# An Experimental Comparison of Complex Object Implementations for Big Data Systems

**Kia Teymourian**

Joint work with
Sourav Sikdar, Chris Jermaine

Partially Published in:

Sourav Sikdar, Kia Teymourian, and Chris Jermaine. 2017. **An experimental comparison of complex object implementations for big data systems.** In Proceedings of the 2017 Symposium on Cloud Computing (**SoCC '17**). ACM, New York, NY, USA, 432-444. DOI: https://doi.org/10.1145/3127479.3129248

# Introduction

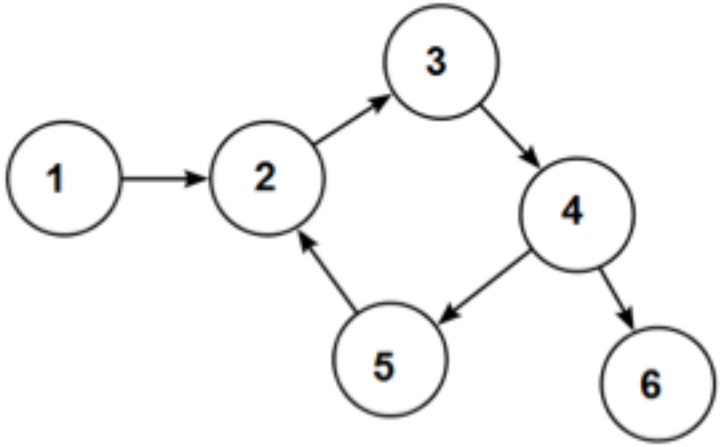
- Relational databases store records made of flat types.
  - integer, float, boolean, char etc.
- All the records have fixed size.

- Example: A student database.

| Last Name | First Name | Student ID | Net ID | SSN | … |
|---|---|---|---|---|---|
| Doe | John | S012141* | jd* | *4768 | … |
| Roe | Jane | S012142* | jr* | *4321 | … |
| … | | | | | |
| … | | | | | |

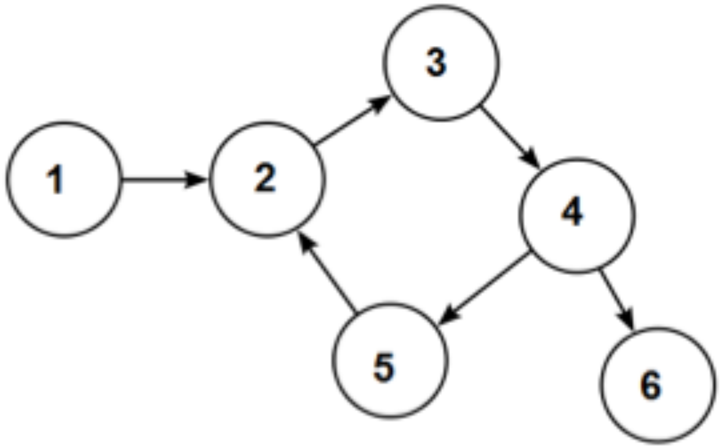
# Introduction

- How do relational databases store complex objects, e.g., graphs?
  - Complex Objects have variable size and are highly nested.



# Introduction

- How do relational databases store complex objects?
  - Complex Objects have variable size and are highly nested.




| Graph ID | … |
|---|---|
| 1 | … |
| … | … |

Graphs

| Graph ID | Vertex ID | … |
|---|---|---|
| 1 | 1 | … |
| 1 | 2 | |
| … | | |

Vertices

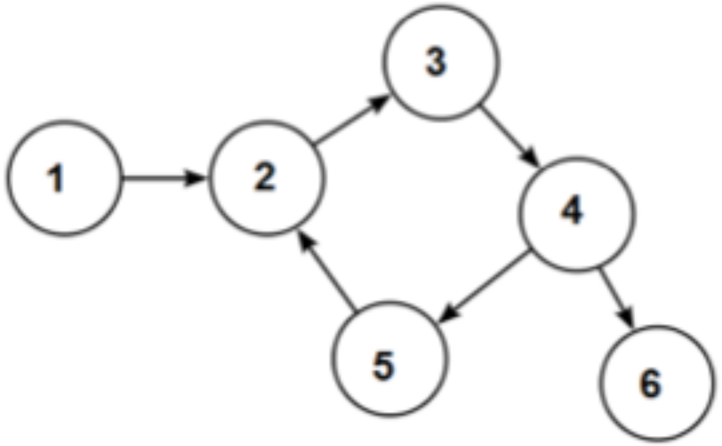
| Graph ID | from | to | … |
|---|---|---|---|
| 1 | 1 | 2 | … |
| 1 | 2 | 3 | |
| 1 | 3 | 4 | |
| … | | | |

Edges

# Introduction

- Modern programming languages provide a lot of useful features.
  - Generics (in Java), Templates (in C++).

- Outside relational database -




```java
public class Graph {
    // Set of nodes
    private Map<Integer, Vertex> vertices;

    // Set of directed edges
    private Map<Integer, List<Edge>> edges;
}
```

# Introduction

```
public class Graph {
   // Set of nodes
   private Map<Integer, Vertex> vertices;

   // Set of directed edges
   private Map<Integer, List<Edge>> edges;
}
```

Big Data System:

There are costs associated with -

- Objectification
- Serialization
- Garbage Collection

# Key Questions

Any big data system designer faces some important choices:

- Which data model to use?
- Which implementation for data model to use?
- Which runtime environment to use?

# Goal

Across a variety of data management tasks, experimentally compare the costs associated with various choices of complex object models and implementations.

# Complex Object Models

- Host Language Objects
- Self-Describing Documents
- Custom Data Models

# 1. Host Language Objects

- Which runtime environment to use?
  - Automatic memory managed vs Not
  - Managed(Java) vs Unmanaged (C++)

- Which serialization framework to use?
  - Serialization: Conversion from in memory to on disk representation.

# 1. Host Language Objects

## Java

Java Default

Java ByteBuffer

Java Kryo

## C++

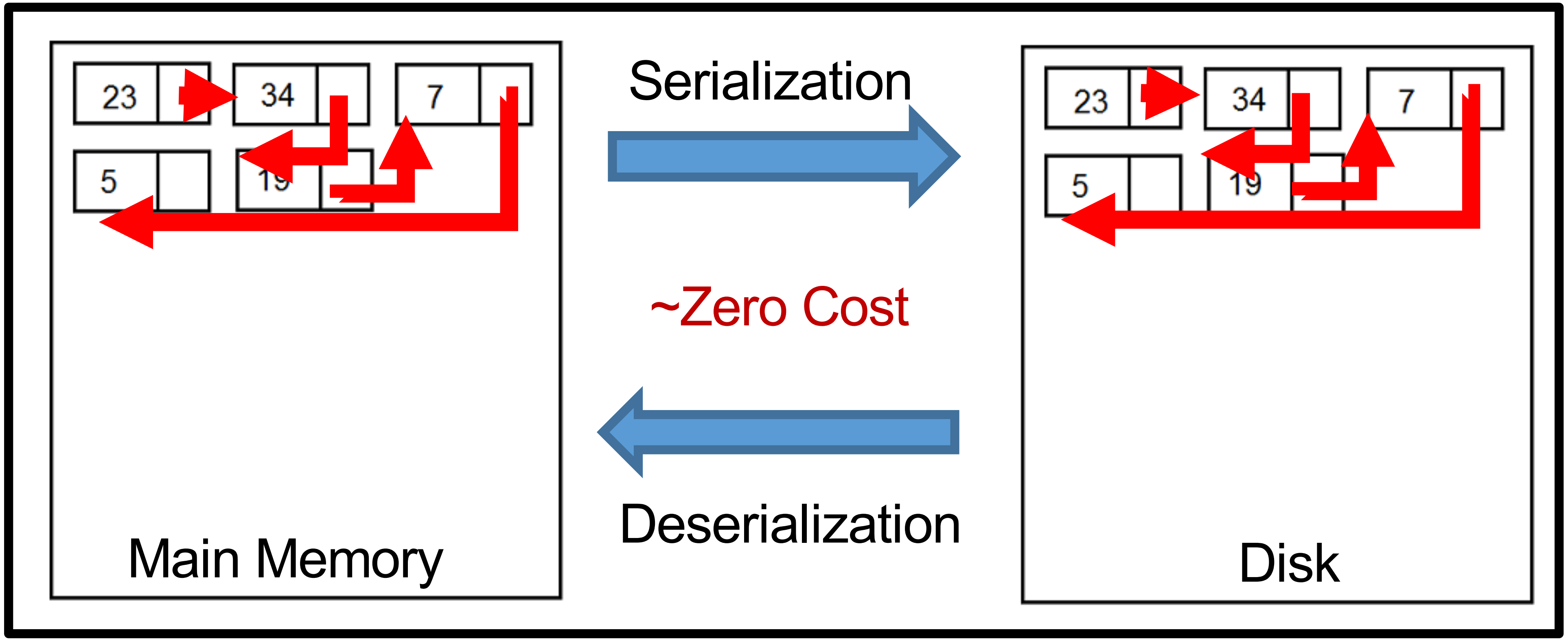
C++ Hand-Coded

C++ Boost

C++ InPlace

# C++ InPlace

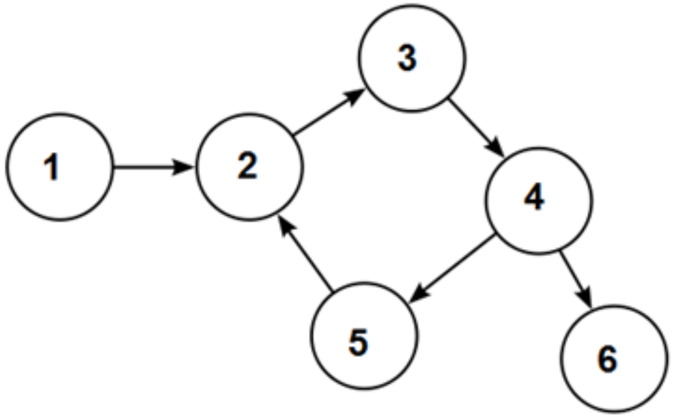
- We borrow the idea from relational database.
  - On disk representation = In memory representation.



# 2. Self-Describing Documents

## JSON + gzip

## BSON




```
{
    "Graph": {
        "Vertices": [1, 2, 3, 4, 5, 6],
        "Edges": {
            "1": [2],
            "2": [3],
            "3": [4],
            "4": [5, 6],
            "5": [2]
        }
    }
}
```

¨....Graph.œ....Vertices./....0..
....1......2......3......4......5.......
Edges.W....1......0.......2......0.
......3......0.......4......0......1.....
..5......0.........

JSON

BSON

# 3. Custom Data Models

Java Protocol Buffers | C++ Protocol Buffers

```
message Graph {
    message Vertex {
        required int32 vertexID = 1;
        //...
    }
    message Edge {
        required int32 fromVertex = 1;
        required int32 toVertex = 2;
        //...
    }
    message AdjacencyList {
        repeated Edge edges = 1;
    }
    map <int32, Vertex> vertices = 1;
    map <int32, AdjacencyList> edges = 2;
}
```

**Graph representation in DSL**

**Compile** →

**JAVA**

```
class Graph {
//…
}
```

**Compile** →

**C++**

```
class Graph {
//…
};
```

# Summary: Object Implementations

| Host-language objects |
|---|
| Java Default |
| Java Kryo |
| Java ByteBuffer |
| C++ Boost |
| C++ HandCoded |
| C++ InPlace |

| Self-Describing Documents |
|---|
| JSON |
| BSON |

| Custom Nested Models |
|---|
| Java Protocol Buffers |
| C++ Protocol Buffers |

# Experiments

- Read from Local Disks

  - Sequential Read (start from random position in file)
  - Random Read (read random pages)

- Network IO

  - Read from 10 Clients RAM push to single server
  - Read from 10 Clients Disk push to single server

- External Sort
- Distributed Data Aggregation

# Dataset

- Average size of a TPC-H Customer object on disk:

| Implementation | Size (Bytes) |
|---|---|
| Java JSON + gzip | 8508 |
| Java Kryo | 16176 |
| Java Protocol Buffers | 17305 |
| C++ Protocol Buffers | 17931 |
| C++ HandCoded | 19275 |
| Java ByteBuffer | 19478 |
| Java Default | 19556 |
| C++ Boost | 21004 |
| C++ InPlace | 25127 |
| Java BSON | 33879 |

Data + Schema + Compression

Data + Light-weight encoding

Data

Data + Headers

Memory Representation of Data

Data + Schema

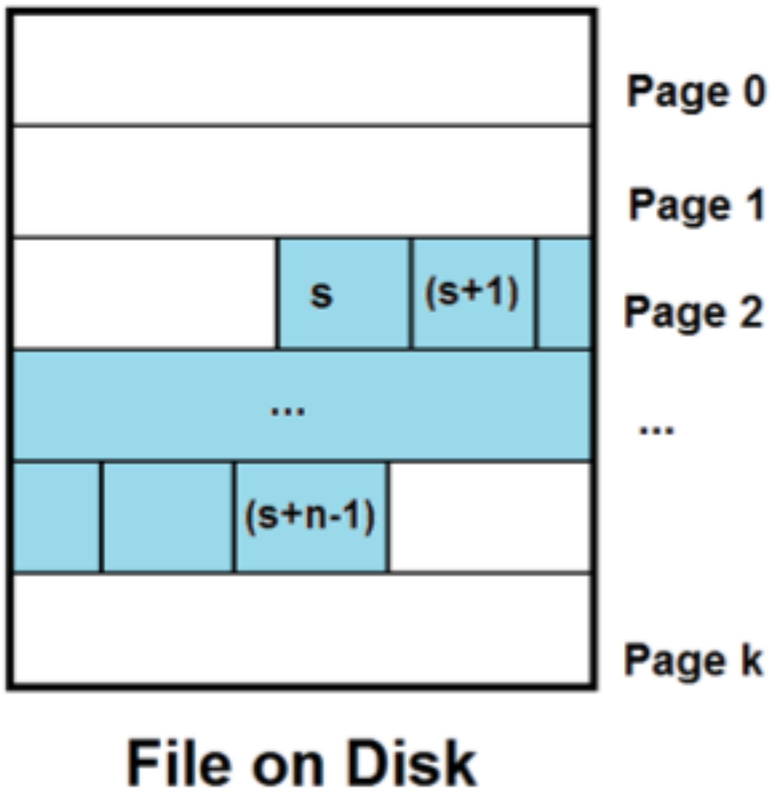
# 1. Sequential Read

**Goal:**
Test the ability to support fast retrieval of objects.

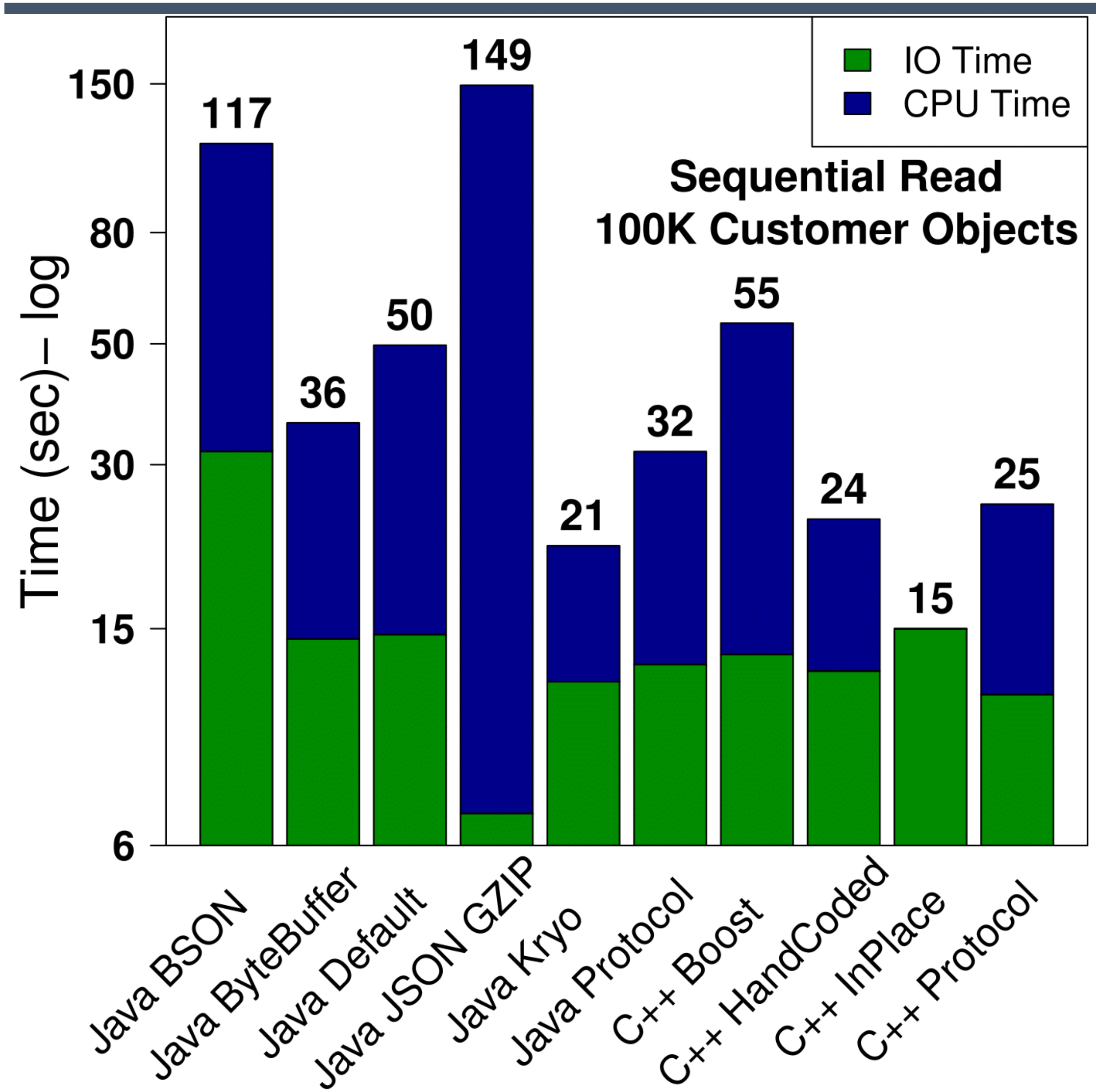
**Task:**

3 million TPC-H Customer objects.

Read 100K objects sequentially.

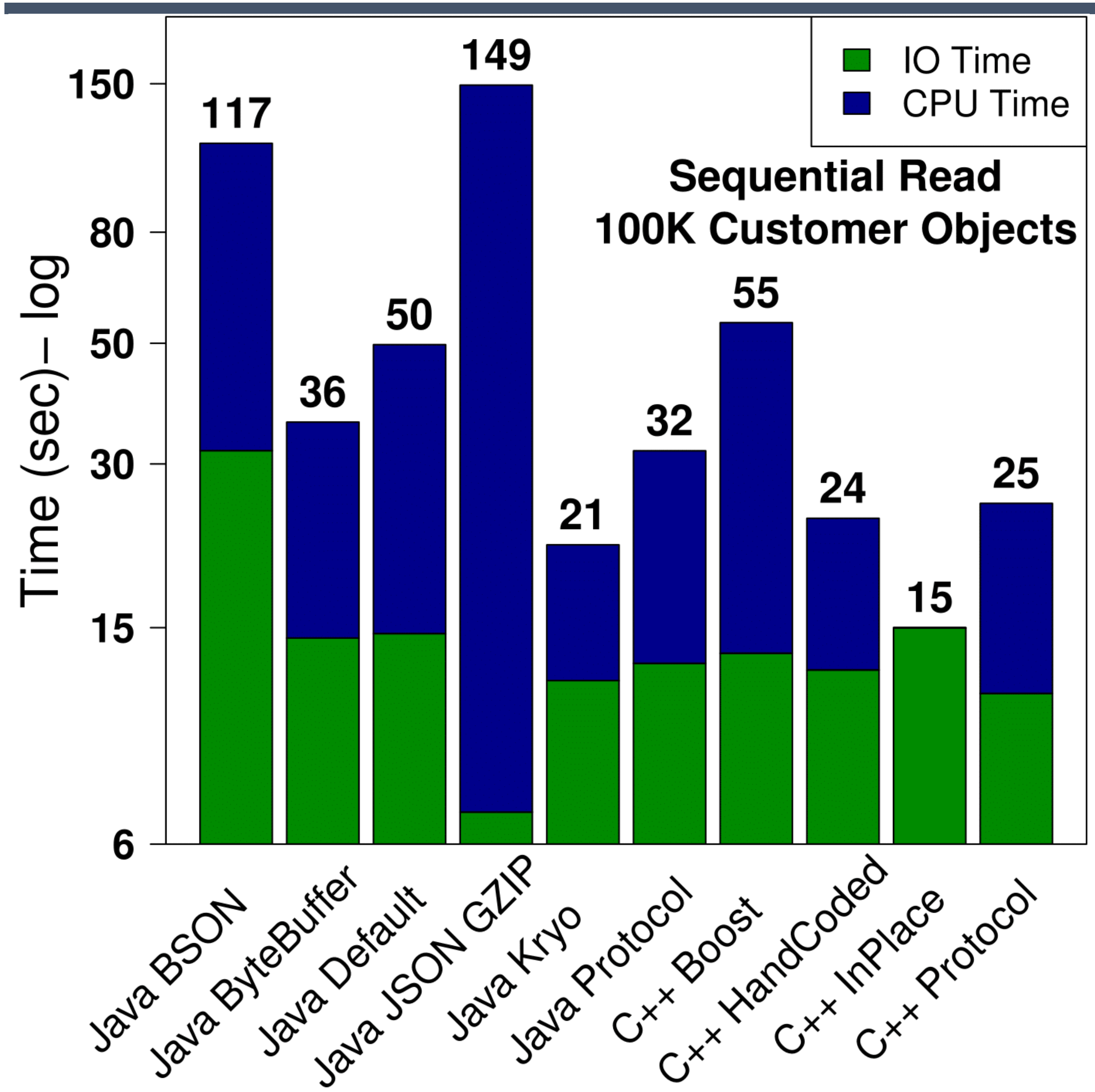


# 1. Sequential Read



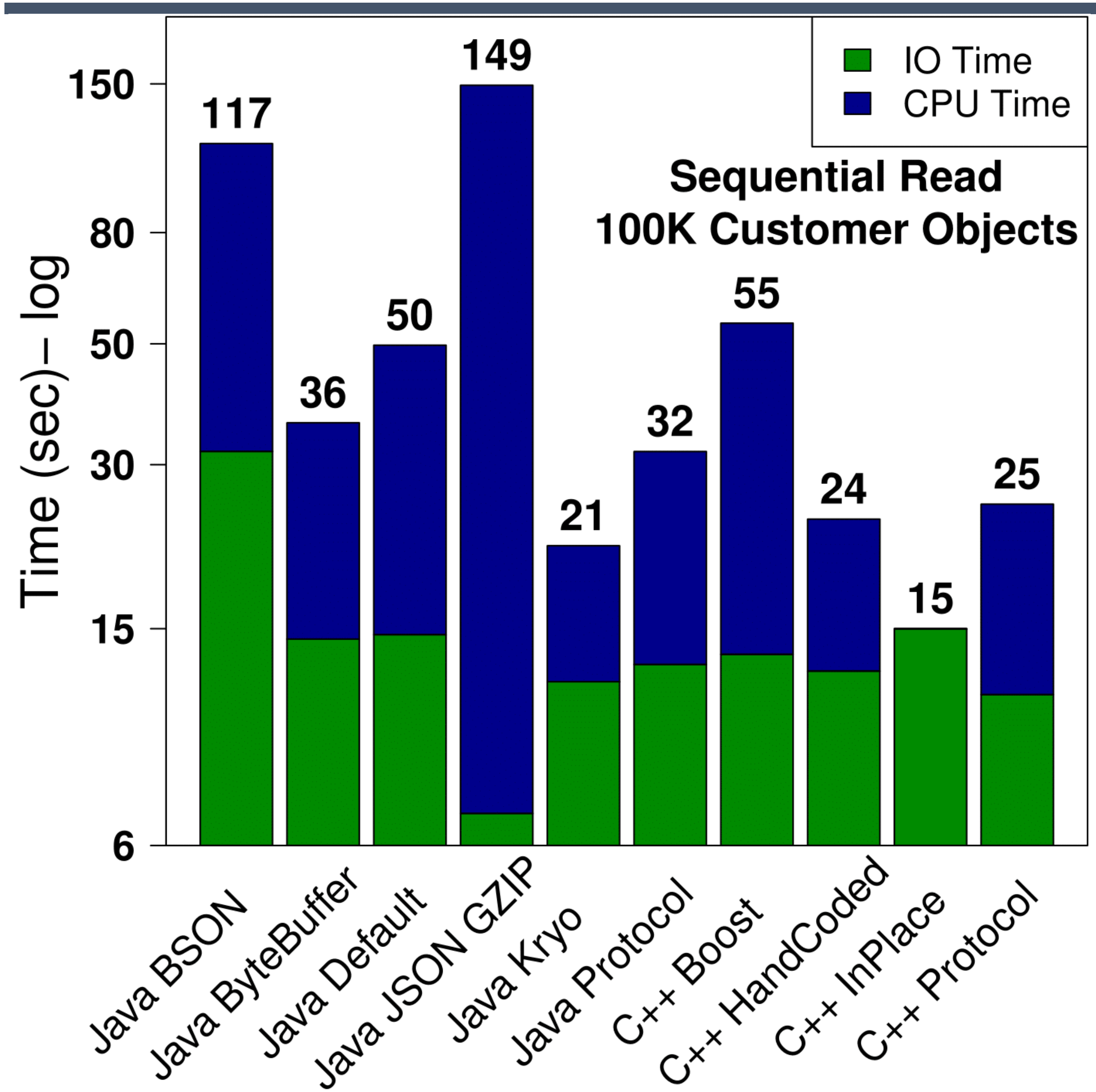

- The fastest C++ implementation (InPlace) is at least 1.5x faster than fastest Java implementation (Kryo) for larger reads.

# 1. Sequential Read




- The fastest C++ implementation (InPlace) is at least 1.5x faster than fastest Java implementation (Kryo) for larger reads.
- The faster C++ implementations are up to 5x-10x faster than document models.

# 1. Sequential Read




- The fastest C++ implementation (InPlace) is at least 1.5x faster than fastest Java implementation (Kryo) for larger reads.

- The faster C++ implementations are upto 5x-10x faster than document models.

- C++ InPlace is IO bound.
- JSON + gzip is CPU bound.

# 2. External Sort

**Goal:**

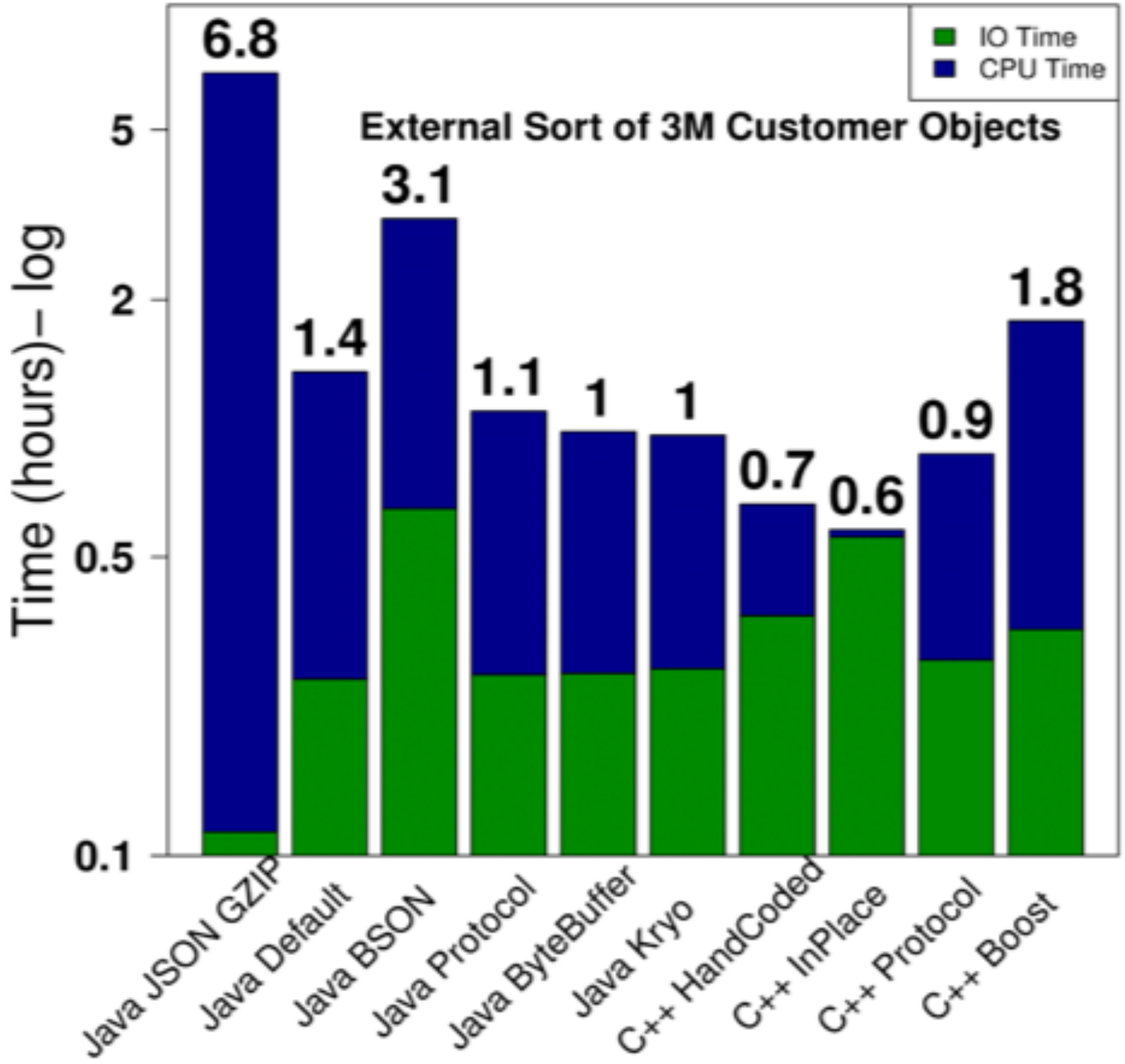
Sorting is common workflow in data management system.

**Details:**

Sorting 3 million TPC-H Customer objects (~ 60GB).

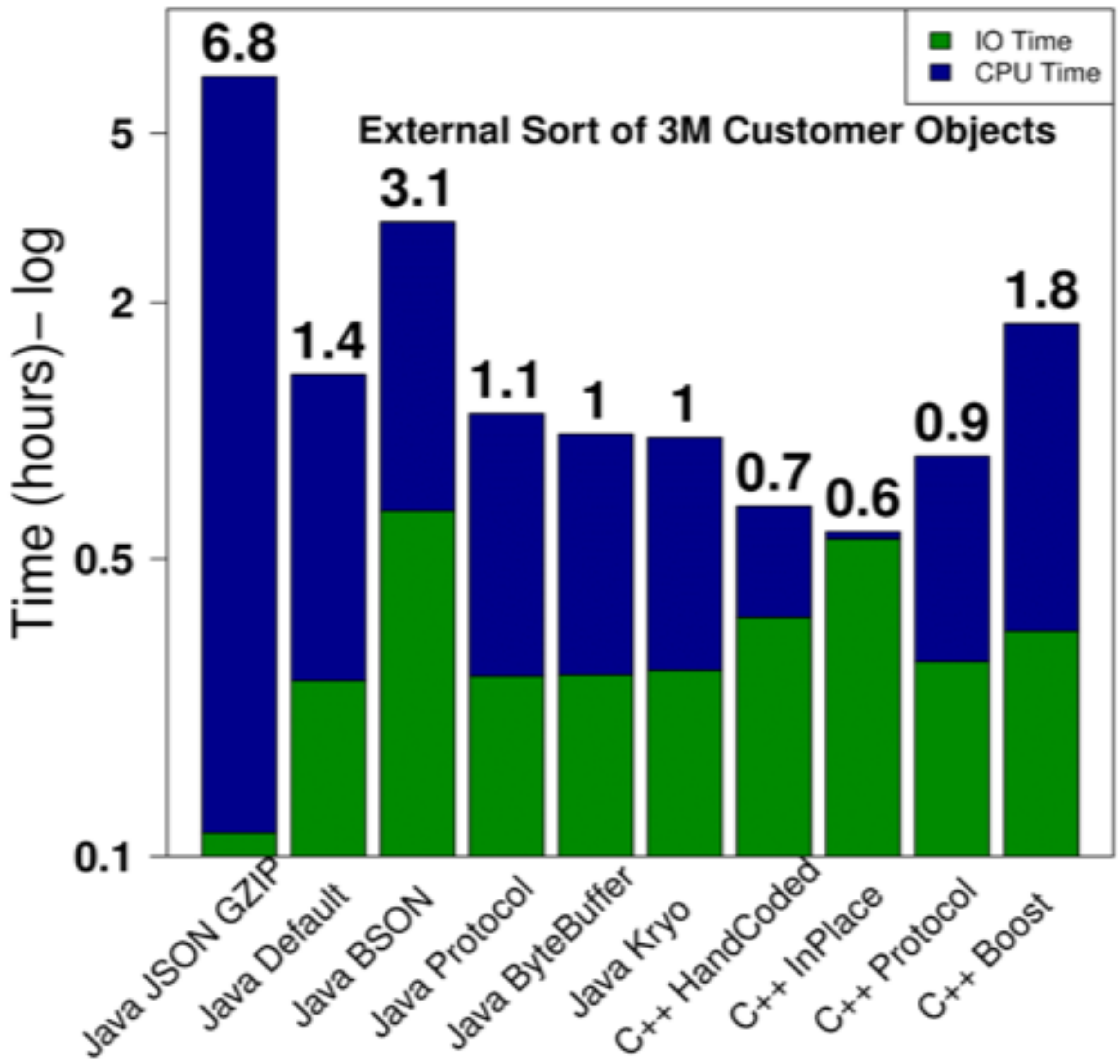
Compute machine has 30GB RAM.

# 2. External Sort




- The fastest C++ implementation (InPlace) is ~2x faster than fastest Java implementation (Kryo).

# 2. External Sort




- The fastest C++ implementation (InPlace) is ~2x faster than fastest Java implementation (Kryo).
- The faster C++ implementations are up to 5x-10x faster than document models.

# Tweets Dataset

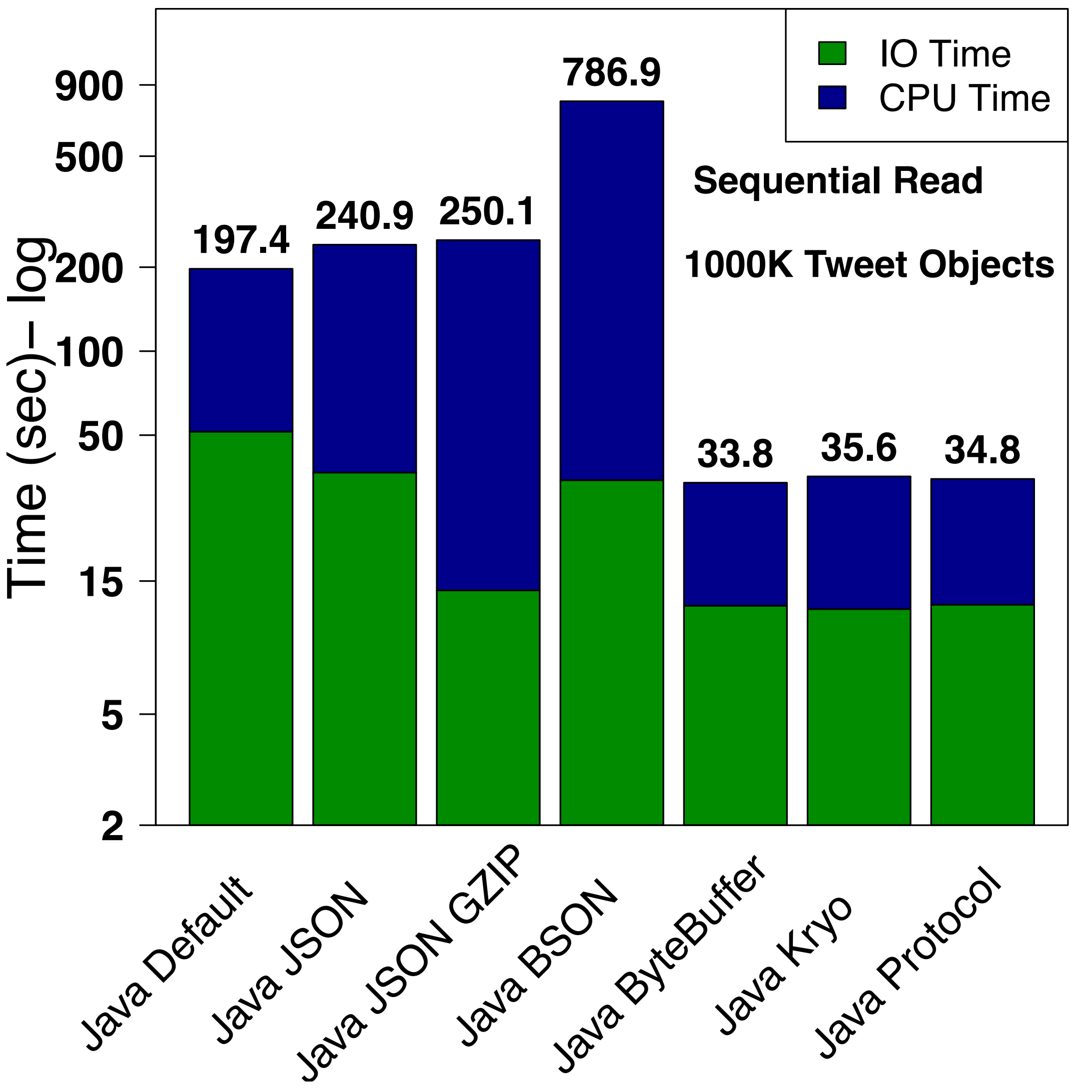
- Tweet objects are highly complex and nested graph objects.
- See JSON Format of it

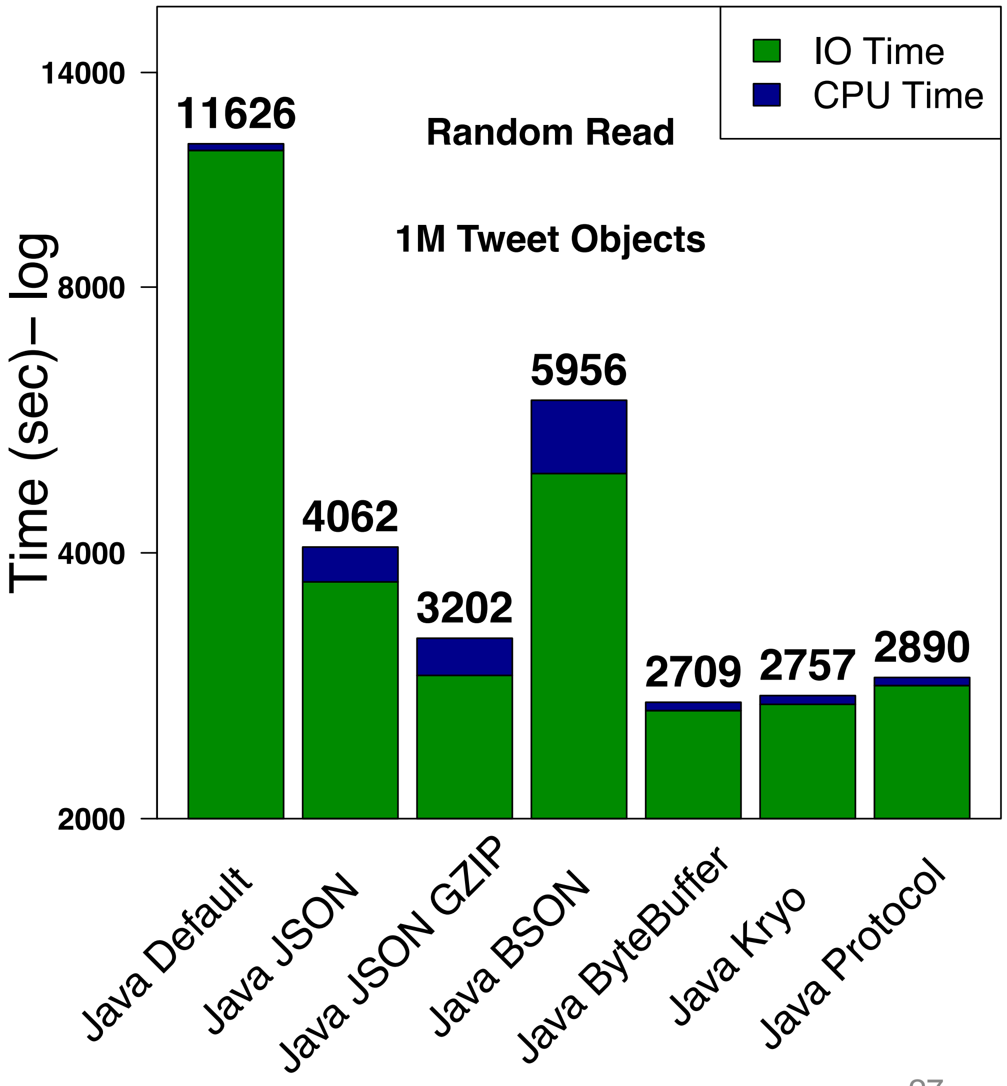
https://developer.twitter.com/en/docs/tweets/data-dictionary/overview/intro-to-tweet-json

Implementation as Java Class TweetStatus

```java
public class TweetStatus {

private User user;
private Coordinates coordinates;
private Place place;
private TweetStatus quotedStatus;
private TweetStatus retweetedStatus;

private List<HashtagEntity> hashtagEntities;
private List<MediaEntity> mediaEntities;
private List<URLEntity> urlEntities;
private List<UserMentionEntity> userMentionEntities;
private List<SymbolEntity> symbolEntities;

…
}
```

# Tweets Dataset – IO Experiments



# Conclusions

- The execution time in a memory managed environment (Java) is significantly higher than an un-managed environment (C++ on Linux).
  - A 1.5x-2x performance penalty even before system is designed.

- The costs are even higher for self-describing document formats like JSON.
  - Sorting JSON objects has 5x-10x penalty compared to C++ solutions.

- There is value in the “classical database” way of doing things – keeping the in-memory and on-disk representation the same.

# Use PlinyCompute

**A platform for high-performance distributed tool and library development.**

http://plinycompute.rice.edu/

https://github.com/riceplinygroup/plinycompute

Published in **SIGMOD2018**

Jia Zou, R. Matthew Barnett, Tania Lorido-Botran, Shangyu Luo, Carlos Monroy, Sourav Sikdar, Kia Teymourian, Binhang Yuan, and Chris Jermaine. **PlinyCompute: A Platform for High-Performance, Distributed, Data-Intensive Tool Development.** In SIGMOD '18.

DOI: https://doi.org/10.1145/3183713.3196933

# Thank You