Boston University

OpenBU **http://open.bu.edu**

Boston University Theses & Dissertations Boston University Theses & Dissertations

2020

An experimental study of memory management in Rust programming for big data processing

https://hdl.handle.net/2144/41789

Downloaded from DSpace Repository, DSpace Institution's institutional repository

BOSTON UNIVERSITY

METROPOLITAN COLLEGE

Thesis

AN EXPERIMENTAL STUDY OF MEMORY MANAGEMENT IN RUST PROGRAMMING FOR BIG DATA PROCESSING

by

SHINSAKU OKAZAKI

B.S., Seikei University, 2018

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2020

© 2020 by
SHINSAKU OKAZAKI
All rights reserved

Approved by

First Reader

Kia Teymourian, PhD
Assistant Professor of Computer Science

Second Reader

Eugene Pinsky, PhD
Associate Professor of the Practice

Third Reader

Reza Rawassizadeh, PhD
Assistant Professor of Computer Sicence

Acknowledgments

Foremost, I would like to show my appreciation to my advisor Prof. Kia Teymourian for his guidance, encouragement and support. His expertise and insights were essential for this research work. His passion for research and education motivates me and his experience is instrumental to the completion of the thesis.

I am especially thankful to Prof. Eugene Pinsky for accepting reviewing of this thesis. I also thank Prof. Reza Rawassizadeh for review and advice on this thesis.

Finally, I would like to express special thanks to my parents for supporting this excellent educational opportunity.

AN EXPERIMENTAL STUDY OF MEMORY MANAGEMENT IN RUST PROGRAMMING FOR BIG DATA PROCESSING

SHINSAKU OKAZAKI

ABSTRACT

Planning optimized memory management is critical for Big Data analysis tools to perform faster runtime and efficient use of computation resources. Modern Big Data analysis tools use application languages that abstract their memory management so that developers do not have to pay extreme attention to memory management strategies.

Many existing modern cloud-based data processing systems such as Hadoop, Spark or Flink use Java Virtual Machine (JVM) and take full advantage of features such as automated memory management in JVM including Garbage Collection (GC) which may lead to a significant overhead. Dataflow-based systems like Spark allow programmers to define complex objects in a host language like Java to manipulate and transfer tremendous amount of data.

System languages like C++ or Rust seem to be a better choice to develop systems for Big Data processing because they do not relay on JVM. By using a system language, a developer has full control on the memory management. We found Rust programming language to be a good candidate due to its ability to write memory-safe and fearless concurrent codes with its concept of memory ownership and borrowing. Rust programming language includes many possible strategies to optimize memory management for Big Data processing including a selection of different variable types, use of Reference Counting, and multithreading with Atomic Reference Counting.

In this thesis, we conducted an experimental study to assess how much these different memory management strategies differ regarding overall runtime performance. Our experiments focus on complex object manipulation and common Big Data processing patterns with various memory management. Our experimental results indicate a significant difference among these different memory strategies regarding data processing performance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Cluster computing tools for Big Data Analysis have been more important as magnitude and quality of data that we can obtain increases. Recently, almost all businesses stand on data, from web marketing analysis to factory automations. The leverage of data is ubiquitous, because there are many open source tools to analyze data and cloud computer infrastructure which can support computation for massive amount of data. The improvement of accessibility to these technologies has democratized data driven businesses by eliminating significant amount of initial investment.

However these computation resources do not come for free; we need to pay money to use these resources. Usually, users need to pay depending on usage of computational resources. If your process of data analysis is too long or needs to use number of clusters with high speck specification, the cost may end up significantly hight. To address these problems, the quality of analysis tool is critical. If the tool can optimize the runtime performance and usage of computational resources, the cost for running the businesses can become efficient.

Multiple cluster computing analysis tools have been developed, such as Hadoop MapReduce [2], Apache Spark [3], and Apache Flink [1] [9]. These tools have brought reliable and scalable ways to deal massive data. These has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing.

These tools are constructed on top of Java Virtual Machine (JVM). JVM abstracts hardware and memory management from the developer so that the development is fairly easy. In addition, Java or Scala compiled code is platform-independent, which can run on any machine with JVM. However, these advantages may be really critical weaknesses when it comes to processing big data.

JVM abstract away most detail regarding memory management from the system designer, including memory deallocation, reuse, and movement, as well as pointers, object serialization and deserialization. Since managing and utilizing memory is one of the most important factors determining Big Data systems' performance, reliance on a managed environment can mean an order-of-magnitude increase in CPU cost for some computations. This cost may be unacceptable for a high-performance tool development by an expert.

To overcome these problems, one can use programming languages with more control on hardware, system languages, for development of Big Data tools. For example, C++ is a general-purpose, statically typed, compiled programming language which supports multiple programming paradigm. It is also a system language which gives full control over hardware. There are several researches or projects [22] where developers and researchers implement Big Data tools with this language. These tools shows significantly better performances than those developed with application languages. Although the evidence of the advantage of building high speed computational tools with C++ has been discovered, the steep learning curve and difficulty of writing memory safe codes are barriers to technology diffusion.

Rust is a system language which gives the similar performance and control of hardware to C++ or C and safety of runtime. The memory-safety, and fearless concurrently in Rust programming make the language one of the ideal candidate for development of Big Data tools. Since the design of the language is different from any other programming languages, implementations that can be selected for algorithms can differ from existing ones. In this thesis, we focus on memory management strategy for Big Data processing algorithms in development with Rust.

1.2 Problem Description

Even though Rust can be a great candidate to develop Big Data processing tools, there are few studies for development on such tools with Rust programming.

Rust has various ways to manage memory. Rust has different variable types for values allocated in sequence of memory region. Each variables take different memory representation that can produce variation of operation time on the variable types.

In addition, Reference Counting takes an important role in Rust ownership concept. By using Reference Counting, a value is able to have multiple owners. This situation may happen quite often, when we want to acquire complex values from contiguous memory regions. Reference Counting has both advantage and disadvantage. Reference count can share data which might decrease unnecessary copy of data, but checking reference count might be a overhead.

Atomic Reference Counting is ubiquitous in Rust multithreading program. Atomic Reference Count also has similar features to Reference Counting. In addition, it can be used among different threads. This may lead additional overhead from atomic operation.

As we can see, we can choose various memory management strategies in Rust programming. Therefore, we assess following research question in this thesis.

- What are better memory management strategies for complex object processing to perform faster runtime performance.
- How much impact do different variable types in Rust have in order to algorithms' runtime performance?
- How much can algorithms runtime be improved or degraded, if we use Reference Count?
- What are better memory management strategies for faster Big Data processing in Rust multithreading?
- How can we improve runtime performance of common Big Data algorithms by Rust memory management?

To answer these question, we conduct 5 experiments.

Chapter 2

Related Work

2.1 Introduction

This chapter describes the related work and concepts to our research in this thesis. In Section 2.2, some topics in Operating Systems are described focusing on memory management in operating systems. Knowledge of operating systems helps us to understand how operating systems handle memory management and how other technologies takes advantage of them. Even though this thesis does not focus on technologies used for Linear Algebra Computation, there are some active researches in technologies in Linear Algebra Computation for Big Data processing. It is worth studying these concepts explained in Section 2.3. In Section 2.4, major Big Data processing tools and improvement of these tools are introduced. Then, Section 2.5 details technologies used for Apache Spark, which is one of the most popular tools for Big Data processing. In Section 2.7 and Section 2.8, the advantages of using System Language and Rust for such tools are discussed respectively. Finally, Section 2.9 describes how LLVM thrives computer language development.

2.2 Operating Systems

2.2.1 Memory and Process in Operating Systems

A process is a subsection of computation job. A process can work on a CPU core, we can divide process as well. Basically, each process does not share their memory. However, for multiprocessing, we could avoid this restriction. Processes can be represented as tree structures, because a process may create other child processes. Processes have 4 states, new, running, waiting, and ready. Processes are represented in process control block (PCB) with state type, process ID, registers, and so on. The scheduling for processes assigning to CPU core is implemented in queues containing PCB.

There are two main queues in this scheduler: ready queue and wait queue. The head of process in ready queue is selected for execution and once the process requested I/O request or production of child process, the running process will be stored n wait queue. Once the request that the process is waiting for its end, the waiting process will be pushed tail of ready queue.

Processes executing concurrently in the operating system may be either independent processes or cooperating processes executing in the system. A process is independent if it does not share data with any other processes. A process is cooperating if it can affect or be affected by the other processed executing in the system. In cooperating processes, there are two kinds, shared memory and message passing. In shared memory, it removes restriction of not interfering memory region. Message passing can be useful for distribution systems as well.

For a pair of processes to communicate through message, a socket is needed to be established. A socket is identified by an IP address concatenated with a port number. When two process communicate, each process will have socket. If another process of the same machine wants to communicate, we need a new socket to be established. The protocol used in the socket connection can be TPC and UDP.

2.2.2 Multi-threading and Parallelism

A thread is a basic unit of CPU utilization, so that a process can have multiple thread. Threads share mainly code and data. Multi-threading is increasingly popular as the multicore programming becomes in common, because we can run multiple thread on different core. Creating threads is much cheaper than creating processes and it shares resources so that we do not need additional methods to allow threads to communicate each other, such as sharing memory and message passing.

2.2.3 Memory management in Operating System

In computer storage hierarchy, the closest storage to CPU is register. It is built into each CPU core and accessible within one cycle of the CPU clock. However, the same cannot be said of main memory, which is accessed via a transaction on the memory bus. This takes many cycles of the CPU clock. The remedy is to add fast memory between the CPU and main memory, typically on the CPU chip for faster access. Such a cache plays a role for this.

For the layout of main memory, it must be ensured that each process has a separate memory space, including operation system. The base register and limit register, whose roles are lower bound of memory region and specific size of range respectively, can achieve that goal.

Usually, a program resides on a disk as binary executable file. To run, the program must be brought into memory and placed within the context of a process. The process is bound to corresponding parts of the physical memory. Starting processes binds programs to the memory address. There three stages: compile time, load time execution time. The source program is compiled by compiler producing object file. After the compilation, the object file is linked with other object file by linker creating executable file . Finally, the executable file will be loaded to run execute. At this run time dynamic library link can be done.

If the process will reside in memory at compile time, absolute code is generated. If this is unknown at the time, the binding will be done at load time. At this time, the compiler must generate relocatable code. Otherwise, the binding will be done at execution time.

A process does not interact with addresses of physical memory, instead virtual memory. The memory-management unit (MMU) takes roles to map logical address to physical address. OS needs to ensure that any of physical memory spaces of processes do not overlap. Since one process can be created and deleted and the corresponding memory space should be de/allocated, optimization for use of physical memory space is important; we need to allocate memory contiguously avoiding fragmentation.

There are several approaches to deal with this problems. However, we will focus on paging here, which is the most used method OS use to manage memory. A frame and page are a unit of Separated physical and virtual memory space in fixed size (4KB or 8KB) respectively. A process can use as many as pages and corresponding frames obtained by page table matching. This strategy does improve external fragmentation, but not for internal fragmentation. The smaller size of page has smaller fragmentation, but mapping from page to frame has overhead and also disk I/O is more efficient when the amount of data transfered is larger.

2.2.4 Demand Paging

A process can have multiple pages. However, loading entire executable code from secondary storage to memory is not necessarily needed to get jobs done. A strategy used in several operating systems is loading only the portion of programs that are needed, demand page.

In the storage, some pages currently used are in memory and the others are in secondary storage. The page table specifies whether pages are valid or invalid, which means are in memory or not. Access to a page marked invalid causes page fault and some steps to resolve the error will be required.

The first part of process of demand paging would be that we check an internal table to check whether the reference is valid or invalid. If the reference is valid, the process reads the content from the memory. Otherwise, we terminate the process and find the free frame in physical memory. Then, we schedule a secondary storage operation to read the desired page into newly allocated frame. When the storage finished reading the page, we modify the internal table to indicate that the page is now in memory. Finally, we restart the instruction that was interrupted.

However, there could be a case where the memory does not have any free frame. In this case, a victim frame that will be replaced with new coming frame should be selected. To perform this selection efficiently, a modify bit is tracked for each frame or page. The modify bit represents whether the page is modified since it is loaded from secondary storage. If the page or frame is modified, when we swap page we need to update the content in the secondary storage. However, if it is not modified, we can simply delete the frame and replace it with new frame.

2.3 Linear Algebra Computation

2.3.1 BLAS LAPACK

Basic Linear Algebra Subprograms (BLAS) [12] are standard building blocks for basic vector and matrix operations. There are 3 levels of operation. The level 1 BLAS performs scalar, vector and vector-vector operations, the level 2 BLAS performs matrix-vector operation, and the level 3 BLAS performs matrix-matrix operation.

LAPACK [7] is developed on BLAS and has advanced functionalities such as LU decomposition

and Singular Value Decomposition (SVD). Dense and banded matrices are handled, but not general sparse matrices. The initial motivation of development of LAPACK was to make the widely used EISPACK [17] and LINPACK [10] libraries run efficiently on shared-memory vector and parallel processors. LINPACK and EISPACK are inefficient because their memory access patterns disregard the multi-layered memory hierarchies of the machines, thereby spending too much time moving data. LAPACK addresses this problem by recognizing the algorithms to use block matrix operations, such as matrix multiplication, in the innermost loops. These block operations can be optimized for each architecture to account for the memory hierarchy, and so provide a portable way to achieve high efficiency on diverse modern machines. However, LAPACK requires that highly optimized block matrix operations be already implemented on each machine.

ARPACK [13] is also a collection of linear algebra subroutines which is designed to compute a few eigenvalues and corresponding eigenvectors from large scale matrix. ARPACK is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method(IRLM). The Arnoldi process only interacts with the matrix via matrix-vector multiplies. Therefore, this method can be applied to distributed matrix operations required in big data analysis.

2.3.2 Netlib-Java

Netlib-java [5] is a Java wrapper of BLAS, LAPACK, and ARPACK. Netlib-java chooses implementation of linear algebra depending on installation of the libraries. First, if we have installed machine optimized system libraries, such as Intel MKL and OpenBLAS, netlib-java will use these as the implementation to use. Next, it try to load netlib references which netlib-java use CBLAS and LAPCKE interface to perform BLAS and LAPACK native call. The last option is to use f2j which is intended to translate the BLAS and LAPACK libraries from their Fortran77 reference source code to Java class files, instead of calling native libraries by using Java Native Interface (JNI).

We can use JNI to call native libraries from Java. The JNI is a native programming interface which allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applica-

tions and libraries written in other programming languages.

2.3.3 Matrix Computation and Optimization in Apache Spark

Matrix operation is a fundamental part of machine learning. Apache Spark provides implementation for distributed and local matrix operation [20]. To translate single-node algorithms to run on a distributed cluster, Spark addresses separating matrix operations from vector operations and run matrix operations on the cluster, while keeping vector operations local to the driver.

Spark changes its behavior for matrix operations depending on the type of operations and shape of matrices. For example, Singular Value Decomposition (SVD) for a square matrix is performed in distributed cluster, but SVD for a tall and skinny matrix is on a driver node. This is because the matrix derived among the computation of SVD for tall and skinny matrix is usually small so that it can fit to single node.

Spark uses ARPACK to solve square SVD. ARPACK is a collection of Fortran77 designed to solve eigenvalue problems. ARPACK is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix A is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). ARPACK calculate matrix multiplication by performing matrix-vector multiplication. So we can distribute matrix-vector multiplies, and exploit the computational resources available in the entire cluster. The other method to distribute matrix operations is Spark TFOCS. Spark TFOCS supports several optimization methods.

To allow full use of hardware-specific linear algebraic operations on single node, Spark uses the BLAS (Basic Linear Algebra Systems) interface with relevant libraries for CPU and GPU acceleration. Native libraries can be used in Scala are ones with C BLAS interface or wrapper and called through the Java native interface implemented in Netlib-java library and wrapped by the Scala library called Breeze. Following is some of the implementation of BLAS.

- f2jblas - Java implementation of Fortran BLAS
- OpenBLAS - open source CPU-optimized C implementation of BLAS
- MKL - CPU-optimized C and Fortran implementation of BLAS by Intel

These have different implementation and they perform differently for the type of operation and matrices shape. In Sark, OpenBlas is the default method of choice. BLAS interface is made specifically for dense linear algebra. Then, there are few libraries that efficiently handle sparse matrix operations.

2.3.4 Memory Management of each Linear Algebra Library

The pure Java linear algebra library, such as La4j, EJML, and Apache Common Math, use normal GC performed by JVM to manage memory. This is because the implementation of these libraries are in purely Java.

Netlib-java, Jblas or other simple Java wrapper of BLAS, LAPACK, and ARPACK with Java Native Interface (JNI) use normal GC as well. This is because the native code deals with Java array by obtaining a reference to it. After the operation, the native method releases the reference to the Java array with or without returning new Java array or Java primitive type object.

ND4J [4] has two types of its own memory management methods, GC to pointer of off-heap NDArray, and MemoryWorkspaces. ND4J used off-heap memory to store NDArrats, to provide better performance while working with NDArrays from vative code such as BLAS and CUDA libraries. Off-heap means that the memory is allocated outside of the Java heap so that it is not managed by the JVM's GC. NDArray itself is not tracked by JVM, but its pointer is. The Java heap stores pointer to NDArray on off-heap. When a pointer is dereferenced, this pointer can be a target of JVM's GC and when it is collected, the corresponding NDArray will be deallocated. When using MemoryWorkspaces, NDArray lives only within specific workspace scope. When NDArray leaves the workspace scope, the memory is deallocated unless explicitly calling method to copy the NDArray out of the scope.

2.4 Hadoop MapReduce to Spark

Among Big Data mining tools that have been developed, the most notable ones are MapReduce and Spark. MapReduce is a cluster computing framework which supports locality-aware scheduling, fault tolerance, and load balancing. Spark is designed to improve performance for iterative jobs

keeping features of MapReduce.

MapReduce provides a programming model where the user creates acyclic data flow graphs to pass input data through a set of operations. This data flow programming model is useful for many of query applications. However, MapReduce framework struggles from two of main resent data mining jobs: iterative jobs and interactive analysis.

Iterative job is especially common in Machine Learning algorithms, such as learning a data model using Gradient Descent. In traditional MapReduce framework, each iteration can be expressed as single MapReduce job so that each job must reload the data from disk. This leads I/O overhead and deteriorates performance of iterative algorithms.

Interactive analysis is also an inevitable task in modern data science. A data scientist wants to perform exploratory analysis in interactive way. Nevertheless, MapReduce is designed in a way more stable for ad-hoc queries, so each analysis can be single MapReduce job. To perform multiple analyses to explore dataset, the data needs to be written to and reloaded from disk many times.

To overcome these limitations, Spark has been developed as a new cluster computing framework maintaining the innovative characteristics of MapReduce and improving its iterative and interactive jobs with in-memory data structure.

Some of the notable improvements are shown here[16]. For aggregation operations, map output selectivity, which is the ration of the map output size to the job input size, can be significantly reduced by using Spark. Spark uses a map side combiner, hash-based aggregation which is more efficient than sort-based aggregation used in MapReduce. For iterative operations, caching the input as Resilient Distributed Datasets (RDDs) can reduce CRU and disk I/O overheads for sequence iteration. This RDD caching takes a significant role to improve iterative job, because it is more efficient than other low-level caching approaches such as OS buffer caches, and HDFS caching. These caching strategy reduces disk I/O overhead, but CPU overhead, such as parsing text to objects.

2.5 Apache Spark

2.5.1 Resilient Distributed Datasets

Major methods in Spark are Resilient Distributed Datasets (RDDs), a data structure that abstracts distributed memory across different clusters. The immutable coarse-grained transformation, spark-scheduler with lazy-evaluation, and memory management with cacheing achieve computation with fault-tolerance, fast execution, and moderate control on memory efficiency [21].

A RDD is essentially a multi-layer Java data structure. A top RDD object references Java array, which in tern, references a set of tuple objects. The coarse-grained transformations and immutability requires a RDD to be deep-copied to produce a new RDD, but efficiently offers fault tolerance. The lost partitions of a RDD can be recomputed in parallel on different nodes rather than rolling back the whole program.

Spark-pipeline consists of sequence of transformations and actions over RDDs. A transformation produces a new RDD from a set of existing RDD. An action is a method that computes statistics from an RDD. Due to lazy-evaluation nature, transformations do not materialize the newly created RDD. Instead, RDD Lineages are created. Lineage is a graph among parent and child RDDs which represents logical execution plan. This enhance fault-tolerance and improve ability to optimize execution plan.

RDDs can be cached in memory for faster access by persist method. Developers can specify a storage level for a persisted RDD, in memory with serialized or deserialized, or on disk. Other than persisted RDD, Spark generates a lot of intermediate RDDs during execution. Since RDD is a Java object, they are managed by Garbage Collection (GC) in the JVM. However, persisted RDDs are never collected by GC. This GC might cause significant deterioration of performance of Spark, because GC shows heavy overhead when there are a number of objects.

2.5.2 Memory Management in Spark

Spark framework allocates multiple executors, JVMs, that run sequence of transformations and actions. As we describe the previous section, data in Spark is mainly stored as Java objects in memory, so that they are allocated on JVM heap and managed by JVM Garbage Collection. The

data may form three types [19] [18]: Cached data, Shuffled data, and Operator-generated data.

Spark can cache data in memory to reduce disk I/O. This Cached data usually long-lived Java objects and span multiple stages in Spark-pipeline. Spark allocates a logical storage space to store the cached data as shown in Figure. After aggregation, Spark generates Shuffled data. Shuffled data is usually long-lived, because it need to be kept in memory until the task ends. Spark allocates execution space to store Shuffled data. The storage space and execution space spans 60 % of JVM heap space in default. Operator-generated data is data generated by user-defined operations. Since Operator-generated data may or may not be used, after the operation finish, the data object can be both short-lived or long-lived objects. These are stored in user space allocated on default 40 % of JVM heap.

All data of these types on JVM heap is managed by JVM GC. GC check reference graphs of objects, mark whether the objects are used and deallocate memory space occupied by unused objects. There are three popular GCs: Parallel, CMS and G1. All of these methods track generation of objects based on the region of memory. The Java logical heap structure is shown in Figure 2·1. As the figure shows, Java logical heap can be separated into three main parts where store objects for each corresponding generation: permanent generation, young generation, and old generation. The region for permanent generation stores metadata required by JVM to describe class and method used in application which will be permanently lived on the region of memory. The overview of Java GC is shown in Figure 2·2. The outer box represents region for particular generation. The inner box and the number represents object and its age in GC.

The region for young generation mainly consists of two parts: Eden and Survivor space. First, Java objects are created in Eden space and promoted to Survivor space when survive from GC. After objects survive several GCs in Survivor space, they are finally promoted to old generation.

In region of old generation, JVM lunches multi-thread to perform GC. GC with multi-threading suffers from Stop-The-World (STW) pauses; GC may suspend application threads while performing object marking and deallocation. Different GC algorithms try to solve this problem with trade-off between GC frequency and memory utilization.

Because of the problem of STW and copying objects to different physical memory pages , JVM

GC cause huge overhead when number of objects is large. Therefore, GC becomes a severe issue in Big Data processing where it might produce significant number of object.

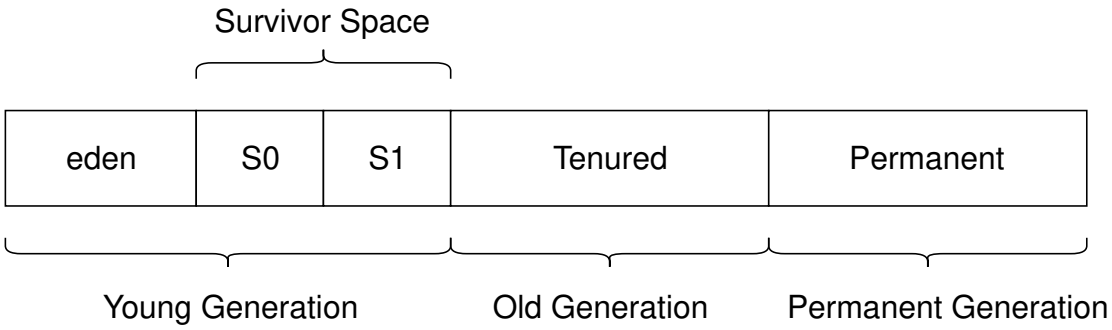

Figure 2·1: Java Heap Structure

2.6 Garbage Collection Tuning

There are many different ways that one can improve the performance of GC in Java. One of these is for example avoiding pointer-based data structures, such as HashMap and LinkedList. These objects have a "wrapper" object for each entry so that number of objects tends to be larger than when when an array is used.

Caching serialized objects in memory also reduces the number of objects and memory usage since the set of objects become a byte or binary array. Spark SQL applications use DataFrames[8], whose intermediate data are managed by an optimized memory manager named Tungsten. Tungsten stores the intermediate data in a serialized binary form and performs aggregation functions directly on he serialized objects. Therefore, the number of Java objects in memory is reduced and it reduces the DC frequency and object marking/sweeping.

In addition, developers can allocate data off-heap of JVM to avoid tracking by GC. Facade[15] proposed a compiler and runtime system to bound the number of in-memory data objects, through storing data in an off-heap region and manipulating the data with control interfaces.

Although these memory management solution for GC help developer improve performance of Spark applications, the effort to discover the best GC tuning afflicts developers.

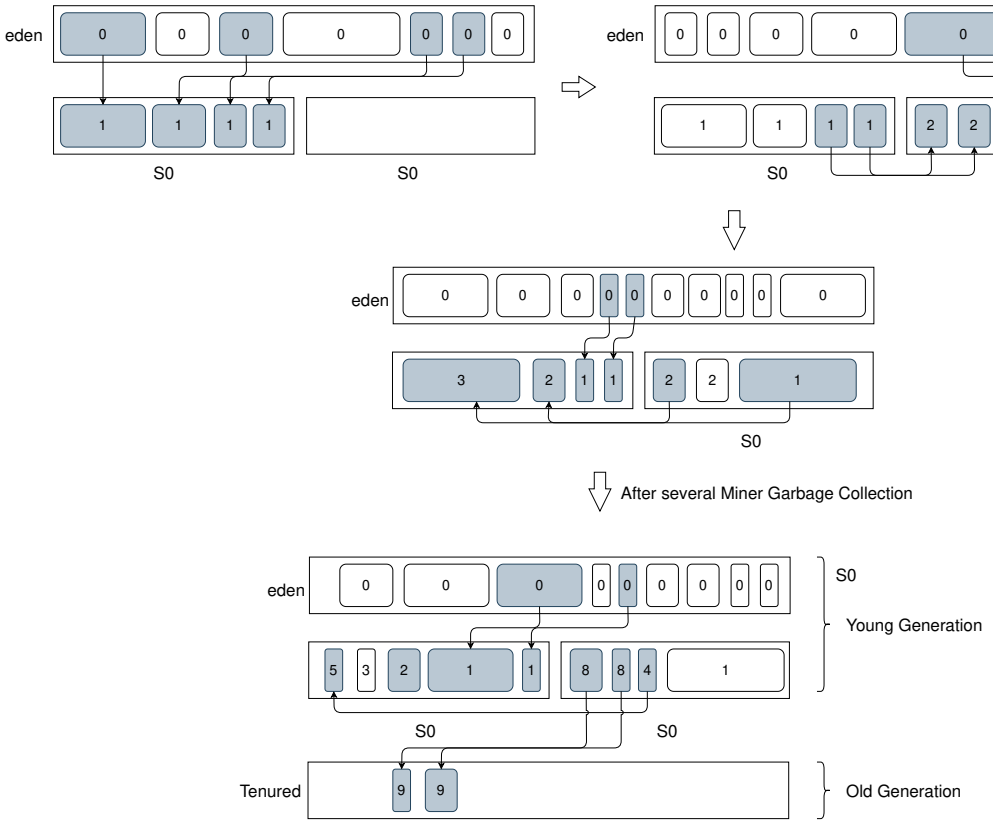

Figure 2·2: Java Garbage Collection

2.7 Application to System Language

Considering the overhead produced by Memory Management in JVM, use of system language for development of Big Data tools can be a better solution rather than application languages, such as Java and Scala. System languages, such as C and C++, are languages that give developers total control over the hardware and de/allocation of memory without GC. These features enable programs written in system languages to optimize performance taking full advantage of hardware.

For example, one can build Big Data tools with C++. C++ is one of the most popular system languages which has Object-oriented features. C++ has functions which provide control over memory to developers. In another words, it is responsible for managing memory properly and safely. The functions, malloc() and free(), take roll for memory allocation and deallocation in respectively. The manual memory de/allocation may cause several problems and require developers attention to the problems with significant effort for debugging and testing. Here, we explain two of the most

common problems regarding memory management in existing system languages.

Dangling pointer or reference is a pointer or reference pointing to object that no longer exists. The situation of dangling pointer happens because of deallocation of memory without modification of value of the pointer. If the memory region is reallocated for other objects and the dangling pointer tries to access the original object, the unpredictable behavior may result.

Memory leaks occur when memory is allocated and no longer referenced so that the object in the memory location cannot be reached and released. This is result of dereferencing object without deallocation. Memory leaks consume more memory than necessary by making unreachable location.

Some solutions are established to address these problems. Actually, GC is a high-level solution that guarantees memory safety. C++ has a different solution called Resource Acquisition is Initialization (RAII). In RAII, objects can live within the scope where they are created. The memory is released when the object goes out of scope. This solution is more predictable and deterministic than GC. However, it is problematic when we need the object out of the scope, returning a value from a function. There are several ways to go around this problems, such as smart pointers, copy constructors, and move semantics. Nevertheless, these non-orthogonal concepts may disorganize code and lead to error prone implementation.

2.8 Rust Memory Management

Rust is a system programming language which provides memory safety without runtime checking like GC and necessity of explicit memory de/allocation. To ensure memory safety, Rust provides restrictive coding patterns and checks lifetime of value and memory safety at compile-time. The restrictive patterns also enables a developer to write fearless concurrent code that is free of data races. Main concepts of Memory Management in Rust are ownership, move, and borrowing.

2.8.1 Ownership

In ownership feature or Rust, each value has a variable called owner. This owner has information about the value, such as location in memory, length and capacity of the value. For example, the ob-

ject representation of Vec<i32> is shown in Figure 2·3. The upper boxes represent owner variable in stack frame. The lower boxes represent contiguous memory allocated to store i32. Its capacity is specified 10, but 7 values of i32 are stored. Therefore, there are still spaces to store 3 values of i32 without reallocation of memory. This owner can live on the scope associated with its lifetime. When the owner is dropped, the value will be dropped too. This feature is similar to how RAII in C++ works. However, acquisition of owner out of the scope where it was constructed is available in Rust with the concept of move.

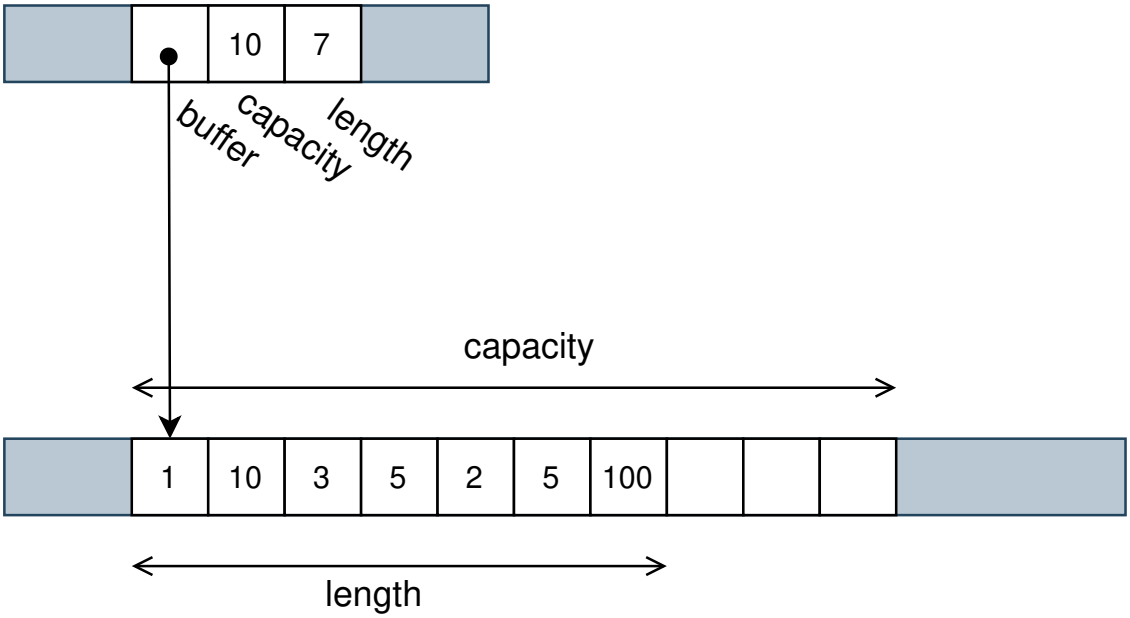

Figure 2·3: Representation of Rust Vec<i32>

2.8.2 Move

In Rust, for most types of operations like assigning a value to a variable, passing it to a function, or returning it from a function does not copy the value: it moves it. With each move, a value can be transferred from one owner to another. The previous variable does not have ownership of the value; it is moved to a newly assigned variable. To understand how this assignment implementation is unique from other programing languages, Java, C++, and Rust code example of assigning list or vector of strings are shown.

Below a few lines of code are initialization of list of string and reassigning it to other variables.

```
List<String >s = new ArrayList<>(
                    Arrays.asList("lemon", "orange", "apple"))
List<String> t = s
List<String> u = s
```

Figure 2·4 represents the memory allocated after a Java ArrayList of strings is initialized and assigned to variable called s. After assigning s to t and to u, the memory representation become Figure 2·5. The assignments are simply setting pointers to the ArrayList object and increment reference count.

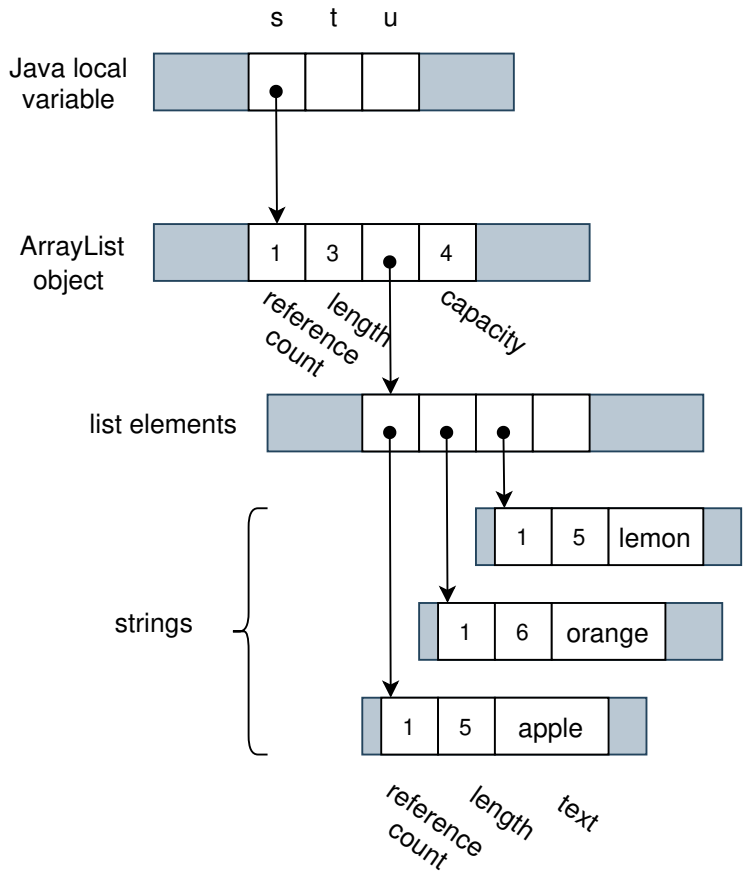

Figure 2·4: Representation of Java ArrayList of String

Now, the analogous C++ code is shown below.

```
vector<string> s = {"lemon", "orange", "apple"};
```

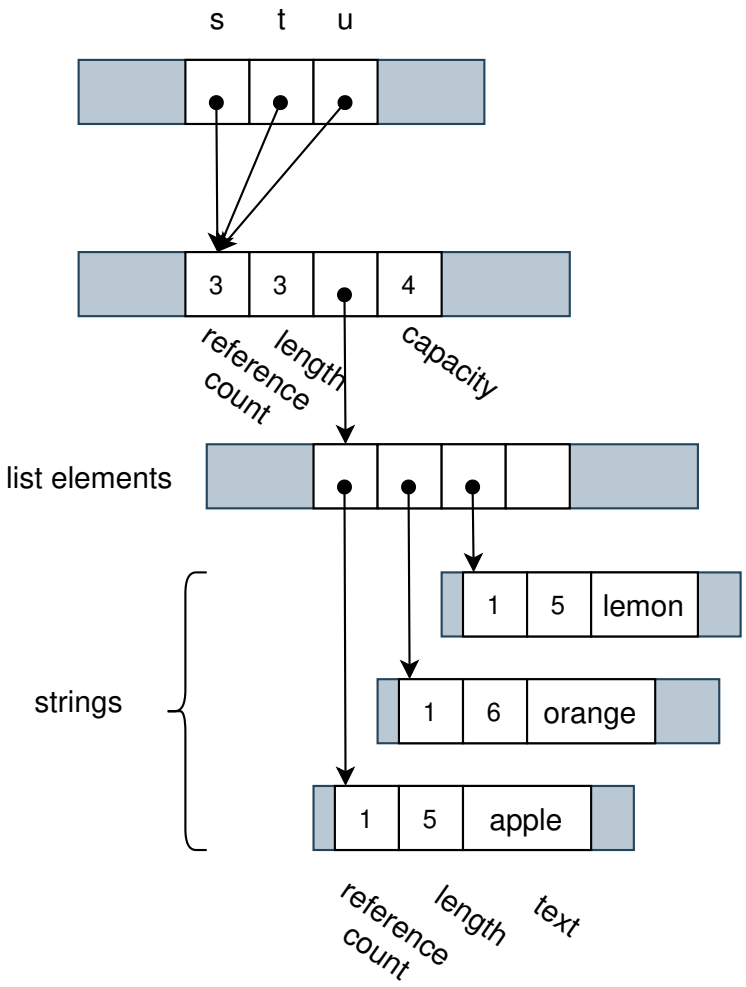

Figure 2·5: Representation of Java ArrayList of String after assignment to another variable

```
vector<string> t = s;
vector<string> u = s;
```

Figure 2·6 shows the memory allocated when the vector is initialized. Figure 2·7 is a representation of after the assignments of s to t and to u. In C++, assigning value of vector to other variables involves allocating memory for new vector and copying the contents of the original vector to newly allocated one.

In Rust code, the code is like below,

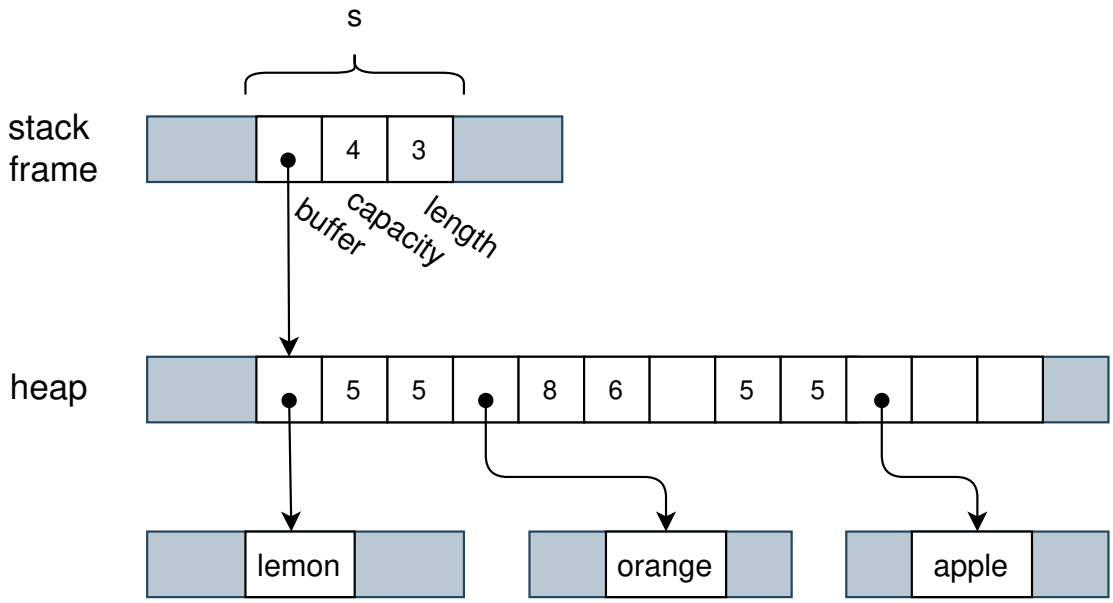

Figure 2·6: Representation of C++ vector of string

```
let s = vec!["lemon".to_string,
             "orange".to_string,
             "apple".to_string];
let t = s;
let u = s;
```

The representation of the original Vec of String in Rust is the almost same in C++ vector of string. Figure 2·8 however shows different behavior when Rust assigns s to t. The value is moved to

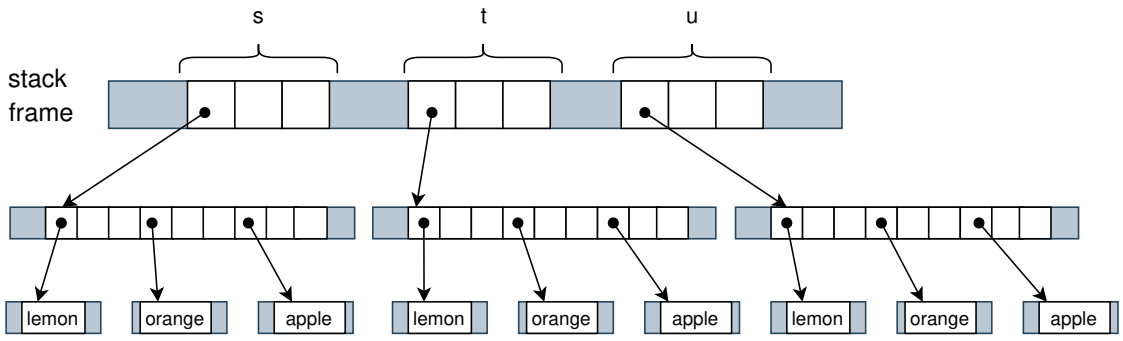

Figure 2·7: Representation of C++ vector of string after assignment to another variable

t from s so that the source variable s is uninitialized. The Rust code actually throws compile error, because we are assigning uninitialized variable at the last line.

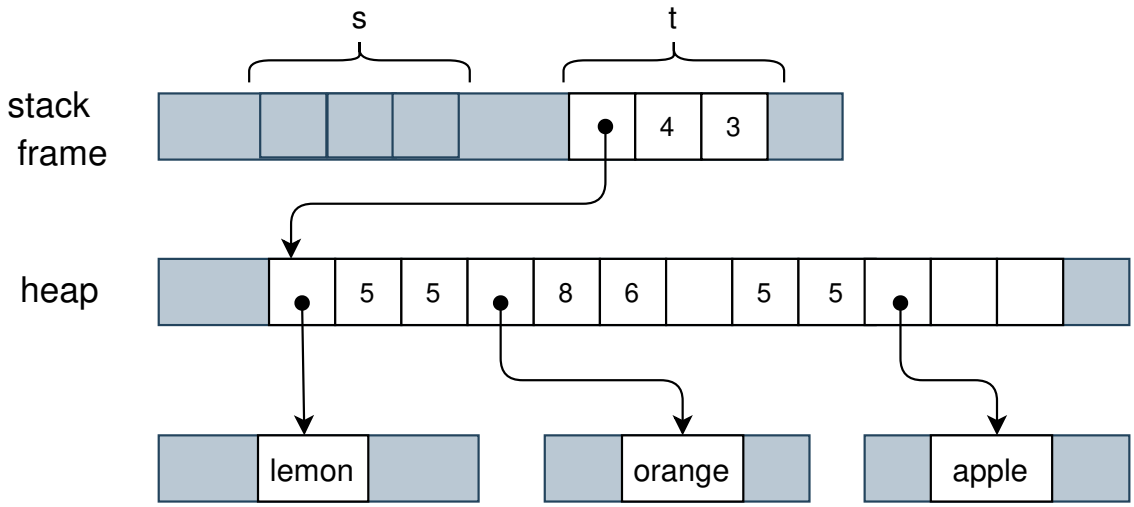

Figure 2·8: Representation of Rust Vec<String> after assignment to another variable

2.8.3 Borrowing

Borrowing lets code use a value temporarily without affecting its ownership so that it reduces unnecessary movement of ownership. One use case is when value is used in function and needed to be passed to the argument. If the argument takes ownership and the function does not return the value, the ownership of value goes out of scope and the memory is deallocated. One can pass reference of the value to the argument instead of owner. The reference goes out of scope, but ownership remains the same.

2.9 LLVM

LLVM (Low Level Virtual Machine) [11] is an umbrella project which contains components of compilation of programming languages. Existing compilers have tightly coupled functionalities so that it is not possible to embed them into other applications. However, the abstract framework of LLVM decouples the functionalities into pieces and the pieces of functionalities can be reused.

In structure of a compiler, there are three main components; frontend, optimizer, and backend.

In frontend, a developer designs the interface of source programming language in a way where it can be optimized by an optimizer. Then, backend takes optimized code and produce the native machine code.

This separation of functionalities gives reusability to parts of the compiler. Establishing new programming language requires new frontend, but the existing optimizer and back end can be reused. This speeds up processes of developing new programming languages.

The reusability of compiler leads to support multiple programming languages so that broader set of programmers are involved in the development. For open source projects like Rust, This ends up larger community of potential contributors to the development.

Since implementation of frontend can be independent on optimizer and backend, development of programming languages becomes easy for people without skills required to implement optimizer and backFend. This thrives new language development.

LLVM has a component called LLVM Intermediate Representation (IR), which places itself across frontend to optimizer. IR is designed to host mid-level analyses and transformations that you find in optimizer section of a compiler. High-level language has many common structures and functionalities, so most of all high-level program languages can be represented with IR. Once source code is represented with IR, optimizer can easily find pattern and optimize it in faster time. IR is useful in terms of frontend. This is because developer of language frontend need to know only how the IR works and use the framework to develop a language.

Thanks to emerging of LLVM, many new programming languages and compilers have been developed. Rust compiler is also developed based on LLVM.

2.10 Summary

In this chapter, we cover some of main concepts to understand our research. The topics of Operating Systems and Linear Algebra Computation are not directly relevant to our research, but it is worth covering these topics for deep analysis of experimental result. The advantage and disadvantage of modern Big Data processing tools are discussed and some solutions are introduced. Finally, how Rust can be a good candidate for development of Big Data processing tools and details of its

technologies are described. In the next chapter, we discuss main concepts needed to understand theoretical assumptions of our experiments.

Chapter 3

Conceptual Design of Experiments

3.1 Introduction

This chapter discusses the concepts studied in our experiments. Section 3.2 explains different variable types we can use in Rust programming. In Section 3.3, we explain the advantages and disadvantages of Reference Count in Rust. Multithread programming in Rust is studied in Section 3.4. Two algorithms are discussed used in Big Data processing and examined in our experiments: Tree-aggregate in Section 3.5 and K-Nearest-Neighbors in Section 3.6. Finally, Section 3.7 shows complex object representations used in our experiments.

3.2 Types of Variables

In Rust, there are three variable types: owner, reference, and slice (only for sequence of values). A developer is sometimes forced to use specific variable types. For example, some of methods are only implemented to specific variable types. However, one can select any variable type for operation in most cases.

These variables have different memory representation shown in Figure 3·1. The owner has a pointer pointing to the memory address of sequence values, length of the values, and capacity allocated to store additional values. Reference and Slice are variables borrowing value owned by other variable. The reference is a pointer that points to the owner. The slice is a pointer that points to memory addresses of sequence values. It has values such as length of sequence values stored in the memory. Since they have different memory representations, an assumption is that it takes different time to access the contents of the memory among these pointer types. We examine this by constructing complex objects whose fields are these variable types. The details are explained in Section 4.3.

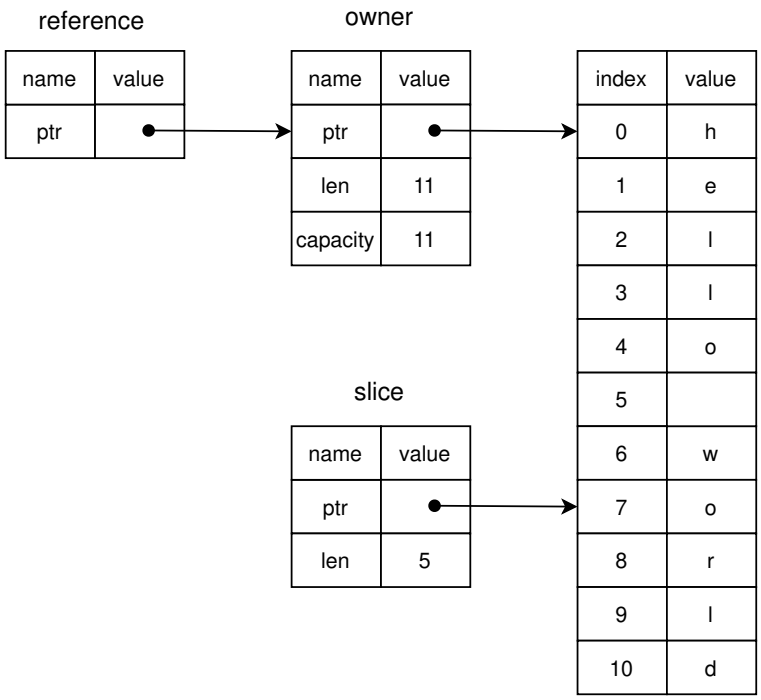

Figure 3·1: Memory Representation of Owner, Reference, and Slice Type

3.3 Reference Count

References are useful to avoid movement of ownership. However, one needs to track its lifetime and explicitly includes it in code, because Rust compiler cannot infer it. This can be another encumbrance. We can instead acquire multiple owners to single value by using Reference Counting (Rc). By leveraging Rc, a value can be shared like what borrowing plays the role in Rust programming.

The difference is that Rc checks the number of owners pointing to the actual data and makes sure the data is not deleted until all the owners are dereferenced. Using Rc is sometimes the preferred approach for developers especially when lifetime planning is extremely difficult. However, the possible problems regarding Rc are the cost for tracking the number of references and allocating memory on heaps instead of stacks. Having these assumption, an experiment is conducted to examine difference of runtime performance of dropping reference and Rc. This is explained in section 3.2.

3.4 Multithread

In Rust programming, writing concurrent code is relatively easy. The care Rust takes with reference, mutability, and lifetimes is valuable enough in single-threaded programs, but it also is in concurrent programming. Rust has tools to write concurrent code, such as threads, locks, atomic reference. One can implement various concurrent codes for the same purpose with different memory management strategies. The most ubiquitous tool used in Rust concurrent code is Atomic Reference Counting (Arc).

Arc is a simple interface that allows threads to share data. Arc allows multiple variable to have ownerships of a particular value similarly to Rc, but also supports atomic feature enabling the ownerships exist in different threads. In many situation where developers write a multithreading code, the deletion of Arc happens a significant amount of times. Similarly to Rc, our assumption is that deletion of Arc also has overhead when we compare to normal references. To assess runtime performances of algorithm with Arc vs normal references, we implement merge-sort algorithm in two different ways.

3.5 Tree-aggregate

Finally, we implement some of common algorithms in Big Data processing. One of them is tree-aggregate.

Tree-aggregate is a communication patten heavily used for Machine Learning algorithms in Spark (MLlib [14]). The topology of aggregation patterns in Apache Spark are shown in Figure 3·2. In the traditional aggregation functions in Spark, results of aggregation in all executor clusters are sent to the driver. That is why this operation suffers from the CPU cost in merging partial results and the network bandwidth limit. Tree-aggregate is a communication pattern which overcomes these problems by breaking aggregate operation in multi-level represented like tree structure.

Spark generates intermediate objects from RDDs. In Tree-aggregation, aggregated HashMap like data structure is created in each thread or node. When it aggregates objects in RDD, copies of objects should be performed to construct intermediate aggregated data structure. In another possible way that might be implemented, one can use references to the objects to perform aggregation instead

of copying the values themselves. In Rust, we can clone or get Arc (Rc if in single thread) of objects to implement these operations.

Since Tree-aggregation algorithms generate and delete a lot of intermediate data structures, how the data structures are constructed and how they are deallocated is an important concern in memory management in this algorithm. In our experiment, tree-aggregation algorithms are examined in multi-threading. The detail is described in section 3.2.

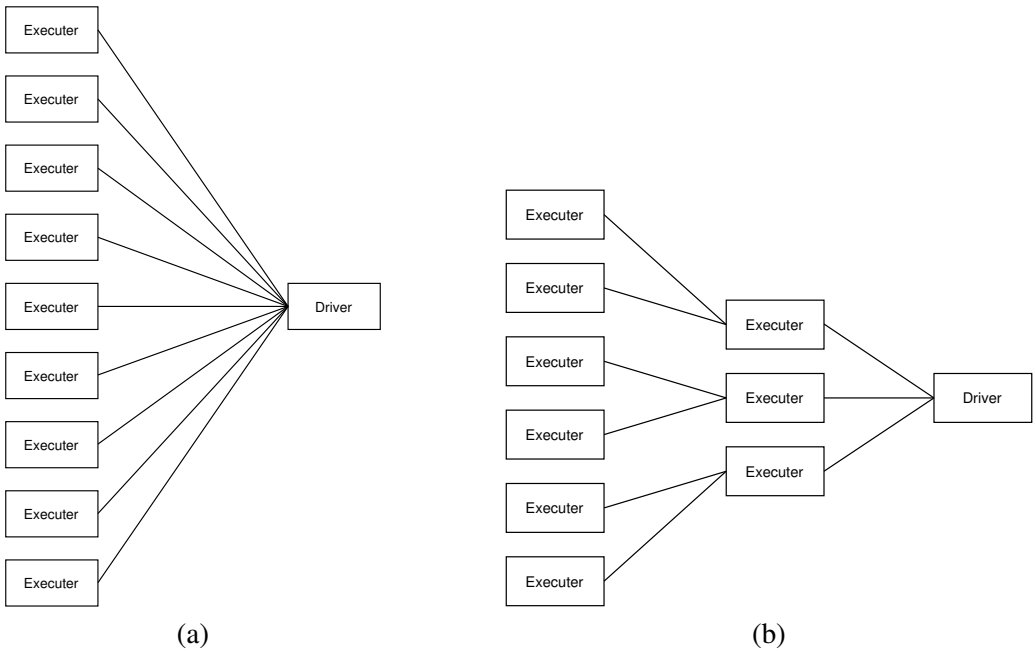

Figure 3·2: Representation of aggregation strategies in Apache Spark: (a) Traditional Aggregation, (b) Tree Aggregation

3.6 K-Nearest-Neighbors

The other algorithm common in Big Data processing is K-nearest-neighbors (KNN). KNN is a traditional Machine Learning algorithm which classifies targets into categories. In training KNN, it simply stores all available training data without calculation. At prediction phase, targets are given and similarity measures are calculated. Based on these similarities, the algorithm selects K (user-defined number) training observations similar to each target. Then, it checks corresponding categories of the K observations to determine predicted categories.

In brute force algorithm, KNN calculates similarity measures for all combinations among training and testing observations. If training data has N observations and test data has M observations, KNN needs to calculate $N \times M$ similarity measures. Sort or binary heaps can be used to select the K most similar training observations.

In our experiment, KNN algorithms performs document classification and implemented in multithread. There are three phases in our algorithm: preprocessing, query, and combine phase. The preprocessing phase, the algorithm calculates Term-frequencies (Tfs) to generate numeric feature vectors and matrices. In the query phase, similarity measures are calculated and K nearest neighbors are selected. For similarity measure, our choice is cosine similarity(3.1). In the combine phase, results of query phase are gathered and combined from each threads.

$$Cos(\vec{x},\vec{t}) = \frac{\sum_{i=0}^{n}(x_i t_i)}{\sqrt{\sum_{i=0}^{n} x_i^2}\sqrt{\sum_{i=0}^{n} t_i^2}} \tag{3.1}$$

Even though the preprocessing phase is not specific process for KNN algorithm, it is common in algorithms used in Natural Language Processing. Therefore, better memory management strategy should be applied in this preprocessing phase. This preprocessing generates many intermediate data structures and copies of String elements are used in these data structure again and again. We can again either clone or get Arc of String.

Our KNN algorithms are implemented with batch processing. In query phase, we control batch size to examine how size of objects allocated simultaneously in memory has impact to algorithm's runtime performance. The detailed implementation is explained in section 3.2.

3.7 Complex Objects

To conduct experiments for the above concepts, we use 4 types of complex objects: CustomerOwned, CustomerBorrowed, CustomerSlice, and CustomerRc. These objects contain other type of objects: OrderOwned, OrderBorrowed, OrderSlice, and OrderRc. The representation of these objects are shown in Figure 3·3 and Figure 3·4. Three Customer objects have 15 fields: 3 fields for i32, 3 fields for f64, 8 fields for String, and 1 field for Order object. All fields of CustomerOwned and OrderOwned are owned by the object. On the other hand, fields of CustomerBorrowed, Customer-

Slice, OrderBorrowed, and OrderSlice are borrowed. References of slices of values are used as the fields and owners of actual values are stored differently in source Vec. CustomerRc acquires Rc of values used for its fields from the source Vec.

```
struct CustomerOwned {
    key: i32,
    age: i32,
    num_purchase: i32,
    total_purchase: f64,
    duration_spent: f64,
    duration_since: f64,
    zip_code: String,
    address: String,
    country: String,
    state: String,
    first_name: String,
    last_name: String,
    province: String,
    comment: String,
    order: OrderOwned
}
```

(a)

```
struct CustomerBorrowed<'a> {
    key: &'a i32,
    age: &'a i32,
    num_purchase: &'a i32,
    total_purchase: &'a f64,
    duration_spent: &'a f64,
    duration_since: &'a f64,
    zip_code: &'a String,
    address: &'a String,
    country: &'a String,
    state: &'a String,
    first_name: &'a String,
    last_name: &'a String,
    province: &'a String,
    comment: &'a String,
    order: &'a OrderBorrowed<'a>
}
```

(b)

```
struct CustomerSlice<'a> {
    key: &'a i32,
    age: &'a i32,
    num_purchase: &'a i32,
    total_purchase: &'a f64,
    duration_spent: &'a f64,
    duration_since: &'a f64,
    zip_code: &'a str,
    address: &'a str,
    country: &'a str,
    state: &'a str,
    first_name: &'a str,
    last_name: &'a str,
    province: &'a str,
    comment: &'a str,
    order: &'a OrderSlice<'a>
}
```

(c)

```
struct CustomerRc {
    key: Rc<i32>,
    age: Rc<i32>,
    num_purchase: Rc<i32>,
    total_purchase: Rc<f64>,
    duration_spent: Rc<f64>,
    duration_since: Rc<f64>,
    zip_code: Rc<String>,
    address: Rc<String>,
    country: Rc<String>,
    state: Rc<String>,
    first_name: Rc<String>,
    last_name: Rc<String>,
    province: Rc<String>,
    comment: Rc<String>,
    order: Rc<OrderRc>
}
```

(d)

Figure 3·3: Representation of Customer objects Whose fields are different variable type: (a) CustomerOwned struct whose fields are all owned (b) CustomerBorrowed struct whose fields are borrowed with reference (c) CustomerSlice struct whose fields are borrowed with slice for sequence value, otherwise reference (d) CustomerRc struct whose fields are reference counting

3.8 Summary

We discussed some concepts to understand goals of our experiments. Theoretical discussions are noted in this chapter, such as assumptions where behavior of different variable types differ from each other, the impact of use of Reference Counting and Atomic Reference Counting, and the best practice to implement algorithms used in Big Data processing. In Chapter4, we discuss detailed settings of our experiments, their results and analysis.

```
struct OrderOwned {
    order_id: i32,
    num_items: i32,
    payment: f64,
    order_time: f64,
    title: String,
    comment: String
}
```

(a)

```
struct OrderBorrowed<'a> {
    order_id: &'a i32,
    num_items: &'a i32,
    payment: &'a f64,
    order_time: &'a f64,
    title: &'a String,
    comment: &'a String
}
```

(b)

```
struct OrderSlice<'a> {
    order_id: &'a i32,
    num_items: &'a i32,
    payment: &'a f64,
    order_time: &'a f64,
    title: &'a str,
    comment: &'a str
}
```

(c)

```
struct OrderRc {
    order_id: Rc<i32>,
    num_items: Rc<i32>,
    payment: Rc<f64>,
    order_time: Rc<f64>,
    title: Rc<String>,
    comment: Rc<String>
}
```

(d)

Figure 3·4: Representation of Order objects Whose fields are different variable type: (a) OrderOwned struct whose fields are all owned (b) OrderBorrowed struct whose fields are borrowed with reference (c) OrderSlice struct whose fields are borrowed with slice for sequence value, otherwise reference (d) OrderRc struct whose fields are reference counting

Chapter 4

Evaluation Result

4.1 Introduction

In this chapter, detailed experiment settings, executions and results are discussed. Section 4.2 details hardware configurations and datasets we use. There are 5 experiments conducted. In Section 4.3, we discuss the experiment where we assess runtime to access object's fields with different variable types. Section 4.4 shows details to examine how behaviors of normal reference and Reference Count are different. Section 4.5 describes our Merge-sort experiment to examine behavior of Atomic Reference Counting. Two algorithms are used in Big Data processing and examined in our experiments: Tree-aggregate in Section 3.5 and K-Nearest-Neighbors in Section 3.6.

4.2 Experimental Set and Detail

We have used two different datasets. One is a real-world data set, Wikipedia page data set, and another is a syntactic randomized generated data set based on Complex objects described in Section 3.7. We use one machine with particular specifications.

4.2.1 Wikipedia Data Sets

Wikipedia page data sets are used to perform document classification with KNN. We have separated the data set into training and test data set, 10^5 pages are used for training data set, and 18724 pages are used for target.

4.2.2 Experimental Details

All experiments are run on VM instances on Google Cloud Platform, n1-standard-8 which has 8 vCPU, 30 GB RAM, and 10 GB Standard persistent disk. In this thesis, we present the result of

runtime as the average of 5 separate runs for each experiment.

4.3 Experiment 1: Accessing Objects with Different Variable Types

This experiment is conducted to provide answers to the following two questions. One is how different variable types impact runtime performance. The other is how initialization of Vec size impacts runtime performance. In this experiment, we focus on owner, reference, and slice as variables of sequence values. Since these variables have different memory representation, there might be differences among time for access to actual values of each variables.

To evaluate this assumption, we use the three types of complex objects: CustomerOwned, CustomerBorrowed and CustomerSlice. At first, we generate source Vecs for all fields, Vecs which contain all elements used for corresponding fields of objects. For example, all of i32 elements used for key field in 1 million Customer object are stored in Vec<i32> with 1 million i32 elements. Later, these i32 elements are moved to be owned or borrowed by the objects' fields.

Next, 3, 13, 23, 33 million Customer objects are created and stored in Vec. When a Customer Vec is created, whether size of Vec is initialized is controlled. Finally, serialization of Customer object is performed as an operation forcing the program to access all of fields in the object. This serialization is performed for each Customer objects in the Vec. We measure total runtime to serialize all of Customer objects stored in Vec.

4.3.1 Result

The result is shown in Figure 4·1 and Figure 4·2. Figure 4·1 is a comparison of the runtime performance among different Customer object types with Vec size initialization. Figure 4·2 is a comparison in the same set of experiment except Vec size is not initialized. The blue, yellow, and green bars represent runtime of access to fields of CustomerOwned, CustomerBorrowed, and CustomerSlice objects respectively.

Whether Vec size is initialized or not, differences of runtime for accessing objects are not remarkable among different object types. The memory usage for algorithms with 33 million of Customer objects are about 26G bytes for any type of variable.

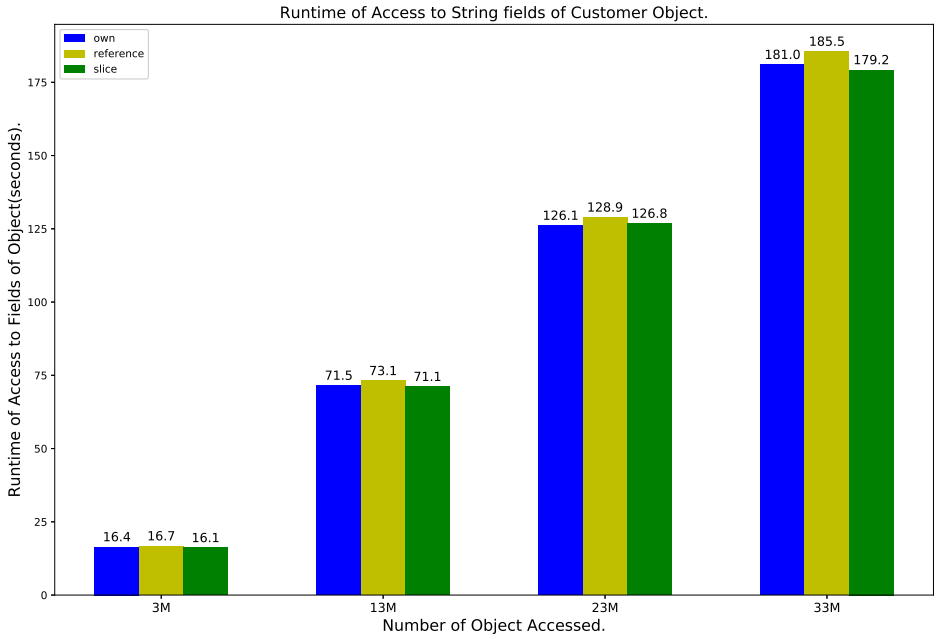

Figure 4·1: Runtime of Access to Different Pointer Types with Vec Size Initialization

4.3.2 Discussion

Difference of variable types does not have huge impact to runtime of accessing to actual value. Even though owner, reference , and slice have different memory representations, the access time to its value is close to each other. As shown in Figure 3·1, the representations of owner and slice are almost identical except slice does not have capacity for values. Reference is pointer pointing to owner, so it has an additional step to access actual value. However, the result shows this additional step does not have huge impact for runtime to access memory region of the value.

4.4 Experiment 2: Assessment of different reference methods in Rust

In this experiment, CustomerBorrowed and CustomerRc are used to see difference of dropping time among reference and Rc. In the CustomerRc and OrderRc struct, all fields take Rc (Rc<T>). Similarly to the experiment in the last section, sets of integer, float, and String vector are created

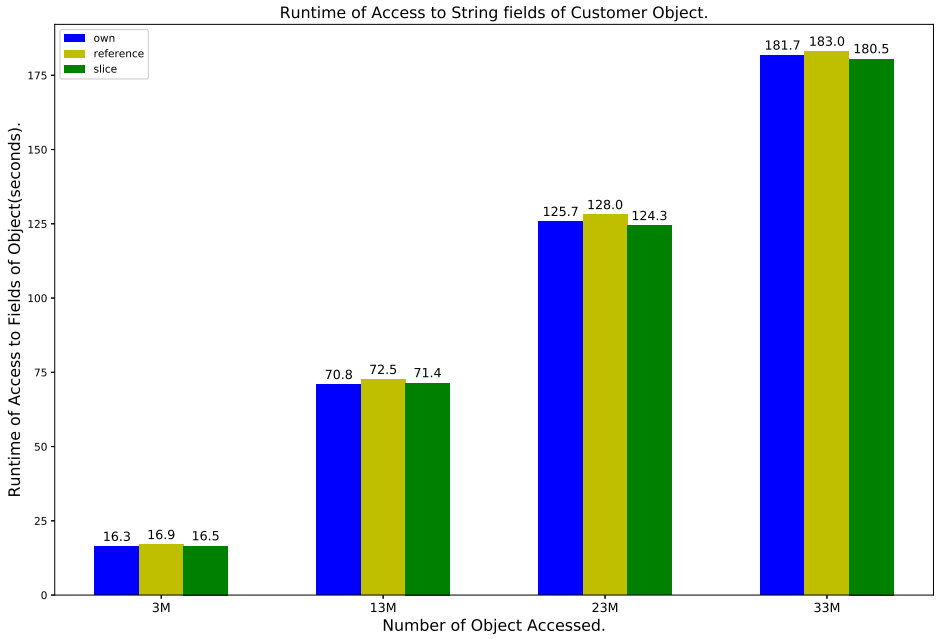

Figure 4·2: Runtime of Access to Different Pointer Types without Vec Size Initialization

and their elements are borrowed or reference counted to create CustomerBorrowed or CustomerRc objects. The dropping of objects deletes references or Rcs used for fields of the objects. However, it does not deallocate values to which they are pointing. Therefore, the evaluated runtime of dropping objects only consists of dropping time of reference or Rc, but deallocation time. We generated 10, 20, 30, and 40 million CustomerBorrowed and CustomerRc objects and performed drop one by one.

4.4.1 Result

Figure4·3 shows comparison of runtime dropping CustomerBorrowed and CustomerRc objects. The result shows significant difference of dropping time among the two objects; deletion of CustomerRc is much slower than CustomerBorrowed. The runtime of dropping CustomerBorrowed is about 60 times faster than dropping CustomerRc. The memory usage for algorithms with 40 million of Customer objects are about 26G bytes for both CustomerRc and CustomerBorrowed.

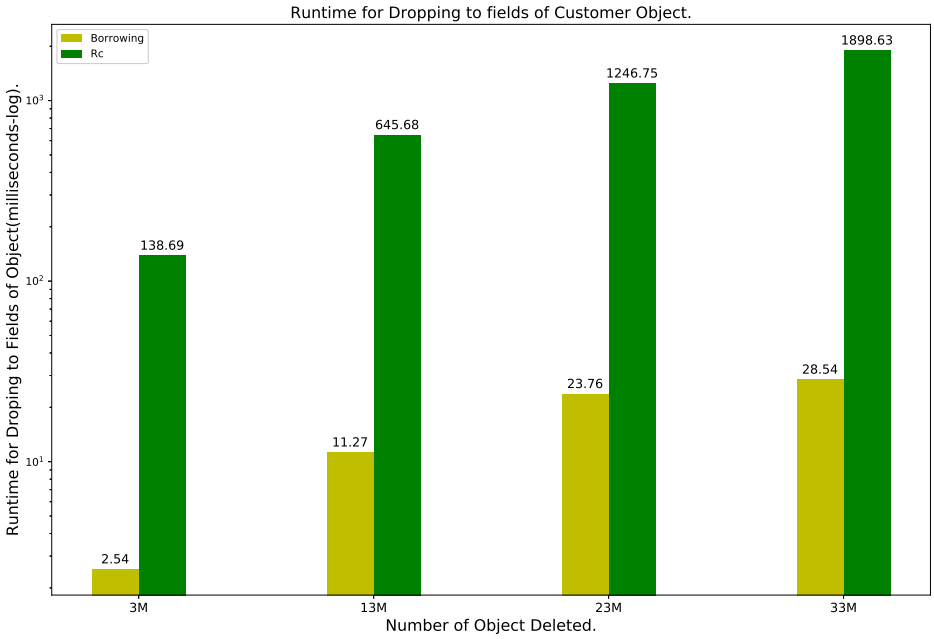

Figure 4·3: Runtime for dropping Customer Object

4.4.2 Discussion

In this experiment, an assessment is conducted to verify whether there is difference between behavior of reference and Rc. The reason why dropping Rc is much slower than dropping reference is that Rc requires runtime overhead to check some states of the variable, but when to drop reference is already determined at compile time. When dropping Rc, Rc has to check the number of variables pointing to the actual content and decide whether to deallocate the memory or not. However, memory management and lifetime strategy of reference is already determined at compile time. This determination of memory management strategy at compile increases runtime performance of dropping complex object constructed with reference type variable. This may say that we should use reference whenever high performance computation is critical.

However, dealing with reference is sometimes cumbersome. Tracking lifetime of reference can be done easily in simple situation. But, if we have complex objects constructed with fields of reference, the lifetime tracking becomes extremely difficult. For example, constructing nested

objects with reference fields requires a developer to plan memory management with many lifetime symbols. Using reference counting eliminates the developer's responsibility to specify lifetime of variables. This may ease and speed up development process, and increase understandability of codes.

Even though we have stack allocated values, such as i32 and f64, in Rc in our experiment, one should avoid wrapping stack allocated values in Rc. Wrapping value in Rc allocates heap memory so that allocating Rc<i32> or Rc<f64> unnecessarily uses space of heap. Additionally, stack allocated values are usually easy to be copied. Therefore, developer does not have to even use reference; one can just copy the value. Copying value in Rust is to copy the original value and to assign the copy to new owner variable.

4.5 Experiment 3: Merge-sort

Sorting algorithm like merge-sort is a very ubiquitous algorithm in any computation. Especially in Big Data processing, merge-sort algorithms de/allocate contiguous memory for partitions of sorted vector multiple times. We are interested in memory de/allocation patterns in merge-sort algorithm, because it generates many intermediate data structures and deletes them many times.

As explained in Experiment 2 in Section 4.4, deletion of Rc has significant overhead than normal reference. By learning this result, our assumption here is that deletion of Arc also has overhead when we compare to normal reference. In many situations where a developer writes a multithreading code, Arc is used and the deletion happens multiple times. To assess runtime overhead of algorithms with Arc, we implement merge-sort algorithm in two different ways. One is sharing source vector with Arc. The other is passing reference of source vector to child thread.

Our merge-sort algorithms are implemented with recursion. For each call of recursive function, Arc or slice of the source vector needs to be passed and deleted when the function returns value. These merge-sort algorithms trigger large number of Arc or reference deletion proportional to the number of call recursive function. Merge-sort algorithm can be separated in three phases: splitting phase, copying phase, and merging phase.

The splitting phase is merely acquiring index of range. At this phase, multiple threads are

generated and Arc or reference of source vector are passed by calling recursive function. Copying phase occurs in the base case of the recursive call. The element in the source vector in a certain index is deep-copied into newly allocated vector. At merge phase, merge function receives two sorted independent vectors and merges them into a single new vector.

We use scope method from crossbeam crate to perform multithread programming. Scoped thread can have reference to value from its parent thread by ensuring children threads are joined before their parent thread returns value. By using scoped thread, we can implement two merge-sort algorithms in an identical way except whether the function receives Arc or reference of source vector. The representations of source vectors for each algorithm are shown in Figure 4·4. The elements of source vector are CustomerOwned objects. We generate source vectors in sizes 4, 8, 12 and 16 million. Finally, merge-sort is performed based on value of key field. The figure shows the result for runtime performance of merge-sort algorithms on difference sizes of vectors.

```
// Source vector for algorithm with Arc.
arr: Arc<VecDeque<T>>

// Source vector for algorithm with reference (slice).
arr: &[T]
```

Figure 4·4: Representation of Source Vector

4.5.1 Result

Figure 4·4 shows the runtime performance of our merge-sort algorithm with specified Vec sizes. The blue and yellow bar charts represent the runtime performance of merge-sort algorithm using reference and Arc respectively. The result says that algorithms with Arc is about 21% slower than algorithms with reference. The memory usage for algorithms with 16 million Customer objects are about 26G bytes.

4.5.2 Discussion

The reason why merge-sort algorithms with Arc is much slower than one with reference is the same for the reason why dropping Rc shows overhead compared to dropping reference. Arc has to check

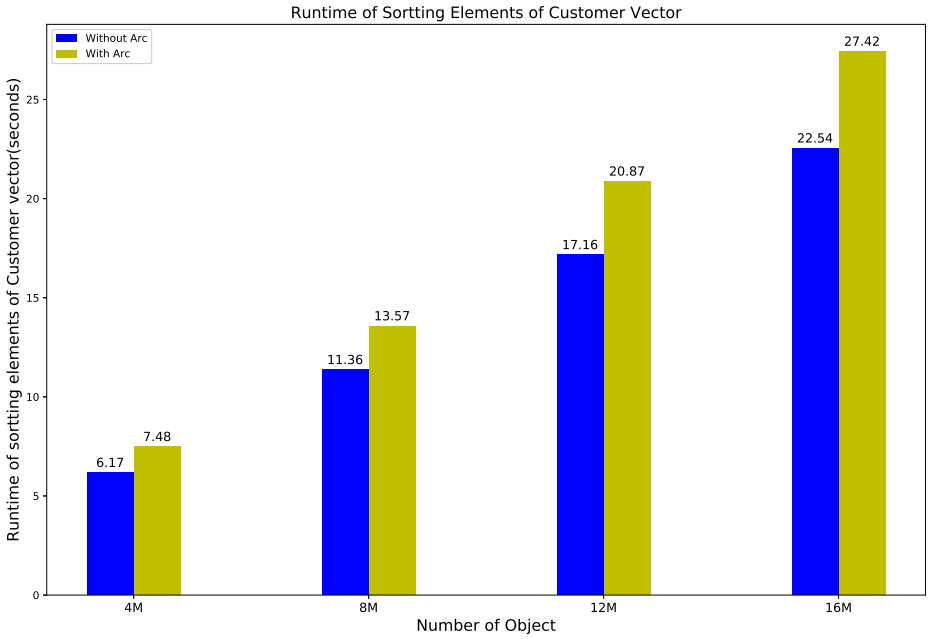

Figure 4·5: Runtime of Sorting Elements of Customer Vector

the number of variable pointing to the actual content and decide deallocate the memory or not.

In addition, Arc uses atomic operations for reference counting. Atomic operations bring thread-safety, but they are more expensive than ordinary memory accesses. Therefore, when sharing reference counting between threads is not required, using Rc is the recommended way [6].

In situations like our experiment, normal reference can be used instead of Arc to share data between threads. This solution results in better runtime performance in our experiment. Therefore, one should use reference to share data among different threads whenever it is possible.

4.6 Experiment 4: Tree-aggregation

In our experiment, tree-aggregation algorithms are examined in multi-threading. This experiment is to evaluate the impact of having Arc (Atomic Reference Counting) as elements of vectors. In Big Data mining tools, such as Spark, it generates intermediate objects from the original source vector. In tree-aggregation, aggregated HashMap like data structures is created in each step or node.

Acquisition of elements in the source vector is required to perform this aggregation. There are several ways.

One way is to deep-copy elements of the vector. This solution allocates newly created objects by deep-copy. Aggregation is performed on copied objects, they are stored in the data structure and sent to next node. Deep-copying generates duplicates of objects in vectors and aggregated data structure. This can lead to memory intensive moments when we need memory space for the additional duplicated objects.

The other way is to get reference to the elements. Since an original source vector is deallocated after a local aggregation, simple references to elements do not live long enough and allow the aggregation result to be sent to next node. Instead of simple borrowing, we need owners in the aggregation result. Reference Counting (Rc) in Rust is a way to have multiple owners to a value. Since our experiment is implemented in multithreading, Arc (Arc) is used instead of Rc. With Arc, multiple ownership pointers can be possessed by different variables across multiple threads. Therefore, a value is not deallocated until all owners to it are dropped. This does not require extra memory allocation, because only acquisition of new ownership to the value is needed. However, as explained in the last section deletion of Arc type checks whether the value is still owned by other variables. This checking may be an overhead in algorithms that generate a lot of intermediate data structures, because deletion of the data structures occurs frequently.

Two algorithms are implemented using the above two methods and their runtime performance is evaluated . We perform aggregation to CustomerOwned based on last_name field. Before tree-aggregation algorithms are run, partitions of Vec<CustomerOwned> or of Vec<Arc<CustomerOwned>> are created, serialized, and stored in disk. A tree-aggregate algorithm has main three phases: loading, aggregating, and combining phase. At loading phase the algorithm generates threads. In each node, it loads serialized CustomerOwned partition from disk and deserialize them. At aggregating phase, aggregation is performed on each partition by last_name field. Once a node finishes aggregation, it sends result to parent node. After parent nodes receive aggregation results from all of its children nodes, it joins all aggregation results including its and sends to the next parent. This joining aggregation results is considered as combining phase.

Two kinds of algorithms are implemented. One algorithm performs aggregation by deep-copying elements from a source vector loaded from a disk. In the other algorithm, each element of the source vector is wrapped in Arc, and its reference is acquired while aggregation. The difference between the both algorithm codes are represented in Figure 4·6 and in Figure 4·7. If we glance at the codes, the notable difference is only when we acquire an element from CustomerOwned Vec to construct an aggregated data structure. Therefore, there are few difference between the two kinds of tree-aggregate algorithm in terms of code appearance.

Numbers of CustomerOwned objects aggregated in our experiment are 2, 4, 6, 8 million.

```
fn aggregate_local(arr :&[Arc<CustomerOwned>])
{
    let mut agg = HashMap::new();
    let n = arr.len();
    for i in 0..n {
        let customer = Arc::clone(&arr[i]);
        let last_name = customer.last_name.clone();
        let vector = agg.entry(last_name).or_insert_with(Vec::new);
        vector.push(customer);
    }
    return agg;
}
```

Figure 4·6: Aggregation function with Arc

```
fn aggregate_local_copy(arr :&[CustomerOwned])
{
    let mut agg = HashMap::new();
    let n = arr.len();
    for i in 0..n {
        let customer = arr[i].clone();
        let last_name = customer.last_name.clone();
        let vector = agg.entry(last_name).or_insert_with(Vec::new);
        vector.push(customer);
    }
    return agg;
}
```

Figure 4·7: Aggregation function with deep-copy

4.6.1 Result

Figure 4·8 shows runtime performance of two tree-aggregate algorithms. The runtime of algorithm with deep-copy is about 40 to 50% slower than algorithm with Arc for every vector size. The memory usage of algorithm with Arc and size of 8 million is about 15G bytes. On the other hand, the memory usage of algorithm with deep-copy and size of 8 million is about 26G bytes.

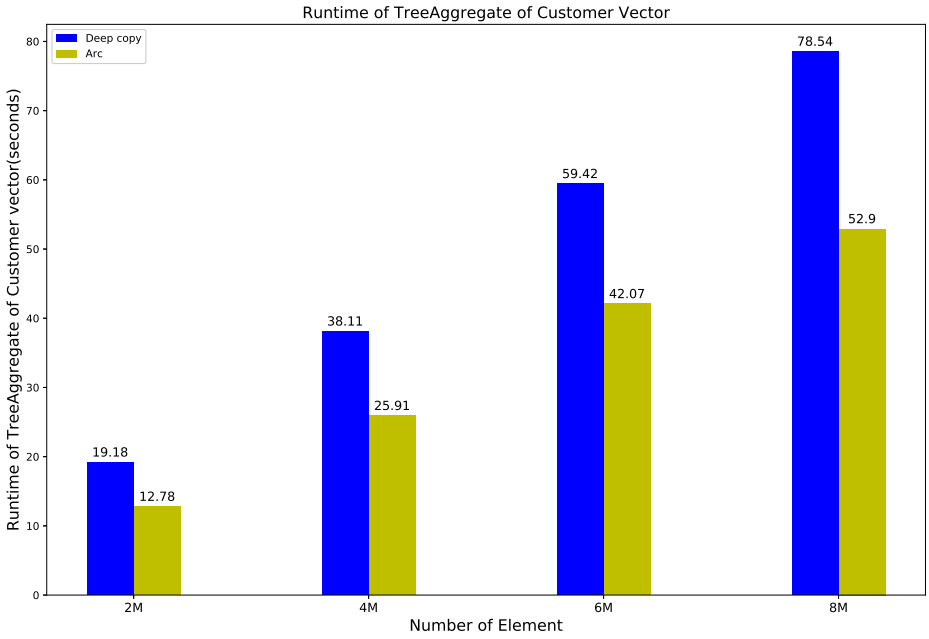

Figure 4·8: Runtime of Tree-aggregate algorithm

4.6.2 Discussion

As we explained, Arc has overhead to be deleted because it has to check if the value is still referred. The atomic operations are more expensive than ordinal memory access. Even though the use of Arc slows down runtime performance, deep copy of complex objects has more impact in deterioration of runtime performance.

At the aggregating phase, each object is deep-copied or acquired with Arc once during the runtime in order to construct aggregated data structures. If total number of objects is 1 million, the all of 1 million objects are deep-copied or cloned with Arc once during execution. Deep-copy allocates new memory for copied object. On the other hand, clone with Arc is merely acquisition of additional owner. Therefore, deep-copy is more expensive in terms of runtime and also use of memory than clone with Arc. Deep-copy processes all the original objects to generate newly deep-copied objects. This process has overhead and the existence of both original and copied objects doubles its memory usage.

After construction of aggregated data structure, the dropping variables of original objects occurs at the end of the aggregating phase. In the algorithm with deep-copy, these variables are owners so that drop of variables triggers deallocation of actual values. In the algorithm with Arc, the variables are Arc. The aggregated data structure contains Arc pointing to the same values pointed by original Arc. Since the aggregated data structures continue to live after aggregating phase, drop of original Arcs does not triggers deallocation of values. Therefore, drop of original variables can shows overhead of memory deallocation in algorithm with deep-copy, and overhead of checking reference count of Arc in algorithm with Arc.

In addition, memory access from Arc is slower than ordinal variables due to the atomic operations. This may be potential overhead of the algorithm with Arc.

Considered these theoretical analysis and result of our experiment, using Arc improves runtime performance and memory usage compared to algorithms using deep-copy in tree-aggregate algorithms.

4.7 Experiment 5: K-Nearest-Neighbors

Finally, we examine the performance of Machine Learning (ML) algorithms. The goal of this experiment is to evaluate better memory management strategies in ML algorithms developed with Rust. We employ K-Nearest-Neighbors (KNN) for ML algorithms to be studied in our experiments. Our KNN algorithms perform document classification on Wikipedia page data set described in Section 4.2.

The algorithms have 4 phases; load, preprocess, query, and combine phase. Before these phases, we separate data sets into 8 partitions to run these in different threads. The algorithms spawn threads at the beginning, and in the load phase partition of files are loaded in each threads. In the preprocessing phase, the algorithm process document strings to generate Term-frequencies (Tfs) matrices and other data structures. In the query phase, it calculates cosine similarities between all combination of train and test observations and select to top K nearest neighbors. In combine the phase, the results from each batch and from each thread are combined. Based on our experiment's result, the runtime in preprocess and query phase are significantly larger than other two phases.

Parameter Name	Values and Description
Method	deep-copy: use deep-copy to generate intermediate objects arc: use atomic reference count to generate intermediate objects
Strategy	1: keep intermediate objects in memory until owner is changed 2: remove intermediate objects as soon as it is not needed
Number of batch	2: generate 2 batches from each partition 3: generate 3 batches from each partition
k	15K: dimension of feature matrices is 15 thousands 20K: dimension of feature matrices is 20 thousands 25K: dimension of feature matrices is 25 thousands

Table 4.1: Parameter of KNN algorithms

Therefore, we focus our discussion in these two phases considering them bottlenecks in our KNN algorithms.

KNN algorithms are implemented with different memory management strategies. We parametrize these memory management and some other values. These parameters are listed in Table 4.1.

Method parameters are specified to select memory management strategy used in preprocess phase. In preprocess the phase, the algorithm generates many intermediate data structures using same String elements. Deep-copy method deeply copies String to generate intermediate data structures. Arc method use Atomic Reference Count (Arc) to wrap String elements in the original data structure and clone Arc when the elements are used in other data structures. As Experiment 4 had shown in in Section 4.6, deep-copy of complex objects is more expensive than cloning Arc. String is a sort of object allocated in heap and copied many times in preprocess phase. Therefore, it is worth assessing which method can be a better memory management strategy in our experiments.

Strategy parameter control memory management strategy used in both preprocess and query phase. Strategy 1 keeps all intermediate data structures and numeric matrix objects from preprocess to query phase. Strategy 2 removes these data structures and objects as soon as they are not needed. These strategies differ from each other in terms of memory usage and frequency of memory deallocations. These may cause significant runtime difference of the algorithms.

Number of batch controls number and size of batch for each partition. This parameter varies size of intermediate data structures and numeric matrices, and frequency of memory deallocations.

The size of each batch is defined in 6,250 pages when we have 2 batches, and 4166 or 4167 when we have 3 batches. The parameter k specifies the number of dimensions of feature matrices created out of the preprocess phase. By controlling this, it determines size of intermediate data structures and numeric matrices.

By controlling these parameter, we compare runtime and memory usage of KNN algorithms.

4.7.1 Result

Figures 4·9, 4·10, and 4·11 show the runtime performance of our KNN algorithms in total, preprocess, and query phase respectively. The algorithms used in our experiments are indexed in Table 4.2. We call each algorithm in algorithm number for easy understanding. The Algorithm 3 with 20K and 25K dimension, and the Algorithm 1 with 25K dimension whose runtime is showing 0 seconds are algorithms that are terminated during execution due to fail of memory allocation.

Figure 4·9 shows Algorithm 8 shows much slower performance compare to the other algorithms. Algorithm 7 also starts to slow down as we increase the number of dimensions.

As shown in Figure 4·10, the algorithms using Arc is much slower than using deep-copy in preprocessing phase. Algorithm 8 is 38% slower than Algorithm 6 in dimension of 25K. In addition, the algorithms with strategy 2 are slower than with strategy 1. For example, Algorithm 6 is about 85% slower than one Algorithm 2 in dimension of 25K. The algorithms with 3 batches are slower that ones with 2 batches. Furthermore, as we increase the numbers of dimension, the number of batches becomes more and more critical to runtime performance of the algorithms with deep-copy. The runtime ratios of difference from Algorithm 7 to Algorithm 8 are about 44%, 28%, and 39% for dimension of 15K, 20K, and 25K respectively. However, those from Algorithm 5 to Algorithm 6 are about 11%, 28%, and 54%. The ratios of difference increase proportionally to the number of dimensions.

In query phase, Algorithm 8 is much slower than the others in different parameter settings. Algorithm 7 starts to slow down as number of dimension is increased.

Algorithm number	Method	Strategy	Number of batch
1	deep-copy	1	2
2	deep-copy	1	3
3	arc	1	2
4	arc	1	3
5	deep-copy	2	2
6	deep-copy	2	3
7	arc	2	2
8	arc	2	3

Table 4.2: Index of Algorithm

4.7.2 Discussion

In the preprocess phase, the algorithms using Arc method perform worse than ones with deep-copy method. This result may seem to contradict to the result of Experiment 4 in Section 4.6. In Experiment 4, complex objects, CustomerOwned, are cloned with Arc or deep-copied. In this Experiment 5, we however get Arc or deep-copy of String instead of complex object. The result explains deep-copying String is more inexpensive than deep-copying the complex object and using deep-copy method is actually more efficient than using Arc to share String. Therefore, we need to make decision which method to use depending on size and complexity of objects.

In addition, the use of different strategies in the preprocess phase explains how deallocations of intermediate data structures and drops of their variables impact algorithm's runtime performance. The result shows deallocations of intermediate data costs a lot; Algorithm 6 is 85% slower than Algorithm 2. This may be a more severe bottleneck for algorithms using Arc; Algorithm 8 is 102% slower than Algorithm 4. This is the same reason explained in Experiment 4. Arc has to check reference count and its atomic operation is expensive.

The impact of batch number can explain how frequency of memory de/allocation affects runtime performance. Dealing with more number of batch has negative impacts on runtime performance because it triggers more frequent memory de/allocation. In addition, as we increase the number of dimension, the ration of runtime difference among different numbers of batch for the algorithms with deep-copy, but not for ones with Arc. This is because algorithm with Arc does not de/allocate

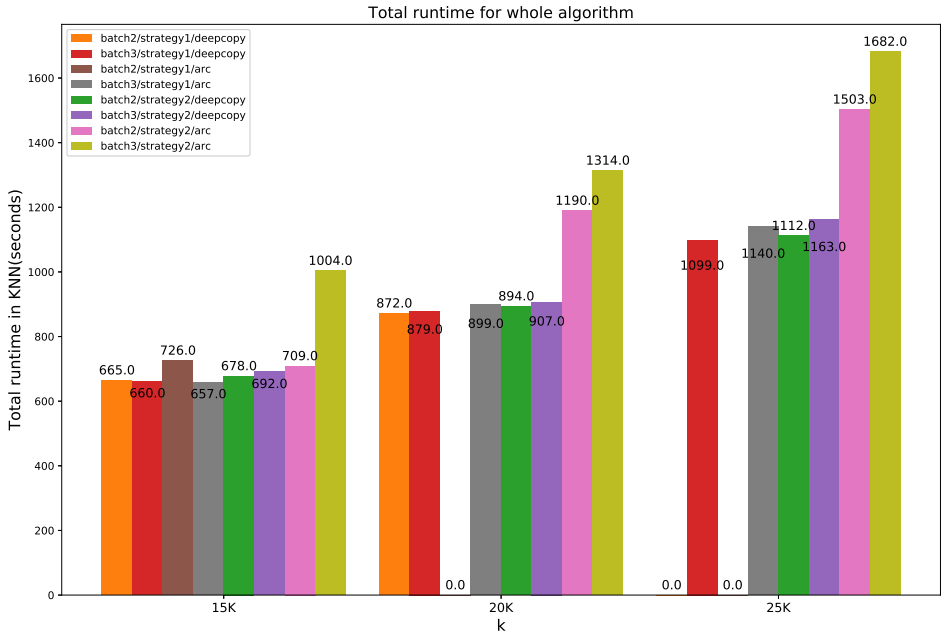

Figure 4·9: Total runtime whole KNN algorithm (seconds)

memory as much frequently as ones with deep-copy. The intermediate data structures merely get reference to String values without actually allocating memory for copied Strings.

In query phase, most of our algorithms shows similar runtime performance among the same number of dimension. Why such differences seen in preprocess phase are not shown in query phase? This is because the data structures processed in this phase are numeric matrices and vectors. These data structures do not contain String so that the selection of method parameter does not have direct impact to algorithm's runtime performance.

Furthermore, numeric matrices and vectors are contiguously allocated initializing their size, so they are easy to be de/allocated. The computation needed for these de/allocations is fairly fast. Therefore, setting parameters, such as number of batch and strategy, does not impacts runtime performance.

However, we can see significant degradation in Algorithm 7 with 20K and 25K dimensions and Algorithm 8 with 15K, 20K, and 25K dimensions. By using strategy 2, we deallocate the

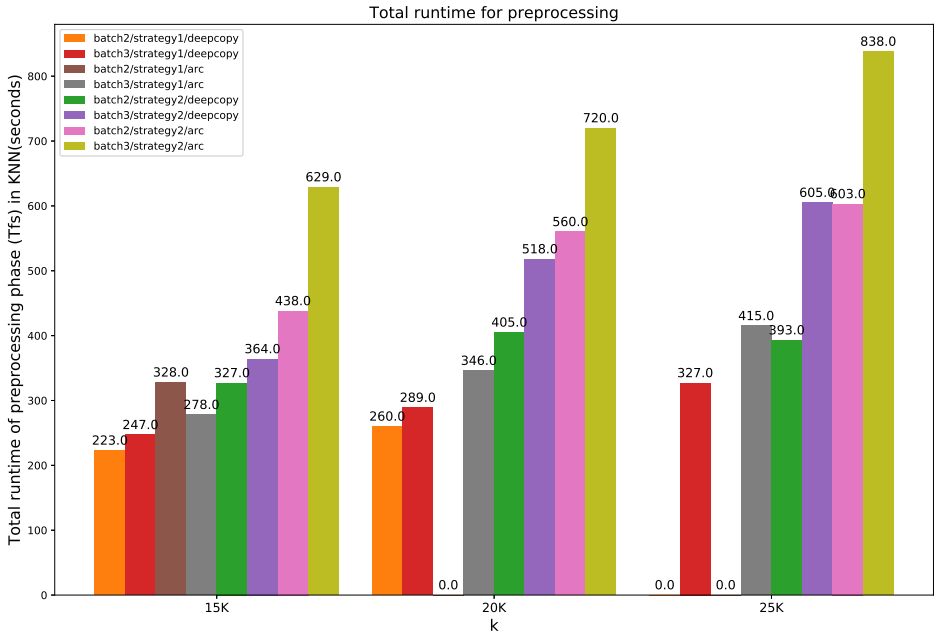

Figure 4·10: Total runtime of preprocessing phase in KNN (seconds)

numeric matrices and vectors in query phase. However, it is difficult to think that these contiguous memory deallocations cause the degradation. If that is so, Algorithm 5 and 6 should also show some overheads. No page-swapping happens during the execution. One possible reason is that these algorithms suffer from the allocation of numeric matrices and vectors due to previous deallocation of intermediate data structures in preprocess phase. The frequent deallocations and low locality of Arc may lead long free-space-list making difficult to search enough spaces to allocate numeric matrices and vectors. The other possibility is that the frequent deallocations and low locality of Arc cause cache of numeric matrices and vectors. The computation on these numeric objects may slow down due to their caching levels. To conclude this discussion, we need to conduct more experiments and deeper analysis. However, we push those discussion to next paper.

As a part of the conclusion, using Arc to String may be overhead. Considering the result from Experiment 4, we need careful consideration which object to use which strategy, deep-copy and Arc. In addition, frequent memory deallocation can lead to slower runtime performance. Therefore,

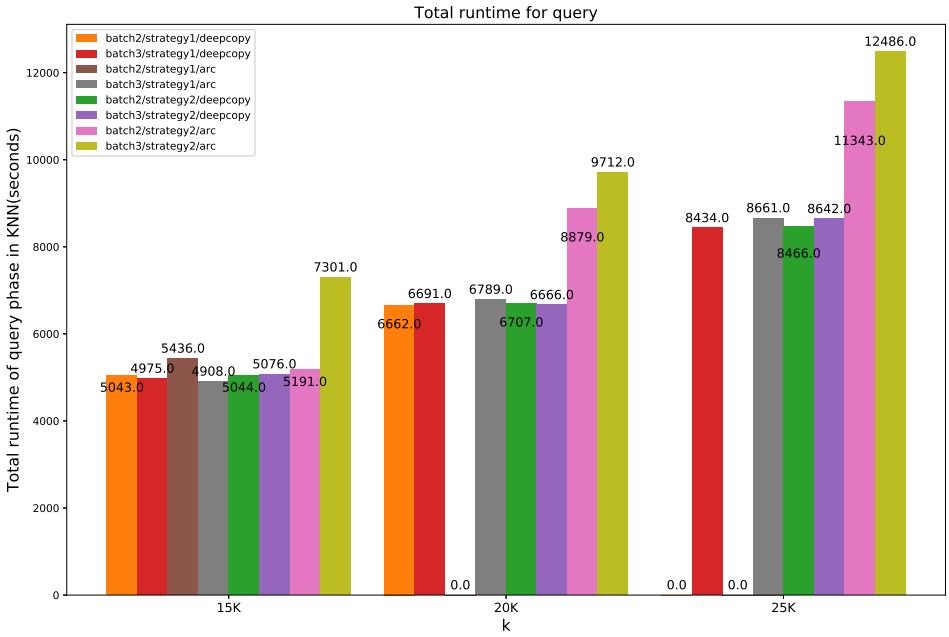

Figure 4·11: Total runtime of query phase in KNN (seconds)

we need to decide when to deallocate unused values taking account of memory capacity.

4.8 Summary

In this chapter, we examine various memory management strategies for different algorithms in Rust programming. There are many factors that might differ performance of algorithms: different variable type, Reference Counting, Atomic Reference Counting, and frequency of memory de/allocation. We should select the best memory management strategies when developing Big Data processing tools. Some tips for the development are proven in our experiments. We conclude this thesis in Chapter 5.

Chapter 5

Conclusions

In this thesis, we have presented a number of experiments to assess better implementation of algorithms when one develops Big Data analysis tools with Rust programming.

Differences between variable types do not have an impact on operations to access memory address where a value is located. This result gives us freedom of choice for variable types to construct complex objects in terms of runtime performance. However when we use borrowed types, such as references and slices, their lifetime should be explicitly defined. Therefore, we may use owner type variables to construct complex objects in order to facilitate lifetime tracking.

Runtime to drop Rc is 60 times slower than normal reference. This is because Rc needs to check reference count to determine whether to deallocate its value. This result may say that we should use normal reference whenever it is possible. Again, tracking lifetime of references used in complex objects can be cumbersome. Choice of Rc and reference is dependent on complexity of objects implemented.

In Rust, we can implement multithread programming by sharing data using Arc. In simple multithread algorithms, one can write code that shares data with Arc and simple reference among different threads. Merge-sort algorithms sharing data with Arc is 21% slower than normal reference. In this situation, sharing data with normal reference require relatively easy lifetime tracking, so we should use normal reference to share data among different thread whenever it is possible.

For more complex algorithms, like tree-aggregate and preprocess phase in KNN, one can implement algorithms sharing data with Arc or simply deep-copy the values. The algorithms with deep-copy are about 40 to 50% slower than the algorithms with Arc for complex objects. However when we deal with String, the algorithms with deep-copy are faster than Arc. Therefore, the decision on whether to use deep-copy or Arc method should be determined based on the complexity of

objects.

Decreasing frequency of memory de/allocation may be an effective solution to improve runtime performance of Big Data processing algorithms. There is a trade-off between memory usage and frequency of memory de/allocation. That is why developers should conduct careful analysis of algorithms' memory usage pattern and timing of memory de/allocation.

As we can see in the results of our experiments, different memory strategies vary the performance of algorithms in Rust programming. Which memory management strategy to take depends on what objects to deal with. Therefore, development of Big Data analysis tools in Rust programming should be started with objects implementation used in the systems. Next, one can select suitable memory management strategies. Finally, algorithms can be optimized with more dedicated strategies to application setting, such as capacity of memory.

Appendix A

Linear Algebra Computation

A.1 Create Java interface of CBLAS with JNI

1. Download BLAS and build using make file. In the figure2.1, the built file is libblas.a and the header file is blas.h.

2. Download CBLAS and build using make file (I am not sure whether we should build archive file or shared library). In figure, the build file would be libcblas.a or libcblas.dylib and header file is cblas.h.

3. Create java file which will be the Java interface of CBLAS.

4. Compile java file with -h header flag to create class file (CBLASJ.class) and header file (CBLASJ.h).

```
$ javac -h . CBLASJ.java
```

5. Create C file (CBLASJ.c) which will bind Java interface and CBLAS library. And compile it with JNI to create object file (CBLASJ.o).

```
$ gcc -c -fPIC -I${JAVA_HOME}/include -I${JAVA_HOME}/include/darwin CBLASJ.c
```

6. Compile shared library linking library (libcblas.a or libcblas.dylib) to object file (CBLASJ.o).

```
$ gcc -o libcblas.dylib (or libcblas.a) CBLASJ.o
```

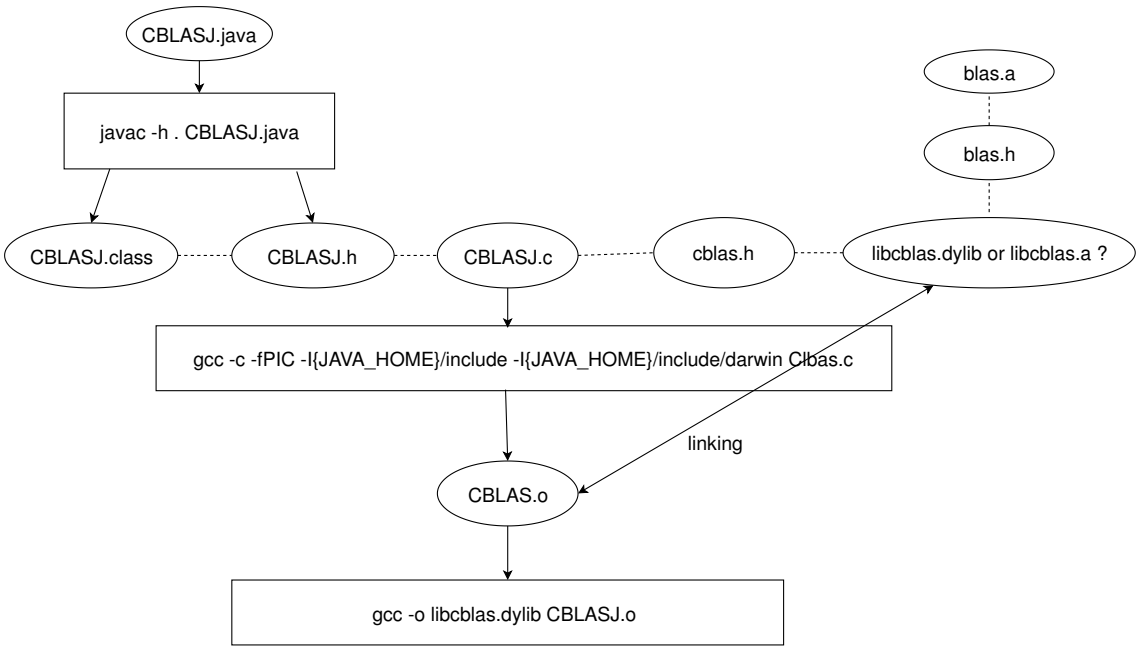

Figure A·1: Integration of Native Methods

References

[1] Apache flink, 2020. https://flink.apache.org/.

[2] Apache hadoop, 2020. http://hadoop.apache.org/.

[3] Apache spark, 2020. http://spark.apache.org/.

[4] Nd4j, 2020. https://nd4j.org/.

[5] netlib, 2020. https://www.netlib.org.

[6] Rust arc documentation, 2020. https://doc.rust-lang.org/std/sync/struct.Arc.html.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[8] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394. ACM, 2015.

[9] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Engineering. Bulletin.*, 38(4):28–38, 2015.

[10] Jack J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Computer Architecture News*, 20(3):22–44, 1992.

[11] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004.

[12] Charles L. Lawson, Richard J. Hanson, D. R. Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions Mathmatical Software*, 5(3):308–323, 1979.

[13] Richard B. Lehoucq, Danny C. Sorensen, and Chao Yang. *ARPACK users' guide - solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. Software, environments, tools. SIAM, 1998.

[14] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17:34:1–34:7, 2016.

[15] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing (Harry) Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, pages 675–690. ACM, 2015.

[16] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowmen*, 8(13):2110–2121, 2015. https://doi.org/10.14778/2831360.2831365.

[17] Brian T. Smith, James M. Boyle, Jack J. Dongarra, Burton S. Garbow, Yasuhiko Ikebe, Virginia C. Klema, and Cleve B. Moler. *Matrix Eigensystem Routines - EISPACK Guide, Second Edition*, volume 6 of *Lecture Notes in Computer Science*. Springer, 1976.

[18] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: holistic memory management for big data processing over hybrid memories. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 347–362. ACM, 2019.

[19] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. An experimental evaluation of garbage collectors on big data applications. *Proceedings of the VLDB Endowmen*, 12(5):570–583, 2019.

[20] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan R. Sparks, Aaron Staple, and Matei Zaharia. Matrix computations and optimization in apache spark. In Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi, editors, *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pages 31–38. ACM, 2016.

[21] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[22] 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 Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1189–1204. ACM, 2018.

CURRICULUM VITAE