Freeing Compute Caches from Serialization and Garbage Collection in Managed Big Data Analytics

Iacovos G. Kolokasis[1]
Insitute of Computer Science, FORTH
kolokasis@ics.forth.gr

Giannos Evdorou[1]
Insitute of Computer Science, FORTH
evdorou@ics.forth.gr

Anastasios Papagiannis
Insitute of Computer Science, FORTH
apapag@ics.forth.gr

Foivos Zakkak
Red Hat. Inc
fzakkak@redhat.com

Christos Kozanitis
Insitute of Computer Science, FORTH
kozanitis@ics.forth.com

Shoaib Akram
Australian National University
shoaib.akram@anu.edu.au

Polyvios Pratikakis[1]
Insitute of Computer Science, FORTH
polyvios@ics.forth.com

Angelos Bilas[1]
Insitute of Computer Science, FORTH
bilas@ics.forth.gr

arXiv:2111.10589v1 [cs.PL] 20 Nov 2021

Abstract

Managed analytics frameworks (e.g., Spark) cache intermediate results in memory (on-heap) or storage devices (off-heap) to avoid costly recomputations, especially in graph processing. As datasets grow, on-heap caching requires more memory for long-lived objects, resulting in high garbage collection (GC) overhead. On the other hand, off-heap caching moves cached objects on the storage device, reducing GC overhead, but at the cost of serialization and deserialization (S/D).

In this work, we propose *TeraHeap*, a novel approach for providing large analytics caches. *TeraHeap* uses two heaps within the JVM (1) a garbage-collected heap for ordinary Spark objects and (2) a large heap memory-mapped over fast storage devices for cached objects. *TeraHeap* eliminates both S/D and GC over cached data without imposing any language restrictions. We implement *TeraHeap* in Oracle's Java runtime (OpenJDK-1.8). We use five popular, memory-intensive graph analytics workloads to understand S/D and GC overheads and evaluate *TeraHeap*. *TeraHeap* improves total execution time compared to state-of-the-art Apache Spark configurations by up to 72% and 81% for NVMe SSD and non-volatile memory, respectively. Furthermore, TeraCache requires 8× less DRAM capacity to provide performance comparable or higher than native Spark. This paper opens up emerging memory and storage devices for practical use in scalable analytics caching.

1 Introduction

Analytics workloads typically perform iterative computations over large datasets until a convergence condition is met. Each iteration produces new transformations of previously computed data, e.g., with *map* operation. Such intermediate results can either be cached for later use or recomputed when they are needed. Compute caches improve performance by up to two orders of magnitude (100×) compared to recomputing intermediate results [59]. For this reason, contemporary managed big-data analytics frameworks widely used by industry, such as Spark [61], employ a *compute cache* to store intermediate results and avoid expensive recomputation.

As datasets expand in size [42,44], they demand larger compute caches [55]. The size of compute caches is proportional to the data footprint during computation, including intermediate results. Our evaluation shows that in Spark, the size of cached data footprint is typically up to 20× larger than the input dataset. Thus, compute caches need to grow to sizes larger than the DRAM available to Spark applications, often by more than one order of magnitude.

The contemporary practice in Spark is to place the compute cache partly in memory (on-heap) and partly on a storage device (off-heap). On-heap caching increases garbage collection (GC) cost as it fills the heap with long-lived objects that are scanned at every GC cycle [53]. Our evaluation shows that with on-heap caching GC time reaches up to 79% of total execution time. Once the compute cache outgrows the JVM heap, Spark serializes cached objects and moves them to persistent storage, introducing serialization/deserialization (S/D) overhead. This overhead worsens as storage technology improves and the performance gap between the processor and memory narrows [35, 41, 49]. Thus, S/D is expensive and unfriendly to technology trends that dictate less data movement [34].

Prior efforts [3,6,7,22,27,29,32,33,36,37,47] either focus on reducing GC or S/D overheads. On the GC side, one line of prior works [22, 32, 33, 36, 37] uses runtime and compiler support and customized annotations to the application code for keeping long-lived objects in a region-based (off-heap) native memory. Unfortunately, these prior approaches (1) require changes to the application code, (2) support specialized objects only, and (3) do not address mitigating the S/D cost. On the other hand, recent work proposes to use non-volatile memory (NVM) to overcome the DRAM capacity limitations. More specifically, Panthera [47] uses NVM to scale on-heap caching in Spark by extending the managed heap over hybrid DRAM and NVM. Unfortunately, NVM-backed

[1]Also with the Computer Science Department, University of Crete

managed heaps exacerbate the GC overhead as traversals over cached data objects in NVM are extremely slow. Recently, new libraries [3, 6, 7] attempt to make S/D more efficient, but they still result in high S/D overhead for emerging big data frameworks. Other efforts [27, 29] propose optimizations for reducing S/D but demand custom hardware extensions without mitigating the GC overhead.

We propose *TeraHeap*, which extends the JVM to use a second, memory-mapped heap (H2) for cached objects and eliminates both GC and S/D overheads for cached intermediate results. *TeraHeap* allows direct, transparent access to cached objects, both for reads and updates, eliminating the cost of transferring objects between on-heap and off-heap as in the S/D approach. Our design introduces four essential functions:

First, *TeraHeap* allows moving arbitrary objects in H2 and ensures the correctness of the Java memory model. In the existing Spark caching approaches, Spark moves in the off-heap compute cache, serializable objects only [38]. Therefore, *TeraHeap* offers higher flexibility and enables various policies to populate H2, minimizing references between H2 and the garbage collected heap (H1). To achieve this, we find that an appropriate policy is to compute each cached data object's transitive closure and then move to H2 only the non-transient fields (serializable fields).

Second, *TeraHeap* tracks references form H1 to H2 (forward) and from from H2 to H1 (backward) without extensive cost, using additional card tables. To reduce GC cost, *TeraHeap* fences the garbage collector from following forward references to avoid traversing objects in H2. In addition, *TeraHeap* has to deal with backward references from H2 to H1, as objects in H1 are garbage collected and change locations during both minor and major GC. For this reason, *TeraHeap* needs to keep track of updates during JVM post-barriers. *TeraHeap* properly handles barriers in interpreted Java code and the C1 and C2 just-in-time (JIT) compilers.

Third, *TeraHeap* leverages the Spark *persist()* operation to inform the JVM about which data objects to move from H1 to H2. Thus, *TeraHeap* requires no modifications to existing Spark application code compared to prior work [36, 37] that requires extra annotations or static analysis [47].

Fourth, *TeraHeap* exploits the *grouped life-time* property of cached data objects that tend to leave the cache at the same time when the application invokes *unpersist()*. *TeraHeap* organizes H2 on the storage device in regions to facilitate bulk-free operations instead of reclaiming individual objects, reducing the I/O traffic to the device. *TeraHeap* maintains reachability information from external objects to regions and reclaims entire regions when they become unreachable.

We implement *TeraHeap* in Oracle's production Java runtime (OpenJDK-1.8), extending the Parallel Scavenge garbage collector. We also modify the interpreter and the C1 and C2 Just-in-Time (JIT) compilers to support updates in *TeraHeap* during application (mutator) execution.

We evaluate *TeraHeap* in Spark using five broadly deployed memory-intensive graph analytics workloads. *TeraHeap* can efficiently use different types of devices, including block-addressable NVMe and byte-addressable NVM. Our evaluation uses datasets that demand compute caches several times larger than the dataset. We find that *TeraHeap* improves overall performance by up to 72% for NVMe SSD and up to 81% for NVM devices compared to state-of-the-art Spark configurations. Regarding DRAM needs, *TeraHeap* consumes 8× less DRAM capacity for similar or better performance than native Spark.

Overall, this paper makes the following contributions:

1. We investigate and show that cached objects in Spark are not accessed for extended intervals of time, indicating that we can place these objects outside the managed heap for extended periods of execution.

2. We design *TeraHeap* that eliminates S/D for cached data by placing arbitrary objects in a large memory-mapped heap, providing the JVM direct access to cached data using load/store operations. Our design precludes expensive lookup mechanisms as the OS performs the translation.

3. We propose the first dual-heap design (memory and storage) that eliminates GC for the second (storage) heap, avoiding expensive scans of cached data objects while ensuring the correctness of the Java memory model by handling references across heaps.

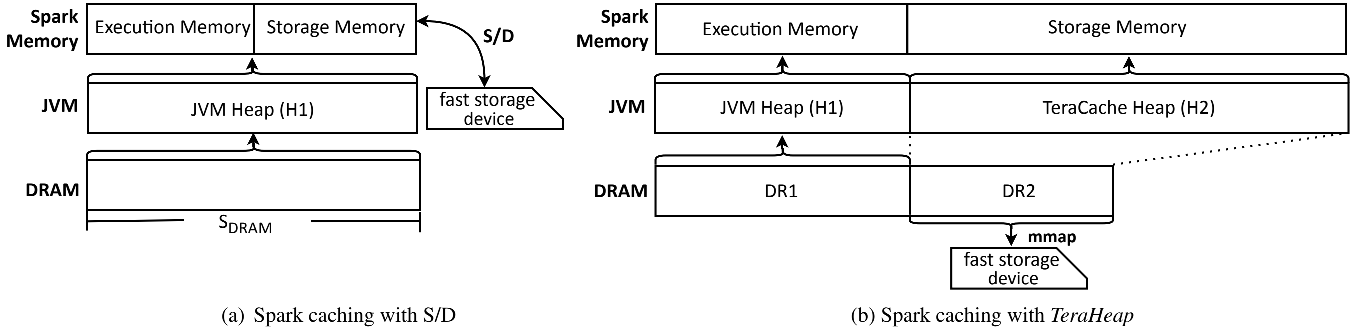
4. We propose a bulk-free mechanism (appropriate for Spark cached objects) that reclaims entire regions of cached data.

2 Background and Motivation

Spark [61] is a widely used framework for large-scale analytics. It consists of a driver process and multiple executor processes. The main data abstraction in Spark is the Resilient Distributed Dataset (RDD) [60]. At a low level, RDDs are read-only collections of similarly typed objects partitioned across a cluster. Spark programs consist of a set of transformations and actions over RDDs. Spark evaluates RDDs lazily; transformations are not evaluated until an action is performed on some RDD. Actions trigger the execution of Spark jobs, which compute all RDDs in their lineage.

To avoid time-consuming [51] recalculation of commonly used RDDs across different jobs, Spark offers developers the flexibility of caching RDDs via its *persist()* API [45]. Users can persist RDDs in different storage levels: memory in a deserialized form, disk in a serialized form, or both.

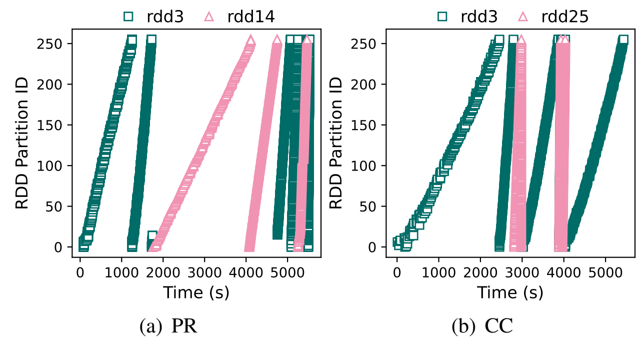
Spark executors run in JVM instances and allocate memory on heap, which resides in DRAM. A Spark executor logically divides its memory into two main spaces (Figure 1(a)):



(a) Spark caching with S/D (b) Spark caching with *TeraHeap*

Figure 1: (a) Vanilla Spark: off-memory caching via S/D. (b) *TeraHeap*: on-heap RDD cache over a memory-mapped fast storage device $S_{DR1} + S_{DR2} = S_{DRAM}$ ($\mathbf{S_{DRAM}}$=DRAM size).

(1) execution memory for computation (shuffle, joins, sorts, and aggregation operations) (2) storage memory (compute cache) for caching. Spark initially reserves 60% of the heap as storage memory and uses the rest for execution. Then, it dynamically adjusts the boundary between execution and storage memory according to the usage of each space. Today, Spark users commonly use both memory and disk for caching. When an RDD partition does not fit in storage memory, Spark serializes (e.g., using Kryo [7]) and moves another RDD partition to a storage device, using an LRU policy.



(a) PR (b) CC

Figure 2: Analysis of cached RDD accesses in PageRank (PR) and Connected Components (CC).

2.1 GC Overhead of Cached RDDs

Two factors increase GC overhead in an executor JVM. First, the combined volume of intermediate results is large, often several times larger than the input dataset [51], incurring high cost during each GC cycle to scan the heap. We measure that the combined volume of cached RDDs in the application we use is 20× higher than the input dataset. Second, cached intermediate results (objects) exhibit long lifetimes [18,47,53] resulting in low return and frequent GC cycles as each GC cycle cannot free much space.

To illustrate the potential for eliminating GC overhead, Figure 2 shows the RDD access patterns for PageRank (PR) and Connected Components (CC). Each RDD has 256 partitions. The horizontal axis shows time (seconds), and the vertical axis shows the IDs of accessed RDD partitions (from 1 to maximum partition ID for each RDD). For each ID, the first dot marks the time when that partition is cached. Each subsequent dot represents access to this partition by the application.

We note that once Spark needs an RDD for computation, it tends to access all partitions of an RDD sequentially. For this reason, in Figure 2 RDD accesses form "lines." There also exist large intervals between accesses to RDD partitions. For example, in PR and CC, Spark accesses the first partition of RDD3, on its creation around $T = 0$ s but then again only after at least 1500 s (PR) and 2500 s (CC), respectively. We observe similar temporal gaps between accesses to partitions of RDD14 (PR) and RDD25 (CC). For certain RDDs, each partition is potentially accessed multiple times. For instance, for RDD25, each partition is accessed twice around $T = 3000$ s. However, there is still a significant period of inactivity (about 1000 s) until the next set of accesses around $T = 4000$ s. Occasionally, successive accesses to the same partition exhibit temporal locality because two jobs take turns to cache and materialize partitions. For example, at $T = 1100$ s, Job0 creates and caches the last partition of RDD3, and then Job1 materializes this partition at $T = 1200$ s. Overall, the time interval between accesses to each partition varies between 100-1500 s in both applications. This behavior motivates placing cached RDDs (unlike temporary, short-lived RDDs) on storage devices (off-heap) that offer high capacity.

Furthermore, all data objects in each partition have similar lifetime. When applications *unpersist* an RDD, Spark drops all RDD partitions from its cache. At this point, in most cases, no references exist to the corresponding JVM objects. This observation reveals an opportunity to reclaim cached objects en masse by organizing the compute cache in groups of data objects with similar lifetimes.

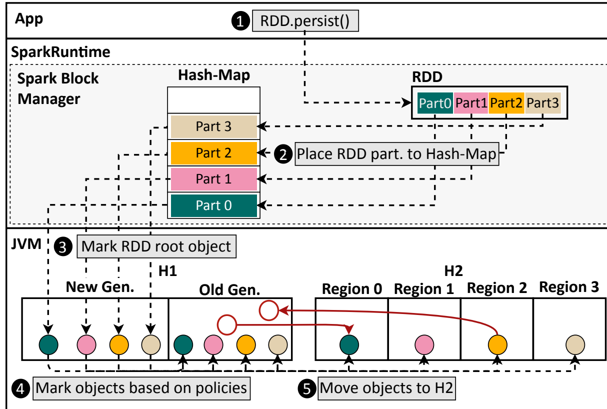


Figure 3: Management of cached objects in *TeraHeap*.

3 Design and Implementation

The main concept behind *TeraHeap* is to provide a separate custom-managed heap for storage memory and use the primary JVM heap as execution memory in Spark. Figure 1(b) shows the high-level design of *TeraHeap*, including the primary heap (H1) and the custom-managed heap (H2). Similar to the regular JVM heap, H1 resides in DRAM and is divided into two generations: young and old. Unlike H1, H2 is memory-mapped over a device to allow direct access via Java references and offer large capacity as the amount of cached data grows. *TeraHeap* requires no changes to the programming model and is fully transparent to Spark applications. It is implemented entirely in the JVM and exploits *persist()* hints from the Spark runtime to mark candidate objects and move them to H2.

Figure 3 shows the flow of Spark caching operations in *TeraHeap*. ❶ The application invokes *persist()* explicitly. ❷ The Spark block manager places the selected RDDs in the compute cache, a hash-map that contains all cached RDDs. The Spark block manager caches each partition independently, maintaining per-partition entries in the hash-map. ❸ *TeraHeap* offers a Java Native Interface (JNI) [31] call to the application layer that is called inside Spark's *persist()* operation. With *TeraHeap*, *persist()* only calls this JNI call and does not perform any other operations in Spark, essentially replacing the Spark block manager. Spark initially allocates all RDD objects in H1. ❹ In the JNI call, *TeraHeap* marks the per-partition root RDD object. ❺ Then, *TeraHeap* marks and moves objects to H2, based on a migration policy.

3.1 Eliminating S/D with Memory-mapped I/O

Separating H2 from H1 allows *TeraHeap* to handle the two heaps with different mechanisms. H1 remains a limited-size, garbage-collected heap. We place H1 in DRAM via anonymous memory mappings in Linux, as is the case with the regular JVM heap. Accesses to H1 (and GC) are not affected by the size or the technology of the H2 backing device.

Instead, H2 can grow significantly using fast, high-capacity storage devices. Fast NVMe SSD and NVM devices, as opposed to HDDs, are amenable to memory-mapped I/O (mmio), due to their high throughput and low latency for small request sizes (4KB) regardless of the access pattern [39]. For this reason, we design H2 as a mmap'd [8] heap to eliminate S/D and to allow using regular pointers to and from cached objects, without need of specialized lookup mechanism in existing JVM code. Thereby, H2 objects remain usable from any program without the need of Java application modifications.

TeraHeap is agnostic to storage device technology and can use (1) Fast NAND-Flash-based storage devices (e.g., NVMe SSDs) via the block-based mmio path of the OS, (2) Non-volatile memory (NVM) via a persistent memory (PMEM) abstraction, or even (3) DRAM, if available in large sizes, via anonymous memory mappings. However, we believe that using fast NVMe SSDs is the most realistic configuration as datasets grow. They provide high density (capacity) and low cost per bit compared to DRAM and NVM [57].

For this reason, we design *TeraHeap* to cope with relatively slow accesses to NVMe devices, as opposed to only faster accesses to NVM. Note that today, the JVM can already allocate its single object heap over a storage device using mmio without any application modifications. However, this does not suffice, as the entire heap is subject to GC overhead. For this reason, a mmap'd JVM heap results in worse performance compared to a combination of GC and S/D, as shown in Section 5. Instead, *TeraHeap* avoids GC traversals in H2, as we discuss next.

3.2 Tracking Cross-heap References to Avoid GC

To avoid GC traversals in H2, we need to identify all references from H1 to H2 (forward references) and from H2 to H1 (backward references), shown as solid arrows in Figure 3.

Forward references are relatively straightforward and require mainly fencing the garbage collector from crossing from H1 to H2. The garbage collector reclaims objects from the young generation during minor GC and the old generation during major GC. In both minor and major GC, the collector performs a breadth-first traversal (BFT) to mark live objects. The garbage collector checks to see if any of the references cross H1 to H2. If they do, the garbage collector stops marking such references to avoid scanning H2.

Tracking backward references is more complex. Both minor and major GC must be aware of backward references to identify live objects in H1. Unfortunately, scanning H2 to identify backward references incurs considerable overhead. To avoid scanning H2 during minor and major GC, we introduce three JVM extensions: (1) keeping track of modified

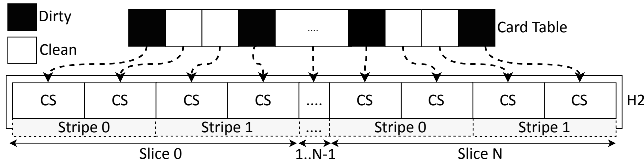


Figure 4: Design and organization of *TeraHeap* cards (CS=Card Segment).

objects in H2, (2) detecting backward references in modified H2 objects, and (3) taking backward references into account during H1 liveliness analysis.

We detect updates to H2 objects during application execution. We use a new card table (Figure 4) to keep track of H2 object writes. The original JVM uses a similar card table for detecting references from the old to the new generation [26, 48]. The H2 card table is a byte-array with one byte per fixed-size H2 segment similar to the JVM card table. We refer to these segments as card segments. The H2 card segment size does not need to match the size of JVM segments for the old generation. Segment size affects both metadata size and GC overhead (both minor and major). In particular, the overhead of scanning H2 cards during minor and major GC can be significant for a large H2. Increasing the H2 card segment size results in fewer cards and faster GC traversals. Our evaluation indicates that a size between 8-16 KB for H2 card segments works well, compared to the 512-byte card segments used by the JVM's old generation.

At first, we initialize all H2 cards as clean. When an application thread updates an H2 object, *TeraHeap* marks the corresponding card as *dirty*. To examine if the object belongs to H1 or H2, we use an additional range check in the post-write barrier used by the JVM after each object update. This range check selects the appropriate (H1 or H2) card table, which we then mark with the existing post-write barrier code. Updates may originate from either interpreted or just-in-time compiled methods with the C1 or C2 JVM compilers. For this reason, we extend the post-write barriers in each compilation level to support the marking of H2 cards. We evaluate the overhead of our modifications to post-write barriers using the DaCapo benchmark suite [15] and find it to be small, within 3% on average across all benchmarks.

An additional issue with the card table is parallel accesses from multiple threads. Minor GC is multithreaded, and all GC threads need to access the H2 card table during minor GC. To avoid contention between GC threads, similar to H1, we divide H2 into slices (Figure 4). Each slice contains a fixed number of fixed-size stripes (each stripe is by default 64K, so it consists of 128 card segments) equal to the number of minor GC threads. Within each slice, each GC thread processes the stripe with the same id. Therefore, each GC thread operates on the same stripe id in all slices of H2.

In the original JVM, dividing the work of scanning the cards among multiple GC threads results in the boundary cards (first and last card) of each stripe being accessed by two neighboring threads. To avoid synchronization between these two threads, the original JVM never marks boundary cards as clean. This means that if boundary cards become dirty, they will remain dirty throughout execution, and the corresponding card segments will always be scanned for objects that contain backward references. This is not as big of an issue for H1 for two reasons: (a) card segments are relatively small, by default 512 bytes. (b) scanning card segments that are placed in memory for backward pointers is relatively fast.

However, for H2, both of these factors introduce significant issues: (1) Card segments are larger to reduce the size of the card table, e.g., 8KB. Thus, if stripes remain at 64KB, then they consist of a small number of large cards, e.g., eight cards, with two of these being boundary cards. Therefore, a quarter of the card segments will be left dirty to always be scanned, creating high overhead. To reduce the number of boundary cards, we use a larger stripe size. Our evaluation indicates that a stripe size between 4-8 MB for H2 works well, compared to the 64 KB stripe size used by the JVM's old generation. (2) In addition, H2 segments are located on the storage device, which results in significantly higher overhead to scan its objects for backward pointers. For this reason, it is essential to ensure that only dirty segments are marked dirty. During the scanning phase of major GC we identify which objects moved to H2 have references to H1. Then, we mark the corresponding cards of these objects in H2 as dirty.

Once a dirty card is encountered by a GC thread, to identify if a modified object contains backward references, we iterate over all object combined in its segment. This suffices, as a minor GC also happens just before a major GC cycle. Scanning these objects may involve I/O if they are not placed in DR2 by mmio. If an object contains no backward references, we clear the corresponding card, otherwise, we push its backward references in a *backward reference stack*.

Finally, during marking phase of major GC, we prevent reclamation of H1 objects referred from H2 objects (backward references). We traverse the backward reference stack and mark referenced H1 objects as live. After H1 compaction, we use the backward reference stack to adjust all backward references to point to the new object locations in H1.

3.3 Populating H2 During Major GC

All objects in *TeraHeap* are initially allocated in H1, similar to the original JVM. Unlike Spark that can cache only RDDs [12], *TeraHeap*, being a transparent, JVM-level mechanism can move arbitrary JVM objects to H2. *TeraHeap* uses a configurable policy to select and mark objects for migration to H2. Then, the garbage collector moves all marked objects to H2. *TeraHeap* uses a new field (8 bytes) in the JVM object header (Figure 5) to mark candidate objects. Although it is possible to avoid this additional field, it may require addi-

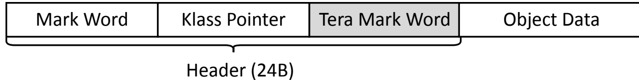


Figure 5: Object layout in Java, including the additional *TeraHeap* word in the object header.

tional metadata in the JVM and may increase minor GC time for book-keeping. For this reason, our current implementation uses the additional header field.

TeraHeap moves marked objects from H1 to H2 during the compaction phase of major GC. We extended standard compaction, where the garbage collector relocates all live objects, to relocate marked objects into H2. Relocation to H2 creates forward and backward references. Forward references are updated during pre-compaction similar to all other references; backward references need to be tracked by *TeraHeap*. To avoid scanning the fields of each object at this point, we merely mark the corresponding *TeraHeap* card as dirty. Therefore, the main overhead of *TeraHeap* for major GC is the actual transfer of objects from H1 to H2. Although H2 is memory-mapped and reads are always performed via mmio, writes to H2 (moving objects) can happen with different mechanisms. In our evaluation, we examine mmio and explicit asynchronous I/O for H1 to H2 transfers.

TeraHeap policy code for populating H2 merely decides which objects to mark. Although *TeraHeap* can place arbitrary objects in H2, ideally, H2 should contain objects that are: (1) long-lived, so high-yield in terms of GC overhead, (2) amenable to bulk free, so high-yield in terms of reclaimed space, and (3) intermittently accessed, so yield in large intervals without accesses them.

Eager, Non-Transient fields (ETR): In our work, we introduce the *ETR policy*, which transfers similar objects to H2 as would happen in the S/D approach. During S/D, the serializer traverses the object graph to identify all objects that need to be serialized, in RDDs that have been persisted. Similarly, ETR during the marking phase of major GC, identifies the transitive closure of each cached data object. In Java a field in a class can be marked with the *transient* modifier. When the object is deserialized, transient fields are initialized to a default/fixed value according to the serializer. Their value is not required to be part of the serialized object. For this reason, the serializer omits transient fields when serializing an object.

In the same way, our ETR policy during the marking phase of major GC, skips transient objects that are part of the closure for each marked object. The transitive closure includes arbitrary JVM objects from both the young and old generations of H1. In the end, ETR will move to H2 only objects of RDD partitions which are immutable, leaving as backward references all transient fields. Moving arbitrary objects to H2 might create new backward references because the application

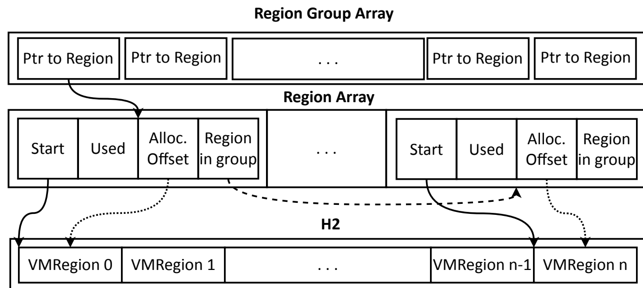


Figure 6: *TeraHeap* allocator overview.

updates some of these objects. Thus, ETR offers the flexibility to populate H2 with immutable objects, minimizing backward references to H1.

Note that *TeraHeap* handles any remaining backward and forward references between H2 and H1 for all policies. Therefore, marking objects is only a matter of performance (policy) and not a correctness mechanism. This approach allows for significant flexibility. Also, if it eventually becomes possible to identify object characteristics at runtime, the persist operation may not be necessary, rendering *TeraHeap* entirely transparent to higher layers that will be able to use a large JVM heap without providing any hints.

3.4 Freeing Space in H2

We design *TeraHeap* to reclaim cached objects in a bulk manner. To achieve this, we organize H2 in virtual memory as a region-based heap. Figure 6 shows the arrangement of H2 in virtual memory. Similar to prior work [22,36], *TeraHeap* frees objects in the same region in one batch. This region-based management precludes traversing and reclaiming individual objects, limiting GC overhead.

TeraHeap places objects of each RDD partition in the same H2 region until it exhausts the region space (Figure 3). We observe typical RDD partition sizes of several tens of megabytes (MB) and as big as 64 MB. Based on that, we size regions to be few GB. We ensure that each region contains objects of the same RDD partition as follows. We patch the Spark block manager to perform a JNI call (as part of the persist operation) which sets the partition id in the *TeraHeap* header word of the corresponding JVM object. During the calculation of the transitive closure we mark each object with the same partition id. Finally, we move all objects related to a partition id to the same region(s) in H2.

To reclaim a region, *TeraHeap* ensures that all objects in the region are not referenced from other objects. To identify such regions, *TeraHeap* tracks two types of region references.

References from H1 to H2: For tracking H1 references, *TeraHeap* uses a USED bit in the per-region metadata in DRAM. The garbage collector clears these bits at the begin-

(GB)	NVMe Server DRAM	NVM Server DRAM	Dataset on DISK	Foot-print D	Spark -SD H= DR1	Spark -MO H= NVM	TC H1= DR1	Other DR2
PR	80	192	32	641	64	1024	64	12+4
CC	84	192	32	679	68	1024	68	12+4
SSSP	58	192	32	420	42	650	42	12+4
SVD	40	192	2	240	24	500	24	12+4
TR	80	192	2	37	64	64	64	12+4

Table 1: Spark-SD, Spark-MO and *TeraHeap* configurations for each workload for setups with NVMe SSD and NVM. *Other* refers to the DRAM reserved for the OS and the Spark driver.

ning of each marking phase (major GC in H1). Upon encountering a reference to an object in H2, the collector sets the corresponding region bit. Thus, the USED bit for each region captures all H1 references to H2 regions.

Internal references between regions in H2: We also need to avoid internal H2 references across regions. To achieve this, *TeraHeap* detects internal references and logically merges the source and destination regions in a single group. Figure 6 shows how *TeraHeap* tracks groups of regions. To track several groups requires an array with region-group metadata. Array entries keep a reference to a list of region groups. During the marking phase of major GC, we detect if objects with the *TeraHeap* mark word enabled reference existing regions in H2. By moving objects to H2 in the compaction phase, the *TeraHeap* allocator logically unifies regions with cross-references by inserting a reference to the region array. If a group already exists, then we append the new region to the group. Note that region-merge incurs constant overhead, as it only involves a single pointer in the region.

At the end of major GC, any H2 region not marked as live is not reachable from any object in H1 nor from any GC root (e.g thread local variables, JNI local and global variables). Therefore, their regions can be freed, which involves constant overhead: We set the allocation offset in the respective region(s) to zero and clean the card tables that refer to the objects of this region.

4 Experimental Methodology

Experimental platform: To evaluate the performance of *TeraHeap* using block-based NVMe SSDs, we use a dual-socket server with Intel Xeon E5-2630 v3 processors clocked at 2.4 GHz with 16 cores (32 hyper-threads), with 256 GB of DDR4 DRAM. The system runs CentOS v7.3 Linux OS using 4.14.182 kernel. The server has a 2 TB Samsung PM983 PCI Express NVMe SSD. We limit the available DRAM capacity in our experiments using a large, statically allocated ramdisk. Table 1 shows DRAM size in each workload.

Also, we investigate the benefits of *TeraHeap* for setups that can use NVM for storing cached objects. We conduct our NVM evaluation on a dual-socket server with two Intel Xeon Platinum 8260M CPUs at 2.4GHz, with 24 cores and (96 hyper-threads), 192GB of DDR4 DRAM. The system runs CentOS v7.8 Linux OS using 3.10 kernel. We use Intel Optane DC Persistent Memory [5] with a total capacity of 3TB, of which 1TB is in Memory mode, and 2TB are in AppDirect mode. The system mounts NVM as a DAX file system (ext4) to establish direct mappings to the device.

We use OpenJDK with the HotSpot JVM built from source (v8u250-b70) with the Parallel Scavenge collector (PSGC) [28]. We use 16 garbage collector threads to reclaim the heap. We use Spark v2.3 with the Kryo serializer [7], a highly optimized S/D library for Java that Spark recommends. We follow commonly used guidelines [10] and use 8 cores for our Spark executor. To reduce variability, we disable swapping, and we set the CPU scaling governor to *performance.*

Baselines: We configure *TeraHeap* to allocate H1 on DRAM and H2 over our NVMe SSD via mmio. Using NVM, we configure *TeraHeap* to allocate H1 on DRAM and H2 over NVM via mmio using direct access to NVM (AppDirect mode). We compare these configurations with two baseline configurations: (1) Spark-SD uses the default storage level of Spark (Memory and Disk) which places executor memory (heap) in DRAM, using an on-heap cache (50% of the total heap size). The rest of the RDDs are serialized over NVMe SSD. In the case of NVM, Spark-SD serializes RDDs over NVM using AppDirect mode, handling NVM as a storage device. In both cases, we use the entire storage device as the off-heap RDD cache. (2) Spark-MO uses a single heap allocated over NVM without code changes. We set the CPU to memory mode, so all available DRAM (192GB) acts as a cache for NVM (1TB). We place all cached data on-heap by using the *Memory Only* storage level of Spark.

Workloads: We use five widely used and memory-intensive graph processing workloads from the Spark-Bench suite [30]: PageRank (PR), Connected Components (CC), Single Source Shortest Path (SSSP), Singular Value Decomposition Plus Plus (SVD), and Triangle Counts (TR). We synthesize datasets using the SparkBench data generators. Table 1 shows the configurations and the dataset sizes we use for Spark-SD, Spark-MO, and *TeraHeap* to run each workload over NVMe SSD and NVM, respectively. We repeat each experiment 5 times and report the average of the end-to-end execution time.

Execution time breakdowns and profiler-based estimation of S/D overhead: We break execution time into four main components: *other* time, S/D + I/O time, minor GC time, and major GC time. *Other* time includes application (mutator) time. In Spark-SD, S/D time includes S/D time both for shuffle and caching. In *TeraHeap* and Spark-MO (in the NVM setup), all S/D time is due to the shuffle operation.

The JVM reports the time spent in major and minor GC. We estimate the overhead of S/D as follows. The Spark executor consists of application (mutator) and GC threads. Application and GC threads are not overlapped in time and JVM reports the time spent in each group of threads. This allows us to plot execution time breakdowns for each application, including minor, major, and mutator times.

We note that all S/D occurs in application threads. We use a sampling profiler [4] to collect execution samples from the application threads. The samples include the stack trace, similar to the flame graph [23] approach. Then we sum the samples for all the paths that originate from the top-level functions *writeObject()* and *readObject()* functions of the KryoSerializationStream and KryoDeserializationStream classes. These samples include both S/D for the compute cache and the shuffle network path of Spark. *We then use the ratio of S/D samples to the total application thread samples* as an estimate of the time spent in S/D, and we plot this separately in our execution time breakdowns. We run the profiler with a 10ms sampling interval, and we verify that this does not create significant overhead (less than 2% of total execution time).

Cached data footprint (D): We introduce a new metric D, the footprint of all cached Spark RDDs. D depends on application behavior. Our measurements show that D can be up to 20× larger than the input dataset and can easily increase the heap required by the executor. D is also affected by the fact that deserialized Java objects on the managed heap occupy 2-3× their size in serialized form [51]. So, for large datasets, only a small portion of D can be cached in memory [2, 19, 24, 24]. In our work, we experimentally determine the D for each application and Table 1 summarizes our results.

Heap size (H): To capture the effect of large datasets and limited DRAM capacity [20], we use a heap size between 2.5%, 5%, 10%, and 20% of D for our workloads running with Spark-SD. Table 1 lists the heap sizes corresponding to 10% of D for our experiments. TR uses a heap size of 173% of D (larger than 10% of D) because it generates massive temporary records for the join operation in each iteration. TR does not execute successfully with smaller heaps.

We devote a fixed (12 GB) amount of DRAM to the OS, used as a cache for I/O and other OS-related functionality. We also devote a fixed amount of memory (4GB) for the Spark executor process, besides the JVM heap. In *TeraHeap*, we use the same amount of DRAM as in the native configurations of each workload, divided between DR1 used to back H1 (10% of the data footprint) and DR2 (12 GB + 4 GB), which is used for the OS, including the mmap'd H2 and the Spark executor.

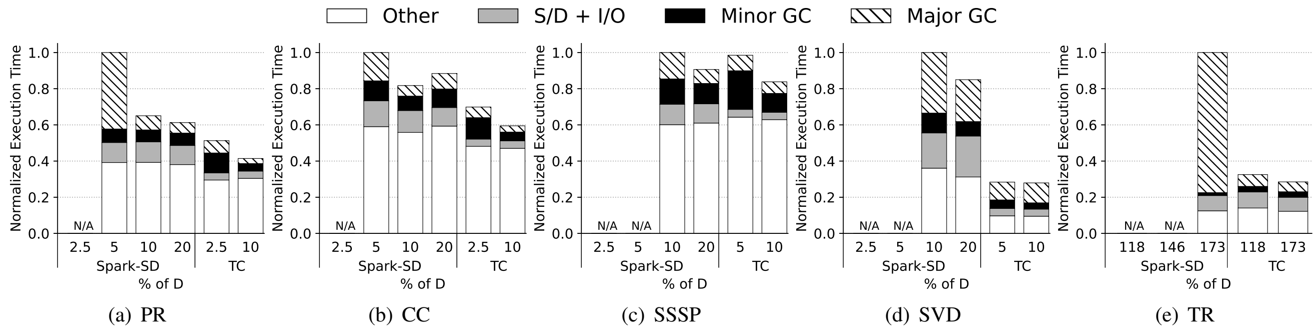
5 Evaluation

5.1 Performance Evaluation with NVMe

We first explore the tradeoff in DRAM capacity and overall performance with Spark-SD and *TeraHeap* for five analytics workloads in Figure 7, using NVMe-SSD setup. Specifically, we show total execution time and its breakdown into different components with varying heap sizes. We vary the heap sizes from 2.5% to 20% of D, except TR that requires a larger heap size to execute successfully (118%-173% of D). The total DRAM capacity is equal to the heap size plus the memory reserved for the OS and Spark driver (16 GB). We show results for *TeraHeap* with H1 set as 2.5% and 10% of D (last two bars in Figure 7). The missing bars in the figure indicate that such configurations suffer from out-of-memory errors.

We first observe that *TeraHeap* does not suffer from out-of-memory errors for similar heap sizes, unlike the baseline configurations. Thus, *TeraHeap* makes more efficient use of the managed heap. Next, across all applications, the best-performing *TeraHeap* configurations consume 8× less DRAM capacity than Spark-SD. For tight heaps, reducing the GC overhead is paramount. *TeraHeap* transfers cached objects to H2, freeing up space in H1 for use by the executors in Spark. As a result, *TeraHeap* reduces the GC overhead by up to 64% (PR) because we avoid to traverse up to 59 million forward references in H2. Specifically, with 5%-10% of D, the GC overhead in the baseline is up to 50%. This overhead is mainly because cached objects on the managed heap occupy almost half of the total heap, triggering GC more frequently. For example, by increasing the heap size from 5% to 20% in Spark-SD (PR) the total number of major GC decreases by 90%. GC is a space-time tradeoff, and as we increase D to 20%, the GC overhead raises on average to 25% of the execution time in the baseline. *TeraHeap* continues to deliver improved performance for a large D. Also, *TeraHeap* reduces significantly S/D overhead by up to 80% across all heap sizes as it eliminates the S/D cost of the cached RDDs objects.

We next discuss the overall performance of *TeraHeap* compared to Spark-SD when the heap size is 10% of D (Figure 7). Each run is in the order of 0.5-2 hours. Overall, *TeraHeap* improves performance compared to Spark-SD by 36%, 27%, 16%, 72%, and 71% in PR, CC, SSSP, SVD, and TR, respectively. We observe that the S/D overhead in TR for *TeraHeap* is similar to Spark-SD because cached data fits in the on-heap cache. Although *TeraHeap* reduces the S/D and GC overhead, it also reduces the *other* time by up to 74% (on average 29%). This reduction is because *TeraHeap* incurs fewer (CPU) cache misses due to GC-triggered object movement. More specifically, the collector scans and copies each live object to a new location inside the heap. The copying cost is proportional to object size and changes the cache behavior [25]. As shown in Table 2, *TeraHeap* achieve 5.54× fewer cache misses, resulting in a dramatic (up to 74% in SVD) reduction in *other* time



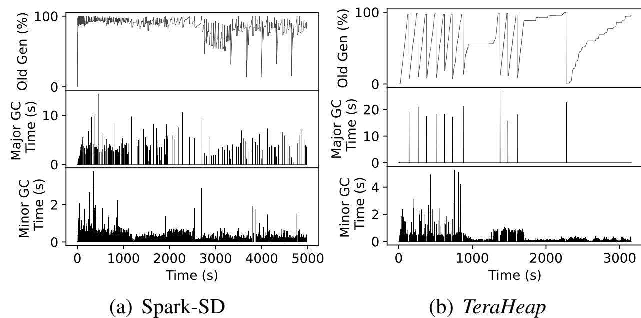
(a) PR (b) CC (c) SSSP (d) SVD (e) TR

Figure 7: Overall performance of *TeraHeap* (TC) compared to Spark-SD using the NVMe SSD setup with heap sizes equal to 2.5%, 5%, 10%, 20% of D. (DRAM capacity varies accordingly.) Note that TR requires heap sizes larger than D.

Performance Counters	PR	CC	SSSP	SVD	TR
Cache References	1.24	1.13	1.10	6.29	1.52
Cache Misses	1.11	1.06	1.04	5.54	1.79

Table 2: Ratio of CPU Cache References and Cache Misses of Spark-SD over *TeraHeap* for each workload with 10% of D heap size, using NVMe SSD configuration.

compared to Spark-SD.

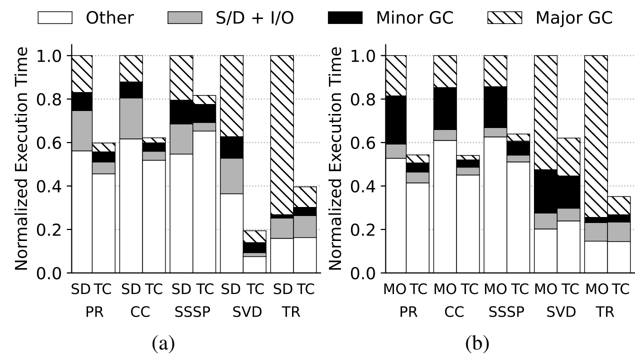


(a) Spark-SD (b) *TeraHeap*

Figure 8: GC time and old generation occupancy in PR for (a) Spark-SD and (b) *TeraHeap*. Heap size is equal to 10% of D.

Next, we discuss GC time and the percentage of heap consumed by the old generation for PR with Spark-SD and *TeraHeap* (heap size is equal to 10% of D) in Figure 8 (a) and (b). We observe similar behavior in the other workloads but omit the full results due to space constraints. We note that Spark-SD suffers from frequent major GC cycles. There are 171 cycles of major GC, with each cycle requiring on average 3.7 secs. Each cycle in Spark-SD can only reclaim 10% of the old generation objects (0-3000 seconds), as the remaining objects are live cached objects. In contrast, *TeraHeap* performs only 13 major GC cycles. During each cycle, *TeraHeap* moves to H2 on average 28,523 objects (out of 313,751 total cached objects) and up to 60% of the old generation objects, reducing stress on the GC. Each cycle in *TeraHeap* takes on average 16 seconds, and a large portion of it is due to I/O during the compaction phase. Finally, transferring objects directly from the young generation to *TeraHeap* reduces total minor GC time by 38% compared to Spark-SD. This reduction of the minor GC time (shown in the figure) is because *TeraHeap* requires scanning fewer cards that track old-to-young references because fewer objects are in the old generation than Spark-SD.

5.2 Performance Evaluation with NVM



(a) (b)

Figure 9: Overall performance of *TeraHeap* (TC) compared to (a) Spark-SD and (b) Spark-MO baselines over NVM.

Now, we examine the benefits of *TeraHeap* for setups with cached objects backed by NVM. Figure 9 (a) and (b) shows the performance breakdown for Spark-SD, Spark-MO, and *TeraHeap* in our NVM-based setup with 10% of D heap size.

Figure 9 (a) shows that *TeraHeap* improves performance by 40%, 38%, 18%, 81%, and 60% compared to Spark-SD in PR, CC, SSSP, SVD, and TR, respectively. We note that block-addressable storage devices suffer from expensive S/D operations that result in read/write I/O system calls for writing data to persistent storage. However, *TeraHeap* exploits the byte-addressability of NVM and loads and stores cached objects from memory, resulting in fine-grain access to the cached data objects. For example, Spark GraphX keeps an index structure in each partition to send and receive data

across vertices. *TeraHeap* uses mmio to directly access this data structure inside the partition. In comparison, the S/D approach needs to deserialize the partition's objects and then access the index structure. Overall, our results show that *TeraHeap* significantly reduces S/D and GC time compared to Spark-SD by up to 89% and 70%, respectively. Also, in SVD workload *TeraHeap* reduces the *other* time by 80% compared to Spark-SD. Reducing the movements of the object produced during GC, *TeraHeap* highly decreases the CPU cache misses by 5.6×.

Figure 9 (b) shows that *TeraHeap* improves performance by 46%, 46%, 36%, 38%, 65% compared to Spark-MO in PR, CC, SSSP, SVD, and TR, respectively. Mainly the performance improvement of *TeraHeap* results from the reduction of the minor GC time and major GC time by up to 82% (on average 64%) and 89% (on average 80%) compared to Spark-MO, respectively. Running the garbage collector on top of NVM (using DRAM as a cache) becomes a severe bottleneck mainly due to the high latency of NVM [54] and the agnostic placements of objects. The garbage collector performs traversal to identify live objects. Traversing the fields (references) inside a live object, the referred objects could reside anywhere in the heap. Some of these objects may not reside in DRAM cache so the access latency increases because the garbage collector has to retrieve them from NVM, increasing the GC time. For this reason, Spark-MO increases the number of read and write requests over NVM compared to *TeraHeap* by 5.3× and 11.8× on average, respectively. Thus, the ability to maintain distinct heaps for the execution and caching parts of the heap, solely use the NVM for caching, and prevent GC in the cache, are all vital to performance.

5.3 GC Analysis with *TeraHeap*

TeraHeap performs extra work during minor and major GC of the H1 heap to avoid GC over the H2 heap. The extra work involves (1) scanning the H2 card table during minor GC and (2) transferring objects from H1 to H2 during major GC.

First, we evaluate the overhead of scanning the H2 card table during minor GC. Figure 10 (a) shows minor GC time using 512 B, 1 KB, 4 KB, 8 KB, and 16 KB card segments, normalized to 512 B card segments. We observe that increasing the card segment from 512 B to 16 KB reduces minor GC time up to 40% (on average by 14.0%). Larger card segments result in a smaller card table and require less time to scan the respective cards. However, H2 objects mostly reside on the storage device. Therefore, increasing the card segment size, increases the cost of scanning each card segment, if the respective card is marked as dirty. We observe that updates to H2 objects are infrequent compared to H1 updates, as RDDs are immutable. For instance, SVD has only three references from H2 to H1, which contains 3,6 billion objects. Thus, in H2, it is preferable to use larger rather than smaller card segments to reduce the number of cards at the cost of larger card segments.

Scanning the H2 card table is also affected by the number of dirty boundary cards. To avoid synchronization for objects that span card segments (across stripes), the garbage collector *does not clean* boundary cards between stripes after scanning the respective objects for back-pointers. As a result, if a boundary card is marked once dirty, then the garbage collector traverses the objects in the respective card segment in every subsequent minor GC. However, these always-dirty-boundary-cards create significant overhead in *TeraHeap* because scanning occurs over the storage device. Therefore, in H2, we use a larger stripe size, which results in a smaller percentage of boundary cards. Using a stripe size of 64 KB with 512KB card segments (Figure 10 (a)) increases the number of boundary cards compared to using 8MB stripe size.

Figure 10 (b) depicts minor GC time for 2 MB, 4 MB, and 8 MB stripes sizes using 4 KB card segment, normalized to 2 MB. We note that in CC, SSSP, and SVD, minor GC time reduces on average by 23% and up to 44%. In PR and TR, minor GC time is similar for all the three stripe sizes. Therefore, because H2 is larger than H1, it is preferable to use a large stripe size to minimize the number of boundary cards.

Second, we investigate the overheads introduced by *TeraHeap* during major GC for H1 by copying objects from the old generation to H2, which involves device I/O. We find that the mark, precompact, and adjust phases of major GC take up 2% of the total major GC time. The compact phase takes the remaining 98% of major GC time due to the required I/O. To reduce the overhead of I/O-based object migration, we explore different approaches for I/O writes during the compaction phase: (1) memory copying over mmio (memcpy) and (2) asynchronous system calls (AsyncIO). *TeraHeap* waits for all asynchronous I/O operations to complete before the compaction phase terminates.

Figure 10 (c) shows major GC time using different I/O mechanisms for moving objects to H2, normalized to the performance of memcpy. AsyncIO reduces major GC time by 15%, 24%, and 27% in PR, CC, and SSSP workloads because we use system calls for large I/Os without polluting the DRAM cache. We observe that AsyncIO in PR delivers throughput up to 550MB/s compared to mmio that delivers up to 400MB/s. The compaction phase in the PSGC is single-threaded by design. For mmio, this results in a single outstanding I/O, under-utilizing the storage device. AsyncIO saturates the storage device with multiple outstanding I/Os (64 in our experiments). However, mmio reduces major GC time by 17× in SVD and 10% in TR because the system time is 3× lower than AsyncIO. To reduce the large number of system calls for small-sized objects, TC can use a 2 MB buffer to collect all small objects and then perform one system call to write 2 MB objects.

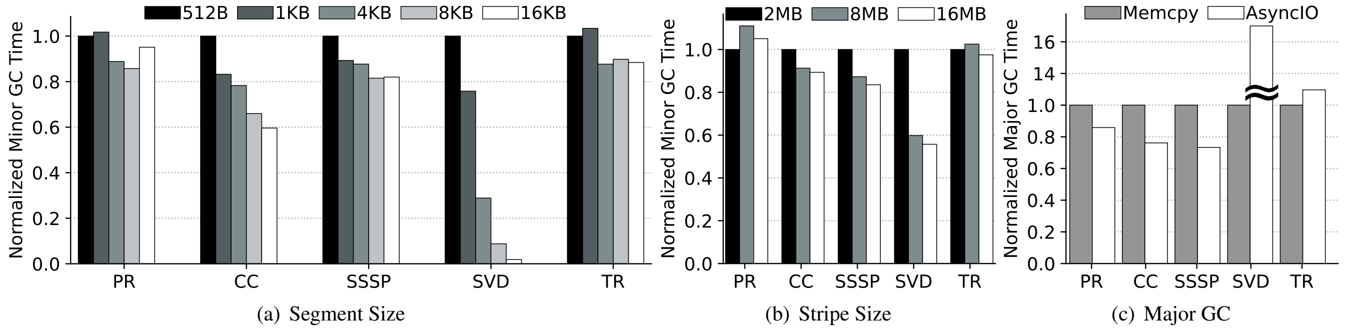


Figure 10: Minor GC time with *TeraHeap* for (a) various card segments using 64KB stripe size and (b) for 2MB, 4MB, and 8MB stripe sizes using 4KB card segment, and (c) major GC time with *TeraHeap* for different I/O mechanisms.

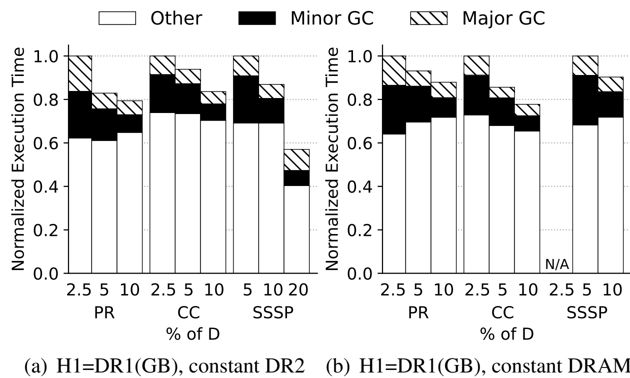


Figure 11: Sensitivity analysis of *TeraHeap* with respect to (a) the size of H1 (DR1) with constant DR2, and (b) the size of H1 (accordingly to H2) with constant amount of DRAM capacity.

5.4 Sensitivity Analysis

Although *TeraHeap* performs well with smaller amounts of DRAM for H1 compared to the native JVM, it is interesting to examine how *TeraHeap* might benefit from increasing the available DRAM capacity. Figure 11 (a) shows normalized execution time in *TeraHeap* as the size of H1 grows from 2.5%, 5%, and 10% of footprint D for PR and CC, and 5%, 10% and 20% of footprint D for SSSP. The size of DR2 is constant across all workloads. We report only PR, CC, and SSSP due to space constraints, and we normalize the results in each group to the bar for 2.5% of D (5% of D for SSSP). As H1 (DR1) size increases, minor and major GC time decreases up to 68% (on average by 62%) and up to 60% (on average 41%), respectively. Therefore, the performance of *TeraHeap* is more sensitive to the size of H1.

Next, we examine the sensitivity of *TeraHeap* to dividing a fixed amount of DRAM between DR1 and DR2. Figure 11 (b) shows the performance of *TeraHeap* when H1 varies from 2.5%, 5% and 10% of D in PR, CC, and SSSP. Since the DRAM capacity remains constant in all configurations (80 GB in PR, 84 GB in CC, and 58 GB in SSSP), the size of DR2 decreases accordingly. The missing bar in SSSP indicates that the configuration with 2.5% of D cannot run due to an out-of-heap error. We normalize the results in each group to the bar 2.5% of D (5% of D in SSSP). By increasing H1 from 2.5%-10% in PR and CC, and from 5%-10% of D in SSSP, both minor and major GC time decrease by up to 62% and 47% (on average by 41% and 29%), respectively. Additionally, as we increase the H1 from 2.5%-10% of D, the garbage collector performs 2× less major GC cycles, moving by 50% more objects to H2 in each GC cycle. Therefore, it is preferable to devote more DRAM capacity to H1 rather than devoting it to the DR2 buffer cache.

5.5 Terabyte Cached Data Footprints

So far, for practical purposes, we have analyzed *TeraHeap* for datasets that create a few hundred GBs of cached data (D). We now perform a limited evaluation with TB-level cached data footprints in our NVMe SSD setup. We use datasets that result in cached data footprints of 1.4 TB, 1.12 TB, 1.32 TB, and 1.33 TB in PR, CC, SSSP, and SVD, respectively. We maintain the ratio of about 10% of D and use an H1 of 140 GB, 112 GB, 132 GB, 133 GB for PR, CC, SSSP, and SVD, respectively. We allocate 16 GB of DRAM for use by the OS and the Spark driver. Each run is in the order of 2-5 hours. We observe improvements with *TeraHeap* in line with our results with smaller datasets. *TeraHeap* improves overall performance compared to Spark-SD by 39%, 33%, 23%, and 48% in PR, CC, SSSP, and SVD, respectively. These improvements are slightly better than our results with small datasets because GC and S/D overheads tend to increase as data footprint grows.

6 Related Work

Mitigation of S/D overhead for big data analytics: Neutrino [51] proposes fine-grain adaptive caching for Spark that serializes RDDs based on available executor memory. LLC

and LRC [21, 58] evict RDD partitions that minimize RDD recomputation time. MemTune [52] dynamically tunes the partitioning of memory between computation and caching. MemTune also offers task-level data prefetching to overlap computation with S/D and I/O operations. Zhang et al. [62] modify cache management to reduce RDD movement between DRAM and disk. These prior works attempt to mitigate the S/D overhead without addressing GC overhead. In comparison, *TeraHeap* uses memory-mapped RDD caching to eliminate S/D and avoids expensive GC traversals over cached data.

Recently, several libraries [3, 7, 9] improve the efficiency of S/D, but they still result in high S/D overhead for big data frameworks [34]. Apache Arrow [1] and Tungsten [11] use off-heap computation but require prior knowledge of the object schema (e.g., Spark SQL) and do not extend to applications that use complex data structures (e.g., graph processing). Skyway [35] reduces the S/D cost by directly transferring objects through the network in distributed managed heaps, but it does not cope with DRAM limitations and GC overheads. Recent work [27, 40] examines techniques to reduce S/D overheads in analytics frameworks using custom hardware and modifications to the programming model. Other works [41, 46, 49] focus on reducing S/D cost by reducing the number of object copies across buffers. *TeraHeap* requires no changes to the application code and works on commodity hardware. Also, *TeraHeap* uses load/store instructions to access cached objects (mmio) without additional copies and transformations.

Scaling heaps and minimizing GC overhead: Recent work targets emerging non-volatile memory (NVM) for storing managed heaps [13, 14, 43, 47, 50, 56]. They focus on (1) scaling the managed heap beyond DRAM capacity [13,14,47,56] and (2) exploiting GC to manage a persistent heap [43, 50]. Panthera [47] requires offline profiling to move infrequently accessed RDDs in NVM [47]. Other works focus on improving NVM write endurance [13, 14]. The authors in [56] report high GC overhead with NVM-backed volatile heaps and optimize the G1 GC for Intel Optane memory. Exploiting GC for managing persistent heaps [43, 50] is relevant but orthogonal to our work. Existing garbage collectors do not handle large heaps over NVM, and unlike *TeraHeap*, they increase GC overhead. Unlike most prior works, *TeraHeap* is generalizable to different types of garbage collectors.

Managed big data frameworks have revived the interest in reducing GC overhead [22, 37]. Yak [36] proposes a new garbage collector that uses program semantics to divide the (DRAM) managed heap into control and data heaps. Yak organizes the data heap into regions of objects with similar lifetimes. On deallocating a region, Yak migrates objects with escape references to newly merged regions. The data heap in Yak is not compatible for placement on a storage device as the cost of object migrations due to region merging leads to a prohibitive overhead. Prior efforts propose techniques to segregate long-lived objects and manage them separately in an unmodified heap. NG2C [18] uses runtime profiling to identify long-lived objects. They incur online profiling overhead. Others use offline allocation-site profiling to manage long-lived objects [16, 17]. Lifetime profiling is complementary to *TeraHeap*, and it can further improve its efficiency. Unlike prior approaches, *TeraHeap* is the first JVM proposal that supports a dual heap over memory and storage and reduces the GC and S/D overhead for analytics caches.

7 Conclusion

Managed data analytics frameworks require processing large datasets with iterative computations that demand large compute caches. Caching intermediate results incur high GC and S/D overheads. Our work proposes *TeraHeap*, a design that uses two heaps, H1 and H2, in the JVM, eliminating GC and S/D cost for H2 objects. H2 is memory-mapped to fast storage devices with high capacity and allows direct access to materialized objects. *TeraHeap* improves Spark performance up to 72% (on average 45%) and 81% (on average 47%) for NVMe and NVM devices, respectively. Also, TeraCache utilizes 8× less DRAM capacity to provide comparable or higher performance than native Spark.

We believe that our approach of managing large memory in the JVM as customized, separate heaps, with policies that match the properties of certain object groups is particularly promising for incorporating huge address spaces in Java without incurring excessive GC overhead. We believe that future work will be successful in identifying other types of objects, such as persistent or network-related that will be amenable to placement in customized heaps.

References

[1] Apache arrow: A cross-language development platform for in-memory data. https://arrow.apache.org/. Accessed: May 07, 2021.

[2] Apache Spark @Scale: A 60 TB+ production use case. https://engineering.fb.com/2016/08/31/core-data/apache-spark-scale-a-60-tb-production-use-case/. Accessed: May 01, 2021.

[3] Apache Thrift. https://thrift.apache.org/. Accessed: May 07, 2021.

[4] Async Profiler. https://github.com/jvm-profiling-tools/async-profiler. Accessed: May 07, 2021.

[5] Intel Optane DC Persistent Memory. http://www.intel.com/optanedcpersistentmemory/. Accessed: Aug 12, 2021.

[6] Java Object Serialization Specification. https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html. Accessed: May 07, 2021.

[7] Kryo. https://github.com/EsotericSoftware/kryo. Accessed: May 07, 2021.

[8] *Limux Programmer's Manual*. Accessed: May 07, 2021.

[9] Protocol Buffers. https://developers.google.com/protocol-buffers/docs/javatutorial. Accessed: May 07, 2021.

[10] Right Sizing Spark Executors. https://ec2spotworkshops.com/running_spark_apps_with_emr_on_spot_instances/right_sizing_executors.html. Accessed: May 07, 2021.

[11] Project tungsten: Bringing apache spark closer to bare metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringingspark-closer-to-bare-metal.html, 2015. Accessed: May 07, 2021.

[12] RDD Programming Guide. https://spark.apache.org/docs/2.3.0/rdd-programming-guide.html, 2018. Accessed: May 07, 2021.

[13] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. *SIGPLAN Not.*, 53(4):62–77, June 2018.

[14] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Crystal gazer: Profile-driven write-rationing garbage collection for hybrid memories. *SIGMETRICS Perform. Eval. Rev.*, 47(1):21–22, December 2019.

[15] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

[16] Stephen M. Blackburn, Matthew Hertz, Kathryn S. Mckinley, J. Eliot B. Moss, and Ting Yang. Profile-based pretenuring. *ACM Trans. Program. Lang. Syst.*, 29(1):2–es, January 2007.

[17] Stephen M. Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S. McKinely, and J. Eliot B. Moss. Pretenuring for java. *SIGPLAN Not.*, 36(11):342–352, October 2001.

[18] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. Association for Computing Machinery, March 2019.

[19] Zhuhua Cai, Zekai J. Gao, Shangyu Luo, Luis L. Perez, Zografoula Vagena, and Christopher Jermaine. A comparison of platforms for implementing and running very large scale machine learning algorithms. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 1371–1382. Association for Computing Machinery, 2014.

[20] Zhiguang Chen, Yutong Lu, Nong Xiao, and Fang Liu. A hybrid memory built by ssd and dram to support in-memory big data analytics. *Knowledge and Information Systems*, 41(2):335–354, 2014.

[21] Yuanzhen Geng, Xuanhua Shi, Cheng Pei, Hai Jin, and Wenbin Jiang. Lcs: An efficient data eviction strategy for spark. *Int. J. Parallel Program.*, 45(6):1285–1297, December 2017.

[22] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingan, Derek Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 2, USA, 2015. USENIX Association.

[23] Brendan Gregg. Visualizing performance with flame graphs. Santa Clara, CA, July 2017. USENIX Association.

[24] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. An experimental comparison of pregel-like graph processing systems. *Proc. VLDB Endow.*, 7(12):1047–1058, August 2014.

[25] Michael W. Hicks, Jonathan T. Moore, and Scott M. Nettles. The measured cost of copying garbage collection mechanisms. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, page 292–305. Association for Computing Machinery, 1997.

[26] Urs Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA'93 Garbage Collection Workshop*, volume 93. Citeseer, October 1993.

[27] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 322–334. IEEE Press, May 2020.

[28] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

[29] Dongyang Li, Fei Wu, Yang Weng, Qing Yang, and Changsheng Xie. Hods: Hardware object deserialization inside ssd storage. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 157–164. IEE, April 2018.

[30] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. Sparkbench: A spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, 20(3):2575–2589, September 2017.

[31] Sheng Liang. *Java Native Interface: Programmer's Guide and Reference*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1999.

[32] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. Lifetime-based memory management for distributed data processing systems. *Proc. VLDB Endow.*, 9(12):936–947, August 2016.

[33] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. Gerenuk: Thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 538–553. Association for Computing Machinery, October 2019.

[34] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, page 291–305. USENIX Association, July 2015.

[35] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 56–69. Association for Computing Machinery, March 2018.

[36] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 349–365, USA, November 2016. USENIX Association.

[37] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. *SIGPLAN Not.*, 50(4):675–690, March 2015.

[38] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 293–307. USENIX Association, May 2015.

[39] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped i/o for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 813–827. USENIX Association, July 2020.

[40] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1203–1216. Association for Computing Machinery, 2020.

[41] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 199–205. Association for Computing Machinery, 2021.

[42] David Reinsel, John Gantz, and John Rydning. Data age 2025: The evolution of data to life-critical. *Don't Focus on Big Data*, 2, 2017.

[43] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 316–332. Association for Computing Machinery, 2019.

[44] Shubhanshi Singhal, Pooja Sharma, Rajesh Kumar Aggarwal, and Vishal Passricha. A global survey on data deduplication. *Int. J. Grid High Perform. Comput.*, 10(4):43–66, October 2018.

[45] Apache Spark. Rdd programming guide, 2020.

[46] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. Naos: Serialization-free RDMA networking in java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 1–14. USENIX Association, July 2021.

[47] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 347–362. Association for Computing Machinery, June 2019.

[48] Paul R Wilson and Thomas G Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989.

[49] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Mingyu Wu, Haibo Chen, Hao Zhu, Binyu Zang, and Haibing Guan. Gcpersist: An efficient gc-assisted lazy persistency framework for resilient java applications on nvm. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 1–14. Association for Computing Machinery, March 2020.

[51] Erci Xu, Mohit Saxena, and Lawrence Chiu. Neutrino: Revisiting memory caching for iterative data analytics. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, pages 16–20, USA, June 2016. USENIX Association.

[52] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu. Memtune: Dynamic memory management for in-memory data analytic platforms. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 383–392. IEEE, May 2016.

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

[54] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[55] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[56] Yanfei Yang, Mingyu Wu, Haibo Chen, and Binyu Zang. Bridging the performance gap for copy-based garbage collectors atop non-volatile memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 343–358. Association for Computing Machinery, April 2021.

[57] Jung Yoon, Ranjana Godse, and Andrew Walls. 3d nand technology scaling helps accelerate ai growth. In *Proceedings of the Flash Memory Summit (FSM)*, 2018.

[58] Yinghao Yu, Wei Wang, Jun Zhang, and Khaled Ben Letaief. Lrc: Dependency-aware cache management for data analytics clusters. *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[59] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy Mccauley, M Franklin, Scott Shenker, and Ion Stoica. Fast and interactive analytics over hadoop data with spark. *Usenix Login*, 37(4):45–51, 2012.

[60] 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, page 2. USENIX Association, April 2012.

[61] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10. USENIX Association, June 2010.

[62] K. Zhang, Y. Tanimura, H. Nakada, and H. Ogawa. Understanding and improving disk-based intermediate data caching in spark. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2508–2517, December 2017.