Leaky Buffer: A Novel Abstraction for Relieving Memory Pressure from Cluster Data Processing Frameworks

by

Zhaolei Liu

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree Master of Science

Approved, Thesis Committee:

T. S. Eugene Ng, Chair
Associate Professor of Computer Science and Electrical and Computer Engineering

Alan L. Cox
Professor of Computer Science and Electrical and Computer Engineering

Chris M. Jermaine
Associate Professor of Computer Science

HOUSTON, TEXAS
JUNE, 2015
ABSTRACT

Leaky Buffer: A Novel Abstraction for Relieving Memory Pressure from Cluster Data Processing Frameworks

by

Zhaolei Liu

The shift to the in-memory data processing paradigm has had a major influence on the development of cluster data processing frameworks. Numerous frameworks from the industry, open source community and academia are adopting the in-memory paradigm to achieve functionalities and performance breakthroughs. However, despite the advantages of these in-memory frameworks, in practice they are susceptible to memory-pressure related performance collapse and failures.

The contributions of this thesis are two-fold. Firstly, we conduct a detailed diagnosis of the memory pressure problem and identify three preconditions for the performance collapse. These preconditions not only explain the problem but also shed light on the possible solution strategies. Secondly, we propose a novel programming abstraction called the leaky buffer that eliminates one of the preconditions, thereby addressing the underlying problem. We have implemented the leaky buffer abstraction in Spark for two distinct use cases. Experiments on a range of memory intensive aggregation operations show that the leaky buffer abstraction can drastically reduce the occurrence of memory-related failures, improve performance by up to 507% and reduce memory usage by up to 87.5%.
Acknowledgements

Ultimate thanks to Prof. Eugene Ng for all intelligent advice and guidance, under which I have made substantial progress on developing research methodology and scientific insights towards the completion of this thesis. At a loss for words, all the experience will become my lifetime treasures.

Special thanks to all members in the network research group. I spent meaningful four years in the group since I was a sophomore. I was able to conduct research and complete this thesis only with all of your supports and friendship.

Special thanks to all committee members. I appreciate all insightful feedback given by Prof. Alan Cox and Prof. Chris Jermaine. Thanks for all your efforts and advices during the thesis defense and thesis editing.
## Contents

Abstract  
List of Illustrations  

1 Introduction  
1.1 Motivation  
1.2 Three preconditions  
1.3 Leaky buffer abstraction  

2 Problem diagnosis  
2.1 Problem diagnosis  
2.2 Preconditions for performance collapse  
2.3 A concrete example: Spark hash aggregation  
2.4 Doesn’t “do not do X” solve the problem?  

3 Leaky buffer  
3.1 Leaky buffer abstraction  
3.1.1 High level ideas  
3.1.2 Conventional design vs. the leaky buffer abstraction  
3.2 Implementation  
3.2.1 groupByKey  
3.2.2 join  
3.2.3 Alternative implementation  

4 Evaluation
4.1 Evaluation ........................................... 24
  4.1.1 Experiment setup .................................. 24
  4.1.2 Workload description .............................. 25
  4.1.3 Leaky buffer on realistic workload: groupByKey .... 25
  4.1.4 Leaky buffer on realistic workload: join .......... 29
  4.1.5 Other benefits of the leaky buffer .................. 30
  4.1.6 Leaky buffer on artificial workload: measuring scalability ... 33
  4.1.7 Overhead analysis .................................. 35
  4.1.8 Alternative implementation evaluation ............... 35
  4.2 Related work ....................................... 37

5 Conclusion ............................................ 40
  5.1 Conclusion .......................................... 40

Bibliography ............................................ 41
Illustrations

2.1 Heap utilization of one node during an in-memory hash aggregation job

2.2 CDF of shuffle flows’ finish times of an in-memory hash aggregation job

2.3 CPU utilization of one node during the reduce stage of an in-memory hash aggregation job

3.1 An illustration of the leaky buffer abstraction

3.2 Original Spark’s `groupByKey` implementation

3.3 Leaky buffer implementation of `groupByKey`

3.4 Leaky buffer implementation of `join`

4.1 `ip-countrycode`

4.2 `ip-keyword`

4.3 `day-countrycode`

4.4 `day-keyword`

4.5 `month-keyword`

4.6 `rank-revenue`

4.7 `rank-countrycode`

4.8 `rank-duration`

4.9 Reduce stage run time with different memory sizes

4.10 Garbage collection time spent in the reduce stage

4.11 CDFs of shuffle flows’ finish times under different scenarios

4.12 Artificial data set evaluation
4.13 Comparing two implementations of the leaky buffer \ldots \ldots . \quad 36
Chapter 1

Introduction

1.1 Motivation

When MapReduce [1] and Hadoop [2] were introduced in 2004 and 2005, a high end multi-processor Intel Xeon server would have 64GB or 128GB of RAM. Fast forward to 2015, an Intel Xeon E7 v2 server can support 1.5TB of RAM per CPU socket; a high end server can have 6TB of RAM. This factor of 50 to 100 increase in memory capacity over the past decade presents a tremendous opportunity for in-memory data processing. It has been widely acknowledged that in-memory computing enables the running of advanced queries and complex transactions at least one order of magnitude faster than doing so using disks, leading to companies and enterprises shifting towards in-memory enabled applications for speed and scalability [3, 4].

This shift to the in-memory paradigm has had a major influence on the development of cluster data processing frameworks. On one hand, old instances of reliance on hard disks (e.g., Hadoop’s shuffle operation stored incoming data on disks, then re-read the data from disks during sorting) that were motivated by a limited amount of RAM are being re-visited [5]. On the other hand, in-memory processing is being exploited to offer new functionalities and accelerated performance. More and more cluster data processing frameworks and platforms from the industry, open source community and academia [6–11] are adopting and shifting to the in-memory paradigm. For example, Spark supports the traditional MapReduce programming model and
enables in-memory caching, which enables much faster data processing and computation [9, 11]. It can achieve 100x performance improvement when compared to Hadoop in some scenarios and can support interactive SQL-like query that leverages in-memory data table caching.

However, despite the advantages of these in-memory cluster data processing frameworks, in practice they are susceptible to memory-pressure related performance collapse and failures. These problems are widely reported by user communities and greatly limit the usability of these systems in practice [12–15]. Take Spark as an example, our experimental results leave us initially quite perplexed (details found in Section 4.1) We define a task-input-data-size to task-allocated-memory ratio, or data to memory ratio (DMR) for short, as the total job input data size divided by the number of parallel tasks divided by the amount of memory allocated to each task.

For a job with the groupByKey operation, a DMR of 0.26, which would seem generous, already causes a collapse of the reduce stage performance by up to 631%; the best performance may require a DMR as small as 0.017 as Figure 4.4 in Section 4.1.3 shows; in another job with the join operation, a moderately aggressive DMR of 0.5 is enough to cause the reduce stage to fail, as Figure 4.7 in Section 4.1.4 shows.

This thesis is motivated by these observations and addresses two questions: Firstly, what are the underlying reasons for the very high sensitivity of performance to DMR? The severity of the performance collapse leads us to suspect that there may be other factors beyond memory management overheads involved. Secondly, what system design and implementation techniques are there to improve performance predictability,

*An alternative definition of DMR for reduce tasks might use the total MapperOutputFile size instead of the total job input data size. However, we do not adopt this definition because users typically do not relate to the size of the MapperOutputFiles.
memory utilization, and system robustness, striking the right balance between relieving memory-pressure and performance? A simple offloading of data from memory to disk will not meet our objectives.

1.2 Three preconditions

A commonality among the aforementioned in-memory data processing frameworks is that they are implemented using programming languages (e.g. Java, Scala) that support automatic memory management\(^\dagger\). However, the memory management overhead associated with garbage collection alone is not enough to explain the severity of the performance collapse. In fact, it is the combination of such overhead and the distributed inter-dependent nature of the processing that is responsible. Specifically, we identify three preconditions for the performance collapse:

1. The data processing task produces a large number of *persistent* objects. By persistent, we mean the objects stay alive in memory for nearly the entire lifetime of the task. This condition leads to costly garbage collections.

2. The data processing task causes a high rate of object creation. This condition is necessary for frequent garbage collections.

3. There are multiple concurrent data processing tasks and they exchange large amount of data simultaneously with memory intensive processing of this exchanged data. This condition dramatically exacerbates the negative impact of

\(^\dagger\)While the pros and cons of automatic memory management are debatable, arguably the pros in terms of eliminating memory allocation and deallocation related bugs outweigh the cons because otherwise many of these frameworks and the whole industry surrounding them might not even exist today!
garbage collections because a task-to-task exchange stalls if just one of the two tasks is stalled.

It is not hard to see why common join and groupByKey aggregation jobs meet all three of these preconditions and are therefore susceptible to performance collapse and failure – In such jobs, the reducers perform a data shuffle over the network (condition 3), and the incoming data are processed into a hash table (condition 1) containing a very large number of key-value objects that persist until the end of the job (condition 2).

These preconditions not only explain the performance collapse problem but also shed light on the possible solution strategies – A technique that eliminates one or more of the preconditions would be a good candidate.

1.3 Leaky buffer abstraction

The solution we propose is a novel abstraction called leaky buffer. It eliminates precondition 1 while ensuring a high degree of overlap between network I/O and CPU-intensive data processing. Specifically, the leaky buffer models incoming data as having a combination of control information and data contents. Furthermore, the control information is the subset of data that the task needs to process the data contents. For example, a task that sorts key-value pairs according to the keys requires the keys as the control information. Upon receiving incoming data, the leaky buffer holds the data contents in an in-memory buffer, but leaks control information in a programmable manner to the data processing task. In this way, the task can still perform computation on the control information (e.g. construct the logical data structures for organizing the incoming data) that is necessary for data processing. We have implemented the leaky buffer abstraction in Spark for both the join
and `groupByKey` operations. We experimentally demonstrate that the leaky buffer abstraction can improve performance predictability, memory utilization, and system robustness. Compared to the original Spark using the same DMR, the reduce stage run time is improved by up to 507%; execution failures that would occur under original Spark are avoided; and memory usage is reduced by up to 87.5% while achieving even better performance.
Chapter 2

Problem diagnosis

2.1 Problem diagnosis

This section presents a diagnosis of the memory pressure problem and the subsequent performance collapse. We explicitly focus on the case where a system is implemented using programming languages (e.g. Java, Scala) that support automatic memory management with tracing garbage collection. Automatic memory management is favored by developers because it frees developers from the burdens of memory management. Among different garbage collection types, tracing is the most common type, so much so that “garbage collection” often refers to tracing garbage collection [16]. Tracing garbage collection works by tracing the reachable objects by following the chains of references from certain special root objects, then all unreachable objects are collected as garbage. The tracing cost is thus positively correlated with the number of objects in the heap.

2.2 Preconditions for performance collapse

The first precondition is that:

The data processing task produces a large number of persistent objects. By persistent, we mean the objects stay alive in memory for nearly the entire lifetime of the task. This condition leads to costly garbage collections.
When there is a large number of persistent objects in the heap, tracing has to visit a large number of objects and thus garbage collection cost will soar. Furthermore, as the heap gets filled with persistent objects and the memory pressure builds, each garbage collection operation may free very little memory – in other words, the system is paying a high price for little benefit.

The second precondition is that:

_The data processing task causes a high rate of object creation. This condition is necessary for frequent garbage collections._

The emphasis here is that when the first precondition holds, the second precondition has a significant negative effect. Garbage collectors are highly optimized for handling high rate of creation and deletion of _short-lived_ objects. For example, the parallel garbage collector in the Java virtual machine (JVM) uses generational garbage collection strategy [17]. The short-lived objects live in the young generation in the heap and get collected efficiently when they die. However, when the first precondition holds, this second precondition will aggravate the memory pressure problem. When a large number of persistent objects occupy a large portion of the heap, creating new objects will often require garbage collection to reclaim space. More frequent object creation will trigger more frequent garbage collection events as well. At the same time, garbage collection in this situation is slow and inefficient because it has to trace through a large number of persistent objects, and can only recover very little space since most of those objects are alive.

The third precondition is that:

_There are multiple concurrent data processing tasks and they exchange large amount of data simultaneously with memory intensive processing of_
this exchanged data. This condition dramatically exacerbates the negative impact of garbage collections because a task-to-task exchange stalls if just one of the two tasks is stalled.

Precondition 3 helps to propagate the delays caused by memory pressure problem on one node to other nodes. When one node is suffering from inefficient, repeated garbage collections, much of its CPU resources may be used by the garbage collector, or even worse the garbage collection forces stop-the-world pause to the whole process. Consequently, other activities such as network transfers can be delayed or paused as well. In turn, tasks running on other nodes can be delayed because they must wait on the network transfers from this node. Take the shuffle operation as an example: during the reduce stage of the MapReduce programming model, the shuffle operation refers to the reducers fetching mapper-output-files over the network from many other mappers. If some of the mapper nodes are busy with garbage collection, the network throughput of the shuffle traffic will degrade and the reducers’ computations will be delayed. Because a network transfer can be disrupted if just one of the two ends is disrupted, similarly, if some of the reducer nodes are busy with garbage collection, the shuffle traffic to those nodes will also slow down.

2.3 A concrete example: Spark hash aggregation

This section illustrates the memory pressure problem more quantitatively by a Spark job with groupByKey operation that satisfies the preconditions for performance collapse. groupByKey is a typical hash aggregation operation in MapReduce systems. It processes the key-value pairs by aggregating the values that have the same key together. This job runs on a cluster of the same setting as in Section 4.1.1 (each task gets 1.5GB of memory). It uses a workload as described in Section 4.1.6 with $2 \times 10^9$
Figure 2.1: Heap utilization of one node during an in-memory hash aggregation job. Short-lived objects reside in the so called eden space, while persistent objects reside in the so called old space.

key-value pairs for a total input size of 13.6GB. Spark chooses a default number of tasks based on the input size so that this job has 206 map tasks and the same number of reduce tasks, which gives a DMR of 0.04.

Figure 2.1 illustrates the utilization statistics of the JVM heap. The map stage of the job ends after approximately 300 seconds followed by the reduce stage. During the map stage, most garbage collection happens in the eden space, a region in the heap where short-lived objects reside, as can be seen by the frequent drops in eden space utilization. At the same time, an increasing amount of objects (both live and dead) accumulates in the old space, where persistent objects reside.

During the reduce stage, because the task executor builds a large in-memory hash table to store incoming key-value pairs, and these objects in the hash table persist until the end of the task, a large amount of objects are created rapidly and are pushed from the eden space to the old space (observe the multiple rounds of steady and rapid increases in old space utilization), satisfying preconditions 1 and 2. Each sudden flattening out and drop of utilization in the old space indicates one or more
full garbage collection events that last several seconds. Overall, 116 seconds out of 515 seconds were spent in garbage collection, and most of these 116 seconds were spent during the reduce stage, resulting in a large degradation in overall performance.

The shuffle operation inherent in this job satisfies precondition 3. Full garbage collection brings a stop-the-world pause to the whole node. Thus, a pause caused by a task executor thread will pause network transfers as well. Therefore, such a stop-the-world pause at a node may cause a pause of executors on other nodes if those executors have finished processing all arrived data and must wait for the network
transfer from the paused node to arrive. Figure 2.2 shows the cumulative distribution of the shuffle flows’ finish times. At each moment there could be 1 to 16 concurrent shuffle flows incoming to or outgoing from one node, because there are four other nodes and each node runs up to four tasks at a time. The average size of those shuffle flows is 5.93MB and the standard deviation is 0.47MB. Note that the nodes have gigabit Ethernet connections. Ideally the flows should complete within 48-768 milliseconds. However, the CDF shows that around 20% of the flows finish in more than 1000 milliseconds. Among the slow flows, 2.6% of them finish in more than 5000 milliseconds. These slow flows suffered from garbage collection interruptions, either on the sender side or the receiver side. Figure 2.3 further shows the CPU utilization of one node during the reduce stage. The low CPU utilization periods indicate that the CPU is waiting for incoming shuffle flows; the use of CPU resources is thus highly inefficient.

2.4 Doesn’t “do not do X” solve the problem?

Do not use automatic memory management: Removing automatic memory management from cluster data processing frameworks is not a viable option. Automatic memory management is favored by developers because it frees them from the burden of manual memory management and fatal memory operation bugs such as dangling pointers, double-free errors, and memory leaks. Without automatic memory management, arguably those highly valuable cluster data processing frameworks such as [2,6–11] and the whole industry surrounding them might not even exist!

Do not use a tracing garbage collector: Among the two fundamental types of garbage collection strategies, tracing and reference counting, tracing is used in practice for multiple reasons. Reference counting incurs a significant space overhead
since it needs to associate a counter to every object, and this overhead exacerbates memory pressure problems. It also incurs a high performance overhead since it requires the *atomic* increment/decrement of the counter each time an object is referenced/dereferenced; this overhead exists regardless of memory pressure. Another source of performance overhead is that reference cycles must be detected by the system or else dead objects cannot be freed correctly. In sum, reference counting brings a new set of problems that together have potentially even more negative effects on system performance than tracing. A far more practical approach is to seek a solution within the tracing garbage collection framework.

**Do not let network transfer threads share the same heap with task execution threads:** If network transfer threads reside in a separate process to isolate the effect of garbage collection on network transfer performance, there will be a high overhead associated with passing data across the process boundary between the data producing/consuming process and the network transfer process. This isolation also does not help when the data producer process is stalled, thereby the network transfer process must wait for the data anyway. Moreover, as we will show in Section 3.1.2, even without network data shuffle, the garbage collection cost of the reduce stage is still far too high. Thus, isolating network transfer threads does not solve the fundamental problem.
Chapter 3

Leaky buffer

3.1 Leaky buffer abstraction

In seeking a solution to the problem, we have focused on addressing precondition 1, i.e. reducing the number of persistent objects in memory. We have also focused on designing a solution that is simple to grasp by developers and general in the following important sense:

- It makes no assumption on the type of processing that is performed on the data. As a result, the solution has to reside on the consumer side of the data, and has to be customizable according to the processing being performed.

- It makes no assumption on whether the data come from another node over the network or from the local storage.

- It is applicable whenever the task does not require the processing of all incoming data at once, thereby opening an opportunity for reducing the number of persistent objects in memory.

Our solution is a programming abstraction called the leaky buffer. In the rest of the thesis, we first present the ideas behind this leaky buffer abstraction, followed by a presentation of the concrete use case specific implementations, and experimental results to demonstrate the leaky buffer abstraction’s effectiveness.
Figure 3.1: An illustration of the leaky buffer abstraction

An illustration of the leaky buffer abstraction. Control information is extracted from incoming data by a program (step 1) and leaked to the processing logic (step 2) in order to allow logical data structures necessary for describing the data to be constructed (step 3). Data contents, however, are kept inside the leaky buffer until they are needed.

3.1.1 High level ideas

Figure 3.1 illustrates the leaky buffer abstraction. There are three high level ideas behind it:

- **Distinguishing control information from data contents**
  The leaky buffer abstraction explicitly models incoming data as a mixture of control information and data contents. Furthermore, the control information is the subset of data that the task needs to process the data contents. For example, a task that sorts key-value pairs according to the keys requires the keys as the control information. To extract the control information, a leaky buffer implementation must define the control information and understand the
data format.

- **Programmable control information extraction**

  Programmability is innate to the leaky buffer abstraction. Depending on the complexity of the control information, the program associated with a leaky buffer either simply selects the corresponding control information from the input stream, or partially processes the stream to infer the control information. For instance, in a Java application that processes key-value pairs, the byte-stream of key-value pairs is usually encoded in the conventional form of key1-value1-key2-value2-key3-value3... and so forth. Look one level deeper, the bytes of keys and values conform to the Java serializer format, so that Java objects can be reconstructed from these raw bytes. The control information could be the key objects themselves. In this case, the leaky buffer needs to invoke the deserializer to extract the key objects as control information.

- **Leakage of control information**

  The leaky buffer leaks a flexible amount of control information to the processing logic, while the data contents remain inside the leaky buffer until they are needed. The processing logic can still perform computation on the control information (e.g. construct the in-memory logical data structures for organizing the incoming data) that is necessary for data processing. Depending on the task, the data contents may need to be accessed eventually. For example, in a sorting task, the data contents need to be written to a sorted output file. In those cases, the processing logic constructs data structures that have references back to the buffered data, and fetches the data contents from the leaky buffer only on demand.
3.1.2 Conventional design vs. the leaky buffer abstraction

The conventional design shared by existing systems uses a minimal buffering mechanism for handling incoming data. For example, in Spark’s hash aggregation, the arrived shuffle data are consumed immediately by the deserializer and inserted into a hash table. This minimal buffering gives the highest degree of overlap between computation and network I/O – data could be transferred over the network while the CPU deserializes them and constructs the hash table. Overlapping computation and I/O is an accepted rule of thumb for gaining good utilization of compute and I/O resources. However, as we have shown in Section 2.1, there is a large potential cost in introducing high memory pressure, which conversely dominates the benefits of overlapping compute and I/O. When slow garbage collection events occur, dependent tasks start to wait on each other, leading to a performance collapse or a system failure.

One immediate idea is to decouple computation from network I/O by buffering all incoming data on disk before computation begins. However, in this design, the CPU is poorly utilized during the network transfer phase. Furthermore, severely slow garbage collection can still occur subsequently during the hash table construction phase. The reason is that the hash table construction process, which happens throughout the execution of the whole task, is very time consuming while the system is subjected to high memory pressure stemming from the large number of persistent hash table objects. During this long period of time, garbage collection events can be slow and ineffective. To prove this point, we ran an experiment using only one worker node on the workload described in Section 4.1.3. That is, all mappers and reducers run on this one worker node. Therefore, by design, all the incoming data to a reducer are from the local disk. Even though all the data is locally buffered before the start
of the hash table construction, 40.7% of the reduce stage run time is still taken by garbage collection, severely degrading performance.

In contrast, using the leaky buffer abstraction, a much better outcome can be achieved. In the hash aggregation scenario, the leaky buffer is programmed to leak the key objects as control information to the processing logic and buffer value objects in their original serialized format in large byte arrays. Consequently, the value objects are not materialized, which greatly reduces the number of persistent objects in memory, and avoids precondition 1 for performance collapse. Furthermore, CPU resources are still used efficiently – the leaked key objects are used immediately to construct a partial hash table. The partially constructed hash table can be used for performing lookup operations just like a normal hash table, and the buffered value objects are only materialized when it becomes necessary. In the next two sections, we present how the leaky buffer abstraction can be applied in use case specific implementations, and quantify its benefits through experiments.

### 3.2 Implementation

We have implemented the leaky buffer abstraction in two Spark reduce operations: `groupByKey` and `join`, with about 700 lines of code in Spark and about 300 lines of code in the Kryo serialization library that Spark depends on. The `groupByKey` operation is to group the values that have the same key together. It is typically one of the first steps in processing unordered data. The `join` operation is to join two tables together based on the same primary key, and it is very common in datatable processing.
3.2.1 groupByKey

The implementation of groupByKey in original Spark

In the original Spark implementation, the execution logic of groupByKey is:

- In the map task, the output key-value pairs are written to MapperOutputFiles in the serialized format as key1-value1-key2-value2-key3-value3...

- At the beginning of the reduce task, the worker fetches MapperOutputFiles from other worker nodes and deserializes each key-value pair in each MapperOutputFile as it arrives (step 1 in Figure 3.2).

- The worker matches the key object in the hash table and appends the value object to the ArrayBuffer corresponding to that key (step 2 and 3 in Figure 3.2).

- The above step is repeated until all key-value pairs have been processed.

- At the end of the reduce task, the iterator iterates the hash table and pass the results to the next stage of the job.

Figure 3.2: Original Spark’s groupByKey implementation. An ArrayBuffer is the Scala counterpart of a Java ArrayList; “bytes of objects” denotes the bytes of serialized JVM objects.
In summary, during the reduce stage, the Spark executor maintains a list of value objects for each key in the hash table. These objects persist through the reduce stage, as illustrated by the numerous JVM objects in Figure 3.2.

**Leaky buffer implementation of groupByKey**

Following the leaky buffer abstraction, we regard the key objects as the control information and the value objects as the data contents. Upon receiving the Mapper-OutputFiles in a reduce task, the leaky buffer invokes the deserializer to recover key objects from the raw bytes to JVM objects, and leaks them as the control information for hash table construction. To buffer the value objects as data contents, the leaky buffer needs to know the byte lengths of those serialized value objects in the incoming data stream to be able to copy the bytes to the leaky buffer. An efficient implementation is to make a format change to the list of key-value pairs in a Mapper-OutputFile in order to indicate the byte lengths of value objects without deserializing them. To do this, we insert an *int* field in the middle of each key-value pair. This *int* indicates the byte length of the following value object. The format thus becomes

```
key1-length1-value1-key2-length2-value2-key3-length3-value3...
```

During the map task, a file writer writes the resulting key-value pairs to a Map-
perOutputFile. To implement the above format, when the writer finishes writing the bytes of the key, it moves 4 bytes forward to the file stream and then writes the bytes of the value, while saving the byte length of the value. After writing the bytes of the value, the writer inserts the length into the 4 reserved bytes.

In the reduce task, the leaky buffer implementation has the following execution logic:

- At the beginning of the reduce task, the worker fetches MapperOutputFiles from the other worker nodes.
- The worker reads and deserializes the next incoming key object from the MapperOutputFile (step 1 in Figure 3.3).
- The worker matches the key object in the hash table. Note that the ArrayBuffer in the hash table of the original Spark is changed to a BufferedOutputStream (an expandable byte array) (step 2 in Figure 3.3).
- The worker reads the next 4 bytes from the MapperOutputFile, which indicates the byte length, L, of the following value object, and copy the next L bytes from the MapperOutputFile to the BufferedOutputStream associated with the matched key (step 3 in Figure 3.3).
- The above 3 steps are repeated until all key-value pairs have been processed.
- At the end of the reduce task, the iterator iterates all the key objects in the hash table, and for each key, the iterator invokes the deserializer to deserialize the bytes of all the value objects in the associated BufferedOutputStream, and returns the list of values in the same data structure and format as the original Spark.
A comparison between Figure 3.2 and Figure 3.3 shows that the original Spark maintains numerous JVM objects in the hash table, while the leaky buffer implementation maintains the bytes of those value objects in a much fewer number of byte arrays. This significantly reduces the number of JVM objects in the heap during most of the task execution time, and thus relieves the memory pressure.

3.2.2 join

The implementation of join in original Spark

The execution logic of join is very similar to that of groupByKey. These two reduce operations use the same hash table implementation. The difference is that the join operation inserts two lists of key-value pairs (two data tables) to the hash table, so that the result will be the lists of primary keys associated with values from both data tables.

Leaky buffer implementation of join

![Figure 3.4: Leaky buffer implementation of join](image)

The leaky buffer implementation of join is very similar to that of groupByKey.
However, different from the case of `groupByKey`, where one key can be associated with a large number of values, in the case of `join`, usually there are only two values associated with one key in the hash table, because the primary keys in each data table are unique and two key-value pairs, each from its own data table, contribute to the two values for each key in the hash table. Considering this, the implementation of `join` is different in that, we write all the serialized value bytes into one large byte array, and only insert the index of the value bytes in that large byte array into the associated array of `int` of size two as Figure 3.4 shows (note that this index array can be expanded on demand if there are more than two indices to be inserted). At the end of the reduce task, we use these indices to reference and deserialize the value objects to return the results.

This implementation of `join` has its advantage. The observation is that expanding the size of a byte array is expensive because it incurs memory allocation and copy. In the `join` operation, since the common case is that there are two values associated with each key, the size of an index array is essentially fixed. Therefore, we choose to consolidate all the bytes of the value objects into one large byte array and expand only this byte array when necessary. This idea is not as applicable to the `groupByKey` case since there can be a large number of value objects associated with each key; thus an index array would still have to be expanded dynamically.

### 3.2.3 Alternative implementation

To know the byte lengths of serialized value objects, an alternative implementation is to keep the original MapperOutputFile format but parse the serialized bytes to learn their lengths. In this alternative, we implement a specialized deserializer that copies the bytes of the value object on the fly while deserializing it. This specialized
deserializer writes the current byte of the object being deserialized to a byte array until reaching the last byte of the object, and thus we have the bytes of the object ready in that byte array. Note that we still need to deserialize those bytes of the object at the very end of the task. Thus, this alternative requires one more round of deserialization of the value objects, but avoids the format change that causes additional shuffle traffic.
Chapter 4

Evaluation

4.1 Evaluation

This section presents evaluation results to show that the leaky buffer abstraction can:

- Achieve consistently good performance across a large range of DMRs and improve the reduce stage performance by up to 507% under the same DMR (Section 4.1.3 and 4.1.4).
- Avoid task execution failures (Section 4.1.4).
- Save memory usage by up to 87.5% (Section 4.1.5).
- Reduce garbage collection cost (Section 4.1.5).
- Improve shuffle flow finish time (Section 4.1.5).
- Scales well to various input sizes (Section 4.1.6).

4.1.1 Experiment setup

All of the experiments in this section are conducted on Amazon EC2. We use five m1.xlarge EC2 instances as Spark workers, each with 4 virtual cores and 15GB of memory. The Spark version is 1.0.2, and the Java runtime version is openjdk-7-7u71. The amount of executor memory per worker is set to 6GB in all experiments (except the one in Section 4.1.5) to emulate an environment where the rest of the memory
is reserved for Spark data caching. Each worker can concurrently run 4 tasks. We set the Spark disk spill option to false so that the entire reduce task execution is in-memory. To further eliminate the effect of disk I/O on experiment results, we do not write the final results of jobs to the disk.

4.1.2 Workload description

The realistic workload we use to evaluate the leaky buffer comes from the Berkeley Big Data Benchmark [18], which is drawn from the workload studied by [19, 20]. We use this data set to evaluate the performance of both the original Spark and the leaky buffer implementations of two Spark reduce operations groupByKey and join.

The groupByKey experiments in Section 4.1.3 use the Uservisits table from the data set. The join experiments in Section 4.1.4 use both the Uservisits table and the Rankings table. The schema of the two tables can be found in [19].

Besides the realistic workload, we generate an artificial workload in the form of key-value pairs. The artificial workload enables us to control different variables such as the number of key-value pairs, the length of the values, the input file size, etc., and thus we can demonstrate the scalability of the leaky buffer under a variety of inputs.

4.1.3 Leaky buffer on realistic workload: groupByKey

In each of the following five use scenarios, we extract two columns from the Uservisits table, so that one column is the list of keys and the other is the list of values, and run groupByKey over this list of key-value pairs such that the values are grouped by the keys. The captions of Figure 4.1, 4.2, 4.3, 4.4, and 4.5 represent the column names of the key-value pairs. Each scenario represents a case where the user applies groupByKey to analyze interesting facts from the Uservisits table. For example, in
the ip-countrycode scenario, where we group countrycodes by prefixes of IP addresses (i.e. subnets), the result data set reflects the affiliation of IP subnets to different countries.

The input file size of the Spark job in each scenario ranges from 11GB to 18GB. The job has a map stage and a reduce stage, and each stage consists of numerous tasks. In the reduce stage, we vary the number of tasks to get different task sizes and thus different DMRs to evaluate with different levels of memory pressure. The number of reduce tasks and the corresponding per-reduce-task DMRs are indicated
on the x-axis of Figures 4.1, 4.2, 4.3, 4.4, and 4.5. The maximum number of tasks being evaluated is 800; an even larger number of tasks will give an unreasonably small DMR. We report the reduce stage run time in above figures as the performance metric.

From the results of the original Spark, we can observe that the reduce stage performance collapses as the number of tasks becomes smaller and the DMR becomes higher. In the day-keyword scenario in Figure 4.4 a seemingly generous DMR of 0.26 with 40 tasks already causes a performance collapse of 631% compared to a DMR
of 0.017 with 600 tasks. In other scenarios where the number of tasks is 40, the original Spark also has a serious performance collapse up to a few hundred percents. In the case of 60 tasks, the original Spark has a less serious but still considerable performance degradation.

In all cases with a small number of tasks, using the leaky buffer achieves a significant performance improvement over the original Spark. When the number of tasks is 40, the performance improvement is 401% in the day-keyword scenario and 303% in the ip-keyword scenario. The performance of leaky buffer is consistently good in all cases, and noticeably better than the original Spark in most of the cases. When the number of tasks is large (above 400) and the DMR is very small (less than 0.02), a performance sweet spot for the original Spark can be achieved and it marginally outperforms the leaky buffer. This is because there is very little memory pressure, while the leaky buffer incurs some memory copy overhead. However, the sweet spot for the original Spark is workload dependent and must be determined manually by trial and error for each job. Furthermore, a very large number of tasks may lead to other performance problems such as I/O fragmentation and scheduling inefficiencies.
4.1.4 Leaky buffer on realistic workload: join

We evaluate the join operation with three scenarios. In each scenario, we join the Rankings table to the specific columns from the Uservisits table using the webpage URL as the primary key in order to analyze the interesting correlation between ranks of webpages and facts from the Uservisits table. For example, the rank-revenue scenario in Figure 4.6 gives insights on whether higher ranked webpages generate more revenue.

The results in Figure 4.6, 4.7, and 4.8 show that the original Spark must use 80
or more tasks to avoid a failure. For cases with 40 and 60 tasks, the task executors of the original Spark throw `OutOfMemoryError` and the reduce stage fails after retries, despite the fact that 60 tasks represent only a moderately aggressive DMR of around 0.50. The leaky buffer can finish the job in all cases, even in the case of 40 tasks with a high DMR of 0.75.

In the case with 80 tasks, the original Spark has a high reduce stage run time in all three scenarios, the worse of which is 711 seconds in Figure [4.6] while the leaky buffer can improve the performance by 507% in the rank-revenue scenario, 174% in the rank-countrycode scenario, and 178% in the rank-duration scenario. In all other cases, the leaky buffer still achieves substantial performance improvements.

4.1.5 Other benefits of the leaky buffer

This section shows other benefits of the leaky buffer using the ip-countrycode scenario from Section [4.1.3]
Figure 4.9 : Reduce stage run time with different memory sizes

Memory usage reduction

Figure 4.9 shows the reduce stage run time for the original Spark and the leaky buffer with 100 reduce tasks with different amount of executor memory per worker node, and thus different DMRs. The original Spark fails to finish the job when the amount of memory is 1GB, and has a very long run time of 154 seconds when it is 2GB. In comparison, the leaky buffer has consistently better performance than the original Spark in all cases. When the amount of memory is 8GB, the original Spark achieves the best performance, while the leaky buffer achieves even better performance with only 1GB of memory, leading to a memory saving of 87.5%.

Garbage collection cost reduction

Figure 4.10 shows the portion of time spent in garbage collection during the reduce stage. For the original Spark, the garbage collection time decreases as the number of tasks increases. For the leaky buffer, the garbage collection time is consistently low. We can conclude that a main saving in reduce stage run time of the leaky buffer results from the reduction of the garbage collection time. Less time spent in garbage collection in the leaky buffer leads to less stop-the-world interruptions, and
consequently reduces shuffle flows’ finish times as will be shown next.

**Shuffle flows’ finish times reduction**

Because severe stop-the-world garbage collection events happen during the reduce stage of the original Spark, the JVM is frequently paused and there is a high chance that incoming and outgoing shuffle flows on workers are interrupted. Figure 4.11 shows CDF plots of shuffle flows’ finish times for different number of tasks for both the original Spark and the leaky buffer. At each moment in the shuffle, there could be 1 to 16 concurrent shuffle flows incoming to or outgoing from one node, because there are four other nodes and each node runs up to four tasks at a time. When the number of tasks is 40, the sizes of the shuffle flows fall within the range of 8-10MB. Note that the nodes have gigabit Ethernet connections. Ideally the flows should complete within 64-800 milliseconds. However, for the original Spark, in the case of 40 tasks, about half of the flows complete in more than 1000ms and around 5% of the flows complete in more than 5000ms. In cases with 60, 80, 100, and 200 tasks, the shuffle flows’ finish times are also far from ideal. In contrast, using the leaky buffer can improve the shuffle flows’ finish times significantly.
4.1.6 Leaky buffer on artificial workload: measuring scalability

To completely evaluate the performance improvement of the leaky buffer and explore other dimensions of the workload, we generate an artificial workload and design experiments to measure scalability. Each experiment is run with the default number of tasks chosen by the original Spark. Note that Spark sets the default number of map and reduce tasks to the number of blocks in the input file, and this default number of reduce tasks gives a DMR of around 0.04.
This artificial data set enables us to evaluate scalability with different input sizes and formats. The data set format is a list of key-value pairs. The keys are five digit numbers following a uniform distribution, and the values are various lengths of characters. To scale the artificial data set, we either increase the number of key-value pairs or increase the lengths of the value characters.

The left graph on Figure 4.12 shows the result of scaling the number of key-value pairs. As expected, the reduce stage run time for both the original Spark and the leaky buffer scales linearly with the input size. The leaky buffer achieves nearly a 100% performance improvement in all cases.

The right graph in Figure 4.12 shows the result of scaling up the input size by increasing the lengths of the values. The performance improvement of the leaky buffer diminishes as the values become longer. The reason is that, as the input size grows with the lengths of the values, the default number of tasks increases. Because the total number of key-value pairs is fixed, the number of key-value pairs per task decreases, so there is fewer number of objects in memory per task, and thus there is less memory pressure and less performance improvement is achieved by the leaky
4.1.7 Overhead analysis

The leaky buffer incurs overhead that mainly affects two aspects. First, in the map stage of the job, under the first implementation, the mapper needs to write the lengths of the values, which is a 4-byte \texttt{int} type, for each key-value pair. This overhead turns out to be negligible. For instance, in the ip-countrycode scenario and the rank-countrycode scenario, the percentage difference between the map stage run time for the original Spark and the leaky buffer ranges from -1.9\% to 1.3\% with a mean difference of less than 0.1\%.

Another source of overhead is the increase in network shuffle traffic due to the aforementioned 4-byte lengths of the values. For the ip-countrycode case in Section 4.1.3 the total shuffle traffic is 4.39GB for the original Spark and 4.71GB for the leaky buffer. The network transfer cost for the extra 0.32GB over 5 nodes is small compared to the overall benefits of the leaky buffer. The performance improvements of the leaky buffer on the reduce stage reported previously already factor in the overhead of the extra shuffle traffic.

4.1.8 Alternative implementation evaluation

Section 3.2.3 describes an alternative implementation of the leaky buffer. Figure 4.13 compares the first implementation against this alternative implementation, using the ip-countrycode scenario from Section 4.1.3. The result shows that the alternative implementation has a little longer reduce stage run time but still achieve consistent performance. The slowdown comes from the fact that this alternative implementation needs to deserialize the values in key-value pairs twice – the first time is to get the
Figure 4.13 : Comparing two implementations of the leaky buffer

size of the serialized value object, and the second time is to actually deserialize those bytes into a JVM object.
4.2 Related work

Managing data in memory: For building in-memory data processing frameworks, there is existing work on better managing data in memory. Tachyon [21] is a memory-centric distributed file system. Tachyon outperforms Spark’s native caching because it caches data outside of the JVM heap and thus bypasses JVM’s memory management to reduce the garbage collection cost.

Similar objectives also exist in datatable analytics application design. There are systems that propose datatable columnar storage, in which a datatable is stored on per column basis [22, 23]. The columnar format gracefully improves storage space efficiency as well as memory usage efficiency. The designer of Shark [10] recognizes the performance impact from garbage collection and leverages the columnar format in in-memory datatable caching to reduce the number of JVM objects for faster garbage collection.

Compared to the above practices, the leaky buffer has a similar objective of reducing the number of persistent objects but addresses an orthogonal problem with a different approach. Tachyon and columnar storage help to manage data caching that is rather static, while the leaky buffer tackles the memory pressure problem from dynamic in-memory data structures during data processing and computation. These efforts together provide insights for building better in-memory computing applications.

Resource allocation and remote memory: There has been research work on resource allocation for MapReduce systems [24–26]. These efforts mainly focus on resource allocation at the machine or compute slot level, and do not specifically tune memory allocation to address memory pressure. DRF [27] is an allocation scheme for multiple resource types including CPU and memory. It requires a job’s resource
demands \textit{a priori} for allocation, but it is hard for the user to predict the potential memory pressure in a job and state an optimal memory demand that both achieves good performance and avoids wasting resources. To the best of our knowledge, there is not yet any resource allocation approach that can address the memory pressure problem for in-memory data processing frameworks, and this may represent a direction for future research.

Remote memory access has been a widely used technique in computer systems research [28–30]. SpongeFiles [31] leverages the remote memory capability to mitigate data skew in MapReduce tasks by spilling excessive, static data chunks to idle memory on remote worker nodes rather than on local disk, and thus offloads local memory pressure to a larger shared memory pool. This approach, however, cannot resolve the memory pressure problem in reduce tasks due to the prohibitively high latency cost of operating on dynamic data structures (e.g. hashtable) in remote memory.

\textbf{Engineering efforts on memory tuning and shuffle redesign:} There are guidelines from engineering experiences on tuning data processing frameworks. The Spark website provides a tuning guide [32]; there are also tuning guides for Hadoop [33, 34]. Those guidelines include instructions for tuning JVM options such as young and old generation sizes, garbage collector types, number of parallel garbage collector threads, etc. From our experience on Spark, tuning JVM options and switching among JVM garbage collectors lead to comparable performance to using the defaults. When the memory pressure is high, frequent full garbage collection events are still inevitable.

The Spark tuning guide [32] provides various instructions for tuning different Spark configurations. The most memory pressure related instruction is to increase the number of map and reduce tasks to avoid the \texttt{OutOfMemoryError}. This simple
practice helps avoid the `OutOfMemoryError`, but the resulting low DMR leads to resource under-utilization, I/O fragmentation, and/or inefficient task scheduling.

Engineers from Cloudera recognize the slowdowns and pauses in Spark reduce tasks are caused by memory pressure, and propose a redesign using a full sort-based shuffle, which merges numerous on-disk sorted blocks from map tasks during shuffle [35]. By moving part of the shuffle execution from memory to disk, it does help to reduce memory pressure, but this approach is a throwback to the traditional Hadoop style on-disk sort-based shuffle, which contradicts the in-memory computing paradigm that Spark aims to leverage.
Chapter 5

Conclusion

5.1 Conclusion

We have made two contributions in the thesis. Firstly, we have diagnosed the memory pressure problem in cluster data processing frameworks and identified three preconditions for performance collapse and failure. It reveals that the memory pressure problem can lead not only to local performance degradations, but also to slow shuffle data transfers and cluster-wide poor CPU utilization, both of which amplify the negative performance effects. Secondly, we have proposed a novel programming abstraction called the \textit{leaky buffer} that is highly effective in addressing the memory pressure problem in numerous experimental use cases – it drastically reduces the occurrence of memory-related failures, improves performance by up to 507\% and reduces memory usage by up to 87.5\%. Furthermore, the leaky buffer abstraction is simple to grasp and has wide applicability since many data processing tasks do not require the processing of all incoming data at once.
Bibliography


apache-spark-its-a-double/ 2015.