Twitter data serialization methods and benchmarks

Saeed Fathollahzadeh
Free University of Bozen-Bolzano
s.fathollahzadeh@gmail.com

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. Twitter data serialization methods and benchmarks. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

Introduction

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.

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

Table 1: Frequency of some Tweet Objects (for 1 million tweets)

Object Name	Parent Object	Frequency
tweet	root object	1,000,000
users	tweet	1,000,000
coordinates	tweet	1586
place	tweet	11974
quoted status	tweet	177537
retweeted status	tweet	598517
entities	tweet	1,000,000
extended entities	tweet	51479
hashtags	entities	398720
media	entities	51481
urls	entities	417176
user mentions	entities	1063216
symbols	entities	7793
sizes	media	7793
media sizes	sizes	51485
thumb	media sizes	51485
large	media sizes	51485
medium	media sizes	51485
small	media sizes	51485

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 2.

2.3 Experimental Details

We run our experiments on a virtual server instances which have 8 vCPU cores, 32 GB RAM and one 800 GB hard disks (Network Instance Store) running with Ubuntu 18.04.3 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

Table 2: Comparison of object size for 1 million tweet

Serialization Methods	Serialized file size(gigabyte)
Java Default	4.6
Java JSON	4.3
Java BSON	4.6
Java Protocol Buffer	1.9
Java Kyro	1.9
Java Hand Coded ByteBuffer	2.3
C++ Hand Coded	2.1
C++ InPlace	3.2
C++ Boost	2.2
C++ Protocol Buffer	1.9

is for running concurrent garbage collection.

We run all of our experiments 5 times and observed that the results have low variance. In this paper we present the average of those runs. Before running each experiment, we deleted th OS cache using the Linux command: $echo3 > /proc/sys/vm/drop_caches$.

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`

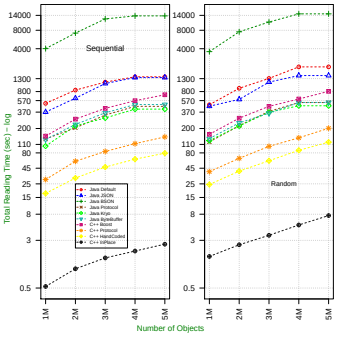

Figure 1: sequential and random read

3.1 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, tweet data set is first loaded onto the HDD 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 $\{ 1 \times 10^6 , 2 \times 10^6, 3 \times 10^6, 4 \times 10^6, 5 \times 10^6 \}$.

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

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

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

3.1.1 Results

3.1.2 Discussion

4. REFERENCES

[1] Tweet objects.