

# An Experimental Study of Memory Management in Rust Programming for Big Data Processing

Shinsaku Okazaki

Boston University

May, 2020

# Motivation

Many of data-flow based Big Data Processing systems like Apache Spark and Flink have the following weaknesses!

- ▷ Java Virtual Machine (JVM)
  - ▷ Wasting CPU cycle
- ▷ Automated Memory Management
  - ▷ Generative Garbage Collection
  - ▷ Stop-The-World

Rust is a promising candidate for development of Big Data processing tools.

# Rust

Rust is a **”system language”** which has unique memory management concept.

- ▷ A system language does not have Garbage Collection.
- ▷ Rust ensures memory safety.
- ▷ It is easy to write safe multithreading code in Rust.
- ▷ Rust uses LLVM and provides high performance.

# Problem Description

Our goal is to determine the magnitude of performance change regarding the following aspects.

- ▷ Memory Management for High Complex Nested Objects
- ▷ Different Rust Memory Management Strategies
- ▷ Automated Reference Counting (Rc) vs. Reference
- ▷ Multithreading using Atomic Reference Counting (Arc)
- ▷ Arc vs. Deep Copy on Overall Performance of Big Data Processing

# Complex Object

In Big Data processing, data is represented by complex objects.

```
struct CustomerOwned {      struct CustomerBorrowed<'a> {
    key: i32,                key: &'a i32,
    total_purchase: f64,    total_purchase: &'a f64,
    zip_code: String,       zip_code: &'a String,
    order: OrderOwned       order: &'a OrderBorrowed<'a>
    .
    .
}                             .
                              .

struct CustomerSlice<'a> {   struct CustomerRc {
    key: &'a i32,            key: Rc<i32>,
    total_purchase: &'a f64, total_purchase: Rc<f64>,
    zip_code: &'a str,       zip_code: Rc<String>,
    order: &'a OrderSlice<'a> order: Rc<OrderRc>
    .
    .
}                             .
                              .
```

# Different Rust Memory Management Strategies

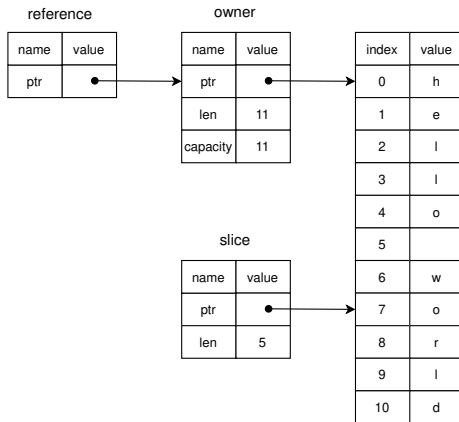
Each one has different memory representation.

- ▷ Owner
- ▷ Reference
- ▷ Slice

# Different Rust Memory Management Strategies

Each one has different memory representation.

- ▷ Owner
- ▷ Reference
- ▷ Slice



Each one may have different memory access time.

# Reference Counting

## Advantage

- ▷ Sharing ownership
- ▷ Dinamic memory de/allocation

## Disadvantage

- ▷ Need to check reference count
- ▷ Heap allocation



# Multithreading with Atomic Reference Counting

## Atomic Reference Counting

### Advantage

- ▷ Sharing ownership
- ▷ Dynamic memory de/allocation
- ▷ Sharing among multithreads

### Disadvantage

- ▷ Need to check reference count
- ▷ Heap allocation
- ▷ Atomic operation

# Impact of Memory Management on performance of Big Data Processing

## ▷ Merge-sort

- ▷ Many contiguous memory de/allocations

## ▷ Tree-aggregate

- ▷ Many intermediate HashMap like data structures

## ▷ K-Nearest-Neighbors (KNN)

- ▷ Document preprocessing

# Experiment 1: Accessing Object with Different Variable Type

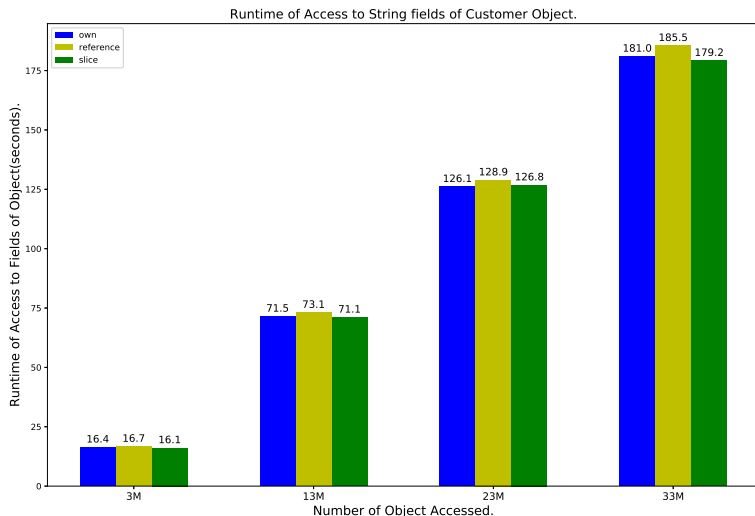
## Question

- ▷ How much are memory access times different among different memory management strategies used in complex objects?

## Evaluation

- ▷ **Custruct complex objects**
- ▷ With different memory management strategies: **Owner, Reference, Slice**
- ▷ CustomerOwned vs. CustomerBorrowed vs. CustomerSlice
- ▷ Measure time to **access to fields of complex objects**

# Experiment 1: Accessing Object with Different Variable Type



## Experiment 2: Assessment of different reference methods in Rust

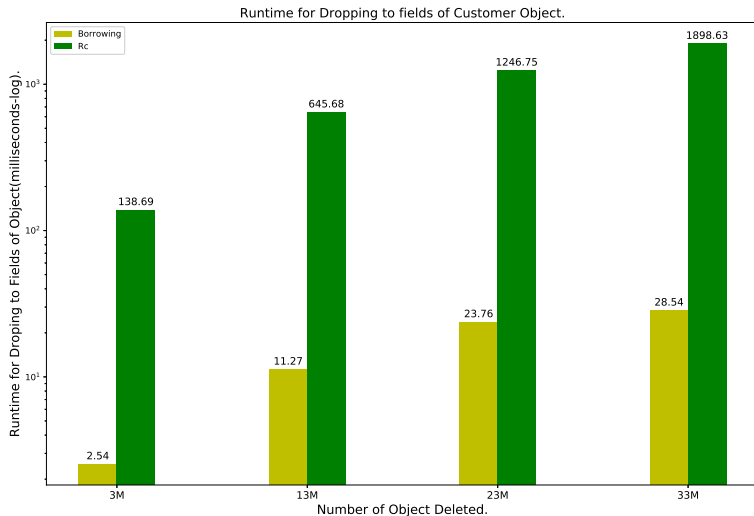
### Question

- ▷ How much does Reference Counting have impact on time for dropping its variable?

### Evaluation

- ▷ **Custruct complex objects**
- ▷ **Reference Counting** vs **reference**
- ▷ CustomerRc vs. CustomerBorrowed
- ▷ Measure time to **drop variables of complex objects**

## Experiment 2: Assessment of different reference methods in Rust



## Experiment 2: Assessment of different reference methods in Rust

### Result

- ▷ Dropping Reference Counting is about **60 times slower** than normal reference.

### Discussion

- ▷ Reference Counting needs to check reference count.
- ▷ In complex objects, overhead is significant.

# Experiment 3: Merge-sort

## Question

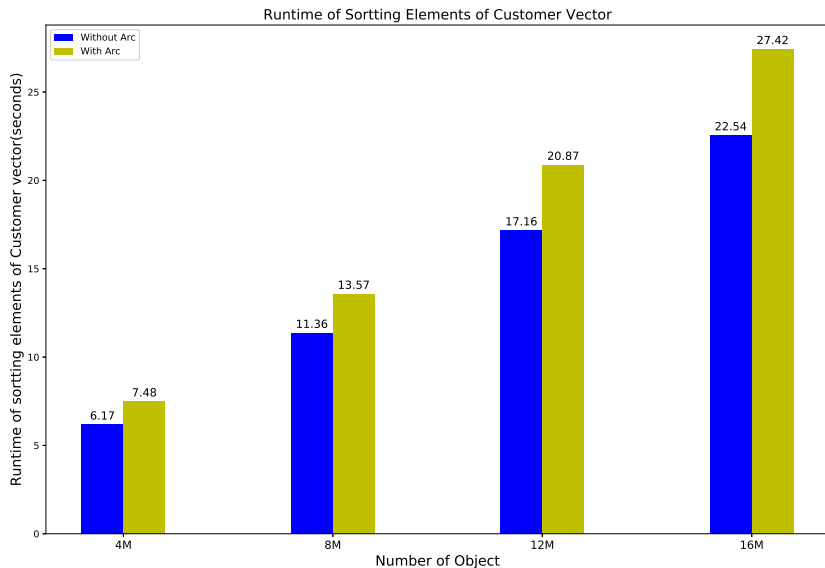
- ▷ How much does sharing set of data with Atomic Reference Counting have impact on performance of merge-sort algorithms?

## Evaluation

- ▷ Share **vector** of complex objects
- ▷ **Atomic Reference Counting (Arc)** vs **normal reference**
- ▷ Measure **runtime of merge-sort algorithms**



# Experiment 3: Merge-sort



## Experiment 3: Merge-sort

### Result

- ▷ Arc are about **21% slower** than normal reference.

### Discussion

- ▷ Atomic Reference Counting needs to check reference count.
- ▷ Atomic operations are more expensive than ordinal operations.

## Experiment 4: Tree-aggregate

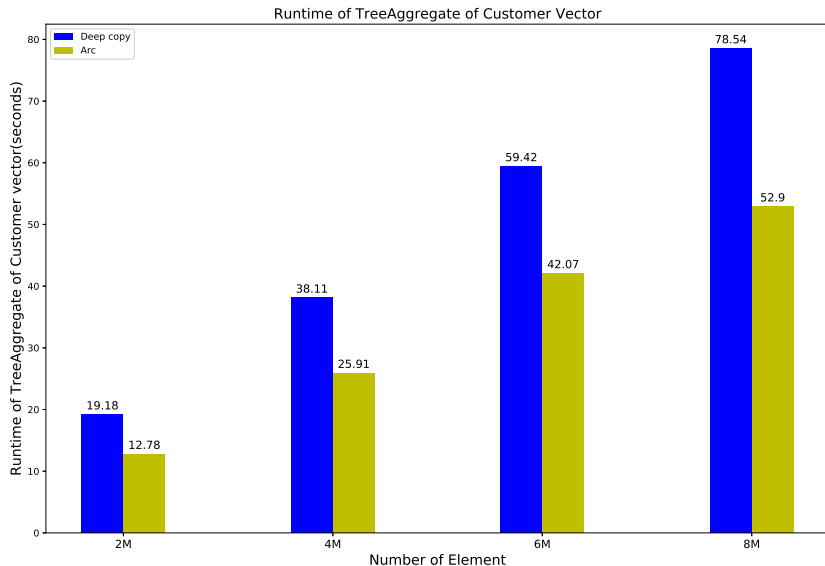
### Question

- ▷ How much are runtime differences between sharing elements of data with Arc and deep-copying elements of data?

### Evaluation

- ▷ Share **elements** of complex object
- ▷ **Atomic Reference Counting (ARC)** vs **Deep copy**
- ▷ Measure **runtime of Tree-aggregate algorithms**

## Experiment 4: Tree-aggregate



## Experiment 4: Tree-aggregate

### Result

- ▷ Deep-copies are from **40 to 50% slower** than Arc.

### Discussion

- ▷ Deep-copy of complex object is expensive.
- ▷ Performing deep-copy many times may lead significant overhead.

# Experiment 5: K-Nearest-Neighbors (KNN)

## Question

- ▷ What are better memory management strategies for common Machine Learning Algorithms?

## Evaluation

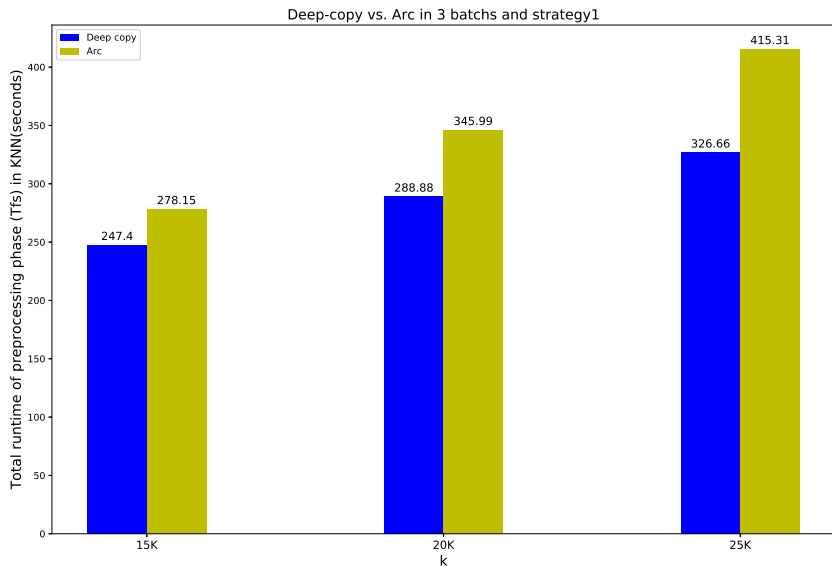
- ▷ Document classification on Wikipedia page dataset
  - ▷ train set:  $100 \times 10^3$  pages
  - ▷ test set:  $18 \times 10^3$  pages
- ▷ Preprocessing phase: calculating **Term-Frequencies (TFs)**
- ▷ String manipulation
- ▷ Measure runtime of **preprocessing time of KNN algorithms**

# Experiment 5: K-Nearest-Neighbors (KNN)

## Parameters

- ▷ Data Acquisition
  - ▷ **Atomic Reference Counting** (Arc)
  - ▷ **Deep-copy**
- ▷ Strategy
  - ▷ keep intermediate objects in memory until owner is changed
  - ▷ remove intermediate objects as soon as it is not needed
- ▷ Dimensions of feature matrices
  - ▷ 15K
  - ▷ 20K
  - ▷ 25K

# Experiment 5: K-Nearest-Neighbors





## Experiment 5: K-Nearest-Neighbors

### Result

- ▷ **Arc** is **at most 38% slower** than deep-copy.

### Discussion

- ▷ Deep-copying String is cheaper operation than Arc:
  - ▷ reference checking
  - ▷ atomic operation

# Conclusion

- ▷ There are **not notable differences** among times to access memory with different memory management strategies.
- ▷ When we drop complex objects constructed with Reference Counting, **additional computation for counting reference results in huge overhead.**
- ▷ **Sharing set of data with Arc is more expensive than reference.**
- ▷ **Deep-copying complex objects is more expensive than sharing them with Arc.**
- ▷ **Sharing Strings with Arc is faster than deep-copying them.**

# Findings

- ▷ Use normal reference rather than Reference Counting whenever it is possible.
- ▷ Trade-off between **runtime performance** and **lifetime tracking**.
- ▷ **Avoid using Arc** when we can use reference.
- ▷ Use Arc instead of deep-copy, when **complexity of objects is large**.
- ▷ Use deep-copy, when **complexity of objects is small**, like String.