An Experimental Comparison of Complex Object Implementations for Big Data Systems

Kia Teymourian

Joint work with Sourav Sikdar, Chris Jermaine
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| ...  |
|           |            |            |       |      | ...  |
|           |            |            |       |      | ...  |
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.

<table>
<thead>
<tr>
<th>Graph ID</th>
<th>Vertex ID</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graph ID</th>
<th>from</th>
<th>to</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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

Big Data System:
There are costs associated with -
- Objectification
- Serialization
- Garbage Collection

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

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

<table>
<thead>
<tr>
<th>Java</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Default</td>
<td>C++ Hand-Coded</td>
</tr>
<tr>
<td>Java ByteBuffer</td>
<td>C++ Boost</td>
</tr>
<tr>
<td>Java Kryo</td>
<td>C++ InPlace</td>
</tr>
</tbody>
</table>
• We borrow the idea from relational database.
  - On disk representation = In memory representation.
2. Self-Describing Documents

JSON + gzip

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

BSON

```
....Graph.œ....Vertices.œ....0...
....1....2....3....4....5....
Edges.œ.W....1....0....2....0.
....3....0....4....0....1....
....5....0....
```
3. Custom Data Models

Java 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
```

C++ Protocol Buffers

```
JAVA
class Graph {
  //...
}

C++
class Graph {
  //...
};
```
## Summary: Object Implementations

<table>
<thead>
<tr>
<th>Host-language objects</th>
<th>Self-Describing Documents</th>
<th>Custom Nested Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Default</td>
<td>JSON</td>
<td>Java Protocol Buffers</td>
</tr>
<tr>
<td>Java Kryo</td>
<td>BSON</td>
<td>C++ Protocol Buffers</td>
</tr>
<tr>
<td>Java ByteBuffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ Boost</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ HandCoded</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C++ InPlace</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Size (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java JSON + gzip</td>
<td>8508</td>
</tr>
<tr>
<td>Java Kryo</td>
<td>16176</td>
</tr>
<tr>
<td>Java Protocol Buffers</td>
<td>17305</td>
</tr>
<tr>
<td>C++ Protocol Buffers</td>
<td>17931</td>
</tr>
<tr>
<td>C++ HandCoded</td>
<td>19275</td>
</tr>
<tr>
<td>Java ByteBuffer</td>
<td>19478</td>
</tr>
<tr>
<td>Java Default</td>
<td>19556</td>
</tr>
<tr>
<td>C++ Boost</td>
<td>21004</td>
</tr>
<tr>
<td>C++ InPlace</td>
<td>25127</td>
</tr>
<tr>
<td>Java BSON</td>
<td>33879</td>
</tr>
</tbody>
</table>
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

```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

Sequential Read
- 1000K Tweet Objects

<p>|</p>
<table>
<thead>
<tr>
<th>Method</th>
<th>IO Time (sec)</th>
<th>CPU Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Default</td>
<td>197.4</td>
<td>33.8</td>
</tr>
<tr>
<td>Java JSON</td>
<td>240.9</td>
<td>35.6</td>
</tr>
<tr>
<td>Java BSON</td>
<td>250.1</td>
<td>34.8</td>
</tr>
<tr>
<td>Java ByteBuffer</td>
<td>786.9</td>
<td></td>
</tr>
<tr>
<td>Java Kryo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java Protocol</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Random Read
- 1M Tweet Objects

<p>|</p>
<table>
<thead>
<tr>
<th>Method</th>
<th>IO Time (sec)</th>
<th>CPU Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Default</td>
<td>11626</td>
<td></td>
</tr>
<tr>
<td>Java JSON</td>
<td>4062</td>
<td></td>
</tr>
<tr>
<td>Java JSON GZIP</td>
<td>3202</td>
<td></td>
</tr>
<tr>
<td>Java BSON</td>
<td>5956</td>
<td></td>
</tr>
<tr>
<td>Java ByteBuffer</td>
<td>2709</td>
<td></td>
</tr>
<tr>
<td>Java Kryo</td>
<td>2757</td>
<td></td>
</tr>
<tr>
<td>Java Protocol</td>
<td>2890</td>
<td></td>
</tr>
</tbody>
</table>
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

PlinyCompute: A Platform for High-Performance, Distributed, Data-Intensive Tool Development. In SIGMOD '18.
DOI: https://doi.org/10.1145/3183713.3196933
Thank You