Understanding and Combating Memory Bloat in Managed Data-Intensive Systems

KHANH NGUYEN, KAI WANG, YINGYI BU, LU FANG, and GUOQING XU,
University of California, Irvine

The past decade has witnessed increasing demands on data-driven business intelligence that led to the proliferation of data-intensive applications. A managed object-oriented programming language such as Java is often the developer's choice for implementing such applications, due to its quick development cycle and rich suite of libraries and frameworks. While the use of such languages makes programming easier, their automated memory management comes at a cost. When the managed runtime meets large volumes of input data, memory bloat is significantly magnified and becomes a scalability-prohibiting bottleneck.

This article first studies, analytically and empirically, the impact of bloat on the performance and scalability of large-scale, real-world data-intensive systems. To combat bloat, we design a novel compiler framework, called Facade, that can generate highly efficient data manipulation code by automatically transforming the *data path* of an existing data-intensive application. The key treatment is that in the generated code, the number of runtime heap objects created for data classes in each thread is (almost) *statically bounded*, leading to significantly reduced memory management cost and improved scalability. We have implemented Facade and used it to transform seven common applications on three real-world, already well-optimized data processing frameworks: GraphChi, Hyracks, and GPS. Our experimental results are very positive: the generated programs have (1) achieved a 3% to 48% execution time reduction and an up to 88× GC time reduction, (2) consumed up to 50% less memory, and (3) scaled to much larger datasets.

Categories and Subject Descriptors: D.2.4 [**Software Enginering**]: Metric—*Performance measures*; D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, memory management, optimization, run-time environments*

General Terms: Languages, Measurement, Performance

Additional Key Words and Phrases: Big data, managed languages, memory management, performance optimization

ACM Reference format:
Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, and Guoqing Xu. 2018. Understanding and Combating Memory Bloat in Managed Data-Intensive Systems. *ACM Trans. Softw. Eng. Methodol.* 26, 4, Article 12 (January 2018), 41 pages.
https://doi.org/10.1145/3162626

This article extends and refines work from two previous conference papers by the authors, published in the Proceedings of the International Symposium on Memory Management (ISMM'13) and International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15).
This work is supported by the National Science Foundation, under grant CNS-1321179, CCF-1409829, CNS-1613023, and CNS-1703598, and by the Office of Naval Research under grant N00014-14-1-0549 and N00014-16-1-2913.
Authors' addresses: K. Nguyen, K. Wang, Y. Bu, L. Fang, and G. Xu (corresponding author), Department of Computer Science, Bren Hall, University of California, Irvine, CA, 92697; emails: {khanhtn1, wangk7, yingyib, lfang3, guoqingx}@ics.uci.edu.

https://doi.org/10.1145/3162626

1 INTRODUCTION

Modern computing has entered the era of "Big Data." Developing systems that can *scale* to massive amounts of data is a key challenge faced by both researchers and practitioners. The mainstream approach to scalability is to enable distributed processing. As a result, existing platforms utilize large numbers of machines in clusters or in the cloud; data are partitioned among machines so that many processors can work simultaneously on a processing task. Typical parallel frameworks include, to name a few, FlumeJava (Chambers et al. 2010), Giraph (Apache 2014b), GPS (Salihoglu and Widom 2013), Hive (Thusoo et al. 2010), Hadoop (Apache 2014c), Hyracks (Borkar et al. 2011), Spark (Zaharia et al. 2010), Storm (Twitter 2014), Flink (Apache 2014a), and Pig (Olston et al. 2008b).

All of these data-intensive systems are written in managed languages such as Java, C#, or Scala, which are known for their simple usage, easy memory management, and abundant library suites and community support. While these languages simplify development effort, their managed runtime has a high cost—often referred to as *runtime bloat* (Mitchell et al. 2006; Mitchell and Sevitsky 2007; Xu et al. 2009, 2010a, 2010b; Xu and Rountev 2010; Xu et al. 2012; Xu 2012, 2013a; Nguyen and Xu 2013; Xu et al. 2014; Yan et al. 2012)—which cannot be amortized by increasing the number of data processing machines in a cluster. Poor performance on each node reduces the scalability of the entire cluster: a large number of machines are needed to process a small dataset, resulting in excessive use of resources and increased communication overhead. This article explores a new direction to scale data-intensive systems, that is, how to effectively optimize the managed runtime of a data processing system to improve its memory and execution efficiency on each node.

1.1 Motivation

Memory bloat in data-intensive applications stems primarily from a combination of inefficient memory usage inherent to the managed runtime as well as huge volumes of input data that need to be represented and processed in memory. As a result, data processing systems written in managed languages frequently face severe memory problems that cause them to fail. For example, in Spark (Zaharia et al. 2010, 2012), even machines with reasonably large memory resources cannot satisfy its need, and out-of-memory errors have been constantly reported on StackOverflow (StackOverflow 2015n) and the Apache mailing list (List 2014). Similarly, numerous examples of Hadoop users facing out-of-memory errors can also be found on StackOverflow (e.g., StackOverflow (2015h, 2015d, 2015k, 2015f, 2015i, 2015j, 2015m, 2015e, 2015c, 2015b), CMU (2015), and StackOverflow (2015a, 2015g, 2015l)). Finally, graph processing systems like Giraph (Apache 2014b) cannot process moderate-sized graphs on large clusters, although the amount of data for each machine is well below its memory capacity (Bu et al. 2013).

Memory inefficiency inherent to managed runtime is a real problem that has seriously concerned developers of data-intensive systems. Debates on whether systems development should go back and use C/C++ can be found almost everywhere (StackExchange 2015; Quora 2015; Cplusplus 2015). While forsaking managed languages and switching back to C/C++ appears to be a reasonable choice, unmanaged languages are more error prone; debugging memory bugs in an unmanaged language is known to be a notoriously painful task, which can be further exacerbated by the many "Big Data" effects, such as distributed execution environment, huge volumes of heap objects, and extremely long running time. Furthermore, since most existing data processing frameworks were already developed in a managed language (e.g., Java, C#, or Scala), it is unrealistic to reimplement them from scratch. Mozilla developed the Rust language (Mozilla 2014) for systems programming that eliminates garbage collection and enforces ownership propagation through new syntax and type system to prevent data races. However, developers are often resistant to new languages and it is unclear how much extra development is needed to write a Rust program compared to a similar Java program.

Due to the increasing popularity of object-oriented data-intensive 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 runtime system design to make applications more scalable.

1.2 Contribution 1: A Real-World Case Study

We first describe a study of memory bloat using two real-world data processing systems: Hive (Apache 2014d) and Giraph (Apache 2014b), 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 creating objects (as encouraged by object orientation), regardless of their different behaviors during execution, is the root cause of the performance bottleneck that prevents these applications from scaling up to large datasets.*

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 data items are represented by Java objects? and (2) given these Java objects, what is the memory management (i.e., GC) cost in a typical data-intensive application? These two questions are related to the spatial and the temporal impact of object creation on performance and scalability, respectively.

On the spatial side, 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 Mitchell and Sevitsky (2007), 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 data-intensive application that contains a huge number of small-sized data item objects. Since the data contents (e.g., a key and a value) in each such object do not take up much space, the space overhead incurred by headers and pointers cannot be easily amortized. This problem becomes increasingly painful when the amount of input data is too large to fit into memory: data processing must be divided into multiple rounds of loading, computation, and storing; with such a large space overhead, each round can only load and process a very small amount of data, resulting in significantly increased I/O costs.

On the temporal side, a typical tracing garbage collector (GC) periodically traverses the live object graph to identify and reclaim unreachable objects. For non-allocation-intensive applications, efficient algorithms such as a generational GC can quickly mark reachable objects and reclaim memory from dead objects, causing only negligible interruptions from the main execution threads. However, once the heap grows (e.g., dozens or even hundreds of GBs) and most objects in the heap are live, a single GC run can become exceedingly long. In addition, because the amount of used memory in a data-intensive application is often close to the heap size, the GC can be frequently triggered and become the major bottleneck that prevents the main threads from making satisfactory progress (e.g., the GC time accounts for up to 50% of the overall execution time). Details of the study will be discussed in Section 2.

1.3 Contribution 2: The Facade System

The key observation made in the study is that to develop a scalable system, the number of *data objects and their references in the heap* must *not* grow proportionally with the cardinality of the

dataset. To achieve this goal, we develop FACADE, a compiler and runtime system, that aims to *statically* bound the number of heap objects during the execution of a data-intensive application, thereby significantly reducing the space and temporal overhead incurred by the object-based data representation. There are three major challenges in FACADE's design. We briefly discuss these challenges and how FACADE overcomes them.

Challenge 1: How to Bound the Size of the Managed Heap. There exists a body of work that attempts to reduce the number of objects in a Java execution by employing different levels of techniques, ranging from programming guidelines (Gamma et al. 1995) through static program analyses (Choi et al. 1999; Blanchet 1999; Dolby and Chien 2000; Shuf et al. 2002; Lhotak and Hendren 2005) to low-level systems support (Xu 2013b). Despite the commendable efforts of these techniques, none of them are practical enough for modern data processing frameworks: sophisticated interprocedural static analyses (such as escape analysis (Choi et al. 1999) and object inlining (Dolby and Chien 2000)) cannot scale to framework codebases that have millions of lines of codes, while runtime-system-based techniques (such as Resurrector (Xu 2013b)) cannot scale to large heaps with billions of objects. In addition, none of the existing techniques can provide any static bound on the number of heap objects created at runtime.

To solve these practical issues, FACADE provides a *nonintrusive* solution, which aims to reduce the cost of the managed runtime by limiting the number of heap objects and references at *the compiler level* without needing to modify a JVM. FACADE advocates to separate data storage from data manipulation: data are stored in the off-heap memory (i.e., not recognized as heap objects), while heap objects are created as *facades* only for control purposes such as function calls (i.e., bounded). As the program executes, a *many-to-one mapping* is maintained between arbitrarily many data items in the native memory and a statically bounded set of facade objects in the heap. In other words, each facade keeps getting reused to represent data items.

To enforce this model, our FACADE compiler transforms an existing data processing program into an (almost) object-bounded program: the number of heap objects created for a data class in one thread is bounded by certain source code properties (i.e., a compile-time constant). More formally, FACADE reduces the number of data objects from $O(s)$ to $O(t * n * m + p)$, where s represents the cardinality of the dataset, t is the number of threads, n is the number of data classes, m is the maximum number of facades in a data class's pool, and p is the number of page objects used to store data. Details of these bounds can be found in Section 4.5.

In practice, the reduction is often in the scale of several orders of magnitude. As an example, for GraphChi (Kyrola et al. 2012), a single-machine graph processing system, FACADE has reduced the number of objects created to represent and process vertices and edges from 14, 257, 280, 923 to 1, 363 when running PageRank over the *twitter-2010* graph. Although t and p cannot be bounded statically, they are usually very small, and hence the total number of objects is "almost" statically bounded. Since data items are no longer represented by heap objects, the space overhead due to headers and pointers is significantly reduced. Furthermore, reductions on memory management costs can also be expected because the GC only scans the managed heap, which contains a very small number of control objects and facades. In the same PageRank example as discussed above, these reductions have led to 27% savings on execution time, 28% savings on memory consumption, and 84% savings on GC time.

Challenge 2: What to Transform. Our experience with various frameworks shows that real-world data-intensive systems have very large codebases and relied heavily on (third-party) libraries. Add in the fact that reflection and dynamic class loading are prevalently used to instantiate types in third-party libraries and frameworks that cannot be resolved statically, and there is little hope that a whole-program analysis/transformation can be done for any real system. No real-world programs

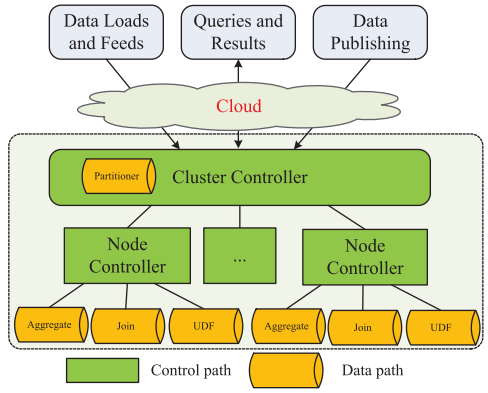

Fig. 1. Graphical illustration of control and data paths.

Table 1. The LoC Statistics of Control and Data Paths in the "Big Data" Projects under the AsterixDB Ecosystem

Project	Overall #LOC	Control #LOC	Data #LOC	Data #LOC Percentage
Hyracks (UCI 2014)	125,930	71,227	54,803	43.52%
Algebricks (UCI 2015a)	40,116	36,033	4,083	10.17%
AsterixDB (UCI 2015b)	140,013	93,071	46,942	33.53%
VXQuery (UCI 2015e)	45,416	19,224	26,192	57.67%
Pregelix (UCI 2015d)	18,411	11,958	6,453	35.05%
Hivesterix (UCI 2015c)	18,503	13,910	4,593	33.01%
Overall	**388,389**	**245,323**	**143,066**	**36.84%**

can be guaranteed to be safely transformed without using prohibitively expensive, sophisticated program analyses.

A key observation from our experience with dozens of real-world systems is that there often exists a clear boundary between a *control path* and a *data path* in a data processing system. As shown in Figure 1, the control path organizes tasks into pipelines, performs optimizations, and interacts with users, while the data path represents and manipulates data by invoking built-in operations such as `Aggregate` and `Join` or user-defined functions such as `Map` and `Reduce`. Although the data path creates most of the runtime objects to represent and process data items, its implementation is rather simple and its code size is often small. This property enables a systematic solution for data-intensive programs.

To better understand control/data paths in real-world programs, we have studied a set of six data-intensive systems built on top of the Hyracks data-parallel system (UCI 2014) and counted the numbers of lines of Java code for the control and data path in each system. These numbers are shown in Table 1. On average, data paths take about 35% in terms of lines of code, which is much smaller than the size of control paths. In a typical data-intensive application, it is often harmless to create objects in the control path, because the number of such objects is very small and independent of the input size. Our ultimate goal is, thus, to significantly reduce the *object representations of data items in the data path* so that they are not subject to the Java memory management. Since a data path often contains simple data manipulation functions, developing a compiler to transform these functions is much more feasible than transforming the entire application.

FACADE requires developers to provide a list of Java classes that form the data path. Each class in this list is transformed by our compiler into a *facade-based* class. Many performance problems result from the extensive use of large collections. For each collection class C (e.g., HashMap or ArrayList) in the standard Java library, we also transform it into a facade-based class C'. The original class C is used in normal ways in the control path, while type C' will be used in the data path to substitute C. Details of our treatment of arrays and collection classes in the Java library can be found in Section 4.10.

Challenge 3: How to Reclaim Data Objects. As data objects are no longer subject to garbage collection, an important question is how and when to reclaim them from native memory. A great deal of evidence shows that a data path is *iteration based* (Bu et al. 2013; Nguyen et al. 2015; Fang et al. 2015). In this article, "iteration" refers to a piece of data processing code that is repeatedly executed. Its definition includes but is more general than that of "computational iteration" performed in graph algorithms. For example, an iteration can also be a MapReduce task or a dataflow operator in data-parallel frameworks. Iterations are very well defined in data processing frameworks and can be easily identified by even novices. For example, in GraphChi (Kyrola et al. 2012), a computational iteration that loads shards into memory, processes vertices, and writes updates back to disk is explicitly defined as a pair of callbacks (iteration_start() and iteration_end()). It took us only a few minutes to find these iterations although we had never studied GraphChi before. In a data-parallel system such as Hadoop, the code for a Map or Reduce task can be considered as an iteration because it is repeatedly executed to process data partitions.

There is a strong correlation between the lifetime of an object and the lifetime of the iteration in which it is created: such objects often stay alive until the end of the iteration but rarely cross multiple iterations. Hence, we develop an *iteration-based memory management* that allocates data objects created in one iteration together in a native region and deallocates the region as a whole when the iteration finishes. There may be a small number of control objects that are also created in the iteration, and naïvely reclaiming the whole region may cause failures. We rely on developers to refactor the program code to move the creation of control objects out of the data path. In reality, this effort is very little because a data path rarely creates control objects (but it does use control objects passed from the control path).

Summary of Results. We have implemented the FACADE compiler based on the Soot compiler framework (McGill 2014; Vallée-Rai et al. 2000), which supports most of the Java 7 features. To use FACADE, the user identifies iterations and specifies the data path by providing a list of Java classes to be transformed. FACADE automatically synthesizes conversion functions for data objects that flow across the boundary and inserts calls to these functions at appropriate program points to convert data formats. We have applied FACADE to seven commonly used applications on three real-world, already well-optimized data processing frameworks: GraphChi, Hyracks, and GPS. Our experimental results demonstrate that (1) the transformation is very fast (e.g., less than 20 seconds) and (2) the generated code is much more efficient and scalable than the original code (e.g., runs up to 2× faster, consumes up to 2× less memory, and scales to much larger datasets).

2 A STUDY OF MEMORY BLOAT IN DATA-INTENSIVE SYSTEMS

In this section, we study two popular data-intensive systems, Giraph (Apache 2014b) and Hive (Apache 2014d), to investigate the impact of creating Java objects to represent and process data on performance and scalability. Our analysis aims to understand two problems: (1) how large the space overhead is due to object headers and references, and how it hurts the packing factor of memory, and (2) how the creation of massive numbers of objects affects the GC performance and why.

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 data-intensive 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 contents. Space inefficiencies are exacerbated by the pervasive utilization of object-oriented data structures. These data structures often use multiple-level delegations to achieve their functionality; a large amount of space is actually used to store *pointers* instead of 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 (Mitchell and Sevitsky 2007) has been conducted to understand the health of Java collections, our analysis is specific to data-intensive applications where a huge amount of data flows through a fixed amount of memory in a batch-by-batch manner.

To analyze the packing factor for the heap of a data-intensive application, we use the PageRank algorithm (Page et al. 1999) 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 weights of its inbound neighbors. This algorithm is widely used to rank web pages in search engines.

We ran PageRank on different open-source cloud computing systems, including Giraph (Apache 2014b), Spark (Zaharia et al. 2010), and Mahout (Apache 2014e), using a six-rack, 180-machine research cluster. Each machine has two 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 that all of these systems crashed with `java.lang.OutOfMemoryError`. A detailed inspection of the size of each partition processed by each node shows that the maximum partition size is 1.2GB—measured by `pmap` after loading the input partition—which is well below the size of the physical memory on each node. The heap was exhausted because data was inflated significantly after being loaded into memory and there was an extremely large volume of auxiliary data structures created to help process it.

To find the root cause of this data inflation, we performed a quantitative analysis using PageRank. Giraph contains an example implementation of the PageRank algorithm. Part of its data representation implementation[1] 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;
```

[1]In revision 1232166.

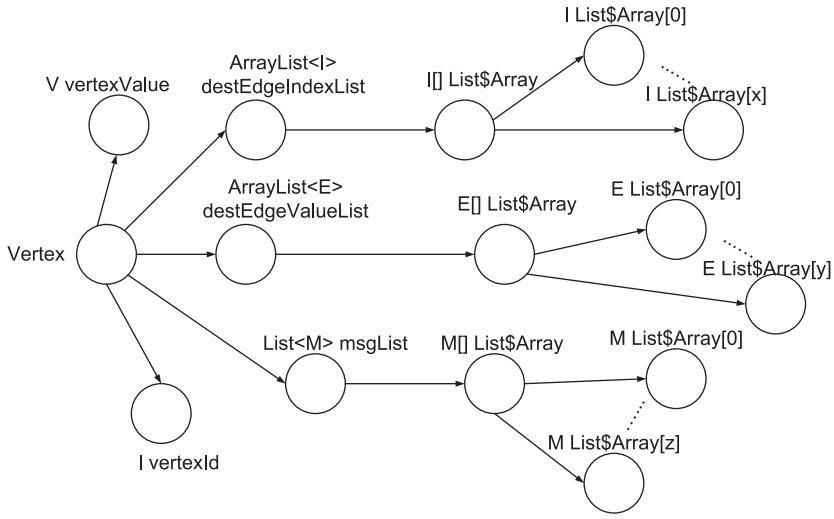

Fig. 2. A Giraph object subgraph rooted at a vertex.

```
    /** 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 processed by 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. Fields `destEdgeIndexList` and `destEdgeValueList` reference, respectively, a list of IDs and a list of values of outgoing edges. `msgList` contains incoming messages sent to the vertex from the previous iteration. Figure 2 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`, `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 2 shows the memory consumption statistics of a vertex data structure in the heap of 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

Table 2. Numbers of Objects per Vertex and Their Space Overhead (in Bytes) in PageRank in the Sun 64-Bit HotSpot JVM

Class	#Objects	Header (bytes)	Pointer (bytes)
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	$m+n+9$	$8(m+n)+84$	$8(m+n)+64$

The vertex has m outgoing edges and n incoming messages.

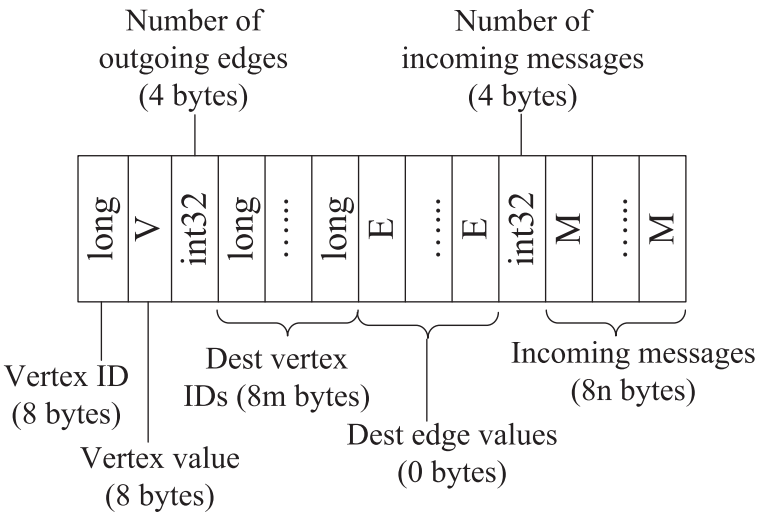

Fig. 3. The compact layout of a vertex in an ideal case.

vertex in the current implementation is $16(m+n)+148$ (i.e., the sum of the header size and pointer size in Table 2).

On the contrary, Figure 3 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. The memory consumption of this ideal layout is even less than half of the space used for object headers and pointers in the object-based representation. In this case, the space overhead of the object-based representation is greater than 200%. One of the challenges the rest of this article tries to address is how to design a memory system that can provide close-to-ideal space efficiency for the storage of data objects.

2.2 Large Volumes of Objects and References

In a JVM, the GC threads periodically traverse the live object graph in the heap to reclaim unreachable objects. If 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 data-intensive application, its object graph often consists of a great number of

isolated object subgraphs, each of which represents either a data item or a data structure created for processing data 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 illustrate the problem. This exception was found in a discussion thread named "how to deal with Java heap space errors":[2]

```
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.RowContainer.next(RowContainer.
  java:263)
  at org.apache.hadoop.hive.ql.exec.persistence.Row(bold)Container.next(RowContainer.
  java:74)(bold)
  at org.apache.hadoop.hive.ql.exec.CommonJoinOperator.checkAndGenObject
  (CommonJoinOperator.java:823)
  at org.apache.hadoop.hive.ql.exec.JoinOperator.(bold)endGroup(JoinOperator.java:263)(bold)
  at org.apache.hadoop.hive.ql.exec.ExecReducer.reduce(ExecReducer.java:198)
  ......
  at org.apache.hadoop.hive.ql.exec.persistence.RowContainer.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 `Row` objects from one of the input branches in a `RowContainer`. This `RowContainer` has the same lifetime as `JoinOperator` (i.e., `JoinOperator` first creates the `RowContainer` object in its initialization, populates and processes it, and does not release it until the processing is about to finish). In cases where a large number of `Row` objects are 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, resulting in the `OutOfMemory` error.

Even if the system has sufficient memory for the application, the large number of `Row` objects would still cause severe performance degradation. Suppose the number of `Row` objects in the `RowContainer` is r. The GC time for traversing the internal structure of the `RowContainer` object is at least O(r). For Hive, r grows proportionally with the size of the input data, which can easily drive the GC cost up substantially. The following example shows a user report from StackOverflow.[3] Although this problem has a different manifestation, its root cause is the same as that of the previous example (i.e., too many objects).

"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."

[2] http://mail-archives.apache.org/mod_mbox/hive-user/201107.mbox/.
[3] http://stackoverflow.com/questions/11387543/performance-tuning-a-hive-query.

In addition to the need to traverse an extremely large number of objects, another important reason for the high memory management cost is that traditional GC algorithms are not designed for data-intensive systems. For example, a generational GC splits objects into a young and an old generation. Objects are allocated in the young generation initially. When a nursery GC runs, it uses cross-generation references as the roots to traverse the young generation, promotes reachable objects to the old generation, and then reclaims the entire young generation. The generational GC is fast because its generational hypothesis—the most recently created objects are also those most likely to become unreachable quickly—holds for most regular non-data-intensive applications.

However, data-intensive applications often violate this hypothesis because their data manipulation functions, while simple in code size, need to process very large datasets and thus, run for a long time. For example, a computational iteration in Giraph holds all vertices in an array and iteratively invokes the user-defined update function on these vertices. Hence, all of these vertex objects and the objects reachable from them cannot be reclaimed until the end of the iteration. The average time span for an iteration when the Yahoo Webgraph was processed on Giraph (under the same configuration as described earlier in this section) is 105 seconds, which contains an average of 44 GC runs. The heap traversal effort of these 44 GC runs is almost completely wasted because the amount of reclaimed memory is very little.

We have also conducted experiments to verify the nongenerational property of data items in Big Data applications. Figure 4 depicts the memory footprint and its correlation with epochs when PageRank (PR) and ConnectedComponents (CC) were executed on GraphChi to process the *twitter-2010* graph on a server machine with two Intel(R) Xeon(R) CPU E5-2630 v2 processors running CentOS 6.6. The default Parallel Scavenge GC was used. In this GraphChi experiment, GraphChi finished, respectively, in 2,337 and 3,227 seconds, of which 1,289 (55.2%) and 1,324 (41.1%) seconds were spent on the GC. Each epoch lasts about 20 seconds (PR) and 40 seconds (CC), denoted by dotted lines in Figure 4. We can observe a clear correlation between the endpoint of each epoch and each significant memory drop (Figure 4(a)) as well as each large memory reclamation (Figure 4(b)). During each epoch, many GC runs occur and each reclaims little memory (Figure 4(b)).

Strawman Given such epochal behaviors, can we solve the problem by forcing GC runs to happen *only* at the end of epochs? This simple approach would not work due to the multithreaded nature of real systems. In systems like GraphChi, each epoch spawns many threads that collectively consume a huge amount of memory. Waiting until the end of an epoch to conduct GC could easily cause out-of-memory crashes. In systems like Hyracks (Borkar et al. 2011), a distributed dataflow engine, different threads have various processing speeds and reach epoch ends at different times. Invoking the GC when one thread finishes an epoch would still make the GC traverse many live objects created by other threads, leading to wasted effort.

The extremely large GC overhead in data-intensive applications has recently received much attention from the memory management community: for example, NumaGiC (Gidra et al. 2015)—a new GC for "Big Data" on NUMA machines—has been proposed to take data location into consideration when performing allocation and collection. Despite this GC support, the large volumes of data objects in the heap still need to be frequently traversed, which would inevitably cause long pauses during the execution. Our own work, Facade (Nguyen et al. 2015) and Broom (Gog et al. 2015) attempt to move data objects to regions so that they are not subject to garbage collection. Work from Maas et al. (2015, 2016) goes in a different direction: they develop a distributed runtime system that can coordinate GC invocations in a centralized manner rather than optimizing memory management on each node.

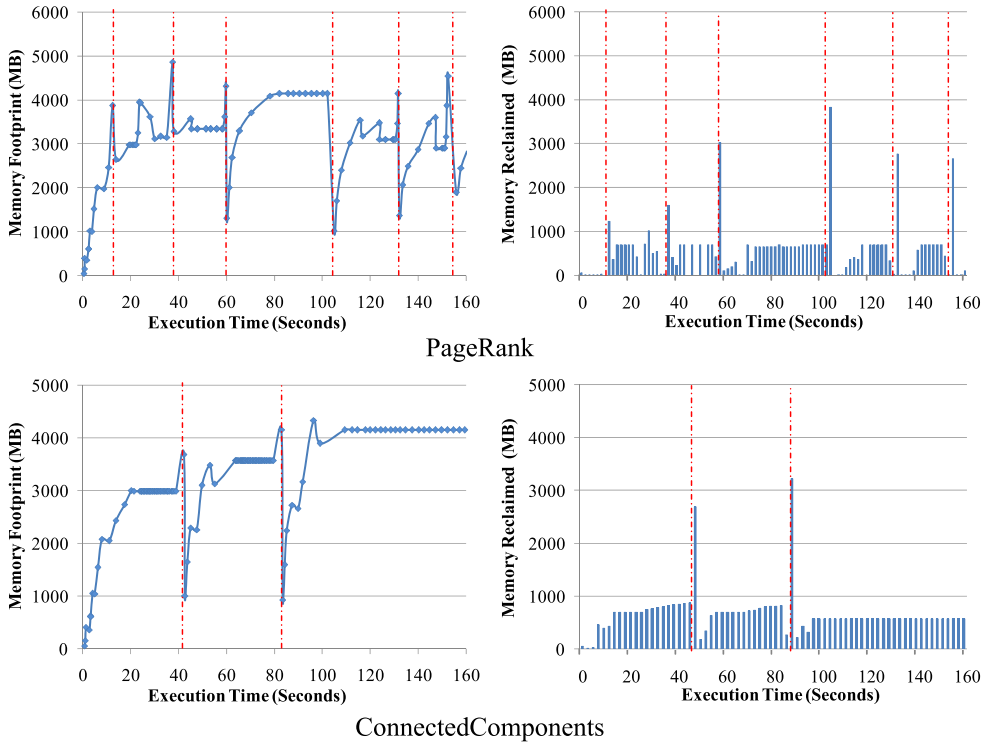

Fig. 4. Memory footprint for the first 160 seconds of the executions when GraphChi runs PageRank (top) and ConnectedComponents (bottom). Each dot in (a) represents the memory consumption measured right after a GC; each bar in (b) shows how much memory is reclaimed by a GC; dotted vertical lines show the iteration boundaries.

Discussion. The study shows that real-world data-intensive applications are designed and implemented in the same way as regular object-oriented programs, by developers educated in the culture of object orientation. They follow the long-held programming principle: *everything is an object*. Objects are used for both *data storage* (i.e., storing data fields) and *data manipulation* (i.e., providing methods that process data in the fields). While creating such objects in the control path to drive the flow of the program may not have a significant impact on performance, doing so in the data path creates a big scalability bottleneck because there is an overhead associated with each object representation of data item (e.g., `Vertex` and `Edge` objects) and the number of data objects is huge.

From the developer's perspective, though, there is not much optimization that can be done, because the problem is inherent to the managed runtime. For example, all of the data-processing-related interfaces in Hive require the passing of Java objects as representations of data items—to manipulate data contained in `Row`, one has to wrap it into a `Row` object, as designated by the interface. If developers want to manually solve this performance problem, they would have no choice but to redesign these interfaces from scratch, a task nobody could afford to do in reality.

While there exist arguments (Mozilla 2014) that GC should be eliminated for systems software, we found that GC is very useful in reclaiming control objects that can flow all over the program.

In a typical data processing system, the control path (that executes the pipeline and manages the distributed runtime) has a much more complex logic than the data path, and its behavior is very similar to regular, non-data-intensive programs. Hence, manually allocating/deallocating control objects is error prone, and it can also significantly slow down the development progress. Adding an extra memory management layer on top of the GC to handle data objects whose lifetime exhibits clear patterns is a more viable choice than completely removing the GC.

These observations motivate us to investigate automated solutions at the compiler/systems level so that we can overcome these fundamental limitations of object orientation while still allowing developers to fully enjoy the benefit of a managed, object-oriented language.

3 THE FACADE EXECUTION MODEL

To overcome the fundamental problem of memory bloat, we design the Facade framework that exploits compiler and runtime system support to separate data storage from data manipulation in a data-intensive program. The key idea is simple: data contents are allocated separately in native memory; heap objects no longer contain data, and they only provide data-manipulating methods. Objects now only represent *data processors*, not *data contents*, and hence, the number of heap objects is no longer proportional to the cardinality of the input dataset.

3.1 Data Storage Based on Native Memory

We propose to store data records in native, non-GCed memory. Similarly to regular memory allocation, our data allocation operates at the page granularity. A memory page is a fixed-length contiguous block of memory in the off-heap memory, obtained through a JVM's native support.

To provide a better memory management interface, each native page is wrapped into a Java object, with functions that can be inserted by the compiler to manipulate the page. Note that the number of page objects (i.e., p in $O(t * n * m + p)$) cannot be statically bound in our system, as it depends on the amount of data to be processed. However, by controlling the size of each page and recycling pages, we often need only a small number of pages to process a large dataset. The scalability bottleneck of an object-oriented data-intensive application lies in the creation of small data objects and data structures containing them; our system aims to bound their numbers.

From a regular Java program P, Facade generates a new program P', in which the data contents of each instantiation of a data class are stored in a native memory page rather than in a heap object. To facilitate transformation, the way a data record is stored in a page is exactly the same as the way it was stored in an object except that the native-memory-based record does not contain a heap object's header and padding.

Figure 5 shows the data layout for an example data structure in our page-based storage system. Each data record (which used to be represented by an object in P) starts with a 2-byte type ID, representing the type of the record. For example, the IDs for `Professor`, `Student[]`, `String`, and `Student` are 12, 25, 4, and 13, respectively. These types will be used to implement virtual method dispatch during the execution of P'. Type ID is followed by a 2-byte lock field, which stores the ID of a lock when the data record is used to synchronize a block of code. We find it sufficient to use 2 bytes to represent class IDs and lock IDs—in our experiments with large systems, the number of data classes is often much smaller than 2^{15}, and so is the number of distinct locks needed. Details of the lock implementation and the concurrency support can be found in Section 4.5.

For an array record, the length of the array (4 bytes) is stored immediately after the lock ID. In the example, the number of student records in the array is nine. The actual data contents (originally stored in object fields) are stored subsequently. For instance, field `id` of the professor record contains an integer 1254; `numStudents` stores an integer 9; the fields `students` and `name` contain memory addresses `0x0504` and `0x070a`, respectively. These references are referred to as *page*

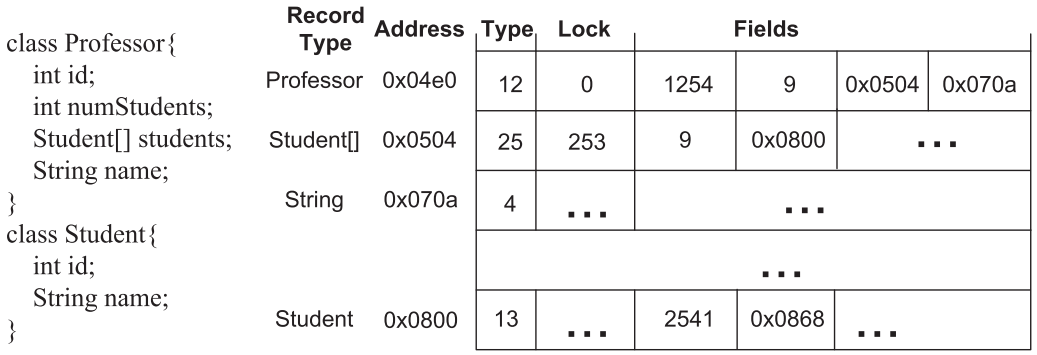

Fig. 5. A data structure in regular Java and its corresponding data layout in a native page.

references, as opposed to *heap references* in a normal Java program. Note that for efficiency, page references are not offsets into the native page where the memory is held; they are the absolute memory address so that the FACADE runtime can operate directly on them without further calculation.

3.2 Using Objects as Facades

We propose to create heap objects as *facades* for a data class; that is, they are used only for *control* purposes such as method calls, parameter passing, or dynamic type checks, but do *not* contain actual data. Figure 6 and Figure 7 depict an example with five transformations using the same `Professor` class in Figure 5. For simplicity of illustration, we show the unoptimized version of the generated program, under the assumption that the program is single-threaded and free of virtual calls. We will discuss the support of these features later.

Class Transformation. The transformations #1 and #2 from Figure 6 show an example of class transformation. For illustration, let us assume both `Professor` and `Student` are data classes. For `Professor`, FACADE generates a facade class `ProfessorFacade`, containing all methods defined in `Professor`. `ProfessorFacade` extends class `Facade`, which has a field `pageRef` that records the page reference of a data record (such as `0x0504` in Figure 5). Setting a page reference to the field `pageRef` of a facade *binds* the data record with the facade, so that methods defined in the corresponding facade class can be invoked on the facade object to process the record. A reader can think of this field as the `this` reference in a regular Java program.

`ProfessorFacade` does not contain any instance fields; for each instance field f in `Professor`, `ProfessorFacade` has a static field f_Offset, specifying the offset (in number of bytes) of f to the starting address of the data record. These offsets will be used to transform field accesses.

Method Transformation. For method `addStudent` in `Professor`, FACADE generates a new method with the same name in `ProfessorFacade`. Because we no longer have any data objects, for each reference of a data object in the original program, we substitute it with either a page reference or a reference of a corresponding facade object using the following criteria:

—For any assignment, load, or store that involves a data object, the reference of the data object is substituted with its page reference. For example, in Figure 7, transformation #4 replaces the variable assignments (lines 14–15) in P with page reference assignments (lines 37–38) in P'. In these cases, the generated statements do not have any heap objects involved.

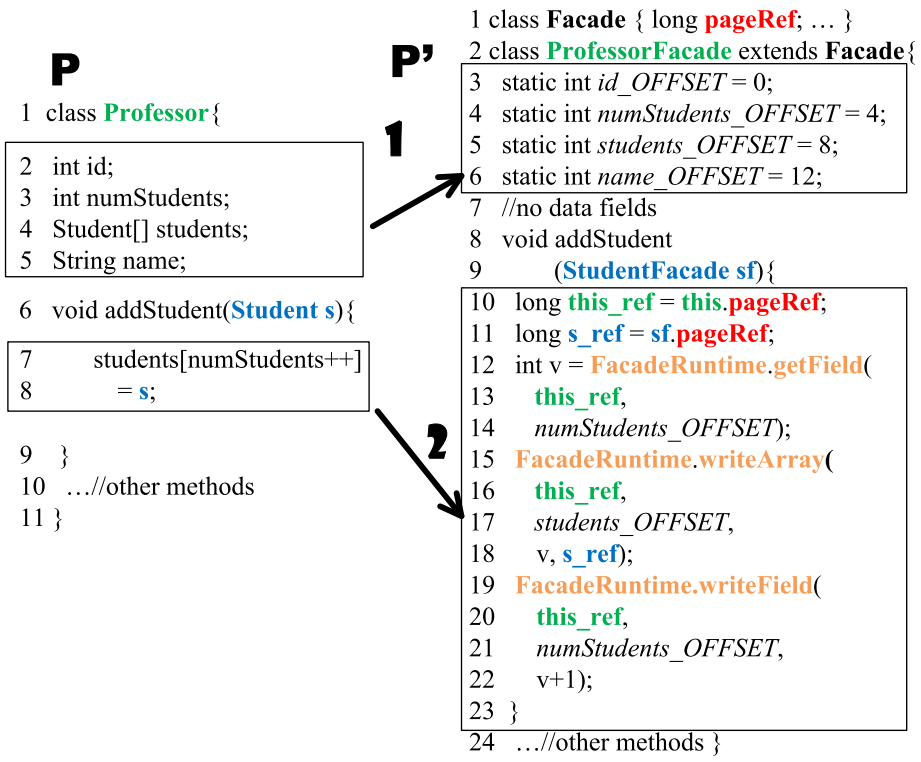

Fig. 6. A transformation example, part (a).

—For parameter passing and value return in a call site, it is difficult to substitute object references completely with page references, because if an object is used as a receiver object to call a method, replacing it with a page reference would make it impossible to make the call. In this case, we replace references to data objects with references to their corresponding facade objects. For example, in Figure 6, the signature of method `addStudent` is changed in a way so that the `Student` type parameter is replaced with a new parameter of type `StudentFacade`.

In the generated `addStudent` method (lines 8–23 in Figure 6), the new facade parameter *sf* is used only to pass the page reference of the data record that corresponds to the original parameter in *P*. The first task inside the generated method is to retrieve the page references (lines 10 and 11 in *P′*) from the receiver (i.e., `this`) and *sf*, and store them in two local variables *this_ref* and *s_ref*. Any subsequent statement that uses `this` and *s* in *P* will be transformed to use the page references *this_ref* and *s_ref* in *P′*, respectively. The field accesses at lines 5 and 6 in *P* are transformed to three separate calls to our library methods that read values from and write values to a native page. Note that what is written into the array is the page reference *s_ref* pointing to a student record—all references to regular data objects in *P* are substituted by page references in *P′*.

Allocation Transformation. In Figure 7, the allocation at lines 12 to 13 in *P* is transformed to lines 29 to 36 in *P′*. Facade allocates space based on the size of type `Student` by calling a library method `allocate`, which performs native-memory-page-based allocation and returns a page

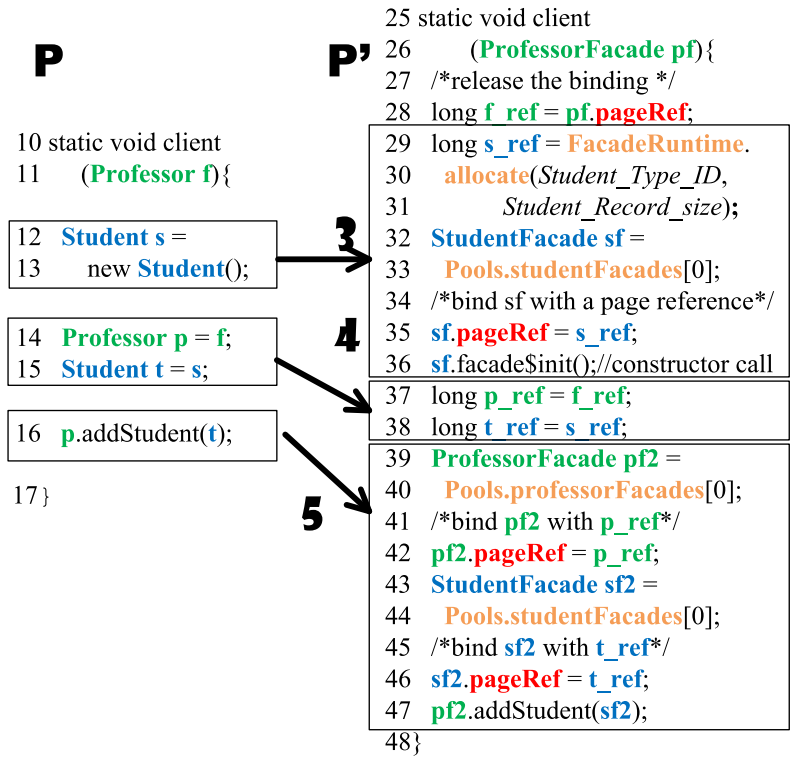

Fig. 7. A transformation example, part (b).

reference *s_ref*, which is a native memory address. Details of the allocation algorithm and memory management are discussed in Section 4.7.

Call Site Transformation. Since statements in P' all use page references, a challenge in transforming a call site is how to generate the receiver object on which the call can be made. Our idea is to use facade objects. If a call is made on a data object in P, we can obtain a facade object to call the same method in P', because a data class and its corresponding facade class have the same methods. However, doing so naïvely would generate a large number of facade objects, which would still cause space and memory management overhead. Hence, special care needs to be taken to minimize the number of facade objects used.

We solve the problem by pooling facade objects. For each facade class, we maintain a pool that contains a small number of objects of the class. This number can be statically bounded as discussed shortly. Before generating a call site, the FACADE compiler first generates code to retrieve an available facade object from the pool (lines 32–33 in P') and bind it with the page reference *s_ref* (lines 35). In this example, the first facade in the pool is available; the reason will be explained shortly. The constructor of class `Student` in P is converted to a regular method `facade$init` in P'. FACADE then generates a call to `facade$init` on the retrieved facade object (line 36).

Similarly, a call to method `addStudent` on the `Professor` object in P (line 16) is transformed to a call to the same method on the `ProfessorFacade` object in P' (line 47). This new call site needs (1) a receiver object and (2) a parameter object. To prepare for these objects, we generate the statements from line 39 to line 44, which retrieve a `ProfessorFacade` object *pf2* for the receiver

and a `StudentFacade` object *sf2* for the parameter from their respective pools, and then bind them with their corresponding page references. Finally, the call site at line 47 is generated.

Note that the `ProfessorFacade` object *pf2* and the `StudentFacade` object *sf2* are needed because the object references *p* and *s* in *P* have been replaced with page references *p_ref* and *s_ref* in *P′*, both of which have a `long` type. It would not be possible to call `addStudent` with these (long) page references. Hence, facade objects are retrieved to (1) enable the method call and (2) take the page references into the callee.

Code Generation Invariants. Our code transformation algorithm maintains the following three major invariants, guaranteeing transformation correctness. First, for each reference *r* of a data object in *P*, *P′* must contain a page reference *pr* pointing to the native memory location at which the same object is stored. Since there is a one-to-one mapping between *r* and *pr*, any noncall statement that reads/writes the object referenced by *r* in *P* must have a corresponding statement that reads/writes the native-memory-based object referenced by *pr* in *P′*.

Second, for two different references *r* and *r′* in *P*, the two corresponding page references *pr* and *pr′* in *P′* must be different as well. While facade objects are reused, page references are never shared among variables. This is straightforward to see given that FACADE performs *literal translation* for allocation sites, loads, stores, and assignments.

Third, for each parameter (including receiver) of a data class *D* at each call site in *P*, a facade object of type *DFacade* is retrieved in *P′* from the *DFacade* object pool. The only purpose of the facade object is to pass a page reference between a caller and a callee. For example, for parameter passing, the page reference is written into a facade object right before the call, and then released from the facade and written into a local variable in the very beginning of the callee. For value returning, the page reference is written into a facade right before the return statement and released and written into a stack variable immediately after the call site.

More formally, the invariant regarding the facade usage is that for a pair of instructions (e.g., *s* and *t*) that bind a facade with a page reference and release the binding, *t* is the immediate successor of *s* on the data dependence graph. In other words, no instructions between *s* and *t* can read or write the facade object accessed by *s* or *t*. We refer to the period between the executions of *s* and *t* as a *use span* of the facade accessed by *s* and *t*. Examples of such instruction pairs include lines 42 and 10, and lines 46 and 11 in *P′* of Figure 7. This invariant guarantees that the page reference read from the facade object by *t* is exactly the one written into the same facade object by *s*, and thus, page references are appropriately propagated between methods.

3.3 Bounding the Number of Facades in Each Thread

Our facade pooling is different from traditional object pooling where the objects requested cannot be reused until they are explicitly returned to the pool. A facade object does not need to be explicitly returned because its goal is only to carry a page reference across the method boundary. Its use span automatically ends when the callee returns to the caller (if used for value return) or the callee is about to execute (if used for parameter passing). In other words, the way facades are used dictates that the use spans of the facade objects requested at different statements are completely *disjoint*. Hence, in most cases, upon a request for a facade (e.g., at a call site), all facades in the pool are available to use. This explains why it is always safe to use the first facade of the pool at lines 33, 40, and 44 in Figure 7.

One exception is that if a call site has multiple parameters of the same data class, multiple objects of the corresponding facade class are needed simultaneously to pass page references. Hence, the number of facades needed for a data class depends on the number of parameters of this class needed in a call site. For example, if a call site in *P* requires *n* parameters of type `Student`, we need

at least n `StudentFacade` objects in P' for parameter passing (e.g., `Pools.studentFacades[0]`, ..., `Pools.studentFacades[n - 1]`). The number of facades for type `StudentFacade` in P' is thus bounded by the maximal number of `Student`-type parameters needed by a method call in P. Based on this observation, we can inspect all call sites in P in a pretransformation pass and compute a bound statically for each data class. The bound will be used to determine the size of the facade pool for that type (e.g., `Pools.studentFacades`) *at compile time*.

3.4 Performance Benefits

P' has the following two performance advantages over P. First, all data records are stored in native pages and no longer subject to garbage collection. This can lead to an orders-of-magnitude reduction in the number of nodes and edges traversed by the GC.

Second, significant reduction in memory consumption can be achieved for the following two reasons: (1) Each data record has only a 4-byte "header" space (8 bytes for an array) in P', while the size of an object header is 12 bytes (16 bytes for an array) in P. This is due to the reduction of the lock space as well as the complete elimination of space used for GC. (2) As discussed shortly in Section 4.10, Facade inlines all data records whose size can be statically determined, which reduces memory consumption for storing object headers and improves data locality.

4 FACADE DESIGN AND IMPLEMENTATION

To use Facade, a user needs to provide a list of data classes that form the data path of an application. Our compiler transforms the data path to page-allocate objects representing data items without touching the control path. This handling enables the design of simple intraprocedural analysis and transformation as well as aggressive optimizations (such as type specialization), making it possible for Facade to scale to large-scale framework-intensive systems.

4.1 Our Assumptions

Based on the (user-provided) list of data classes, Facade makes two important "closed-world" assumptions based on our experience with dozens of real-world data-intensive systems. The first one is a *reference-closed-world* assumption that requires all reference-typed fields declared in a data class to have data types. This is a valid assumption—there are two major kinds of data classes in a data-intensive application: classes representing data tuples (e.g., graph nodes and edges) and those representing data manipulation functions, such as sorter, grouper, and so forth. Both kinds of classes rarely contain fields of nondata types. Java supports a collections framework, and data structures in this framework can store both data objects and nondata objects. In Facade, a collection (e.g., HashMap) is treated as a data class; a new class (e.g., HashMapFacade) is thus generated in the data path. The original class is still used in the control path. If Facade detects that a data object flows from the control path to the data path or a paged data record flows the other way around, it automatically synthesizes a *data conversion function* to convert data formats. Detailed discussion can be found in Section 4.6.

The second assumption is a *type-closed-world* assumption, requiring that for a data class c, c's superclasses (except `java.lang.Object`, which is the root of the class hierarchy in Java) and subclasses must be data classes. This is also a valid assumption because a data class usually does not inherit a nondata class (and vice versa). The assumption makes it possible for us to determine the field layout of a data record in a page—fields declared in a superclass are stored before fields in a subclass and their offsets can all be statically computed. The Facade compiler computes a closure of classes to be transformed from an initial list of user-specified data classes based on the inheritance relationships.

We allow both a data class and a nondata class to implement the same Java interface (such as `Comparable`). Doing this will not create any page layout issue because an interface does not contain instance fields. FACADE checks these two assumptions before transformation and reports compilation errors upon violations. The developer needs to refactor the program to fix the violations.

4.2 Data Class Transformation

Class Hierarchy Transformation. For each method m in a data class D, FACADE generates a new method m' in a facade class *DFacade* such that m and m' have the same name; for each parameter of a data class type T in m, m' has a corresponding parameter of a facade type *TFacade*. No special treatment is required for overloaded methods since each method m has its distinct FACADE method m'. If D extends another data class E, this relationship is preserved by having *DFacade* extend *EFacade*. All static fields declared in D are also in *DFacade*; however, *DFacade* does not contain any instance fields.

One challenge here is how to appropriately handle Java interfaces. If an interface I is implemented by both a data class C and a nondata class D, and the interface has a method that has a data-class type parameter, changing the signature of the method will create inconsistencies. In this case, we create a new interface *IFacade* with the modified method and make all facades *DFacade* implement *IFacade*. While traversing the class hierarchy to transform classes, FACADE generates a type ID for each transformed class. This type ID is actually used as a pointer that points to a facade pool corresponding to the type—upon a virtual dispatch, the type ID will be used to retrieve a facade of the appropriate type at runtime.

Instruction Transformation. Instruction transformation is performed on the control flow graph (CFG) of a single static assignment (SSA)-based intermediate representation (IR). The output of the transformation is a new CFG containing the same basic block structures but different instructions in each block. The transformations for different kinds of instructions are summarized in Table 3. Here we discuss only a few interesting cases. For a field write-in (i.e., $a.f = b$ in case 3), if b has a data type but a does not (case 3.3), FACADE considers this write as an *interaction point* (IP), an operation at which data flows across the control-data boundary. FACADE synthesizes a data conversion function `long convertToB(B)` that converts data format from a paged data record back to a heap object (see Section 4.6). If a has a data type but b does not (case 3.4), FACADE generates a compilation error as our first assumption (that data types cannot reference nondata types) is violated. The developer needs to refactor the program to make it FACADE transformable.

An IP may also be a load that reads a data object from a nondata object (case 4.3) or a method call that passes a data object into a method in the control path (case 6.3). At each IP, data conversion functions will be synthesized and invoked to convert data formats. Note that data conversion often occurs before the execution of the data path or after it is done. Hence, these conversion functions would often not be executed many times and cause much overhead.

Resolving Types. In two cases, we need to emit a call to a method named `resolve` to resolve the runtime type corresponding to a page reference. First, when a virtual call $a.m(b, \ldots)$ is encountered (case 6.1), the type of the receiver variable a often cannot be statically determined. Hence, we generate a call `resolve`(a_ref), which uses the type ID of the record pointed to by a_ref to find a facade of the appropriate type. However, since this information can be obtained only at runtime, it creates difficulties for the compiler to select a facade object as the receiver from the pool (i.e., what index i should be used to access `Pools.aFacades`[i]).

To solve the problem, we maintain a separate *receiver facade pool* for each data class. The pool contains only a single facade object; the `resolve` method always returns the facade from this

Table 3. A summary of Code Generation (Suppose Variables a and b Have Types A and B, Respectively)

Instructions in P	***Conditions***	***Code generation in*** P'
(1) Method prologue	(1.1) s is a parameter of data type in P	Create a variable s_ref for each facade parameter sf; emit instruction $s\_ref = sf.pageRef$; add $\langle s, s\_ref \rangle$ into the variable-reference table v
(2) $a = b$	(2.1) a has a data type	Look up table v to find the reference variable b_ref for b; emit instruction $a\_ref = b\_ref$; add $\langle a, a\_ref \rangle$ into v
	(2.2) Otherwise	Generate $a = b$
(3) $a.f = b$	(3.1) Both a and b have data types	Retrieve a_ref and b_ref from table v; emit a call `setField` (a_ref, f_Offset, b_ref)
	(3.2) Neither of them have a data type	Emit $a.f = b$
	(3.3) b has a data type, a doesn't (Interaction Point)	Synthesize a data conversion function $BcovertToB(long)$; emit a call $a.f = convertToB(b\_ref)$
	(3.4) a has a data type, b doesn't	Assumption violation; generate a compilation error
(4) $b = a.f$	(4.1) Both a and b have data types	Retrieve a_ref from table v; emit a call b_ref = `getField` (a_ref, f_Offset); add $\langle b, b\_ref \rangle$ into v
	(4.2) Neither of them have a data type	Emit instruction $b = a.f$
	(4.3) a has a data type, b doesn't (Interaction Point)	Synthesize a data conversion function $longcovertFromB(B)$; emit a call $b\_ref = convertFromB(a.f)$; add $\langle b, b\_ref \rangle$ into v
	(4.4) b has a data type but a doesn't	Assumption violation; generate a compilation error
(5) *returna*	(5.1) a has a data type	Retrieve a_ref from v; emit three instructions: $AFacadeaf = Pools.aFacades[0]$; $af.pageRef = a\_ref$; *returnaf*
	(5.2) Otherwise	Emit instruction *returna*
(6) $a.m(\ldots, b, \ldots)$	(6.1) Both a and b have data types; b is the i-th parameter that has type B	Retrieve a_ref and b_ref from table v; emit five instructions: $AFacadeaf = resolve(a\_ref)$; $BFacadebf = Pools.bFacades[i]$; $af.pageRef = a\_ref$; $bf.pageRef = b\_ref$; $af.m(\ldots, bf, \ldots)$
	(6.2) a has a data type, b doesn't	Emit the same instructions as (6.1), except the last call is $af.m(\ldots, b, \ldots)$
	(6.3) b has a data type, a doesn't (Interaction Point)	Synthesize function $BcovertToB(long)$; emit a call $a.m(\ldots, covertToB(b\_ref), \ldots)$
	(6.4) Neither of them have a data type	Emit a call $a.m(\ldots, b, \ldots)$
(7) $booleant = a$ *instanceof* B	(7.1) a has a data type and B is a data type	Retrieve a_ref from v; emit two instructions: $AFacadeaf = resolve(a\_ref)$; $t = af instanceof BFacade$
	(7.2) B is an array type	Emit $t = arrayTypeID(a) == ID(B)$
	(7.3) Neither of them have a data type	Emit $t = a instanceof B$
(8) $a == b$	(8.1) a and b have a data type	Retrieve a_ref and b_ref from table v; emit a call $a\_ref == b\_ref$
	(8.2) They neither have a data type	Emit $a == b$

pool, which is distinct from the parameter pool. Note that we do not need to resolve the type of a parameter (say, b), because b is not used as a receiver to call a method. We can simply obtain a facade from the parameter pool based on b's declared (static) type, and use it to carry b's page reference.

The second case in which we need a `resolve` is the handling of an `instanceof` type check, which is shown in case 7 of Table 3.

4.3 Type Specialization for `Object` and `Object[]`

Variables and parameters whose static types are `Object` and `Object[]` introduce additional challenges, because whether they reference data objects or not cannot be determined statically. Instead of using a complicated case analysis that generates different handling for different runtime types, Facade speculatively treats these variables as data-typed variables and generates code to validate this assumption at runtime. Upon a violation (e.g., a variable/parameter is not an instance of `Facade`), the generated program P' will throw an exception, and the developer can "blacklist" these variables/parameters to disable the speculation and recompile the program.

The usage of these general types depends heavily on applications. In fact, in the three frameworks we have experimented with, they rarely declare variables with `Object` and `Object[]`. We have only encountered six methods (in the application code) with parameters of the `Object` or `Object[]` type, and these parameters were indeed used to pass data objects. However, for other applications such as Hive, methods with general-type parameters are extensively used. After a detailed inspection of Hive, we found almost all of these parameters represent data objects—since Hive is a data warehouse, it is designed to process queries in a way that is very similar to a database. Many methods simply perform filtering or aggregation on generic data records regardless of their types. Hence, we expect our speculative handling to be still effective for those applications.

4.4 Computing Bounds

Before the transformation, Facade inspects the parameters of each method in the data path to compute a bound for each data class. This bound will be used as the length of the facade array (i.e., the parameter pool) for the type. Note that the bound computation is based merely on the static types of parameters. Although a parameter with a general type may receive an object of a specific type at runtime, a facade of the general type will be sufficient to carry the page reference of the data record (as discussed above) from a caller to a callee. Since we use a separate pool for receivers, the target method will always be executed appropriately. If the declared type of parameter is an *abstract* type (such as interface) that cannot have concrete instances, we find an arbitrary (concrete) subtype c of this abstract type and attribute the parameter to c when computing bounds. Facade generates code to retrieve a facade from c's pool to pass the parameter.

Once the bound for each data class is calculated, Facade generates the class `Pools` by allocating, for each type, an array as a field whose length is the bound of the type. The array will be used as the parameter pool for the type. Facade generates an additional field in `Pools` that references its receiver pool (i.e., one single facade) for the type. Eventually, Facade emits an `init` method in `Pools`, which will be invoked by our library to create facade instances and populate parameter pools.

4.5 Supporting Concurrency

Naïvely transforming a multithreaded program may introduce concurrency bugs. For example, in P', two concurrent threads may simultaneously write different page references into the same facade object, leading to a data race. The problem can be easily solved by performing thread-local facade pooling: for each data class, the receiver pool and the regular pool are maintained for each

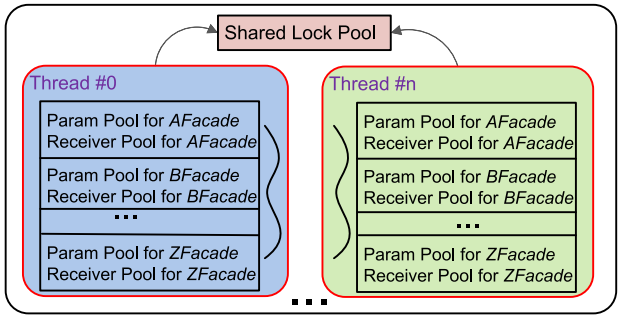

Fig. 8. A graphical representation of threads and pools, where *AFacade*, *BFacade*, . . . , and *ZFacade* are facade types.

thread. We implement this by associating one instance of class `Pools` with each thread; the `init` method (discussed in Section 4.4) is invoked upon the creation of the thread.

Both implicit and explicit locks are supported in Java. Explicit locking is automatically supported by FACADE: all `Lock`- and `Thread`-related classes are in the control path and not modified by FACADE. For implicit locking (i.e., the intrinsic lock in an object is used), we need to add additional support to guarantee the freedom of race conditions. One possible solution is as follows: for each object o that is used as a lock in a `synchronized` (o){. . .} construct (i.e., which is translated to an `enterMonitor`(o) and an `exitMonitor`(o) instruction to protect the code in between), FACADE emits code to obtain a facade o' corresponding to o (if o has a data type) and then generates a new construct `synchronized` (o'){. . .}. However, this handling may introduce data races—for two code regions protected by the same object in P, two different facades (and thus distinct locks) may be obtained in P' to protect them.

We solve the problem by implementing a special lock class and creating a new *lock pool* (shown in Figure 8) that is shared among threads; each object in the pool is an instance of the lock class. The lock pool maintains an atomic bit vector, each set bit of which indicates a lock being used. For each `enterMonitor`(o) instruction in P, FACADE generates code that first checks whether the lock field of the data record corresponding to o already contains a lock ID. If it does, we retrieve the lock from the pool using the ID; otherwise, our runtime consults the bit vector to find the first available lock (say, l) in the pool, writes its index into the record, and flips the corresponding bit. We replace o with l in `enterMonitor` and `exitMonitor`, so that l will be used to protect the critical section instead.

Each lock has a counter that keeps track of the number of threads currently blocking on the lock; it is incremented upon an `enterMonitor` and decremented upon an `exitMonitor`. If the number becomes zero at an `exitMonitor`, we return the lock to the pool, flip its corresponding bit, and zero out the lock space of the data record. Operations such as `wait` and `notify` will be performed on the lock object inside the block.

Worst-Case Object Numbers in P and P'. In P, each data item needs an object representation, and thus, the number of heap objects needed is $O(s)$, where s is the cardinality of the input dataset. In P', each thread has a facade pool for a data class. Suppose the maximum number of facades needed for a data class is m, a compile-time constant. The total number of facades in the system is thus $O(t * n * m)$, where t and n are the numbers of threads and data classes, respectively. Considering the additional objects created to represent native pages, the number of heap objects needed in P' is $O(t * n * m + p)$, where p is the number of native pages.

Note that the addition of the lock pool does not change this bound. The number of lock objects needed first depends on the number of synchronized blocks that can be concurrently executed (i.e., blocks protected by distinct locks), which is bounded by the number of threads t. Since intrinsic locks in Java are re-entrant, the number of locks required in each thread also depends on the depth of nested synchronized blocks, which is bounded by the maximal depth of runtime call stack in a JVM, a compile-time constant. Hence, the number of lock objects is $O(t)$ and the total number of objects in the application is still $O(t * n * m + p)$. In our evaluation, we have observed that the number of locks needed is always less than 10.

4.6 Data Conversion Functions

For each IP that involves a data class D, Facade automatically synthesizes a conversion function for D; this function will be used to convert the format of the data before it crosses the boundary. An IP can be either an *entry* point at which data flows from the control path into the data path or an *exit* point at which data flows in a reverse direction. For an entry point, a `long convertFromA(A)` method is generated for each involved data class A; the method reads each field in an object of A (using reflection) and writes the value into a page. Exit points are handled in a similar manner. Our experiments on three systems show that the number of conversion functions needed is very small ($\leq$4). This is expected and consistent with our observation that the control path and data path are well separated in data-intensive programs.

4.7 Memory Allocation and Page Management

The Facade runtime system maintains a list of pages, each of which has a 32KB space (i.e., a common practice in the database design (Graefe 1993)). To improve allocation performance, we classify pages into size classes (similarly to what a high-performance allocator would do for a regular program), each used to allocate objects that fall into a different size range. When allocating a data record on a page, we apply the following two allocation policies whenever possible: (1) back-to-back allocation requests (of the same size class) get contiguous space to maximize locality; (2) large arrays (whose sizes are $\geq$32KB) are allocated on empty pages: allocating them on nonempty pages may cause them to span multiple pages, therefore increasing access costs. Otherwise, we request memory from the first page on the list that has enough space for the record. To allow fast allocation for multithreading, we create a distinct *page manager* (that maintains separate size classes and pages) per thread so that different threads concurrently allocate data records on their thread-local pages. Having distinct page managers also eliminates the potential fragmentation issue associated with size classes.

The data path is *iteration based*. We define an iteration to be a repeatedly executed block of code such that the lifetimes of data objects created in different executions of this block are completely disjoint. In a typical data-intensive program, a dataset is often partitioned before being processed; different iterations of a data manipulation algorithm (e.g., sorting, hashing, or other computations) then process distinct partitions of the dataset. Hence, pages requested in one iteration of P' are released all at once when the iteration ends. Although different data processing frameworks have different ways of implementing the iteration logic, there often exists a clear mark between different iterations, e.g., a call to start to begin an iteration and a call to flush to end it.

We rely on a user-provided pair of iteration_start and iteration_end calls to manage our pages. Upon a call to iteration_start that signals the beginning of an iteration, we create a page manager that will perform page allocation and memory allocation as discussed above. All pages are recycled immediately upon a call to iteration_end. While this may sound nontrivial, our experience with a variety of applications shows that iterations are often very well defined and program points to place these calls can be easily found even by novices without much understanding of the program

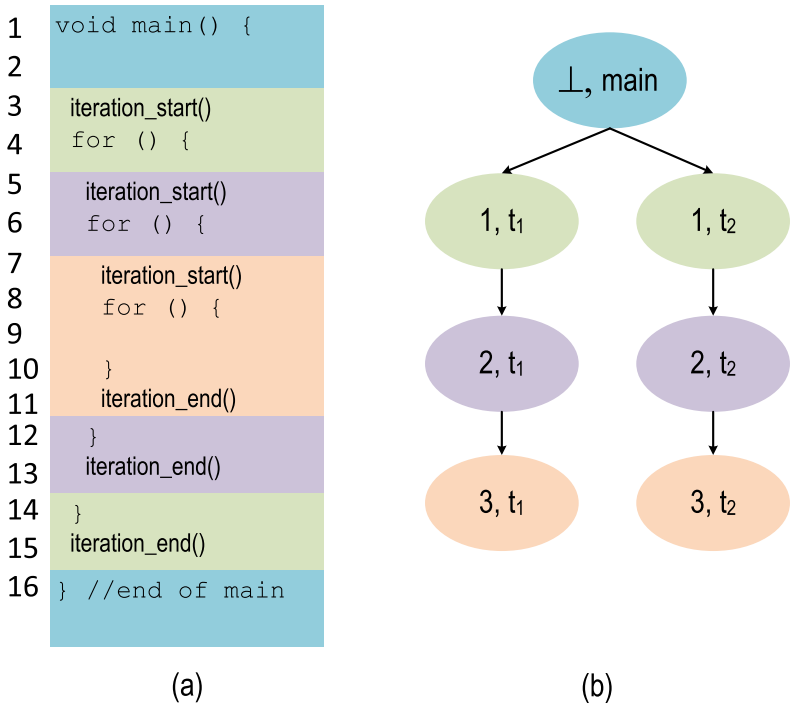

Fig. 9. A program (a) and its corresponding page manager tree after line 7 is executed (b), assuming two threads t_1 and t_2 are executing.

logic. For example, in GraphChi (Kyrola et al. 2012), a single-machine graph processing framework, iteration_start and iteration_end are the callbacks explicitly defined by the framework. Although we had had zero knowledge about this framework, it took us only a few minutes to find these events. Note that iteration-based memory management is used only to deallocate data records and it is unsafe to use it to manage control objects. Those objects can cross multiple iterations and, hence, we leave them to the GC for memory reclamation.

In order to quickly recycle memory, we allow the developer to register *nested iterations*. If a user-specified iteration_start occurs in the middle of an already-running iteration, a *subiteration* starts; we create a new page manager, make it a child of the page manager for the current iteration, and start using it to allocate memory. The page manager for a thread is made a child of the manager for the iteration where the thread is created. Hence, each page manager has a pair $\langle iterationID, thread \rangle$ identifier and they form a tree structure at runtime. When a (sub)iteration finishes, we simply find its page manager m and recursively release pages controlled by the managers in the subtree rooted at m. Recycling can be done efficiently by creating a thread for each page manager and letting them reclaim memory concurrently.

Since each thread t is assigned a page manager upon its creation, the pair identifier for its default page manager is $\langle \perp, t \rangle$; $\perp$ represents the fact that no iteration has started yet. Data records that need to be created before any iteration starts (e.g., usually large arrays) are allocated by this default page manager and will not be deallocated until thread t terminates.

To illustrate, Figures 9(a) and 9(b) show, respectively, a simple program and the corresponding page managers created in Facade. In the beginning, no iteration has started yet; all allocation

requests are handled by the default page manager $\langle \perp, main \rangle$. Assuming there are two worker threads t_1 and t_2 executing the program, upon the call to iteration_start in line 3, two page managers $\langle 1, t_1 \rangle$ and $\langle 1, t_2 \rangle$ are created and placed as the children of the default manager to handle allocations in thread t_1 and t_2, respectively, for iteration #1. The call to iteration_start in line 5 signals the start of subiteration #2. Consequently, the page managers $\langle 2, t_1 \rangle$ and $\langle 2, t_2 \rangle$ are created as the children of the managers $\langle 1, t_1 \rangle$ and $\langle 1, t_2 \rangle$, respectively. Facade allows arbitrarily nested iterations. When the execution encounters another iteration_start in line 7, Facade creates two page managers $\langle 3, t_1 \rangle$ and $\langle 3, t_2 \rangle$. The calls to iteration_end in lines 11, 13, and 15 trigger page reclamation in the reverse order of page allocation.

4.8 Correctness and Profitability Arguments

It is easy to see the correctness of the class transformation and the generation of data-accessing instructions as shown in Table 3, because the data layout in a native memory page is the same as in a heap object. This subsection focuses on the following:

Facade Usage Correctness. If a page reference is assigned to a facade that has not released another page reference, a problem would result. However, it is guaranteed that this situation will not occur because (1) a thread will never use a facade from another thread's pool and (2) for any index i in a facade pool p, the page reference field of $p[i]$ will never be written twice without a read of the field in between. The read will load the page reference onto the thread's stack and use it for the subsequent data accesses.

Memory Management Correctness. Iteration-based memory management converts dynamic memory reclamation to static reclamation, and it is very difficult to make it correct for general objects in a scalable way. Facade performs iteration-based deallocation only for data items in native memory. Data items allocated in one iteration represent the data partition processed in the iteration. These items will often not be needed when a different data partition is processed (in a different iteration). Since practicality is our central design goal, we choose not to perform any conservative static analysis (e.g., escape analysis (Choi et al. 1999)) to verify whether data items can escape. A real-world data-intensive application often relies heavily on (third-party) libraries and the heavy use of interfaces in the program code makes it extremely difficult for any interprocedural analysis to produce precise results. Instead, we simply assume that instances of the user-specified data classes can never escape the iteration boundary. However, if a data object escapes an iteration through a control object, the synthesized conversion function will convert the paged record to an object.

Transformation Safety. Our transformation is mostly *local* and does not change the control flow of the original program. This feature makes it easier for Facade to preserve semantics in the presence of complicated language constructs such as exception handling (i.e., exception-throwing logic is not changed by Facade: checked exception will be thrown in regular ways, while unchecked exception still crashes the program). In other words, our transformation is safe. The memory management correctness thus relies solely on the user's correct specification of data classes. Section 5 reports our own experiences with finding data classes for real-world programs that we have never studied.

Transformation Profitability. Not all Java programs are suitable for Facade transformations. As mentioned in Section 1, it is profitable using Facade when (1) the number of runtime objects of a class is exceptionally large and (2) the lifetimes of these objects follow epochal patterns; that is, they align with computational iterations in which they are created. While Facade works well

for data-intensive systems, they may not be as effective to transform regular Java applications for performance improvement.

4.9 Modeling of Important Java Library Classes

All the primitive-type wrappers in Java (e.g., `Integer`, `Float`, etc.) are considered as data classes. We manually implement the `StringFacade` class to support all string-related operations instead of generating it automatically from the Java `String` class. This is first because we want to inline characters, rather than creating a two-layer structure (as done in the Java String implementation). In addition, the Java String implementation has many references and dependencies, which would make Facade transform classes we believe to be in the control path (e.g., classes representing formats and calendars). Records for all primitive-type wrappers and strings are inlined.

Commonly used native methods such as `System.arraycopy` and `Unsafe.compareAndSwap` are manually modeled to operate on Facade-page-based data because it is not possible to transform native library methods. In generated programs, the original native methods are replaced with our own version. In addition, we provide implementations of the methods `hashCode`, `equals`, and `clone` in class `Facade`. For example, `equals` is implemented by using page reference as a substitution of object identity, while `hashCode` is implemented by computing a hash code based on the page reference contained in the facade.

A real-world program makes heavy use of collection classes in the `java.util` framework. Their implementations in the Oracle JDK often reference many other classes; for example, more than 100 classes all over the JDK can be transitively reached from class `HashMap`, and many of these classes have nothing to do with data processing. Instead of transforming all these classes, we create our own (page-based) implementations for all important collection classes, including various types of maps, sets, and lists. For example, since `HashMap` is in the data path, all its superclasses except `java.lang.Object` are included in the data path and they are all transformed manually into facade classes.

It took us about 2 weeks of programming to develop our own versions of library classes. The major part of the development was easy—we simply followed the logic from the original collection classes in the JDK. However, one challenge is how to break dependencies to classes that are in the control path. If a data class has a field of a noncollection class type, we carefully inspected the class to understand whether removing the field would cause any semantic inconsistencies. If it would, we include it in the data path and transform it into a facade class; otherwise, we remove the field and replace the code that uses the field with our own version that implements the same logic in different ways. Rigorous testing was performed to ensure that our class collections have the same semantics as their JDK counterparts.

4.10 Implementation and Optimizations

We have implemented Facade based on the Soot Java compiler infrastructure and made it publicly available on BitBucket (https://bitbucket.org/khanhtn1/facade). Facade consists of approximately 40,000 lines of Java code. Our transformation works on Jimple, an SSA-based three-address IR; the transformation occurs after all traditional dataflow optimizations are performed to eliminate redundancies in P, such as dead code and unnecessary loads. We develop a few additional optimizations that target common operations we have observed in data-intensive applications.

— Array inlining: for a data array whose element size can be statically determined, we inline its elements to improve data locality and reduce pointer dereference costs. Upon the creation of an array, we allocate $l \times s$ bytes of memory, where l and s are the array length and its element size. We perform a static analysis that identifies objects that *must* be written into

the array and remove their allocation sites. Their indices in the array will be used as their page references. Although array inlining may lead to wasted space for general programs, it is very effective for data-intensive applications in which large arrays are often created and their elements are often "owned" by the arrays (i.e., accessed only through the arrays).

—Special handling of oversized objects: keeping large arrays that are no longer used in a page can lead to excessive memory usage. To solve the problem, we create a special allocate function in the page manager that allows us to allocate pages bigger than 32KB for large arrays. Each large array will take one single page, instead of spanning multiple (32KB) pages. These pages are located in an "oversize" class and can be freed manually (after they are no longer used) without waiting until the end of an iteration. Right now this optimization is enabled only in the implementation of the resize function in various collection classes, and it has been shown to be effective in improving memory efficiency of applications that make heavy use of maps and lists. We are in the process of developing an automated analysis that can detect such early deallocation opportunities for large objects in the generated code.

—Static resolution of virtual calls: we use Spark (Lhoták and Hendren 2003), a simple and inexpensive context-insensitive points-to analysis to statically resolve virtual calls. For the resolved calls, their receiver facades can be statically determined.

5 EVALUATION

We selected three different data processing frameworks and used Facade to transform their data paths. Our evaluation on seven common data analytical applications on both single machines and clusters shows that, even for already well-optimized systems, Facade can still improve their performance and scalability considerably.

5.1 GraphChi

Transformation. GraphChi (Kyrola et al. 2012) is a high-performance graph analytical framework that has been well optimized for efficient processing of large graphs on a single machine. Since we had not had any previous experience with GraphChi, we started out by profiling instances of data classes to understand the control and data path of the system. The profiling results show that `ChiVertex`, `ChiPointer`, and `VertexDegree` are the only three classes whose instances grow proportionally with the input data size. From these three classes, Facade detected 18 *boundary classes* that interact with data classes but do not have many instances themselves. Boundary classes have both data and nondata fields. We allow the user to annotate data fields with Java pragmas so that Facade can transform these classes and only page-allocate their data fields.

With about 40 person-hours of work (to understand data classes, profile their numbers, and annotate boundary classes for a system we had never studied before), Facade transformed all of these classes (7,753 Jimple instructions) in 10.3 seconds, at a speed of 752.7 instructions per second. Iterations and intervals are explicitly defined in GraphChi—it took us only a few minutes to add callbacks to define iterations and subiterations.

Test Setup. We tested the generated code and compared its performance with that of the original GraphChi code. The experiments were performed on a four-core server with four Intel Xeon E5620 (2.40GHz) processors and 50GB of RAM, running Linux 2.6.32. We experimented extensively with two representative applications, page rank (PR) and connected components (CC). The graph used was the *twitter-2010* graph (Kwak et al. 2010), consisting of 42M vertices and 1.5B edges.

We used the Java HotSpot(TM) 64-bit Server VM (build 20.2-b06, mixed mode) to run all experiments. The state-of-the-art parallel generational garbage collector was used for memory reclamation. This GC combines parallel Scavenge (i.e., copying) for the young generation and parallel

Table 4. Performance Summary of GraphChi on *twitter-2010*

App	ET(s)	ET'(s)	%ET	%UT	%LT	%GT	PM(MB)	PM'(MB)	%PM
PR-8g	1,540.8	1,180.7	23.4%	23.7%	25.7%	84.2%	8,469.8	6,135.4	27.6%
PR-6g	1,561.2	1,146.2	26.6%	25.2%	30.5%	81.7%	6,566.5	6,152.6	6.3%
PR-4g	1,663.7	1,159.2	30.3%	34.5%	28.5%	86.7%	4,448.7	6,127.4	−37.7%
CC-8g	2,338.1	2,207.8	5.6%	6.4%	8.5%	77.0%	8,398.3	6,051.6	27.9%
CC-6g	2,245.8	2,143.4	4.6%	5.4%	10.0%	72.5%	6,557.8	6,045.3	7.8%
CC-4g	2,288.5	2,120.9	7.3%	9.4%	11.7%	74.4%	4,427.4	6,057.0	−36.8%

Reported are execution time of original run (ET) and facade run (ET') in seconds, the reduction of total execution times (*%ET*), engine update times (*%UT*), data load times (*%LT*), garbage collection times (*%GT*), and peak memory consumptions (*PM* and *PM'*) in megabytes under three different memory budgets (e.g., 8GB, 6GB, and 4GB); peak memory is computed by calculating the maximum from a set of samples of JVM memory consumptions collected periodically from pmap; graph preprocessing time is not included.

Mark-Sweep-Compact for the old generation to quickly reclaim unreachable objects. GraphChi uses a parallel sliding-window algorithm that partitions data into shards. Since the number of shards has only little impact on performance (as reported in Figure 8(c) in Kyrola et al. (2012) and also confirmed in our experiments), we fixed the number of shards to 20 in our experiments.

Performance. GraphChi determines the amount of data to load and process (i.e., memory budget) in each iteration dynamically based on the maximum heap size. This is a very effective approach to reduce memory pressure and has been shown to be much more efficient than loading a fixed amount data per iteration. We ran P and P' with the same maximal heap size so that the same amount of data is loaded in each iteration (i.e., guaranteeing the same I/O time in both executions). Note that P' actually does not need a large heap because of the use of native memory. We tried various heap sizes and found that the smallest heap size for running P' was 2.5GB, while P could not execute when the heap was smaller than 4GB.

Table 4 shows the detailed performance comparisons. Note that our performance numbers may look different from those reported in Kyrola et al. (2012), because their experiments used SSD and a C++ version of GraphChi. In Table 4, P' outperforms P for all configurations. The performance improvements FACADE has achieved for PR and CC over *twitter-2010* on average are, respectively, **26.8%** and **5.8%**; larger gains were seen when we experimented with smaller graphs (discussed shortly). Not only does the generated program P' have much less GC time (i.e., an average 5.1× reduction), but also data load and engine update time have been reduced primarily due to inlining and direct memory accesses.

For PR, the number of objects for its data classes has been reduced from $14,257,280,923$[4] to 1,363, of which 1,000 is the number of memory page objects and 363 is the number of total facade objects. Other than the main thread, GraphChi uses two thread pools, each containing 16 threads, and each thread has a pool of 11 facades. Hence, the total number of data objects equals $1{,}000 + 11 * (16 * 2 + 1) = 363$, leading to dramatically decreased GC effort. The cost of page creation and recycling is negligible: the time it took to create and recycle pages was less than 5 seconds during the execution of PR' with five major iterations and 159 sub-iterations.

For P, its memory consumption is bounded by the maximum heap size, while the memory usage for P' is quite stable across different memory budget configurations. This is because our heap contains only objects in the control path, whose numbers are very small; the off-heap data storage

[4]Obtained from our profiling that counts the number of runtime objects of data classes such as `ChiVertex`, `ChiPointer`, and `VertexDegree`.

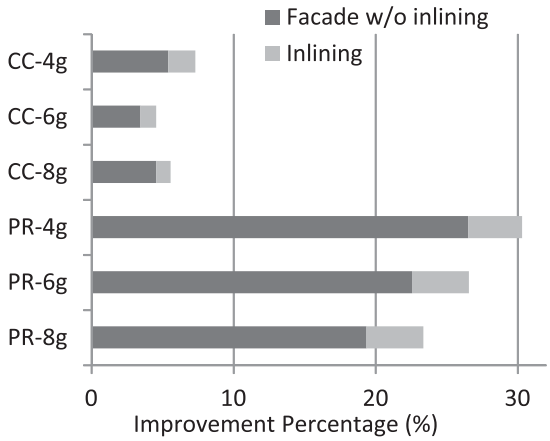

Fig. 10. Performance improvements with and without data record inlining.

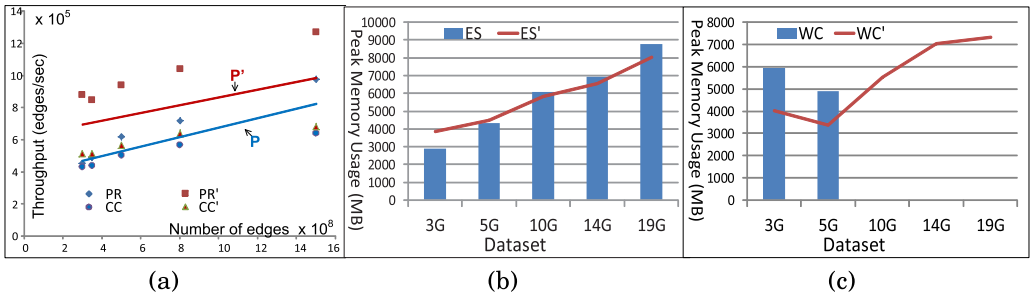

Fig. 11. (a) **Computational throughput of GraphChi** on various graphs (x-axis is the number of edges); each trend line is a least-squares fit to the average throughput of a program. (b) **Memory usage of external sort (ES) on Hyracks**. (c) **Memory usage of word count (WC) on Hyracks**.

is not subject to the GC and is only determined by the amount of data processed. For both P and P', their running time does not vary much as the memory budget changes. This is primarily due to the adaptive data loading algorithm used by GraphChi. For systems that do not have this design, a significant time increase and the GC efforts can often be seen when the heap becomes smaller, and thus, further performance improvement can be expected from FACADE's optimization. Note that under a 4GB heap, P consumes less memory than P'. This is because the GC reclaims objects immediately after they become unreachable, while FACADE allows dead data records to accumulate until the end of a (sub)iteration (i.e., trades off space for time).

To have a better understanding of how much inlining contributes to the performance improvements, we have compared the performance of the transformed GraphChi with and without inlining enabled. The results are shown in Figure 10. Inlining contributes to 4% and 1.5% of the improvements for PR and CC, respectively. The majority of the improvement comes from the reduction in GC efforts.

Scalability. We measured scalability by computing *throughput*, the number of edges processed in a second. From the *twitter-2010* graph, we generated four smaller graphs with different sizes. We fed these graphs to PR and CC to obtain the scalability trends, which are shown in Figure 11(a). An 8GB heap was used to run P and P'. While 8GB appears to be a large heap for these relatively smaller

Table 5. GraphChi Performance Comparisons on Different Datasets

Input Size			*Page Rank (PR)*				*Connected Component (CC)*			
$\|S\|$	$\|V\|$	$\|E\|$	T(s)	T'(s)	%ET	%TR	T(s)	T'(s)	%ET	%TR
5.1GB	33.9M	293.7M	661.3	**341.1**	48.4%	93.9%	696.3	**582.0**	16.4%	19.6%
6.4GB	35.6M	367.1M	711.4	**414.6**	41.7%	71.6%	796.8	**680.1**	14.6%	17.2%
8.5GB	37.2M	489.5M	808.5	**533.2**	34.1%	51.6%	991.6	**880.1**	11.3%	12.7%
12.8GB	38.7M	734.2M	1,113.0	**770.9**	30.7%	44.4%	1,411.0	**1,250.8**	11.4%	12.8%
26.2GB	41.0M	1.5B	1,540.8	**1,180.7**	23.4%	30.5%	2,338.1	**2,207.8**	5.6%	5.9%

Reported are the total execution times of the original program (T) and its FACADE-version (T') measured in seconds; %ET reports FACADE's reduction in execution time and %TR reports the throughput improvement.

graphs, our goal is to demonstrate, even with large memory resources and thus less GC effort, that FACADE can still improve the application performance substantially. For both versions, they scale very well with the increase of the data size. The generated program P' has higher throughput than P for all the graphs. In fact, for some of the smaller graphs, the performance difference, shown in Table 5, between P and P' is even larger than what is reported in Table 4. For example, on a graph with 300M edges, PR' and CC' are **48%** and **17%** faster than PR and CC, respectively. Since the cost of single-PC-based graph processing is dominated by disk accesses, the fraction of the I/O cost is smaller when processing smaller graphs and thus the effectiveness of the FACADE optimizations becomes more obvious. This explains the reason that the performance difference is larger on smaller graphs.

5.2 Hyracks

Hyracks (UCI 2014; Borkar et al. 2011) is a data-parallel platform that runs data-intensive jobs on a cluster of shared-nothing machines. It has been optimized manually to allow only byte buffers to store data and has been shown to have better scalability than object-based frameworks such as Hadoop. However, user functions can still (and mostly likely will) use object-based data structures for data manipulation.

After FACADE transformed a significant portion of the high-level data manipulation functions in Hyracks, we evaluated performance and scalability with two commonly used applications: word count (WC) and external sort (ES). It took us 10 person-hours to find and annotate these user-defined operators; FACADE transformed eight classes in 15 seconds, resulting in a speed of 990 instructions per second.

Other than the eight classes automatically transformed, the application code uses 15 collection classes from the Java library, which we already modeled manually. Among these eight classes, two are user-defined functions (one for WC and the other for ES), and the remaining six classes are Hyracks internal classes specifying properties and operator states. Three interaction points were detected: one for passing data from the control path into the data path and the other two passing processed data back into the control path (both via `ByteBuffer`). Data conversion functions were synthesized and calls to them were added at these points. Iterations are easy to identify: calls to *iteration_start* and *iteration_end* are placed at the beginning and the end of each Hyracks operator (i.e., one computation cycle), respectively.

Note that these six Hyracks internal classes do not include those that perform built-in data manipulation functions such as join and filter, because our search started from the user-defined application code and we were not familiar enough with the Hyracks framework to identify all built-in data processing functions. In general, whether a class is transformed or not relies on whether it is in the user-specified list, and Facade can transform different parts of the data path individually.

Table 6. Hyracks Performance Comparisons on Different Datasets

Input Size			*External Sort (ES)*				*Word Count (WC)*			
$\|S\|$	$\|V\|$	$\|E\|$	T(s)	T'(s)	%GC	%PM	T(s)	T'(s)	%GC	%PM
3GB	25.0M	313.8M	95.5	**89.3**	88.3%	−29.3%	**48.9**	57.4	98.2%	31.9%
5GB	33.2M	562.4M	178.2	**167.1**	96.9%	−3.3%	**72.5**	180.8	96.5%	31.5%
10GB	75.6M	1.1B	326.3	**302.5**	90.8%	4.4%	OME(683.1)	**1,887.1**	*N/A*	*N/A*
14GB	143.1M	1.5B	459.0	**426.0**	93.4%	5.4%	OME(943.2)	**2,693.0**	*N/A*	*N/A*
19GB	310.8M	1.8B	806.4	**607.5**	98.9%	8.5%	OME(772.4)	**3,160.2**	*N/A*	*N/A*

Reported are the total execution times of the original program (T) and its FACADE-version (T') measured in seconds; OME(n) means the program runs out of memory in n seconds. %GC and %PM report FACADE's reduction in GC time and peak memory consumption, respectively.

Test Setup. We ran Hyracks on a 10-slave-node (c3.2x large) Amazon EC2 cluster. Each machine has two quad-core Intel Xeon E5-2680 v2 processors (2.80GHz) and 15G RAM, running Linux 3.10.35, with enhanced networking performance. The same JVM and GC were used in this experiment. We converted a subset of Yahoo!'s publicly available AltaVista Web Page Hyperlink Connectivity Graph dataset (Yahoo 2014) into a set of plain text files as input data. The dataset was partitioned among the slaves in a round-robin manner. The two applications were executed as follows: we created a total of 80 concurrent workers across the cluster, each of which reads a local partition of the data. Both WC and ES have a MapReduce-style computation model: each worker computes a local result from its own data partition and writes the result into the Hadoop Distributed File System (HDFS) running on the cluster; after hash-based shuffling, a reduce phase is then started to compute the final results.

Unlike GraphChi that adaptively loads data into memory, Hyracks loads all data upfront before the update starts. We ran both P and P' with an 8GB heap. When the heap is exhausted in P, the JVM terminates immediately with out-of-memory errors. Naïvely comparing scalability would create unfairness for P, because P' uses a lot of native memory. To enable a fair comparison, we disallowed the total memory consumption of P' (including both heap and native space) to go beyond 8GB. In other words, an execution of P' that consumes more than 8GB memory is considered as an "out-of-memory" failure.

Performance and Scalability. Table 6 shows a detailed running time comparison between P and P' on datasets of different sizes (which are all generated from the Yahoo! web graph data). P' outperforms P for all the inputs except the two smallest (3GB and 5GB) ones for WC. For these datasets, each machine processes a very small data partition (i.e., 300MB and 500MB). The GC effort for both P and P' is very small, and hence, the extra effort of pool accesses and page-based memory management performed in P' slows down the execution. However, as the size of the dataset increases, this effort can be easily offset from the large savings of GC costs. We can also observe that P' scales to much larger datasets than P. For example, WC fails in 683.1 seconds when processing 10GB, while WC' successfully finishes in 3,160.2 seconds for the 19GB dataset. Although both ES and ES' can scale to 19GB, ES' is about **24.7%** faster than ES.

Figures 11(b) and 11(c) show the memory usage comparisons for ES and WC, respectively. Each bar represents the memory consumption (in GB) of the original program P, while a red line connects the memory consumptions of P' for different datasets. If P runs out of memory, its memory consumption is not shown. It is clear to see that P' has a smaller memory footprint than P in almost all the cases. In addition, P' has achieved an overall 25× reduction in the GC time, with a maximum 88× (from 346.2 seconds to 3.9 seconds).

Table 7. GPS Performance Comparisons on Different Datasets

	Input Size			*Page Rank (PR)*		*K-Means (KM)*		*Random Walk (RW)*	
	$\|S\|$	$\|V\|$	$\|E\|$	%ET	%GC	%ET	%GC	%ET	%GC
LJ	0.5GB	4.8M	68.0M	−0.3%	15.5%	0.1%	20.5%	0.2%	39.8%
LJ-A	2.5GB	24.0M	340.0M	4.7%	12.8%	2.9%	6.0%	9.4%	34.0%
LJ-B	5.0GB	48.0M	680.0M	1.6%	11.6%	6.4%	10.0%	7.2%	27.8%
LJ-C	7.5GB	72.0M	1.0B	2.4%	12.2%	8.2%	4.8%	6.5%	30.1%
LJ-D	10.0GB	96.0M	1.5B	5.5%	18.3%	7.7%	4.4%	7.6%	32.5%
LJ-E	12.5GB	120.0M	1.7B	17.3%	30.8%	13.5%	23.1%	10.9%	32.1%
TW	26.2GB	41.0M	1.5B	2.3%	14.4%	3.5%	7.19%	4.9%	19.9%

Reported are the FACADE's reduction in execution time (%ET) and GC time (%GC); LJ-A. . .LJ-E are different synthetic supergraphs of the LiveJournal (LJ) Graph; TW is the *twitter-2010* graph.

5.3 GPS

GPS (Salihoglu and Widom 2013) is a distributed graph processing system developed for scalable processing of large graphs. We profiled the execution and identified a total number of four (vertex- and graph-related) data classes whose instances grow proportionally with the data size. Starting from these classes, FACADE further detected 44 data classes and 13 boundary classes. After an approximate 30-person-hour effort of understanding these classes, FACADE transformed a total number of 61 classes (including 10,691 Jimple instructions) in 9.7 seconds, yielding a 1,102-instructions-per-second compilation speed.

We used three applications—page rank, k-means, and random walk—to evaluate performance. The same (Amazon EC2) cluster environment was used to run the experiments. We created a total of 20 workers, each with a 4GB heap.

GPS uses the distributed message-passing model of Pregel (Malewicz et al. 2010). The behavior of each vertex is encapsulated in a function `compute`, which is executed exactly once in each superstep. GPS is overall less scalable than GraphChi and Hyracks due to its object-array-based representation of an input graph. GPS expects vertices to be labeled contiguously starting from 0, and therefore, vertices can be efficiently stored in an array. While the goal of this design is to improve performance by avoiding using Java data structures (e.g., ArrayList), it leads to memory inefficiencies in many cases. This is because each node processes an arbitrary set of vertices and thus the array becomes a sparse structure with a lot space wasted. To solve the problem, we replaced the array with an ArrayList before performing transformation—since Facade will allocate the ArrayList in the native memory, this replacement saves a lot of space without incurring extra GC overhead.

The set of input graphs we used includes the *twitter-2010* graph, the *LiveJournal* graph, and five synthetic supergraphs of *LiveJournal* (e.g., the largest supergraph has 120M vertices and 1.7B edges). Table 7 shows detailed performance comparisons. Compared to the original implementation P, the generated version P' has achieved a 3% to 15.4% running time reduction, a 10% to 39.8% GC time reduction, and an up to 14.4% space reduction. P and P' have about the same running time on the smallest graph (with 4.8M vertices and 68M edges). However, for all the other graphs in the input set, clear performance improvements can be observed on P', which is especially significant on the largest graph (`LJ-E`). The reason FACADE can improve GPS' performance is that GPS still allows the creation of objects to represent vertex values (e.g., `DoubleWritable`, `TwoIntWritable`, `LongWritable`, etc.). These (small) objects in turn are stored in a huge array. FACADE automatically inlines all of them into the array, therefore improving the performance of GPS.

Unlike GraphChi where the majority of the improvement comes from the reduction in GC efforts, in GPS, data record inlining contributes the most to the improvement. Without inlining, FACADE-generated programs run slightly faster (1%–2%) than the original. This is because the design of GPS is very similar in spirit to what FACADE intends to achieve in many ways. First, it has extensive use of raw (i.e., primitive) arrays to store data such as adjacent list or message content. Second, GPS reduces the memory cost of allocating many Java objects by using canonical objects. Instead of storing the value and the adjacency list of each vertex inside a separate `Vertex` object and calling `compute` on each object as in Giraph, GPS workers use a single canonical `Vertex` object to perform program logic (i.e., invoke `compute`) with vertex values and adjacency list stored in separate data structures. GPS also uses a single canonical `Message` object for message passing. Incoming messages are stored as raw bytes in the message queues, and a message is deserialized into the canonical `Message` object only when the canonical `Vertex` object iterates over it.

5.4 Discussion

Admittedly, to use FACADE, a considerable amount of user effort is needed to understand the program and provide a correct list of data classes as well as iteration markers (i.e., *iteration_start* and *iteration_end* calls). While the markers are often well defined in a Big Data platform, the user should understand such semantic information by profiling the application (as we did in our experiments) to distinguish data classes (e.g., classes that have a large number of runtime objects) from control classes. The FACADE compiler will assist users in identifying the list of data classes by throwing compilation errors when assumptions (Section 4.1) are violated. Users are then to refactor the violating classes to make the application FACADE transformable. However, although we had never studied any of these frameworks before this work, we found that the majority of the manual effort was spent on profiling each system to understand the data path and setting up the execution environments (e.g., on average, it took us a day's worth of work for each framework). Once we identified an initial set of data classes, the effort to specify iterations and annotate boundary classes was almost negligible. It would have taken much less time had the developers of these frameworks used FACADE themselves.

6 RELATED WORK

"Big Data" Optimizations. While there exists a large body of work on optimizing data-intensive applications, these existing efforts focus on domain-specific optimizations, including, for example, data pipeline optimizations (Isard et al. 2007; Agrawal et al. 2008; Chaiken et al. 2008; Olston et al. 2008a; Yu et al. 2008; Zhou et al. 2010; Chambers et al. 2010; Borkar et al. 2011; Guo et al. 2012; Kyrola et al. 2012), query optimizations (Olston et al. 2008b; Condie et al. 2010; Dittrich et al. 2010; Nykiel et al. 2010; Lee et al. 2011; Murray et al. 2011), and Map-Reduce-related optimizations (Pike et al. 2005; Yang et al. 2007; Dean and Ghemawat 2008; Thusoo et al. 2009; Afrati and Ullman 2010; Thusoo et al. 2010; Liu et al. 2012).

Cascading (Cascading 2015) is a Java library built on top of Hadoop. It provides abstractions for developers to explicitly construct a dataflow graph to ease the challenge of programming data-parallel tasks. Similarly to Cascading, FlumeJava (Chambers et al. 2010) is another Java library that provides a set of immutable parallel collections. These collections present a uniform abstraction over different data representations and execution strategies for MapReduce. StarFish (Herodotou et al. 2011) is a self-tuning framework for Hadoop that provides multiple levels of tuning support. At the heart of the framework is a Just-in-Time optimizer that profiles Hadoop jobs and adaptively adjusts various framework parameters and resource allocation.

Despite the commendable accomplishments of these optimizations, data processing performance is fundamentally limited by memory inefficiencies inherent to the underlying programming

systems. Zing (Azul 2014) is a commercial system developed by Azul that can lower the latency for Java-based data-intensive applications by making larger in-memory indexes. Apache Flink and Apache Spark support storing data in memory in a serialized form and include many "binary" versions of common data structures like hash tables. This article attempts to solve the memory problem by limiting the number of objects used to represent data records *automatically*, an approach that is orthogonal to, and will provide benefit for, these existing optimization techniques.

There are a few other active projects that were developed in parallel with or after Facade. For example, Tungsten (tun 2015) is an ongoing effort under the Spark umbrella that aims to explicitly manage memory and eliminate the overhead of the JVM object model and GC. Broom (Gog et al. 2015) is a memory management technique that provides similar optimizations to C# programs. Holly (Maas et al. 2016) attempts to provide global GC coordination to reduce the latency of managed data-intensive systems. Our own work, ITask (Fang et al. 2015), aims to reduce memory pressure by making data-parallel tasks interruptible. These efforts demonstrate the community's realization of the importance of efficient memory management for data-intensive applications and will hopefully lead to the design of Big-Data-friendly runtime systems in the future.

Software Bloat Analysis. Software bloat analysis (Mitchell et al. 2006; Mitchell and Sevitsky 2007; Xu and Rountev 2008; Shankar et al. 2008; Xu et al. 2009; Mitchell et al. 2009; Xu and Rountev 2010; Xu et al. 2010a; Altman et al. 2010; Mitchell et al. 2010; Xu et al. 2010b; Xu et al. 2012; Xu 2012) attempts to find, remove, and prevent performance problems due to inefficiencies in the code execution and the use of memory. Prior work (Mitchell et al. 2006; Mitchell and Sevitsky 2007) proposes metrics to provide a performance assessment of the 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 measuring 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. (2008) 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 (2008), 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. (2009) detects memory bloat by profiling copy activities, and their later work (Xu et al. 2010a) looks for high-cost/low-benefit data structures to detect execution bloat. Our prior work (Bu et al. 2013) analyzes bloat under the context of data-intensive applications, and other work (Nguyen et al. 2015) performs effective optimizations to remove bloat.

Region-Based Memory Management. Region-based memory management was first used in the implementations of functional languages (Tofte and Talpin 1994; Aiken et al. 1995) such as Standard ML (Hallenberg et al. 2002), and then was extended to Prolog (Makholm 2000), C (Gay and Aiken 1998, 2001; Grossman et al. 2002; Hicks et al. 2004), and real-time Java (Beebee and Rinard 2001; Kowshik et al. 2002; Boyapati et al. 2003). More recently, some mark-region hybrid methods such as Immix (Blackburn and McKinley 2008) combine tracing GC with regions to improve GC performance for Java. Although our iteration-based memory management is similar in spirit to region-based memory management, the Facade execution model is novel and necessary to reduce objects in Java applications without modifying a commercial JVM. There are many static analyses (such as region types (Beebee and Rinard 2001; Boyapati et al. 2003)) developed to support region-based memory management. Most of these analyses focus on the detection of region-allocatable objects, assuming that (1) a new programming model will be used to allocate them and (2) there already exists a modified runtime system (e.g., a new JVM) that supports region-based allocation. On the contrary, Facade is a nonintrusive technique that compiles the program and allocates

objects based on an existing JVM, without needing developers to write new programs as well as any JVM modification.

Reducing the Number of Objects via Program Analysis. Object inlining (Dolby and Chien 2000; Lhotak and Hendren 2005) is a technique that statically inlines objects in a data structure into its root to reduce the number of pointers and headers. Free-Me (Guyer et al. 2006) adds compiler-inserted frees to a GC-based system. Pool-based allocation proposed by Lattner (2005), Lattner and Adve (2005), and Lattner et al. (2007) uses a context-sensitive pointer analysis to identify objects that belong to a logical data structure and allocate them into the same pool to improve locality. Design patterns (Gamma et al. 1995) such as `Singleton` and `FlyWeight` aim to reuse objects. However, these techniques have limited usefulness—even if we can reuse data objects across iterations, the number of heap objects in each iteration is not reduced and these objects still need to be traversed frequently by the GC.

Shuf et al. (2002) propose a static technique that exploits *prolific types*—types that have large numbers of instances—to enable aggressive optimizations and fast garbage collection. Objects with prolific types are allocated in a prolific region, which is frequently scanned by the GC (analogous to a nursery in a generation collector); objects with nonprolific types are allocated in a regular region, which is less frequently scanned (analogous to an old generation). The insight is that the instances of prolific types are usually temporary and short-lived. Facade is motivated by a completely opposite observation: data classes have great numbers of objects, which are often long-lived; frequently scanning those objects can create prohibitively high GC overhead. Hence, we allocate data records in native memory without creating objects to represent them. Moreover, Facade adopts a new execution model and does not require any profiling.

Object pooling is a well-known technique for reducing the number of objects. For example, Java 7 supports the use of thread pools to save thread instances. Our facade pool differs from traditional object pooling in three important aspects. First, while they have the same goal of reducing objects, they achieve the goal in completely different ways: Facade moves data objects out of the heap to native memory, while object pooling recycles and reuses instances after they are no longer used by the program. Second, the facade pool has a bound; we provide a guarantee that the number of objects in the pool will not exceed the bound. On the contrary, object pooling does not provide any bound guarantee. In fact, it will hurt performance if most of the objects from the pool cannot be reused, because the pool will keep growing and consume a lot of memory. Finally, retrieving/returning facades from/to the pool is automatically done by the compiler, while object pooling depends on the developer's insight—the developer has to know what objects have disjoint lifetimes and write code explicitly to recycle them.

Resource Limits Systems. Starting with mechanisms as simple as the setrlimit system call, limits have long been supported by POSIX-style operating systems. Recent work such as resource containers (Banga et al. 1999) provides a hierarchical mechanism for enforcing limits on resources, especially the CPU. HiStar (Zeldovich et al. 2006) organizes space usage into a hierarchy of containers with quotas. Any object not reachable from the root container is garbage collected. At the programming language level, a lot of work (Hawblitzel and von Eicken 2002; Back and Hsieh 2005) has gone toward resource limits for Java. Facade can be thought of as a special resource limits system that statically bounds object usage for each thread. However, Facade does not bound the general memory usage, which still grows with the size of dataset.

PADS, Value Types, and Rust. Most of the existing efforts for language development focus on providing support for data representation (such as the PADS project (Fisher et al. 2006; Mandelbaum et al. 2007)), rather than improving performance for data processing. Expanded types in Eiffel and

value types in C# are used to declare data with simple structures. Value types can be stack allocated or inlined into heap objects. While using value types to represent data items appears to be a promising idea, its effectiveness is actually rather limited. For example, if data items are stack allocated, they have limited scope and cannot easily flow across multiple functions. On the other hand, always inlining data items into heap objects can significantly increase memory consumption, especially when a data structure grows (e.g., resizing of a hash map) and two copies of the data structure are needed simultaneously.

Moreover, these data items are no longer amenable to iteration-based memory management—they cannot be released until their owner objects are reclaimed, leading to significant memory inefficiencies. Rust (Mozilla 2014) is a systems programming language designed by Mozilla that allows developers to specify what memory gets managed by the GC and managed manually. While Rust may enable future development of scalable "Big Data" systems, the goal of Facade is to transform a large number of existing programs written in Java without requiring developers to rewrite programs.

7 CONCLUSION AND FUTURE WORK

Growing datasets require efficiency on all levels of the processing stack. This article studies the problem of memory bloat caused by excessive object creation in a managed data processing system and proposes a compiler and runtime Facade that achieves high efficiency by performing a semantics-preserving transformation of the data path of a data-intensive program to statically bound the number of heap objects representing data items. Our experimental results demonstrate that the generated programs are more (time and memory) efficient and scalable than their object-based counterparts.

One interesting direction of future work is to adapt Facade to Scala, which powers the entire Spark framework. In fact, all techniques discussed in this article can be conceptually applied to optimize Scala programs as well. One challenge is that there are many compiler-generated objects in Scala that do not exist in Java; leaving all of them in the heap would still cause large GC overhead. We need to modify the Scala compiler and develop techniques to identify those objects for native allocation.

Since Facade can also reduce the memory footprint of a program, the transformed program may likely need fewer machines to process a dataset compared to the original program. Hence, it is interesting to conduct additional experiments to understand the resource reductions that can be achieved by Facade.

REFERENCES

Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a map-reduce environment. In *International Conference on Extending Database Technology (EDBT'10)*. 99–110.

Parag Agrawal, Daniel Kifer, and Christopher Olston. 2008. Scheduling shared scans of large data files. *Proc. VLDB Endow.* 1, 1 (2008), 958–969.

Alexander Aiken, Manuel Fähndrich, and Raph Levien. 1995. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. 174–185.

Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance analysis of idle programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*. 739–753.

Apache 2014a. Apache Flink. Retrieved from http://flink.apache.org/.

Apache 2014b. Giraph: Open-source implementation of Pregel. Retrieved from http://incubator.apache.org/giraph/.

Apache 2014c. Hadoop: Open-source implementation of MapReduce. Retrieved from http://hadoop.apache.org.

Apache 2014d. The Hive Project. Retrieved from http://hive.apache.org/.

Apache 2014e. The Mahout Project. Retrieved from http://mahout.apache.org/.

Azul. 2014. Zing: Java for the real time business. Retrieved from http://www.azulsystems.com/products/zing/whatisit.

Godmar Back and Wilson C. Hsieh. 2005. The Kaffeos Java runtime system. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 27, 4 (2005), 583–630.

Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. 1999. Resource containers: A new facility for resource management in server systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*. 45–58.

William S. Beebee and Martin C. Rinard. 2001. An implementation of scoped memory for real-time Java. In *International Conference on Embedded Software (EMSOFT'01)*. 289–305.

Stephen M. Blackburn and Kathryn S. McKinley. 2008. 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'08)*. 22–32.

B. Blanchet. 1999. Escape analysis for object-oriented languages. applications to Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. 20–34.

Vinayak R. Borkar, Michael J. Carey, Raman Grover, Nicola Onose, and Rares Vernica. 2011. Hyracks: A flexible and extensible foundation for data-intensive computing. In *International Conference on Data Engineering (ICDE'11)*. 1151–1162.

Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. 2003. Ownership types for safe region-based memory management in real-time Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*. 324–337.

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

Cascading. 2015. The Cascading Ecosystem. Retrieved from http://www.cascading.org.

Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.* 1, 2 (2008), 1265–1276.

Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 363–375.

Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. 1–19.

CMU. 2015. Out of memory error in efficient sharded positional indexer. Retrieved from http://www.cs.cmu.edu/~lezhao/TA/2010/HW2/.

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*. 21–21.

Cplusplus. 2015. Why is Java more popular than C++. Retrieved from http://www.cplusplus.com/forum/general/79656/.

DataBricks. 2015. Project Tungsten. Retrieved from https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

Jens Dittrich, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.* 3 (2010), 515–529.

Julian Dolby and Andrew Chien. 2000. An automatic object inlining optimization and its evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. 345–357.

Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2008. 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'08)*. 59–70.

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 *ACM Symposium on Operating Systems Principles (SOSP'15)*. 394–409.

Kathleen Fisher, Yitzhak Mandelbaum, and David Walker. 2006. The next 700 data description languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*. 2–15.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

David Gay and Alex Aiken. 1998. Memory management with explicit regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. 313–323.

David Gay and Alex Aiken. 2001. Language support for regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*. 70–80.

Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A garbage collector for big data on big NUMA machines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 661–673.

Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G. Murray, Steven Hand, and Michael Isard. 2015. Broom: Sweeping out garbage collection from big data systems. In *15th USENIX Workshop on Hot Topics in Operating Systems*.

Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv.* 25, 2 (1993), 73–170.

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-based memory management in cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. 282–293.

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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 121–133.

Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: A static analysis for automatic individual object reclamation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*. 364–375.

Niels Hallenberg, Martin Elsman, and Mads Tofte. 2002. Combining region inference and garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'02)*. 141–152.

Chris Hawblitzel and Thorsten von Eicken. 2002. Luna: A flexible Java protection system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*. 391–403.

Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A self-tuning system for big data analytics. In *Conference on Innovative Data Systems Research (CIDR)*. 261–272.

Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. 2004. Experience with safe manual memory-management in cyclone. In *ACM SIGNPLAN International Symposium on Memory Management (ISMM'04)*. 73–84.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys'07)*. 59–72.

Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. 2002. Ensuring code safety without runtime checks for real-time control systems. In *International Conference on Architecture and Synthesis for Embedded Systems (CASES'02)*. 288–297.

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is twitter, a social network or a news media? In *International World Wide Web Conference (WWW'10)*. 591–600.

Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 31–46.

Chris Lattner. 2005. *Macroscopic Data Structure Analysis and Optimization*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

Chris Lattner and Vikram Adve. 2005. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. 129–142.

Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. 278–289.

Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. 2011. YSmart: Yet another SQL-to-MapReduce translator. In *IEEE International Conference on Distributed Computing Systems (ICDCS'11)*. 25–36.

Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using SPARK. In *International Conference on Compiler Construction (CC'03)*. 153–169.

Ondrej Lhotak and Laurie Hendren. 2005. Run-time evaluation of opportunities for object inlining in Java. *Concurrency Comput. Practice Exper.* 17, 5–6 (2005), 515–537. DOI: http://dx.doi.org/10.1002/cpe.848

Jun Liu, Nishkam Ravi, Srimat Chakradhar, and Mahmut Kandemir. 2012. Panacea: Towards holistic optimization of mapreduce applications. In *International Symposium on Code Generation and Optimization (CGO'12)*. 33–43.

Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. 2015. Trash day: Coordinating garbage collection in distributed systems. In *5th USENIX Workshop on Hot Topics in Operating Systems*.

Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. 2016. Holly: A multi-node language runtime system for coordinating distributed managed language applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*.

Henning Makholm. 2000. A region-based memory manager for prolog. In *ACM SIGNPLAN International Symposium on Memory Management (ISMM'00)*. 25–34.

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. 135–146.

Yitzhak Mandelbaum, Kathleen Fisher, David Walker, Mary F. Fernández, and Artem Gleyzer. 2007. PADS/ML: A functional data description language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*. 77–83.

McGill. 2014. Soot framework. Retrieved from http://www.sable.mcgill.ca/soot/.

Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP'09)*. 77–97.

Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2010. Four trends leading to Java runtime bloat. *IEEE Software* 27, 1 (2010), 56–63.

Nick Mitchell and Gary Sevitsky. 2007. The causes of bloat, the limits of health. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*. 245–260.

Nick Mitchell, Gary Sevitsky, and Harini Srinivasan. 2006. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP'06)*. 429–451.

Mozilla. 2014. The Rust programming language. Retrieved from http://www.rust-lang.org/.

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

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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 675–690.

Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting cacheable data to remove bloat. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'13)*. 268–278.

Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. 2010. MRShare: Sharing across multiple queries in MapReduce. *Proc. VLDB Endow.* 3, 1–2 (2010), 494–505.

Christopher Olston, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. 2008a. Automatic optimization of parallel dataflow programs. In *USENIX USENIX Annual Technical Conference (ATC'08)*. 267–273.

Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008b. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. 1099–1110.

Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.*Technical Report 1999-66. Stanford InfoLab. Retrieved from http://ilpubs.stanford.edu:8090/422/.

Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.* 13, 4 (2005), 277–298.

Quora. 2015. For Big Data, Java or C++. Retrieved from https://www.quora.com/For-big-data-Java-or-C++.

Semih Salihoglu and Jennifer Widom. 2013. GPS: A graph processing system. In *Scientific and Statistical Database Management (SSDBM'13)*. 22:1–22:12.

Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. 2008. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'08)*. 127–142.

Yefim Shuf, Manish Gupta, Rajesh Bordawekar, and Jaswinder Pal Singh. 2002. Exploiting prolific types for memory management and optimizations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. 295–306.

Spark User List. 2014. Help understanding - Not enough space to cache RDD. Retrieved from http://apache-spark-user-list.1001560.n3.nabble.com/Help-understanding-Not-enough-space-to-cache-rdd-td20186.html.

StackExchange. 2015. Choose C++ or Java for applications requiring huge amounts of RAM? Retrieved from http://programmers.stackexchange.com/questions/130108/choose-c-or-java-for-applications-requiring-huge-amounts-of-ram.

StackOverflow. 2015a. Out of memory error due to appending values to StringBuilder. Retrieved from http://stackoverflow.com/questions/12831076/.

StackOverflow. 2015b. Out of memory error due to large spill buffer. Retrieved from http://stackoverflow.com/questions/8464048/.

StackOverflow. 2015c. Out of memory error in a web parser. Retrieved from http://stackoverflow.com/questions/17707883/.

StackOverflow. 2015d. Out of memory error in building inverted index. Retrieved from http://stackoverflow.com/questions/17980491/.

StackOverflow. 2015e. Out of memory error in computing frequencies of attribute values. Retrieved from http://stackoverflow.com/questions/23042829/.

StackOverflow. 2015f. Out of memory error in customer review processing. Retrieved from http://stackoverflow.com/questions/20247185/.

StackOverflow. 2015g. Out of memory error in hash join using DistributedCache. Retrieved from http://stackoverflow.com/questions/15316539/.
StackOverflow. 2015h. Out of memory error in map-side aggregation. Retrieved from http://stackoverflow.com/questions/16684712/.
StackOverflow. 2015i. Out of memory error in matrix multiplication. Retrieved from http://stackoverflow.com/questions/16116022/.
StackOverflow. 2015j. Out of memory error in processing a text file as a record. Retrieved from http://stackoverflow.com/questions/12466527/.
StackOverflow. 2015k. Out of memory error in word cooccurrence matrix stripes builder. Retrieved from http://stackoverflow.com/questions/12831076/.
StackOverflow. 2015l. The performance comparison between in-mapper combiner and regular combiner. Retrieved from http://stackoverflow.com/questions/10925840/.
StackOverflow. 2015m. Reducer hang at the merge step. Retrieved from http://stackoverflow.com/questions/15541900/. (2015).
StackOverflow. 2015n. Spark worker insufficient memory. Retrieved from http://stackoverflow.com/questions/31830834/spark-worker-insufficient-memory.
Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.* 2, 2 (2009), 1626–1629.
Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive - A petabyte scale data warehouse using Hadoop. In *International Conference on Data Engineering (ICDE'10)*. 996–1005.
Mads Tofte and Jean-Pierre Talpin. 1994. 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'94)*. 188–201.
Twitter. 2014. Storm: dstributed and fault-tolerant realtime computation. Retrieved from https://github.com/nathanmarz/storm.
UCI. 2014. Hyracks: A data parallel platform. Retrieved from http://code.google.com/p/hyracks/.
UCI. 2015a. Algebricks. Retrieved from https://code.google.com/p/hyracks/source/browse/#git%2Ffullstack%2Falgebricks.
UCI. 2015b. AsterixDB. Retrieved from https://code.google.com/p/asterixdb/wiki/AsterixAlphaRelease.
UCI. 2015c. Hivesterix. Retrieved from http://hyracks.org/projects/hivesterix/.
UCI. 2015d. Pregelix. Retrieved from http://hyracks.org/projects/pregelix/.
UCI. 2015e. VXQuery. Retrieved from http://incubator.apache.org/vxquery/.
Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java bytecode using the soot framework: Is it feasible? In *International Conference on Compiler Construction (CC'00)*. 18–34.
Guoqing Xu. 2012. Finding reusable data structures. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. 1017–1034.
Guoqing Xu. 2013a. CoCo: Sound and adaptive replacement of Java collections. In *European Conference on Object-Oriented Programming (ECOOP'13)*. 1–26.
Guoqing Xu. 2013b. Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'13)*. 111–130.
Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010a. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 174–186.
Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*. 419–430.
Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Trans. Softw. Eng. Methodol.* 23, 3, Article 23 (June 2014), 50 pages.
Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010b. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *ACM SIGSOFT FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER'10)*. 421–426.
Guoqing Xu and Atanas Rountev. 2008. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*. 151–160.
Guoqing Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. 160–173.

Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming (ECOOP'12)*. 738–763.

Yahoo. 2014. Yahoo! Webscope program. Retrieved from http://webscope.sandbox.yahoo.com/.

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Uncovering performance problems in Java applications with reference propagation profiling. In *International Conference on Software Engineering (ICSE)*. 134–144.

Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, and D. Stott Parker. 2007. Map-reduce-merge: Simplified relational data processing on large clusters. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. 1029–1040.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. 1–14.

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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, 2.

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. 10.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in hiStar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*. 263–278.

Jingren Zhou, Per-Åke Larson, and Ronnie Chaiken. 2010. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *International Conference on Data Engineering (ICDE'10)*. 1060–1071.

Received July 2016; revised October 2017; accepted October 2017