Deca: A Garbage Collection Optimizer for In-Memory Data Processing

XUANHUA SHI and ZHIXIANG KE, Huazhong University of Science and Technology, China
YONGLUAN ZHOU, University of Copenhagen, Denmark
HAI JIN, Huazhong University of Science and Technology, China
LU LU, Alibaba Group, China
XIONG ZHANG, Huazhong University of Science and Technology, China
LIGANG HE, University of Warwick, United Kingdom
ZHENYU HU and FEI WANG, Huazhong University of Science and Technology, China

In-memory caching of intermediate data and active combining of data in shuffle buffers have been shown to be very effective in minimizing the recomputation and I/O cost in big data processing systems such as Spark and Flink. However, it has also been widely reported that these techniques would create a large amount of long-living data objects in the heap. These generated objects may quickly saturate the garbage collector, especially when handling a large dataset, and hence, limit the scalability of the system. To eliminate this problem, we propose a lifetime-based memory management framework, which, by automatically analyzing the user-defined functions and data types, obtains the expected lifetime of the data objects and then allocates and releases memory space accordingly to minimize the garbage collection overhead. In particular, we present Deca,[1] a concrete implementation of our proposal on top of Spark, which transparently decomposes and groups objects with similar lifetimes into byte arrays and releases their space altogether when their lifetimes come to an end. When systems are processing very large data, Deca also provides field-oriented memory pages to ensure high compression efficiency. Extensive experimental studies using both synthetic and real datasets show that, in comparing to Spark, Deca is able to (1) reduce the garbage collection time by up to 99.9%, (2) reduce the memory consumption by up to 46.6% and the storage space by 23.4%, (3) achieve 1.2× to 22.7× speedup in terms of execution time in cases without data spilling and 16× to 41.6× speedup in cases with data spilling, and (4) provide similar performance compared to domain-specific systems.

[1] The preliminary results of this work have been published in VLDB 2016 [28].

This work is supported by the National Key R&D Program of China (No. 2017YFC0803700), the NSFC (No. 61772218, 61433019, U1435217), and the Outstanding Youth Foundation of Hubei Province (No. 2016CFA032).

Authors' addresses: X. Shi and Z. Ke, Huazhong University of Science and Technology, Services Computing Technology and System Lab/Big Data Technology and System Lab, School of Computer Science and Technology, 1037 Luoyu Rd., WuHan, 430074, China; emails: {xhshi, zhxke}@hust.edu.cn; Y. Zhou, University of Copenhagen, Department of Computer Science, DK-2100, Copenhagen, Denmark; email: zhou@di.ku.dk; H. Jin, Huazhong University of Science and Technology, Services Computing Technology and System Lab/Big Data Technology and System Lab, School of Computer Science and Technology, 1037 Luoyu Rd., WuHan, 430074, China; email: hjin@hust.edu.cn; L. Lu, Alibaba Group, HangZhou, China; email: shiyi.ll@alibaba-inc.com; X. Zhang, Huazhong University of Science and Technology, China; email: wxzhang@hust.edu.cn; L. He, University of Warwick, Department of Computer Science, Coventry, CV4 7AL, United Kingdom; email: liganghe@dcs.warwick.ac.uk; Z. Hu and F. Wang, Huazhong University of Science and Technology, China; emails: {cszhenyuhu, feiwg}@hust.edu.cn.

0734-2071/2019/03-ART3 $15.00
https://doi.org/10.1145/3310361

CCS Concepts: • **Information systems** → **Data management systems**;

Additional Key Words and Phrases: Data processing system, distributed system, garbage collection, in-memory, memory management

ACM Reference format:
Xuanhua Shi, Zhixiang Ke, Yongluan Zhou, Hai Jin, Lu Lu, Xiong Zhang, Ligang He, Zhenyu Hu, and Fei Wang. 2019. Deca: A Garbage Collection Optimizer for In-Memory Data Processing. *ACM Trans. Comput. Syst.* 36, 1, Article 3 (March 2019), 47 pages.
https://doi.org/10.1145/3310361

1 INTRODUCTION

The big data processing systems that have emerged recently, such as Spark [49], can process huge volumes of data in a scale-out fashion. Unlike traditional database systems using declarative query languages and relational (or multidimensional) data models, these systems allow users to implement application logics through *User-Defined Functions* (UDFs) and *User-Defined Types* (UDTs) using high-level imperative languages (such as Java, Scala, C#, etc.), which can then be automatically parallelized onto a large-scale cluster.

Existing research in these systems mostly focuses on scalability and fault tolerance issues in a distributed environment [50, 48, 23, 2]. However, some recent studies [3, 31] suggest that the execution efficiency of individual tasks in these systems is low. A major reason is that both the execution frameworks and user programs of these systems are implemented using high-level imperative languages running in managed runtime platforms (such as JVM, .NET CLR, etc.). These managed runtime platforms commonly have built-in automatic memory management, which brings significant memory and CPU overheads. For example, the modern tracing-based garbage collectors (GCs) may consume a large amount of CPU cycles to trace living objects in the heap [24, 10, 45].

Furthermore, to improve the performance of multistage and iterative computations, recently developed systems support caching of intermediate data in the main memory [36, 40, 51] and exploit eager combining and aggregating of data in the shuffling phases [26, 39]. These techniques would generate massive *long-living* data objects in the heap, which usually stay in the memory for a significant portion of the job execution time. However, the unnecessary continuous tracing and marking of such a large amount of long-living objects by the GC would consume significant CPU cycles.

In this article, we argue that the big data processing systems such as Spark should employ a lifetime-based memory manager, which allocates and releases memory according to the lifetimes of the data objects rather than relying on a conventional tracing-based GC. To verify this concept, we present Deca, an automatic Spark optimizer, which adopts a lifetime-based memory management scheme for efficiently reclaiming memory space. Deca automatically analyzes the lifetimes of objects in different data containers in Spark, such as UDF variables, cached data blocks, and shuffle buffers, and then transparently decomposes and stores a massive number of objects with similar lifetimes into a few number of memory pages, which are in the form of byte arrays. These memory pages can be allocated either on or off the JVM heap. In this way, the massive objects essentially bypass the continuous tracing of the GC and the space that they occupy can be released by the destruction of the memory pages.

Last but not least, Deca automatically transforms the user programs so that the new memory layout is transparent to the users. By using the aforementioned techniques, Deca significantly optimizes the efficiency of Spark's memory management and at the same time keeps the generality and expressibility provided in Spark's programming model.

In summary, the main contributions of this article include:

- We propose a program optimization mechanism for in-memory big data processing systems and implement a prototype, Deca, on top of Spark. Deca is able to automatically remove the object encapsulation of UDT data objects for Spark programs, thereby minimizing the GC overhead and eliminating the memory bloat problem.
- We design a method that changes the in-memory representation of the object graph of each data item by discarding all the reference values. In optimized programs, the raw data of the fields of primitive types in the original object graph are compactly stored as a byte sequence.
- We propose a data-object lifetime analysis method based on traditional static program analysis techniques. In optimized programs, the byte sequences of data items with the same lifetime are grouped into a few byte arrays, called memory pages, thereby simplifying space reclamation. Deca also automatically validates the memory safety of data accessing based on the sophisticated analysis of memory usage of UDT objects.
- We conduct extensive evaluation on various Spark programs using both synthetic and real datasets. The experimental results demonstrate the superiority of our approach by comparing with existing methods.

2 OVERVIEW OF DECA

2.1 Java GC

In typical JVM implementations, a GC attempts to reclaim memory occupied by objects that will no longer be used. A tracing GC traces which objects are reachable by a sequence of references from some root objects. The unreachable ones, which are called garbage, can be reclaimed. Oracle's Hotspot JVM implements three GC algorithms. The default Parallel Scavenge (PS) algorithm suspends the application and spawns several parallel GC threads to achieve high throughput. The other two algorithms, namely, Concurrent Mark-Sweep (CMS) and Garbage-First (G1), attempt to reduce GC latency by spawning concurrent GC threads that run simultaneously with the application thread.

All the above collectors are *generational garbage collectors* that segregate objects into multiple generations: young generation containing recently allocated objects, old generation containing older objects, and permanent generation containing class metadata. Based on the weak generational hypothesis [27], the generational design assumes that most objects would soon become garbage; a minor GC, which only attempts to reclaim garbage in the young generation, can be run to reclaim enough memory space. However, if there are too many old objects, then a full (or major) GC would be run to reclaim space occupied by the old objects. Usually, a full GC is much more expensive than a minor GC.

There are also nongenerational garbage collectors implemented in other language runtime systems. For example, Go has a nongenerational concurrent mark-and-sweep garbage collector [18]. JikesRVM [1], a research-oriented JVM implementation, provides a suite of various generational and nongenerational GC implementations [7]. Nongenerational garbage collectors scan more heap area than the generational minor GC does, and hence spend more CPU time on GC if the workload follows the weak generational hypothesis.

2.2 Motivating Example

A major concept of Spark is the *Resilient Distributed Dataset* (RDD), which is a fault-tolerant dataset that can be processed in parallel by a set of UDF operations.

We use *Logistic Regression* (LR) as an example to motivate and illustrate the optimization techniques adopted in Deca. It is a classifier that attempts to find an optimal hyperplane that separates

```
1  class DenseVector[V](val data: Array[V],
2     val offset: Int,
3     val stride: Int,
4     val length: Int) extends Vector[V] {
5    def this(data: Array[V]) =
6     this(data, 0, 1, data.length)
7    ...
8  }
9  class LabeledPoint(var label: Double,
10    var features: Vector[Double])
11
12 val lines = sparkContext.textFile(inputPath)
13 val points = lines.map(line => {
14   val features = new Array[Double](D)
15   ...
16   new LabeledPoint(new DenseVector(features), label)
17 }).cache()
18 var weights =
19   DenseVector.fill(D){2 * rand.nextDouble - 1}
20 for (i <- 1 to ITERATIONS) {
21   val gradient = points.map { p =>
22     p.features * (1 / (1 +
23        exp(-p.label * weights.dot(p.features))) -
24        1) * p.label
25   }.reduce(_ + _)
26   weights -= gradient
27 }
```

Fig. 1. Demo spark program of scala *Logistic Regression*. It is an iterative machine-learning application that caches the train-set data in memory. The top-level UDT of cached data objects is LabeledPoint.

the data points in a multidimensional feature space into two sets. Figure 1 shows the code of LR in Spark. The raw dataset is a text file with each line containing one data point. Hence, the first UDF is a map function, which extracts the data points and stores them into a set of DenseVector objects (lines 12–16). An additional LabeledPoint object is created for each data point to package its feature vector and label value together.

To eliminate disk I/O for the subsequent iterative computation, LR uses the cache operation to cache the resulting LabeledPoint objects in the memory. For a large input dataset, this cache may contain a massive number of objects.

After generating a random separating plane (lines 18–19), it iteratively runs another map function and a reduce function to calculate a new gradient (lines 20–26). Here each call of this map function will create a new DenseVector object. These intermediate objects will not be used anymore after executing the reduce function. Therefore, if the aforementioned cached data leaves little space in the memory, then GC will be frequently run to reclaim the space occupied by the intermediate objects and make space for newly generated ones. Note that, after running a number of minor GCs, JVM would run a full GC to reclaim spaces occupied by the old objects. However, such highly expensive full GCs would be nearly useless because most cached data objects should not be removed from the memory.

We illustrate the impact of GC on Spark's performance using the following simple experiment. We run an LR application on Spark in local mode. A local-mode Spark application runs all the components in one JVM process, thereby eliminating the impact of RPC messaging and data transferring on performance. The machine used in the experiment has two Xeon-2670 CPUs, 64GB memory, and a SAS disk. The input training dataset contains 10GB of randomly generated labeled feature vectors with 10 dimensions. We measure the runtime spent on GC with different JVM heap sizes, ranging from 20GB to 40GB. As shown in Figure 2, when the memory space is not sufficient (i.e., in the cases of 20GB and 30GB), the system has significant GC overhead, such that the GC time can be nearly 5× longer than the pure computation time. The main reason is that the long-living

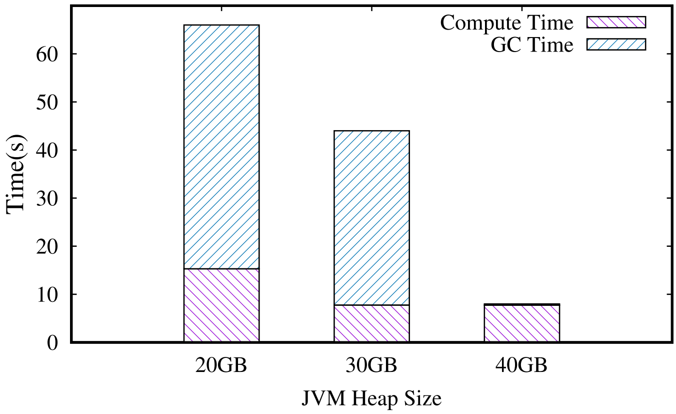


Fig. 2. Total execution time and GC time of Spark LR with different JVM heap sizes, running in Spark local mode (single JVM process). The input train-set data are 10GB randomly generated 10-dimension labeled feature vectors.

in-memory data objects frequently cause the triggering of major (or full) GCs. When the heap memory is sufficient (i.e., in the case of 40GB), the heap can contain all the objects, and hence there are only a few minor GCs, whose cost is negligible.

2.3 Lifetime-Based Memory Management

We implement the prototype of Deca based on Spark. In Deca, objects are stored in three types of data containers: UDF variables, cached RDDs, and Shuffle buffers. In each data container, Deca allocates a number of fixed-sized byte arrays. By using the *points-to analysis* [25], we map the UDT objects with their appropriate containers. UDT objects are then stored in the byte arrays after eliminating the unnecessary object headers and object references. This compact layout would not only minimize the memory consumption of data objects but also dramatically reduce the overhead of GC, because GC only needs to trace a few byte arrays instead of a huge number of UDT objects. One can see that the size of each byte array should not be too small or too large; otherwise, it would incur high GC overheads or large unused memory spaces.

As an example, the `LabeledPoint` objects in the LR program can be transformed into byte arrays as shown in Figure 3. Here, all the reference variables (in orange color, such as `features` and `data`), as well as the headers of all the objects, are eliminated. All the cached `LabeledPoint` objects are stored into byte arrays.

The challenge of employing such a compact layout is that the space allocated to each object is fixed. Therefore, we have to ensure that the size of an object would not exceed its allocated space during execution so that it will not damage the data layout. This is easy for some types of fields, such as primitive types, but less obvious for others. Code analysis is necessary to identify the change patterns of the objects' sizes. Such an analysis may have a global scope. For example, a global code analysis may identify that the `features` arrays of all the `LabeledPoint` objects (created in line 14 in Figure 1) actually have the same fixed size `D`, which is a global constant. Furthermore, the `features` field of a `LabeledPoint` object is only assigned in the `LabeledPoint` constructor. Therefore, all the `LabeledPoint` objects actually have the same fixed size. Another interesting pattern is that in Spark applications, objects in cached RDDs or shuffle buffers are often generated sequentially and their sizes will not be changed once they are completely generated. Identifying such useful patterns by a sophisticated code analysis is necessary to ensure the safety of decomposing UDT objects and storing them compactly into byte arrays.

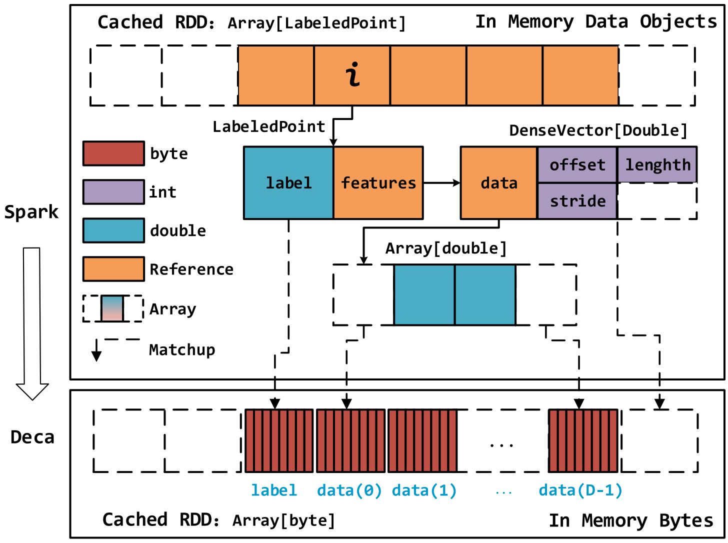


Fig. 3. The cached RDD data layout of Spark LR in both the original version and the Deca transformed version. Each LabeledPoint object (with its contained objects) stores all raw data in one double field, three int fields, and a double array field. In the Deca version, the data of one LabeledPoint object occupy a contiguous memory region that contains $20 + 8 \times D$ bytes.

As mentioned earlier, during the execution of programs in a system like Spark, the lifetimes of data containers created by the framework can be predetermined explicitly. For example, the lifetimes of objects in a cached RDD are determined by the invocations of cache() and unpersist() in the program. Recall that the UDT objects stored in the compact byte arrays would bypass the GC. We put the UDT objects with the same lifetime into the same container. For example, the cached LabeledPoint objects in LR have the same lifetime, so they are stored in the same container. The byte arrays can be allocated on or off the JVM heap. When a container's lifetime comes to an end, in the on-heap implementation, we simply release all the references of the byte arrays in the container, and then the GC can reclaim the whole space occupied by the massive amount of objects. In the off-the-heap implementation, the memory space of a container can simply be released at the end of its lifetime.

Deca extracts information of programs by using intraprocedural analysis, decomposes the UDTs, and then modifies the application programs by replacing the bytecodes of object creation, field access, and UDT methods with new codes that directly write and read the byte arrays. Finally, the converted programs will be submitted to Spark for execution.

In Figure 4, we illustrate the optimized code corresponding to the gradient computation code in lines 21 to 24 of Figure 1. While the codes produced by Deca are JVM bytecodes, we provide the equivalent Scala code here to illustrate the logic of the transformed LR code.

2.4 Technical Scope of Deca

So far, we only implemented the prototype of Deca on Spark Core. However, the optimization techniques of Deca can also be used in other big data processing frameworks as long as they satisfy the assumption of Deca. Roughly speaking, we assume that most long-living objects of a job can be placed into containers, which have predetermined lifetimes. Some other open-source frameworks

```
def computeGradient() = {
  val result = new Array[Double](D)
  var offset = 0
  while(offset < block.size) {
    var factor = 0.0
    val label = block.readDouble(offset)
    offset += 8
    for (i <- 0 to D) {
      val feature = block.readDouble(offset)
      factor += weights(i) * feature
      offset += 8
    }
    factor = (1 / (1 + exp(-label * factor)) - 1) * label
    offset -= 8 * D
    for (i <- 0 to D) {
      val value = block.readDouble(offset)
      result(i) = result(i) + feature * factor
      offset += 8
    }
  }
  result
}
```

Fig. 4. A transformed source code fragment of *Logistic Regression*.

also fulfill this assumption, like Hadoop, Storm, and GraphX, and hence they can also be modified to benefit from Deca's design. Further discussions of this issue can be found in Section 9.

3 UDT CLASSIFICATION ANALYSIS

3.1 Data Size and Size Type of Objects

To allocate enough memory space for objects, we have to compute the object sizes and their change patterns during runtime. Due to the complexity of object models, to accurately compute the size of a UDT, we have to dynamically traverse the runtime object reference graph of each target object and compute the total memory consumption. Such a dynamic analysis is too costly at runtime, especially with a large number of objects. Therefore, we opt for static analysis, which only uses static object reference graphs and would not incur any runtime overhead. We define the *data size* of an object to be the sum of the sizes of the primitive-type fields in its static object reference graph. An object's *data size* is only an upper bound of the actual memory consumption of its raw data, if one considers the cases with object sharing.

To see if UDT objects can be safely decomposed into byte sequences, we should examine how their data sizes change during runtime. There are two types of UDTs that can meet the safety requirement: (1) the data sizes of all the instances of the UDT are identical and do not change during runtime, or (2) the data sizes of all the instances of the UDT do not change during runtime. We call these two kinds of UDTs **Static Fixed-Sized Type (SFST)** and **Runtime Fixed-Sized Type (RFST)**, respectively.

In addition, we call UDTs that have type-dependency cycles in their type definition graphs as **Recursively Defined Type**. Even without object sharing, the instances of these types can have reference cycles in their object graphs. Therefore, they cannot be safely decomposed. Furthermore, any UDT that does not belong to any of the aforementioned types is called a **Variable-Sized Type (VST)**. Once a VST object is constructed, its data size may change due to field assignments and method invocations during runtime.

The objective of the UDT classification analysis is to generate the **Size Type** of each target UDT according to the above definitions. As demonstrated in Figure 3, Deca decomposes a set of objects into primitive values and stores them contiguously into compact byte sequences in a byte array. A safe decomposition requires that the original UDT objects are either of an SFST or an RFST.

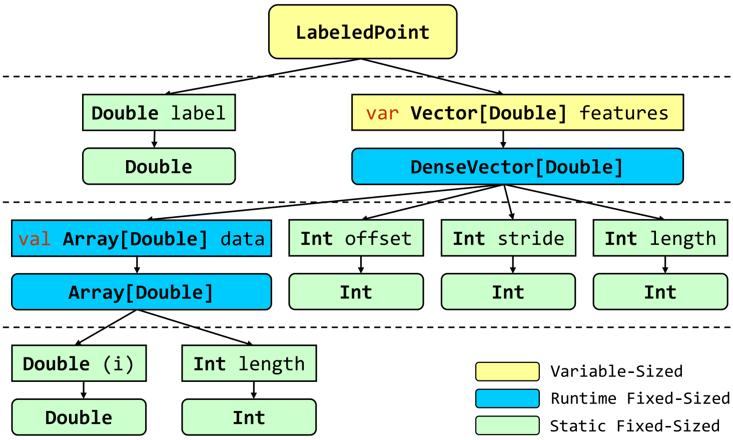


Fig. 5. An example of the local classification. LabeledPoint in Spark LR is classified as **Variable-Sized Type**.

Otherwise, the operations that expand the byte sequences occupied by an object may overwrite the data of the subsequent objects in the same byte array. Furthermore, as we will discuss later, an SFST can be safely decomposed in more cases than an RFST. On the other hand, most objects that do not belong to an SFST or an RFST will not be decomposed into byte sequences in Deca. In Section 5, we will discuss the objects that belong to VST but can still be decomposed subject to some restrictions. Apparently, to maximize the effect of our approach, we should avoid overestimating the variability of the data size of the UDTs, which is the design goal of our following algorithms.

3.2 Local Classification Analysis

The local classification algorithm analyzes a UDT by recursively traversing its type dependency graph. For example, Figure 5 illustrates the type dependency graph of LabeledPoint. The size type of LabeledPoint can be determined based on the size type of each of its fields.

Algorithm 1 shows the procedure of the local classification analysis. The input of the algorithm is an *annotated type* that contains the information of fields and methods of the target UDT. Because the objects referenced by a field can be of any subtype of its declared type, we use a *typeset* to store all the possible runtime types of each field. The typeset of each field is obtained in a preprocessing phase of Deca by using the points-to analysis [25] (see Section 6).

In lines 1 and 2, the algorithm first determines whether the target UDT is a recursively defined type. It builds the type-dependent graph and searches for cycles in the graph. If a cycle is found, the algorithm immediately returns the recursively defined type as the final result.

Two indirect recursive functions, AnalyzeType (lines 4–22) and AnalyzeField (lines 23–34), are used to further determine the size type of the target UDT. The stop condition of the recursion is when the current type is a primitive type (line 5). We treat each array type as having a length field and an element field. Since different instances of an array type can have different lengths, arrays with static fixed-sized elements will be considered as an RFST (lines 8–9).

We define a total ordering of the variability of the size types (except the recursively defined type) as follows: $SFST < RFST < VST$. Based on this order, the size type of each UDT is determined by its field that has the highest variability (lines 12–20). Furthermore, each field's final size type is determined by the type with the highest variability in its typeset. But a nonfinal field of an RFST will be finally classified as VST, because the same field can possibly point to objects with different data sizes (lines 28–29). Consider that whenever we find a VST field, the top-level UDT must also

ALGORITHM 1: Local classification analysis

```
Input: The top-level annotated type $T$;
Output: The size type of $T$;
build the type dependency graph $G$ for $T$;
if $G$ contains the circle path then return RecurDef;
else return AnalyzeType($T$);

Function AnalyzeType($t_{arg}$)
    if $t_{arg}$ is a primitive type then return StaticFixed;
    else if $t_{arg}$ is an array type then
        $f_e \leftarrow$ array element field of $t_{arg}$;
        if AnalyzeField($f_e$) = StaticFixed then
            return RuntimeFixed;
        else return Variable;
    else
        $result \leftarrow$ StaticFixed;
        foreach field $f$ of type $t_{arg}$ do
            $tmp \leftarrow$ AnalyzeField($f$);
            if $tmp$ = Variable then return Variable;
            else if $tmp$ = RuntimeFixed then
                $result \leftarrow$ RuntimeFixed;
            end
        end
        return $result$;
    end
end

Function AnalyzeField($f_{arg}$)
    $result \leftarrow$ StaticFixed;
    foreach runtime type $t$ in $f_{arg}$.getTypeSet do
        $tmp \leftarrow$ AnalyzeType($t$);
        if $tmp$ = Variable then return Variable;
        else if $tmp$ = RuntimeFixed then
            if $f_{arg}$ is not final then return Variable;
            else $result \leftarrow$ RuntimeFixed;
        end
    end
    return $result$;
end
```

be classified as a VST. In this case, the function can immediately return without further traversing the graph.

We take the type `LabeledPoint` in Figure 1 as a running example. In Figure 5, every field has a typeset with a single element and the declared type of each field is equal to its corresponding runtime type except that the `features` field has a declared type (`Vector`), while its runtime type is `DenseVector`. Moreover, for a more sophisticated implementation of logistic regression with high-dimensional datasets, the `features` field can have both `DenseVector` and `SparseVector` in its typeset.

```
1 val a = input.readString().toInt() // a == Symbol(1)
2 val b = 2 + a - 1 // b == Symbol(1) + 1
3 val c = a + 1 // c == Symbol(1) + 1
4 if (foo()) array = new Array[Int](b)
5 else array = new Array[Int](c)
6 // array.length == Symbol(1) + 1
```

Fig. 6. An example of the symbolized constant propagation. $Symbol(1)$ is the `int` value read from the external data source at line 1. After symbolized constant propagation, Deca can determine that all `int` array instances created at line 4 and 5 have the same length ($Symbol(1) + 1$).

Since there is no cycle in the type dependency graph, `LabeledPoint` is not a recursively defined type. As shown in Figure 5, `LabeledPoint` contains a primitive field (i.e., `label`) and a field of the `Vector` type (i.e., `features`). Therefore, the *size type* of `LabeledPoint` is determined by the *size type* of `features`, i.e., the size type of `DenseVector`. It contains four fields: one of the array type and the other three of the primitive type. The `data` field will be classified as an RFST but not a VST due to its `final` modifier (`val` in Scala). Furthermore, the `DenseVector` objects assigned to `features` can have different *data-size* values because they may contain different arrays. Therefore, both `features` and `LabeledPoint` belong to VST.

3.3 Global Classification Analysis

The local classification algorithm is easy to implement and has negligible computational overhead. But it is conservative and often overestimates the variability of the target UDT. For example, the local classifier conservatively assumes that the `features` field of a `LabeledPoint` object may be assigned with `DenseVector` objects with different *data-size* values. Therefore, it mistakenly classifies it as a VST, which cannot be safely decomposed.

Furthermore, the local classifier assumes that the `DenseVector` objects contain arrays (`features.data`) with different lengths. Even if we change the modifier of `features` from `var` to `val`, i.e., only allowing it to be assigned once, the local classifier still considers it as an RFST rather than an SFST.

For UDTs that are categorized as RFST or VST, we further propose an algorithm to refine the classification results via global code analysis on the relevant methods of the UDTs. To break the assumptions of the local classifier, the global one uses code analysis to identify *init-only fields* and *fixed-length array type* according to the following definitions.

Init-only field. A field of a nonprimitive type `T` is *init-only* if, for each object, this field will only be assigned once during the program execution.[2]

Fixed-length array types. An array type `A` contained in the typeset of field `f` is a *fixed-length* array type w.r.t. `f` if all the `A` objects assigned to `f` are constructed with identical length values within a well-defined scope, such as a single Spark job stage or a specific cached RDD. An example of symbolized constant propagation is shown in Figure 6. Here, `array` is constructed with the same length for whatever `foo()` returns. The fixed-length array types with its element fields being SFST (or RFST) can be refined to SFST (or RFST).

In Figure 1, the `features` field is only assigned in the constructor of `LabeledPoint` (lines 1–8), and the length of `features.data` is a global constant value `D` (lines 14–16). Thus, the size-class of `LabeledPoint` can be refined to SFST.

[2] We always treat the array element fields as non-init-only; otherwise, the analysis needs to trace the element index value in each assignment statement, which is not feasible in static code analysis.

ALGORITHM 2: Global classification analysis

```
Input: The top-level nonprimitive type $T$; The locally classified size type $S_{local}$; Call graph of the
       current analysis scope $G_{call}$;
Output: The refined size type of $T$;
if SRefine($T, G_{call}$) then return StaticFixed;
else if $S_{local}$ = RuntimeFixed or RRefine($T, G_{call}$) then
    return RuntimeFixed;
else return Variable;
```

ALGORITHM 3: Static fixed-sized type refinement: SRefine(t_{arg}, g_{arg})

```
1 Function SRefine($t_{arg}, g_{arg}$)
      Input: A nonprimitive type $t_{arg}$; A call graph $g_{arg}$;
      Output: true or false that $t_{arg}$'s size type can be refined to StaticFixed;
2     foreach field $f$ of type $t_{arg}$ do
3         foreach runtime type $t$ in $f$.getTypeSet do
4             if $t$ is not a primitive type and not SRefine($t, g_{arg}$) then return false;
5         end
6     end
7     if $t_{arg}$ is an array type and $t_{arg}$ is not Fixed-Length in call graph $g_{arg}$ then return false;
8     else return true;
9 end
```

Algorithm 2 shows the procedure of the global classification. The input of the algorithm is the target UDT and the call graph of the current analysis scope. The refinement is done based on the following lemmas.

Lemma 3.1 (SFST Refinement). *An array type that is an RFST or a VST can be refined to an SFST if and only if for every array type in the type-dependent graph, the following points are true:*

(1) It is a fixed-length array type.
(2) Every type in the typeset of its element field is an SFST.

Lemma 3.2 (RFST Refinement). *An array type that is a VST can be refined to an RFST if and only if:*

(1) every type in the typesets of its fields is either an SFST or an RFST, and
(2) each field with an RFST in its typeset is init-only.

The call graph used for the analysis is built in the preprocessing phase (Section 6). The entry node of the call graph is the main method of the current analysis scope, usually a Spark job stage, while all the reachable methods from the entry node as well as their corresponding calling sequences are stored in the graph.

In line 7 of Algorithm 3, we use the following steps to identify the fixed-length array types: (1) Perform the copy/constant propagation in the call graph. The values passed from the outside of the call graph or returned by the I/O operations will be represented by symbols considered as constant values. (2) For a field `f` and an array type `A`, find all the allocation sites of the `A` objects that are assigned to `f` (i.e., the methods where these objects are created). If all the length values used in all these allocation sites are equivalent, `A` is of fixed length w.r.t. `f`.

```
// The first loop is the input loop.
var source = stage.getInput()
var sink = stage.nextCollection()
while (source.hasNext()) {
  val dataIn = source.next()
  ...
  val dataOut = ...
  sink.write(dataOut)
}
// Optional inner loops
source = sink
sink = stage.nextCollection()
while (source.hasNext()) {...}
...
// The last loop is the output loop
source = sink
sink = stage.getOutput()
while (source.hasNext()) {...}
```

Fig. 7. A typical task template of the Spark job stage. It consists of one or more top-level computation loops; each reads a data object from its source and writes the results into the sink. Every two successive loops are bridged by a data collector, such as an in-memory RDD block or a shuffle buffer.

ALGORITHM 4: Runtime fixed-sized type refinement: RRefine(t_{arg}, g_{arg})

```
1  Function RRefine($t_{arg}, g_{arg}$)
       Input: A nonprimitive type $t_{arg}$; A call graph $g_{arg}$;
       Output: true or false that $t_{arg}$'s size type can be refined to RuntimeFixed;
2      foreach field $f$ of type $t_{arg}$ do
3          $analyze\_field$ ← false;
4          foreach runtime type $t$ in $f.getTypeSet$ do
5              if $t$ is not a primitive type and not SRefine($t, g_{arg}$) then
6                  if RRefine($t, g_{arg}$) then
7                      $analyze\_field$ ← true;
8                  else return false;
9              end
10         end
11         if $analyze\_field$ and $f$ is not Init-Only in call graph $g_{arg}$ then return false;
12     end
13     return true;
14 end
```

In line 11 of Algorithm 4, we use the following rules to identify init-only or non-init-only fields: (1) a final field is init-only, (2) an array element field is not init-only, and (3) in addition, a field is init-only if it will not be assigned in any method in the call graph other than the constructors of its containing type, and it will only be assigned once in any constructor calling sequence.

3.4 Phased Refinement

In a typical data parallel programming framework, such as Spark, each job can be divided into one or more execution **phases**, each consisting of three steps: (1) reading data from materialized (on-disk or in-memory) data collectors, such as cached RDD; (2) applying a UDF on each data object, and (3) emitting the resulting data into a new materialized data collector. Figure 7 shows the framework of a job in Spark. It consists of one or more top-level computation loops; each reads a data object from its source and writes the results into the sink. Every two successive loops are bridged by a data collector, such as an RDD or a shuffle buffer.

We observe that the data sizes of object types may have different levels of variability at different phases. For example, in an early phase, data would be grouped together by their keys, and their values would be concatenated into an array whose type is a VST at this phase. However, once the resulting objects are emitted to a data collector, e.g., a cached RDD, the subsequent phases might not reassign the array fields of these objects. Therefore, the array types can be considered as RFSTs in the subsequent phases. We exploit this phenomenon to refine a data type's size class in each particular phase of a job, which is called *phased refinement*. This can be achieved by running the global classification algorithm for the VSTs on each phase of the job.

4 LIFETIME-BASED MEMORY MANAGEMENT

4.1 The Spark Programming Framework

Spark provides a functional programming API, through which users can process Resilient Distributed Datasets (RDDs), the logical data collections partitioned across a cluster. An important feature is that RDDs can be explicitly cached in the memory to avoid recomputation or disk I/O overhead.

While Spark supports many operators, the ones most relevant for memory management are some *key-based* operators, including `reduceByKey`, `groupByKey`, `join`, and `sortByKey` (analogs of GroupBy-Aggregation, GroupBy, Inner-Join, and OrderBy in SQL). These operators process data in the form of *Key-Value* pairs. For example, `reduceByKey` and `groupByKey` are used for (1) aggregating all *Values* with the same *Key* into a single *Value* and (2) building a complete *Value* list for each *Key* for further processing.

Furthermore, these operators are implemented using data shuffling. The shuffle buffer stores the combined value of each Key. For example, for the case of `reduceByKey`, it stores a partial aggregate value for each Key, and for the case of `groupByKey`, it stores a partial list of *Value* objects for each Key. When a new *Key-Value* pair is put into the shuffle buffer, eager combining is performed to merge the new *Value* with the combined value.

For each Spark application, a driver program negotiates with the cluster resource manager (e.g., YARN [46]), which launches executors (each with a fixed amount of CPU and memory resource) on worker machines. An application can submit multiple jobs. Each job has several *stages* separated by data shuffles, and each stage consists of a set of tasks that perform the same computation. Each executor occupies a JVM process and executes the allocated tasks concurrently in a number of threads.

4.2 Lifetimes of Data Containers in Spark

In Spark, all objects are allocated in the running executors' JVM heaps, and their references are stored in three kinds of *data containers* described below. A key challenge for Deca is deciding when and how to reclaim the allocated space. In the *lifetime* analysis, we focus on the endpoints of the lifetime of the object references. The lifetime of an object ends once all its references are dead.

UDF variables. Each task creates function objects according to its task descriptor. UDF variables include objects assigned to the fields of the function objects and the local variables of their methods. The lifetimes of the function object end when the running tasks complete. In addition, as long-living objects are recommended to be stored in cached RDDs, in most applications, local variables are dead after each method invocation. Therefore, we treat all the data objects referenced only by the local variables as short-living temporal objects.

Cache blocks. In Spark, each RDD has an object that records its data source and the computation function. Only the cached RDDs will be materialized and retained in memory. A cached RDD consists of a number of cache blocks, each being an array of objects. The lifetimes of cached RDDs are explicitly determined by the invocations of `cache()` and `unpersist()` in the applications. Whenever a cached RDD has been "unpersisted," all of its cache blocks will be released immediately. For noncached RDDs, the objects only appear as local variables of the corresponding computation functions and hence are also short-living.

Shuffle buffers. A shuffle buffer is accessed by two successive phases in a job: one creates the shuffle buffer and puts data objects into it, while the other reads out the data for further processing. Once the second phase is completed, the shuffle buffer will be released. With regard to the lifetimes of the object references stored in a shuffle buffer, there are three situations: (1) In a sort-based shuffle buffer, objects are stored in an in-place sorting buffer sorted by the Key. Once object references are put into the buffer, they will not be removed by the subsequent sorting operations. Therefore, their lifetimes end when the shuffle buffer is released. (2) In a hash-based shuffle buffer with a `reduceByKey` operator, the *Key-Value* pairs are stored in an open hash table with the *Key* object as the hash key. Each aggregate operation will create a new *Value* object while keeping the *Key* objects intact. Therefore, a *Value* object reference dies upon an aggregate operation over its corresponding *Key*. (3) In a hash-based shuffle buffer with a `groupByKey` operator, a hash table stores a set of *Key* objects and an array of *Value* objects for each *Key*. The combining function will only append *Value* objects to the corresponding array and will not remove any object reference. Hence, the references will die at the same time as the shuffle buffer. Note that these situations cover all the key-based operators in Spark. For example, `aggregateByKey` and `join` are similar to `reduceByKey` and `groupByKey`, respectively. Other key-based operators are just extensions of the above basic operators and hence can be handled accordingly.

4.3 Data Containers in Deca

As discussed above, object references' lifetimes can be bound with the lifetimes of their containers. Deca builds a data-dependent graph for each job stage by points-to analysis [25] to produce the mapping relationships between all the objects and their containers. Objects are identified by either their creation statements if they are created in the current stage or their source cached blocks if they are read from cached blocks created by the previous stage.

However, an object can be assigned to multiple data containers. For example, if objects are copies between two different cached RDDs, then they can be bound to the cached blocks of both RDDs. In such cases, we assign a sole *primary container* as the owner of each data object. Other containers are treated as *secondary containers*. The object ownership is determined based on the following rules:

(1) Cached RDDs and shuffle buffers have higher priority of data ownership than UDF variables, simply due to their longer expected lifetimes.
(2) If there are objects assigned to multiple high-priority containers in the same job stage, the container created first in the stage execution will own these objects.

In the rest of this subsection, we present how data are organized within the primary and secondary containers under various situations.

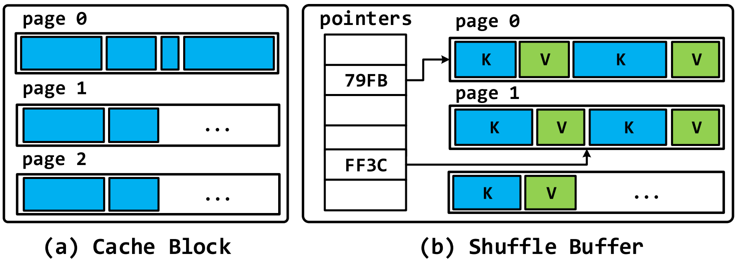


Fig. 8. Memory layouts of primary containers in Deca's memory manager.

4.3.1 Memory Pages in Deca. Deca uses unified byte arrays with a common fixed size as logical memory pages to store the decomposed data objects. A page can be logically split into consecutive byte segments, one for each top-layer object. Each of such segment can be further split into multiple segments, one for each lower-layer object, and so on. The page size is chosen to ensure that there is only a moderate number of pages in each executor's JVM heap so that the GC overhead is negligible. On the other hand, the page size should not be too large either, so that there would not be a significant unused space in the last page of a container.

For each data container, a group of pages are allocated to store the objects it owns. Deca uses a *page info* structure to maintain the metadata of each page group. The page info of each page graph contains (1) *pages*, a page array storing the references of all the allocated pages of this page group; (2) *endOffset*, an integer storing the start offset of the unused part of the last page in this group; and (3) *curPage* and *curOffset*, two integer values that store the progress of sequentially scanning or appending to this page group.

4.3.2 Primary Container. The way that Deca stores objects in their primary container depends on the type of the container:

UDF variables. Deca does not decompose objects owned by UDF variables. These objects do not incur significant GC overheads, because (1) the objects only referenced by local variables are short-living objects and they belong to the young generation, which will be reclaimed by the cheap minor GCs, and (2) the objects referenced by the function object fields may be promoted to the part of old generation, but the total number of these objects in a task is relatively small compared to the big input dataset.

Cache blocks. Deca always decomposes the SFST or RFST objects and stores their raw data bytes in the page group of a cache block. While cache blocks are designed to be immutable for the purpose of fault tolerance, VST objects can also be decomposed based on the typeset of fields. Figure 8(a) shows the structure of a cache block of a cached RDD, which contains decomposed objects.

A task can read objects from a decomposed cache block created in a previous phase. If this task changes the data sizes of these objects, Deca has to reconstruct the objects and release the original page group. To avoid thrashing, when such reconstruction happens, Deca will not re-decompose these objects again even if they can be safely decomposed in the subsequent phases.

Shuffle buffers. Figure 8(b) shows the structure of a shuffle buffer. Similar to cache blocks, data of an RFST or an SFST in a shuffle buffer will be decomposed into the shuffle buffer's page group. However, unlike cached RDD, where data are accessed in a sequential manner, data in a shuffle buffer will be randomly accessed to perform sorting or hashing operations. Therefore, as illustrated on the left-hand side of Figure 8(b), we use an array to store the pointers to the keys and values within a page. The hashing and sorting operations are performed on the pointer arrays. However, the pointer array can be avoided for a hash-based shuffle buffer with both the *Key* and the *Value* being of primitive types or SFSTs. This is because we can deduce the offsets of the data within the page statically.

As we discussed in Section 4.2, for a hash-based shuffle buffer with a GroupBy-Aggregation computation, a combining operation would kill the old *Value* object and create a new one. Therefore, *Value* objects are not long-living, and frequent GCs of these objects are generally unavoidable. However, if the *Value* object is of an SFST, then we can still decompose it and whenever a new object is generated by the combining operation, we can just reuse the page segment occupied by the old object, because the old and the new objects are of the same size. Doing this would save the frequent GCs caused by these temporary *Value* objects.

When the working set size is larger than the available memory space of an executor, Spark moves part of its data out of the memory. For cached RDDs, Spark uses the LRU strategy to select the cache blocks for eviction. The evicted data will be directly discarded or swapped to the local disk according to the user-specified *storage level*. For shuffles, Spark always spills the partial data into temporary files and merges them into final files at the end of task executions.

For cached RDDs, Deca modifies the original LRU strategy to evict page groups rather than cache blocks. Accessing in-page data through either page infos or pointers will refresh the corresponding page group's recently used counter. Spark serializes cache block data before writing them into disk files or transferring them through the network for nonlocal accesses. In Deca, the decomposed data bytes can be directly used for disk and network I/O.

For shuffles, like Spark, Deca sorts the pointers before spilling and writes the spilled data into files according to the order of the pointers. If a shuffle buffer has only pointers that reference page segments, Deca does not spill these pointers because normally they only occupy a small memory space. It pauses the shuffling and triggers the cache block eviction to make enough room. Deca uses a small memory space (normally only one page) to merge sorted spilled files. Once the merging space is fully filled, the merged data will be flushed to the final output file.

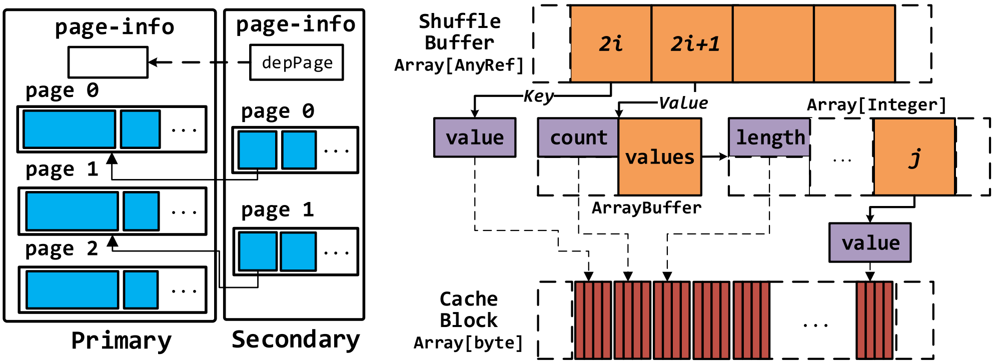
4.3.3 *Secondary Container.*

There are common patterns of multiple data containers sharing the same data objects in Spark programs, such as (1) manipulating data objects in cache blocks or shuffle buffers through UDF variables, (2) copying objects between cached RDDs, (3) immediately caching the output objects of shuffling, and (4) immediately shuffling the objects of a cached RDD.

If a secondary container is UDF variables, it will be assigned pointers to page segments in the page group of the objects' primary container. Otherwise, Deca stores data in the secondary container according to the following two different scenarios: (1) fully decomposable, where the objects can be safely decomposed in all the containers, and (2) partially decomposable, where the objects cannot be decomposed in one or more containers.

Fully decomposable. This scenario is illustrated in Figure 9(a). To avoid copy-by-value, a secondary container only stores the pointers to the page group owned by the primary, one for each object. Furthermore, we add an extra field, *depPages*, to the page info of the secondary container to store the page info(s) of the primary container(s).

Deca further performs optimizations for a special case, where a secondary container stores the same set of objects as the primary and does not require a specific data ordering. In such a case, Deca only generates a copy of the page info of the page group owned by the primary container and stores it in the secondary container. In this way, both containers actually share the same page group. The memory manager uses a reference-counting method to reclaim memory space. Creating a new page info of a page group increments its reference counter by one, while destroying a container



(a) Fully Decomp. (b) Partially Decomp.

Fig. 9. Examples of memory layout in Deca when data objects have multiple containers: (a) data objects can be decomposed in both the primary container and the secondary container; (b) data objects cannot be decomposed in its primary container (shuffle buffer) but are decomposable in its secondary container (cache block).

(and its page info) does the opposite. Once the reference counter becomes zero, the space of the page group can be reclaimed.

Partially decomposable. In general, if the objects cannot be safely decomposed in one of the containers, then we cannot decompose them into a common page group shared by all the containers. However, if the objects are immutable or the modifications of objects in one container do not need to be propagated to the others, then we can decompose the objects in some containers and store the data in their object form in the nondecomposable containers. This is beneficial if the decomposable containers have long lifetimes.

Figure 9(b) depicts a representative example, where the output of a groupByKey operator, implemented via a hash-based shuffle buffer, is immediately cached in an RDD. Here, groupByKey creates an array of *Value* objects in the hash-based shuffle buffer (see the middle of Figure 9(b)), and then the output is copied to the cache blocks. The *Value* array is of a VST and hence cannot be decomposed in the shuffle buffer. However, in this case, the shuffle buffers would die after the data are copied to the cache blocks, and the subsequent modifications of the objects in the cache blocks do not need to be propagated back to the shuffle buffers. Therefore, as shown in Figure 9(b), we can safely decompose the data in the cache blocks, which have a long lifetime, and hence significantly reduce the GC overhead.

5 DECOMPOSITION OF VST AND FIELD-ORIENTED MEMORY PAGES

5.1 Decomposition of VST

In the previous sections and also in [28], only decomposing of SFST and RFST objects is considered. The key reason we do not decompose a VST object is that the memory manager cannot safely allocate a fixed space for it due to its variable size. Overallocating space to account for the maximum possible size of a VST object would incur prohibitive waste of memory space, especially when the sizes of the objects of the same VST vary a lot.

Fortunately, in JVM, the runtime change of the actual size of an object is usually implemented by creating a new object with a new size. In other words, the size of any object is actually of fixed size during its lifetime. For example, Figure 10 shows the typical VST used in a groupByKey operator. As mentioned in Section 4.3.1, groupByKey creates an array of *Value* objects in a hash-based shuffle buffer (see the middle of Figure 10). The *Value* objects whose keys have the same hash values are

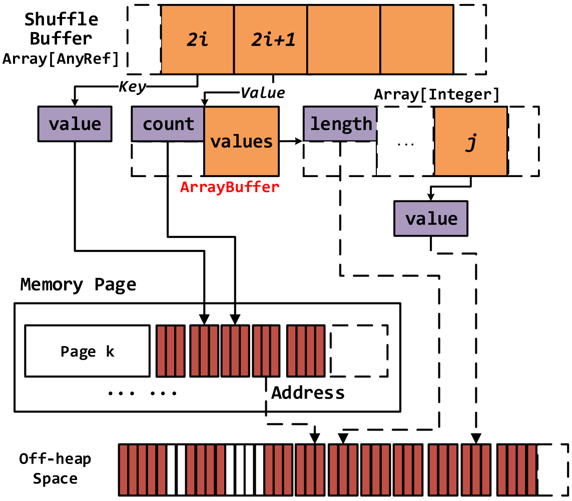


Fig. 10. The memory layout of data containers for VST decomposition.

put into *Values*, which is an array. When the current array is full, a new array object with a larger size is created and used to replace the old one.

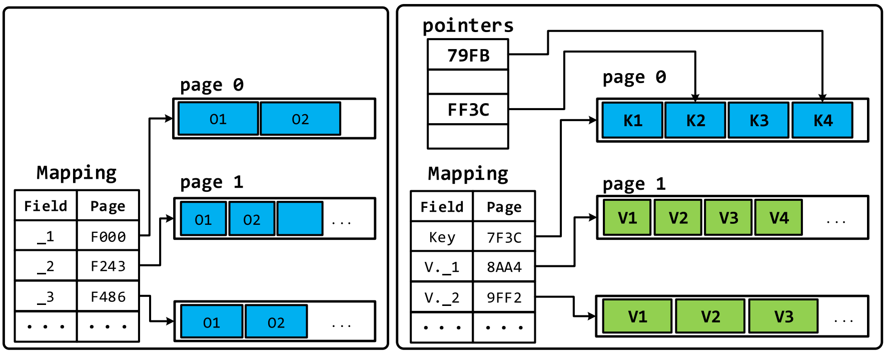
This means that JVM actually creates several fixed-size objects during the lifetime of a VST object. Deca makes use of this property and decomposes each of such fixed-size objects. If we store these objects in the heap, it would result in frequent (but lightweight) GCs. To avoid this overhead, Deca stores a decomposed VST object in the off-heap space by using *sun.misc.Unsafe* to allocate the necessary space. We maintain a pointer in the memory page to store the address of the decomposed object in the off-heap space. When JVM creates a new object caused by a size change of the VST object, Deca frees the off-heap space of the old object via Unsafe and allocates the necessary off-heap space for the new one. The address in the memory page would be updated accordingly. Note that to replace the old object with a new one, if the original program contains the copy statements, such as the array mentioned above, we copy the original bytes to new space additionally. This technique works only if the VST object is appropriately encapsulated as stated in the following lemma.

Lemma 5.1 (Safe VST). *The object of a VST can be safely decomposed if and only if all the following conditions hold:*

(1) *It is a private field.*
(2) *The new object reassigned to the field cannot be built outside of the class.*
(3) *The reference cannot be assigned to other fields.*

The conditions of the lemma are consistent with encapsulation rules in object-oriented languages. A class with good encapsulation designs should protect the important fields from being changed outside of the class. When the VST field satisfies Lemma 5.1, the invocations of the method that reassigns the field means the end of the current object's lifetime. On the contrary, without good encapsulation, the VST object can be assigned anywhere in the code; therefore, it is computationally infeasible to analyze its lifetime.

Lastly, in Spark, cached RDDs are designed to be immutable to provide resiliency. In other words, objects stored in a cache container would never be updated, including their sizes, and hence can



(a) Cache Block (b) Shuffle Buffer

Fig. 11. Memory layouts of data containers with field-oriented memory pages.

always be safely decomposed. Therefore, we only need to use the method introduced here to decompose the VST objects stored in shuffle buffers, which are mutable.

5.2 Field-Oriented Memory Page

In the aforementioned design of memory pages, the UDT objects are stored one by one in a contiguous region. We call them object-oriented memory pages. When the data processing system caches a large number of intermediate objects, it is often desirable to compress the in-memory cached objects in order to maximize the usage of memory space and avoid swapping data to secondary disks. As the byte sequence of one UDT object contains different types of fields, its *information entropy* is high, which would have poor compression rates [54, 53].

To enhance the compression performance, we adopt field-oriented memory pages, which have a lower information entropy. A field-oriented memory page only stores one field of a UDT. In other words, a UDT data object may be split and stored in a few memory pages. A mapping table is needed to manage the map from a field to the corresponding memory pages. The mapping table can be produced by Algorithm 1 in local classification (Section 3). The algorithm runs recursively until it reaches a primitive type or an array type with primitive elements. We then allocate one page group to each of these fields. The name of this field and the address of the first allocated memory page are put into the mapping table. Based on the mapping table, the structure of a field-oriented memory page is designed as follows.

Cache blocks. With the mapping table, the UDT objects are decomposed into several pages, as shown in Figure 11(a).

Shuffle buffers. Figure 11(b) shows the structure of a shuffle buffer. Besides the mapping table, a pointer array is used for sorting or hashing operations, which is similar to the object-oriented memory pages. As the values of all the fields in a UDT have the same offset in their memory pages, the pointer array only needs to point to the first memory page. If the sorting operations compare other fields, they can also be accessed by the same offset.

Note that we treat a field of an array type as two fields: a length field storing the length of the array and an element field being of the same type as the array elements. Thus, if the array is a decomposable VST, Deca allocates two memory pages to the array type with the memory pages of the element field storing the addresses of the actual off-heap spaces allocated to the arrays. If the array type is of SFST or RFST, we simply store the length field and the element field together into one memory page group instead of two, which will have little effect on the compression rate but can simplify the operations on the arrays.

6 IMPLEMENTATION

We implement Deca based on Spark in roughly 12,800 lines of Scala code. It consists of an optimizer used in the driver and a memory manager used in every executor. The memory manager allocates and reclaims memory pages. It works together with the Spark cache manager and shuffle manager, which manage the undecomposed data objects.

6.1 Optimizer in Deca

The optimizer analyzes and transforms the code of each job when it is submitted in the driver. Intuitively, Deca can be implemented as a standalone tool that transforms the compiled jar files of a Spark program before its execution. However, a Spark driver program may execute many jobs, each consisting of several stages separated by shuffles. The job submission will be implicitly triggered by an *action*, such as `reduce`, which returns a value to the driver after running a UDF on a dataset. According to the results returned by the current job, the driver decides how to submit the next job.

A driver program can freely use the control statements (if/for/while) to control the computation. Therefore, it may submit different jobs with different input datasets and configuration parameters. A static optimization has to enumerate and process all the possible jobs by exhaustively exploring a large number of possible execution paths of the program, which is the well-known *path explosion* problem [37]. This is infeasible especially when the program has loop structures, which render the number of execution paths unbounded.

To address these challenges, we implement Deca in a hybrid way, which contains a static analyzer and a runtime optimizer. The static analyzer extracts a priori knowledge about the UDFs and UDTs of the target programs, which can be used to reduce the runtime optimization overheads. The runtime optimizer intercepts the submitted jobs at runtime and optimizes each job before actually submitting it to the Spark platform. With this approach, Deca only optimizes the actually submitted jobs, and thereby completely eliminates the need for exploring all the execution paths.

6.2 Implementation of Optimizer

The Deca optimizer uses the Soot framework [42] to analyze and manipulate the Java bytecode. The Soot framework provides a rich set of utilities that implement the classical program analysis and optimization methods.

In the preprocessing phase, Deca uses *iterator fusion* [33] to bundle the iterative and isolated invocations of UDFs into larger, hopefully optimizable code regions to avoid the complex and costly interprocedural analysis. The per-stage call graphs and per-field typesets are also built using Soot in this phase. Building per-phase call graphs will be delayed to the analysis phase if a phased refinement is necessary. In the analysis phase, Deca uses the methods described in Section 3 and Section 4 to determine whether and how to decompose particular data objects in their containers.

Based on the obtained decomposability information, in each stage, for the data objects of the UDT that can be safely decomposed, Deca transforms the corresponding code and leaves the nonoptimizable part unchanged. The transformation phase can be further split into two subphases, decomposition and linking, which are described below.

6.2.1 Preprocessing. In this phase, Deca mainly performs *iterator fusion* to avoid interprocedural analysis, which simplifies the analysis of Spark codes. First, for the code of a Spark job, Deca builds a DAG based on RDD's lineage graph. We make use of the implementation of `DAGScheduler` in Spark to divide the DAG built above into several stages according to the shuffle operations in the job. Each stage is represented as a stage DAG. Second, Deca determines the hierarchical structure of the loop body and uses Soot to generate the corresponding loop code. If cached RDDs exist in a stage, Deca generates new loop bodies to manage data objects in the cache buffer. Third, we need to extract the UDFs to fill the loop body. By comprehending the semantic of the RDD and the operators like `filter`, `flatMap`, and `mapValues`, Deca uses Java reflection to extract UDFs and then inlines these UDFs in the loop body. Here, we need to guarantee the consistency between the return type of the previous UDF and the next parameter type of the next UDF. Finally, Deca generates a `Stage-Function-Class`, which is similar to the class structure of UDF, and puts the loop body into this class. The detailed structure of `Stage-Function-Class` is displayed in Appendix A. Moreover, as for closures in UDF, they are created at the driver, and we choose to inject them into the final UDF object we have transformed.

6.2.2 Analysis. In this phase, in order to provide the important information of UDTs for code transformation, Deca uses Soot to perform points-to analysis in `Stage-Function-Class` and then analyzes the results of points-to analysis to guide UDT classification. First, the `Stage-Function-Class` after performing *iterator fusion* is an independent class, which cannot be analyzed by soot-SPARK, a built-in points-to analysis tool in Soot. To address this problem, Deca generates an entry method and an entry class linking to the `Stage-Function-Class` temporarily to make soot-SPARK applicable. Second, we modify the implementation of soot-SPARK to realize full-context sensitivity, field sensitivity, and object sensitivity. It is worth noting that soot-SPARK needs to load classes integrally, but the `Stage-Function-Class` may get involved in many irrelevant classes. So it is time-wasting to load all these classes. In our modification of soot-SPARK, we load classes, fields, and methods incrementally during building the call graph on the fly, and hence the time overhead of UDT decomposition and code transformation can be reduced significantly. Third, based on the result of points-to analysis, Deca invokes both local and global analysis to determine the types of UDTs in the shuffle buffer and cache buffer. If we cannot decompose the UDT safely, Deca will simply terminate subsequent optimization.

For the safety of transformation, Deca also determines whether there is object sharing in the target stage program. There are three possible object sharing cases: intercontainer sharing, intracontainer sharing, and intra-top-level object sharing. Intercontainer sharing is detected by the object-container mapping analysis described in Section 4.3. The pointer-based memory layout for this case is also described in Section 4.3.

Based on the information from points-to analysis, Deca applies escape analysis over the loop body to detect intracontainer sharing. If the program assigns any local objects of a substage loop-body to a UDF variable or a (nested/nonnested) field of the object contained in a UDF variable, Deca conservatively assumes there would be object sharing at runtime. For intra-top-level object sharing, if there is any object assigned to different (nested/nonnested) fields of a top-level data object, Deca assumes there should be object sharing. Whenever intracontainer sharing or intra-top-level object sharing is detected, Deca gives up performing program transformation.

6.2.3 Decomposition. In this phase, for the UDTs that can be decomposed safely, Deca transforms the UDT methods to directly access the bytes in the memory pages. Deca generates a synthesized class for each target UDT (called **SUDT**). Logically, the `this` reference of a decomposed UDT object is transformed to the start offset (the index of the first byte of its raw data) of its containing memory page. All the bytecodes accessing the fields of this object are transformed to access the memory pages based on the absolute field offset (`object_start_offset + relative_field_offset`); the fields also include each field defined in the parent classes of the UDT. The offset computation depends on the raw data size of each UDT instance. In each SUDT, Deca synthesizes some `static` fields or methods to obtain the values of the data sizes of all the UDT fields. While the data sizes of the primitive type fields are defined in the official JVM specification, the sizes of fields of nonprimitive types can be recursively obtained from the corresponding SUDTs. If a field's data size can be determined statically, then it is stored as a global constant value in a static field in the corresponding SUDT. Otherwise, Deca synthesizes a static method of the SUDT to compute the data size at runtime. Similarly, for each UDT, Deca synthesizes some `static` fields or methods to provide the relative offsets of all the UDT fields in the SUDT. The relative offsets can be computed based on the data sizes and the ordering of the fields in the UDT definition. As an optimization technique, we reorder the UDT fields by putting the fields with statically determinable sizes in the front, and hence enable more fields' offsets to be determined statically.

After doing the above on all the fields, Deca transforms the methods of the UDT into its SUDT. During the transformation, all the field-accessing codes in the methods are replaced with the array-accessing code. At this point, transforming the UDT fields and UDT methods to the array-accessing codes is similar to Facade [35]. What distinguishes Deca from Facade is that we flatten all the fields of a UDT in the synthesized class to realize a flat memory layout and remove the references structure in UDT, which still exists in the transformed classes in Facade, so that the offset of each UDT instance or a field of the UDT instance can be computed accurately and statically.

The memory pages storing the decomposed objects can be stored in two ways in Deca: on-heap and off-heap. We can allocate the memory space for each page by creating an object on the heap. At the end of the lifetime of all the objects in the page, we simply discard the references to the page and rely on the GC to manage the space reclamation. The advantage of this method is its ease of implementation. It is convenient to define the memory page class with all the necessary data access methods, and the allocation of memory space can be simply performed by instantiating the page objects. Furthermore, as the number of memory pages are most likely to be relatively small, the GC overhead will be low.

The off-heap memory can be completely managed with `Unsafe`, which contains operations such as `allocate()`, `put()`, `release()`, and so forth. As managing the off-heap memory space will not incur any GC overhead, it is expected that using this approach in Deca has the lowest GC cost.

Another important thing to take into account is sharing information among the stages in a job. Some stages may need synthesized classes generated by previous stages to transform their own bytecode. So we cache the synthesized classes of all transformed stages if necessary.

6.2.4 Linking. In this phase, Deca will try to ensure the new RDD and the job that contains the synthesized classes can be executed on Spark successfully. Deca processes the code of the `Stage-Function-Class` of each stage and the other involved classes that use the synthesized classes generated in the decomposition phase. Deca traverses the Stage-Function-Class and detects which statement and reachable method referenced use the synthesized classes, then traces the new site of each synthesized class. If the new site of the synthesized class is included in the results provided by points-to analysis, it means that the objects of the synthesized class exist in a data container (shuffle buffer or cache buffer) or objects are only temporary. As for the temporary objects of the synthesized classes, Deca creates temporary memory pages to store them. Since these temporary objects are only referenced by local variables and will be reclaimed by minor GCs, Deca creates and reclaims such temporary pages at the start and the end of a loop body, respectively.

In Spark, every task processes data objects sequentially in each UDT object array in a loop. Since Deca allocates byte arrays for data caching and shuffling instead of object arrays as in vanilla Spark, we have to calculate the array index values in the loop based on the raw data sizes of each object in the optimized code. In fact, the array index variable stores the value of the start offset of the data object currently being processed. Based on these new array index values, Deca changes the code of `Stage-Function-Class` and other involved classes that use the synthesized classes with the following steps: (1) remove the invocations of the UDT object constructors and directly write the initial values of the constructor parameters into the byte arrays based on the absolute field offset; (2) replace all the field-accessing codes with the array-accessing code; and (3) replace each invocation of a UDT method with the corresponding SUDT method and add the byte array and the start offset as the additional parameters of the invocation.

Finally, when the transformation of all the stages of a job finishes, Deca compresses `Stage-Function-Class` and all classes involved into a jar file and transfers the jar file to each executor to ensure the job runs effectively. Then Deca creates a new RDD, injects a function object created by `Stage-Function-Class` into a new RDD for each stage, and resubmits the job that contains the DAG formed by the new RDDs.

Since executing an iterative application may involve submitting a massive amount of jobs to Spark, it is time consuming (and wasting) to repeat the aforementioned optimization for every submitted job. To solve the problem, Deca stores in memory the RDD-DAG structure, the UDFs of each stage in the very first job, and the synthetic `Stage-Function-Class` of each stage. Before optimizing a stage, if Deca detects that an equivalent stage (by comparing their RDD-DAG structure, and UDFs) has already been optimized, it can simply reuse the `Stage-Function-Class` stored in memory.

We present the optimized code corresponding to the `Stage-Function-Class` of LR in Figure 20, which illustrates more details of Deca's optimization.

7 EVALUATION

We use five nodes in the experiments, with one node as the master and the rest as workers. Each node is equipped with two eight-core Xeon-2670 CPUs, 64GB memory, and one SAS disk, running RedHat Enterprise Linux 5 (kernel 2.6.18) and JDK 1.7.0 (with default GC parameters). We compare the performance of Deca with Spark 1.6. For serializing cached data in Spark, we use Kryo, which is a very efficient serialization framework. All the experiments are repeated 5 times and we report the average execution time, garbage collection time, and their standard deviation (SD).

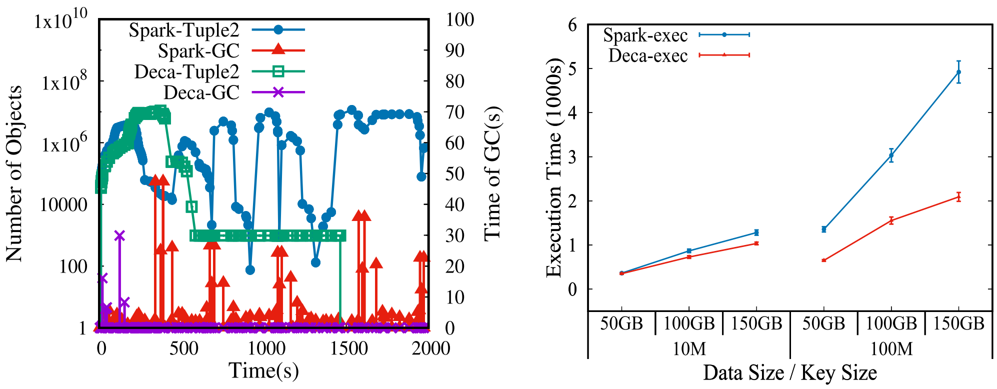
Five typical benchmark applications in Spark are evaluated in these experiments: *WordCount* (WC), *LogisticRegression* (LR), *KMeans*, *PageRank* (PR), and*ConnectedComponent* (CC). As shown in Table 1, they exhibit different characteristics and hence can verify the system's performance in various different situations. For WC, we use the datasets produced by Hadoop *RandomWriter* with different unique key numbers (10M and 100M) and sizes (50GB, 100GB, and 150GB). LR and KMeans use 4,096-dimension feature vectors (40GB and 80GB) extracted from the Amazon image dataset [30] and randomly generated 10-dimension vectors (ranging from 40GB to 200GB). For PR and CC, we use three real graphs: LiveJournal social network [5] (2GB), webbase-2001 [8] (30GB), and a 60GB graph generated by HiBench [22]. The maximum JVM heap size of each executor is set to be 30GB for the applications with only data caching or data shuffling, and 20GB for those with both caching and shuffling.

7.1 Impact of Shuffling

WC is a two-stage MapReduce application with data shuffling between the "map" and "reduce" stages. We examine the lifetimes of data objects in the shuffle buffers with the smallest dataset. We periodically record the alive number of objects and the GC time with JProfiler 9.0. The result is shown in Figure 12(a). WC uses a hash-based shuffle buffer to perform eager aggregation, which is implemented in `Tuple2`. The number of `Tuple2` objects, which fluctuates during the execution, can indicate the number of objects in shuffle buffers. While the number of `Tuple2` objects is also

Table 1. Applications Used in the Experiments

Application	Stages	Jobs	Cache	Shuffle
WC	two	single	non	aggregated
LR	single	multiple	static	non
KMeans	two	multiple	static	aggregated
PR/CC	multiple	multiple	static	grouped/aggregated



(a) Live data object numbers and GC time.

(b) Application execution time.

Fig. 12. Comparison of performance for the shuffle-only *WordCount* application of Spark and Deca.

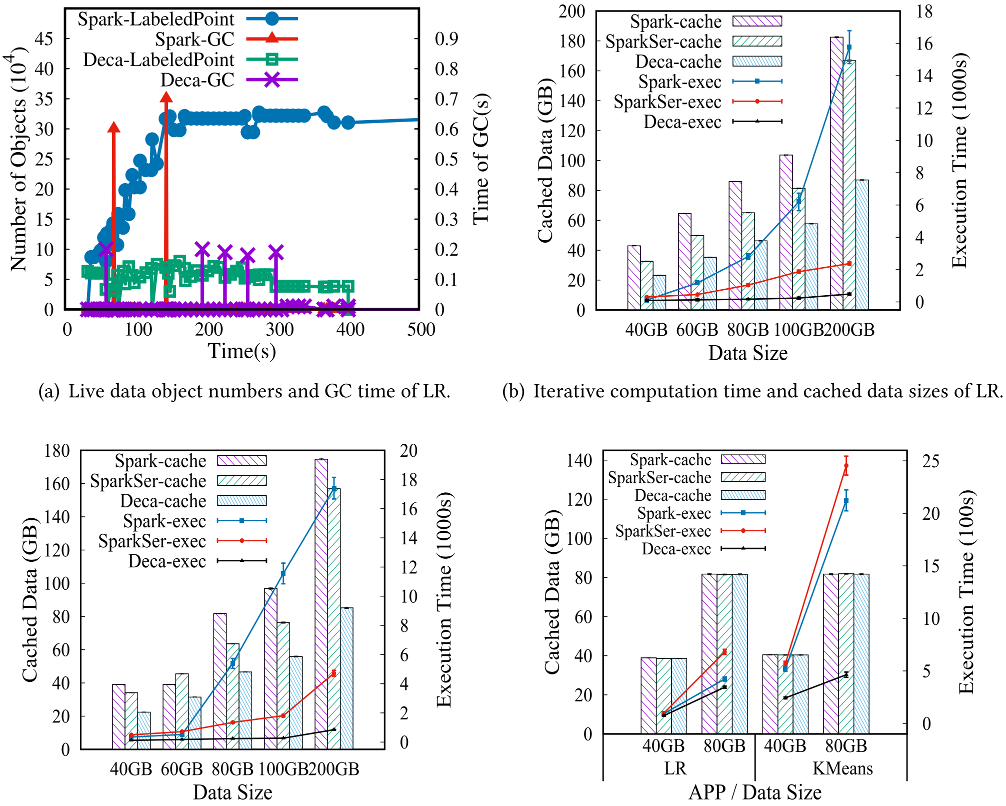
large in the "map" stage but decreases in shuffle in Deca, GCs are triggered frequently to release the space occupied by the temporary objects in the shuffle buffers.

To avoid such frequent GC operations, Deca reuses the space occupied by the partially aggregated *Value* for each *Key* in the shuffle buffer. Figure 12(b) compare the execution times of Deca and Spark. In all cases, Deca can reduce the execution time by 10% to 58%. One could also see that the performance improvement increases with a higher number of keys. This is because the size of a hash-based shuffle buffer with eager aggregation mainly depends on the number of keys. The reduction of GC overhead would become more prominent with a larger number of keys. Furthermore, since Deca stores the objects in the shuffle buffer as byte arrays, it also saves the cost of data (de-)serialization by directly outputting the raw bytes.

7.2 Impact of Caching

LR and KMeans are representative machine-learning applications that perform iterative computations. Both of them first load and cache the training dataset into memory, then iteratively update the model until the predefined convergence condition is met. In our experiments, we only run 30 iterations. We do not account for the time to load the training dataset, because the iterative computation dominates the execution time, especially considering that these applications can run up to hundreds of iterations in a production environment. We set 90% of the available memory to be used for data caching.

We first examine the lifetimes of data objects in cache RDDs for LogisticRegression (LR) using the 40GB dataset. The result is shown in Figure 13(a). We find that the number of objects is rather stable throughout the execution in Spark, but full GCs have been triggered several times in vain (the peaks of the GC time curve). This is because most objects are long-living and hence their



(a) Live data object numbers and GC time of LR.

(b) Iterative computation time and cached data sizes of LR.

(c) Iterative computation time and cached data sizes of KMeans.

(d) Iterative computation time and cached data sizes of LR and KMeans using Amazon Image Dataset.

Fig. 13. Comparison of performance for the caching-only *LogisticRegression* and *KMeans* applications of Spark and Deca.

space cannot be reclaimed. While these objects are less in Deca because they are transformed to bytes after being read from the HDFS, some objects still live in an old generation of the JVM heap because no full GC is active.

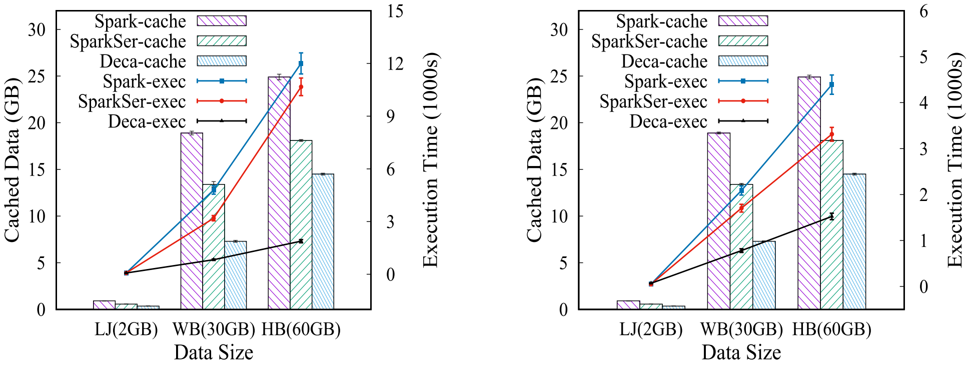
By grouping massive objects with the same lifetime into a few byte arrays, Deca can effectively eliminate the GC problem of repeatedly scanning alive data objects for their liveness. Figure 13(b) and Figure 13(c) show the execution times of LR and KMeans for both Deca and Spark. Here we also examine the cases using Kryo to serialize the cached data in Spark, which is denoted as “SparkSer” in the figures.

For the 40GB and 60GB datasets, the improvement is moderate and can be mainly attributed to the elimination of object creation and minor GCs. In these cases, the memory is sufficient to store the temporary objects, and hence full GC is rarely triggered. Furthermore, serializing the cached data also helps in reducing the GC time. Therefore, with the 40GB dataset, SparkSer outperforms Spark by reducing the GC overhead. However, for larger datasets, the overhead of data (de-)serialization cannot pay off the reduced GC overhead. Therefore, simply serializing the cached data is not a robust solution.

For the three larger datasets the improvement is more significant. The speedups of Deca range from 16× to 41.6×. In these datasets, the long-living objects consume almost all available memory

Table 2. Graph Datasets Used in *PageRank* and *ConnectedComponent*

Graph	LiveJournal (LJ)	WebBase (WB)	HiBench (HB)
Vertices	4.8M	118M	602M
Edges	68M	1B	2B
Data size	2GB	30GB	60GB



(a) Iterative computation time and cached data sizes of PR. (b) Iterative computation time and cached data sizes of CC.

Fig. 14. Comparison of performance for the mixed shuffling and caching *PageRank* and *ConnectedComponent* applications of Spark and Deca.

space, and therefore full GCs are frequently triggered, which just repeatedly and unavailingly trace the cached data objects in the old generation of the JVM heap. With the 100GB and 200GB datasets, the additional disk I/O costs of cache swapping also prolong the execution times of Spark. Deca keeps a smaller memory footprint of cached data and swaps a smaller portion of data to the disks.

We also conduct the experiments on a real dataset, the Amazon image dataset with 4,096 dimensions. Figure 13(d) shows that the speedups achieved by Deca range from 1.2× to 5.3×. With such a high-dimensional dataset, the size of object headers becomes negligible and therefore, the sizes of the cached data of Spark and Deca are nearly identical.

7.3 Impact of Mixed Shuffling and Caching

PageRank (PR) and ConnectedComponent (CC) are representative iterative graph computations. Both of them use `groupByKey` to transform the edge list to the adjacency lists and then cache the resulting data. We use three datasets with different edge numbers and vertex numbers as shown in Table 2. We set 40% and 100% of the available heap space for caching and shuffling, respectively. Edges will be cached during all iterations, and shuffling is used in every iteration to aggregate messages for each target vertex. We run 10 iterations in all the experiments.

Figure 14(a) and Figure 14(b) show the execution times of iterative computation without the time to load the dataset, and we only run 10 iterations for PR and CC of both Spark and Deca. The data-loading phase uses the `groupByKey` operator for transforming the input edge-list data to the adjacency-list data, which are cached in memory. The speedups of Deca range from 1.1× to 6.4×, which again can be attributed to the reduction of GC overhead and shuffle serialization overhead. However, it is less dramatic than the previous experiment. This is because each iteration of these applications creates new shuffle buffers and releases the old ones. Then GC may be triggered to reclaim the memory occupied by the shuffle buffers that are no longer in use. This reduces the

Table 3. Comparison of GC Time Reductions for Five Applications of Deca

App	**Spark**					**Deca**		
	exec. (s)	**SD (s)**	**GC (s)**	**SD (s)**	**Ratio**	**GC (s)**	**SD (s)**	**Reduction**
WC: 150GB	4,885	250	2,001.3	136.8	40.1%	11.8	0.9	99.4%
LR: 80GB	2,899	158	2,032.8	139.2	70.1%	2.1	0.23	99.9%
KMeans: 80GB	5,546	290	4,392.7	240.1	79.2%	7.5	0.42	99.8%
PR: 30GB	5,432	278	3,512.6	180.1	64.7%	23.2	1.6	99.3%
CC: 30GB	2,136	118	1,498.7	83.8	70.1%	38.6	2.4	97.4%

memory stress of Spark. We also see that SparkSer, which simply serializes the cache in Spark, has little impact on the performance. The additional (de-)serialization overhead offsets the reduction of GC overhead.

7.4 GC Improvement

Table 3 shows the times to run GC and the ratios of the GC time to the entire job execution time (the entire time of 10 iterations for PR and CC) for the seven applications. For each application, we only present the case where the input dataset is the largest subject to the condition that there is no data swapping or spitting, in order to avoid the impact of disk I/O operations on the execution time. For each case, the GC time listed in the table is the average of the GC times by all executors. The result demonstrates the effect of GC elimination implemented by Deca and how it improves the overall application performance.

As shown in the result, the GC running times of LR and KMeans, which are 70.1% and 79.2%, respectively, occupy the largest portion of the total execution time among all cases. With the 80GB input dataset, the cached data objects consume almost all the memory space of the old generation of the JVM heap. Deca reduces the GC running time in two ways: (1) smaller cache datasets trigger much less full GCs, and (2) once a full GC is triggered, the overhead of tracing objects is significantly reduced.

Since all the other applications have the shuffle phases in their executions, the disk and network I/O time accounts for a certain portion of the total execution time. Furthermore, a consequence of reserving the memory space for the shuffle buffers is that the long-living cached objects occupy no more than 60% of the total available memory. Therefore, in these cases, the GC running time ranges from 40.1% to 79.2%. This explains the reason that different types of applications reported above demonstrate different improvement ratios.

Next, we compare Deca with the GC tuning methods. The Spark document [43] states that it is an effective GC tuning method to adjust the ratio of the memory allocated to cache blocks to the memory allocated to shuffle blocks. Furthermore, we also compare with two GC algorithms available in Hotspot JVM, i.e., CMS and G1. Table 4 shows the results. LR is very sensitive to GC tuning. By setting the ratios between the memory allocations for cache and shuffle buffers to be 0.6 and 0.4 (which are the optimal settings based on our experiments), respectively, or replacing PS with CMS or G1 with tuned parameters, Deca can significantly improve the performance of LR. However, PR is much less sensitive to GC tunings, which is consistent with the experiments reported previously [13]. However, we cannot achieve the same performance gain by setting a higher number of concurrent GC threads in G1 as reported in [13]. We conjecture that it may be because of the difference between the machine configurations, which is not stated in [13]. This experiment result indicates that GC tuning is an effective way to improve the GC performance for some applications. However, it is a cumbersome process and is highly dependent on the applications and the system environment.

Table 4. Comparison of Performance Improvements of Spark LR/PR with Different GC Tuning Configurations

App	Storage Fraction					GC Algorithm				
	Frac.	Exec. (s)	SD (s)	GC (s)	SD (s)	Algo.	Exec. (s)	SD (s)	GC (s)	SD (s)
LR: 80GB Deca: 152s/ 1.6s	0.8:0.2	2,432	148	1,901	117	PS	3,216	172	2,308	136
	0.6:0.4	448	26	31	1.9	CMS	405	23	50	2.7
	0.4:0.6	616	34	20	3	G1	348	21	22	1.3
PR: 30GB Deca: 828s/ 21.7s	0.4:1.0	5,498	312	3,601	203	PS	5,533	292	3,518	192
	0.1:1.0	3,688	205	1,498	83	CMS	6,523	342	3,598	195
	0.0:1.0	3,756	196	1,455	81	G1	7,408	392	2,042	128

Deca performance with the default GC setting is used as the baseline.

Table 5. Comparison of Performance for LR and PR with Different Executor Memory Sizes

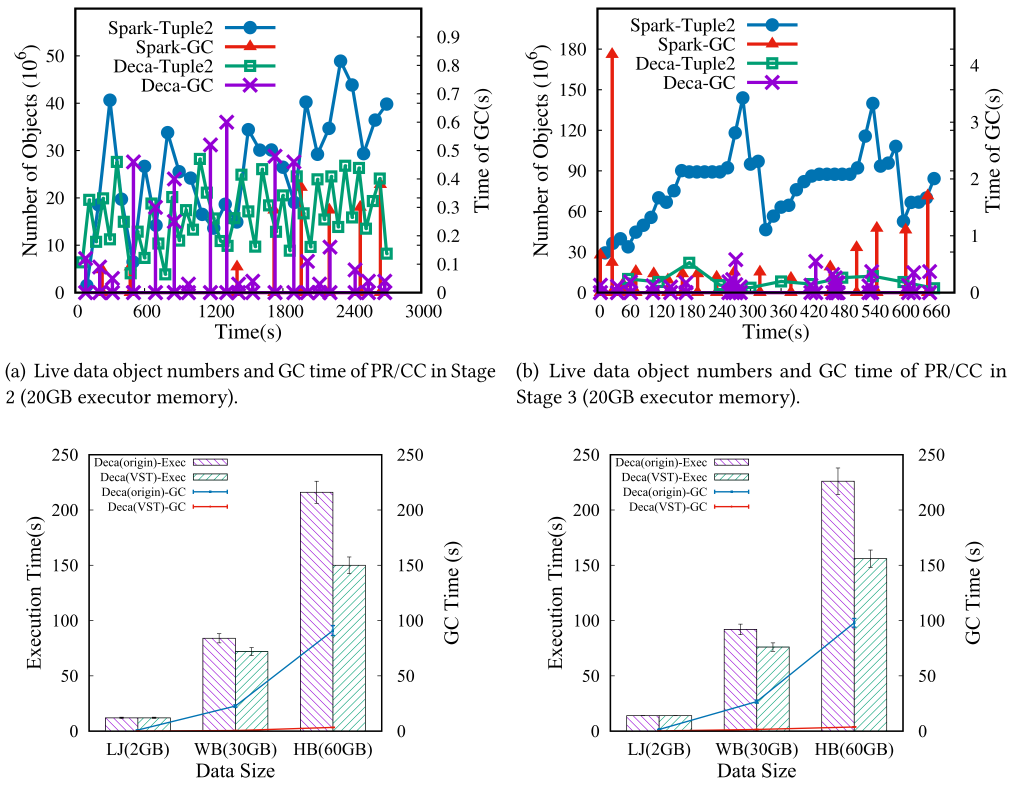
App	System	Executor Memory (GB)	Cache Rate	Execute Time (s)	SD (s)	GC Time (s)	SD (s)
LR	Spark	30	70%	2,432	201	2,149	198
	Deca	30	37.5%	218.3	10	2.92	0.3
		16	70%	295.4	14	98.12	6
	Off-heap Deca	30	37.5%	178	9.3	4.2	0.4
		16	70%	188	11	73.39	4
PR	Spark	20	23%	5,498	278	3,602	197
	Deca	20	9.1%	661	32	50.8	3
		8	23%	893.67	51.1	261	14
	Off-heap Deca	20	9.1%	658	38	33.21	1.98
		8	23%	702	39	115.69	6.2

Although GC tuning can reduce the cost of GC, the best advantage of Deca is the elimination of data objects. However, the memory pages are also accessed as the data objects in JVM. Thus, what would happen if the size of caching data objects reaches the same percent?

The above experiments show that Deca can reduce the memory consumption of the long-living cached data and hence the system has a lower memory pressure during execution. In other words, Deca has a lower GC overhead not only because it has fewer long-living objects but also because the system has less need to perform GC. In the next experiment, we investigate how the Deca optimizer performs under the memory pressure similar to the vanilla Spark. In the experiment, we set less memory space for Deca so that the long-living cached data consumes the same percentage of memory (70%) as Spark. The result is shown in Table 5. Even under the above memory settings, Deca still outperforms Spark significantly, although there is a slight increase of GC time and execution time when comparing with the case with the larger memory space. The increase in GC time is due to the fact that there is a higher demand for GC and Deca creates a number of long-living memory pages as objects on the JVM heap, which is subject to full (or major) GC from time to time.

7.5 Impact of VST Decomposition

Both LR and CC use the `groupByKey` operator. The VST objects produced in both these cases meet the requirements mentioned in Section 5.1, so these objects can be decomposed safely. We first examine the lifetime of the data objects and the GC overhead of the two major stages in PR. The



(a) Live data object numbers and GC time of PR/CC in Stage 2 (20GB executor memory).

(b) Live data object numbers and GC time of PR/CC in Stage 3 (20GB executor memory).

(c) Execution time of Stage2 in PR (10GB executor memory).

(d) Execution time of Stage2 in CC (10GB executor memory).

Fig. 15. Comparison of performance for PR and CC of Deca with and without VST decomposition.

results are shown in Figures 15(a) and 15(b). In stage 2, PR runs groupByKey and caches the output into memory. In this stage, the UDT data objects in the shuffle buffer are of a VST, because the value in scala.Tuple2 of groupByKey is of a VST. We decompose such objects using the VST decomposition approach and store the data off-heap. To put the objects into the cached block, we have to create some temporary objects. This is the reason that the number of objects in Deca fluctuates a bit, which incurs quite some GC overheads. On the other hand, in Spark, the number of cached data objects keeps increasing along with the execution, which incurs more GC overhead later in this stage. However, in general, the GC overhead in this stage is low when compared to the next stage.

In stage 3 and the subsequent ones, PR performs iterative computation over the cached data. In this stage, the UDT data objects in the shuffle buffer are of an SFST, and the value in the scala.Tuple2 of reduceByKey is of a primitive type. Deca can decompose the objects into memory pages and the memory pages can be shared between the cached blocks and the shuffle buffer. Therefore, Deca does not need to create temporary objects in this stage to convert data from shuffle buffers to cached blocks. On the other hand, the cache container in Spark contains a massive amount of long-living objects, which incurs a very high GC cost.

Because of Deca's effective optimization, even without VST decomposition, the cached data of PR and CC occupy a small memory space, so the remaining memory is sufficient for groupByKey with the memory configuration mentioned above. This is why effects of VST decomposition in

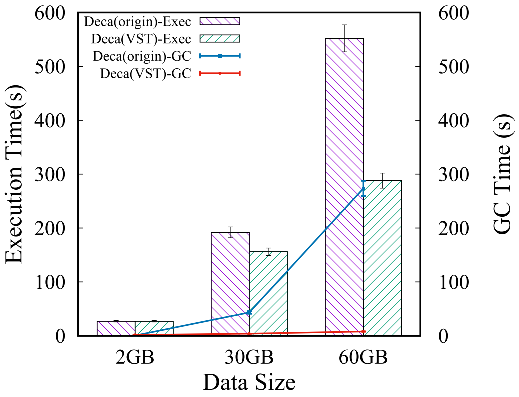


Fig. 16. Comparison of performance for *PScoreReducer* of Deca with and without VST decomposition.

stage 2 are not observable. To further examine the effect of VST decomposition, we adjust the heap size of each executor to 10GB for both caching and shuffling and examine the runtime of stage 2 with and without VST decomposition, respectively. As is shown in Figure 15(c) and Figure 15(d), VST decomposition can reduce the execution time by nearly 30% for both PR and CC with the 60GB dataset, mainly due to the reduced number of *long-living* objects in stage 2.

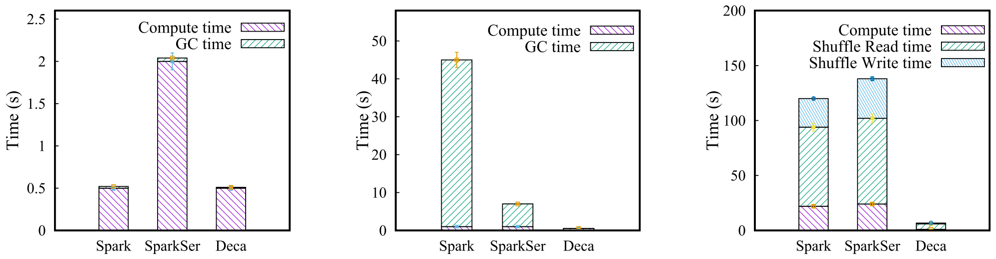
Although VST decomposition definitely reduces the execution time and GC time of stage 2, it does not have a significant impact on the whole job of PR and CC, mainly because the other parts dominate the total execution time. We further examine the effectiveness of VST decomposition using another real-world application, named *PScoreReducer* [20], where the stage involving VST objects is more significant in terms of execution time. Similar to PR and CC, this application has two stages and uses `groupByKey` to aggregate data. We conduct a similar experiment on this application to compare the cases with and without VST decomposition. We randomly generate three datasets (2GB, 30GB, 60GB) and set the maximum JVM heap size of each executor to be 10GB for data shuffling while guaranteeing no spilling of the shuffle buffer for all the datasets. Figure 16 shows the execution time and GC time of the entire job of PScoreReducer. VST-Deca can reduce the entire execution time by 47.8% and the GC time by 97.1% with the 60GB dataset, where there is limited memory space for data shuffling. This further confirms the effectiveness of VST decomposition.

7.6 Impact of Off-Heap Memory

The last experiment shows that Deca still incurs some GC overheads if we use the on-heap memory pages. As explained in Section 6, Deca can also use off-heap memory pages to avoid creating the long-living objects. In doing so, the GC overhead can be further reduced. In this section, we run the experiments using off-heap memory pages and present the results in Table 5. As can be observed in the table, when the memory consumption of cached data increases to the same amount as in Spark, Deca incurs much smaller GC overhead than using on-heap memory pages. This is because the long-living memory pages are allocated off-heap and therefore the system only needs to run minor GC to reclaim the space occupied by the temporary objects.

7.7 Microbenchmark

To make a closer comparison, we attempt to break down the running time of a single task in LR and PR in Figure 17. We use the optimized memory fractions obtained in the previous subsection. Note that tasks run concurrently in the system and we present the slowest tasks in the respective approaches, which somehow indicate the system bottleneck. In the LR-40G job, there is minimum



(a) LR-40G (b) LR-100G (c) PR-60G

Fig. 17. Breakdown of the task execution time for LR and PR of Spark, SparkSer, and Deca.

Table 6. Comparison of Performance for the Single-Node Microbenchmark of Spark, Deca, and SparkSer

App	JVM heap	Time	Spark (s)	SD (s)	Deca (s)	SD (s)	SparkSer (s)	SD (s)
LR	1.1GB	Exec.	162.8	8.9	9.2	0.5	81.8	5.3
		GC	138.8	6.8	0.1	0.02	6.0	0.4
	20GB	Exec.	10.9	0.6	9.5	0.3	44.6	2.1
		GC	0.03	0.01	0.16	0.08	0.035	0.01
PR	2GB	Exec.	353.2	19.3	27.8	1.52	806.8	45.3
		GC	104.6	6.3	0.16	0.02	503.2	26.9
	30GB	Exec.	231.8	12.9	23.5	1.51	263.8	14.3
		GC	12.8	0.67	0.04	0.01	2.1	0.15
Avg. time to serialize one object			-	-	3.7ms	-	3.9ms	-
Avg. time to deserialize one object			-	-	-	-	27ms	-

GC overhead for all approaches, but the deserialization overhead of SparkSer is obvious. For the LR-100G job, SparkSer can also minimize GC overhead, but it needs to deserialize data into temporary objects and hence still have some GC overhead. This shows the advantage of Deca's code modification over serialization. Furthermore, for PR-60G, there is high shuffling overhead in both Spark and SparkSer. This is because the disk swapping of the input cached RDD slows down the shuffle I/O. Due to the smaller footprint, Deca does not suffer from this problem. Note that GC and shuffle I/O can run in parallel and shuffling is the bottleneck in this setup.

To further analyze the CPU overhead, we run LR and PR within a controlled environment to eliminate the impact of memory footprint and other non-CPU factors, such as Spark's task scheduling delay and shuffling I/O. This is done by using a multithreaded Java program in a single machine to emulate the workflow of Spark without task scheduling and shuffling I/O.

In the LR job, we use 8 million randomly generated 10-dimensional labeled feature vectors as the input dataset. The input is first partitioned and cached in object arrays (Spark) or byte arrays (SparkSer and Deca). The cached partitions are then evenly dispatched to computing threads. The number of iterations is set to 50. The results in Table 6 show that, when the heap is large enough (20GB) and hence there is negligible GC overheads, Deca is almost identical to Spark, but SparkSer has a poor performance due to the high deserialization overhead. Furthermore, when the JVM heap size is relatively small (1.1GB), Spark suffers from high GC overheads, while both SparkSer and Deca can keep the GC overheads low. Again, Deca outperforms SparkSer because it does not require deserialization and has a lower GC overhead. We also measure the average time for Deca and Kryo to serialize and deserialize each object. The results are reported at the bottom of

Table 7. Execution Time of the Exploratory SQL Query

App	Exec. (s)	SD (s)	GC (s)	SD (s)	Cache (GB)
Spark	394	18	192.4	11.2	97.9 (23.1)
Spark SQL	180	5.1	3.0	0.2	47.1
Deca	192	6.2	4.2	0.2	55.6

The number in parenthesis is the amount of cached data swapped to disk.

Table 6. We can see that Deca has a similar serialization cost as Kryo, while Deca does not involve a significant deserialization overhead as Kryo does.

A similar experiment is done on PR, which uses both cache and shuffle buffers. The Pokec graph [41] with 1.6M vertices and 30M edges is used as the input. Note that Spark does not support in-memory serialization for shuffle buffers, so SparkSer only serializes the cached data. As shown in Table 6, when the GC overhead is negligible with a large heap, SparkSer again suffers from high deserialization overhead, while Deca runs significantly faster than Spark. This is because Spark needs to access auto-boxed objects in generic-type containers in the shuffle buffers, while Deca directly operates on the primitive values (note that this is not the case for LR, because LR does not involve shuffling). When the GC overhead is high with a smaller heap, SparkSer works worse than Spark because of the `join` operation in PR. During the `join` operation, SparkSer deserializes the cached data and stores the resulting objects in the shuffle buffers. On the other hand, Spark just stores references of the cached objects in the shuffle buffers. Thus, SparkSer suffers from even higher GC overheads than Spark does. Here, Deca's superiority is attributed to its ability to decompose not only cache blocks but also shuffle buffers.

In summary, Deca has lower GC overhead, a smaller footprint, no data deserialization, and no data boxing and unboxing. All these factors can be important to job execution time, and their significance depends on the actual scenarios.

7.8 Comparing with Spark SQL

In this experiment, we compare Deca with Spark SQL, which is optimized for SQL-like queries. Spark SQL uses a serialized column-oriented format to store in-memory tables. Moreover, with the project Tungsten, the shuffled data of certain (built-in or user-defined) aggregation functions, such as AVG and SUM, are also stored in memory with a serialized form. The serialization can be either autogenerated or manually written by users. In the experiment, we use the *uservisits* table from the Common Crawl document corpus [6], the size of which is 111GB. We use the table schemas and the following SQL queries provided in Spark's benchmark [6], which is a typical GroupBy aggregate query. We then write a semantically identical Spark program using RDDs. The input tables are entirely cached in memory before being queried. Spark serializes and compresses the in-memory data with a built-in mechanism.

```
SELECT SUBSTR(sourceIP, 1, 7), SUM(adRevenue)
  FROM uservisits
  GROUP BY SUBSTR(sourceIP, 1, 7);
```

To examine the case with a smaller dataset, where compression is not recommended due to its extra overhead, we randomly sampled 44GB of data from the complete dataset. The results of this experiment are shown in Table 7. Deca and Spark SQL significantly outperform Spark mainly thanks to their much lower GC overhead.

With a larger dataset, compression of cached data may be beneficial since it can avoid intensive disk I/O. Deca-field and Deca-object refer to the cases using field-oriented and object-oriented

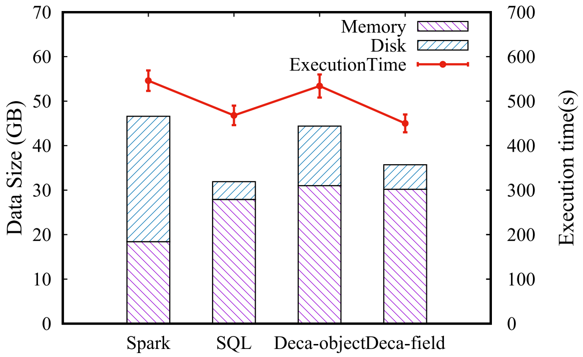


Fig. 18. Comparison of performance for the SQL query with in-memory compression of Spark, Spark SQL, and Deca with and without the field-oriented memory layout.

memory pages, respectively. The results of the experiment with data compression are shown in Figure 18. As the cost of disk I/O varies in different runs of the application, we run each application 7 times and draw the error bars in the figure. Also, we allocate as much memory space as possible and set a low fraction for caching, so that the shuffle and other operations cause few GCs, but cached data are spilled to disk instead. Thus, the size of the compressed data determines the cost of disk I/O and the job execution time. The result shows that Spark has a larger data size after compression and hence more data are swapped to disk. Another reason is that the estimation of runtime size of a large number of objects is inaccurate, which causes Spark to swap cached data to the disk unnecessarily early. Deca-object can reduce the size of swapped data due to more accurate estimation of the size of the byte sequences. However, it suffers from low compression performance. On the contrary, both Deca-field and Spark SQL can achieve a higher compression rate and accurate size estimation. Deca-field can achieve a similar performance as Spark SQL.

7.9 Cost and Applicability of Deca

In this experiment, we aim to explore the cost and applicability of the Deca optimizer by examining a large number of applications selected from *MLlib*, an open-source distributed machine-learning library based on Spark. The applications cover almost all kinds of machine-learning algorithms, such as classification, regression, decision tree, and rule mining. We report the optimization time of the first iteration because the subsequent iterations can reuse the optimized code (see Section 6.2.4). Table 8 and Table 9 show the time overhead of the three optimization phases, namely , preprocessing, analysis, and transforming, which includes decomposing, writing the classes, and job packaging. For all the applications, the overall time overhead of the Deca optimizer is often negligible compared to the execution time of the job submitted. We note that the time of points-to analysis in *LR* and *KMeans* is longer than that of other applications. This is because the points-to analysis involves the *breeze* package and causes more reachable methods to analyze. We also report the number of generated classes and methods to illustrate Deca's overhead from another perspective. The generated classes include the synthesized classes in the decomposition phase and linking phase (`Stage-Function-Class`).

As for the applicability of Deca, we also discover that, in several stages in the applications that use decision tree algorithms to terminate, such as *DecisionTreeRegression* and *RandomForestClassification*, the optimization terminates at the analysis phase. In those stages, the UDT objects in the data containers contain *Map* objects that belong to **Recursively Defined Type**; therefore, Deca cannot decompose these UDT objects or optimize the jobs at all.

Table 8. Comparison of Optimization Overheads for Different Applications (Set-a)

App	**Stage**	**Reachable Methods**	**Preprocess Time (ms)**	**SD (ms)**	**Analysis Time (ms)**	**SD (ms)**	**Transform Time (ms)**	**SD (ms)**	**Generate (Classes/Methods)**
WC	Job0-0	46	510	24	2,338	122	428	16	6 / 31
	Job0-1	21	133	4	1,426	58	48	3	1 / 3
PR	Job0-0	547	568	25	3,196	128	269	7	2 / 15
	Job0-1	956	243	6	3,468	175	209	8	3 / 29
	Job0-2	23	176	4	1,367	39	128	7	1 / 4
CC	Job0-0	547	548	22	3,162	124	240	6	2 / 15
	Job0-1	103	195	6	1,435	58	165	5	2 / 13
	Job0-2	126	166	8	2,635	106	161	6	2 / 15
	Job0-3	21	118	5	1,293	47	42	3	1 / 3
LR	Job0-0	5,164	668	31	24,836	1,005	767	31	7 / 116
KMeans	Job0-0	3,489	587	26	21,447	982	223	18	3 / 28
	Job1-0	1,894	183	8	8,350	340	536	31	7/ 129
	Job1-1	969	155	7	5,498	225	46	3	1 / 4
ChiSq-Selector	Job0-0	1,213	568	27	6,708	288	412	10	5 / 90
	Job1-0	1,228	208	7	6,108	208	231	9	5 / 93
	Job2-0(Iter)	1,239	218	6	4,535	201	201	6	5 / 93
	Job2-1(Iter)	210	117	6	1,643	40	43	3	1 / 3
	Job3-0	1,247	209	8	4,592	108	268	11	4 / 30
Elementwise-Product	Job0-0	853	588	26	3,288	137	229	12	2 / 9
	Job1-0	984	169	9	3,592	167	202	8	2 / 9
FPGrowth	Job(0 1)-0	901	370	13	3,473	118	244	11	3 / 51
	Job2-0	1,354	168	5	2,930	98	201	9	3 / 51
	Job2-1	240	133	4	1,622	67	53	2	1 / 3
	Job3-0(Iter)	1,241	186	7	3,967	162	322	11	6 / 148
	Job3-1(Iter)	640	163	7	3,998	129	148	6	1 / 3
Gaussian-Mixture	Job(0 2)-0	908	376	10	2,018	98	185	9	3 / 20
	Job3-0(Iter)	1,225	238	8	4,498	188	283	13	3 / 24
KDEExample	Job0-0	554	609	31	2,954	90	73	3	1 / 3
LBFGS	Job0-0	1,213	647	23	5,778	188	456	11	5 / 90
	Job1-0	1,211	218	6	4,535	179	456	18	5 / 93
	Job2-0(Iter)	1,228	246	9	4,487	159	554	22	12 / 152
	Job2-1(Iter)	692	158	6	2,581	101	46	2	1 / 5
	Job3-0	1,282	186	8	4,562	203	308	13	5 / 93
LinearRegressionWithSGD	Job(0 1)-0	908	419	12	3,489	147	337	14	4 / 29
	Job2-0(Iter)	1,080	203	9	4,956	225	309	14	9 / 141
	Job2-1(Iter)	470	176	8	1,757	63	53	2	1 / 4
	Job3-0	928	198	9	4,685	199	267	12	4 / 31
Normalizer	Job0-0	1,213	629	21	5,248	218	421	19	5 / 90
	Job1-0	1,227	206	11	4,487	200	243	11	4 / 30
Summary-Stat	Job0-0	1,053	799	37	4,852	240	438	23	8 / 174
	Job0-1	230	169	9	2,230	107	208	13	1 / 4

Table 9. Comparison of Optimization Overheads for Different Applications (Set-b)

App	Stage	Reachable Methods	Preprocess Time (ms)	SD (ms)	Analysis Time (ms)	SD (ms)	Transform Time (ms)	SD (ms)	Generate Classes/Methods
SVD	Job0-0	853	576	22	3,728	148	185	9	2 / 9
	Job1-0(Iter)	903	355	16	3,228	142	257	11	6 / 110
	Job1-1(Iter)	241	178	8	1,368	58	47	2	1 / 4
	Job2-0	855	189	8	2,688	109	145	6	2 / 49
SVMWith-SGD	Job0-0	1,213	648	28	2,587	98	396	18	5 / 84
	Job(1 3)-0	1,206	172	8	2,442	101	302	13	5 / 89
	Job4-0(Iter)	1,240	208	10	1,839	89	186	7	5 / 89
	Job4-1(Iter)	472	168	7	1,438	68	45	2	1 / 4
	Job5-0	1,223	173	8	4,537	218	326	15	5 / 89
StandardScaler	Job0-0	1,213	653	30	4,762	212	452	21	5 / 90
	Job(1 3)-0	1,281	246	11	3,698	169	698	29	14 / 301
	Job4-0(Iter)	230	132	5	1,473	55	203	10	4 / 14
	Job4-1(Iter)	472	168	8	1,757	83	47	2	1 / 92
	Job5-0	1,223	173	6	3,678	158	319	14	12 / 92
TallSkinnySVD	Job0-0	887	566	21	3,782	172	199	9	2 / 9
	Job1-0(Iter)	937	347	16	3,134	149	269	12	6 / 110
	Job1-1(Iter)	240	193	9	1,288	58	46	3	1 / 4
DecisionTreeRegression	Job0-0	1,213	732	32	5,388	242	419	20	5 / 90
	Job1-0	1,246	225	11	3,703	180	296	14	5 / 93
	Job2-0	1,245	194	7	4,363	205	245	13	5 / 93
	Job3-0	1,270	188	5	3,609	169	532	26	12 / 163
	Job3-1	1,074	162	7	3,358	149	209	9	1 / 3
	Job4-0(Iter)	1,327	205	11	4,589	223	/	/	/
	Job4-1(Iter)	2,156	235	11	5,728	290	/	/	/
	Job5-0	1,215	162	8	3,218	152	334	14	5 / 93
LogisitcRegressionWithLBFGS	Job0-0	1,213	538	22	4,748	233	468	22	5 / 90
	Job1-0	207	189	8	2,568	124	278	14	5 / 93
	Job2-0	1,206	161	8	4,308	188	223	11	5 / 93
	Job3-0	1,282	242	12	3,539	177	648	30	14 / 301
	Job3-1	230	138	6	1,490	68	209	9	1 / 4
	Job4-0	1,224	169	7	4,209	201	243	11	5 / 92
	Job5-0(Iter)	241	218	10	1,846	87	568	25	12 / 173
	Job5-1(Iter)	692	149	7	2,192	103	46	2	1 / 4
	Job6-0	1,282	167	8	4,833	237	332	15	5 / 93
RandomForestClassification	Job0-0	1,213	613	28	5,314	249	405	20	5 / 90
	Job1-0	1,246	208	10	4,592	220	318	14	5 / 93
	Job2-0	1,245	178	8	4,204	205	265	12	5 / 3
	Job3-0	1,270	179	9	4,478	212	492	24	12 / 163
	Job3-1	1,074	158	7	3,296	143	218	11	1 / 3
	Job4-0(Iter)	1,326	238	9	4,582	239	/	/	/
	Job4-1(Iter)	2,156	145	6	5,899	284	/	/	/
	Job5-0	1,361	183	8	3,287	138	379	18	5 / 93
	Job6-0	1,205	162	8	3,264	152	220	10	5 / 93

(Continued)

Table 9. Continued

App	Stage	Reachable Methods	Preprocess Time (ms)	SD (ms)	Analysis Time (ms)	SD (ms)	Transform Time (ms)	SD (ms)	Generate Classes/Methods
PCA	Job0-0	905	638	31	3,010	138	259	12	4 / 26
	Job1-0	1,891	248	11	6,509	280	568	27	9 / 151
	Job1-1	1,449	176	8	4,309	210	449	20	1 / 4
	Job2-0(Iter)	2,015	193	9	6,294	292	255	12	9 / 141
	Job2-1(Iter)	1,462	172	8	4,318	202	43	2	1 / 4
	Job3-0	2,140	208	9	6,532	303	304	15	9 / 145
	Job3-1	1,862	172	8	6,458	312	268	13	4 / 31
	Job4-0	2,101	196	10	6,732	299	276	13	4 / 40

The Deca optimizer is shown to be applicable to all the applications examined in this subsection. But if some user-defined RDDs with unknown semantics appear in an application and Deca cannot successfully complete *iterator fusion*, then the optimizer will lose applicability. This is because the current *iterator fusion* is based on the built-in RDD operators of Spark, and the Deca optimizer cannot extract the UDFs from those RDDs in the preprocessing phase. *MLlib* includes a small number of such applications. There is an approach to solve the problem, which we plan to implement in the future. Specifically, Spark processes data objects and computes the results after linking the specific iterators in a specific type of RDD. Deca can handle the *iterator fusion* based on these iterators in an RDD with fine granularity; therefore, we can complete the *iterator fusion* successfully without knowing the RDD types. With this extension, the applicability of Deca can be extended significantly.

8 RELATED WORK

The inefficiency of memory management in managed runtime platforms for big data processing systems has been widely acknowledged. Existing efforts to minimize the overhead of memory management can be categorized into the following directions.

8.1 In-Memory Serialization

Many distributed data processing systems, such as Hadoop, Spark, and Flink, support serializing in-memory data objects into byte arrays in order to reduce memory management overhead. ITask [15] adopts a novel programming abstraction, namely, interruptible tasks, for JVM-based big data processing frameworks. Framework developers can implement the task execution engines by using ITask APIs. The ITask library runtime monitors the heap usage of the task execution process. Whenever the memory usage exceeds the predefined threshold value, ITask suspends selected tasks and releases in-memory buffers for their input, output, and intermediate data. For large intermediate data that are costly to recompute, ITask serializes the data into memory buffers or disk files instead of discarding them. The suspended tasks can be resumed when memory usage decreases. However, object serialization has long been acknowledged to have a high overhead [11, 47, 32].

8.2 GC Optimization

Most traditional GC tuning techniques are proposed for long-running latency-sensitive web servers. Some open-source distributed NoSQL systems, such as Cassandra [12] and HBase [21], use these latency-centric methods to avoid long GC pauses by replacing the Parallel GC with, e.g., CMS or G1 and tuning their parameters.

Implementing better GC algorithms is another line of work. Taurus [29] is a holistic runtime system for distributed data processing that coordinates the GC executions on all workers to improve the overall job performance. Later NumaGiC [16] was implemented as a NUMA-aware garbage collector for data processing running on machines with a large memory space. A phenomenon was discovered that big data applications do not meet the weak generational hypothesis; NG2C [9] uses a dynamic N-Generational mechanism as an extension of the 2-Generation of HotSpot GC, and contains a lifetime profiling tool in order to pretenure objects into older generations. This design aims to avoid long GC pauses but cannot increase application throughputs. These approaches' requirements of modifying JVMs prevent them from being adopted in production environments. On the other hand, Deca employs a nonintrusive approach and requires no JVM modification.

8.3 Region-Based Memory Management

In region-based memory management (RBMM) [44], all objects are grouped into a set of hierarchical regions, which are the basic units for space reclamation. The reclamation of a region automatically triggers the recursive reclamation of its subregions. This approach requires the developers to explicitly define the mapping from objects to regions, as well as the hierarchical structures of regions. Broom [17] uses RBMM for distributed data processing that runs on .NET CLR as an early work. However, the evaluation is conducted using task emulation with manually implemented operators, while the details about how to transparently integrate RBMM with user codes remain unclear.

Bloat-aware design [10] is based on an observation of the properties of big data processing programs, most of which can be split into the control path and the data path. In the data path, each object represents a small fine-grained data item. The memory bloat problem is caused by the huge amount of data objects created at runtime, whose lifetimes show a clear staged pattern. In this method, users implement the RBMM in the application programs by following an accessor-pattern design paradigm for data objects. Homogeneous data objects with the same type and lifetime should be "fused" to a large object storing all their raw data. At runtime, the modified program uses a few accessors to get and set the data inside the large objects.

Similar designs have been adopted by some realistic open-source big data processing applications. In the *Alternating Least Squares* (ALS) matrix factorization application of Spark MLlib, data items of input files are represented by `Rating` objects, which can then be transformed into the matrix-tile form. ALS creates `RatingBlock` objects to store the matrix-tile data. Each `RatingBlock` object has only three primitive-type arrays, thereby significantly reducing the in-memory object number. Applications written for GraphX [19], a high-level graph-parallel framework built on top of Spark, create one `VertexPartition` object and one `EdgePartition` object for each task to store data of its assigned subgraph.

RBMM programming designs are flexible enough for different object lifetime patterns, but users have to directly control the physical memory layout of in-memory data objects. Considering that both the big data processing frameworks (such as Spark) and the user-defined programs can access the UDT data objects, application and framework codesigns are necessary to implement RBMM. For example, manually optimized programs like ALS often use low-level RDD operators such as `mapPartitions`, `zipPartitions`, and `partitionByKey`, which require deep knowledge about the Spark framework implementation. These partition-parallel operators also break the original data-parallel abstraction of RDD. Moreover, if, for example, a graph-computing application uses non-primitive UDTs as the types of vertex/edge attributes, both the application and GraphX per se have to be modified to implement RBMM for the vertex and edge data objects.

FACADE [35] implements an approach that transforms user programs automatically and stores all the alive objects of user-annotated types in a single region managed by a simplified version of RBMM, thereby bypassing the garbage collection. The occupied space of data objects will be reclaimed at once at a user-annotated reclamation point. This approach heavily depends on its assumption for data objects' lifetime. It requires that the computation process be periodical so that it can allocate memory space for data objects at the beginning of a computation phase and release the memory space allocated at once until the phase ends. For general frameworks, such as Spark, the data objects are at least divided into temporary objects, medium-living objects, and long-living objects; therefore, this approach is not applicable.

In order to support hierarchical computation phases, Yak [34] further combines RBMM and JVM memory management. In Yak, data objects are first created in the memory regions of the innermost computation phase. When the current phase ends, the alive objects will be promoted to the memory regions of the parent phase. This approach can divide the lifetime of long-living objects with fine granularity but still cannot handle the memory management of temporary objects. Meanwhile, Yak needs to trace the reference relationship among cross-regional objects, which would incur extra overhead.

Unlike FACADE and Yak, Deca is a framework-specific approach and is tightly coupled with the underlying big data processing framework. It requires modifications of the memory management module of the framework (such as ShuffleManager and CacheBlockManager in Spark) in order to work with the memory layout of Deca. In comparing to the aforementioned previous work, Deca has two strengths: (1) data objects of an optimized program can be divided into fine-grained lifetime types and therefore does not have the implicit lifetime limitations of FACADE and Yak, and (2) in comparing to ITask and Yak, Deca can almost completely eliminate GC overheads, especially when using off-heap memory pages.

8.4 Domain-Specific Systems

Some domain-specific data-parallel systems make use of its specific computation structure to realize more complex memory management. Since the early adoption of JVM in implementing SQL-based data-intensive systems [38], efforts have been devoted to making use of the well-defined semantics of SQL query operators to improve the memory management performance in managed runtime platforms. Spark SQL [4] transforms relational tables to serialized bytes in a main-memory columnar storage. Tungsten [45], a Spark subproject, enables the serialization of hash-based shuffle buffers for certain Spark SQL operators. Deca has a similar performance as Spark SQL for structured data processing but provides more flexible computation and data models, which eases the implementation of advanced iterative applications such as machine-learning and graph mining algorithms.

8.5 Extension from Our Previous Work

The preliminary results of this work have been published in VLDB 2016 [28]. In addition to those results, this article documents our further research for Deca in the following aspects.

First, there are three types of UDT objects: SFST, RFST, and VST. In the previous work, we only consider the decomposition of SFST and RFST objects, not VST objects. The decomposition of VST objects is more challenging. This is because the size of a VST object is variable and therefore the memory manager cannot safely allocate a fixed memory space for it. In this article, we propose a novel method to decompose the VST objects. This method allows Deca to decompose more UDT data objects. Hence, this extension improves the applicability and performance of Deca, compared with the earlier version. We conduct new experiments to verify the effectiveness of the new method.

Second, in the memory page design in the VLDB version, the UDT objects are stored in sequence in the contiguous memory region. We call this type of memory page the object-oriented memory page. A data processing system may need to cache a large number of intermediate objects when processing large-scale applications. Under this circumstance, it is desired to compress the objects cached in memory, which enables Deca to cache more objects in memory and consequently minimize the need for swapping data to secondary disks. In doing so, the performance of Deca can be further improved when processing large-scale data. Our further research shows that the design of the object-oriented memory page results in a poor compression rate. To address this issue, we design the field-oriented memory page in this article. We also conduct new experiments and demonstrate in this article that this new design of memory page is very effective in data compression and significantly improves the performance of Deca when processing large-scale datasets. The experiments we conducted include comparing Deca with Spark SQL, which compress the in-memory table in a column-oriented manner. The experiment results show that Deca manifests the similar performance as Spark SQL in terms of both execution time and compression rate.

Finally, in previous work, Deca stores the decomposed objects in the heap. We found that if we put the memory pages in the heap, it may cause frequent (but light) GCs. To avoid this overhead, Deca in this work stores the decomposed VST objects in the off-heap space. Moreover, we conduct the experiments in this article to investigate the impact of off-heap memory.

In addition to the above added technical improvement and related experimental evaluation, this article presents the implementation of Deca in detail, which is only lightly touched upon in our previous work.

9 DISCUSSION

Deca's design can be used in big data processing frameworks that have the following two properties:

Fine-grained data-parallel task execution. Every job can be split into one or more stages that are separated by synchronous data transferring. Each stage has a homogeneous set of parallel tasks that execute the same computation procedure. Task programs can be divided into several execution phases. One execution phase is a loop (can have nested loops) that sequentially reads data items from source data containers and writes the processed data items to the destination data containers. Most data objects can only cross phase boundaries through data containers. As an exception, tasks can have a small number of state objects in the global scope, while their GC overheads are negligible. Data containers are either cached in memory or persist on secondary storages. Note that the code sizes and complexities of task programs are not related to the task execution time. Stream processing systems like Storm can have long-running tasks processing unbounded data streams.

Framework-managed in-memory data containers with predetermined lifetimes. Based on the knowledge of container lifetimes, the container space can be reclaimed at a proper time point. RBMM is implemented through binding data objects with their containers by Deca's program analyzer. Frameworks still have to control the container layouts by themselves. If they use the memory layouts described in this article, the current Deca optimizer can be used to synthesize the data accessor classes.

Some open-source frameworks are present below, which meet the two properties and can be modified to benefit from Deca's design:

Extended Hadoop. The main work of the Hadoop task is iteratively reading data items from input splits, invoking the map/reduce function for each item, and writing the generated data items to output splits. The only type of in-memory data containers in Hadoop is the shuffle buffer. Currently Hadoop's shuffler only uses a sort-based grouping/combining mechanism. The output objects of map functions are first serialized and then append to the shuffle buffer. Whenever data of the shuffle buffer have to be written to the disk, data items are sorted by their keys in serialized form. After sorting, the original *Value* objects have to be deserialized for combining. The hash-based shuffling implementation not only is cheaper than a sort-based one but also enables more effective eager aggregation [26]. With Deca's approach, the data items in memory can be stored as byte arrays to minimize memory consumption.

While Hadoop does not cache long lifetime objects in memory, some big data processing frameworks implemented based on Hadoop adopt the in-memory data caching design. M3R [40] caches key-value sequences in memory and shares heap states between jobs. User programs can explicitly manage the cached data with the provided APIs. Twister [14] and PRIter [52] also cache data in memory for iterative applications. Similar to cached RDDs in Spark, the Deca optimizer can be extended to decompose the cached data objects to minimize GC overheads in these variants of Hadoop.

Storm. The Storm topology concept resembles the Spark job DAG, except that Storm topology instances are long running and different stages in one topology instance can execute simultaneously. Tasks process data items by invoking user-defined `execute` methods in subclasses of the `Bolt` class. On top of the `Bolt` interface, Storm also provides a high-level functional programming model, Trident, to facilitate the application development. Trident topologies split stream data records into batches according to predefined strategies. Users can use `aggregate()` and `persistentAggregate()` operators to perform GroupBy-Aggregation over each data batch. Trident stores the aggregated values in memory *Key-Value* buffers. Therefore, Deca can be extended to decompose the data objects stored in aggregation buffers to byte arrays. Eliminating GC overheads is even more important for latency-sensitive stream processing applications.

GraphX. GraphX provides a graph-parallel programming model that hides the underlying RDD model. The UDFs and UDTs of GraphX applications are only related to the attribute data of vertices and edges. The physical layout of the graph structure is managed by the GraphX framework. The subgraph structure of each task partition is stored in a few primitive-type arrays and therefore is not necessary to be optimized for GC overheads. To decompose the vertex/edge attribute objects, we can extend Deca to analyze and transform the graph-parallel UDFs and UDTs.

10 CONCLUSION

In this article, we identify that GC overhead in distributed data processing systems is unnecessarily high, especially with a large input dataset. By presenting Deca's techniques of analyzing the variability of object sizes and safely decomposing objects in different containers, we show that it is possible to develop a general and efficient lifetime-based memory manager for distributed data processing systems to largely eliminate the high GC overhead. The experiment results show that Deca can significantly reduce Spark's application running time for various cases without losing the generality of its programming framework. Finally, to take advantage of Deca's optimization, a user is recommended not to create a massive number of long-living objects of a VST or design the VST based on the encapsulation rules in object-oriented languages if necessary.

APPENDIX

A CODE TRANSFORMATION

We explain the implementation of Deca optimizer in Section 6 in detail, and now use *Logistic Regression* (LR) as an example to illustrate all the optimization techniques of Deca. Moreover, we simplify and present the code segments of LR after the decomposition or transformation to help readers comprehend the process more clearly. While the code produced by Deca are JVM bytecodes, we provide the equivalent three-address code to illustrate the logic of the transformed LR code. LR's transformation is basically performed in several sub-phases, which are described below.

Running the iterative LR program would submit massive repetitive jobs. Deca only optimize the first submitted job and reuse the optimized code for the subsequent jobs. The unique job that we need to analyze only contains one stage (`ResultStage`), which have a cached RDD, but no shuffle operation. Deca generates two loop bodies: one for producing data objects in the cache buffer (data container), and another for processing data and computing the final results. Deca then extracts all the UDFs from the stage into its loop bodies, mainly the UDFs in the `map` function (see Figure 1, line 13 and 21).

LR generates a `LabeledPoint` object for each text line in the input dataset, each contains a `DenseVector` object and a `double` value. It then caches the resulting `LabeledPoint` objects in the memory. Deca determines the specific type of the objects written into the cache buffer via points-to analysis, and classifies the target UDT, i.e. `LabeledPoint` in this case, as a RFST. Since `LabeledPoint` and its fields can be decomposed safely, Deca generates a synthesized class named `DenseVector$Double$Deca` for `DenseVector`. In `DenseVector$Double$Deca`, there are some static fields and methods to provide the relative offsets of all the fields as explained in Section 6.2.3, and all the field-accessing code in each method are replaced with the array-accessing code, such as the read and write operations over `stride`, a primitive field of `DenseVector`. Deca also transforms the `dot` method in the original `DenseVector` into the new method, as shown in lines 24-52 of Figure 19.

After all the UDTs in LR are decomposed successfully, Deca needs to link them to the `Stage-Function-Class`, which is synthesized in the preprocessing phase. The details of the transformation in `Stage-Function-Class` are explained in Section 6.2.4, which mainly modifies the places where `LabeledPoint`'s fields and methods are accessed and invoked. In Figure 20, we provide the three-address code of the `apply` method in the final `Stage-Function-Class`. All the UDTs that need to be decomposed are represented as synthetic classes, such as `LabeledPoint` and `DenseVector` (line 17 and line 19). In the first code branch, the user program processes the data objects that are read either directly from the cache blocks if they exist in the cache buffer (line 10–31) or from the data sources (line 32–39). In the latter case, the data will be cached in the buffer by invoking the `parsePoint` method. Figure 21 presents the detail about the `parsePoint` method and the process of creating `LabeledPoint`'s raw data. The `parsePoint` method calculates the start offset and the end offset of the current `LabeledPoint` object, and in a similar way calculates the offsets of the nested objects based on the start offset (such as line 27). Finally a new function object is created by this class, named `StageFunc(Stage-Function-Class)`, and then Deca injects the function into a new RDD as mentioned in Section 6.2.4.

```
public int get_stride() {
   DenseVector$Double$Deca r0;
   int $i0;
   long $l0, $l1;
   r0 := @this: DenseVector$Double$Deca;
   $l0 = r0.<DenseVector$Double$Deca: long start_address>;
   $l1 = $l0 + 4L;
   $i0 = staticinvoke <Unsafe: int getInt(Object,long)>(null, $l1);
   return $i0;
}

public void set_stride(int) {
   DenseVector$Double$Deca r0;
   int i0;
   long $l0, $l1;
   r0 := @this: DenseVector$Double$Deca;
   i0 := @parameter0: int;
   $l0 = r0.<DenseVector$Double$Deca: long start_address>;
   $l1 = $l0 + 4L;
   staticinvoke <Unsafe: void putInt(Object,long,int)>(null, $l1, i0);
   return;
}

public final double dot(DenseVector$Double$Deca) {
   DenseVector$Double$Deca r0, r1;
   DoubleArray$Deca r4;
   double $d2, d3, $d1, $d0;
   int i1, $i2, i5, i0, i4, $i3;
   r0 := @this: DenseVector$Double$Deca;
   r1 := @parameter0: DenseVector$Double$Deca;

   i0 = virtualinvoke r0.<DenseVector$Double$Deca: int get_length()>();
   i1 = virtualinvoke r1.<DenseVector$Double$Deca: int get_length()>();
   if i0 != i1 goto label3;
   r4 = virtualinvoke r0.<DenseVector$Double$Deca: DoubleArray$Deca
       get_data()>();
   i4 = virtualinvoke r0.<DenseVector$Double$Deca: int get_offset()>();
   d3 = 0.0;
   i5 = 0;
  label1:
   $i2 = virtualinvoke r0.<DenseVector$Double$Deca: int get_length()>();
   if i5 >= $i2 goto label2;
   $d0 = virtualinvoke r4.<DoubleArray$Deca: double get(int)>(i4);
   $d1 = virtualinvoke r1.<DenseVector$Double$Deca: double
       apply(int)>(i5);
   $d2 = $d0 * $d1;
   d3 = d3 + $d2;
  $i3 = virtualinvoke r0.<DenseVector$Double$Deca: int get_stride()>();
   i4 = i4 + $i3;
   i5 = i5 + 1;
   goto label1;
  label2:
   return d3;
}
```

Fig. 19. The dot method of DenseVector$Double$Deca.

```
public final class StageFunc {
   ...
   public DenseVectorDoubleDeca compute(DecaTaskContext, int,
       DataSourceIterator) {
      ...
      r0 := @this: StageFunc;
      r3 := @parameter2: DataSourceIterator;
      ...
      z0 = staticinvoke <DecaBlockManager: boolean
          isContainBlock(String)>(blockId);
      if z0 == 0 goto label2;
      r14 = staticinvoke <DecaBlockManager: DecaCacheIterator
          getIter(String)>(blockId);
     label1:
      z2 = virtualinvoke r14.<DecaCacheIterator: boolean hasNext()>();
      if z2 == 0 goto label5;
      r18 = virtualinvoke r14.<DecaCacheIterator: LabeledPoint$Deca
          next()>();
      d0 = (double) 1;
      d1 = (double) 1;
      d2 = virtualinvoke r18.<LabeledPoint$Deca: double get_label()>();
      d3 = neg d2;
      r22 = r0.<StageFunc: DenseVector$Double$Deca w>;
      r23 = virtualinvoke r18.<LabeledPoint$Deca: DenseVector$Double$Deca
          get_features()>();
      d4 = virtualinvoke r22.<DenseVector$Double$Deca: double
          dot(DenseVector$Double$Deca)>(r23);
      d5 = d3 * d4;
      d6 = staticinvoke <math: double exp(double)>(d5);
      d7 = d1 + d6;
      d8 = d0 / d7;
      d9 = (double) 1;
      d10 = d8 - d9;
      r31 = virtualinvoke r23.<DenseVector$Double$Deca:
          DenseVector$Double$Deca times(double)>(d10);
      r36 = virtualinvoke r31.<DenseVector$Double$Deca:
          DenseVector$Double$Deca times(double)>(d2);
      r37 = virtualinvoke r0.<StageFunc: DenseVector$Double$Deca
          reduce(DenseVector$Double$Deca,DenseVector$Double$Deca)>(r37,
          r36);
      goto label1;
     label2:
      staticinvoke <DecaBlockManager: void createBlock(String)>(blockId);
     label3:
      z3 = virtualinvoke r3.<DataSourceIterator: boolean hasNext()>();
      if z3 == 0 goto label3;
      r39 = virtualinvoke r3.<DataSourceIterator: String next()>();
      virtualinvoke r0.<StageFunc: LabeledPoint$Deca
          parsePoint(String)>(r39);
      goto label5
     label4:
      r44 = staticinvoke <DecaBlockManager: DecaCacheIterator
          getIter(String)>(blockId);
     label5:
      z4 = virtualinvoke r44.<DecaCacheIterator: boolean hasNext()>();
     ...  //the same code with label1
     label6:
      return r37;
  }
}
```

Fig. 20. The compute method of synthetic Stage-Function-Class.

```
public final class StageFunc {
   static final int D = 10;
   public LabeledPoint$Deca parsePoint(String) {
      ...
      r0 := @this: StageFunc;
      r1 := @parameter0: java.lang.String;

      $l0 = staticinvoke <DecaBlockManager: long getBlockOffset()>();
      $i0 = r0.<StageFunc: int D>;
      $l1 = (long) $i0;
      $l2 = $l1 * 8L;
      $l3 = $l2 + 24L;
      $l4 = $l0 + $l3;
      staticinvoke <DecaBlockManager: void setBlockOffset(long)>($l4);

      $r4 = new StringTokenizer;
      specialinvoke $r4.<StringTokenizer: void <init>(String,String)>(r1, "
          ");
      r2 = $r4;
      $r5 = new StringOps;
      $r6 = <scala.Predef$: scala.Predef$ MODULE$>;
      $r7 = virtualinvoke r2.<StringTokenizer: String nextToken()>();
      $r8 = virtualinvoke $r6.<scala.Predef$: String
          augmentString(String)>($r7);
      specialinvoke $r5.<StringOps: void <init>(String)>($r8);
      d0 = virtualinvoke $r5.<StringOps: double toDouble()>();

      r3 = new DoubleArray$Deca;
      $l5 = $l0 + 12L;
      specialinvoke r3.<DoubleArray$Deca: void <init>(long,int)>($l5, $i0);
      i2 = 0;
     label1:
      if i2 >= $i0 goto label2;
      $r11 = new StringOps;
      $r12 = <scala.Predef$: scala.Predef$ MODULE$>;
      $r13 = virtualinvoke r2.<StringTokenizer: String nextToken()>();
      $r14 = virtualinvoke $r12.<scala.Predef$: String
          augmentString(String)>($r13);
      specialinvoke $r11.<StringOps: void <init>(String)>($r14);
      $d1 = virtualinvoke $r11.<StringOps: double toDouble()>();
      virtualinvoke r3.<DoubleArray$Deca: void set(int,double)>(i2, $d1);
      i2 = i2 + 1;
      goto label1;
     label2:
      $r9 = new DenseVector$Double$Deca;
      specialinvoke $r9.<DenseVector$Double$Deca: void <init>(long)>($l0);
      $r10 = new LabeledPoint$Deca;
      specialinvoke $r10.<LabeledPoint$Deca: void <init>(long,double)>($l0,
          d0);
      return $r10;
   }
   ...
}
```

Fig. 21. The parsePoint method of synthetic Stage-Function-Class.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments on earlier versions of this article, and we thank Alibaba Computing Platform team members for their support and collaboration.

REFERENCES

[1] Bowen Alpern, Dick Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Mark Mergen, Ton Ngo, Janice Shepherd, and Stephen Smith. 1999. Implementing JalapeÑO in java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, 314–324.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, 265–278.

[3] Eric Anderson and Joseph Tucek. 2010. Efficiency matters! *Special Interest Group on Operating Systems (SIGOPS)* 44, 1 (March 2010), 40–45.

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM, New York, 1383–1394.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*. ACM, New York, 44–54.

[6] Benchmark. 2014. Big Data Benchmark. Retrieved from http://tinyurl.com/qg93r43.

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*. IEEE Computer Society, Washington, DC, 137–146.

[8] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*. ACM, New York, 595–602.

[9] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. 2017. NG2C: Pretenuring garbage collection with dynamic generations for HotSpot big data applications. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management (ISMM'17)*. ACM, New York, 2–13.

[10] Yingyi Bu, Vinayak Borkar, Guoqing Xu, and Michael J. Carey. 2013. A bloat-aware design for big data applications. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM'13)*. ACM, New York, 119–130.

[11] Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. 1999. Object serialization for marshalling data in a java interface to MPI. In *Proceedings of the ACM 1999 Conference on Java Grande (JAVA'99)*. ACM, New York, 66–71.

[12] Cassandra. 2010. Cassandra Garbage Collection Tuning. Retrieved from http://tinyurl.com/5u58mzc.

[13] Databricks. 2015. Tuning Java Garbage Collection for Spark Applications. Retrieved from http://tinyurl.com/pd8kkau.

[14] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. 2010. Twister: A runtime for iterative MapReduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*. ACM, New York, 810–818.

[15] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, 394–409.

[16] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A garbage collector for big data on big NUMA machines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, 661–673.

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

[18] GoGC. 2015. Go GC: Prioritizing Low Latency and Simplicity. Retrieved from https://blog.golang.org/go15gc.

[19] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, 599–613.

[20] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting code optimizations in data-parallel pipelines through PeriSCOPE. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, 121–133.

[21] HBaseGC. 2016. Tuning Java Garbage Collection for HBase. Retrieved from http://tinyurl.com/j5hsd3x.

[22] HiBench. 2016. HiBench Suite. Retrieved from http://tinyurl.com/cns79vt.

[23] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009 (SOSP'09)*. ACM, New York, 261–276.

[24] Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC.

[25] Ondřej Lhoták and Laurie Hendren. 2003. Scaling java points-to analysis using SPARK. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer-Verlag, Berlin, 153–169.

[26] Boduo Li, Edward Mazur, Yanlei Diao, Andrew McGregor, and Prashant Shenoy. 2011. A platform for scalable one-pass analytics using MapReduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. ACM, New York, 985–996.

[27] Henry Lieberman and Carl Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* 26, 6 (June 1983), 419–429.

[28] Lu Lu, Xuanhua Shi, Yongluan Zhou, Xiong Zhang, Hai Jin, Cheng Pei, Ligang He, and Yuanzhen Geng. 2016. Lifetime-based memory management for distributed data processing systems. *Proceedings of the VLDB Endowment (PVLDB)* 9, 12 (Aug. 2016), 936–947.

[29] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2016. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, New York, 457–471.

[30] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'15)*. ACM, New York, 43–52.

[31] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*. USENIX Association, Berkeley, CA, 14–14.

[32] Heather Miller, Philipp Haller, Eugene Burmako, and Martin Odersky. 2013. Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. ACM, New York, 183–202.

[33] Derek Gordon Murray, Michael Isard, and Yuan Yu. 2011. Steno: Automatic optimization of declarative queries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*. ACM, New York, 121–131.

[34] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A high-performance big-data-friendly garbage collector. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, 349–365.

[35] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. 2015. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. ACM, New York, 675–690.

[36] Russell Power and Jinyang Li. 2010. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, 293–306.

[37] Veselin Raychev, Madanlal Musuvathi, and Todd Mytkowicz. 2015. Parallelizing user-defined aggregations using symbolic execution. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*. ACM, New York, 153–167.

[38] Mehul A. Shah, Michael J. Franklin, Samuel Madden, and Joseph M. Hellerstein. 2001. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *SIGMOD Record* 30, 4 (Dec. 2001), 103–114.

[39] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the titans: MapReduce vs. spark for large scale data analytics. *The Proceedings of the VLDB Endowment (PVLDB)* 8, 13 (Sept. 2015), 2110–2121.

[40] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. 2012. M3R: Increased performance for in-memory hadoop jobs. *The Proceedings of the VLDB Endowment (PVLDB)* 5, 12 (Aug. 2012), 1736–1747.

[41] SNAP. 2016. SNAP Dataset Collection. Retrieved from https://snap.stanford.edu/data/index.html.

[42] Soot. 2016. Soot Framework. Retrieved from http://sable.github.io/soot/.

[43] SparkGC. 2016. Spark Garbage Collection Tuning. Retrieved from http://tinyurl.com/hzf3gqm.

[44] Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* 132, 2 (Feb. 1997), 109–176.

[45] Tungsten. 2015. Project Tungsten of Spark. Retrieved from http://tinyurl.com/mzw7hew.

[46] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. 2013. Apache hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*. ACM, New York, Article 5, 16 pages.

[47] Matt Welsh and David Culler. 2000. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency - Practice and Experience* 12, 7 (2000), 519–538.

[48] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, 265–278.

[49] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. 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)*. USENIX Association, Berkeley, CA, 2–2.

[50] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 29–42.

[51] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (July 2015), 1920–1948.

[52] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2011. PrIter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*. ACM, New York, Article 13, 14 pages.

[53] Chaojun Zhao, Chen Chen, Zhijian Chen, and Jianyi Meng. 2016. Value locality based storage compression memory architecture for ECG sensor node. *Science China Information Sciences* 59, 4 (2016), 1–11.

[54] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (May 1977), 337–343.

Received April 2017; revised September 2018; accepted November 2018