RICE UNIVERSITY

An Experimental Comparison of Complex Objects Implementations in Big Data Systems

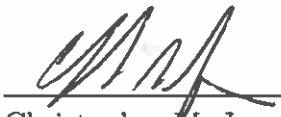
by

Sourav Sikdar

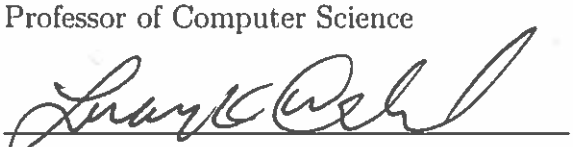
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

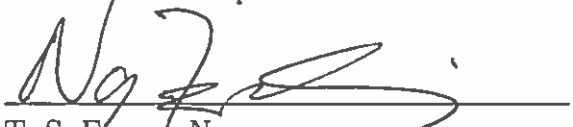
Approved, Thesis Committee:



Christopher M. Jermaine, *Chair*
Professor of Computer Science



Luay K. Nakhleh
Professor of Computer Science



T. S. Eugene Ng
Professor of Computer Science

Houston, Texas

December, 2016

Abstract

An Experimental Comparison of Complex Object Implementations in Big Data Systems

by

Sourav Sikdar

Many 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. 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. That said, performance is just one of the considerations a system designer must keep in mind, and many popular systems are built upon technologies such as JSON. The question I ask in this thesis 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 thesis is an attempt to remedy that.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Chris Jermaine for his guidance, encouragement and support. His expertise and insights were indispensable for this research work. His guidance and motivation were invaluable towards completion of this research and thesis.

Besides my advisor, I would like to thank the rest of my thesis committee members: Prof. Luay Nakhleh and Prof. Eugene Ng for their insightful comments and valuable feedback on the thesis.

The research presented in this thesis was a collaborative effort and this would not have been possible without the help and support of Dr. Kia Teymourian, my collaborator and mentor.

I would like to thank my group members, Risa, Jia, Matt, Carlos, Jacob, Shangyu, Letao and Dimitrije for their helpful discussions and suggestions.

I would like to thank my fellow graduate students in Duncan Hall: Peng, Suguman, Sriraj and others for the insightful discussions about broader research topics.

I have been fortunate to have wonderful friends like Sagnik Dasgupta, Ankush Mandal, Rohan Mukherjee, Rabimba Karanjai, Arkabandhu Chowdhury, Hamim Zafar and Peshal Nayak. I would also like to thank my other friends all of whom I cannot mention here.

Finally, I would like to thank my parents, Mr. Mrinal Kanti Sikdar and Mrs. Sumitra Sikdar. Without their love, support and encouragement, I would not be able to pursue my dreams. I would also like to thank my extended family, especially my cousins, for supporting my decisions.

Contents

Abstract i

1 Introduction **2**

2 Related Work **8**

3 Complex Object Models **11**

3.1 Host Language Objects 11

3.2 Self-Describing Documents 12

3.3 Nested Data Models 14

4 Object Implementations **15**

4.1 Host Language Objects 15

4.2 Self-Describing Documents 19

4.3 Nested Data Models 20

4.4 Methods Tested 21

5 Experimental Overview **25**

5.1 TPC-H Data Sets 26

5.2 Encoding sizes 28
5.3 Sparse Vector Data Set 29
5.4 Experimental Details 30
5.5 Garbage Collection 31
5.6 Collecting Timings 33

6 Experiments **34**

6.1 Experiment 1: I/O From Local Disk 34
6.2 Experiment 2: Networked Requests 51
6.3 Experiment 3: External Sort 60
6.4 Experiment 4: Aggregation 64

7 Conclusions **69**

List of Figures

6.1 **Reading times for `part` objects.** 36

6.2 **Reading times for `lineitem` objects.** 37

6.3 **Reading times for `customer` objects.** 38

6.4 **Total/IO time for sequential read of 100K `Part` objects** . . . 39

6.5 **Total/IO time for sequential read of 100K `LineItem` objects** . 40

6.6 **Total/IO time for sequential read of 100K `Customer` objects** . 41

6.7 **Total/IO time for random read of 10K `Part` object** 42

6.8 **Total/IO time for random read of 10K `LineItem` object** . . . 43

6.9 **Total/IO time for random read of 10K `Customer` object** . . . 44

6.10 **CPU vs. I/O time; `Part`** 45

6.11 **CPU vs. I/O time; `Customer`** 46

6.12 **Sequential reads with four threads; `Part`** 47

6.13 **Sequential reads with four threads; `LineItem`** 48

6.14 **Sequential reads with four threads; `Customer`** 49

6.15 **Network requests; data in RAM. `Part`** 54

6.16 **Network requests; data in RAM. `LineItem`** 55

6.17 **Network requests; data in RAM.** `Customer` 56
6.18 **Network requests; data in Disk.** `Part` 57
6.19 **Network requests; data in Disk.** `LineItem` 58
6.20 **Network requests; data in Disk.** `Customer` 59
6.21 External sort times for `Customer` objects. 60
6.22 **External sort times;** `Part` 63
6.23 **External sort times;** `LineItem` 64
6.24 **External sort times;** `Customer` 65
6.25 Sparse Vector Aggregation over Network `(7 Nodes)`. 66

List of Tables

5.1 Comparison of average object sizes (in bytes) 29

Chapter 1

Introduction

Classically, relational data management systems supported records made of flat types. Using Java-like syntax, relational [1] and pre-relational systems [2] generally stored records like:

```
class Part {
        public long partkey;
        public String name;
        public String mfgr;
        public String brand;
        public String type
        public int size;
        public decimal relailprice;
}
```

Objects might have links between them, either implicitly (as in the relational model) or explicitly (in hierarchical database system [3]) but they were essentially flat.

Over time, various efforts were aimed at extending this simple model—most notably object-oriented [4] (or object-relational [5]) 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 [6] and Java lacked generics until 2004 [7]) 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 [8–14]. 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. Outside of data management systems, programmers are used to writing code such as:

```
class DataDependencyGraph {
        public ElementInfo header;
        // the list of nodes
        public Map<Integer, NodeInfo> nodes;
        // directed edges in the graph
        public Map<Integer, Integer> edges;
}
```

Conceptually, this object represents a small, self-contained graph. Programmers increasingly expect the ability to manage such complex objects within modern data management systems as well. If the task is to store and analyze a large number of such graphs, a typical engineer is going to want to store each within a single database object or record, and is going to be hesitant to listen to abstract arguments touting the benefits of normalizing (breaking apart) the individual graphs.

Objects In Modern Systems. In response, some modern data processing systems (such as Hadoop [8], Spark [9], DryadLinq [10], or Flink [11] 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 by the data processing software, they go through a separate serialization/deserialization process that likely involves (at least an indirect) invocation of a garbage collector as well as many allocations on the receiver side.

There are other ways of supporting complex types. Some popular data management systems (such as MongoDB [14]) use a standard, self-describing interchange format—the most popular being JSON [15]—as the basic data or storage model. JSON is hierarchically organized, and allows for sets of values. Standard JSON tools are then used to provide the on-disk or in-network serialization format for records/objects. When they are moved into memory they are then fully parsed into an internal representation before processing.

Still other systems such as Dremel [12] use their own data definition language, but use a high-level language and runtime along with a tool, such as Protocol Buffers (PBs) [16] for serialization and deserialization.

At What Cost? The constant across all of these methods for supporting complex objects is that they are all significant departures from the classical "database" way of doing things. Classically, a relational database brings in a page from disk via I/O performed into a buffer, and then the system processes data directly from the newly-read page. True, some sort of parsing is almost always required—for example, in the case of a textual attribute in a record, at a minimum it is necessary to read an integer encoding the length of the text so it is possible to know where the next attribute value begins. But using the database approach, there is no need to "objectify" or otherwise pre-process the text. If the data are ever used, they are simply processed, in place, with no translations or extra deserialization steps, and no garbage collections. "Deallocating" a page in a classical database system simply means writing a new page on top of an old one.

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 I ask in this thesis 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 data management 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.

My Contributions. Across a variety of data management tasks, I experimentally compare different programming 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). I offer a detailed account of the pros and cons of the various approaches, with a special emphasis in measuring the relative costs of the various approaches on realistic tasks in a networked environment. The results should be useful to anyone looking to implement a new system with the goal of managing complex objects.

Some specific contributions of my thesis are:

- I compare ten different complex object implementations across four different, representative data management tasks, from simple object reads, to an external sort for duplicate removal, to networked computations.

- I 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. I also consider the task of implementing

a sparse vector for a statistical computation.

- I observe 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 best solutions in a managed environment (Java) are generally at least twice as slow as the best solutions running directly on the operating system, implemented in C++.

These results suggest that object implementation is one of the most important issues facing 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.

Chapter 2

Related Work

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. The primary goal of this thesis is to alleviate that problem. However, there exists literature that compares performance of serialization frameworks in real world circumstances. This chapter briefly reviews some the prior experimental work in this regard.

Serialization performance has been evaluated indirectly in the performance analysis of distributed object models in communication middle-ware. Conventional communication middle-ware when used over high-speed gigabit networks face considerable overhead making them unfit for performance sensitive tasks. With the popularity of high-speed gigabit networks, programmers resort to lower-level mechanisms to achieve desired performance. Gokhale et al. [17], [18], and [19] have evaluated the performance of CORBA middleware for the purposes of remote method invocations (RMI).

Java programming language and development environment provides remote method invocations (RMI) to programmers, using which objects on different computers can interact in a distributed network. Java remote method invocation uses serialization to pass objects between JVMs, either as return values from a method invocation or as arguments in a method invocation from a client to a server. The overhead problems of remote method invocations have been addressed in [20], where serialization occurs under the hood.

Marjan et al. [21] focuses on performance analysis of XML and binary serialization frameworks build into Java and .NET platforms and assess their efficiency in terms of time and space. Java binary serialization outperforms .NET binary serialization because obtaining data through reflection is more efficient in Java, and .NET spends a lot of time in creation of objects by deserialization. However, for XML serialization .NET is more efficient than Java, primarily because .NET parser is more efficient than Java parser for XML. The study also provides direction for future improvements.

Maeda [22] compares a suite of twelve object serialization frameworks in XML, JSON and binary formats from qualitative and quantitative aspects. The serialization frameworks include: XStream in XML; Jsonlib, XStream, FlexJson, JsonMarshaller, Jsonic, Avro, Gson, JsonSmart, Thrift in JSON; Avro, Thrift in binary; Protostuff with static and dynamic schema and Java default serialization. The frameworks are compared for resulting object sizes as well as execution times. However, the only task considered in this study is de/serialization of data from local disk. There is no multipurpose "best solution" since each framework works well enough for the context it was developed.

Chapter 3

Complex Object Models

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

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

Each of them are described briefly.

3.1 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.

Our own belief is that support for host language objects led directly to the immense popularity of Hadoop, and later refinements to this model were at least partially responsible for the popularity of next-generation platforms such as DryadLINQ, Spark and Flink. Programmers have found this data model to be very natural and quite high performant. One the most obvious benefits of this approach 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.

3.2 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 [23] and JSON [15], 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 [14] and CouchDB

[24] use JSON. Microsoft Azure supports DocumentDB [25] (which is a JSON-based NoSQL database), and Amazon DynamoDB [26] supports JSON. Increasingly, classical relational database systems such as Oracle and Postgres also support native JSON storage.

JSON objects or "documents" are essentially lists of (attribute, value) pairs. Attributes are represented by strings, and values can be numbers, strings, booleans, arrays, or objects; hence, JSON allows for objects built up from other objects. An example JSON document is:

```
{
        "fname": "John",
        "lname": "Smith",
        "siblings": [
                {
                        "fname": "Jen",
                        "lname": "Smith"
                },
                {
                        "fname": "Tom",
                        "lname": "Smith"
                }
        ]
}
```

3.3 Nested Data Models

Several modern Big Data management and data analytics systems provide custom, nested data models. For example, AsterixDB [13] supports ADM (which stands for "Asterix Data Model"), and Dremel [12] 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.

Chapter 4

Object Implementations

In this section, I enumerate the possible implementations and implementation choices available for each of these logical models that I evaluate in this thesis, and give the final list of implementations that are experimentally evaluated in the thesis.

4.1 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

garbage collection and other tasks associated with 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*, and is one of the most computationally intensive aspects of using host-language objects.

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 [27], which is used by Spark.

Another, somewhat different option for Java is to use Google's Protocol Buffers (PBs) [16] 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 that are manipulated by the programmer. 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. A similar hand-coded deserialization would then be used to retrieve the objects.

Serialization for Un-Managed Objects. Arguably, the C/C++ language family is still the most popular for un-managed computing. Since C is not a object-oriented language, it has no support for objects, making C++ objects the most viable option.

For C++ objects, there is no default object serialization/deserialization option, since C++ has no reflection. That is, a compiler implementing the C++11 standard does not give the programmer programmatic access to the structure of an object, so the object can be automatically serialized/deserialized by the program.

A common approximation to this is to use something like the BOOST serialization framework [28], which requires a programmer to annotate a C++ class using macros, which are then expanded to generate serialization/deserialization code.

Just as with Java, it is also possible to write custom C++ code that directly serializes to memory; it is also possible to use a solution such as PBs that automatically generates a serialization format.

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 pre-processing the page's contents into a separate, in-memory representation. Since records are written directly to pages, deserialization is not required, either; the page is simply written back to disk or sent across the network and the records are moved as a result.

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.

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 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. In this way, *in-place objects require no serialization and deserialization or memory allocation or freeing.* They can be accessed as soon as the page holding them is moved into RAM, and once the page is swapped out, they are gone. 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.

Google's Flat Buffers [29] is a solution halfway in-between in-place objects and a serialization package such as Protocol Buffers. Flat buffers do not need to be deserialized, as their in-memory representation is fully readable. But a true in-place solution should also allow for write-in-place (including modification) so that there is no need to serialize, either.

4.2 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.

There are three standard approaches to actually implementing self-describing objects such as JSON documents.

The first is to simply represent the JSON document as a text string and to store the string. This has the obvious disadvantage of resulting in relatively large storage sizes and long I/O times, and so the second (and more common) option is to compress the string using a standard compression algorithm before sending the object over a wire, to the network or to the disk. The obvious cost of this is that it requires more CPU to compress/decompress the data.

It is also possible to design a special-purpose storage and data transfer implementation for JSON, so that the over-the-wire encoding is not something that is a

textual JSON document (or a compressed, textual JSON document). This can help to alleviate the very high cost of writing out and then re-parsing text-based JSON documents, at the cost of having a representation that is not human readable. The most common such encoding is BSON [30], which is used by MongoDB.

4.3 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. Thus, a custom nested data model allows for a number of possible implementation approaches.

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.

The performance is likely to be similar since in-place objects do not require serialization and deserialization. However, if a user specifies a nested data model using a domain specific language, there is the problem of interpreting the bytes at runtime. In the case of in-place objects, the code for interpreting the objects is supplied by the user, and compiled into machine code by a C++ compiler. In the case of a nested data model, the schema for the object is typically going to be used to interpret the object at runtime, leading to a higher CPU cost. There are exceptions to this (for example, TupleWare [31] generates code LLVM code), but in general, in-place objects should be seen as something of an upper-bound on the performance of any system-specific implementation.

4.4 Methods Tested

Given all of these considerations, I now list the set of implementations for complex objects that I evaluated in the thesis. Details of how each of these were used are included in the Appendix of the thesis.

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);`) one time at startup. This registration enables fast subsequent de/serialization.

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. As I describe in the Appendix, the PB classes are instead converted into our own Java classes.

4. Java objects with hand-coded using Java `ByteBuffer`. I have also hand-coded our own Java Object serialization. This should serve as something of an upper bound on the performance of any Java method that does not use the Java "unsafe" interface (which has perennially faced removal with each subsequent release of Java). All hand-coded serialization directly encodes the various data structures by writing bytes, ints, and doubles into a Java `ByteBuffer`, and has expectedly higher-performance than any automated serialization method.

5. Java JSON objects compressed using GZip. JSON is a common interchange format, often associated with document databases. In my implementation, I use standard Java as the in-memory representation, and then use JSON purely as in 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.

6. Java objects with BSON, `bson4jackson` version 2.7.0 and the `FasterXML/jackson` packages. Again, I use standard Java objects as our in-memory representation.

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. Such an implementation is analogous to classical database implementation techniques, where there is no distinction between the in-memory and on-disk versions of records. In-place objects are allocated directly to a page by overriding the `new` operator.

The above major implementations of complex objects are chosen after careful consideration. However, there exist more libraries like Apache Colfer[1], Apache Avro[2],

[1] `https://github.com/pascaldekloe/colfer`
[2] `https://avro.apache.org/`

Protostuff[3], Google Flatbuffers[4], DSL-JSON[5], Fast Serialization[6], Apache thrift[7] and Java unsafe that support implementation of complex objects. However, there is no loss of generality, since most of these libraries fall in either of the three broad categories discuused above and are similar in approach or performance to at least one of our selected methods.

[3] http://www.protostuff.io/
[4] https://google.github.io/flatbuffers/
[5] https://github.com/ngs-doo/dsl-json
[6] https://github.com/RuedigerMoeller/fast-serialization
[7] https://thrift.apache.org/

Chapter 5

Experimental Overview

At the highest level, this thesis considers a large space of possible implementations for complex objects, and experimentally examines how those implementations affect performance of various representative tasks that might be performed in a data management or data processing system.

In the next few sections of the thesis, I will give detailed explanations of the experimental tasks I consider. As a preview, the tasks I 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.

5.1 TPC-H Data Sets

For the various experiments, I 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 benchmark data set [32], 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 data models and implementations for 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]

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

```
class Part {
        int partID;
        String name;
        String mfgr;
        /* six more members... */}
```

The definition of `Lineitem` objects relies on `Supplier` objects:

```
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... */}
```

Note that rather than having foreign keys – the part sold and the supplier, `Supplier` and `Part` objects are actually nested 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... */}
```

Thus, objects are nested four levels deep within the `Customer` class.

To actually create the data, I use the TPC-H data generator (`dbgen`) and then de-normalize the generated data to create `lineitem`, `order` and `customer` data sets.

5.2 Encoding sizes

The ten different complex object implementations that I 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 5.1. 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`, there is nearly a 4×

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 5.1: Comparison of average object sizes (in bytes)

difference in terms of average object size comparing gzipped JSON (the best option) to BSON (the worst option).

5.3 Sparse Vector Data Set

For the distributed aggregation data set, the sparse vector is encoded using the following objects:

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

One `SparseVector` object is stored 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 decreases as vectors are added together. This represents the realistic case where we have a distributed machine learning computation (such as a stochastic gradient descent) [33] over a very sparse data set.

5.4 Experimental Details

I run my experiments on Amazon EC2 `c3.4xlarge` instances which have 16 vCPU cores, 30 GB RAM and two 160 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, the Java Garbage Collector (GC) is "warmed up" by creating a large number of objects. However, this warm-up-time is not included in our time calculations. I 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 in the background.

All of the experiments described are performed 5 times. In this thesis, I present the average of 5 separate runs for each experiment, unless otherwise mentioned.

Before running each experiment, the OS cache is deleted using the Linux command:

`echo 3 > /proc/sys/vm/drop_caches`.

The Java implementation is done using Java 8 with the Oracle JDK version "1.8.0_101" and for C++ implementation I have used the C++11 standard, compiled using GCC (Ubuntu 5.4.0).

5.5 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. Given this difference, comparing C++ and Java programs can be challenging. For example, consider a single-threaded external sort of a large set of `customer` objects run using the Unix/Linux `time` command, we might see the following:

```
$ time java -XX:-UseGCOverheadLimit \
  -XX:+UseConcMarkSweepGC Test
real    29m44.986s
user    63m46.488s
sys     0m57.751s
```

Surprisingly, a single-threaded program runs in 29:44 but uses more than an hour of user CPU time. The explanation for this seeming contradiction is that the garbage collector was running in parallel with the user code, meaning that multiple CPU

cores were concurrently performing the sort. This implicit parallelism could render the effect of memory management on the overall running time negligible. Since a single-threaded C++ program necessarily have the total CPU time plus the I/O time equal the wall-clock running time, this means that comparing the wall-clock running time of two single-threaded Java and C++ programs is not a useful comparison.

One option would be to record the total CPU time plus the I/O time for Java and C++. However, it seems that a Java garbage collector typically uses 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.

5.6 Collecting Timings

For Java the `System.nanoTime()` was used to collect timings; for C++, `std::chrono::high_res` is used . For various experiments, the amount of time spent in specific I/O tasks are collected as a fraction of the overall time. This can be impossible to do exactly in Java. Consider the following code:

```
ByteBuffer bb = ByteBuffer...
long tmpTime = System.nanoTime();
fileChannel.position(seekPosition);
fileChannel.read(bb);
elapsedIOTime += System.nanoTime()-tmpTime;
```

While there is nothing happening between the two timings except for I/O operations, it is entirely possible that the JVM garbage collector performs a "stop the world" pause during some I/O operation, throwing off the timing. Thus, the Java I/O vs. CPU breakdown numbers should be considered approximate.

Chapter 6

Experiments

6.1 Experiment 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 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 a large number of objects are retrieved from secondary storage using a secondary index.

Before the experiment is run, the operating system buffer cache is emptied. We do not utilize a dedicated buffer cache for these experiments, 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.

6.1.1 Results

In Figures 6.1, 6.2 and 6.3, we show, for each of the ten implementations, the total running time required as a function of the number of `Part`, `Lineitem` and `Customer`

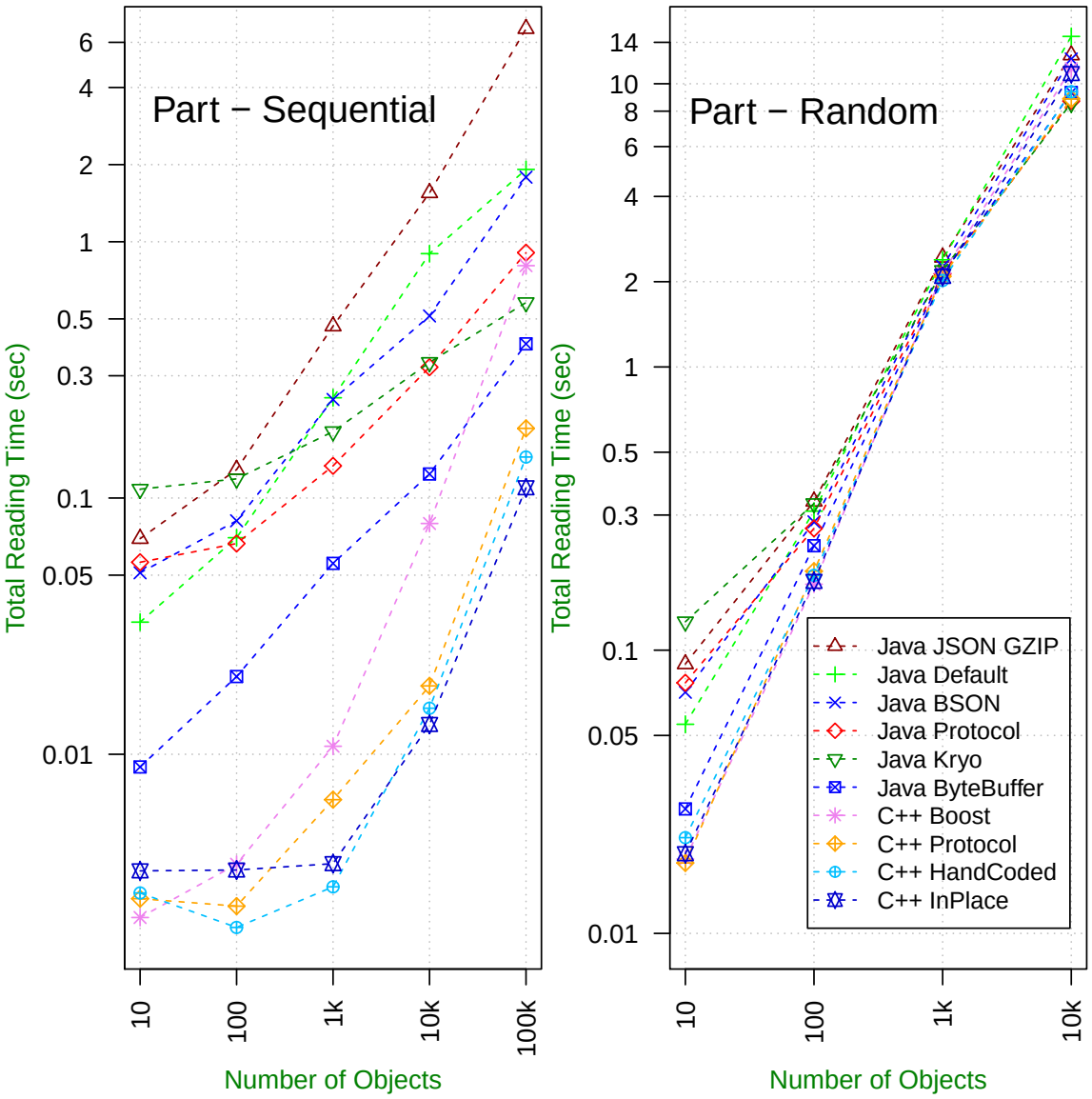


Figure 6.1: Reading times for `part` objects.

objects retrieved during the single-threaded experiments.

In Figures 6.4, 6.5 and 6.6, we show a few of the sequential read results as bar

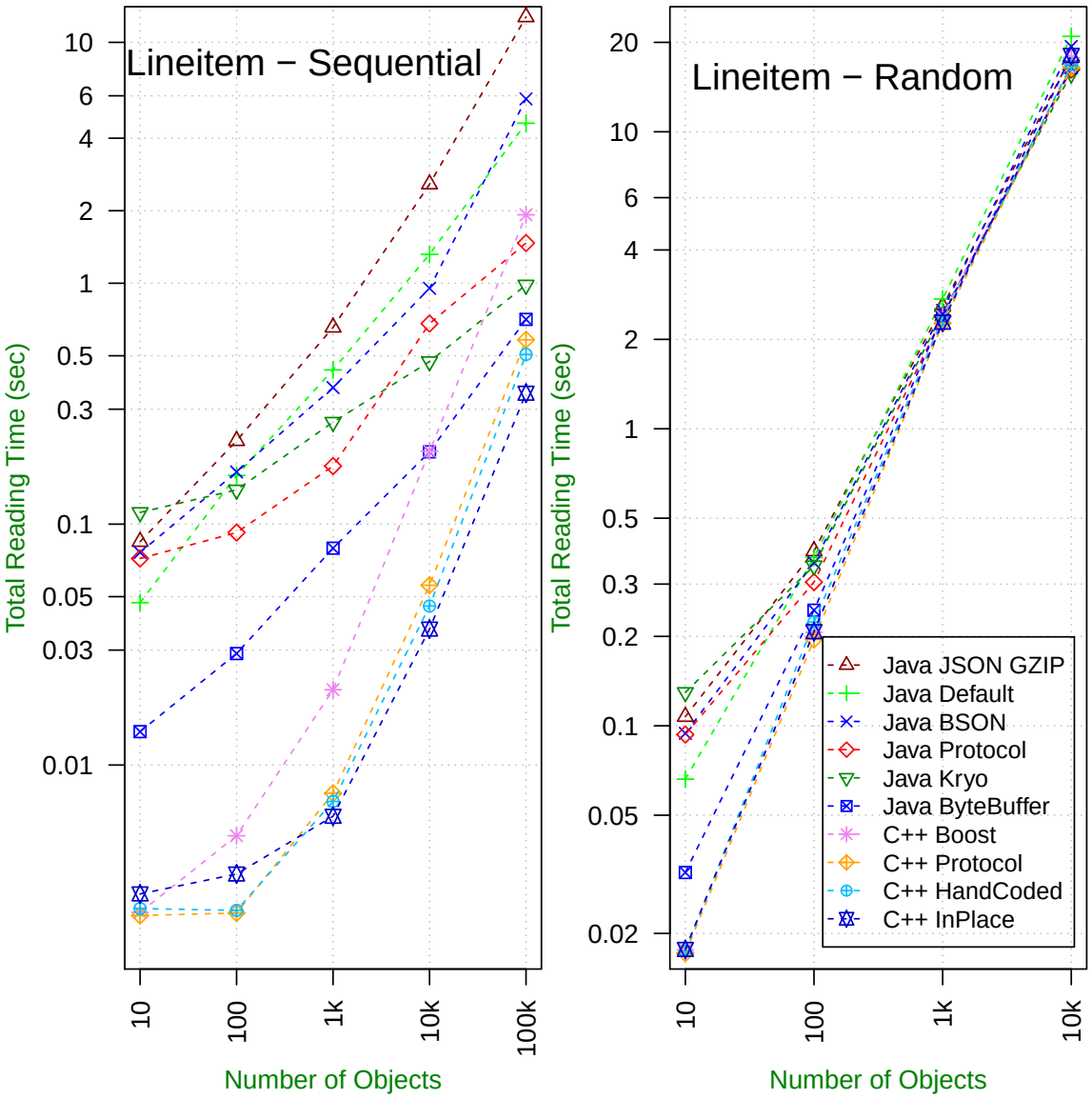


Figure 6.2: Reading times for `lineitem` objects.

charts, where the performance differences are a bit easier to see, and where we can also breakdown the total time into I/O and CPU.

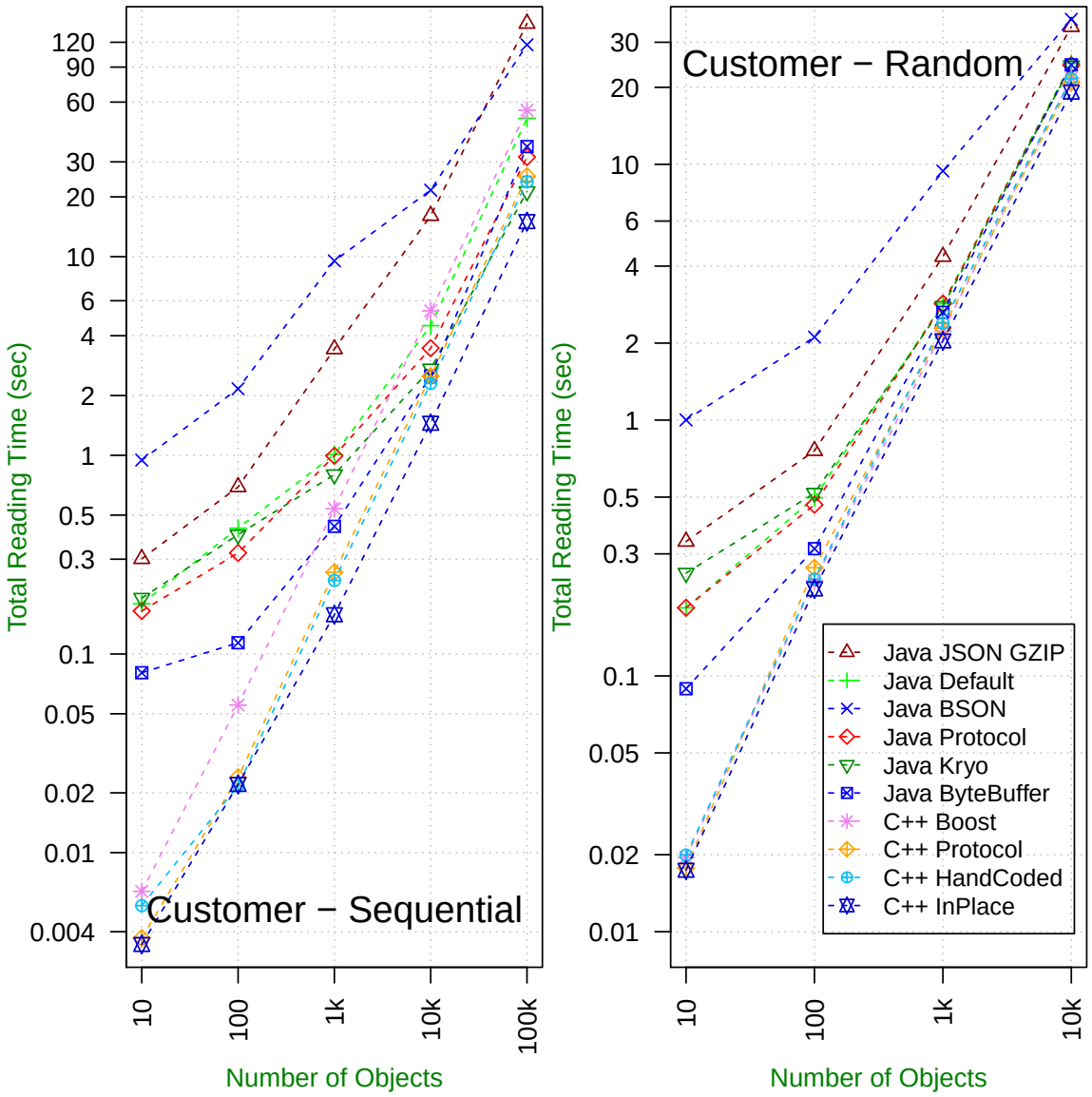


Figure 6.3: Reading times for `customer` objects.

Similarly, Figures 6.7, 6.8, 6.9 and show bar charts for some of the random read results.

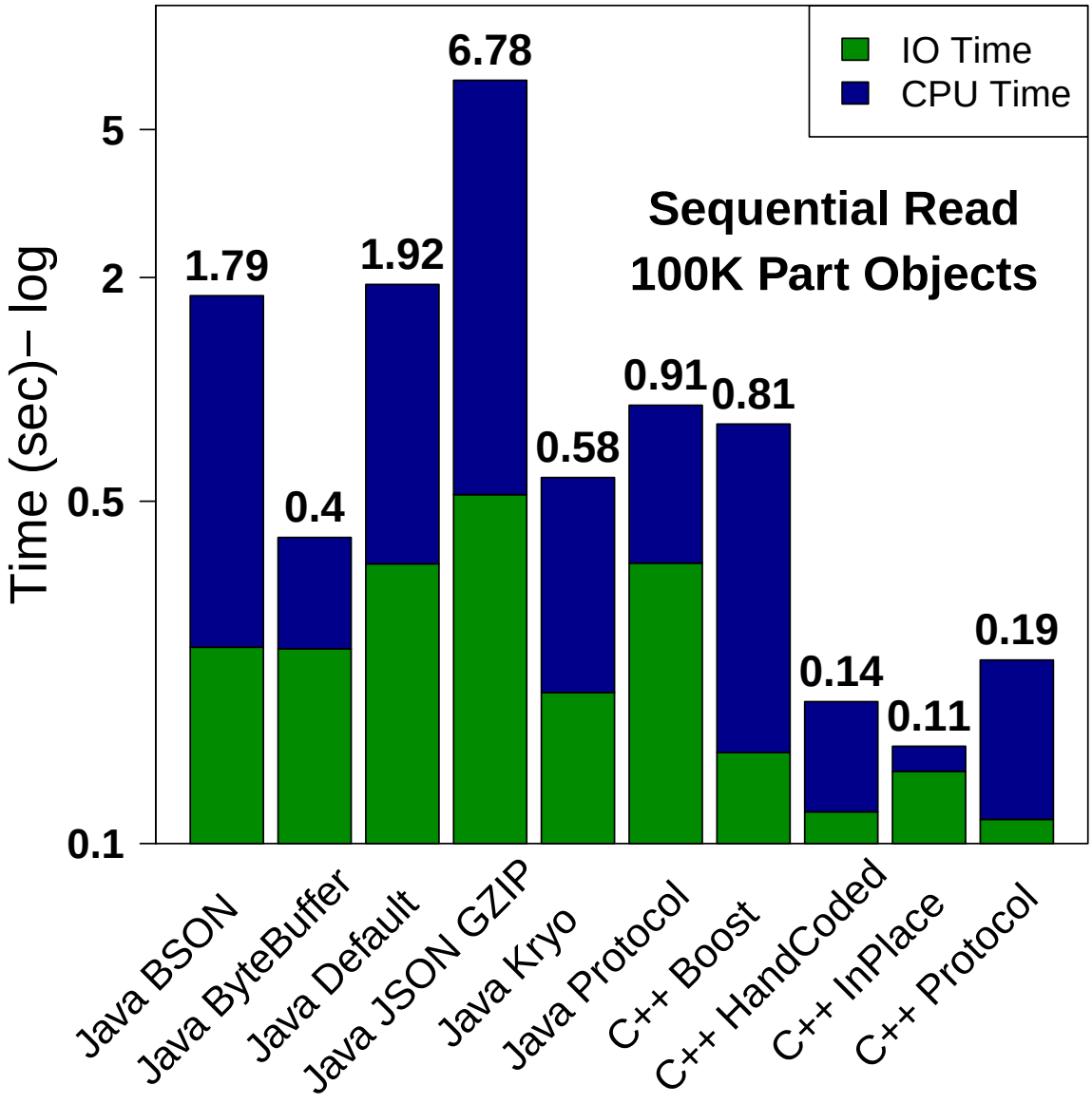


Figure 6.4: Total/IO time for sequential read of 100K `Part` objects

In Figures 6.10 and 6.11, we show, for reading `Part` and `Customer` objects during the single-threaded experiments, the time spent waiting for the CPU (which would

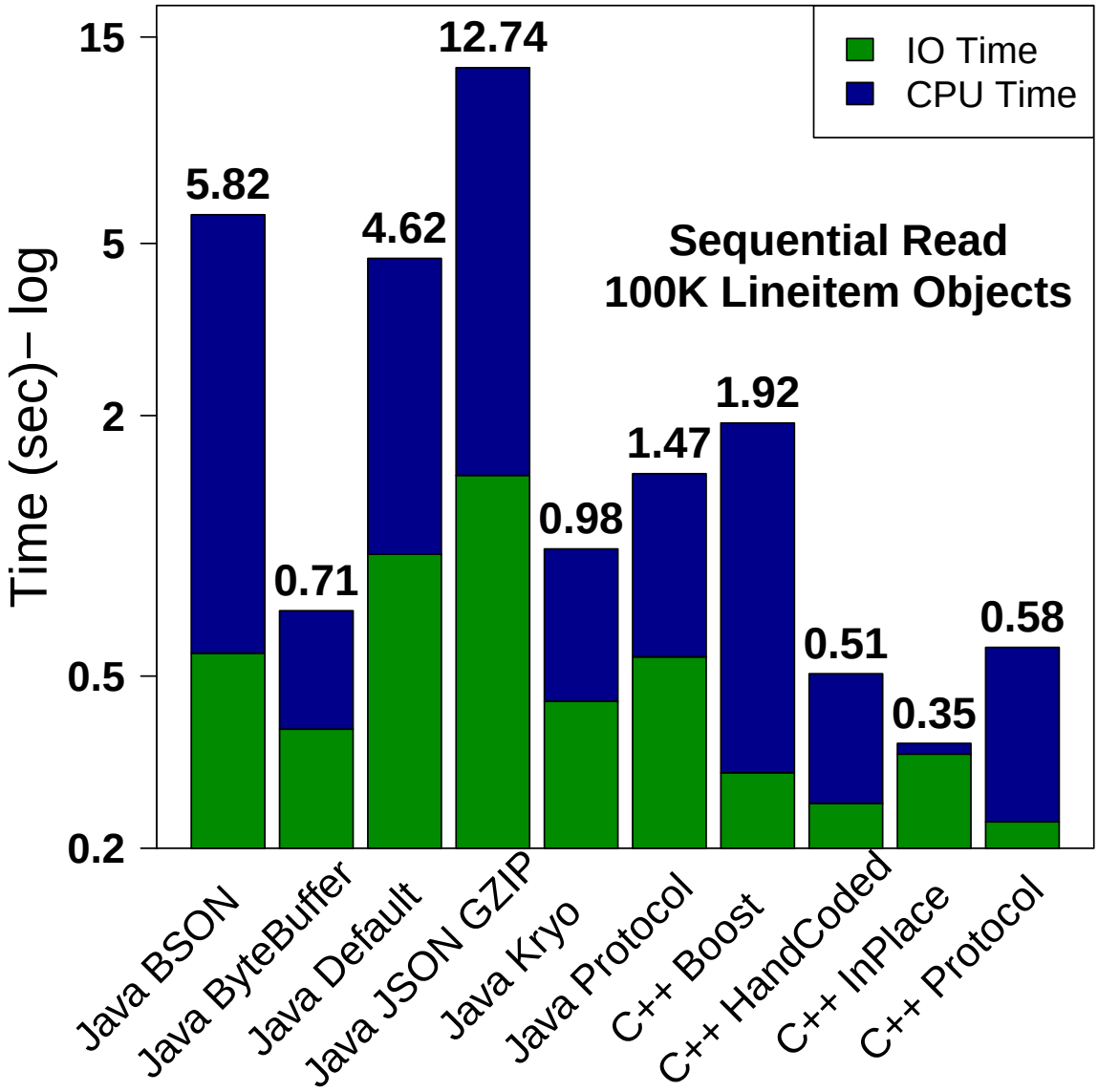


Figure 6.5: Total/IO time for sequential read of 100K `LineItem` objects

include deserialization and memory management) as a fraction of the total read time (which also includes I/O time).

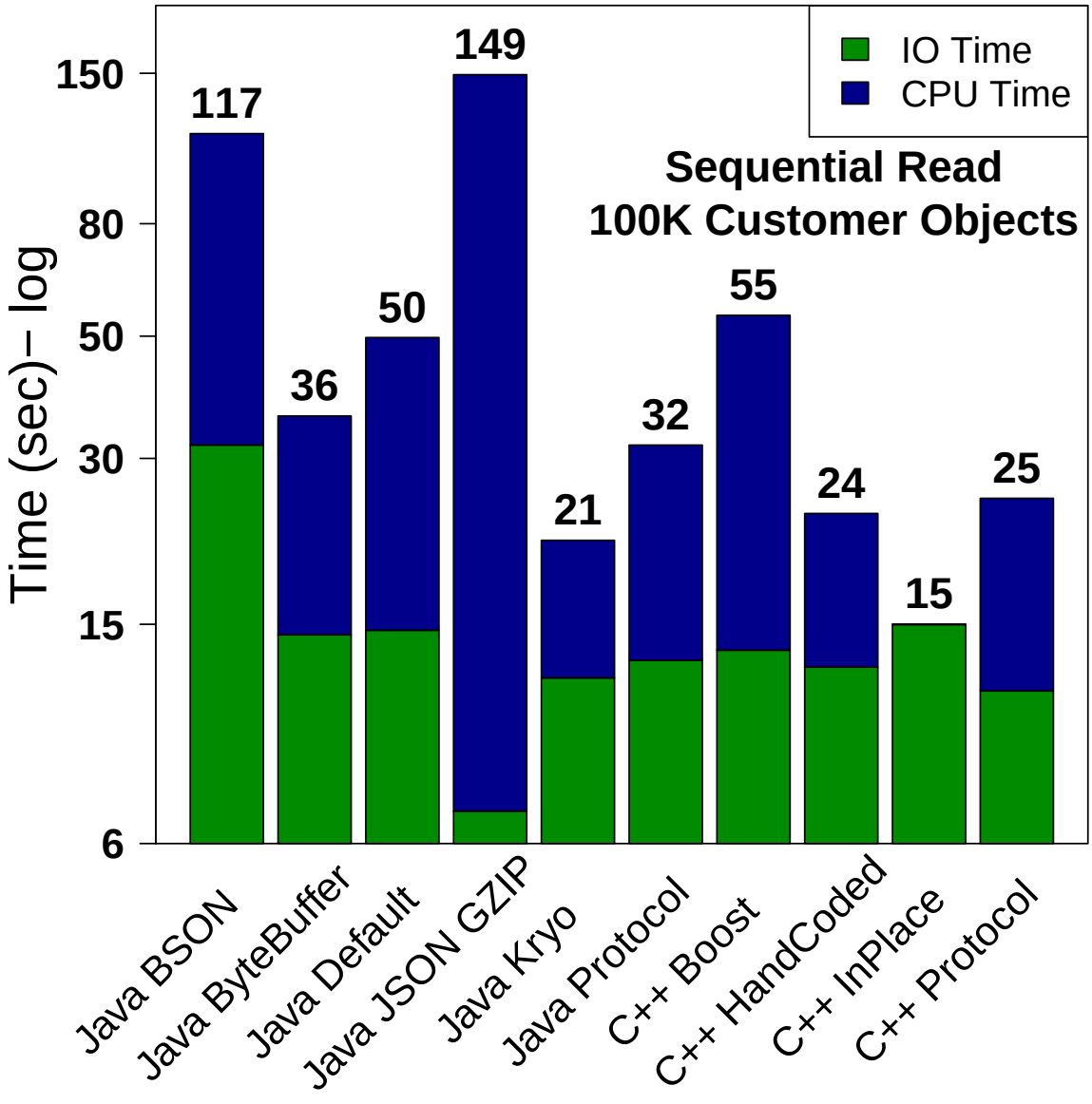


Figure 6.6: Total/IO time for sequential read of 100K `Customer` objects

In Figures 6.12, 6.13 and 6.14, we give a subset of the multi-threaded experimental results, showing the times required for sequential reads of various sizes for the three

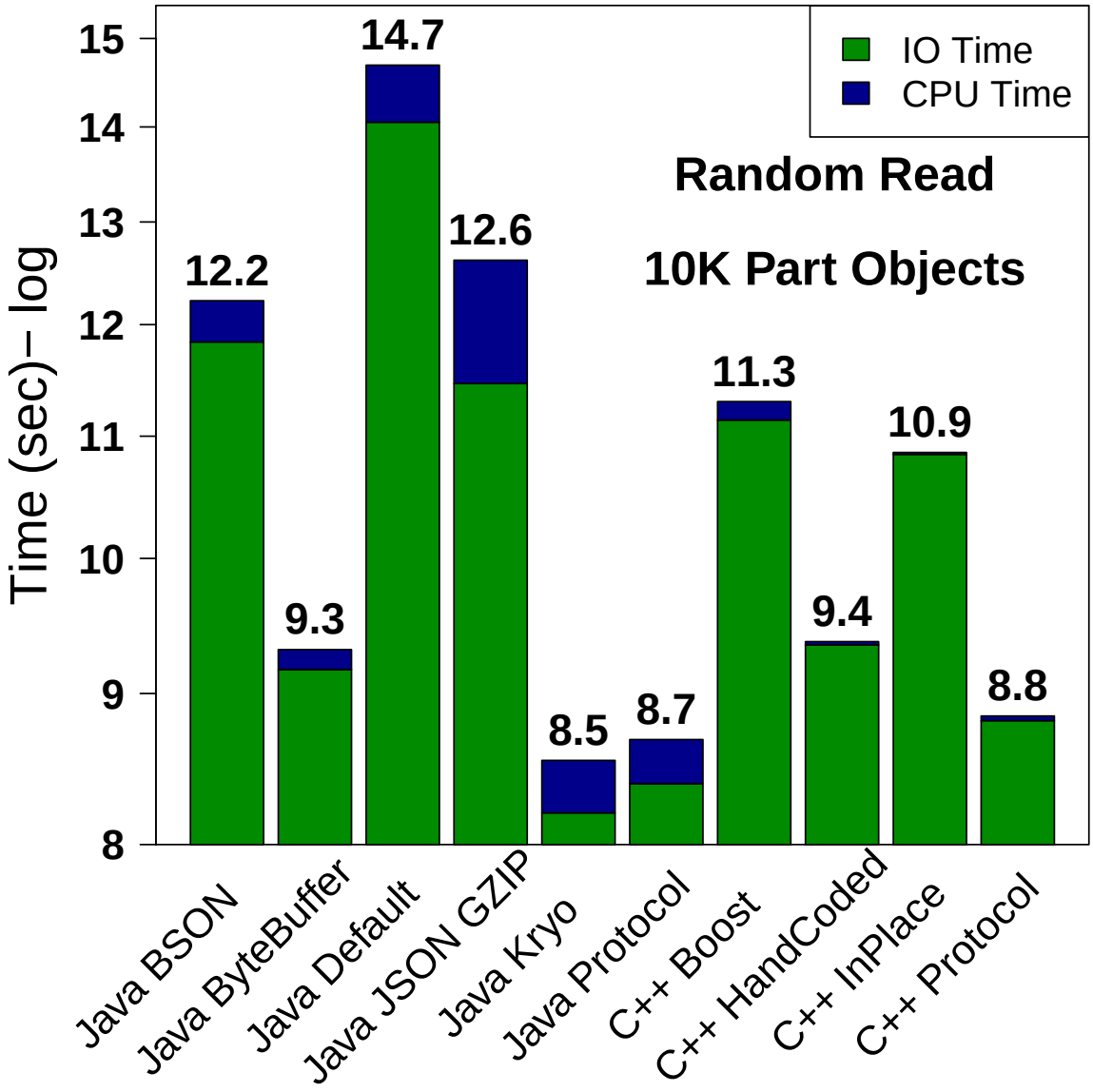


Figure 6.7: Total/IO time for random read of 10K `Part` object

different data sets.

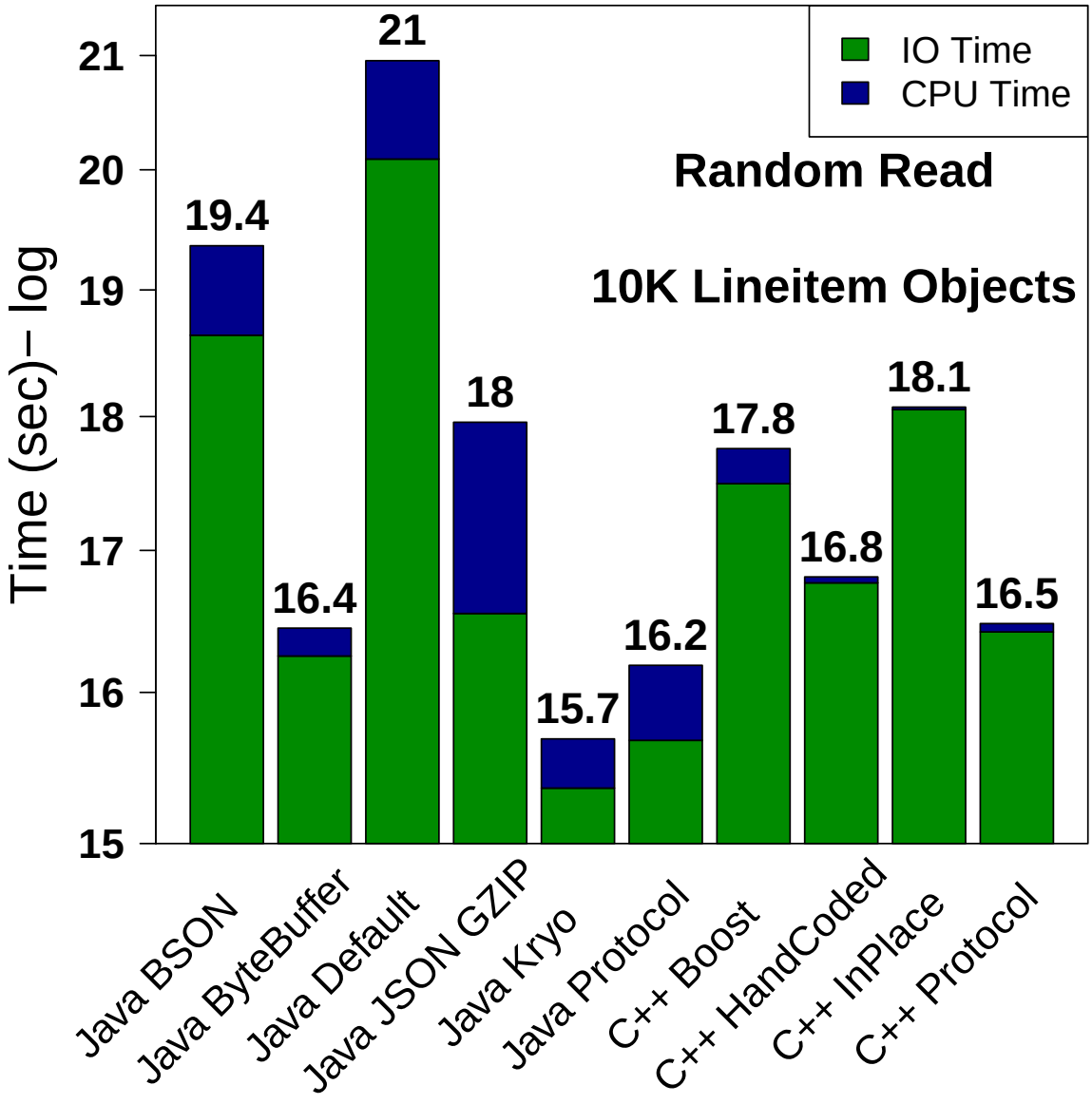


Figure 6.8: Total/IO time for random read of 10K `LineItem` object

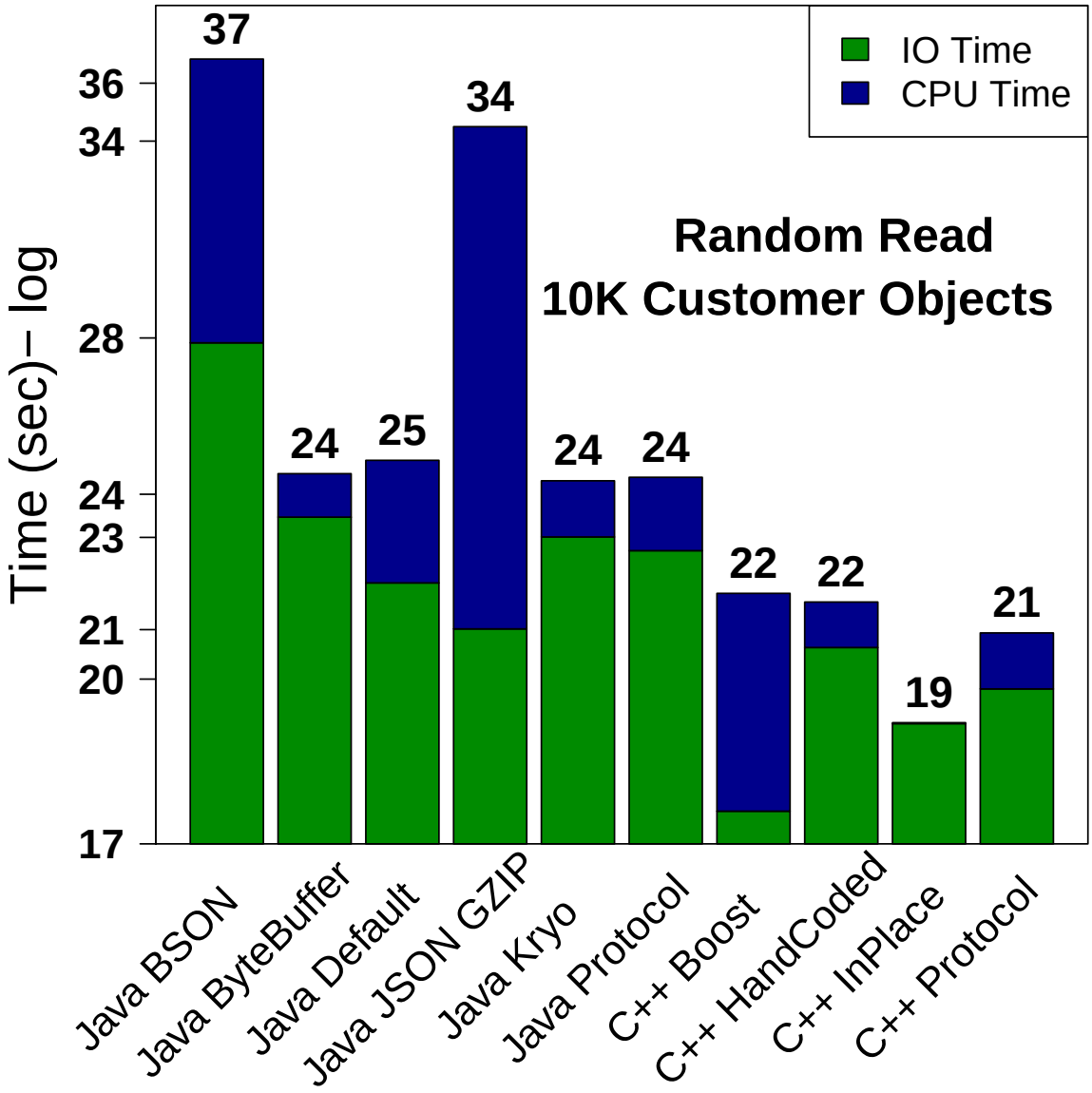


Figure 6.9: Total/IO time for random read of 10K `Customer` object

6.1.2 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

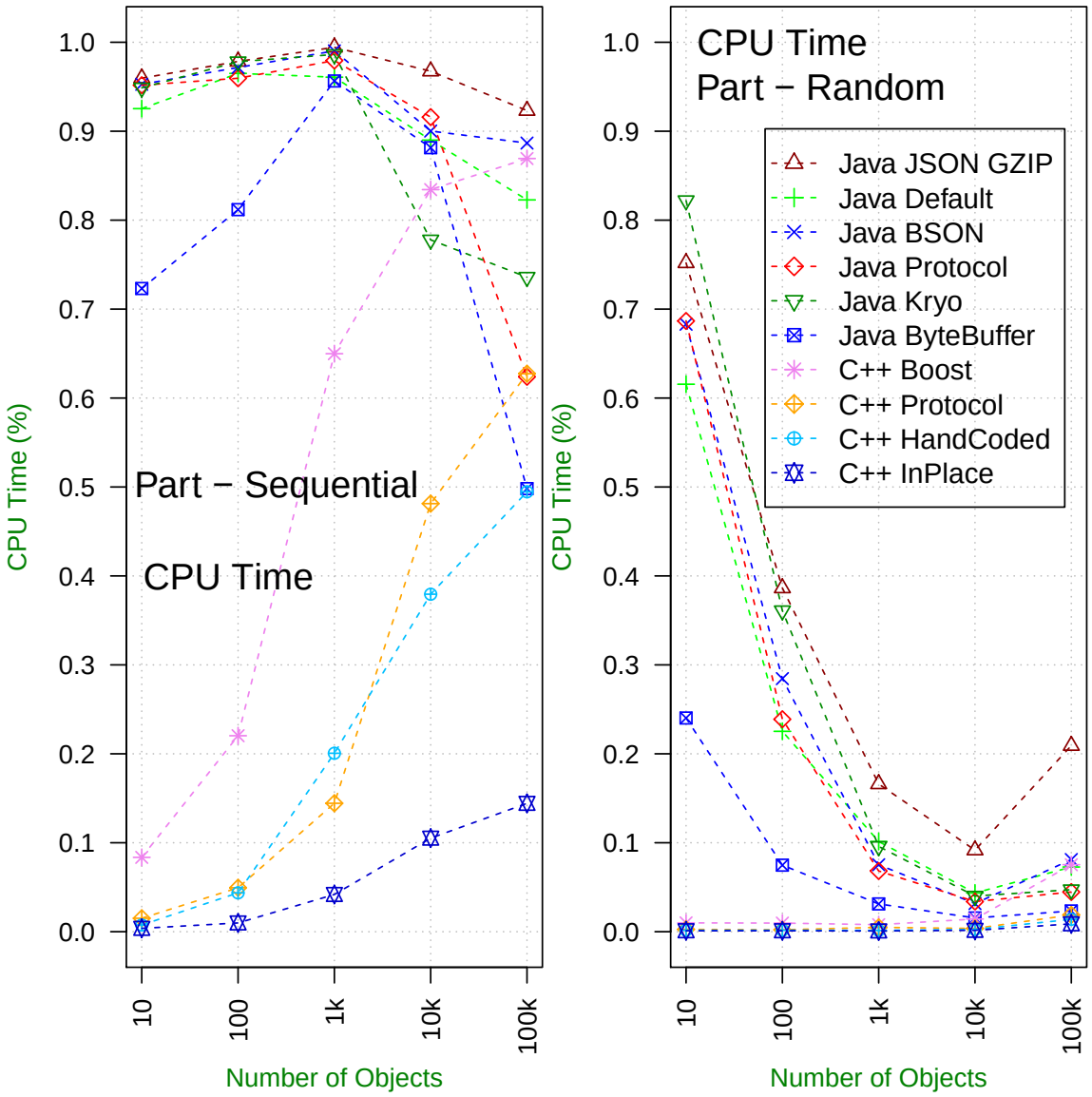


Figure 6.10: CPU vs. I/O time; `Part`

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

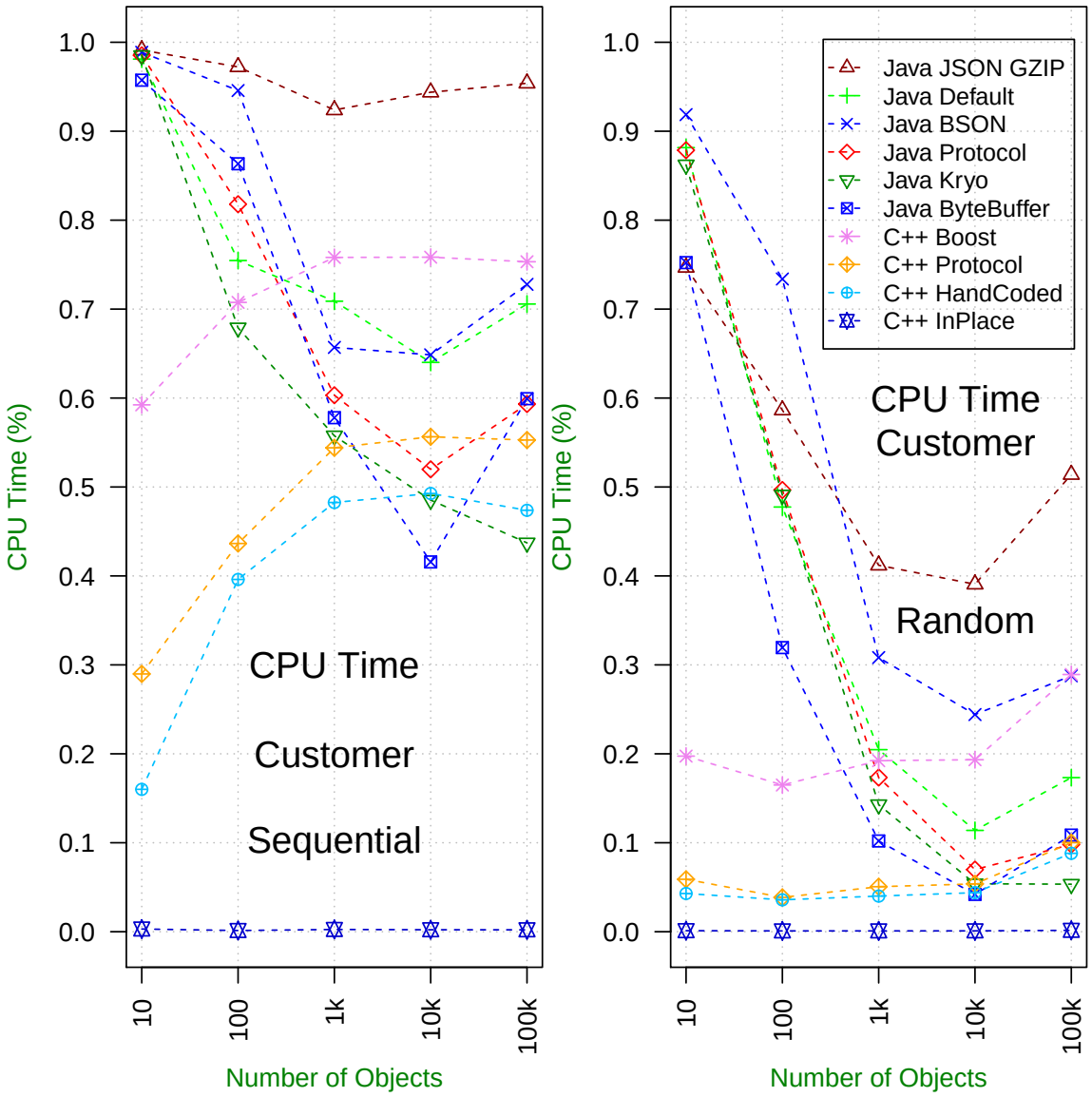


Figure 6.11: CPU vs. I/O time; `Customer`

sequential reads—ensure that I/O is relatively inexpensive, magnifying the cost of deserialization.

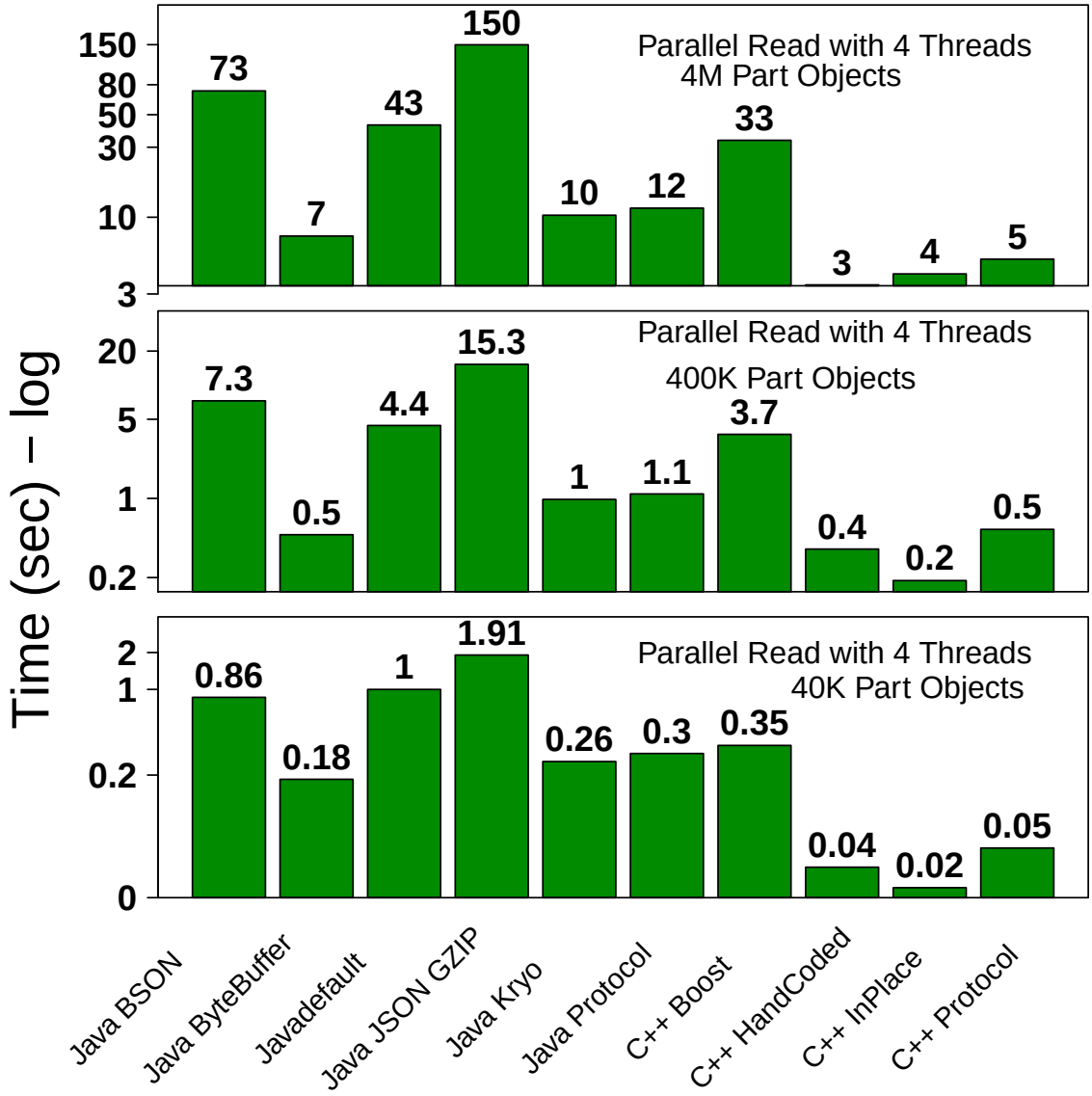


Figure 6.12: Sequential reads with four threads; `Part`

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 sin-

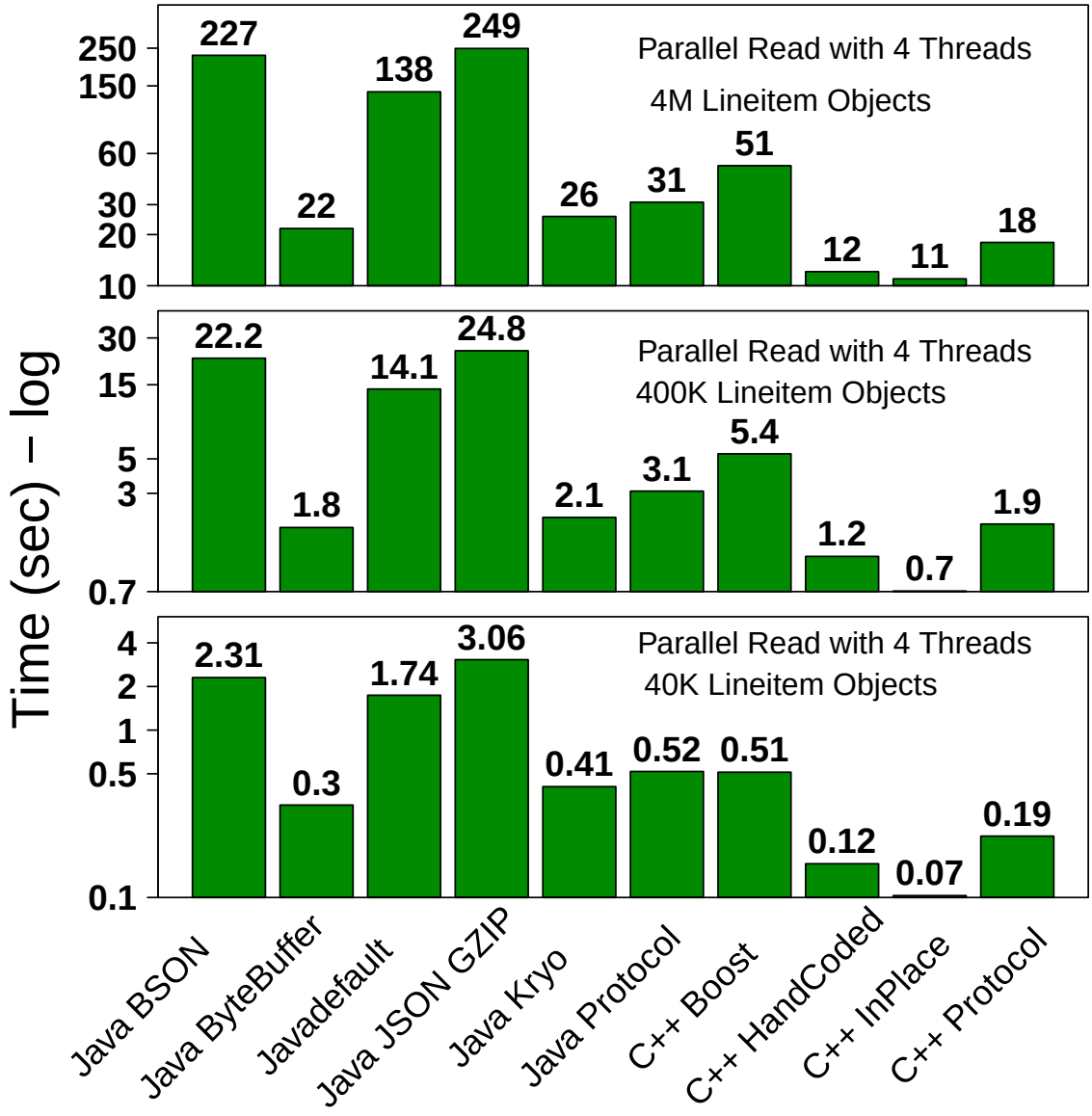


Figure 6.13: Sequential reads with four threads; `LineItem`

gle thread. For small to medium-sized sequential reads, there is consistently a 100× difference in total read time between the fastest deserialization methods—generally

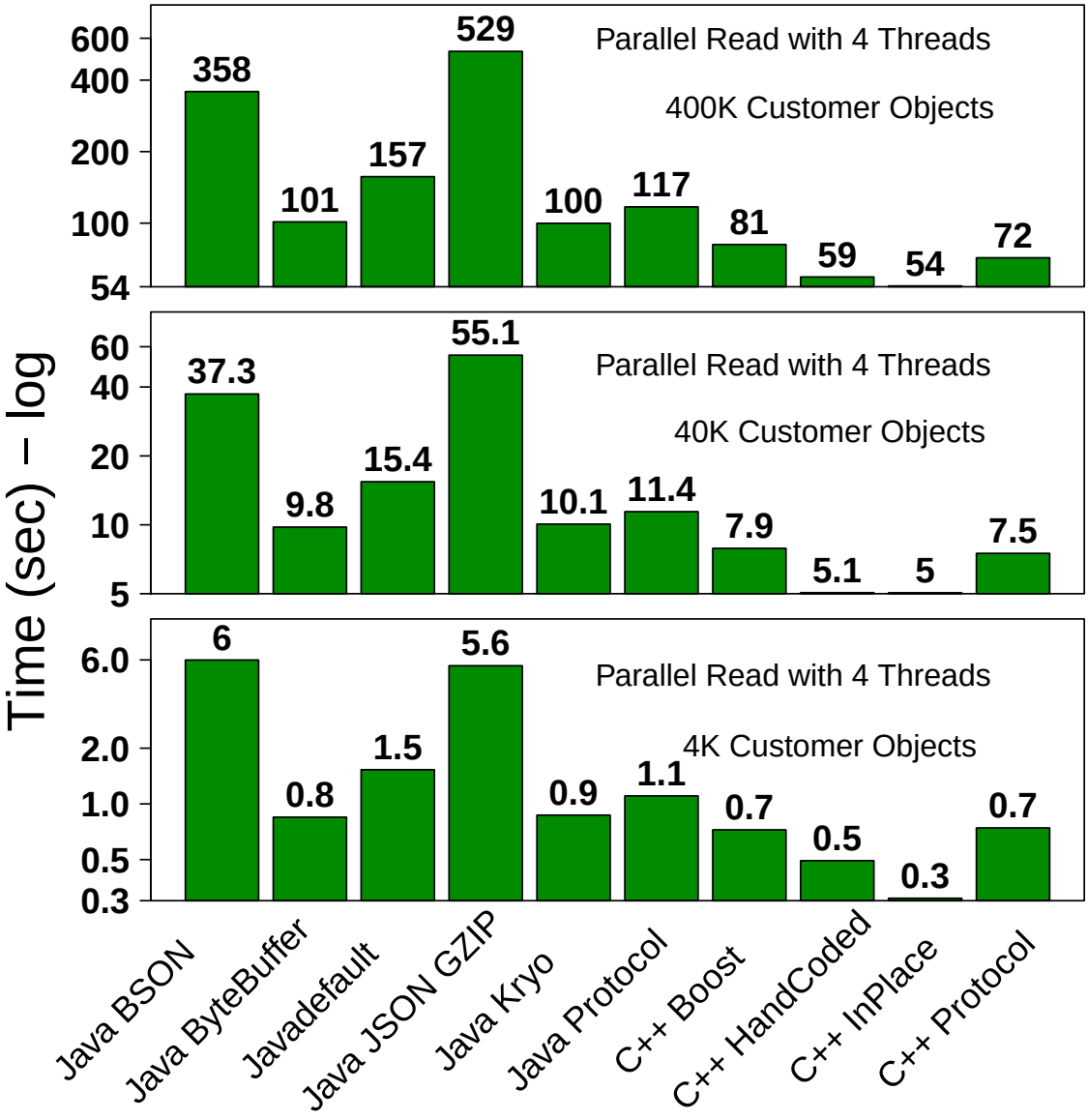


Figure 6.14: Sequential reads with four threads; `Customer`

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 indexing, query processing algorithms, and optimization/query planning, but a 100× performance hit incurred via deserialization 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` on more modestly-sized sequential reads.

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 6.1,6.2,6.3 than for larger, random reads (at the right of the plots, and in detail in Figures 6.7, 6.8, 6.9). 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 Figures 6.10, 6.11, which shows a dramatic decrease in the relative CPU cost for random reads, as the size of the read 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 experiment, the fastest Java option (a hand-coded deserialization from a `ByteBuffer`) was roughly 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.

6.2 Experiment 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 data distributed across the other ten machines).

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, and serialized object size is the most important factor.

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.

6.2.1 Results

Figures 6.18, 6.19 and 6.20 depict the time required to send data from disk, for each of the object implementations. Figures 6.15, 6.16 and 6.17 depicts the time required to send data from RAM.

6.2.2 Discussion

We begin with the most obvious observation: serialization size matters. In the disk-to-disk experiment (Figures 6.18, 6.19, 6.20) 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 each; this is around 300MB of total data), the client is able to process the objects at the speed of

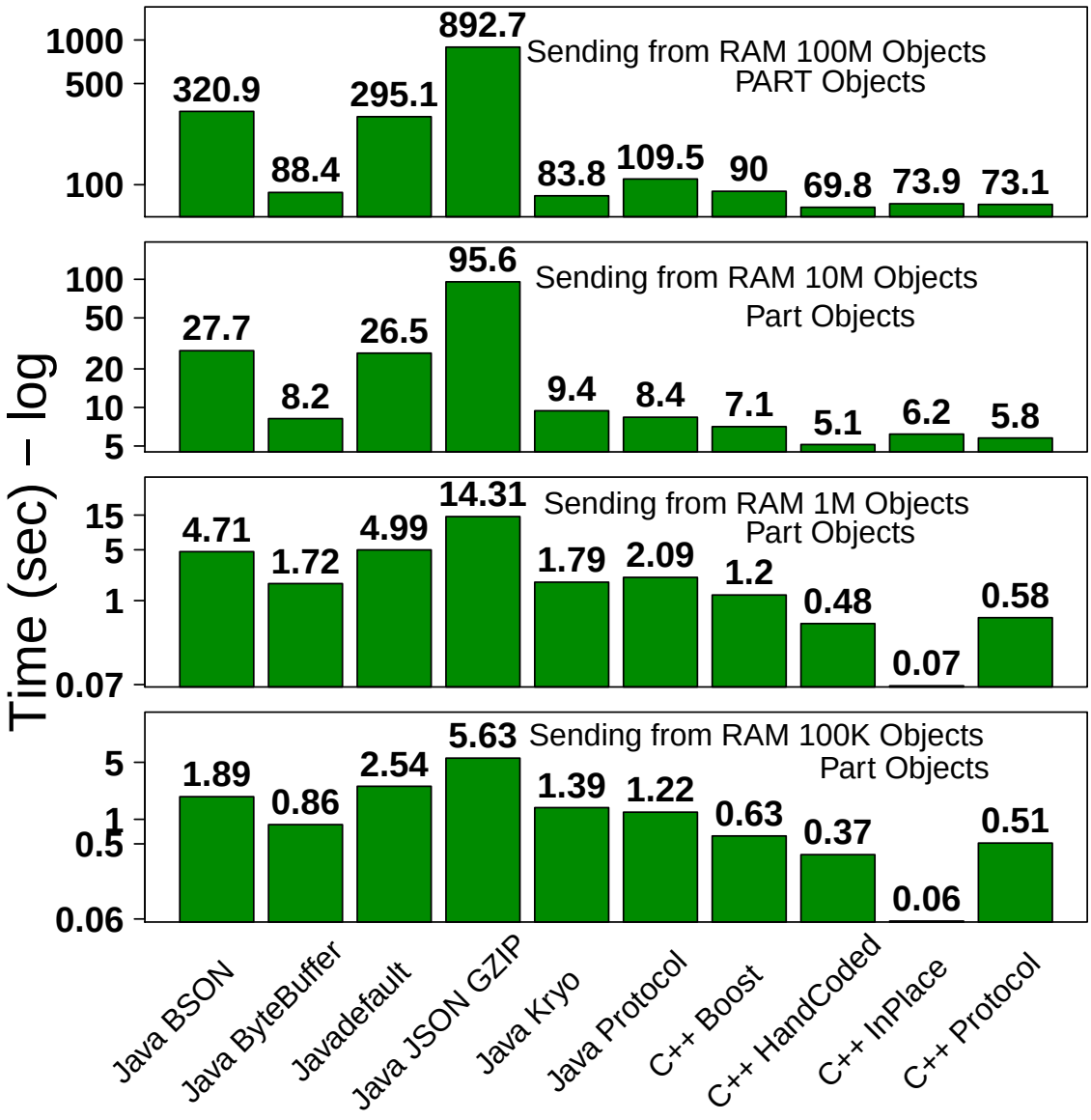


Figure 6.15: Network requests; data in RAM. `Part`

receipt using 10 threads. But when this increases to 1M objects (30GB of data) the becomes overwhelmed due to the large amount of disk and network I/O, and times

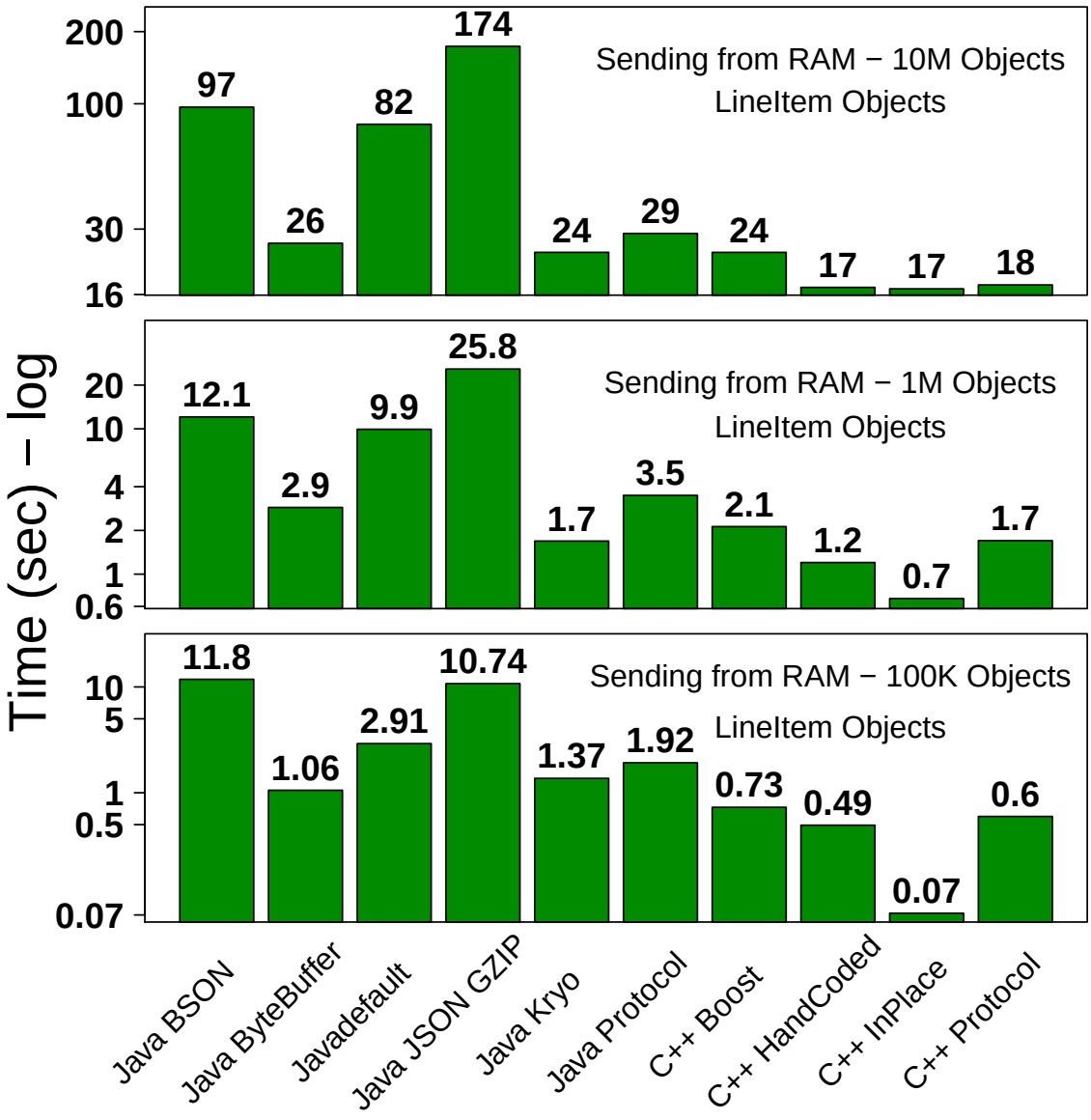


Figure 6.16: Network requests; data in RAM. `LineItem`

skyrocket.

However, as soon as one factors in the cost to serialize data, serialization size

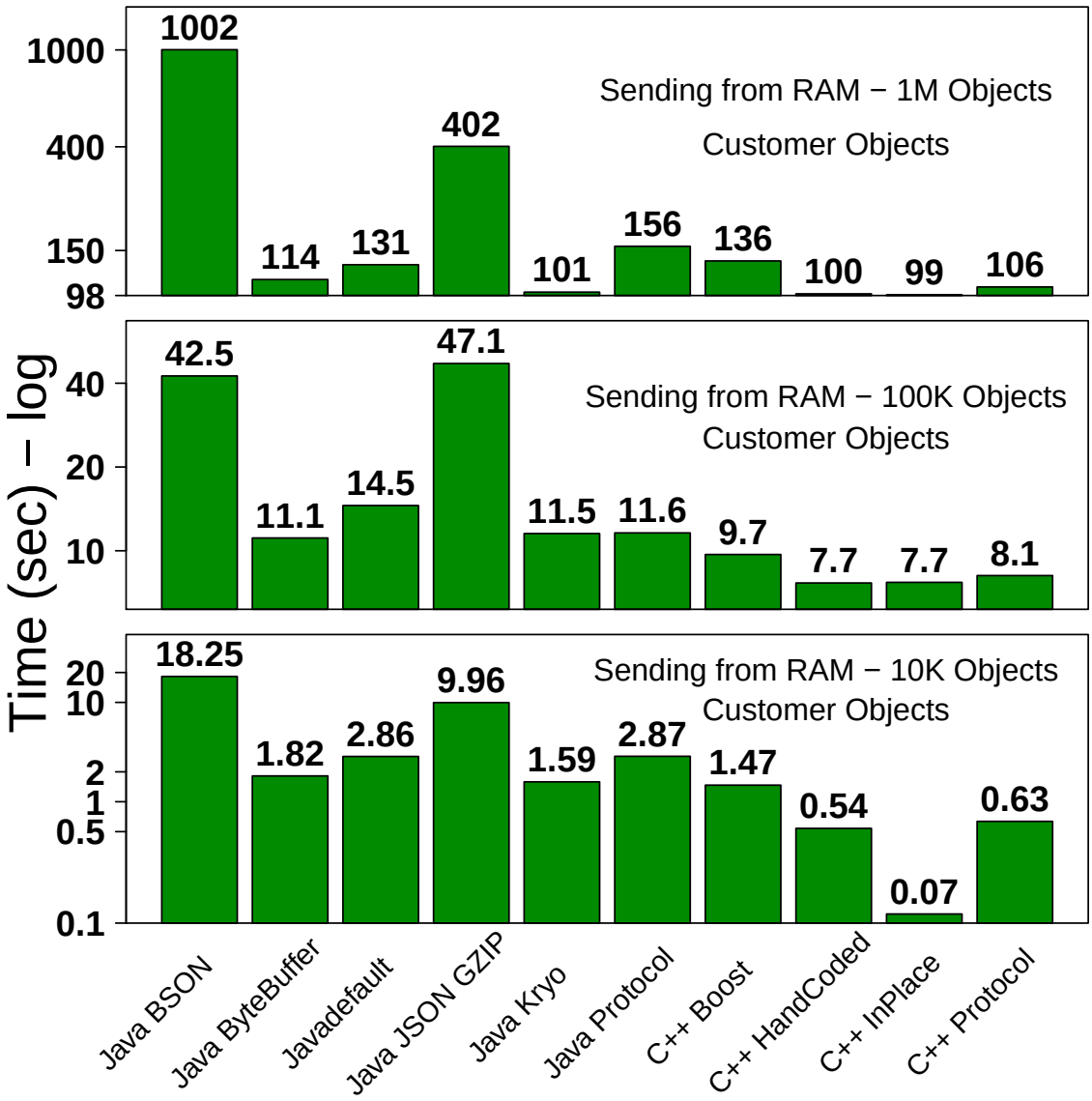


Figure 6.17: Network requests; data in RAM. `Customer`

matters a lot less. Consider Figures 6.15, 6.16, 6.17. C++ in-place serialization (which has no serialization cost) is consistently the fastest across all experiments (the

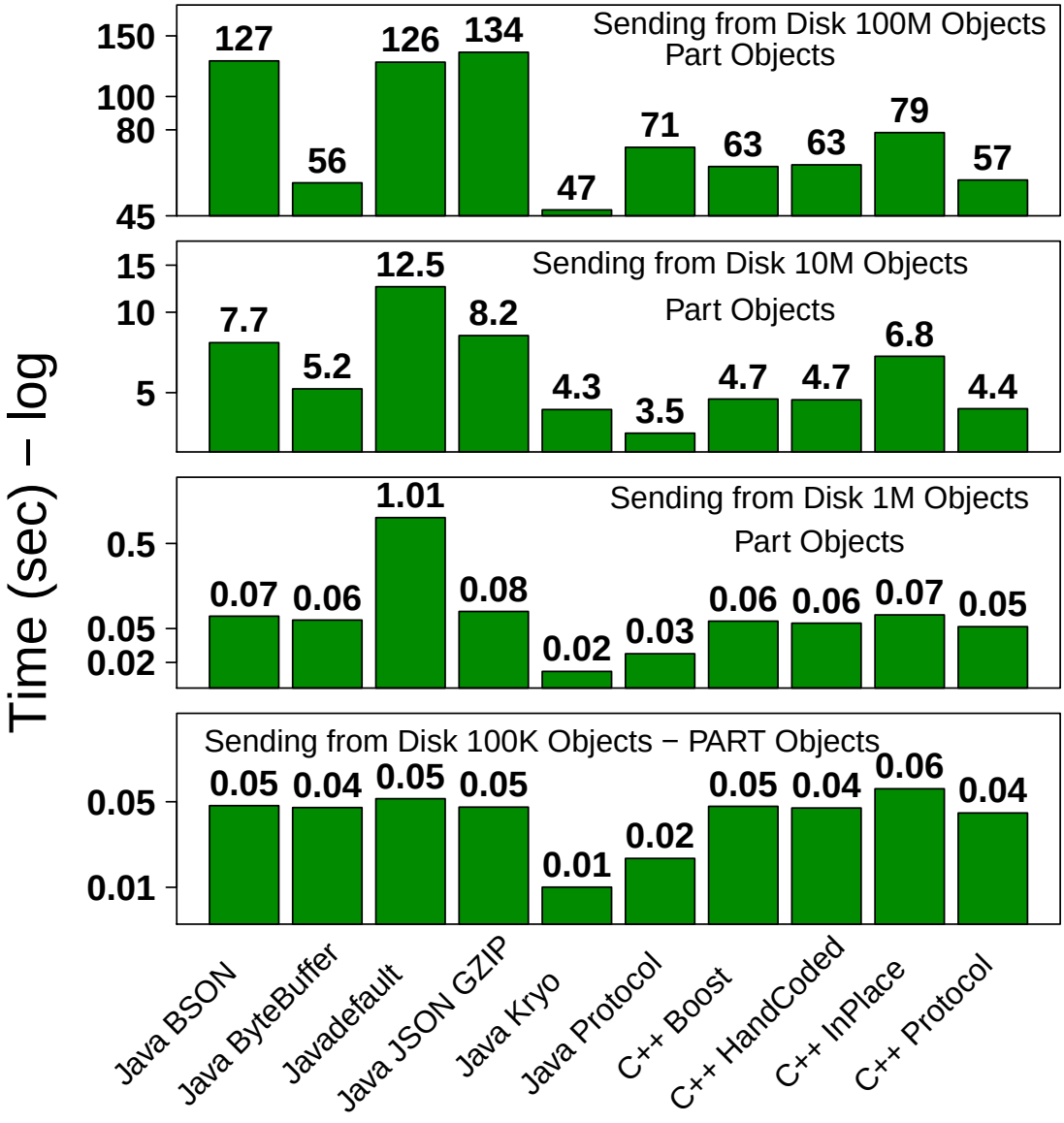


Figure 6.18: Network requests; data in Disk. `Part`

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.

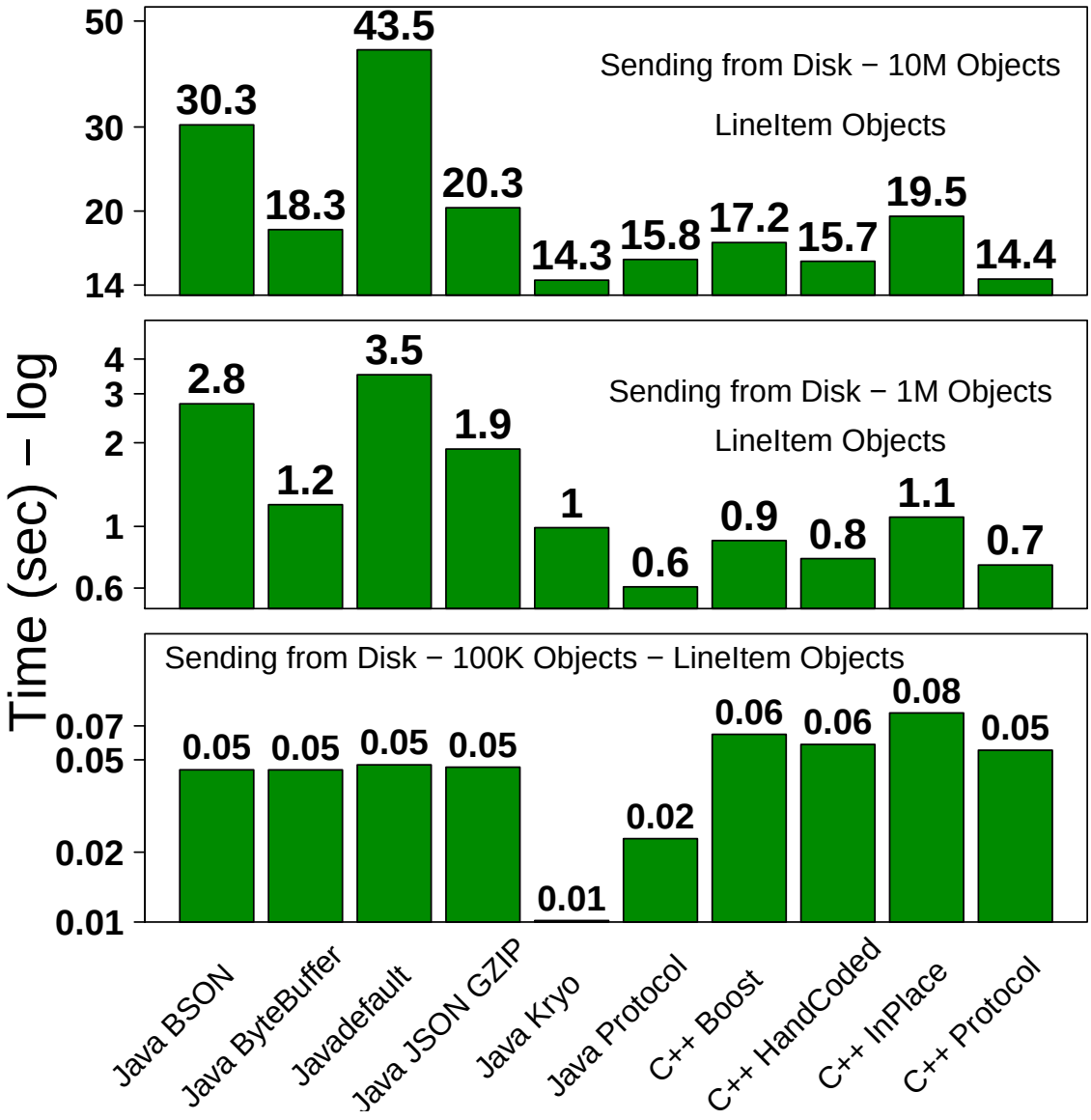


Figure 6.19: Network requests; data in Disk. `LineItem`

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-

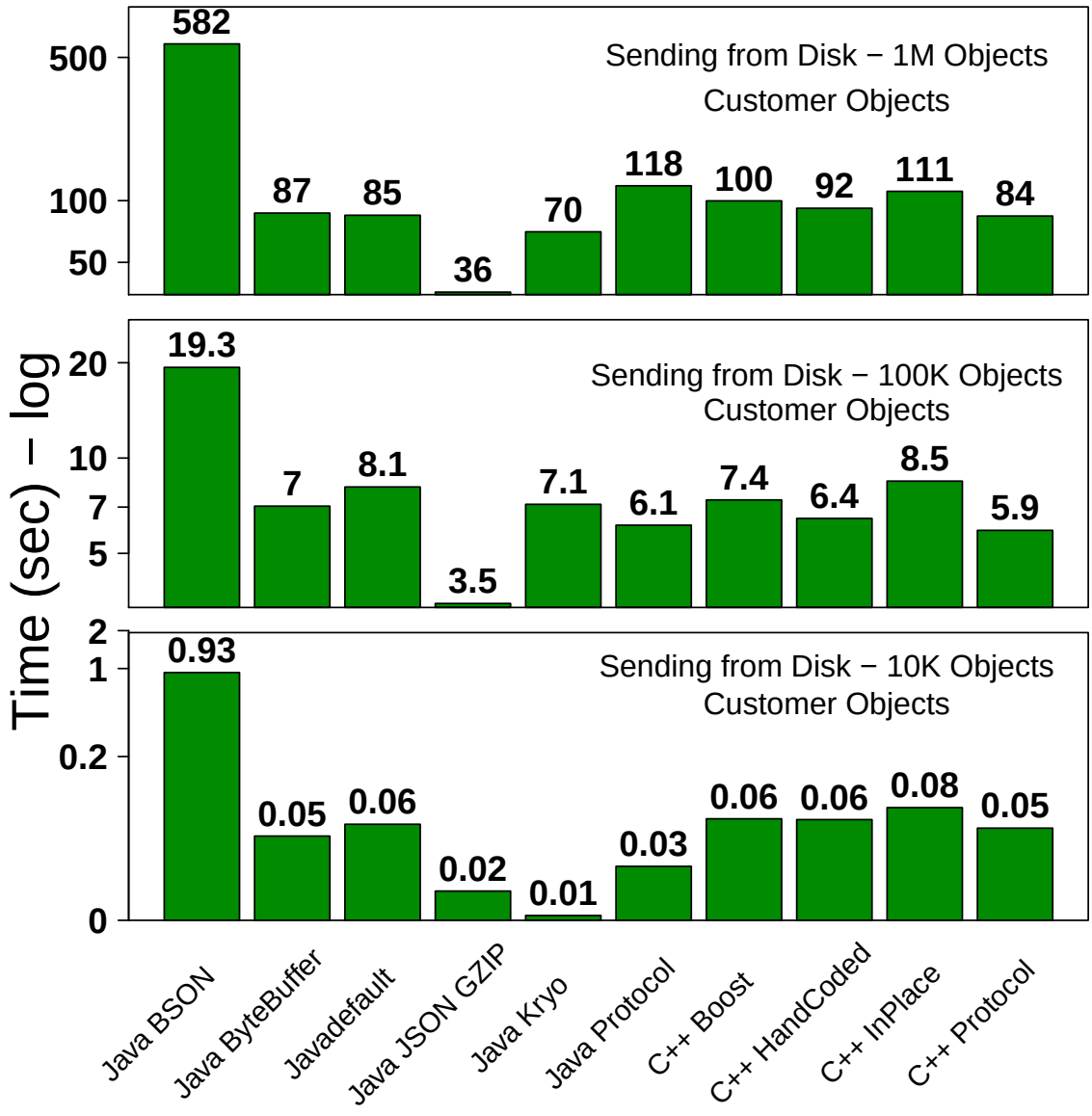


Figure 6.20: Network requests; data in Disk. `Customer`

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

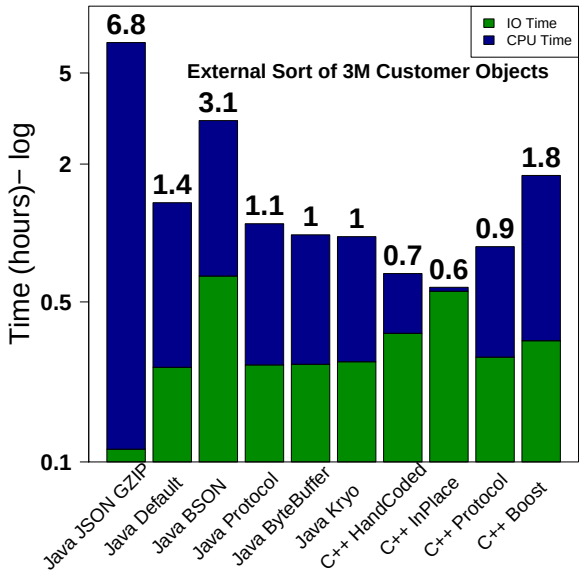


Figure 6.21: External sort times for `Customer` objects.

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. If it can be eliminated, radical speedups are observed.

6.3 Experiment 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 10 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 5.1.

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.

6.3.1 Results

Results are given in Figures 6.22, 6.23, 6.24. 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.

6.3.2 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× differ-

ence 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.

Finally, we note that the more complex the object, the closer together the various methods were in terms of performance. This seems a bit counter-intuitive, but the reason for this is that for more complex objects, the ratio of the number of objects sorted to the total I/O decreases. Hence, per-object costs (such as the number of object comparisons) tend to decrease.

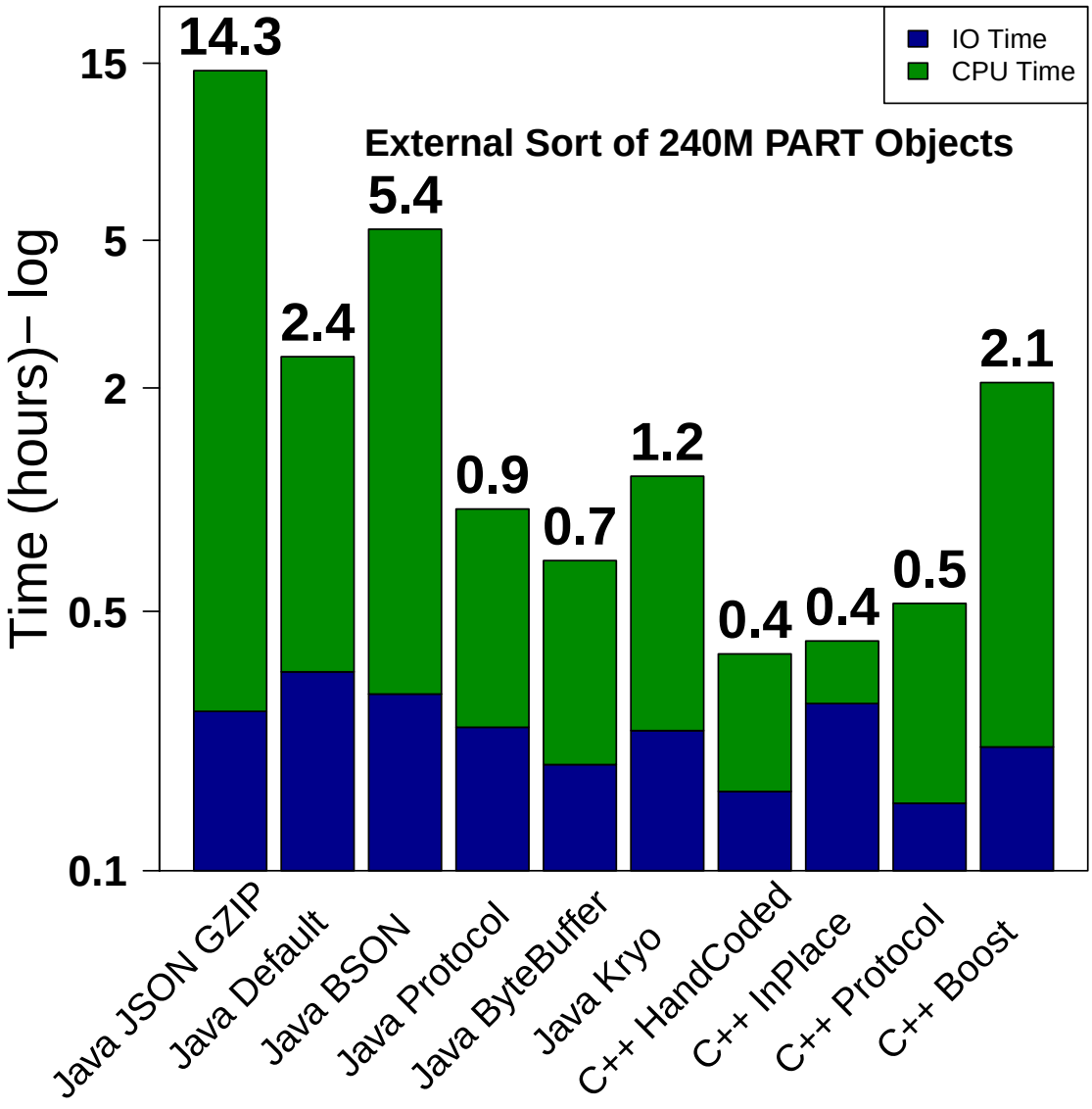


Figure 6.22: External sort times; `Part`

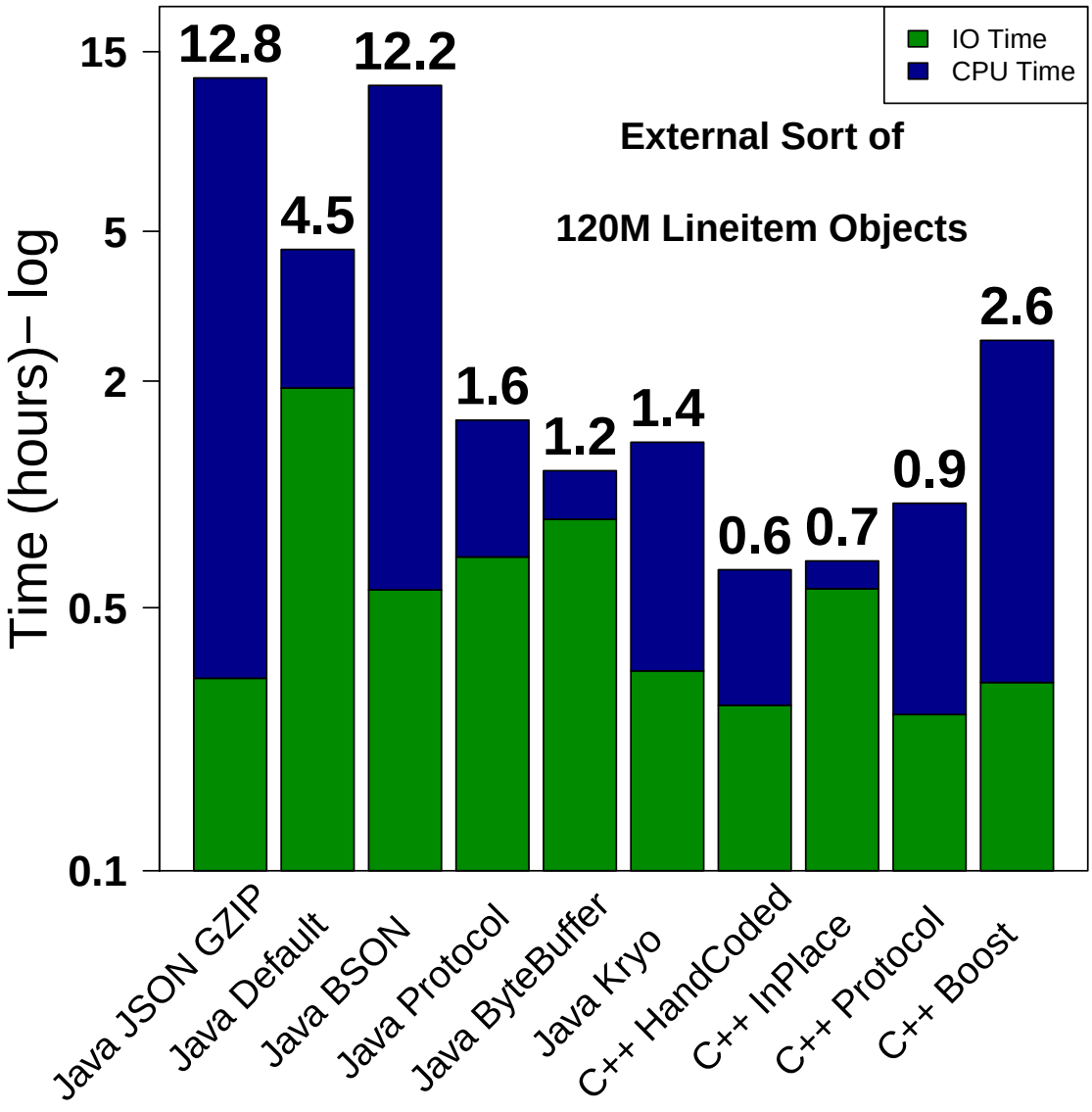


Figure 6.23: External sort times; `LineItem`

6.4 Experiment 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.

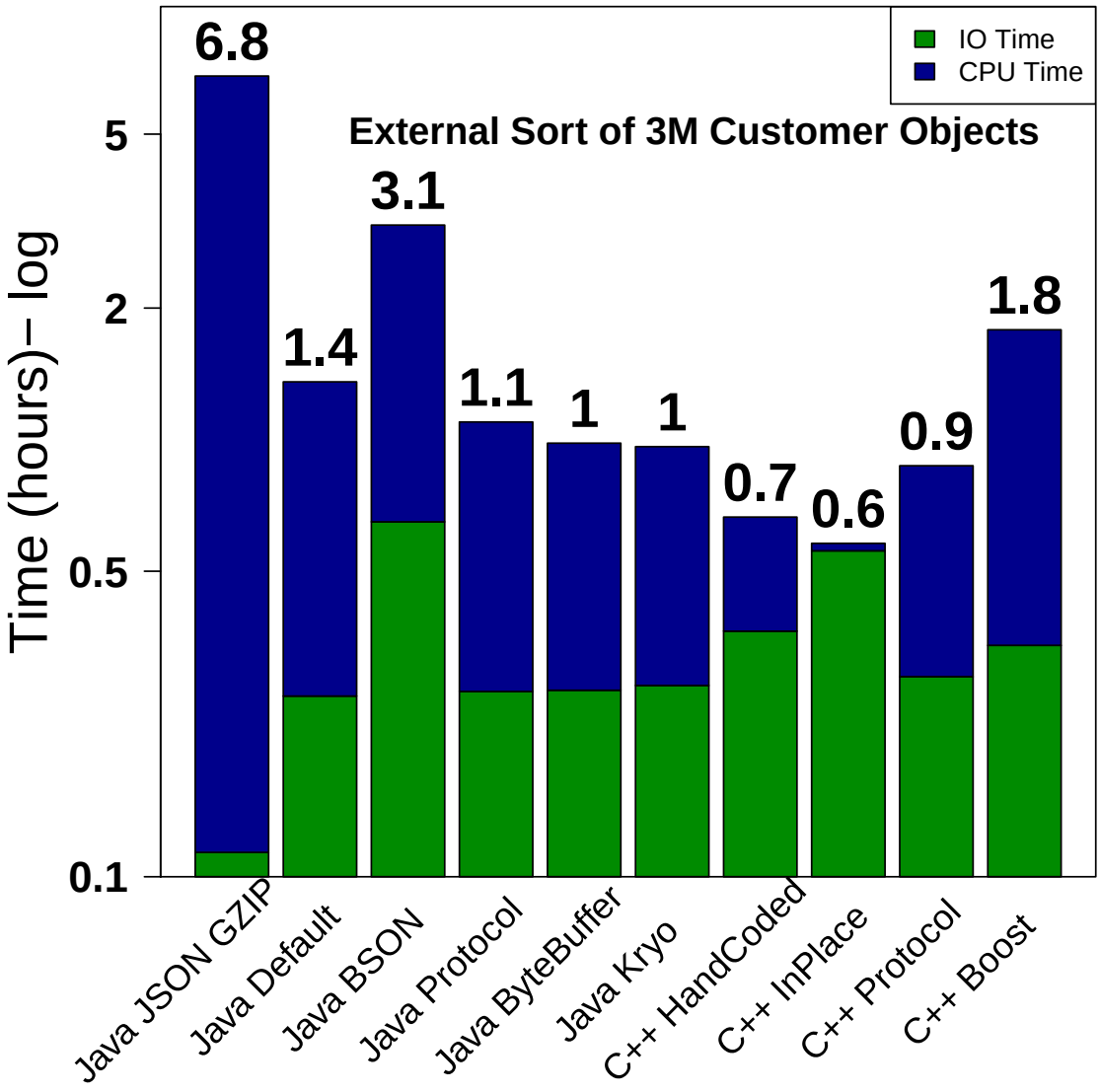


Figure 6.24: External sort times; `Customer`

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.

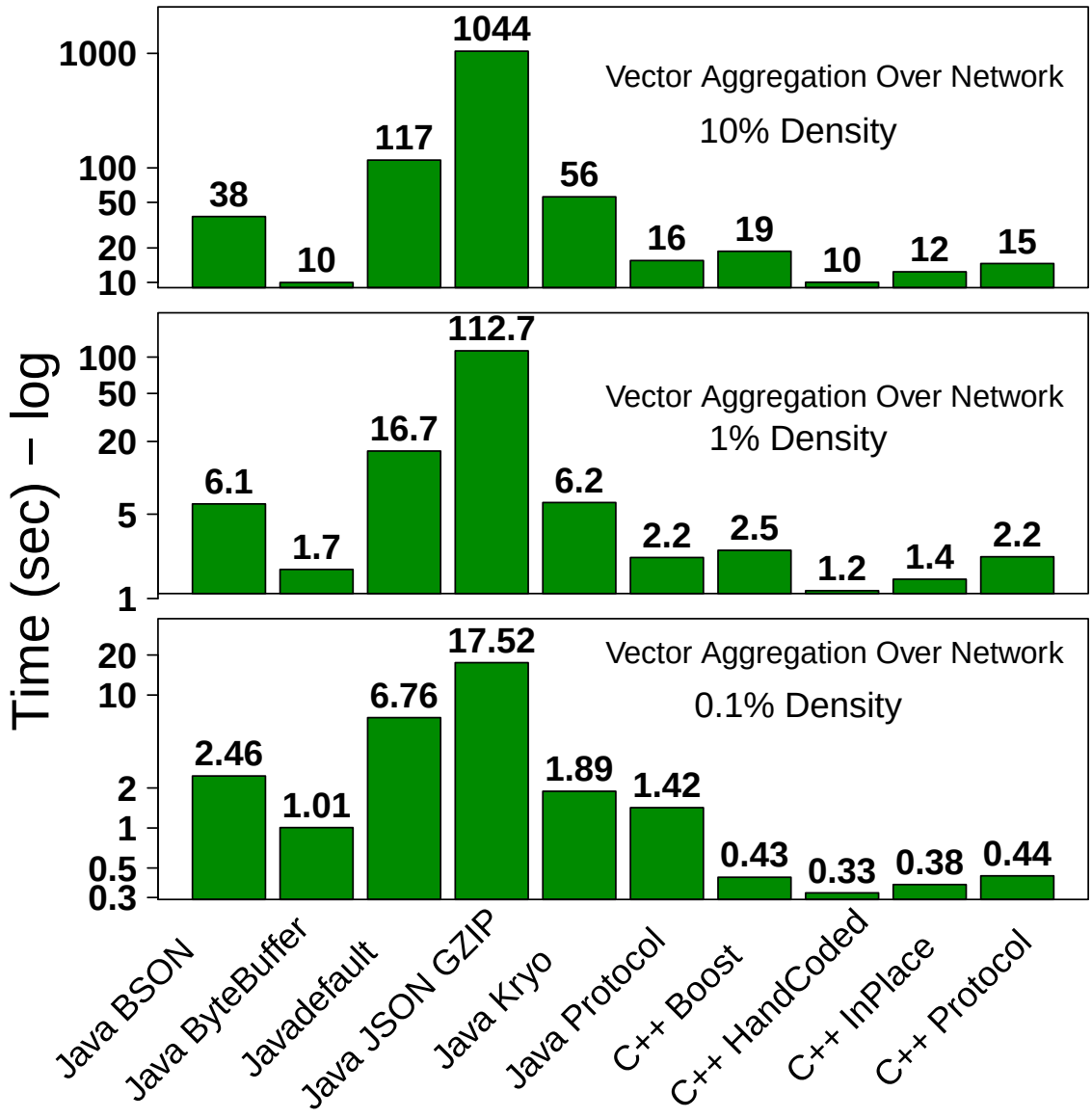


Figure 6.25: Sparse Vector Aggregation over Network (`7 Nodes`).

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%.

6.4.1 Results and Discussion

Figure 6.25 depicts the results of our sparse vector aggregation experiment.

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.

Chapter 7

Conclusions

In this thesis, I have presented a number of experiments that examined the performance of various implementations of complex objects with a focus on implementations suitable for use in a Big Data analytics or management system.

The results show that no matter how objects are implemented, there are significant costs associated with manipulating objects on a managed infrastructure (Java) as opposed to an un-managed 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 (again, in-place), despite the in-place solution's high I/O requirements. We expect that in a multi-threaded environment, where multiple sorts are taking place concurrently (during the implementation of a join operation, for example) this disparity will grow even greater.

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 note that the costs are even higher when an interchange format such as JSON is used in conjunction with Java. Sorting JSON objects results in a 10× to 20× performance hit compared to C++, and yet JSON is still a popular storage format in practice.

The costs may be worth it. Most would agree that Java is a much easier language to implement a Big Data system in, and Java codes are portable. Further, self-documenting interchange formats such as JSON have obvious benefits. But there are very significant costs associated with choosing to use these technologies that should not be taken lightly. A 2× (or greater) performance penalty before one begins to design a system is significant.

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, even when performing tasks that are popularly considered to be I/O bound. 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 [34], 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 thesis 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.

Appendix

In this Appendix, the various implementations tested in this thesis are described in detail.

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. Strings such as `Part.comment` are implemented using the Java `String` class. Types are built up recursively, so that `Customer` contains an `ArrayList` of `Order` objects, for example. Objects are de/serialized from/to a Java `ByteArrayOutputStream` which then converted to a `ByteArray` in the case of serialization, and from a `ByteArray` for deserialization.

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);`) one time at startup. This registration enables fast subsequent de/serialization. Each of our Java classes have a Base class that has the following code to self-serialize:

```
public byte[] kryoSerialization() {
  ByteArrayOutputStream bos =
    new ByteArrayOutputStream();
  Output output = new Output(bos);
  KryoSinglton.getInstance().getKryo()
    .writeObject(output, this);
  output.flush();
  byte[] buf = bos.toByteArray();
  return buf;
}
```

Deserialization is implemented similarly.

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). For example,

```
message PartP {
  required int32 partID = 1;
  required string name = 2;
  ...
}
```

In practice, the resulting PB classes are typically used directly, without conversion to/from a "regular" Java class. To ensure that we could use the same Java code for

our experiments, we instead converted the PB classes into our own Java classes after deserialization (as we will describe subsequently, the cost of doing this is negligible). Each of our Java Classes has `toProtocolBuffer()` method that converts a Java class to a byte array. First we need to instantiate a PB builder that can be used for serialization:

`PartP.Builder m_part = PartP.newBuilder();` Then we need to access our object attributes and set them to the PB own object; for example:

```
m_part.setPartID(this.partID);
m_part.setName(this.name);
m_part.setBrand(this.brand);
...
```

And then we use again a ByteArrayOutputStream and write the serialized data into it and convert it to a byte array:

```
ByteArrayOutputStream bos =
  new ByteArrayOutputStream();
m_part.build().writeTo(bos);
```

To transform a PB object to our own Java class during deserialization, we need to create a new empty object, then build a PB object and deserialize data to PB object, and finally call the getter and setter methods to set the content of the our empty object and return it. For example,

```
Part protocolBufferRead(byte[] buf) {
  Part m_part = new Part();
```

```
    ByteArrayInputStream b =
      new ByteArrayInputStream(buf);
    PartP.Builder protocPart =
     ProtocolBufferSinglton.getInstance().
     getPart();
    CodedInputStream stream =
       CodedInputStream.newInstance(b);
    m_part.setName(protocPart.getName());
    m_part.setBrand(protocPart.getBrand());
    ....
    return m_part;
}
```

Since conversion between our Java implementation and the PB representation is, strictly speaking, unnecessary, we wanted to make sure that this did not introduce any additional cost. Through a series of micro-experiments, we found that conversion was costless. For example, we found that deserialization of a million `Part` objects *without* conversion (that is, simply calling all of the PB getter methods) took 1.185 seconds. Doing the same thing, converting into Java objects, took 1.183 seconds. This is not surprising; conversion into a `Part` object requires only a single additional Java allocation (a new `Part` object). Populating this object requires only simple Java Object reference assignments, or primitive type assignments.

4. Java objects with hand-coded using Java ByteBuffer. We also hand-coded

our own Java Object serialization. This should serve as something of an upper bound on the performance of any Java method that does not use the Java "unsafe" interface (which has perennially faced removal with each subsequent release of Java). All handcoded serialization directly encodes the various data structures by writing bytes, ints, and doubles into a Java `byteBuffer`. In our implementation, to serialize an Object, we first convert all strings to byte arrays. After this, we compute the size of the object and allocate a `ByteBuffer` with that size. After this step, write all bytes, in order into the buffer. Strings are pre-prended with their size, in bytes. We then convert this `ByteBuffer` to a byte array. Deserialization reverses this process. Here is an example serialization code:

The following Java code snippet illustrates our process for the handcoded serialization in Java.

```
byte[] commentBytes =
  comment.getBytes(Charset.forName("UTF-8"));
ByteBuffer byteBuffer =
  ByteBuffer.allocate(40 + ... +
  commentBytes.length);
byteBuffer.putInt(partID);
byteBuffer.putInt(size);
byteBuffer.putDouble(retailPrice);
...
byteBuffer.putInt(commentBytes.length);
byteBuffer.put(commentBytes);
```

```
return byteBuffer.array();
```

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 in interchange and storage format. This is required since performing the computations required by our experiments directly on a `javax.json.JsonObject` object is not possible.

An example of the text associated with a JSON `Part` object is:

```
{"partid":6038,
 "name":"dim almond red burnished lace",
 "mfgr":"Manufacturer#2",
 "brand":"Brand#24",
 "type":"MEDIUM ANODIZED TIN",
 "container":"WRAP PKG",
 "retailprice":944.03,
 "comment":" deposits."
}
```

For the Java JSON serialization we use the `javax.json` package. We use the `JsonObjectBuilder` to create a JSON object in Java and the `JsonWriterFactory` to write the JSON object properties to the `JsonObjectBuilder`, and then use `JsonWriter` to write the created JSON object into an a `ByteArrayOutputStream`. To compress the output array we first use the `JsonWriter` to write the output stream to a `GZIPOutputStream`

which is initialized with a `ByteArrayOutputStream`. At the end we can convert the `ByteArrayOutputStream` to a byte array. In the case of class that include lists of sub-objects, we need to have separate `JsonObjectBuilder`, and use a `for` loop to iterate over the list, and generate each of the sub-objects, which are added to the parent JSON object.

The deserialization process is the inverse of the serialization process. We first obtain an array bytes, and use `GZIPInputStream` and `ByteArrayInputStream` and convert the input stream to a `JsonReader` and then to a `JsonObject`, which is then converted to a `Part`, `Lineitem` or `Customer` type by using the JSON object's getter/setter methods to get the JsonObject content.

6. Java objects with BSON, `bson4jackson` version 2.7.0 and the `FasterXML/jackson` packages. Again, we use standard Java objects as our in-memory representation.

To serialize those objects, we first initialize a `ByteArrayOutputStream` and access a `de.undercouch.bson4jackson.BsonFactory` which is set to our output stream and create a `JSONGenerator` of type `com.fasterxml.jackson.core.JsonGenerator`. Then we access the object attributes using the getter methods and write them to `JsonGenerator`. The `BsonFactory` coverts this JSON to BSON format and writes it to the output stream. For each attribute of the object we first write a field name and then write values, is in the following Java code snippet:

```
ByteArrayOutputStream baos =
  new ByteArrayOutputStream();
com.fasterxml.jackson.core.JsonGenerator gen
```

```
  = BsonSinglton.getInstance().getFactory().
  createJsonGenerator(baos);
gen.writeStartObject();
gen.writeFieldName("partid");
gen.writeNumber(this.getPartID());
gen.writeFieldName("name");
gen.writeString(this.getName());
...
return baos.toByteArray();
```

To avoid re-instantiation of `BsonFactory` objects we implemented the singleton pattern.

The deserialization process starts with with conversion of a byte array to an input stream, and then we use `JsonParser` and `BsonFactory` to convert the stream to a parser and access its tokens. We create a new object of the requested object type and use the parser's getters to access the various fields (`parser.getIntValue()`, `parser.getText()`, etc.). This is given in the following Java code snippet:

```
byte[] buf;
ByteArrayInputStream bais =
  new ByteArrayInputStream(buf);
Part tmp = new Part();
com.fasterxml.jackson.core.JsonParser
  parser = BsonSinglton.getInstance().
```

```
    getFactory().createJsonParser(bais);
parser.nextToken();
while (parser.nextToken() !=
    JsonToken.END_OBJECT) {
    String fieldname =
        parser.getCurrentName();
    parser.nextToken();
    switch (fieldname) {
    case "partid":
        tmp.setPartID(parser.getIntValue());
        break;
    case "name":
        tmp.setName(parser.getText());
        break;
    ...
    }
return tmp;
```

For the serializaiton of this kind of complex, recursive objects, the `bson4jackson` package provides an API (the `ObjectMapper` API) that iterates over a list of objects and serializes them.

7. C++ objects with BOOST serialization/deserialization, version 1.59. BOOST is the classical C++ serialization package, and one of the options that we consider. To

use BOOST, we simply design C++ classes for each of the various data types, using primitive types such as `int`, `double`, and the STL type `string`. To represent collections of sub-objects, we use the STL vector. All of those types can be automatically serialized by BOOST.

One challenge with BOOST was providing random access to objects on a page, without deserializing the entire page. For de/serializing a stream of objects. Boost provides *archives* that can be used as de/serializer, bound to a C++ stream (either a stringstream or a fstream). During serialization, boost archive prepends some internal information (like a small header) at the beginning of the stream. The deserializer archive needs to parse this header first, before it can parse the stream of objects.

```
// Serialization
stringstream ss = new stringstream();
boost::archive::binary_oarchive oa(ss);
oa<<Object_1<<Object_2<<...<<Object_n;

// Deserialization
stringstream ss = new stringstream();
boost::archive::binary_iarchive ia(ss);
ia>>Object_1>>Object_2>>...>>Object_n;
```

In our data aggregation experiment, since we de/serialize an entire sparse vector at once, we use BOOST in this way. Each entry in the sparse vector is stored as a separate object, with a single header for all objects. However, for other experiments,

we need random access to objects (to perform random reads of objects or to perform sequential reads of objects starting from a random location on a page). Hence, we serialize each object with a separate instance of boost archive. Thus, a small header is appended at the beginning of each object. The downside of this is that it increased the average object size and adds some extra overhead to processing each each object; the overhead is higher, relatively speaking, for smaller objects. In a sense, this arises from the fact that we are using BOOST in a way that it was not intended to be used: for random access to individual objects in a large page.

8. C++ objects with hand-coded serialization/deserialization. Since we hand-code de/serialization when implementing this option, the choice should be considered to provide an upper-bound on the performance of any C++-based method that requires serialization/ deserialization (note, however, that C++ in-place objects *do not* require de/serialization, and hence may have better performance).

The encoding of data was nearly identical to the encoding used for BOOST: `int`, `double`, `vector`, etc. The difference is that de/serialization was implemented by hand. The primary key-foreign key relationships in the TPC-H relational diagram are implemented as object pointers/references.

For example, a `Lineitem` object has a `Part` and a `Supplier`:

```
class LineItem  {
        // Primitive variables:
        // ...
        // Object types:
```

```
        Supplier *supplier;
        Part *part;
}
```

An order object has a collection of lineitems. To represent collections we use the STL vector:

```
class Order {
        // Primitive variables:
        // ...
        // collection of lineitems:
        vector<LineItem*> lineItems;
}
```

During serialization, primitive types are written directly to the output page. `std::string` objects are serialized by first recording the length of the string, followed by the characters. Serializing object pointers or collections of pointers to objects requires a recursive serialization.

9. C++ objects with Protocol Buffers, version 2.6.1. Protocol buffers have been described above, and the C++ implementation does not differ significantly from the Java implementation. However, in the case of C++, we do not translate into a separate representation. The C++ PB classes are simple wrappers over the PB compiler generated classes.

10. In-Place C++ objects. We have previously described the philosophy behind

in-place objects in detail. In practice, in-place objects rely on C++ to pack primitive types within a contiguous region of memory. `offset_ptr`s are used to implement character strings and arrays of sub-objects. For example, we have:

```
class CustomerIP : public Object {
private:
  // Primitive variables:
  int numOrderSupplied;
  int custKey;
  int nationKey;
  double accbal;
  // Strings:
  offset_ptr <char> name;
  offset_ptr <char> address;
  offset_ptr <char> phone;
  offset_ptr <char> mktsegment;
  offset_ptr <char> comment;
  // Collection of orders:
  offset_ptr <OrderIP> orders;
};
```

During object creation, allocation is performed using a special, in-place memory manager that allocates objects directly to a page. This is accessed via a custom, templated `malloc` function. For example, to initialize the `name` field from a C++ `std::string`,

we have:

```
this->name = malloc <char> (strlen (name.c_str ()) + 1);
strcpy (this->name, name.c_str ());
```

Since the `Object` base class over-rides `new`, collections (arrays) of objects are allocated in-place using the C++ `new` operator:

```
// Size of the order vector:
this->numOrderSupplied = numOrderSupplied;
// Dynamically create offset_ptr's
this->orders = new OrderIP [numOrderSupplied];
// Create orders using InPlace memory manager
for (int i = 0; i < numOrderSupplied; i++) {
  (this->orders)[i].setUp(...);
}
```

Bibliography

[1] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[2] R. W. Taylor and R. L. Frank, "Codasyl data-base management systems," *ACM Computing Surveys (CSUR)*, vol. 8, no. 1, pp. 67–103, 1976.

[3] J. P. Strickland, P. P. Uhrowczik, and V. L. Watts, "Ims/vs: An evolving system," *IBM Systems Journal*, vol. 21, no. 4, pp. 490–510, 1982.

[4] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik, "The object-oriented database system manifesto.," in *DOOD*, vol. 89, pp. 40–57, 1989.

[5] M. Stonebraker and D. Moore, *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc., 1995.

[6] B. Stroustrup, "A history of c++: 1979–1991," in *History of programming languages—II*, pp. 699–769, ACM, 1996.

[7] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of java generics," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1047–1089, 2013.

[8] T. White, *Hadoop: the definitive guide*. O'Reilly Media, Inc., 2009.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *USENIX HotCloud*, pp. 1–10, 2010.

[10] Y. Yu *et al.*, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.," in *OSDI*, vol. 8, pp. 1–14, 2008.

[11] A. Alexandrov *et al.*, "The Stratosphere platform for big data analytics," *VLDBJ*, vol. 23, no. 6, pp. 939–964, 2014.

[12] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *Proc. VLDB Endow.*, vol. 3, pp. 330–339, Sept. 2010.

[13] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann, "Asterixdb: A scalable, open source bdms," *Proc. VLDB Endow.*, vol. 7, pp. 1905–1916, Oct. 2014.

[14] K. Chodorow, *MongoDB: the definitive guide.* " O'Reilly Media, Inc.", 2013.

[15] D. Crockford, "The application/json media type for javascript object notation (json)," 2006.

[16] K. Varda, "Protocol buffers: Google's data interchange format," *Google Open Source Blog*, 2008.

[17] A. Gokhale and D. C. Schmidt, "Measuring the performance of communication middleware on high-speed networks," in *ACM SIGCOMM Computer Communication Review*, vol. 26, pp. 306–317, ACM, 1996.

[18] A. S. Gokhale and D. C. Schmidt, "Evaluating corba latency and scalability over high-speed atm networks," in *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pp. 401–410, IEEE, 1997.

[19] A. S. Gokhale and D. C. Schmidt, "Measuring and optimizing corba latency and scalability over high-speed networks," *IEEE Transactions on Computers*, vol. 47, no. 4, pp. 391–413, 1998.

[20] F. Plasil, P. Tuma, and A. Buble, "Corba benchmarking," *Charles University, Prague*, 1998.

[21] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, and A. Zivkovic, "Object serialization analysis and comparison in java and .net," *SIGPLAN Not.*, vol. 38, pp. 44–54, Aug. 2003.

[22] K. Maeda, "Performance evaluation of object serialization libraries in xml, json and binary formats," in *Digital Information and Communication Technology and it's Applications (DICTAP), 2012 Second International Conference on*, pp. 177–182, IEEE, 2012.

[23] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml)," *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, vol. 16, p. 16, 1998.

[24] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: the definitive guide.* " O'Reilly Media, Inc.", 2010.

[25] D. Shukla, S. Thota, K. Raman, M. Gajendran, A. Shah, S. Ziuzin, K. Sundaram, M. G. Guajardo, A. Wawrzyniak, S. Boshra, *et al.*, "Schema-agnostic indexing with azure documentdb," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1668–1679, 2015.

[26] S. Sivasubramanian, "Amazon dynamodb: a seamlessly scalable non-relational database service," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 729–730, ACM, 2012.

[27] "Kryo serialization package." https://github.com/EsotericSoftware/kryo, 2016.

[28] B. Karlsson, *Beyond the C++ standard library: an introduction to boost*. Pearson Education, 2005.

[29] "Google flat buffers." https://google.github.io/flatbuffers/flatbuffers_white_paper.html, 2016.

[30] "Bson interchange format." http://bsonspec.org, 2016.

[31] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, and S. B. Zdonik, "Tupleware:" big" data, big analytics, small clusters.," in *CIDR*, 2015.

[32] T. P. P. Council, "The tpc benchmark (tpc-h) a decision support benchmark for an ad-hoc decision support system.." http://www.tpc.org/tpch/, 1999.

[33] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*, pp. 177–186, Springer, 2010.

[34] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, *et al.*, "C-store: a column-oriented dbms," in *Proceedings VLDB*, pp. 553–564, VLDB Endowment, 2005.