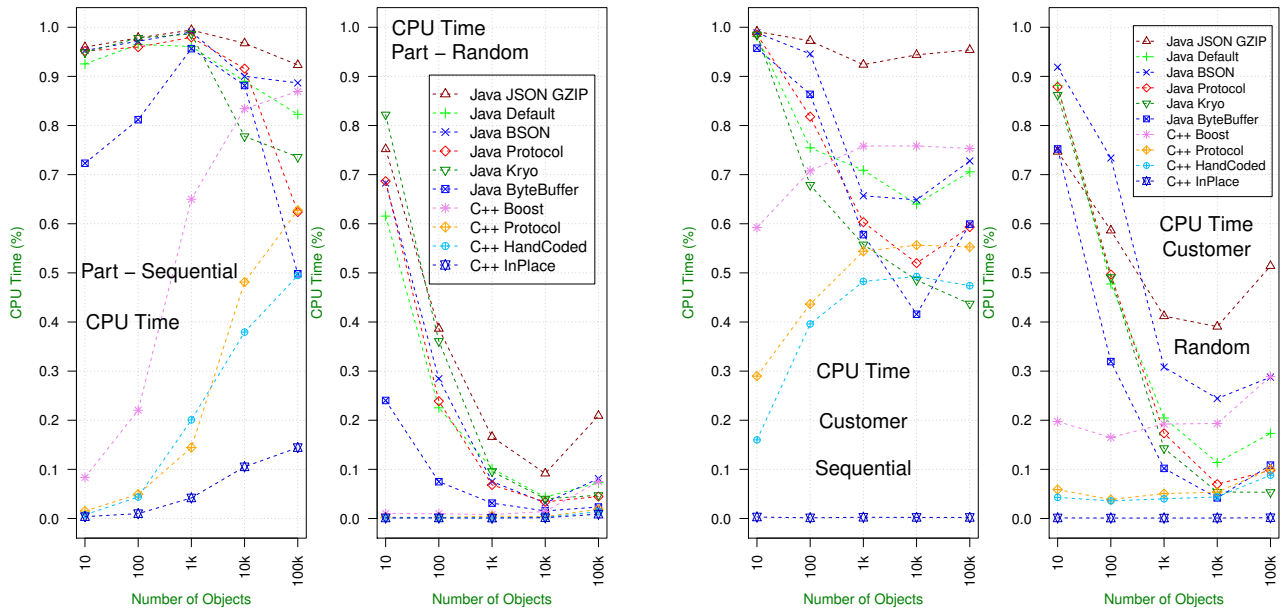

Figure 3: CPU vs. I/O time, `Customer` objects; random reads left, sequential right.

deserialized and made ready for processing. This tests the ability of the object implementation to support fast processing of objects in sequence. We test n in $\{10^1, 10^2, 10^3, 10^4, 10^5\}$.

In the second experiment, a list of n, randomly-selected objects are accessed, in order. For each object, the location of the object in the database is looked up in the index, and then the corresponding page is loaded into RAM. The desired object is then deserialized from the page. This simulates a scenario where objects are retrieved from secondary storage using a secondary index.

Before the experiment, the operating system buffer cache is emptied. We do not utilize a dedicated buffer cache, but we do allow the operating system to cache disk pages.

One concern is that since there was only one thread active at a given time during each experiment, this might give an unfair advantage to those solutions running in the JVM. One of the classical performance problems observed during garbage collection is long pauses during which worker threads are largely locked out of allocations. When only a single thread is available, serial execution is already forced, and hence the cost of such a pause is minimized. A single-threaded environment might also give an unfair advantage to the non-in-place C++ solutions, as `malloc()` and `free()` may be less of a bottleneck when only a single thread is running. Hence, we also ran a set of sequential read experiments where four threads we allowed to concurrently read and then process different ranges of the required data. This may give a better idea of the performance in a realistic, multi-threaded environment.

5.1 Results

In Figure 1 and Figure 2, we show, for each of the ten implementations, the total running time required as a function of the number of `Part`, `Lineitem`, and `Customer` objects retrieved during the single-threaded experiments. In Figures 4 and 5 we show a few of those results as bar plots, where the performance differences are easier to see; we also breakdown the total time into I/O and CPU.

In Figure 3 we show, for reading `Part` and `Customer` objects during the single-threaded experiments, the time spent waiting for the CPU (including deserialization and memory management) as a fraction of the total read time (which includes I/O time).

In Figure 6 we give a subset of the multi-threaded experimental results, showing the times required for sequential reads of various sizes for the three different data sets.

5.2 Discussion

There are a number of interesting results here. First off, the precise serialization method matters far more for sequential reads than for random reads. For random reads, only a single object from a page is used, meaning that no matter the object size, the I/O cost is constant, and huge. However, sequential reads—especially larger sequential reads—ensure that I/O is relatively inexpensive, magnifying the cost of deserialization.

One of the most surprising results is the sheer scale of the time difference between the various serialization strategies when performing sequential reads even for a single thread. For small to medium-sized sequential reads, there is consistently a 100× difference in total read time between the fastest deserialization methods—generally C++ PBs, C++ hand coded serialization, and C++ in-place—and the slowest—Java JSON `gzip`. It is difficult to over-state the significance of a 100× performance hit. A data management system can have excellent

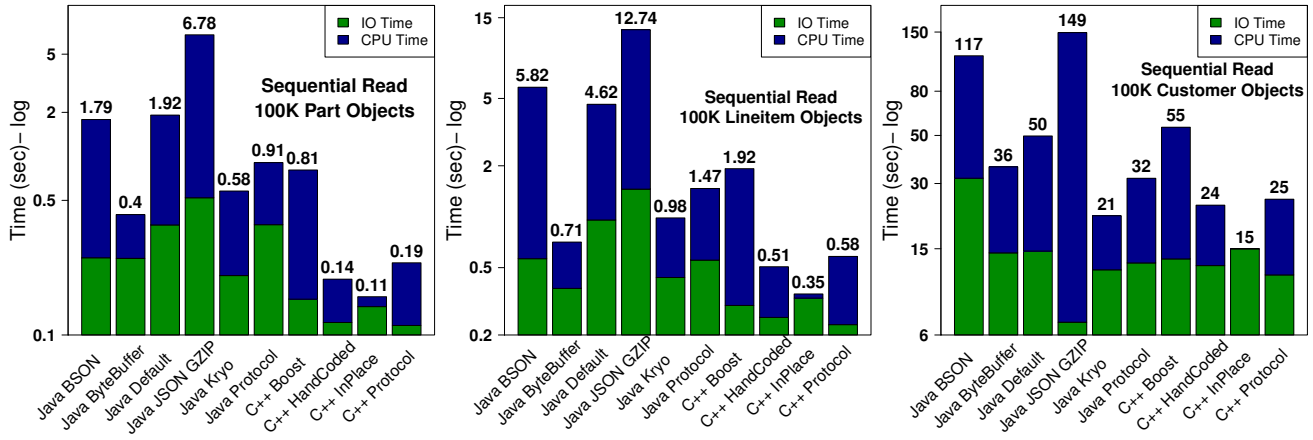

Figure 4: Total/IO time for sequential read of 100K objects; `Part` (left), `Lineitem` (center), `Customer` (right).

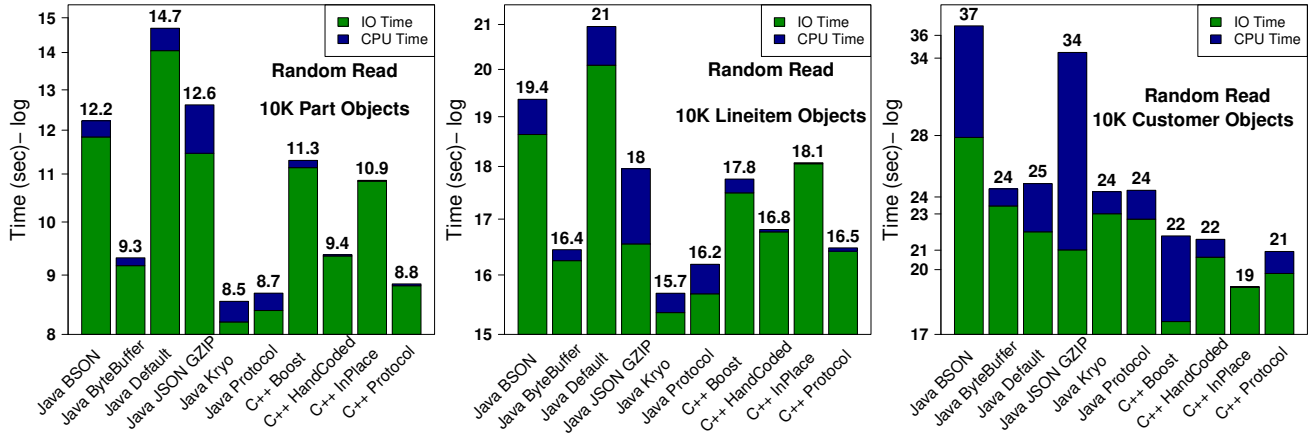

Figure 5: Total I/O time for 10K object random read; `Part` (left), `Lineitem` (center), `Customer` (right).

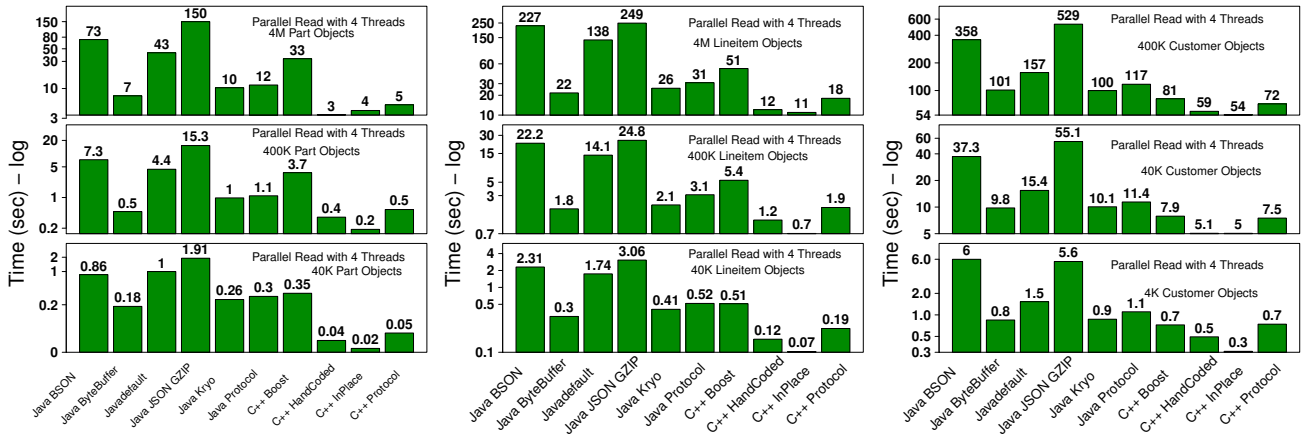

Figure 6: Sequential reads with four threads; `Part` (left), `Lineitem` (center), `Customer` (right).

indexing, query processing algorithms, and optimization/-query planning, but a 100× performance hit is going to be debilitating.

On the single-threaded experiments, the Java methods generally performed worse than the C++ methods, though there

was significant in-group variance, with C++ Boost serialization generally being outperformed by a hand-coded Java deserialization. Still, for large sequential reads, the difference between the best C++ method (in-place) and the best Java implementation (hand-coded) was approximately 4×, 2.5×, and 1.5× for large sequential reads on `Part`, `Lineitem`, and `Customer`, respectively, opening up to 20× for `Part` and 5× for `Customer`.

It is also interesting that there is generally a much bigger gap between Java and C++ for smaller, random reads (at the left of each of the plots in Figures 1 and 2) than for larger, random reads (at the right of the plots, and in detail in Figure 5). This is surprising, given that (aside from `Customer` objects, where each object takes up a reasonable fraction of a 256KB page) I/O times should dominate. We conjecture that this has to do with JIT compilation in the JVM. The first few reads incur exceeding high CPU costs for deserialization, until JIT compilation can be used to reduce the cost. This conjecture is also borne out by Figure 3, which shows a dramatic decrease in the relative CPU cost as read size increases.

Also as expected, the differences between C++ and Java become even more pronounced when looking at the multi-threaded experiments, as the cost associated with garbage collection is magnified, since all threads must share the same garbage collector. In each case, the fastest Java option (a hand-coded deserialization from a `ByteBuffer`) was twice as slow as the fastest C++ option.

It is interesting to examine the C++ in-place serialization in a bit more detail. As expected, there is almost zero CPU cost associated with C++ in-place serialization; this method is dominated by I/O time. It is, overall, the fastest method, being consistently 1.5× as fast as its closest competitor, hand-coded C++ deserialization, but it has the advantage that it can be automated and hence is a reasonable candidate for use in a Big Data system. The only reason it is not more dominant is that because objects are created in-place, they tend to be large. As depicted in Table 1, in-place `Customer` objects are 3× the size of Java JSON `gzip` objects.

6 EXP. 2: NETWORKED REQUESTS

In this experiment, we set up an eleven node cluster, mimicking a distributed database server. For the computation, we run ten machines as servers, and one machine as a client. Data are partitioned across the ten server machines, and then the client requests data from the ten machines. This mimics a situation during distributed query processing where a subset of the data must be sent to a single machine (for example, a set of data stored on that machine may need to be joined with a second set of distributed data).

The basic setup is as follows. The client spawns one thread for each of the ten servers. In parallel, those threads connect with the various servers via a socket and request subsets of the data. In response to the request, a thread on the server machine uses an in-RAM index to locate the data, and then sends the data back to the client. On the client we do not deserialize the data—we simply receive the data and write it back to disk.

Data are sent in 20 MB pages; the client aggregates batches of objects and then flushes them to disk. Each client thread creates its own data file, so locking of the output file is not necessary. All eleven Amazon EC2 instances making up our cluster are in the same Amazon Virtual Private Cloud, inside the same EC2 availability zone. They are created within the same subnet.

Timings are collected as follows. We start the timer at the client once all of the ten threads are connected to the ten servers, and once the locations of the requested objects have been computed, and a signal has been sent to begin retrieving the data.

We ran two different types of experiments. In the first, data at the server are buffered in RAM, as Java or C++ objects. In response to the request, the objects must be serialized to be sent across the network. In the second, data at the server are stored on disk. Thus they can be sent across the network without any data deserialization or serialization. In both cases, since data are already serialized when they come across the network, they are written to disk without deserialization. Thus, the second experiment does not depend on de/serialization cost.

We ran various experiments, where from each server the client requests n objects, for n in $(10^4, 10^5, 10^6, 10^7)$ though for `Lineitem` and `Custotmer`, we do not run the sized-10^7 experiment, since the data set is too large to buffer in RAM at the server side.

6.1 Results

Figure 8 depicts the time required to send data from disk. Figure 7 depicts the time required to send data from RAM.

6.2 Discussion

We begin with the most obvious observation: serialization size matters. In the disk-to-disk experiment (Figure 8) it is *all* that matters, since there is virtually no CPU cost; completing the task requires repeatedly reading serialized objects from disk, sending them across the network, and writing them at the other side. JSON `gzip` has by far the smallest `Customer` object size, and hence it has the fastest task completion time. BSON has by far the largest `Customer` object size, and hence the slowest time.

What is perhaps a bit surprising is that the task can significantly magnify the effect of object size. For example, the ratio in object size between JSON `gzip` and BSON is 1:4. But surprisingly, for sending 10K, 100K, and 1M objects, the ratio in task completion time is 1:4, 1:6, and 1:16. The explanation seems to be that for the large BSON objects, the client becomes overwhelmed as more and more objects are sent. That is, when receiving a total of 100K objects from 10 nodes (10K from each; this is around 3GB of total data), the client is able to process the objects at the speed of receipt using 10 threads. But when this increases to 1M objects (30GB of total data) the becomes overwhelmed due to the large amount of disk and network I/O, and times skyrocket.

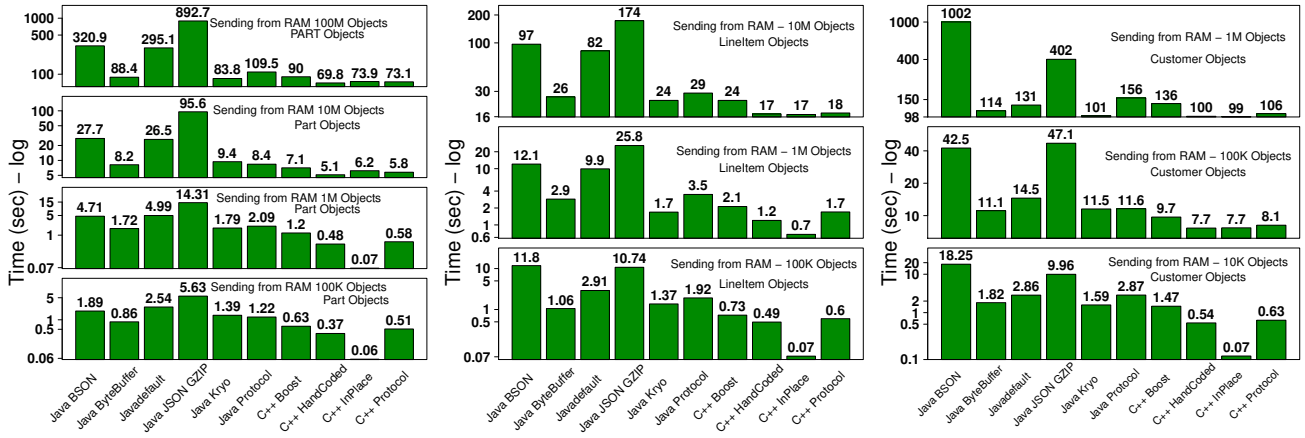

Figure 7: Network requests; data in RAM. `Part` objects (left), `Lineitem` objects (center), `Customer` object (right).

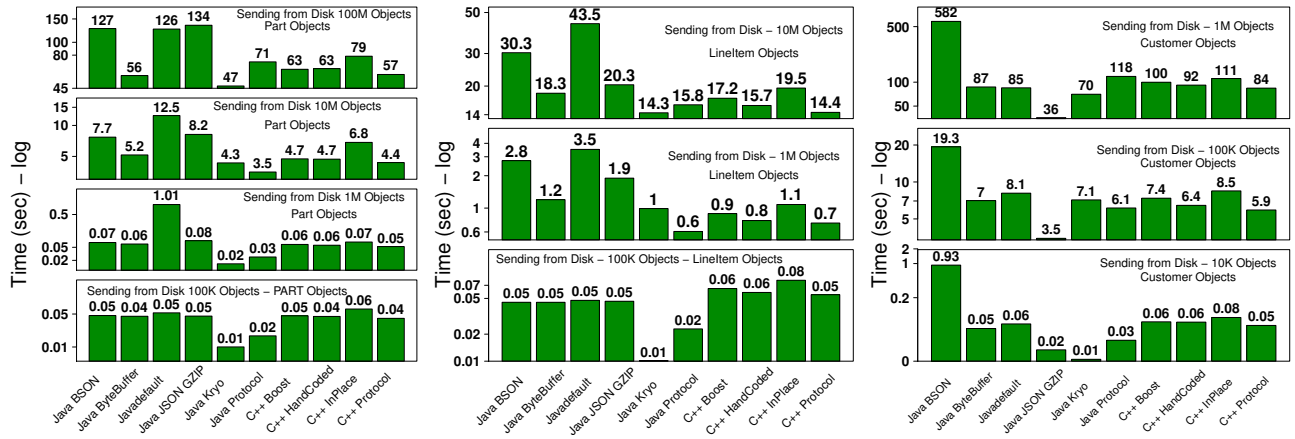

Figure 8: Network requests; data on disk. `Part` objects (left), `Lineitem` objects (center), `Customer` object (right).

However, as soon as one factors in the cost to serialize data, serialization size matters a lot less. Consider Figure 7. C++ in-place serialization (which has no serialization cost) is consistently the fastest across all experiments (the one exception being larger `Part` requests)—sometimes dramatically so. This is despite the fact that the C++ in-place implementation results in relatively large objects.

One of the most surprising findings is that for smaller requests, C++ in-place can be *five to seven times faster* than its closest competitor, if data are stored, de-objectified, in RAM. The reason for this is that if all objects being sent can fit on a single page, then no round-trip messages are required. That is, the client need only receive a page of objects from each server and write those pages, at which time the task ends. In such a scenario, serialization turns out to be the *dominant* cost. Elimination results in radical speedups.

7 EXP. 3: EXTERNAL SORT

In this section, we describe an experiment concerning one of the most fundamental computations performed in data management: sorting. Specifically, in three sub-experiments, we consider sorting a large number of `Part`, `Lineitem`, and `Customer` objects with the goal of performing duplicate removal. For `Part`, `Lineitem`, and `Customer`, we create data sets of 240 million, 120 million, and 3 million objects respectively. Depending upon the serialization method, this resulted in a different-sized file to perform duplicate removal on, though all files were larger than the available main memory; the approximate file size can be computed by multiplying the data set size by the average object size in Table 1.

For each object type, we check for duplicate objects by deserializing the object, and then checking for equality of each of the fields, including sub-objects. Since each file is too large to fully deserialize into memory all at once, all sorts are implemented using a two-phase multi-way merge sort. In the first

phase, the input file is read in chunks that consume all available RAM; each chunk is sorted and written out as a sorted run. In the second phase, pages are objects are read from those sorted runs and inserted into an in-memory priority queue that is used to perform the final sorting and duplicate removal.

7.1 Results

Results are given in Figure 9. The height of each bar indicates the total duplicate removal time. Each total time is broken into I/O time and CPU time. Note that the *y*-axis of each plot is in logarithmic scale.

7.2 Discussion

These results are remarkable to the extent that they show how important the selected serialization/method is to obtaining high performance for an external sort. For example, the fastest sort for `Part` objects—C++ in-place—took 0.38 hours. The slowest sort on part (Java JSON `gzip`) took 14.3 hours. This is almost a 38× difference between the two methods. The algorithms are the same—the difference is the language, the runtime, and the serialization methodology. This alone can result in a 38× difference. Even among Java-based methods, the fastest (hand-coded Java) was nearly 21 times faster than Java JSON `gzip`.

We also found it interesting that serialized object size (and hence I/O time) was a relatively unimportant predictor of sort speed. Serialization/deserialization cost appears to be far more important. Consider sorting of `Customer` objects. C++ in-place objects had the largest serialized size (except for BSON object) and hence had the highest I/O cost (again, except for BSON). And yet, C++ in-place sorting of `Customer` was by far the fastest of all of the methods tested, due to the sort algorithm having negligible CPU cost. In comparison, fastest Java implementation, using Kryo, took 66% more time to sort the `Customer` objects, despite the fact that the I/O time for Kryo was only half of the I/O time of C++ in-place. Java JSON `gzip` had by far the lowest I/O time, but it had by far the greatest overall time.

8 EXP. 4: AGGREGATION

In this set of experiments, we consider the problem of tree-structured distributed aggregation. We logically arrange seven machines in a binary tree of seven nodes. To perform distributed aggregation, the four nodes at the leaves of the network send data to be aggregated to their parents, which receive two objects, one from each child. Those parents aggregate the data obtained from the two children with their own data, and then send the result to the root, which then aggregates the result obtained from *its* two children, along with its own data. The result of this is then the final result of the distributed aggregation.

In our experiments, the items to be aggregated are sparse vectors of 100M *double* values; we are trying to compute a summation of all of those values across the network. This simulates the aggregation that must be performed, for example, when computing a distributed gradient descent to learn a logistic regression model over a large set of sparse data. The vector to be aggregated at each node would be the result of summing all of the individual vectors.

Each node in the cluster starts with a 100 million-entry long sparse vector. We experiment with different densities for this vector: 0.1% of the entires are filled at each node, 1% of the entries are filled and 10% of the entries are filled. The entries that are initially filled are chosen randomly. Hence, as the vectors are aggregated, more and more of the entries have data, so that after the last aggregation, the probability that an arbitrary entry in the 0.1% experiment is empty is $1 - (1 - 10^{-3})^7$, or 99.3%.

8.1 Results

Figure 10 depicts the results of our sparse vector aggregation experiment. Note that the *y*-axis of the plot is in logarithmic scale.

8.2 Discussion

The most striking observation is not unique to this experiment: though JSON `gzip` generally allows for very small encodings, it is a terrible choice due to the tremendous CPU overhead. It is up to 100 times slower than the fastest options. This underscores an important observation across all of the experiments: encoding size can be important, but small I/O is not the only consideration, nor it is even the most important consideration.

C++ hand-coded is generally the fastest, but that is because for this application, it is nearly equivalent to C++ in-place. Unlike the case of the `Part`, `Lineitem`, and `Customer` data sets, which contain variable-sized, recursive data structures the most natural hand-coded implementation is in-place as well. To aggregate incoming vectors, the hand-coded implementation directly interprets the bytes in the incoming stream, and to write out each entry in the result, a single integer and a double are written to the output stream. Thus, there is no de/serialization overhead, and no overhead associated with `offset_ptr`s and the like, which are used in the in-place implementation.

It is also interesting that for the sparsest vectors, the C++ implementations were all 2× to 3× faster than the Java implementations, and this holds even for the hand-coded, Java `ByteBuffer` implementation (in 0.1% and 1% density cases). This was quite surprising given the simplicity of the task, which means that the `ByteBuffer` implementation was essentially an in-place Java implementation, with expectedly little or no garbage collection overhead. This advantage disappears with increasing density in the vector, where Java `ByteBuffer` becomes just as fast as the fastest C++ implementation. One may conjecture that this illustrates the benefit of JIT compilation. The core computation here is a tight loop that performs memory lookups and floating-point additions, where JIT compilation would expectedly be very helpful.

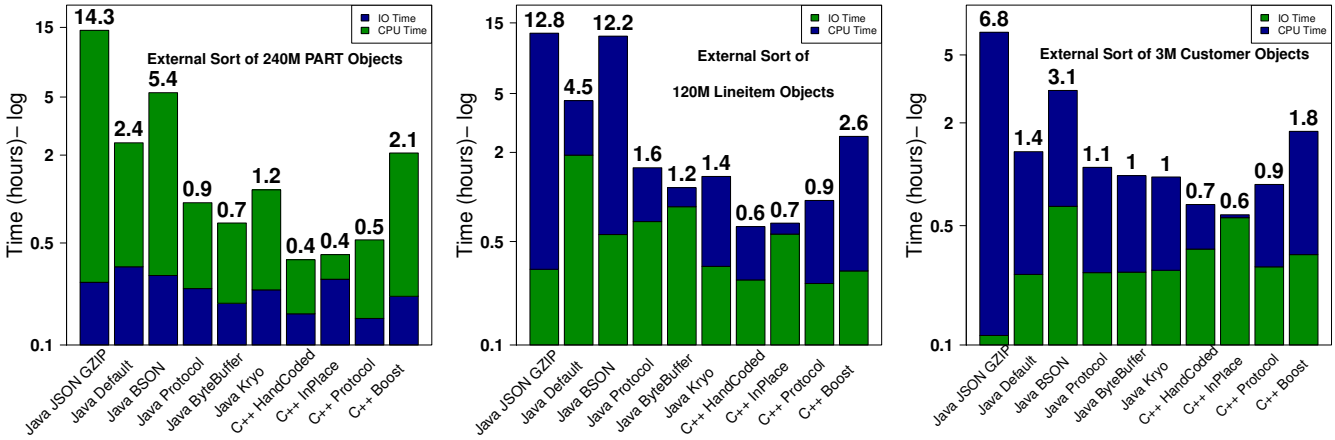

Figure 9: External sort times; `Part` (left), `Lineitem` (center), `Customer` (right).

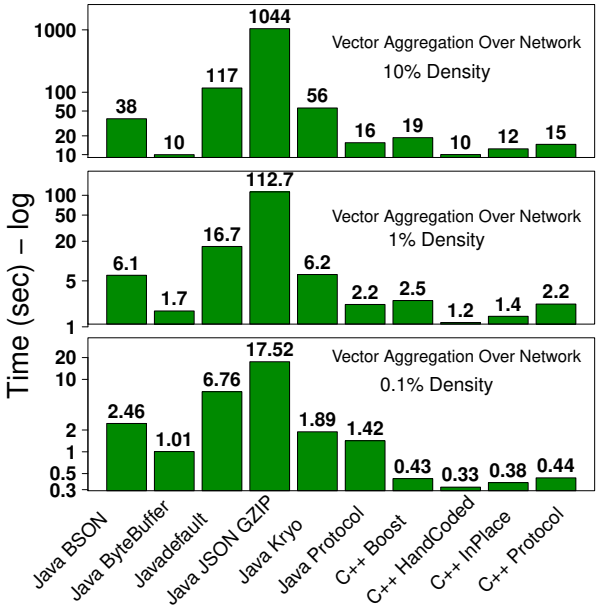

Figure 10: Sparse Vector Aggregation over Network (`7 Nodes`).

9 SUMMARY

There are significant costs associated with manipulating objects on a managed infrastructure (Java) as opposed to an unmanaged infrastructure—directly on top of Linux, using C++. Further, we found that storage and transmission of objects using an interchange format such as JSON imposes significant additional costs. In some ways our results are not surprising; one would expect some cost associated with using a managed environment.

What *is* surprising is the potential magnitude of the costs. For large, multi-threaded sequential reads of the most complex `Customer` objects, the fastest Java solution—hand-coded de/serialization with data represented as in-memory Java objects—takes twice as long to read and deserialize compared to the fastest C++ solution (in-place objects), even though the C++ solution has greater I/O requirements. This difference increases significantly for smaller reads of less complex objects. For external sorts, which in theory should be I/O bound, the fastest Java solutions are still twice as expensive as the in-place C++ solution, despite the in-place solution's high I/O requirements. These results suggest that choosing to implement a Big Data solution for complex objects in Java (as opposed to an un-managed language) will result in a 2× performance hit off-the-bat, with the potential cost being even worse.

We point out that our results serve to affirm the value of the classical, "database" way of doing things where there is no distinction between the in-memory and over-the-wire data representation, and where memory management is done "in the large" via paging, rather than "in the small", by garbage collection. The exceedingly low CPU cost associated with manipulating in-place objects—the closest analog that we tested—more than compensates for their somewhat larger size. This flies in the face of what some may consider to be the conventional wisdom: reduce I/O costs as much as possible, even at the cost of increasing CPU using compression. To be fair, this conventional wisdom comes mostly from word on columnar relational database systems [18], which are a different type of system from what we have considered. And we did not explicitly consider the possibility of using compression along with in-place objects. But this paper has shown that CPU costs are significant, even when performing I/O intensive tasks, and the decision to increase CPU costs even more should not be taken lightly.

ACKNOWLEDGMENTS

Material in this paper has been supported by the NSF under grant nos. 1355998 and 1409543, and by the DARPA MUSE program, award #FA8750-14-2-0270.

REFERENCES

[1] 2016. Kryo Serialization Package. https://github.com/EsotericSoftware/kryo. (2016).
[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. ACM, 671–682.
[3] Alexander Alexandrov et al. 2014. The Stratosphere platform for big data analytics. *VLDBJ* 23, 6 (2014), 939–964.
[4] Sattam Alsubaiee et al. 2014. AsterixDB: A Scalable, Open Source BDMS. *VLDB Endow.* 7, 14 (Oct. 2014), 1905–1916. https://doi.org/10.14778/2733085.2733096
[5] J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: the definitive guide*. "O'Reilly Media, Inc.".
[6] Malcolm P Atkinson et al. 1989. The Object-Oriented Database System Manifesto.. In *DOOD*, Vol. 89. 40–57.
[7] Léon Bottou. 2010. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT'2010*. 177–186.
[8] Tim Bray et al. 1998. Extensible markup language (XML). *W3C REC-xml-19980210. http://www.w3. org/TR/1998/REC-xml-19980210* 16 (1998), 16.
[9] Kristina Chodorow. 2013. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.".
[10] Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.
[11] Transaction Processing Performance Council. 1999. The TPC Benchmark (TPC-H) A decision support benchmark for an Ad-hoc Decision Support System. http://www.tpc.org/tpch/. (1999).
[12] Douglas Crockford. 2006. The application/json media type for javascript object notation (json). (2006).
[13] Avrilia Floratou, Jignesh M Patel, Eugene J Shekita, and Sandeep Tata. 2011. Column-oriented storage techniques for MapReduce. *Proceedings of the VLDB Endowment* 4, 7 (2011), 419–429.
[14] Sergey Melnik et al. 2010. Dremel: Interactive Analysis of Web-scale Datasets. *VLDB Endow.* 3, 1-2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886
[15] Chris Parnin, Christian Bird, and Emerson Murphy-Hill. 2013. Adoption and use of Java generics. *Empirical Software Engineering* 18, 6 (2013).
[16] Dharma Shukla et al. 2015. Schema-agnostic indexing with Azure DocumentDB. *VLDB Endow.* 8, 12 (2015), 1668–1679.
[17] Swaminathan Sivasubramanian. [n. d.]. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *SIGMOD 2012*.
[18] Mike Stonebraker et al. 2005. C-store: a column-oriented DBMS. In *VLDB Endow.*
[19] Michael Stonebraker and Dorothy Moore. 1995. *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc.
[20] Jimmy P. Strickland, Peter P. Uhrowczik, and Vern L. Watts. 1982. IMS/VS: An evolving system. *IBM Systems Journal* 21, 4 (1982), 490–510.
[21] Bjarne Stroustrup. 1996. A history of C++: 1979–1991. In *History of programming languages—II*. 699–769.
[22] Robert W Taylor and Randall L Frank. 1976. CODASYL data-base management systems. *ACM CSUR* 8, 1 (1976), 67–103.
[23] Kenton Varda. 2008. Protocol buffers: Google's data interchange format. *Google Open Source Blog* (2008).
[24] Tom White. 2009. *Hadoop: the definitive guide*. O'Reilly Media, Inc.
[25] Matei Zaharia et al. 2010. Spark: cluster computing with working sets. In *USENIX HotCloud*. 1–10.