COMPSCI 590S Systems for Data Science **Fall 2017**

Lecture 11: FlumeJava, Memory Allocation and Garbage Collection

Lecturer: Emery Berger *Scribe(s): Anmol Singh Suag, Cen Wang*

Last time, we talked about the widespread usage of JVM languages (Java, Scala, Clojure, Kotlin, etc.) in big data area. JVM has the advantages of:

1. Garbage collection makes it easy to write correct programs.
2. JVMs JIT compiler is optimized so that JVM has high performance.

JIT startup time is a little concerning, but if our program will keep running for a long time then the cost is amortized.

11.1 FlumeJava

Because memory is much faster than disks and networks, we'd like to do as much computation as possible in RAM to minimize IO cost. For a MapReduce stage, both the map result and the reduce result have to be written to disk or GFS (which is network IO). If the problem at hand (e.g. wordcount in MapReduce) requires only one MapReduce stage, this IO cannot be avoided. But usually, real life problems requires a pipeline of MapReduce jobs, meaning that multiple stages of MapReduce will be started. A typical example will be running stochastic gradient descent algorithm in MapReduce.

```
epsilon = 0.0001
last_result # Stores result for last iteration
while True:
    result = do_some_mapreduce_job()
    if last_result - result <= epsilon: # The result has converged
        break
```

In the above example, multiple MapReduce jobs will be started until we reach the convergence condition. In this situation, IO cost between the previous and the current MapReduce slows the entire computation down. FlumeJava's idea is that instead of starting multiple MapReduce phases, we can first try to optimize the computation process so that IO costs can be reduced.

The FlumeJava library does this by deferring the actual computation to a later time. When the user is defining a series of transformation of the data, there is actually no real computation happening in the MapReduce cluster. Instead, FlumeJava secretly constructs a computation graph (or dataflow graph) of the transformation. The dataflow graph exposes opportunities to optimize the computation process, saving the unnecessary MapReduce phases and thus saves IO and time. The FlumeJava library works in the following way:

1. Defer the computation and construct dataflow graph
2. Optimize the dataflow graph

3. Execute the optimized dataflow graph

For example, if we are going to transform data first by function f and then by function g, a handwritten MapReduce pipeline will be:

```
y = mapreduce(f(x))
# A lot of IO happening here to save the result
z = mapreduce(g(y))
# A lot of IO happening here to save the result
```

The FlumeJava library will first draw a computation graph and then notice that the intermediate result y is used immediately after the first MapReduce job, so it will fuse the two MapReduce jobs into one:

```
mapreduce(g(f(x)))
# A lot of IO happening here to save the result
```

This saves one unnecessary IO phase. FlumeJavas strategy can also be referred as lazy evaluation, which is different from a handwritten MapReduce pipelines eager evaluation where there will be no opportunities for optimization.

There is a tradeoff between eager and lazy evaluation.

1. Lazy evaluation: suitable for **batch processing**
   (a) - Higher latency
   (b) + Higher throughput
2. Eager evaluation: suitable for **stream processing, to be discussed**
   (a) + Lower latency
   (b) - Lower throughput

11.2 Memory Management

In C/C++, programmers have to manually manage their memory.

C: new/malloc

C++: delete/free

Manual memory management is risky. Many kinds of security problems have been discovered to be related to manual memory management.

11.2.1 Memory allocator

A traditional memory allocator is called bump pointer allocation. When you call sbrk(8), you can get 8 bytes of heap memory, and a pointer pointing to where the heap used to be will be returned.

11.2.2 Memory utilization

Imagine we have the following program:

```
for (int i = 0; i < 10000000; i++) {
    void* p = malloc(8);
    // Do something here
    free(p);
}
```

It first requests 8 bytes of memory, does some computation, and then free the that 8 bytes. This program needs at most 8 bytes of heap memory. Then consider the following naive memory allocator implementation:

```
void* malloc(size_t size) {
    return sbrk(size);
}

void free(void* p) {
    // Do nothing
}
```

What this memory allocator does is that it give whatever size of memory the user want but it never tries to free any memory. If we use the above naive memory allocator, then our memory consumption goes linearly with the time but the actually memory needed is only 8 bytes. There is a large amount of the memory being wasted with this memory allocator. We can think with the following formula:

Fragmentation = (memory consumed by allocator - memory requested by user) / memory requested by user

An ideal allocator would be that the allocator only consumes the amount of memory requested by the user. In this case, fragmentation is 0.

11.2.3 Memory fragmentation

11.2.3.1 Internal fragmentation (means memory is wasted inside the allocated object)

Modern processors usually can only access the memory at the granularity and alignment of their word size, known as alignment restriction. If memory is misaligned, the system may have a poor performance. Because of this, when you try to request 1 byte of heap memory by malloc(1), Linux operating system will actually allocate 8 bytes of memory for you to meet the alignment requirement. The extra 7 bytes of memory is wasted inside the object.

A second form of internal fragmentation is memory bookkeeping. In Linux, another 8 bytes of heap metadata is attached for every object that is allocated. Notice that when we free heap memory by free(p), we dont have to specify how many bytes we want to free. This information is actually bookkept in heap metadata, so when the operating system receive a request to free some part of the memory, it only needs to look ahead 8 bytes to the place where the metadata is stored, then it knows how many bytes to free.

Pros and cons:

1. + Cheap implementation
2. - Space utilization

3. - Cache utilization
4. - Security

BiBoP allocator

An alternative allocation strategy is BiBoP. The allocator creates different pages for different sizes of objects. So there may be 8-byte, 16-byte, 32-byte object page. At the beginning of the page, there is a pointer pointing to other memory address where the page metadata is stored such as size. So when we allocate for new objects, those objects of the same size go to the same page. When we do free(p), under the hood, we will first find the pointer at the beginning of the page and then follow the pointer to get the page metadata which is the same for every object in that page.

11.2.3.2 External fragmentation (means memory is wasted outside the allocated object or between allocated objects)

Imagine we have 8 bytes of memory:

```
----------
|________|
```

We first request 1 byte of memory eight times (suppose we do not have alignment restriction here):

```
----------
|xxxxxxxx|
----------
```

Then we free the first, third, fifth and seventh allocation:

```
----------
| x x x x|
----------
```

At this time, we have 4 bytes of free memory. But even if the space is there, the program cannot even allocate a 2-byte memory. This phenomenon is known as external fragmentation.

One solution to this problem is compaction. We can compact the above memory to eliminate external fragmentation:

```
----------
|xxxx____|
```

We can now allocate a 2-byte memory. Memory compaction has high cost, as you have to move objects and edit the pointers. Also, this is not doable in C/C++.

11.3 Garbage Collection

C/C++ are not type-safe and they dont differentiate between pointers and data. If data addresses were to be changed, the pointer algorithms would fail. Hence, compaction and garbage collection doesnt work with

C/C++, memory has to be manually freed.

There are 2 reasons why Garbage Collection is interesting : It is easy and safe; There is no manual deleting required of unused objects It prevents memory fragmentation through memory compaction

LISP was the first language to be shipped with in-built Garbage Collectors around 1960.

There are 2 Garbage Collection Algorithms:

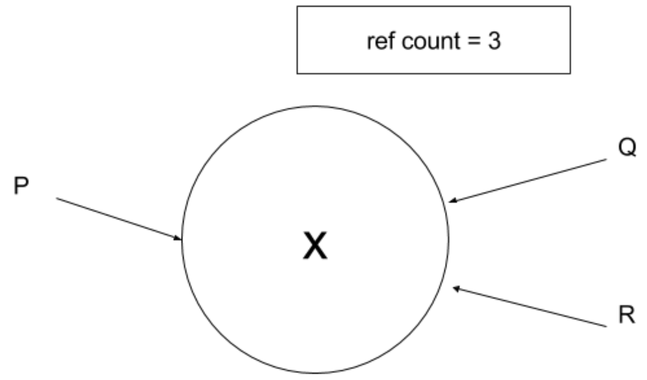
11.3.1 Reference Counting

Reference Counting is a very simple algorithm, but has a fundamental problem. In this algorithm, each object maintains a count of the number of pointers that are pointing to it. This count is called its Reference Count. The object is immediately deleted by the algorithm when its Reference Count becomes 0.

Say, for eg:

```
P = new(X)
Q = P
R = P
```

The object X now has a Reference Count of 3.

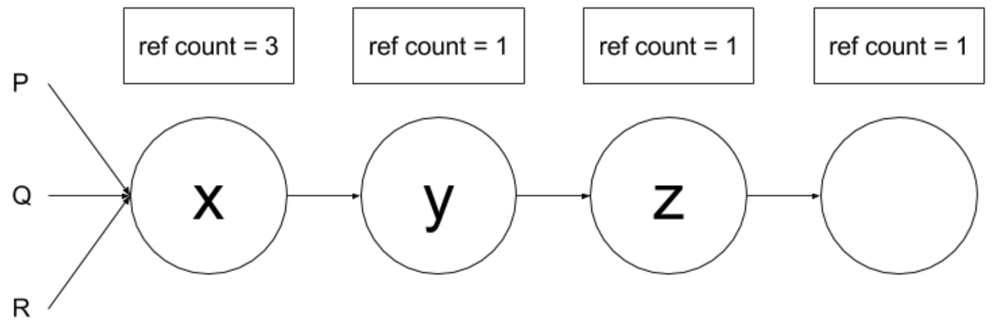


Now say we perform:

```
Q = null // Ref Count = 2
R = null // Ref Count = 1
P = null // Ref Count = 0 and X is deleted by the algorithm
```

X is deleted as soon as its Reference count drops to 0.

Developing further on the previous example, say X points to another object Y that points to another object Z and so on. The Reference Counts can be showed as follows:



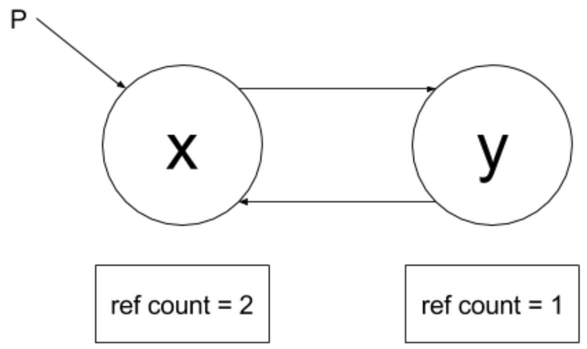
When P,Q are R are set to null, the Reference Count of X becomes 0 and it is deleted. Hence, the Reference Count of Y becomes 0 and it is deleted as well. Similarly, Z and so on are deleted by the Reference Counting Garbage Collector.

This seems like a great approach! Reference Counting was developed in the 1950s and was used in LISP. It was also adopted in Perl.

Reference Counting algorithm is:

1. Super Easy (+)
2. Immediate (+)
3. Incomplete (-)

Reference count garbage collector can reclaim a piece of memory only when that piece of memorys reference count reaches 0. It is incomplete in its fundamental nature. The following figure explains the problem:



Object X has a Reference Count of 2 and Y has a Reference Count of 1. When P is marked as null, we are left with a cycle where X and Y point to each other and hence have Reference Counts of 1 each. This cycle would not be deleted by the algorithm as their Reference Counts are not 0. Such cycles are called Reference Cycles.

Memory can not be reclaimed from cycles by Reference Counting Garbage Collector even if they are not reachable from the outside world. Avoiding cycles is hard; Like Doubly Linked-Lists have a forward and backward pointers for each node. This is the reason Perl servers have to be periodically restarted; dumping everything and starting afresh is an easy work-around.

The other garbage collecting algorithm that can detect cycles is Mark-And-Sweep.

11.3.2 Mark-And-Sweep

The Mark-And-Sweep Garbage Collector follows a simple rule : Objects that are unreachable by pointers are garbage. The algorithm considers reachability of an object to detect its aliveness.

Every object has a Mark Bit. Starting from the globals, stack and registers (the roots) , the algorithm follows the paths along the pointers to mark all the objects it crosses as black.

```
Mark(P) {
   if( Markbit (P) == White ) {
      Markbit (P) = Black }
   Mark( All_Outgoing_Links)   Recursion
   }
```

Eventually, all reachable (alive) objects are marked as Black. This is the Mark phase of the algorithm. All objects that remain marked as White are considered to be garbage and are deleted in the Sweep phase.

Mark-And-Sweep is a complete algorithm that is guaranteed to free all garbage. It is a deferred collection algorithm and may result in big pauses.

1. + Complete

2. - Deferred Collection

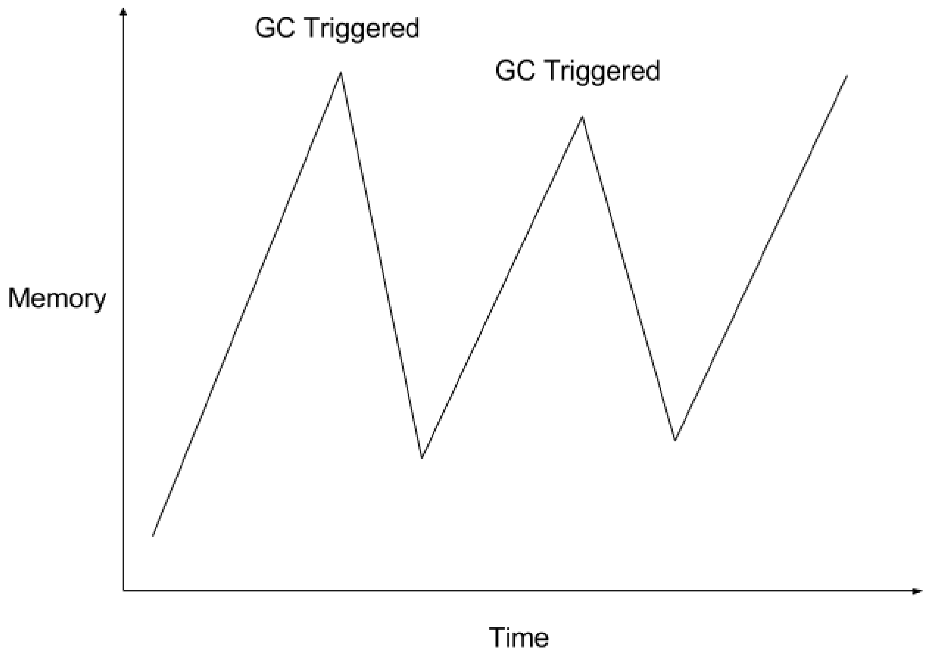
3. - Big Pause

Garbage collection is computationally expensive and involves a lot of operations. If Garbage Collection was to be triggered after every operation, it would slow down the throughput of the machine by a huge margin. Therefore, Mark-And-Sweep does a Deferred Collection.

1 Op $\rightarrow$ 1 Billion Ops for GC $\rightarrow$ 1 Op $\rightarrow$ 1 Billion Ops for GC $\Rightarrow$ Not Good

1 Billion Ops $\rightarrow$ 1 Billion Ops for GC $\Rightarrow$ Good

We want to accumulate garbage for some while and then clean them to amortize the GC cost rather than start GC every operation.



As we chose to defer Garbage Collection and let garbage accumulate while immediately getting more memory as long it lasts, we have to keep a lot of extra memory to accommodate the alive and dead objects until the collection happens. This wastes a lot of memory and results in poor memory utilization. Maximising Memory Utilization is desirable. In C, C++ and MapReduce if you need 1 GB memory, you use just 1 GB memory. Hence, the memory utilization is 1. In Java with Garbage Collection, to use 1 GB memory, you need about 3-5 GB memory. Therefore the utilization is quite small.

Memory Used	Memory Needed	Utilization	Examples
1 GB	1 GB	1	C, C++, MapReduce
1 GB	about 3-5 GB	1/3 to 1/5	GC, Java, Hadoop