This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2546909, IEEE Transactions on Parallel and Distributed Systems

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

Zhaolei Liu T. S. Eugene Ng
Rice University

***Abstract*—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 paper 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 a leaky buffer enabled hashtable in Spark, and we believe it is also able to substitute the hashtable that performs similar hash aggregation operations in any other programs or data processing frameworks. 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%.

I. Introduction

When MapReduce [24] 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 [12, 13].

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 revisited [36]. 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 [8, 1, 5, 4, 40, 41] 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 [41, 4]. 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 [16, 17, 18, 19]. Take Spark as an example, our experimental results leave us initially quite perplexed (details found in Section V) – 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.[1] 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 10 in Section V-C 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 13 in Section V-D shows.

Our work 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, 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.

A commonality among the aforementioned in-memory data processing frameworks is that they are implemented using programming languages (e.g. Java, Scala) that support au-

[1] 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.

tomatic memory management[2]. 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. The data processing task generates high number of objects extremely frequently, while most of those objects persist in memory for nearly the entire lifetime of the task, resulting in frequent garbage collections. Moreover, the mutual-dependence of those concurrent tasks exacerbates the negative impact of 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 the above conditions and are therefore susceptible to performance collapse and failure – In such jobs, the reducers perform a data shuffle over the network, and the incoming data are processed into a hash table containing a very large number of key-value objects that persist until the end of the job.

The solution we propose is a novel abstraction called *leaky buffer*. It dramatically reduces the number of in-memory objects 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 a leaky buffer enabled hashtable in Spark, and we believe it is also able to substitute the hashtable that performs similar hash aggregation operations in any other programs or data processing frameworks such as Apache Pig [3] and Apache Tez [6]. 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.

II. 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. Among different garbage collection types, tracing is the most common type, so much so that "garbage collection" often refers to tracing garbage collection [21]. 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.

A. 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.

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. The parallel garbage collector in the Java virtual machine (JVM) uses generational garbage collection strategy [7]. The short-lived objects live in the young generation in the heap and get collected efficiently when they die.

However, 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, while more frequent object creation will trigger more frequent garbage collection events as well. In this situation, garbage collection 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, 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.

[2] 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!

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2546909, IEEE Transactions on Parallel and Distributed Systems

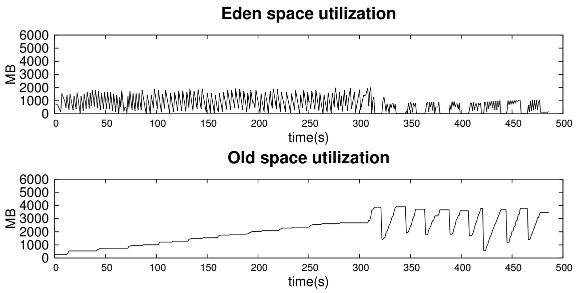


Fig. 1. Heap utilization of one node during an in-memory hash aggregation job. Short-lived objects reside in the so called eden space, while persistent objects reside in the so called old space.

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.

B. 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. 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 V-A. It uses a workload as described in Section V-H, with $2 * 10^9$ 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 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 for short-lived objects, 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, the task executor builds a large in-memory hash table to store incoming key-value pairs, so that a large amount of objects are created rapidly and are pushed from the eden space to the old space (observe the multiple rounds of 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.

The shuffle operation inherent in this job satisfies precondition 3. Full garbage collection brings a stop-the-world pause to the whole node and network transfers. Figure 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. The average size of those shuffle flows is 5.93MB with the standard deviation of 0.47MB. With gigabit Ethernet, ideally the flows should complete within 48-768ms. However, the CDF shows that around 20% of the flows finish in more than 1000ms , while 2.6% of them finish in more than 5000ms. Figure 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.

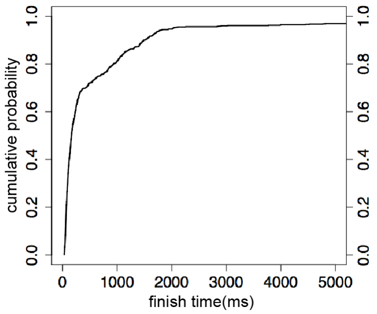


Fig. 2. CDF of shuffle flows' finish times of an in-memory hash aggregation job

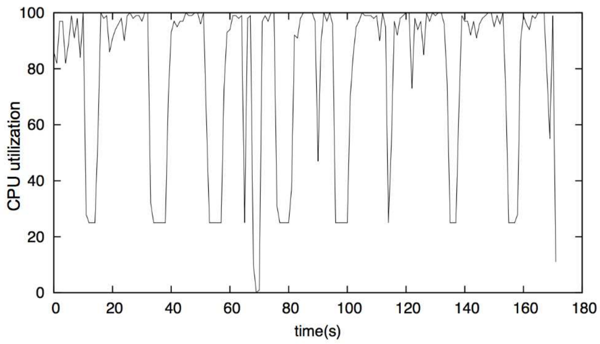


Fig. 3. CPU utilization of one node during the reduce stage of an in-memory hash aggregation job

C. 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, 8, 1, 5, 4, 40, 41] and the whole industry surrounding them might not even exist![3]

[3] The reader may observe that Google is widely known to use C/C++ for infrastructure software development. However, the important point is, for each company like Google, there are many more companies like Facebook, Amazon, LinkedIn that use Java for infrastructure software.

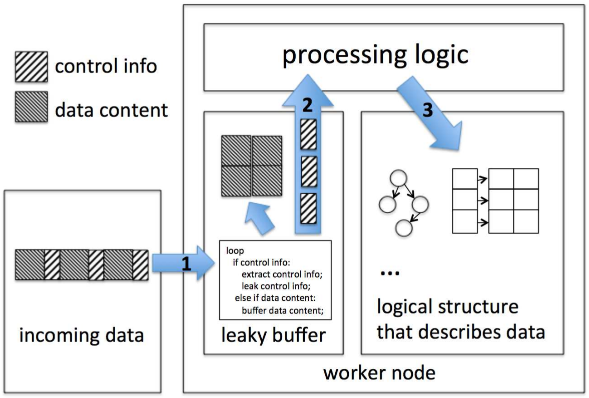
Use a non-tracing garbage collector: Among the two fundamental types of garbage collection strategies, tracing and reference counting, tracing is used in practice for multiple reasons. Reference counting incurs a significant space overhead since it needs to associate a counter to every object. It also incurs a high performance overhead since it requires the *atomic* increment/decrement of the counter each time an object is referenced/dereferenced. 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.

Tuning the garbage collector: To investigate whether tuning the garbage collector can relieve memory pressure, we have performed extensive tuning and testing on the original Spark in the ip-countrycode scenario described in Section V-B. We have done this for two latest Java garbage collectors – the parallel garbage collector of Java 7 and the G1 garbage collector of Java 8.[4] The result shows that it is extremely hard to obtain any performance improvement through tuning (and in most cases performance degraded) compared to simply using the Java 7 parallel garbage collector with its default settings.

With default settings, the G1 garbage collector produces a 46% worse reduce stage runtime than the parallel garbage collector. We have explored different settings for two key parameters for the G1 garbage collector, namely `InitiatingHeapOccupancyPercent` for controlling the heap occupancy level at which the marking of garbage is triggered, and `ConcGCThreads` for controlling the number of concurrent garbage collection threads [22, 14]. We varied `InitiatingHeapOccupancyPercent` from 25% to 65% in 10% increments. The best result is 39% worse than that of the parallel collector. We varied `ConcGCThreads` from 2 to 16. The best result is 41% worse than that of the parallel collector.

For the parallel garbage collector, we have explored `newRatio` for controlling the old generation to young generation size ratio, and `ParallelGCThreads` for controlling the number of garbage collection threads. We varied `newRatio` from 0.5 to 4. No setting was able to outperform the default ratio of 2, and on average performance was 2% worse. We varied `ParallelGCThreads` from 2 to 16. No setting was able to outperform the default value of 4, and with 2 threads, performance was 32% worse.

Do not let network transfer threads share the same heap with task execution threads: The network transfer threads can reside in a separate process in order to isolate the effect of garbage collection on network transfer performance, while the memory copy overhead can be minimized. However, this isolation does not help when the data producer process is stalled, thereby the network transfer process must wait for the data anyway. Moreover, 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.



```
loop
  if control info:
    extract control info;
    leak control info;
  else if data content:
    buffer data content;
```

Fig. 4. 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.

III. Leaky buffer

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*. Figure 4 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, or alternatively we could define the control information as both the keys and the locations of the corresponding value, depending on the implementation. 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.

[4] We do not consider the serial garbage collector which is designed for single core processor, nor the CMS garbage collector which is replaced by the G1 garbage collector.

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.

A part of the benefits of the leaky buffer stems from the principle of lazy evaluation, a well-known system design strategy. Specifically, the leaky buffer lazily fetches and processes the data content to avoid unnecessary early commitment of resources. However, another part of its benefits stems from its proactive processing of the control information and construction of the in-memory data structure. This design of the leaky buffer trades a reasonable amount of CPU resources, used by the extra buffering and data processing logic, for much better memory utilization and garbage collection efficiency.

IV. Leaky buffer enabled hashtable

We implement leaky buffer on hashtable and replace the original hashtable in Spark. The leaky buffer enabled hashtable has 300 lines of code. This leaky buffer enabled hashtable is able to improve the performance of hash aggregation operations in Spark, and it is also able to substitute the hashtable that performs similar hash aggregation operations in any other programs or data processing frameworks. For example, Apache Pig [3] supports hash aggregation on the map side, and its hashtable can be replaced by the leaky buffer enabled hashtable. As another example, in Apache Tez [6], the leaky buffer enabled hashtable can be directly used as a processor on a vertex to perform the hash aggregation operation. Despite the general applicability of the leaky buffer enabled hashtable, we mainly evaluate it with Spark in Section V.

A. Hash aggregation in original Spark

In the original Spark, the execution logic of hash aggregation is:

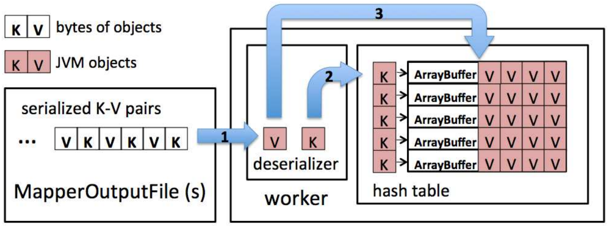


Fig. 5. Original Spark's hashtable implementation. An ArrayBuffer is the Scala counterpart of a Java ArrayList; "bytes of objects" denotes the bytes of serialized JVM objects.

- 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 5).
- 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 5).
- 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 thee results to the next stage of the job.

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 5.

B. Leaky buffer enabled hashtable implementation

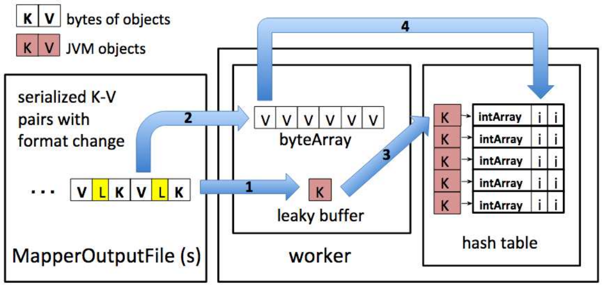


Fig. 6. Leaky buffer enabled hashtable. The figure is simplified in that it does not show the way to handle hash collision, which is the same as in the original Spark hashtable implementation. "i" in the hashtable denotes the index of the bytes of value in the byteArray (leaky buffer).

Following the abstraction in figure 4 of section III, we implemented the leaky buffer on hashtable as figure 6 shows. When the leaky buffer processes the incoming data stream of key-value pairs, it extracts the key objects as part of the control information, and buffers the bytes of value objects as the data content. The leaky buffer leaks both the key objects and the indexes of the values in the buffer as full control information to the data processing logic to construct the hashtable.

Specifically, upon receiving the MapperOutputFiles in a reduce task, the leaky buffer invokes the deserializer to recover

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2546909, IEEE Transactions on Parallel and Distributed Systems

key objects from the raw bytes to JVM objects. In order 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 MapperOutputFile 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 MapperOutputFile. 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.

The execution logic of hash aggregation with leaky buffer enabled hashtable is:

- At the beginning of the reduce task, the worker fetches MapperOutputFiles from the other worker nodes.
- The leaky buffer reads and deserializes the next incoming key object from the MapperOutputFile (step 1 in Figure 6).
- The leaky buffer reads the next 4 bytes from the MapperOutputFile, which indicates the byte length, L, of the following value object, copies the next L bytes from the MapperOutputFile to the byteArray in leaky buffer, and records the position of that value object in the byteArray as i (step 2 in Figure 6). When the byteArray is full, it is extended by allocating a new byteArray with double size and copying the content of the old byteArray to the first half of the new byteArray.
- The leaky buffer leaks the key object to the worker to construct the hashtable. The worker matches the key object in the hash table, and writes the index of the value, i, to the intArray associated with its key (step 3 and 4 in Figure 6).
- 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. For each key, the iterator invokes the deserializer to deserialize the bytes of all the value objects in the associated with that key, using the indexes in the intArray, and returns the list of values as deserialized objects.

A comparison between Figure 5 and Figure 6 shows that the original Spark hashtable maintains numerous JVM objects, while the leaky buffer enabled hashtable maintains the bytes of those value objects in one large byteArray. This significantly reduces the number of JVM objects in the heap during most of the task execution time, and thus relieves the memory pressure.

C. Optimization of leaky buffer enabled hashtable for large load factor

The implementation described above works well for the scenario where the hashtable has relatively small load factors and large hashtable size so that the size of the intArrays are fixed or rarely needs to be extended. For instance, in the `join` operation, 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. The implementation above is able to effectively consolidate the bytes of all value objects into one large byteArray, and thus relieves memory pressure.

However, when the hashtable has a large load factor, such as in the case of `groupByKey` operation where many values that have the same key are grouped together, maintaining many large intArrays incurs space overhead, and extending those intArrays as well as the large byteArray is expensive. An optimization is to remove the large byteArray and replace each intArray with a byteArray, so that the bytes of value objects are directly written to the byteArray associated with its key in the hashtable. This optimization avoids the space overhead incurred by the intArrays, and the double and copy mechanism is able to effectively extend the size of the byteArrays based on the fact that those byteArrays could become very large.

D. 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, with about 100 lines of code change on the Kryo serialization library that Spark depends on. 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.

V. 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 V-C and V-D).
- Avoid task execution failures (Section V-D).
- Save memory usage by up to 87.5% (Section V-E).
- Reduce garbage collection cost (Section V-F).
- Improve shuffle flow finish time (Section V-G).
- Scales well to various input sizes (Section V-H).

A. 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

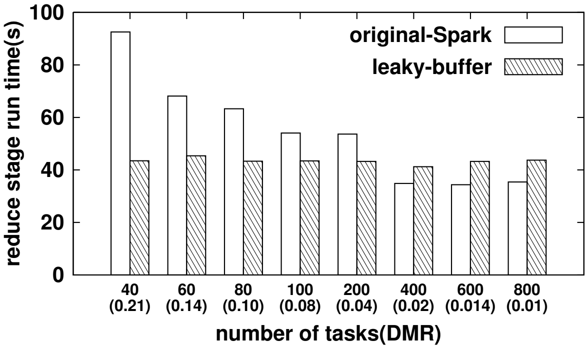


Fig. 7. ip-countrycode

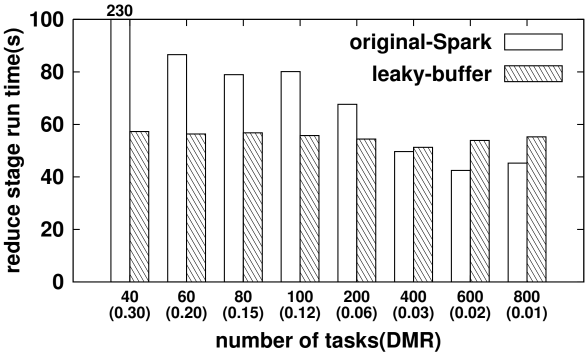


Fig. 8. ip-keyword

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 V-E) 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.

B. Workload description

The realistic workload we use to evaluate the leaky buffer comes from the Berkeley Big Data Benchmark [15], which is drawn from the workload studied by [34, 27]. We use this data set to evaluate the performance of both the original Spark and the leaky buffer on two reduce operations `groupByKey` and `join`, which are the two most representative hash aggregation operations in Mapreduce systems. 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.

The `groupByKey` experiments in Section V-C use the Uservisits table from the data set. The `join` experiments in Section V-D use both the Uservisits table and the Rankings table. The schema of the two tables can be found in [34].

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.

C. 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 7, 8, 9, 10, and 11 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.

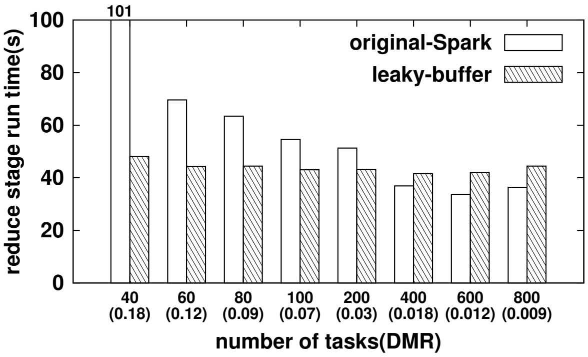


Fig. 9. day-countrycode

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 7, 8, 9, 10, and 11. 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 10, 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 per-

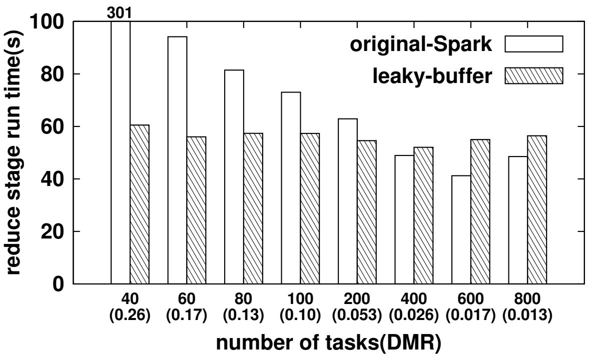


Fig. 10. day-keyword

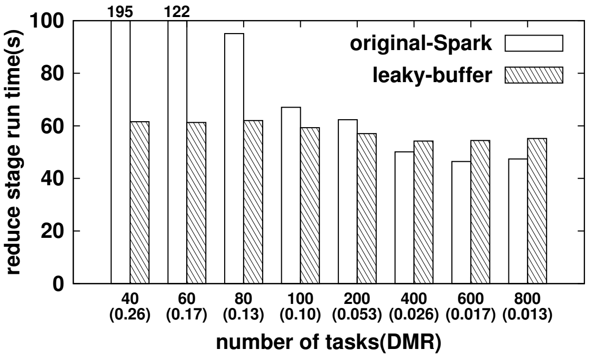


Fig. 11. month-keyword

formance 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. In the cases where the DMR is extremely low (less than 0.02) and the number of tasks is high, there is already very little memory pressure for original Spark such that leaky buffer cannot improve the performance by relieving memory pressure, while it has slightly worse performance due to the extra memory copy required in the design.

D. 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 12 gives insights on whether higher ranked webpages generate more revenue.

The results in Figure 12, 13, and 14 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.

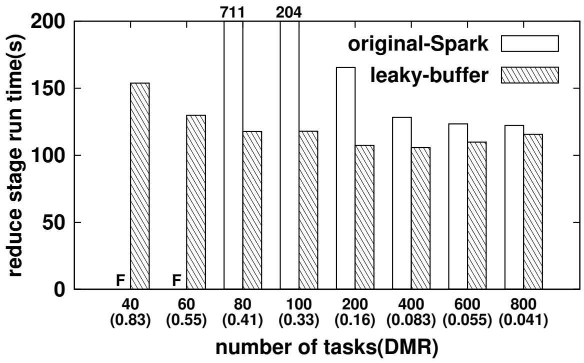


Fig. 12. rank-revenue

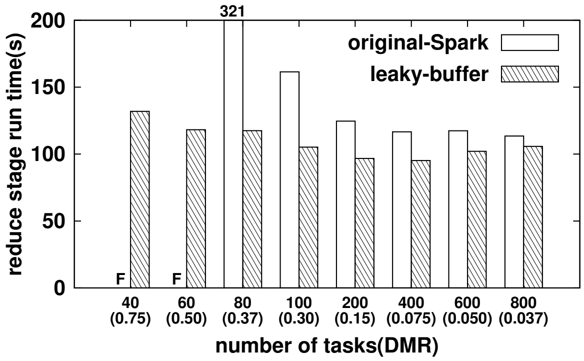


Fig. 13. rank-countrycode

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 12, 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.

E. Tuning vs leaky buffer: which is better?

The previous results show that the original Spark must be carefully tuned to achieve acceptable performance. Unfortunately, performance tuning in today's systems is manual and tedious; efficient, automatic performance tuning remains an open problem. Spark uses the number of HDFS blocks of the input dataset as the default number of tasks but allows this setting to be manually changed. Another popular framework Tez [35], which shares some similar functionalities as Spark but belongs to the Hadoop and Yarn ecosystem, relies on the manual configuration of a parameter called `DESIRED_TASK_INPUT_SIZE` to determine the number of tasks. Since the level of memory pressure is highly sensitive to the availability of cluster resources and workload characteristics, the user must tediously perform numerous tests at different number-of-task settings for each workload and for each cluster environment in order to determine the corresponding performance sweet spot. In contrast, leaky buffer does not require such manual tuning to achieve good performance.

Even if the user were able to manually determine the performance sweet spot for a particular workload, the leaky

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2016.2546909, IEEE Transactions on Parallel and Distributed Systems

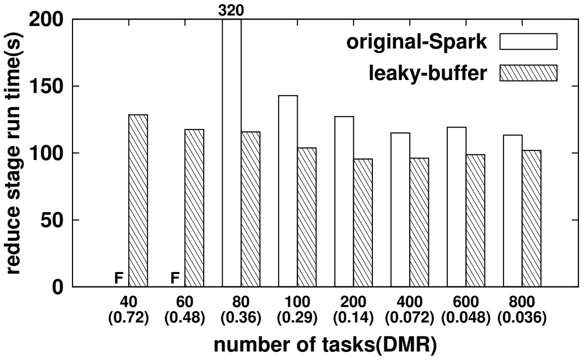


Fig. 14. rank-duration

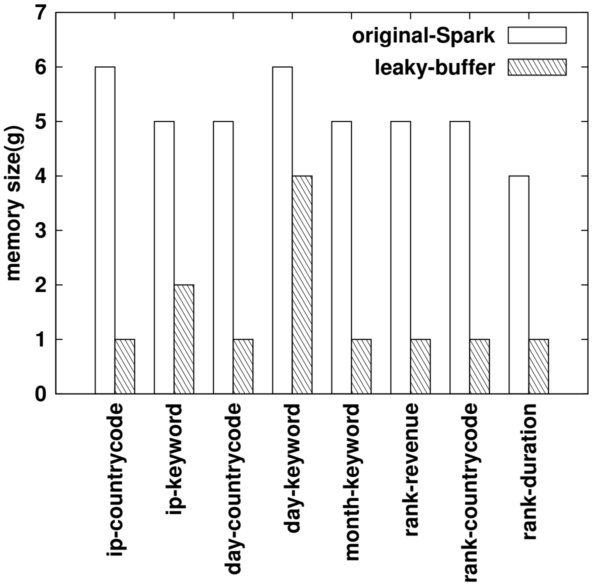


Fig. 15. Minimum memory size to achieve comparable performance to the best performing spots in Figure 7, Figure 8, Figure 9, Figure 10, Figure 11, Figure 12, Figure 13, and Figure 14.

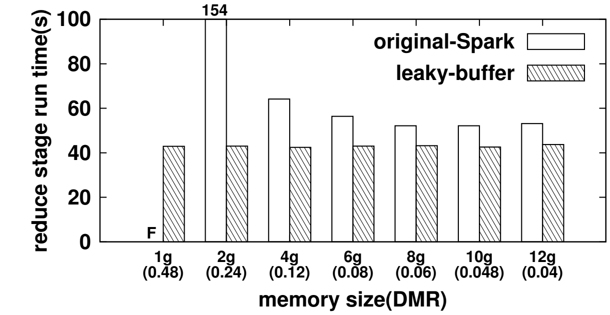


Fig. 16. Reduce stage run time of ip-countrycode scenario with 100 tasks and various memory sizes

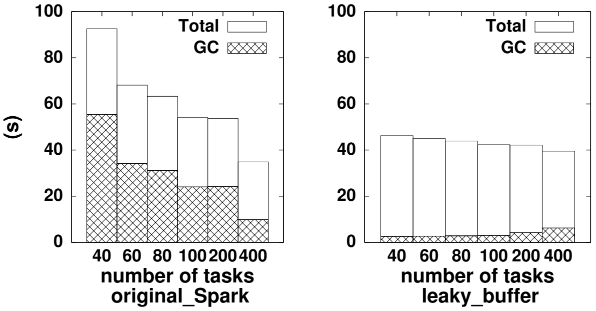


Fig. 17. Garbage collection time spent in the reduce stage

buffer still has one predominant advantage that, it requires less amount of memory than original Spark to archive comparable performance. Figure 15 shows the minimum amount of memory that original Spark and leaky buffer require to achieve comparable performance (no more than 5% worse) to using 6G memory and the optimal number of tasks as found in previous experiments. Leaky buffer is able to save 33% - 87.5% in memory usage in the 8 scenarios. In the day-keyword scenario, the improvement is still 33% despite the fact that it is a particularly memory intensive scenario as there are more keys (days) in the hashtable per task and the value (keyword) could be very long. Leaky buffer is also able to save memory with non-optimal number of tasks. Figure 16 shows the reduce time in the ip-countrycode scenario with 100 tasks and varying amount of memory. Leaky buffer can reduce memory usage from 8GB to 1GB while achieving even better performance.

With leaky buffer's lower memory requirement, the same cluster hardware can handle larger datasets and/or more simultaneous jobs. Alternatively, the extra memory can be utilized by RAMDisk or in-memory storage frameworks such as Tachyon [29] to further improve the system performance.

F. Garbage collection cost reduction

Figure 17 shows the portion of time spent in garbage collection during the reduce stage in the ip-countrycode scenario. 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.

G. 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 18 shows CDF plots of shuffle flows' finish times for different number of tasks for both the original Spark and the leaky buffer using the ip-countrycode scenario. 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

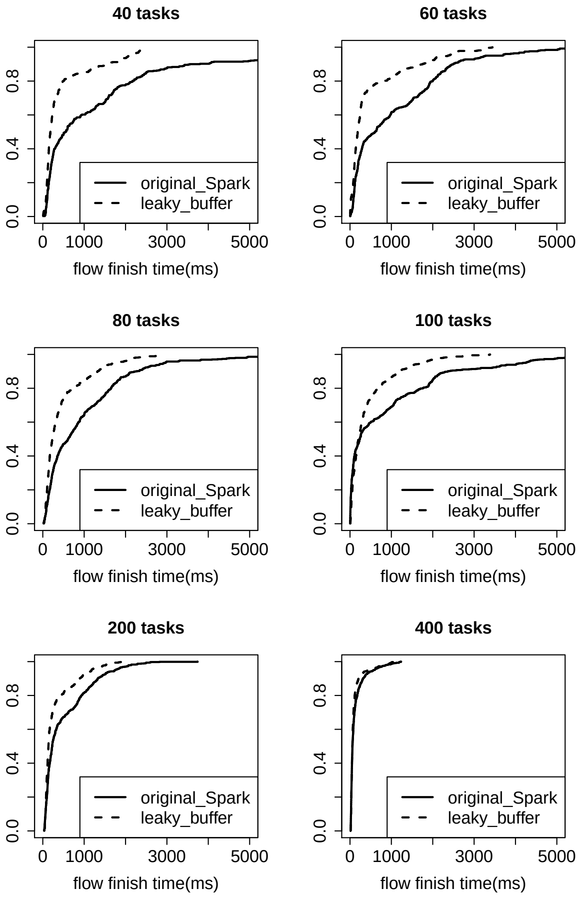


Fig. 18. CDFs of shuffle flows' finish times under different scenarios

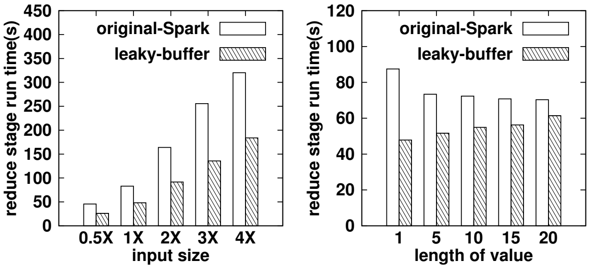


Fig. 19. Artificial data set evaluation. 1X on the left graph denotes 10^9 key-value pairs of total size 6.8GB; the experiment on the right graph uses a fixed number of 10^9 key-value pairs with different lengths of the value characters.

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.

H. *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 19 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 19 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 buffer.

I. *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 `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 V-C, 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.

J. *Alternative implementation evaluation*

Section IV-D describes an alternative implementation of the leaky buffer. Figure 20 compares the first implementation against this alternative implementation, using the ip-countrycode scenario from Section V-C. 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 size of the serialized value object, and the second time is to actually deserialize those bytes into a JVM object.

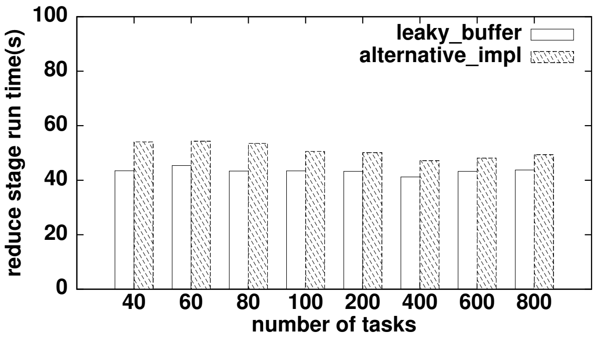


Fig. 20. Comparing two implementations of the leaky buffer

VI. Related work

Managing data in memory: For building in-memory data processing frameworks, there is existing work on better managing data in memory. Tachyon [29] 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 [37, 30]. The columnar format gracefully improves storage space efficiency as well as memory usage efficiency. The designer of Shark [40] 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.

Trash Day is a runtime system for coordinating the garbage collection activities among multiple worker nodes in cloud systems [31] to avoid workers from waiting on each other's garbage collection activities and thus improve overall performance. In contrast, the leaky buffer represents an orthogonal approach which focuses on relieving the memory pressure and eliminating the source of expensive garbage collection activities, instead of optimizing the garbage collection behavior. The leaky buffer and the Trash Day techniques can be employed simultaneously to maximize performance.

Resource allocation and remote memory: There has been research work on resource allocation for MapReduce systems [38, 28, 39]. 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 [26] is an allocation scheme for multiple resource types including CPU and memory. It requires a job's resource demands *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 [33, 23, 32]. SpongeFiles [25] 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.

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 [9]; there are also tuning guides for Hadoop [11, 10]. 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, JVM tuning cannot lead to any noticeable performance improvement as demonstrated in Section II-C.

The Spark tuning guide [9] provides various instructions for tuning different Spark configurations. The most memory pressure related instruction is to increase the number of map and reduce tasks. This simple practice helps relieve the memory pressure and avoid the `OutOfMemoryError`, but how to determine the right number of tasks remains as a challenge to the users as stated in Section V-E. Increasing the number of tasks also exposes the risk of resource under-utilization, I/O fragmentation, and/or inefficient task scheduling. In the ip-countrycode scenario, the reduce time with 2000 tasks is 9.6% longer than that with 400 tasks.

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 [20]. 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.

VII. Conclusion

We have made two contributions in this paper. 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 *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.

References

[1] Apache giraph. http://giraph.apache.org/.

[2] Apache hadoop. https://hadoop.apache.org/.

[3] Apache pig. https://pig.apache.org/.

[4] Apache spark. https://spark.apache.org/.

[5] Apache storm. https://storm.apache.org/.

[6] Apache tez. https://tez.apache.org/.

[7] Java se 6 hotspot[tm] virtual machine garbage collection tuning. http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html.

[8] Kognitio. http://kognitio.com/.

[9] Tunning spark. https://spark.apache.org/docs/1.2.0/tuning.html.

[10] Amd hadoop performance tuning guide. http://www.admin-magazine.com/HPC/Vendors/AMD/Whitepaper-Hadoop-Performance-Tuning-Guide, 2012.

[11] Java garbage collection characteristics and tuning guidelines for apache hadoop terasort workload. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/GarbageCollectionTuningforHadoopTeraSort1.pdf, 2012.

[12] Gartner says in-memory computing is racing towards mainstream adoption. http://www.gartner.com/newsroom/id/2405315, 2013.

[13] It revoluton: How in memory computing changes everything. http://www.forbes.com/sites/ciocentral/2013/03/08/it-revolution-how-in-memory-computing-changes-everything, 2013.

[14] Tips for tning the garbage first garbage collector. http://www.infoq.com/articles/tuning-tips-G1-GC, 2013.

[15] Berkeley big data benchmark. https://amplab.cs.berkeley.edu/benchmark/, 2014.

[16] Spark gc overhead limit exceeded. http://apache-spark-user-list.1001560.n3.nabble.com/GC-overhead-limit-exceeded-td3349.html, 2014.

[17] Spark groupby outofmemory. http://apache-spark-user-list.1001560.n3.nabble.com/Understanding-RDD-GroupBy-OutOfMemory-Exceptions-td11427.html, 2014.

[18] Spark job failures talk. http://www.slideshare.net/SandyRyza/spark-job-failures-talk, 2014.

[19] Storm consumes 100% memory. http://qnalist.com/questions/5004962/storm-topology-consumes-100-of-memory, 2014.

[20] Improving ort performance in apache spark: It is a double. http://blog.cloudera.com/blog/2015/01/improving-sort-performance-in-apache-spark-its-a-double/, 2015.

[21] Tracing garbage collection. http://en.wikipedia.org/wiki/Tracing_garbage_collection, 2015.

[22] Tuning jav garbage collection for spark applications. https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html, 2015.

[23] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[25] Khaled Elmeleegy, Christopher Olston, and Benjamin Reed. Spongefiles: Mitigating data skew in mapreduce using distributed memory. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 551–562, New York, NY, USA, 2014. ACM.

[26] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association.

[27] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 41–51, March 2010.

[28] Gunho Lee, Niraj Tolia, Parthasarathy Ranganathan, and Randy H. Katz. Topology-aware resource allocation for data-intensive workloads. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, pages 1–6, New York, NY, USA, 2010. ACM.

[29] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[30] Yuting Lin, Divyakant Agrawal, Chun Chen, Beng Chin Ooi, and Sai Wu. Llama: Leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 961–972, New York, NY, USA, 2011. ACM.

[31] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[32] Evangelos P. Markatos and George Dramitinos. Implementation of a reliable remote memory pager. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[33] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramclouds: Scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[34] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[35] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1357–1369, New York, NY, USA, 2015. ACM.

[36] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, August 2012.

[37] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pages 553–564. VLDB Endowment, 2005.

[38] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Aria: Automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244, New York, NY, USA, 2011. ACM.

[39] D. Warneke and Odej Kao. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *Parallel and Distributed Systems, IEEE Transactions on*, 22(6):985–997, June 2011.

[40] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

T. S. Eugene Ng is an Associate Professor of Computer Science at Rice University. He is a recipient of a NSF CAREER Award (2005) and an Alfred P. Sloan Fellowship (2009). He received a Ph.D. in Computer Science from Carnegie Mellon University in 2003. His research interest lies in developing new network models, network architectures, and holistic networked systems that enable a robust and manageable network infrastructure.

Zhaolei Liu was a master student at Rice University and currently works for Facebook.inc. He received a Bachelor of Science in Computer Science (2010-2013) and a Master of Science in Computer Science (2013-2015) both from Rice University. His research interest has been in computer networking, big data processing frameworks, and distributed systems.