A Bloat-Aware Design for Big Data Applications

Yingyi Bu Vinayak Borkar Guoqing Xu Michael J. Carey

Department of Computer Science, University of California, Irvine

{yingyib,vborkar,guoqingx, mjcarey}@ics.uci.edu

Abstract

Over the past decade, the increasing demands on data-driven business intelligence have led to the proliferation of large-scale, data-intensive applications that often have huge amounts of data (often at terabyte or petabyte scale) to process. An object-oriented programming language such as Java is often the developer's choice for implementing such applications, primarily due to its quick development cycle and rich community resource. While the use of such languages makes programming easier, significant performance problems can often be seen — the combination of the inefficiencies inherent in a managed run-time system and the impact of the huge amount of data to be processed in the limited memory space often leads to memory bloat and performance degradation at a surprisingly early stage.

This paper proposes a bloat-aware design paradigm towards the development of efficient and scalable Big Data applications in object-oriented GC enabled languages. To motivate this work, we first perform a study on the impact of several typical memory bloat patterns. These patterns are summarized from the user complaints on the mailing lists of two widely-used open-source Big Data applications. Next, we discuss our design paradigm to eliminate bloat. Using examples and real-world experience, we demonstrate that programming under this paradigm does not incur significant programming burden. We have implemented a few common data processing tasks both using this design and using the conventional object-oriented design. Our experimental results show that this new design paradigm is extremely effective in improving performance — even for the moderate-size data sets processed, we have observed $2.5\times$+ performance gains, and the improvement grows substantially with the size of the data set.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management, optimization, runtime environment; H.4 [*Information Systems Applications*]: Miscellaneous

General Terms Languages, Design, Performance

Keywords Big Data Applications, Memory Bloat, Design

1. Introduction

Modern computing has entered the era of Big Data. The massive amounts of information available on the Internet enable computer scientists, physicists, economists, mathematicians, political scientists, bio-informaticists, sociologists, and many others to discover interesting properties about people, things, and their interactions. Analyzing information from Twitter, Google, Facebook, Wikipedia, or the Human Genome Project requires the development of scalable platforms that can quickly process massive-scale data. Such frameworks often utilize large numbers of machines in a cluster or in the cloud to process data in a parallel manner. Typical data-processing frameworks include data-flow and message passing runtime systems. A data-flow system (such as MapReduce [19], Hadoop [10], Hyracks [16], Spark [49], or Storm [40]) uses distributed file system to store data and computes results by pushing data through a processing pipeline, while a message passing system (such as Pregel [29] or Giraph [9]) often loads one partition of data per processing unit (machine, process, or thread) and sends/receives messages among different units to perform computations. High-level languages (such as Hive [11], Pig [34], FlumeJava [18], or AsterixDB [14]) are designed to describe data processing at a more abstract level.

An object-oriented programming language such as Java is often the developer's choice for implementing data-processing frameworks. In fact, the Java community has already been the home of many data-intensive computing infrastructures, such as Hadoop [10], Hyracks [16], Storm [40], and Giraph [9]. Spark [49] is written in Scala, but it relies on a Java Virtual Machine (JVM) to execute. Despite the many development benefits provided by Java, these applications commonly suffer from severe memory bloat—a situation where large amounts of memory are used to store information not strictly necessary for the execution—that can lead to significant performance degradation and reduced scalability.

Bloat in such applications stems primarily from a combination of the inefficient memory usage inherent in the run time of a managed language as well as the processing of huge volumes of data that can exacerbate the already-existing inefficiencies by orders of magnitude. As such, Big Data applications are much more vulnerable to runtime bloat than regular Java applications. As an interesting reference point, our experience shows that the latest (Indigo) release of the Eclipse framework with 16 large Java projects loaded can successfully run (without any noticeable lag) on a 2GB heap; however, a moderate-size application on Giraph [9] with 1GB input data can easily run out of memory on a 12 GB heap. Due to the increasing popularity of Big Data applications in modern computing, it is important to understand why these applications are so vulnerable, how they are affected by runtime bloat, and what changes should be made to the existing design and implementation principles in order to make them scalable.

In this paper, we describe a study of memory bloat using two real-world Big Data applications: Hive [11] and Giraph [9], where Hive is a large-scale data warehouse software (Apache top-level project, powering Facebook's data analytics) built on top of Hadoop and Giraph is an Apache open-source graph analytics framework initiated by Yahoo!. Our study shows that *freely creat-*

ISMM'13, June 20–21, 2013, Seattle, Washington, USA.

ing objects (as encouraged by object-orientation) is the root cause of the performance bottleneck that prevents these applications from scaling up to large data sets.

To gain a deep understanding of the bottleneck and how to effectively optimize it away, we break down the problem of excessive object creation into two different aspects, (1) what is the space overhead if all data items are represented by Java objects? and (2) given all these objects, what is the memory management (i.e., GC) costs in a typical Big Data application? These two questions are related, respectively, to the spatial impact and the temporal impact that object creation can have on performance and scalability.

On the one hand, each Java object has a fixed-size header space to store its type and the information necessary for garbage collection. What constitutes the space overhead is not just object headers; the other major component is from the pervasive use of object-oriented data structures that commonly have multiple layers of delegations. Such delegation patterns, while simplifying development tasks, can easily lead to wasteful memory space that stores *pointers* to form data structures, rather than *the actual data* needed for the forward execution. Based on a study reported in [32], the fraction of the actual data in an IBM application is only 13% of the total used space. This impact can be significantly magnified in a Big Data application that contains a huge number of (relatively small) data item objects. For such small objects, the space overhead cannot be easily amortized by the actual data content. The problem of inefficient memory usage becomes increasingly painful for highly-parallel data-processing systems because each thread consumes excessive memory resource, leading to increased I/O costs and reduced concurrency.

On the other hand, a typical tracing garbage collection (GC) algorithm periodically traverses the entire live object graph to identify and reclaim unreachable objects. For non-allocation-intensive applications, efficient garbage collection algorithms such as a generational GC can quickly mark reachable objects and reclaim memory from dead objects, causing negligible interruptions from the main execution threads. However, once the heap grows to be large (e.g., a few dozens of GBs) and most objects in the heap are live, a single GC call can become exceedingly longer. In addition, because the amount of used memory in a Big Data application is often close to the heap size, GC can be frequently triggered and would eventually become the major bottleneck that prevents the main threads from making satisfactory progress. We observe that in most Big Data applications, a huge number of objects (representing data to be processed in the same batch) often have the same lifetime, and hence it is highly unnecessary for the GC to traverse each individual object every time to determine whether or not it is reachable.

Switch back to an unmanaged language? Switching back to an unmanaged language such as C++ appears to be a reasonable choice. However, our experience with many Big Data applications (such as Hive, Pig, Jaql, Giraph, or Mahout) suggests that a Big Data application often exhibits clear distinction between a *control path* and a *data path*. The control path organizes tasks into the pipeline and performs optimizations while the data path represents and manipulates data. For example, in a typical Big Data application that runs on a shared-nothing cluster, there is often a driver at the client side that controls the data flow and there are multiple run-time data operators executing data processing algorithms on each machine. The execution of the driver in the control path does not touch any actual data. Only the execution of the data operators in the data path manipulates data items. While the data path creates most of the run-time objects, its development often takes a very small amount of coding effort, primarily because data processing algorithms (e.g., joining, grouping, sorting, etc.) can be easily shared and reused across applications.

One study we have performed on seven open-source Big Data applications shows that the data flow path takes an overage 36.8% of the lines of source code but creates more than 90% of the objects during execution. Details of this study can be found in Section 4.3. Following the conventional object-oriented design for the control path is often unharmful; *it is the data path that needs a non-conventional design and an extremely careful implementation*. As the control path takes the majority of the development work, it is unnecessary to force developers to switch to an unmanaged language for the whole application where they have to face the (old) problems of low productivity, less community resource, manual memory management, and error-prone implementations.

Although the inefficient use of memory in an object-oriented language is a known problem and has been studied before (e.g., in [32]), there does not exist any systematic analysis of its impact on Big Data applications. In this paper, we study this impact both analytically and empirically. We argue that the designers of a Big Data application should strictly follow the following principle: *the number of data objects in the system has to be bounded and cannot grow proportionally with the size of the data to be processed.* To achieve this, we propose a new design paradigm that advocates to merge small objects in the storage and access the merged objects using data processors. The idea is inspired from old memory management technique called *page-based record management*, which has been used widely to build database systems. We adopt the proposed design paradigm in our "build-from-scratch" general-purpose Big Data processing framework (called Hyracks) at the application (Java) level, without requiring the modification of the JVM or the Java compiler. We demonstrate, using both examples and experience, that writing programs using the proposed design paradigm does not create much burden for developers.

We have implemented several common data processing tasks by following both this new design paradigm and the conventional object-oriented design principle. Our experimental results demonstrate that the implementations following the new design can scale to much larger data sizes than those following the conventional design. We believe that this new design paradigm is valuable in guiding the future implementations of Big Data applications using managed object-oriented languages. The observations made in this paper strongly call for novel optimization techniques targeting Big Data applications. For example, optimizer designers can develop automated compiler and/or runtime system support (e.g., within a JVM) to remove the identified inefficiency patterns in order to promote the use of object-oriented languages in developing Big Data applications. Furthermore, future development of benchmark suites should consider the inclusion of such applications to measure JVM performance.

Contributions of this paper include:

- an analytical study on the memory usage of common Big Data processing tasks such as the graph link analysis and the relational join. We find that the excessive creation of objects to represent and process data items is the bottleneck that prevents Big Data applications from scaling up to large datasets (Section 2);
- a bloat-aware design paradigm for the development of highly-efficient Big Data applications; instead of building a new memory system to solve the memory issues from scratch, we propose two application-level optimizations, including (1) merging (inlining) a chunk of small data item objects with the same lifetime into few large objects (e.g., few byte arrays) and (2) manipulating data by directly accessing merged objects (i.e., at the binary level), in order to mitigate the observed memory bloat patterns (Section 3);
- a set of experimental results (Section 5) that demonstrate significant memory and time savings using the design. We report

our experience of programming for real-world data-processing tasks in the Hyracks platform (Section 4). We compare the performance of several Big Data applications with and without using the proposed design; the experimental results show that our optimizations are extremely effective (Section 5).

2. Memory Analysis of Big Data Applications

In this section, we study two popular data-intensive applications, Giraph [9] and Hive [11], to investigate the impact of creating of Java objects to represent and process data on performance and scalability. Our analysis drills down to two fundamental problems, one in space and one in time: (1) large space consumed by object headers and object references, leading to low packing factor of the memory, and (2) massive amounts of objects and references, leading to poor GC performance. We analyze these two problems using examples from Giraph and Hive, respectively.

2.1 Low Packing Factor

In the Java runtime, each object requires a header space for type and memory management purposes. An additional space is needed by an array to store its length. For instance, in the Oracle 64-bit HotSpot JVM, the header spaces for a regular object and for an array take 8 and 12 bytes, respectively. In a typical Big Data application, the heap often contains many small objects (such as `Integers` representing record IDs), in which the overhead incurred by headers cannot be easily amortized by the actual data content. Space inefficiencies are exacerbated by the pervasive utilization of object-oriented data structures. These data structures often use multiple-level of delegations to achieve their functionality, leading to large space storing *pointers* instead of the actual data. In order to measure the space inefficiencies introduced by the use of objects, we employ a metric called *packing factor*, which is defined as the maximal amount of actual data that be accommodated into a fixed amount of memory. While a similar analysis [32] has been conducted to understand the health of Java collections, our analysis is specific to Big Data applications where a huge amount of data flow through a fixed amount memory in a batch-by-batch manner.

To analyze the packing factor for the heap of a typical Big Data application, we use the PageRank algorithm [35] (i.e., an application built on top of Giraph [9]) as a running example. PageRank is a link analysis algorithm that assigns weights (ranks) to each vertex in a graph by iteratively computing the weight of each vertex based on the weight of its inbound neighbors. This algorithm is widely used to rank web pages in search engines.

We ran PageRank on different open-source Big Data computing systems, including Giraph [9], Spark [49], and Mahout [12], using a 6-rack, 180-machine research cluster. Each machine has 2 quad-core Intel Xeon E5420 processors and 16GB RAM. We used a 70GB web graph dataset that has a total of $1,413,511,393$ vertices. We found none of the three systems could successfully process this dataset. They all crashed with `java.lang.OutOfMemoryError`, even though the data partitioned for each machine (i.e., less than 500MB) should easily fit into its physical memory.

We found that many real-world developers experienced similar problems. For example, we saw a number of complaints on `OutOfMemoryError` from Giraph's user mailing list, and there were 13 bloat-related threads on Giraph's mailing list during the past 8 months[1]. In order to locate the bottleneck, we perform a quantitative analysis using PageRank. Giraph contains an example

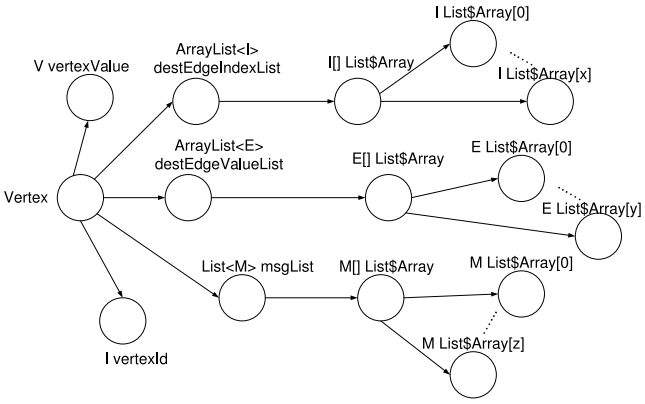


Figure 1. Giraph object subgraph rooted at a vertex.

Class	#Objects	Header (b)	Pointer (b)
Vertex	1	8	40
List	3	24	24
List$Array	3	36	$8(m+n)$
LongWritable	$m+1$	$8m+8$	0
DoubleWritable	$n+1$	$8n+8$	0
Total	$m+n+9$	$8(m+n)+84$	$8(m+n)+64$

Table 1. Numbers of objects per vertex and their space overhead (in bytes) in PageRank in the Sun 64-bit Hopspot JVM.

implementation of the PageRank algorithm. Part of its data representation implementation[2] is shown below.

```
public abstract class EdgeListVertex<
        I extends WritableComparable,
        V extends Writable,
        E extends Writable, M extends Writable>
        extends MutableVertex<I, V, E, M> {
    private I vertexId = null;

    private V vertexValue = null;

    /** indices of its outgoing edges */
    private List<I> destEdgeIndexList;

    /** values of its outgoing edges */
    private List<E> destEdgeValueList;

    /** incoming messages from
        the previous iteration */
    private List<M> msgList;
    ......

    /** return the edge indices starting from 0 */
    public List<I> getEdegeIndexes(){
                ...
    }
}
```

Graphs handled in Giraph are labeled (i.e., both their vertices and edges are annotated with values) and their edges are directional. Class `EdgeListVertex` represents a graph vertex. Among its fields, `vertexId` and `vertexValue` store the ID and the value of the vertex, respectively. Field `destEdgeIndexList` and `destEdgeValueList` reference, respectively, a list of IDs and a list of values of its outgoing edges. `msgList` contains incoming messages sent to the vertex from the previous iteration. Figure 1 visualizes the Java object subgraph rooted at an `EdgeListVertex` object.

In Giraph's PageRank implementation, the concrete types for I, V, E, and M are `LongWritable`, `DoubleWritable`,

[1] http://mail-archives.apache.org/mod_mbox/giraph-user/

[2] in revision 1232166.

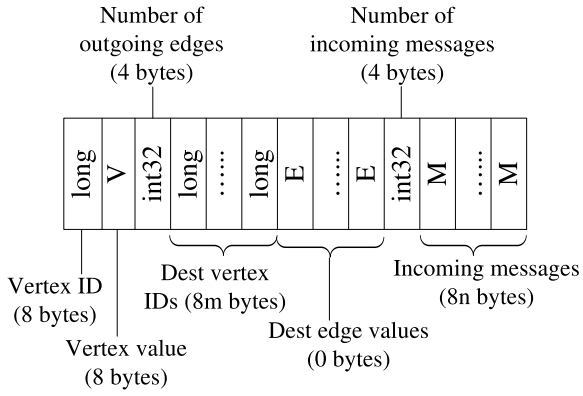


Figure 2. The compact layout of a vertex.

`FloatWritable`, and `DoubleWritable`, respectively. Each edge in the graph is equi-weighted, and thus the list referenced by `destEdgeValueList` is always empty. Assume that each vertex has an average of m outgoing edges and n incoming messages. Table 1 shows the memory consumption statistics of a vertex data structure in the Oracle 64-bit HotSpot JVM. Each row in the table reports a class name, the number of its objects needed in this representation, the number of bytes used by the headers of these objects, and the number of bytes used by the reference-typed fields in these objects. It is easy to calculate that the space overhead for each vertex in the current implementation is $16(m+n)+148$ (i.e., the sum of the header size and pointer size in Table 1).

On the contrary, Figure 2 shows an ideal memory layout that stores only the *necessary* information for each vertex (without using objects). In this case, the representation of a vertex requires $m+1$ long values (for vertex IDs), n double values (for messages), and two 32-bit int values (for specifying the number of outgoing edges and the number of messages, respectively), which consume a total of $8(m+n+1)+16 = 8(m+n)+24$ bytes of memory. This memory consumption is even less than half of the space used for object headers and pointers in the object-based representation. Clearly, the space overhead of the object-based representation is greater than 200%.

2.2 Large Volumes of Objects and References

In a JVM, the GC threads periodically iterate all live objects in the heap to identify and reclaim dead objects. Suppose the number of live objects is n and the total number of edges in the object graph is e, the asymptotic computational complexity of a tracing garbage collection algorithm is $O(n+e)$. For a typical Big Data application, its object graph consists of a great number of isolated object subgraphs, each of which represents either a data item or a data structure created for processing items. As such, there often exists an extremely large number of in-memory data objects, and both n and e can be orders of magnitude larger than those of a regular Java application.

We use an exception example from Hive's user mailing list to analyze the problem. This exception was found in a discussion thread named "how to deal with Java heap space errors"[3]:

```
FATAL org.apache.hadoop.mapred.TaskTracker:
  Error running child : java.lang.OutOfMemoryError:
  Java heap space
   org.apache.hadoop.io.Text.setCapacity(Text.java:240)
  at org.apache.hadoop.io.Text.set(Text.java:204)
  at org.apache.hadoop.io.Text.set(Text.java:194)
  at org.apache.hadoop.io.Text.<init>(Text.java:86)
  ......
  at org.apache.hadoop.hive.ql.exec.persistence.Row
    Container.next(RowContainer.java:263)
  org.apache.hadoop.hive.ql.exec.persistence.Row
    Container.next(RowContainer.java:74)
  at org.apache.hadoop.hive.ql.exec.CommonJoinOperator.
    checkAndGenObject(CommonJoinOperator.java:823)
  at org.apache.hadoop.hive.ql.exec.JoinOperator.
    endGroup(JoinOperator.java:263)
  at org.apache.hadoop.hive.ql.exec.ExecReducer.
    reduce(ExecReducer.java:198)
  ......
  at org.apache.hadoop.hive.ql.exec.persistence.Row
    Container.nextBlock(RowContainer.java:397)
  at org.apache.hadoop.mapred.Child.main(Child.java:170)
```

We inspected the source code of Hive and found that the top method `Text.setCapacity()` in the stack trace is not the cause of the problem. In Hive's join implementation, its `JoinOperator` holds all the `Row` objects from one of the input branches in the `RowContainer`. In cases where a large number of `Row` objects is stored in the `RowContainer`, a single GC run can become very expensive. For the reported stack trace, the total size of the `Row` objects exceeds the heap upper bound, which causes the `OutOfMemory` error.

Even in cases where no `OutOfMemory` error is triggered, the large number of `Row` objects can still cause severe performance degradation. Suppose the number of `Row` objects in the `RowContainer` is n. Hence, the GC time for traversing the internal structure of the `RowContainer` object is at least O(n). For Hive, n grows proportionally with the size of the input data, which can easily drive up the GC overhead substantially. The following is an example obtained from a user report at StackOverflow[4]. Although this problem has a different manifestation, the root cause is the same.

"I have a Hive query which is selecting about 30 columns and around 400,000 records and inserting them into another table. I have one join in my SQL clause, which is just an inner join. The query fails because of a Java GC overhead limit exceeded."

In fact, complaints about large GC overhead can be commonly seen on either Hive's mailing list or the StackOverflow website. What makes the problem even worse is that there is not much that can be done from the developer's side to optimize the application, because the inefficiencies are inherent in the design of Hive. All the data processing-related interfaces in Hive require the use of Java objects to represent data items. To manipulate data contained in `Row`, for example, we have to wrap it into a `Row` object, as designated by the interfaces. If we wish to completely solve this performance problem, we would have to re-design and re-implement all the related interfaces from scratch, a task that any user could not afford to do. This example motivates us to look for solutions at the design level, so that we would not be limited by the many (conventional) object-oriented guidelines and principles.

3. The Bloat-Aware Design Paradigm

The fundamental reason for the performance problems discussed in Section 2 is that the two Big Data applications were designed and implemented the same way as regular object-oriented applications: *everything is object*. Objects are used to represent both *data processors* and *data items* to be processed. While creating objects to represent data processors may not have significant impact on performance, the use of objects to represent data items creates a big scalability bottleneck that prevents the application from processing large data sets. Since a typical Big Data application does similar data processing tasks repeatedly, a group of related data items of-

[3] http://mail-archives.apache.org/mod_mbox/hive-user/201107.mbox/

[4] http://stackoverflow.com/questions/11387543/performance-tuning-a-hive-query.

ten has similar liveness behaviors. They can be easily managed together in large chunks of buffers, so that the GC does not have to traverse each individual object to test its reachability. For instance, all vertex objects in the Giraph example have the same lifetimes; so do `Row` objects in the Hive example. A natural idea is to allocate them in the same memory region, which is reclaimed as a whole if the contained data items are no longer needed.

Based on this observation, we propose a bloat-aware design paradigm for developing highly efficient Big Data applications. This paradigm includes the following two important components: (1) merging and organizing related small data record objects into few large objects (e.g., byte buffers) instead of representing them explicitly as one-object-per-record, and (2) manipulating data by directly accessing buffers (e.g., at the byte chunk level as opposed to the object level). The central goal of this design paradigm is to bound the number of objects in the application, instead of making it grow proportionally with the cardinality of the input data. It is important to note that these guidelines should be considered explicitly at the early design stage of a Big Data processing system, so that the resulting APIs and implementations would comply with the principles. We have built our own Big Data processing framework Hyracks from the scratch by strictly following this design paradigm. We will use Hyracks a running example to illustrate these design principles.

3.1 Data Storage Design: Merging Small Objects

As described in Section 2, storing data in Java objects adds much overhead in terms of both memory consumption as well as CPU cycles. As such, we propose to store a group of data items together in *Java memory pages*. Unlike a system-level memory page, which deals with virtual memory, a Java memory page is a fixed-length contiguous block of memory in the (managed) Java heap. For simplicity of presentation, we will use "page" to refer to "Java memory page" in the rest of the paper. In Hyracks, each page is represented by an object of type `java.nio.ByteBuffer`. Arranging records into pages can reduce the number of objects created in the system from the total number of data items to the number of pages. Hence, the packing factor in such a system can be much closer to that in the ideal representation where data are explicitly laid out in memory and no bookkeeping information needs to be stored. Note that grouping data items into a binary page is just one of many ways to merge small objects; other merging (inlining) approaches may also be considered in the future to achieve the same goal.

Multiple ways exist to put records into a page. The Hyracks system takes an approach called "slot-based record management" [36] used widely in the existing DBMS implementations. To illustrate, consider again the PageRank algorithm. Figure 3 shows how 4 vertices are stored in a page. It is easy to see that each vertex is stored in its compact layout (as in Figure 2) and we use 4 slots (each takes 4 bytes) at the end of the page to store the start offset of each vertex. These offsets will be used to quickly locate data items and support *variable-length* records. Note that the format of data records is invisible to developers, so that they could still focus on high-level data management tasks. Because pages are of fixed size, there is often a small *residual space* that is wasted and cannot be used to store any vertex. To understand the packing factor for this design, we assume each page holds p records on average, and the residual space has r bytes. The overhead for this representation of a vertex includes three parts: the offset slot (4 bytes), the amortized residual space (i.e., r/p), and the amortized overhead of the page object itself (i.e., `java.nio.ByteBuffer`). The page object has an 8-byte header space (in the Oracle 64-bit HotSpot JVM) and a reference (8 bytes) to an internal byte array whose header takes 12

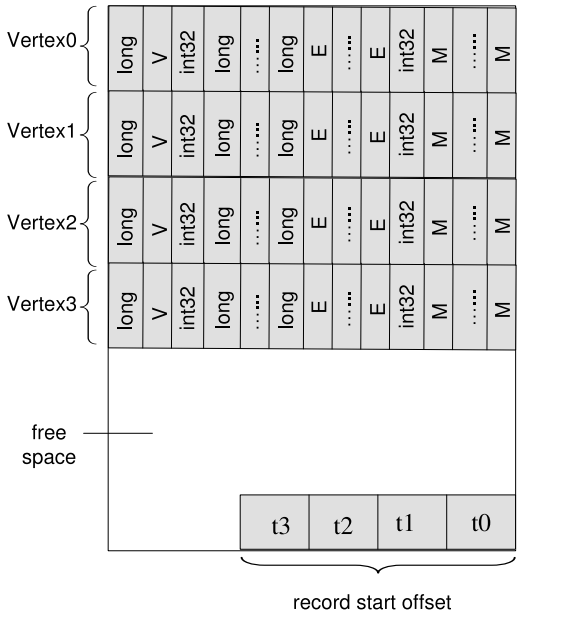


Figure 3. Vertices aligning in a page (slots at the end of the page are to support variable-sized vertices).

bytes. This makes the amortized overhead of the page object $28/p$. Combining this overhead with the result from Section 2.1, we need a total of $(8m + 8n) + 24 + 4 + \frac{r+28}{p}$ bytes to represent a vertex, where $(8m + 8n) + 24$ bytes are used to store the necessary data and $4 + \frac{r+28}{p}$ is the overhead. Because r is the size of the residual space, we have:

$$r \leq 8m + 8n + 24$$

and thus the space overhead of a vertex is bounded by $4 + \frac{8m+8n+52}{p}$. In Hyracks, we use 32KB (as recommended by literature [23]) as the size of a page, and p ranges from 100 to 200 (as seen in experiments with real-world data). To calculate the largest possible overhead, consider the worst case where the residual space has the same size as a vertex. The size of a vertex is thus between $(32768 - 200 * 4)/(200 + 1)$=159 bytes and $(32768 - 100 * 4)/(100 + 1)$=320 bytes. Because the residual space is as large as a vertex, we have $159 \leq r \leq 320$. This leaves the space overhead of a vertex in the range between 4 bytes (because at least 4 bytes are required for the offset slot) and $(4 + (320 + 28)/100)$=7 bytes. Hence, the overall overhead is only $2 - 4\%$ of the actual data size, much less than the 200% overhead of object-based representation (as described in Section 2.1).

3.2 Data Processor Design: Access Buffers

To support programming with the proposed buffer-based memory management, we propose an *accessor-based* programming pattern. Instead of creating a heap data structure to contain data items and represent their logical relationships, we propose to define an accessor structure that consists of multiple accessors, each to access a different type of data. As such, we need only a very number of accessor structures to process all data, leading to significantly reduced heap objects. In this subsection, we discuss a transformation that can transform regular data structure classes into their corresponding accessor classes. We will also describe the execution model using a few examples.

3.2.1 Design Transformations

For each data item class D in a regular object-oriented design, we transform D into an accessor class D_a. Whether a class is a data item class can be specified by the developer. The steps that this transformation includes are as follows.

- *S1*: for each data item field f of type F in D, we add a field f_a of type F_a into D_a, where F_a is the accessor class of F. For each non-data-item field f' in D, we copy it directly into D_a.
- *S2*: add a public method `set(byte[] data, int start, int length)` into D_a. The goal of this method is to bind the accessor to a specific byte region where the data items of type D are located. The method can be implemented either by doing *eager materialization*, which recursively binds the binary regions for all its member accessors, or by doing *lazy materialization*, which defers such bindings to the point where the member accessors are actually needed.
- *S3*: for each method M in D, we first create a method M_a which duplicates M in D_a. Next, M_a's signature is modified in a way so that all data-item-type parameters are changed to use their corresponding data accessor types. A parameter accessor provides a way for accessing and manipulating the bound data items in the provided byte regions.

Note that transforming a regular object-oriented design into the above design should be done at the early development stage of a Big Data application in order to avoid the potential development overhead of re-designing after the implementation. Future work will develop compiler support that can automatically transform designs to make them compatible with our memory system.

3.2.2 Execution Model

At run time, we form a set of *accessor graphs*, and each accessor graph processes a batch of top-level records. Each node in the graph is an accessor object for a field and each edge represents a "member field" relationship. An accessor graph has the same skeleton as its corresponding heap data structure but does not store any data internally. We let pages flow through the accessor graphs, where a accessor binds to and processes a data item record one-at-a-time. For each thread in the program, the number of accessor graphs needed is equal to the number of data structure types in the program, which is statically bounded. Different instances of a data structure can be processed by the same accessor graph.

If one uses eager materialization in the accessor implementation, the number of accessor objects that need to be created during a data processing task is equal to the total number of nodes in all the accessor graphs. If lazy materialization is chosen to implement accessors, the number of created accessor objects can be significantly reduced, because a member accessor can often be reused for accessing several different data times of the same type. In some cases, additional accessor objects are needed for methods that operate on multiple data items of the same type. For example, a `compare` method defined in a data item class compares two argument data items, and hence, in order to the transformed version of the method would need two accessor objects at run time to perform the comparison. Despite these different ways of implementing accessors, the number of accessor objects needed is always bounded at compile time and does not grow proportionally with the cardinality the dataset.

3.2.3 A Running Example

Following the three steps, we manually transform the vertex example in Section 2.1 into the following form.

```
public abstract class EdgeListVertexAccessor<
       /** by S1: */
```

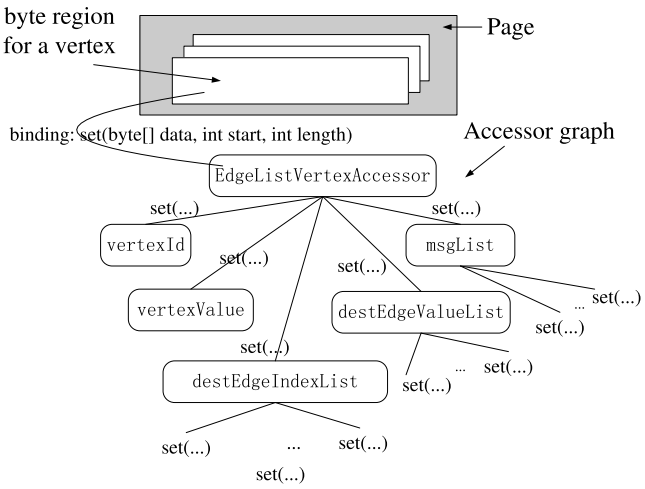


Figure 4. A heap snapshot of the example.

```
      I extends WritableComparableAccessor,
      V extends WritableAccessor,
      E extends WritableAccessor,
      M extends WritableAccessor>
       extends MutableVertexAccessor<I, V, E, M> {
    private I vertexId = null;

    private V vertexValue = null;

    /** by S1: indices of its outgoing edges */
    private ListAccessor<I> destEdgeIndexList = new
                               ArrayListAccessor<I>();

    /** by S1: values of its outgoing edges */
    private ListAccessor<E> destEdgeValueList = new
                                ArrayListAccessor<E>();

    /** by S1: incoming messages from
        the previous iteration */
    private ListAccessor<M> msgList = new
                                 ArrayListAccessor<M>();
    ......

    /** by S2:
      * binds the accessor to a binary region
      * of a vertex
      */
    public void set(byte[] data, int start, int length){
        /* This may in turn call the set method
           of its member objects. */
           ......
    }

    /** by S3:  replacing the return type*/
    public ListAccessor<I> getEdegeIndexes(){
        ...
    }
}
```

In the above code snippet, we highlight the modified code and add the transformation steps as the comments. Figure 4 shows a heap snapshot of the running example. The actual data are laid out in pages while an accessor graph is used to process all the vertices, each-at-a-time. For each vertex, the `set` method binds the accessor graph to its byte region.

4. Programming Experience

The bloat-aware design paradigm separates the *logical* data access and the *physical* data storage to achieve both compact in-memory data layout and efficient memory management. However, it appears that programming with binary data is a daunting task that creates

much burden for the developers. To help reduce the burden, we have developed a comprehensive data management library in Hyracks. In this section, we describe case studies and report our own experience with three real-world projects to show the development effort under this design.

4.1 Case Study 1: Quick Sort

In this case study, we compare the major parts of two implementations of quick sort, one in Hyracks manipulating binary data using our library, and the other manipulating objects using the standard Java library (e.g., `Collections.sort()`). The code shown below is the method `siftDown`, which is the core component of the quick sort algorithm. We first show its implementation in JDK.

```
private static void siftDown(Accessor acc,
                int start, int end) {
   for (int parent = start ; parent < end ; ) {
     int child1 = start + (parent - start) * 2 + 1;
     int child2 = child1 + 1;
     int child = (child2 > end) ? child1
                   : (acc.compare(child1, child2)
                < 0) ? child2 : child1;
     if (child > end)
        break;
     if (acc.compare(parent, child) < 0)
        acc.swap(parent, child);
     parent = child;
  }
}
```

The corresponding implementation using our library is a segment in the `sort` method, shown as follows:

```
private void sort(int[] tPointers, int offset, int length){
  ......
  while(true){
  ......
     while (c >= b) {
        int cmp = compare(tPointers, c, mi, mj, mv);
        if (cmp < 0) {
           break;
        }
        if (cmp == 0) {
          swap(tPointers, c, d--);
        }
        --c;
     }
     if (b > c)
        break;
     swap(tPointers, b++, c--);
  }
  ......
}
```

The `compare` method eventually calls a user-defined comparator implementation in which two accessors are bound to the two input byte regions once-at-a-time for the actual comparison. The following code snippet is an example comparator implementation.

```
public class StudentBinaryComparator implements
                          IBinaryComparator{
  private StudentAccessor acc1 =
            StudentAccessor.getAvailableAccessor();
  private StudentAccessor acc2 =
            StudentAccessor.getAvailableAccessor();

  public int compare(byte[] data1, int start1, int len1,
     byte[] data2, int start2, int len2){
     acc1.set(data1, start1, len1);
     acc2.set(data2, start2, len2);
     return acc1.compare(acc2);
  }
}
```

Method `getAvailableAccessor` can be implemented in different ways, depending on the materialization mode. For example, it can directly return a new `StudentAccessor` object or use an object pool to cache and reuse old objects. As one can tell, the data access code paths in both implementation are well encapsulated with libraries, which allows application developers to focus on the business logic (when to call `compare` and `swap`) rather than writing code to access data. As such, the two different implementations have comparable numbers of lines of code.

4.2 Case Study 2: Print Records

The second case study is to use a certain format to print data records from a byte array. A typical Java implementation is as follows.

```
private void printData(byte[] data) {
  Reader input = new BufferedReader(new InputStreamReader(
           new DataInputStream(new ByteArrayInputStream(
           data))));
  String line = null;
  while((line = input.readLine())!= null){
    String[] fields = line.split(',');
    //print the record
    ......
  }
  input.close();
 }
}
```

The above implementation reads `String` objects from the input, splits them into fields, and prints them out. In this implementation, the number of created `String` objects is proportional to the cardinality of records multiplied by the number of fields per record. The following code snippet is our printer implementation using the bloat-aware design paradigm.

```
private void printData(byte[] data) {
  PageAccessor pageAcc =
       PageAccessor.getAvailableAccessor(data);
  RecordPrintingAccessor recordAcc =
       RecordPrintingAccessor.getAvailableAccessor();
  while(pageAcc.nextRecord()){
    //print the record in the set call
    recordAcc.set(toBinaryRegion(pageAcc));
  }
}
```

In our implementation, we build one accessor graph for printing, let binary data flow through the accessor graph, and print every record by traversing the accessor graph and calling the `set` method on each accessor. It is easy to see that the number of created objects is bounded by the number of nodes of the accessor graph while the programming effort is not significantly increased.

4.3 Big Data Projects using the Bloat-Aware Design

The proposed design paradigm has already been used in six Java-based open-source Big Data processing systems, listed as follows:

- Hyracks [4] is a data parallel platform that runs data-intensive jobs on a cluster of shared-nothing machines. It executes jobs in the form of directed acyclic graphs that consist of user-defined data processing operators and data redistribution connectors.
- Algebricks [1] is a generalized, extensible, data model-agnostic, and language-independent algebra/optimization layer which is intended to facilitate the implementation of new high-level languages that process Big Data in parallel.
- AsterixDB [2] is a semi-structured parallel Big Data management system, which is built on-top-of Algebricks and Hyracks.
- VXQuery [6] is data-parallel XQuery processing engine on top of Algebricks and Hyracks as well.
- Pregelix [5] is Big Giraph analytics platform that supports the bulk-synchronous vertex-oriented programming model [29]. It internally uses Hyracks as the run-time execution engine.

- Hivesterix [3] is a SQL-like layer on top of Algebricks and Hyracks; it reuses the modules of the grammar and first-order run-time functions from HiveQL [11].

In these six projects, the proposed bloat-aware design paradigm is used in the data processing code paths (e.g., the runtime data processing operators such as join, group-by, and sort, as well as the user-defined runtime functions such as plus, minus, sum, and count) while the control code paths (e.g., the runtime control events, the query language parser, the algebra rewriting/optimization, and the job generation module) still use the traditional object-oriented design. Note that for Big Data applications, usually the control path is well isolated from the data path changes (e.g., data format changes) because they execute on different machines — the control path is on the master (controller) machines while the data path is on the slave machines.

We show a comparison of the lines-of-code (LOC) of the control paths and the data paths in those projects, as listed in Table 2. On average, the data path takes about 36.84% of the code base, which is much smaller than the size of the control path. Based on the feedback from the development teams, programming and debugging the data processing components consumes approximately $2\times$ as much time as doing that with the traditional object-orientated design. However, overall, since the data processing components take only 36.84% of the total development effort, the actual development overhead should be much smaller than $2\times$.

4.4 Future Work

The major limitation of the proposed technique is that it requires developers to manipulate low-level, binary representation of data, leading to increased difficulty in programming and debugging. We are in the process of adding another-level-of-indirection to address this issue— we are developing annotation and compiler support to allow for the declarative specifications of data structures. The resulting system will allow developers to annotate data item classes and provide relational specifications. The compiler will automatically generate code that uses the Hyracks library from the user specifications. We hope this system will enable developers to develop high-performance Big Data applications with a low human effort.

5. Performance Evaluation

This section presents a set of experiments focusing on performance comparisons between implementations with and without the proposed design paradigm. All experiments were conducted on a cluster of 10 IBM research machines. Each machine has a 4-core Intel Xeon 2.27 GHz processor, 12GB RAM, and 4 300GB 10,000 rpm SATA disk drives, and runs CentOS 5.5 and Java HotSpot(TM) 64-bit server VM (build17.0-b16, mixed node). JVM command line option "-Xmx10g" was used for all our experiments. We use a parallel generational GC in HotSpot, which combines parallel Scavenge (i.e., copying) for the young generation and parallel Mark-Sweep-Compact for the old generation. We collected the application running times and the JVM heap usage statistics in all the 10 machines. Their average is reported in this section. The rest of this section presents experimental studies of our design paradigm on the overall scalability (Section 5.1), the packing factors (Section 5.2), and the GC costs (Section 5.3).

5.1 Effectiveness On Overall Scalability: Using PageRank

The Pregelix [5] system mentioned in Section 4 supports nearly the same user interfaces and functionalities as Giraph, but uses the bloat-aware design paradigm. Internally, Pregelix employs a Hyracks runtime dataflow pipeline that contains several data processing operators (i.e., sort, group-by and join) and connectors (i.e., m-to-n hash partitioning merging connector) to execute graph processing jobs. Pregelix supports out-of-core computations, as it uses disk-based operators (e.g., external sort, external group-by, and index outer join) in the dataflow to deal with large amounts of data that cannot be accommodated in the main memory. In Pregelix, both the data storage storage design and data processor design are enabled: it stores all data in pages but employs accessors during data processing. The code shown in Section 3.2 is exactly what we used to process vertices.

In order to quantify the potential efficiency gains, we compared the performance of PageRank between Pregelix and Giraph. The goal of this experiment is not to understand the out-of-core performance, but rather to investigate the impact of memory bloat. Therefore, the PageRank algorithm was performed on a subset of Yahoo!'s publicly available AltaVista Web Page Hyperlink Connectivity Graph dataset [48], a snapshot of the World Wide Web from 2002. The subset consists of a total of $561,365,953$ vertices, each of which represents a web page. The size of the decompressed data is 27.8GB, which, if distributed appropriately, should easily fit into the memory of the 10 machines (e.g., 2.78GB for each machine). Each record in the dataset consists of a source vertex identifier and an array of destination vertex identifiers forming the links among different webpages in the graph.

In the experiment, we also ran PageRank with two smaller (1/100 and 1/10 of the original) datasets to obtain the scale-up trend. Figure 5 (a), (b), and (c) show, respectively, the overall execution times for the processing of the three datasets, their GC times, and their heap usages. It is easy to see that the total GC time for each dataset is less than 3% of the overall execution time. The JVM heap size is obtained by measuring the overall JVM memory consumption from the operating system. In general, it is slightly larger than the raw data size for each machine (i.e., 2.78GB). This is because extra memory is needed to (1) store object metadata (as we still have objects) and (2) sort and group messages. We also ran the Giraph PageRank implementation on the same three input datasets, but could not succeed for any of the datasets. Giraph crashed with `java.lang.OutOfMemoryError` in all the cases. We confirmed from the Giraph developers that the crash was because of the skewness (e.g., www.yahoo.com has a huge number of inbound links and messages) in the Yahoo! AltaVista Web Page Hyperlink Connectivity Graph dataset, combined with the object-based data representation.

5.2 Effectiveness On Packing Factor: Using External Sort

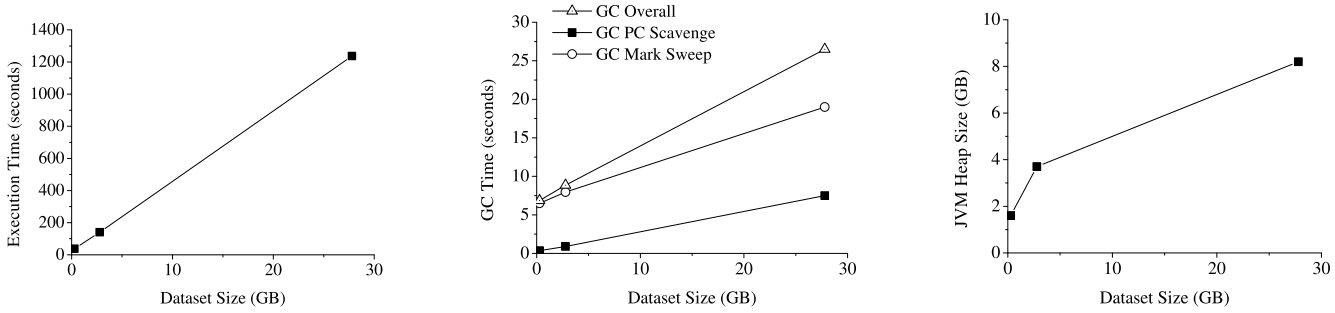
As the second experiment, we compared the performance of two implementations of the standard external sort algorithm [36] on Hyracks. One uses Java objects to represent data records, while the other employs Java memory pages. The goal here is to understand the impact of the low packing factor of the object-based data representation on performance as well as the benefit of the bloat-aware design paradigm.

In the Java object-based implementation, the operator takes deserialized records (i.e., in objects) as input and puts them into a list. Once the number of buffered records exceeds a user-defined threshold, method `sort` is invoked on the list and then the sorted list is dumped to a run file. The processing of the incoming data results in a number of run files. Next, the merge phase starts. This phase reads and merges run files using a priority queue to produce output records. During the merge phase, run files are read into the main memory one-block(32KB)-at-a-time. If the number of files is too large and one pass can not merge all of them, multiple merge passes need be performed.

In the page-based implementation, we never load records from pages into objects. Therefore, the page-based storage removes the header/pointer overhead and improves the packing factor. We im-

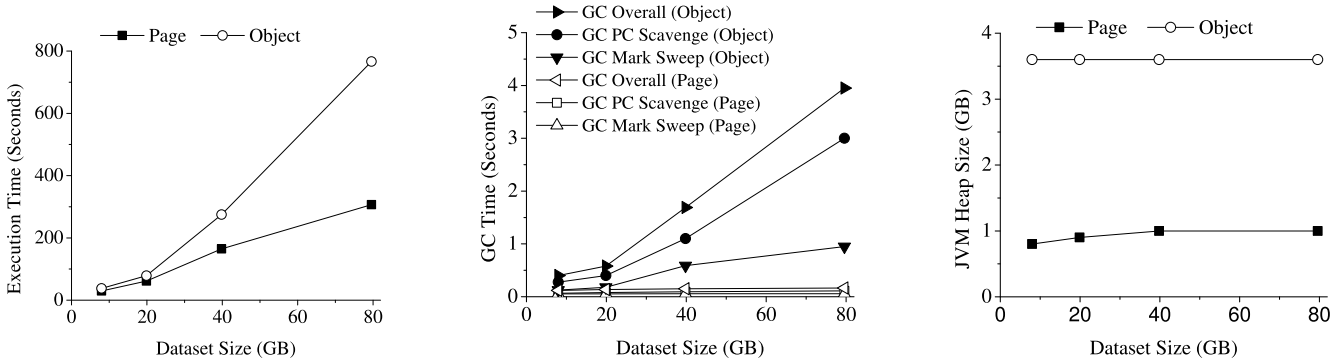
Project	Overall #LOC	Control #LOC	Data #LOC	Data #LOC Percentage
Hyracks	125930	71227	54803	43.52%
Algebricks	40116	36033	4083	10.17%
AsterixDB	140013	93071	46942	33.53%
VXQuery	45416	19224	26192	57.67%
Pregelix	18411	11958	6453	35.05%
Hivesterix	18503	13910	4593	33.01%
Overall	**388389**	**245323**	**143066**	**36.84%**

Table 2. The line-of-code statistics of Big Data projects which uses the bloat-aware design.



(a) Overall Performance (b) GC Time (c) JVM Heap Usage

Figure 5. PageRank Performance on Pregelix.



(a) Overall Performance (b) GC Time (c) JVM Heap Usage

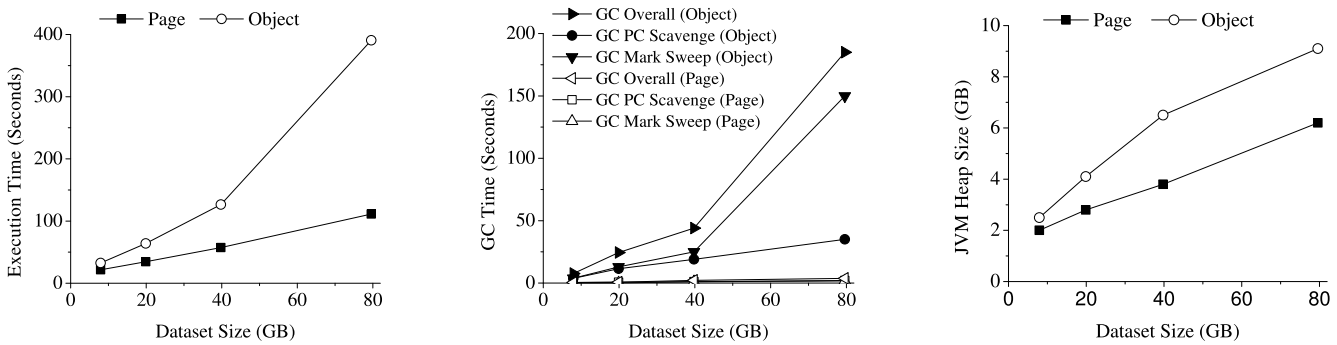
Figure 6. ExternalSort Performance.

plemented a quick sort algorithm to sort in-memory records at the binary level. In this experiment, we used the TPC-H[5] lineitem table as the input data, where each record represents an item in a transaction order. We generated TPC-H data at 10×, 25×, 50× and 100× scales, which correspond to 7.96GB, 19.9GB, 39.8GB and 79.6GB line-item tables, respectively. Each dataset was partitioned among the 10 nodes in a round-robin manner. Particularly, it was divided into 40 partitions, one partition per disk drive (recall that each machine has 4 disk drives).

The task was executed as follows: we created a total of 40 concurrent sorters across the cluster, each of which reads a partition of data locally, sorted them, and wrote the sorted data back to the disk. In this experiment, the page-based implementation used a 32MB sort buffer. The object-based implementation used 5000 the maximal number of in-memory records. The results of two different external sort implementations are plotted in Figure 6. The run-time statistics include the overall execution time (Figure 6 (a)), the GC overhead (Figure 6 (b)), and the JVM heap usage (Figure 6 (c)). Note that the Scavenge and Mark-Sweep lines show the GC costs for the young generation and the old generation, respectively. Because sorting a single memory buffer (either 32MB pages or 5000 records) is fast, most data objects are *short-lived and small*, and their lifetimes are usually limited within the iterations where they are created. They rarely get copied into old generations and thus the nursery scans dominate the GC effort.

From Figure 6 (a), it is clear to see that as the size of dataset increases, the page-based implementation scales much better (e.g., 2.5× faster on 79.6GB input) than the object-based implementation, and the improvement factor keeps increasing with the increase of the dataset size. The following two factors may contribute to this improvement: (1) due to the low packing factor, the object-

[5] The standard data warehousing benchmark, http://www.tpc.org/tpch/



(a) Overall Performance (b) GC Time (c) JVM Heap Usage

Figure 7. Hash-based Grouping Performance.

based implementation often has more merge passes (hence more I/O) than the page-based implementation, and (2) the use of objects to represent data items leads to increased JVM heap size and hence the operating system has less memory for the file system cache. For example, in the case where the size of the input data is 79.6GB, the page-based implementation has only one merge pass while the object-based one has two merge passes. The page-based implementation also has much less GC time than the object-based implementation (Figure 6 (b)). Figure 6 (c) shows that the amount of memory required by the object-based implementation is almost $4\times$ larger than that of the page-based implementation, even though the former holds much less actual data in memory (i.e., reflected by the need of an additional merge phase).

5.3 Effectiveness On GC Costs: Using Hash-Based Grouping

The goal of the experiment described in this subsection is to understand the GC overhead in the presence of large volumes of objects and references. The input data and scales for this experiment are the same as those in Section 5.2. For comparison, we implemented a logical SQL query by hand-coding the physical execution plan and runtime operators, in order to count how many items there are in each order:

```
SELECT l_orderkey, COUNT(*) AS items
FROM lineitem
GROUP BY l_orderkey;
```

We created two different implementations of the underlying hash-based grouping as Hyracks operators. They were exactly the same except that one of them used the object-based hash table (e.g., `java.util.Hashtable`) for grouping and the other used a page-based hash table implementation (e.g., all the intermediate states along with the grouping keys were stored in pages). The hash table size (number of buckets) was 10,485,767 for this experiment, because larger hash tables are often encouraged in Big Data applications to reduce collisions.

The Hyracks job of this task had four operators along the pipeline: a file scan operator (FS), a hash grouping operator (G1), another hash grouping operator (G2), and a file write operator (FW). In the job, FS was locally connected to G1, and then the output of G1 was hash partitioned and fed to G2 (using a m-to-n hash partitioning connector). Finally, G2 was locally connected to FW. Each operator had 40 clones across the cluster, one per partition. Note that G1 was used to group data locally and compress the data in order for it to be sent to the network. G2 performed the final grouping and aggregation.

Figure 7 shows the performance of the two different hash-based grouping implementations over the four input datasets. The measurements include the overall performance (Figure 7 (a)), the GC time (Figure 7 (b)), and the JVM heap size (Figure 7(c)). From Figure 7(a), one can see that the implementation with the page-based hash table scales much better to the size of data than the other one: in the case where the size of input data is 79.6GB, the former is already $3.5\times$ faster than the latter and the improvement factor keeps growing. In Figure 7 (b), we can clearly see that the use of the object-based hash table makes the GC costs go up to 47% of the overall execution time while the page-based hash table does not add much overhead ($<$3% of the overall execution time). A similar observation can be made on the memory consumption: Figure 7 (c) shows that the object-based implementation leads to much larger memory consumption than the page-based one.

Summary Our experimental results clearly demonstrate that, for many data-processing tasks, the bloat-aware design paradigm can lead to far better performance. The source code of all the experiments can be found at: http://code.google.com/p/hyracks.

6. Related Work

There exists a large body of work on efficient memory management techniques and data processing algorithms. This section discusses only those that are most closely related to the proposed work. These techniques are classified into three categories: data-processing infrastructures implemented in managed languages, software bloat analysis, and region-based memory management systems.

Java-based data-processing infrastructures Telegraph [37] is a data management system implemented in Java. It uses native byte arrays to store data and has its own memory manager for object allocation, deallocation, and memory de-fragmentation. However, because it is built on top of native memory, developers lose various benefits of a managed language and have to worry about low-level memory correctness issues such as dangling pointers, buffer overflow, and memory leaks. Our approach overcomes the problem by building applications on top of the Java managed memory, providing high efficiency and yet retaining most of the benefits provided by a managed runtime.

Hadoop [10] is a widely-used open-source MapReduce implementation. Although it provides a certain level of object reuse for sorting algorithms, its (major) map and reduce interfaces are object-based. The user has to create a great number of objects to implement a new algorithm on top of Hadoop. For example, the reduce-side join implementation in Hive [11] uses a Java `HashMap` in the reduce function to hold the inner branch records, which is very likely to suffer from the same performance problem (i.e., high GC costs) as discussed in Section 2.

Other data-processing infrastructures such as Spark [49] and Storm [40] also make heavy use of Java objects in both their core data-processing modules and their application programming interfaces, and thus they are vulnerable to memory bloat as reported in this paper.

Software bloat analysis Software bloat analysis [8, 30–33, 38, 41–47] attempts to find, remove, and prevent performance problems due to inefficiencies in the code execution and the use of memory. Prior work [32, 33] proposes metrics to provide performance assessment of use of data structures. Their observation that a large portion of the heap is not used to store data is also confirmed in our study. In addition to measure memory usage, our work proposes optimizations specifically targeting the problems we found and our experimental results show that these optimizations are very effective.

Work by Dufour *et al.* [20] uses a blended escape analysis to characterize and find excessive use of temporary data structures. By approximating object lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic areas. Shankar *et al.* propose Jolt [38], an approach that makes aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Work by Xu *et al.* [43] detects memory bloat by profiling copy activities, and their later work [42] looks for high-cost-low-benefit data structures to detect execution bloat. Our work is the first attempt to analyze bloat under the context of Big Data applications and perform effective optimizations to remove bloat.

Region-based memory management Region-based memory management was first used in the implementations of functional languages [7, 39] such as Standard ML [25], and then was extended to Prolog [28], C [21, 22, 24, 26], and real-time Java [13, 17, 27]. More recently, some mark-region hybrid methods such as Immix [15] combine tracing GC with regions to improve GC performance for Java. Our work uses a region-based approach to manage pages, but in a different context — the emerging Big Data applications. Data are stored in pages in the binary form leading to both increased packing factor and decreased memory management cost.

Value types Expanded types in Eiffel and value types in C# are used to declare data with simple structures. However, these types cannot solve the entire bloat problem. In these languages, objects still need to be used to represent data with complicated structures, such as hash maps or lists. The scalability issues that we found in Big Data applications are primarily due to inefficiencies inherent in the object-oriented system design, rather than problems with any specific implementation of a managed language.

7. Conclusion

This paper presents a bloat-aware design paradigm for Java-based Big Data applications. The study starts with a quantitative analysis of memory bloat using real-world examples, in order for us to understand the impact of excessive object creation on the memory consumption and GC costs. To alleviate this negative influence, we propose a bloat-aware design paradigm, including: merging small objects and accessing data at the binary level. We have performed an extensive set of experiments, and the experimental results have demonstrated that implementations following the design paradigm have much better performance and scalability than the applications that use regular Java objects. The design paradigm can be applied to other managed languages such as C# as well. We believe that the results of this work demonstrate the viability of implementing efficient Big Data applications in a managed, object-oriented language, and open up possibilities for the programming language and systems community to develop novel optimizations targeting data-intensive computing.

Acknowledgements

We thank anonymous reviewers for their thorough comments. Our Big Data projects using the bloat-aware design are supported by NSF IIS awards 0910989, 0910859, 0910820, and 0844574, a grant from the UC Discovery program, a matching donation from eBay, and generous industrial gifts from Google, HTC, Microsoft and Oracle Labs.

References

[1] Algebricks. https://code.google.com/p/hyracks/source/browse/#git%2Ffullstack%2Falgebricks.

[2] AsterixDB. https://code.google.com/p/asterixdb/wiki/AsterixAlphaRelease.

[3] Hivesterix. `http://hyracks.org/projects/hivesterix/`.

[4] Hyracks: A data parallel platform. `http://code.google.com/p/hyracks/`.

[5] Pregelix. `http://hyracks.org/projects/pregelix/`.

[6] VXQuery. `http://incubator.apache.org/vxquery/`.

[7] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, 1995.

[8] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753, 2010.

[9] Giraph: Open-source implementation of Pregel. `http://incubator.apache.org/giraph/`.

[10] Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`.

[11] The Hive Project. http://hive.apache.org/.

[12] The Mahout Project. http://mahout.apache.org/.

[13] W. S. Beebee and M. C. Rinard. An implementation of scoped memory for real-time java. In *International Conference on Embedded Software (EMSOFT)*, pages 289–305, 2001.

[14] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. ASTERIX: towards a scalable, semistructured data platform for evolving-world models. *Distrib. Parallel Databases*, 29:185–216, June 2011.

[15] S. M. Blackburn and K. S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–32, 2008.

[16] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *International Conference on Data Engineering (ICDE)*, pages 1151–1162, 2011.

[17] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 324–337, 2003.

[18] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[19] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[20] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[21] D. Gay and A. Aiken. Memory management with explicit regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 313–323, 1998.

[22] D. Gay and A. Aiken. Language support for regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 70–80, 2001.

[23] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[24] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 282–293, 2002.

[25] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 141–152, 2002.

[26] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in cyclone. In *International Symposium on Memory Management (ISMM)*, pages 73–84, 2004.

[27] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *International Conference on Architecture and Synthesis for Embedded Systems (CASES)*, pages 288–297, 2002.

[28] H. Makholm. A region-based memory manager for prolog. In *International Symposium on Memory Management (ISMM)*, pages 25–34, 2000.

[29] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[30] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–97, 2009.

[31] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[32] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[33] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[34] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[35] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[36] R. Ramakrishnan and J. Gehrke. *Database Management Systems (3. ed.).* McGraw-Hill, 2003.

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

[38] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[39] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lamda-calculus using a stack of regions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 188–201, 1994.

[40] Storm: dstributed and fault-tolerant realtime computation. https://github.com/nathanmarz/storm.

[41] G. Xu. Finding reusable data structures. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1017–1034, 2012.

[42] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[43] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[44] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER)*, pages 421–426, 2010.

[45] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[46] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[47] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 738–763, 2012.

[48] Yahoo! Webscope Program. http://webscope.sandbox.yahoo.com/.

[49] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. USENIX Workshop on Hot Topics in Cloud Computing, page 10, Berkeley, CA, USA, 2010.