Complex Object Implementations for Big Data Systems

Saeed Fathollahzadeh
Free University of Bozen-Bolzano
saeed.fathollahzadeh@unibz.it

Kia Teymourian
Boston University
kiat@bu.edu

Chris Jermaine
Rice University
cmj4@rice.edu

ABSTRACT

abstract

PVLDB Reference Format:
Ben Trovato, G. K. M. Tobin, Lars Thørväld, Lawrence P. Leipuner, Sean Fogarty, Charles Palmer, John Smith, Julius P. Kumquat, and Ahmet Sacan. Complex Object Implementations for Big Data Systems. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Introduction

The contribution of our work are the following:

1. We implement all serialization methods in C++ and Java programming language in a single thread system. But, we evaluate the methods with *taskset* for restrict run the method on a special core and without *taskset* to allow run method on the arbitrary cores. In the experimental section we mention which methods use thread or which platforms want to improve performance of processing.

2. We implement same method in both C++ and Java programming language and we demonstrated same technique haven't same performance in differ languages(e.g, google protobuf).

3. We compare all methods with a complex data sets. In academic setting over the last decade, there has been significant progress in serialization methods. However, much of this work makes assumptions that are simply unrealistic for deployed industrial applications. In this work, we used twitter dataset. This dataset include more objects type with deep hierarchy. Some methods need to save object meta data in serialization step and will be use it in the de-serialization section.

4. We investigate which methods are easy to used. It means is which methods create transparent view in develop step. For example in C++ *InPlace* we need just one line for de-serialization, But in the serialization step we should spend more times for convert object to the method schema.

5. We evaluate multiple famous serialization methods in big data systems. We focused specially on C++ and Java programming language. So, we deeply compared CPU, Memory and I/O for HDD resources. Our empirical experiments demonstrate best way for choose best method in a big data system.

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

2. EXPERIMENTAL OVERVIEW

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

1. A set of serialized objects stored externally on an HDD; 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.

2.1 Twitter Data Set

For the various experiments, we use twitter data sets [1], implemented using each of the ten different physical implementations.

2.2 Encoding sizes

The ten different complex object implementations that we considered have very different encoding densities when the objects are serialized for storage or transmission across the network. The average, per-object sizes are given in Table

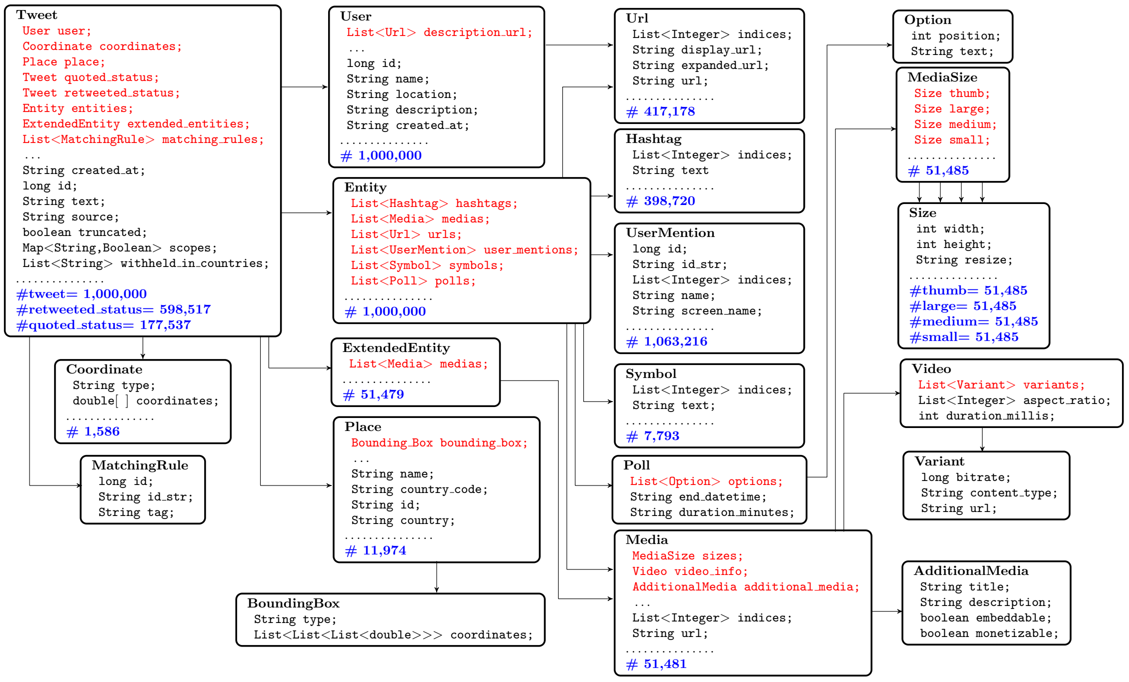

Figure 1: Object relationship and frequency of Tweet Objects (for one million tweets)

Table 1: tweet complexity

Tweet type	Frequency
Simple tweets(retweet & quote are null)	332,901
Retweets	489,562
Quote	68,582
Retweet & Quote	108,955
Total	1,000,000

Table 2: Object size for 1 million tweets and Lines of code for serialize/de-serialize

Method	File size(Gig)	Serialize *Lines of code*	De-Serialize *Lines of code*
Java Default	4.6	4	4
Java Json+Gzip	1.4	2	4
Java Bson	4.9	50	120
Java ProtoBuf	1.9	200	1
		with 20 extra files	
Java Kyro	1.9	40	40
Java HandCoded ByteBuf.	2.3	150	150
Java FaltBuffers	2.9	250	1
		with 42 extra files	
C++ HandCoded	2.1	70	100
C++ InPlace	3.2	80	1
C++ Boost	2.2	1	2
C++ ProtoBuf	1.9	200	1
		with 20 extra files	
C++ Bson	4.6	40	100
C++ FaltBuffers	2.9	250	1
		with 42 extra files	
Rust Serde Json	4.8	1	1
Rust Serde Bincode	2.4	1	1
Rust Serde MessagePack	1.9	1	1
Rust Serde Bson	4.5	1	1
Rust Serde FlexBuffers	4.3	1	1

2.3 Experimental Details

We run our experiments on Google Cloud costume instances which have 4 vCPU cores, 32 GB RAM and 3000 GB standard persistent disk(Sustained random IOPS limit: read=2,250 and write=4,500) running with Ubuntu Ubuntu 18.04.4 LTS. Before running each experiment task, we "warmed up" the Java Garbage Collector (GC) by creating a large number of objects. We do not include this warm-up-time in our performance time calculations.

We used two Java GC flags $-XX:-UseGCOverheadLimit$ and $-XX:+UseConcMarkSweepGC$. The first flag is used to avoid OutOfMemoryError exceptions while using the complete RAM size for data processing and the second flag is for running concurrent garbage collection.

We run all of our experiments 3 times and observed that the results have low variance. In this paper we present the average of those runs. Before running each experiment, we deleted th OS cache using the Linux command: $echo3 > /proc/sys/vm/drop\_caches$.

Our Java implementation is written using Java 8 with the Oracle JDK version "1.8.0_241" and for our C++ implementation we use the C++11, compiled using clang++ (version 6.0.0).

The source codes of our implementation and a brief description of technical details can be found on the Github Repository [1]
.

3. EXPERIMENTS

[1]The source code of our Implementation is available at `https://github.com/fathollahzadeh/serialization`

```java
/* Kryo serialization method */
public byte[] serialize( Kryo kryo) {
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  Output output = new Output(bos);
  kryo.writeObject(output, this);
  output.flush();
  return bos.toByteArray();}
/* Kryo de-serialization method */
public Tweet deserialize(byte[] buf, Class<?> cls, Kryo kryo) {
  ByteArrayInputStream bis = new ByteArrayInputStream(buf);
  Input input = new Input(bis);
  Tweet obj= kryo.readObject(input, cls);
  return obj;}
```

a) Java Kryo

```cpp
/* HandCoded serialization method */
void serialize(char *buffer, int &objSize) {
  //Write "long" value.
  memcpy(buffer, &id, sizeof(id));
  buffer += sizeof(id);
  objSize += sizeof(id);
  //Write "length" of the string.
  int strlen = created_at.length();
  memcpy(buffer, &strlen, sizeof(strlen));
  buffer += sizeof(strlen);
  objSize += sizeof(strlen);
  //Write the content of the string.
  memcpy(buffer, created_at.c_str(), strlen);
  buffer += strlen;
  objectSize += strlen;}
/* HandCoded de-serialization method */
Tweet deserialize(char *buffer, int &bytesRead) {
  Tweet obj;
  //Read "long" value.
  memcpy(&obj.id, buffer, sizeof(long));
  bytesRead += sizeof(obj.id);
  //Read "length" of the string.
  int strlen;
  memcpy(&strlen, buffer+bytesRead, sizeof(strlen));
  buffer += sizeof(strlen);
  //Read the content of the string.
  obj.create_at.assign(buffer, strlen);
  bytesRead += sizeof(int) + strlen;
  //Read bool value.
  memcpy(&obj.truncated, buffer, sizeof(bool));
  bytesRead +=sizeof(bool);
  return obj; }
```

b) C++ HandCoded

```rust
/* Serde Json serialization method */
pub fn serialize( Tweet obj)-> &[u8] {
  let r=serde_json::to_string(&obj).unwrap();
  return r.as_bytes(); }
/* Serde Json de-serialization method */
pub fn deserialize(buff: &[u8])-> Tweet {
  let obj=serde_json::from_slice(buff).unwrap();
  return obj;}
```

c) Rust Json

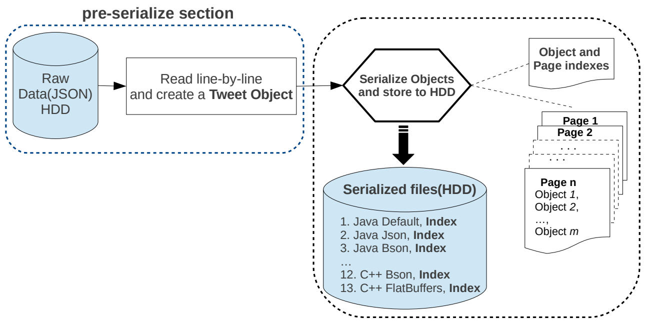

Figure 2: serialize process

3.1 Serialize RAW Data to Local Disk

The first step of experiments are serialize various complex objects and the write into a file in disk. In this experiment, the raw tweet data set read line-by-line and convert to a objects. The serialization tasks for each of the thirteen implementation method run. In the serialization process each object serialized or copy the final serialization result into $256KB$ pages and the objects indexed in separate file.

3.1.1 Results

In Figure 3, we show, for each of the thirteen implementations, for both *taskset* TRUE and FALSE the total running time required as a function of the number of Tweet objects write experiments. In the figure where the performance differences are easier to see; we also breakdown the total time into I/O and CPU.

3.1.2 Discussion

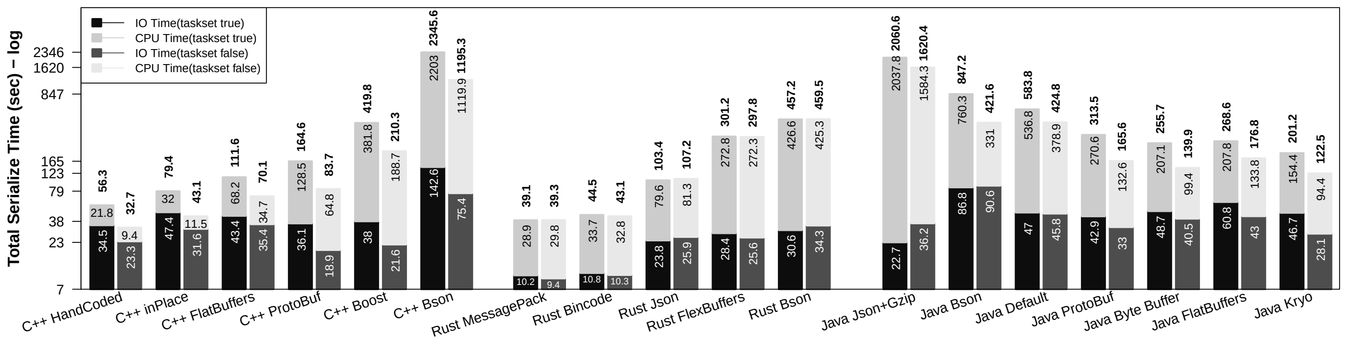

Figure 3: Serialize Objects for 5M Tweets

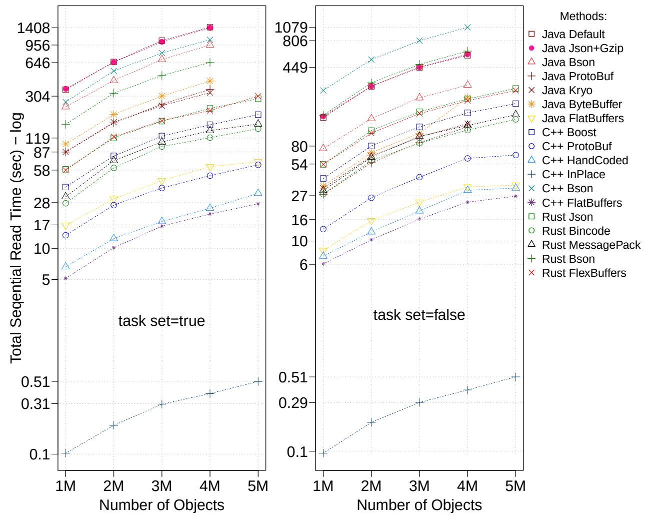

Figure 4: sequential read

3.2 I/O FROM LOCAL DISK

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 tweet data set is first loaded onto the HDD drive of a machine 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 de-serialized 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 $\{\ 1 \times 10^6$, 2×10^6, 3×10^6, 4×10^6, $5 \times 10^6\ \}$.

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 de-serialized from the page. This simulates a scenario where objects are retrieved from secondary storage using a secondary index.

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

3.2.1 Results

3.2.2 Discussion

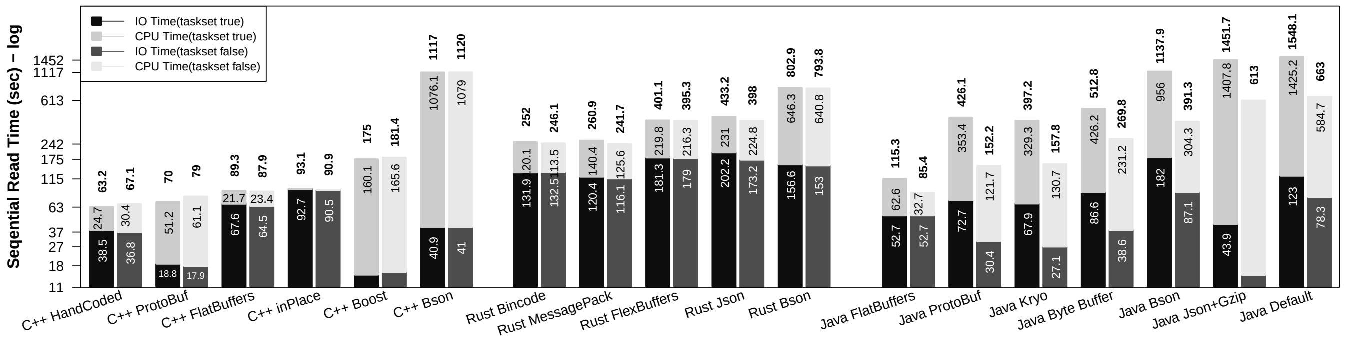

Figure 5: CPU and IO details of sequential read for 4M tweets

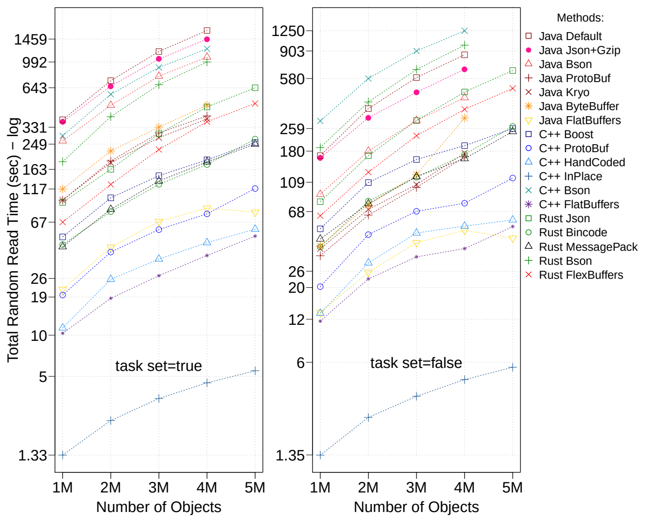

Figure 6: random read

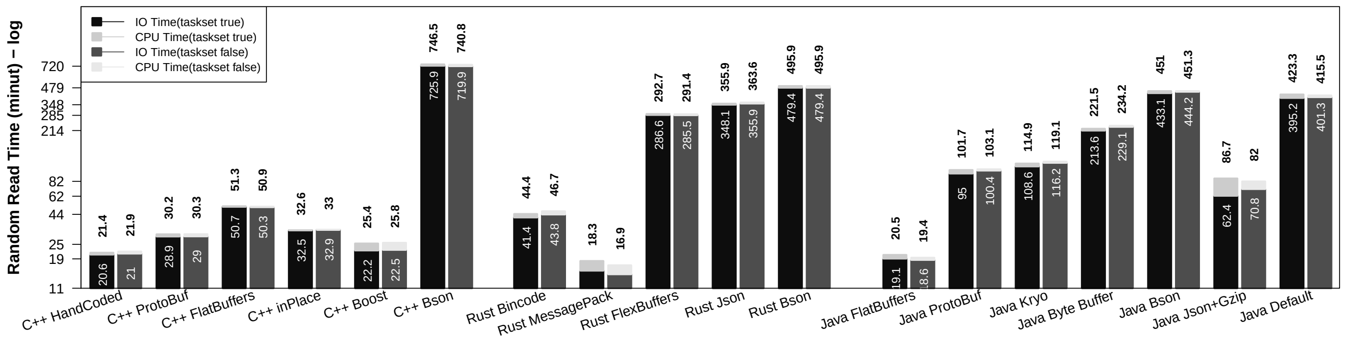

Figure 7: CPU and IO details of random read for 4M tweets

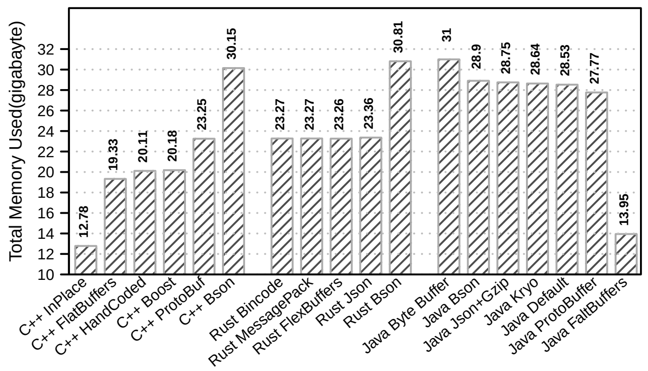

Figure 8: memory used in read objects for 4M Tweets

3.3 Exp. Memory usage

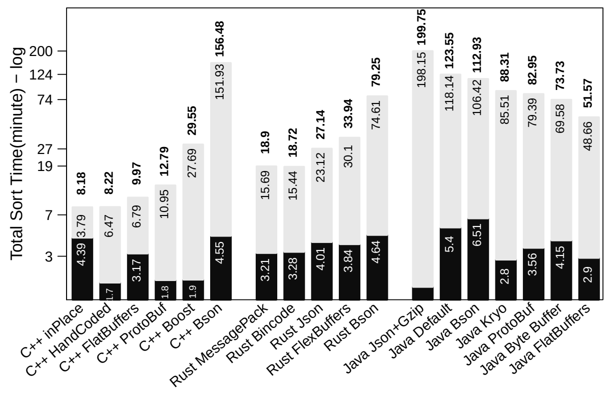

Figure 9: external sort for 10M tweet

3.4 Exp. External Sort

4. REFERENCES

[1] Tweet objects.