# FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications

Khanh Nguyen Kai Wang Yingyi Bu Lu Fang Jianfei Hu Guoqing Xu

University of California, Irvine

{khanhtn1, wangk7, yingyib, lfang3, jianfeih, guoqingx}@ics.uci.edu

## Abstract

The past decade has witnessed the 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 community resource. While the use of such languages makes programming easier, their automated memory management comes at a cost. When the managed runtime meets Big Data, this cost is significantly magnified and becomes a scalability-prohibiting bottleneck.

This paper presents a novel compiler framework, called FACADE, that can generate highly-efficient data manipulation code by automatically transforming the *data path* of an existing Big Data application. The key treatment is that in the generated code, the number of runtime heap objects created for data types 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 7 common applications on 3 real-world, already well-optimized Big Data frameworks: GraphChi, Hyracks, and GPS. Our experimental results are very positive: the generated programs have (1) achieved a 3%–48% execution time reduction and an up to 88× GC reduction; (2) consumed up to 50% less memory, and (3) scaled to much larger datasets.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Code generation, compilers, memory management, optimization, run-time environments

***General Terms*** Language, Measurements, Performance



*ASPLOS '15*, March 14–18, 2015, Istanbul, Turkey.



http://dx.doi.org/10.1145/2694344.2694345

***Keywords*** Big Data applications, managed languages, memory management, performance optimization

## 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 task. Typical parallel frameworks include, to name a few, FlumeJava [20], Giraph [9], GPS [58], Hive [61], Hadoop [10], Hyracks [16], Spark [74], Storm [63], and Pig [56].

However, all of these Big Data systems are written in managed languages (i.e., Java and Scala), which are known for their simple usage, easy memory management, and large community support. While these languages simplify development effort, their managed runtime has a high cost—often referred to as *runtime bloat* [53, 65, 67–71]—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 paper explores a new direction to scale Big Data systems, that is, how to effectively optimize the managed runtime of a data processing system to improve its performance and scalability on each machine.

### 1.1 Motivation

The managed runtime suffers from two major performance issues: excessive use of pointers and references leading to high space overhead (and thus low memory packing factors) as well as frequent GC runs preventing the main threads from making satisfactory progress. Comprehensive studies across many contemporary Big Data systems [18] confirm that these overheads lead to significantly reduced scalability—e.g., applications crash with `OutOfMemoryError`, although the size of the processed

dataset is much smaller than the heap size—as well as exceedingly high memory management costs—e.g., the GC time accounts for up to 50% of the overall execution time. Despite the many optimizations [6, 7, 16, 19, 21, 23–25, 33, 38, 41, 45, 48, 49, 52, 54–57, 60, 61, 72, 73, 76] from various research communities, poor performance inherent with the managed runtime remains a serious problem that can devaluate these domain-specific optimization techniques.

Switching back to an unmanaged language such as C or C++ appears to be a reasonable choice. However, unmanaged languages are more error-prone; debugging memory problems in an unmanaged language is known to be a painful task, which can be further exacerbated by the many "Big Data" effects, such as distributed execution environment, extremely large numbers of data items, and long running time. Furthermore, since a great number of existing Big Data frameworks were already developed in a managed language (e.g., Java and Scala), it is unrealistic to re-implement them from scratch. The proposed work aims to solve this fundamental problem by developing a systematic optimization technique that can dramatically improve the performance of a wide variety of Java-based Big Data systems while still allowing the developers to enjoy all the benefits of a managed programming language.

### 1.2 Observations

Our first observation is that there exists a clear boundary between the *control path* and the *data path* in a Big Data system. The control path organizes tasks into pipelines and performs optimizations, while the data path represents and manipulates data. Although the data path creates most of the runtime objects, its implementation is rather simple and its code size is often small. For instance, in a typical Big Data application that runs on a shared-nothing cluster, a driver controls the pipeline execution on the master node, while each slave node runs data manipulation algorithms (e.g., hashing, sorting, or grouping) to manipulate data. The driver belongs to the control path that does not touch any data, while data manipulation functions belong to the data path that creates massive number of objects to represent and process data items.

Our second observation 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. It is often harmless, however, to create objects in the control path, because their numbers are very small and independent of the size of the input. Our ultimate goal is, thus, to significantly reduce the object representations of data items so that they are not subject to the regular Java memory management. A straightforward way to achieve the goal is to modify a JVM to create a new (data) heap (in parallel with the original heap) and allocate all data objects in this heap. The GC does not scan the data heap, whose memory can be reclaimed either manually or by techniques such as region-based memory management [14, 29, 30, 32, 35, 37, 50]. While this approach appears promising, its implementation dictates a major re-design of the runtime system in a commercial JVM—a metacircular JVM such as Jikes RVM often does not support very large heaps—which makes it difficult to prototype and deploy in real-world settings.

### 1.3 Contributions

To improve practicality, this paper presents a *non-intrusive* technique, named FACADE, an alternative approach that aims to reduce the cost of the managed runtime by limiting the number of heap objects and references at the compiler level, without needing any JVM modification. FACADE contains a novel compiler framework—as well as its runtime support—that can *statically bound* the number of heap objects representing data items in each thread. This is achieved by breaking a long-held object-oriented programming principle: objects are used both to store data and to provide data manipulation interfaces.

FACADE advocates to separate data storage from data manipulation: data are stored in the off-heap, native memory (i.e., unbounded) 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 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. An iteration-based memory management mechanism is used to reclaim data items from native memory: data records allocated in one iteration are deallocated as a whole at the end of the iteration. The GC only scans the managed heap, which contains a very small number of control objects and facades.

To enforce this model, we develop a compiler that can transform an existing Big Data program into an (almost) object-bounded program: the number of heap objects created for a data type 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 $O(t * n + p)$, where $s$ represents the cardinality of the dataset, $t$ is the number of threads, $n$ is the number of data types, and $p$ is the number of page objects used to store data. Details of these bounds can be found in §3.4.

In practice, the reduction is often in the scale of several orders of magnitude. As an example, for GraphChi [41], a single-machine graph processing system, FACADE has reduced the number of objects created for vertices and edges from $14,257,280,923$ to $1,363$. 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 are significantly reduced; furthermore, the managed heap becomes much smaller, resulting in reduced GC effort. For instance, the execution of the transformed page rank program

in GraphChi with the *twitter-2010* graph [40] is 27% faster, consumes 28% less memory, and has 84% less GC time than the original program.

***Why does* FACADE *operate at the right level?*** 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 [28] through static program analyses [15, 22, 26, 46, 59] to low-level systems support [66]. Despite the commendable efforts of these techniques, none of them are practical enough to improve performance for large-scale Big Data programs: sophisticated interprocedural static analyses (such as escape analysis [22] and object inlining [26]) cannot scale to highly framework-intensive codebases while purely systems-based techniques (such as Resurrector [66]) cannot scale to large heaps with billions of objects.

The design of FACADE crosses the layers of compiler and runtime system, exploiting native memory to represent data objects instead of using static analysis to eliminate them. Practicality is the main reason for this design. On one hand, the design enables our compiler to perform simple local (method[1]-based) code transformation, making it possible for FACADE to scale to a large codebase. On the other hand, the combination of code transformation and the leveraging of the native memory support from a commercial JVM eliminates the need to modify the JVM, enabling FACADE to scale to a very large heap.

The FACADE compiler is implemented in the Soot compiler framework [4, 64] and supports most of the Java 7 features. The user's effort is reasonably small: she only needs to (1) identify iterations, which are often very well-defined in Big Data frameworks, as well as (2) specify the data path by providing a list of Java classes to be transformed. FACADE automatically synthesizes data conversion functions for data types that flow across the boundary and inserts calls to these functions at appropriate program points to convert data formats. We have applied FACADE to 7 commonly-used applications on 3 real-world, already well-optimized Big Data 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. The FACADE Execution Model

This section discusses the FACADE execution model and gives an overview of the proposed transformation technique.

### 2.1 Data Storage Based on Native Memory

We propose to store data records in native 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 native memory, obtained through a JVM's native support.

[1] We use terms "method" and "function" interchangeably.

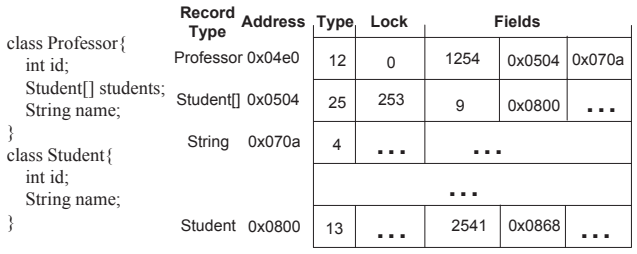


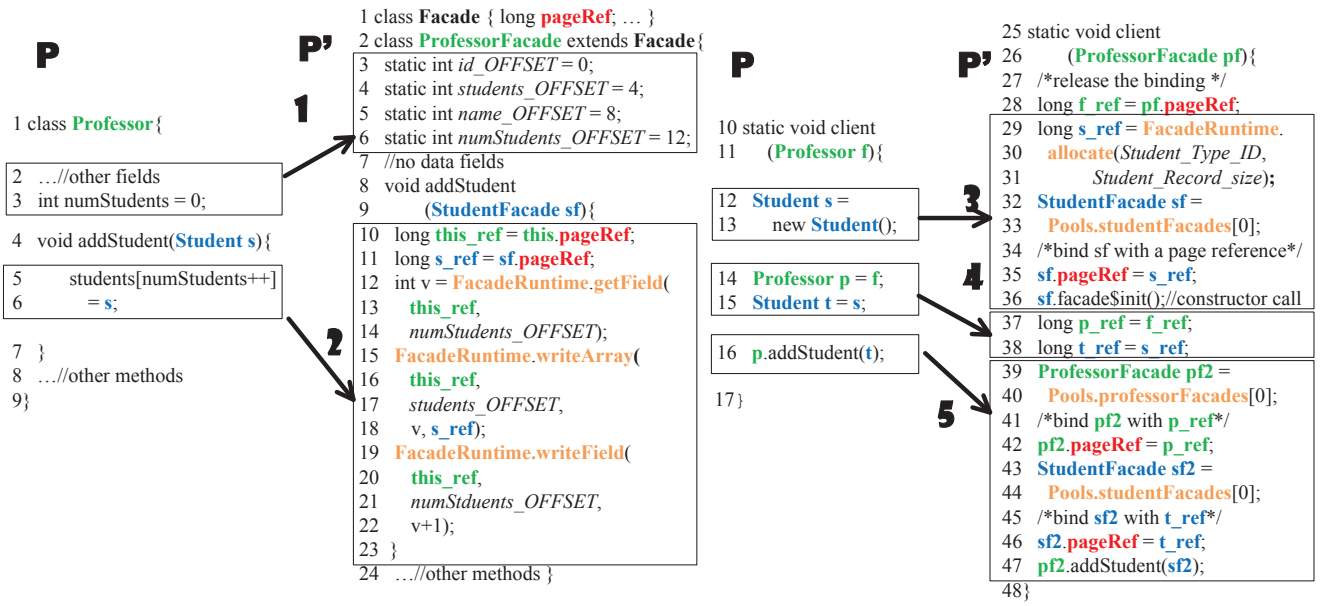

**Figure 1.** A data structure in regular Java and its corresponding data layout in a native page.

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 $O(t * n + p)$) cannot be statically bounded 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 Big Data 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.

Figure 1 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: the number of data classes is often much smaller than $2^{15}$; so is the number of distinct locks needed. Details of the lock implementation and the concurrency support can be found in §3.4.

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 9. The actual data contents (originally stored in object fields) are stored subsequently. For instance, field `id` of the professor record contains an integer 1254; the fields `students` and `name` contain memory addresses `0x0504` and `0x070a`, respectively.




(a) Part I (b) Part II

**Figure 2.** A transformation example.

### 2.2 Using Objects as Facades

We propose to create heap objects as *facades* for a data type, 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 2 depicts an example with five transformations. Suppose all fields of the `Professor` class in Figure 1 are also in the `Professor` class in Figure 2 (a). For simplicity of illustration, we show the original, 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.

Figure 2 (a) shows the class transformation. For illustration, let us suppose 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-based reference of a data record (such as `0x0504` in Figure 1). Setting a page reference to the field `pageRef` of a facade *binds* the data record with the facade, so that methods defined in the facade will be invoked to process this record. A reader can think of this field as the `this` reference in a regular Java program.

***Transformation 1*** `ProfessorFacade` does not contain any instance field; for each instance field $f$ in `Professor`, `ProfessorFacade` contains a static field $f\_Offset$, specifying the offset (in numbers of bytes) of $f$ to the starting address of the data record. These offsets will be used to transform field accesses.

***Transformation 2*** For method `addStudent` in `Professor`, FACADE generates a new method with the same name in `ProfessorFacade`. First, its signature is changed in such a way that if a parameter has a data class type (e.g., `Student`) in $P$, it now has a facade type (e.g., `StudentFacade`) in $P'$. Since a facade does not contain actual data, the new facade parameter $sf$ in $P'$ 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 (line 10 and 11 in $P'$) from the receiver (i.e., `this`) and $sf$, and keeps them in two local variables $this\_ref$ and $s\_ref$. Any subsequent statement that uses objects (pointed to by) `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 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 $P'$.

***Transformation 3*** The allocation at lines 12–13 in $P$ is transformed to lines 29–36 in $P'$. FACADE allocates space based on the student size by calling a library method `allocate`, which returns a page reference $s\_ref$. Since the allocation in $P$ involves a constructor call, we need to generate a new call in $P'$. The challenge is how to find a receiver (facade) object on which the call can be made. FACADE generates code to retrieve an available `StudentFacade` object from the pool (lines 32–33 in $P'$) and bind it with the page reference $s\_ref$ (lines 35). In this case, the first facade in the pool is always available; the reason will be ex-

plained shortly. The constructor in $P$ is converted to a regular method `facade$init` in $P'$. FACADE generates a call to `facade$init` on the retrieved facade object (line 36).

***Transformation 4*** Variable assignments (lines 14–15) in $P$ are transformed into page reference assignments (lines 37–38) in $P'$.

***Transformation 5*** Similarly, a call to `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). However, before generating the call site at line 47, we have to emit additional code to prepare for (1) the receiver `ProfessorFacade` object *pf2* and (2) the parameter `StudentFacade` object *sf2*. This preparation can be done in a similar manner by requesting available facades from different pools and binding them with the corresponding references (lines 39–46).

### 2.3 Bounding the Number of Facades in Each Thread

Since a facade is used only to carry a page reference for a control task (e.g., parameter passing, value returning, etc.), the facade is available for reuse once the page reference it carries is loaded to a stack variable—from this point on, the page reference will be propagated and used. A key treatment here is that for a pair of instructions (e.g., $s$ and $t$) that bind a facade with a page reference and release the binding, our compiler guarantees that $t$ 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 $t$. Examples of such instruction pairs include lines 42 and 10, and lines 46 and 11 in $P'$ of Figure 2.

The facade is reusable immediately after instruction $t$ to carry other page references. This treatment enables an important property in (each thread of) $P'$: *for any control instruction that needs a facade of a data type, all facades in the pool for the type are available for use*. This explains why it is always safe to use the first facade of the pool at lines 33, 40, and 44. Data instructions that access the heap do not need facades because they have been transformed to access native memory using page references.

Clearly, the number of facades needed for a data type depends on the number of operands of this type needed in a control instruction. For example, if a call site in $P$ requires $n$ arguments of type `Student`, we need at least $n$ `StudentFacade` objects in $P'$ for parameter passing (e.g., `Pools.studentFacades[0]`, ..., `Pools.studentFacades[`$n$ `- 1]`). Since a call instruction takes many more operands than other kinds of instructions, it is often the case that the number of facades for type `DFacade` in $P'$ is bounded by the maximal number of arguments of type `D` required by a call in $P$. Based on this observation, we can inspect all call sites in $P$ in a pre-transformation pass and compute a bound statically for each data type. The bound will be used to determine the size of the facade pool for the type (e.g., `Pools.studentFacades`) *statically*.

This property of $P'$ distinguishes our approach from traditional object pooling, which often requires explicit (runtime) support for requesting/returning objects from/to the pool and does not provide any bound guarantee. A detailed discussion on how FACADE differs from object pooling can be found in § 5.

It is easy to see that at different points, different facades may be retrieved from the pool to carry the same page reference. For instance, in Figure 2 (b), although variable $p$ (line 16) and parameter $f$ (line 11) refer to the same object in $P$, their corresponding facades *pf* and *pf2* in $P'$ may not be the same. In a single-threaded execution, this would not cause any inconsistency because page references determine data records and facades are used only to execute control flow. Multithreading will be discussed in §3.4.

### 2.4 Performance Benefits

$P'$ has the following clear advantages over $P$. First, 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. Second, all data records are stored in native pages and no longer subject to garbage collection. This can lead to an orders-of-magnitude reduction on the number of nodes and edges traversed by the GC. Third, native-memory-based data storage reduces the memory access cost. In addition, FACADE inlines all data records whose size can be statically determined, which improves data locality and reduces the cost of memory dereferences.

## 3. 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. While our transformations can be formalized and their correctness can be proved, we describe them in plain language to make the paper accessible to a broad community of researchers and practitioners.

### 3.1 Our Assumptions

Based on the (user-provided) list of data classes, FACADE makes two important "closed-world" assumptions. 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 Big Data application: classes representing data tuples (e.g., graph nodes and edges) and those representing data manipulation functions, such as sorter, grouper,

etc. Both kinds of classes rarely contain fields of non-data types. Java supports a collections framework and data structures in this framework can store both data objects and non-data objects. In FACADE, a collection (e.g., HashMap) is treated as a data type; a new type (e.g., HashMapFacade) is thus generated in the data path. The original type is still used in the control path. If FACADE detects 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 §3.5.

The second assumption is a *type-closed-world* assumption that requires 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 non-data 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. A special handling here is that we allow both a data class and a non-data 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.

### 3.2 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 $TFacade$. If $D$ extends another data class $E$, this relationship is preserved by having $DFacade$ extends $EFacade$. All static fields declared in $D$ are also in $DFacade$; however, $DFacade$ does not contain any instance field.

One challenge here is how to appropriately handle Java interfaces. If an interface $I$ is implemented by both a data class $C$ and a non-data 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 run time.

***Instruction transformation*** Instruction transformation is performed on the control flow graph (CFG) of a 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 1. 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 §3.5). 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 non-data 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 non-data 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 data 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 run time, 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 type. The pool contains only a single facade object; the `resolve` method always returns the facade from this pool, which is separated 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 1.

### 3.3 Computing Bounds

Before the transformation, FACADE inspects parameters of each method in the data path to compute a bound for each data type. 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 run time, a facade of the

| ***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) None 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 $B\ covertToB(long)$; 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) None 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 $long\ covertFromB(B)$; 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) $return\ a$ | (5.1) $a$ has a data type | Retrieve $a\_ref$ from $v$; emit three instructions: $AFacade\ af = Pools.aFacades[0]$; $af.pageRef = a\_ref$; $return\ af$ |
| | (5.2) Otherwise | Emit instruction $return\ a$ |
| (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: $AFacade\ af = resolve(a\_ref)$; $BFacade\ bf = Pools.bFacades[i]$; $af.pageRef = a\_ref$; $bf.pageRef = b\_ref$; $af.m(\ldots, bf, \ldots)$ |
| | (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 $B\ covertToB(long)$; emit a call $a.m(\ldots, covertToB(b\_ref), \ldots)$ |
| | (6.4) None of them have a data type | Emit a call $a.m(\ldots, b, \ldots)$ |
| (7) $boolean\ t = a\ instanceof\ B$ | (7.1) $a$ has a data type and $B$ is a data type | Retrieve $a\_ref$ from $v$; emit two instructions: $AFacade\ af$ = $resolve(a\_ref)$; $t$ = $af\ instanceof\ BFacade$ |
| | (7.2) $B$ is an array type | Emit $t = arrayTypeID(a) == ID(B)$ |
| | (7.3) None of them have a data type | Emit $t = a\ instanceof\ B$ |

**Table 1.** A summary of code generation; suppose variables $a$ and $b$ have types $A$ and $B$, respectively.

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 a 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 type 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.

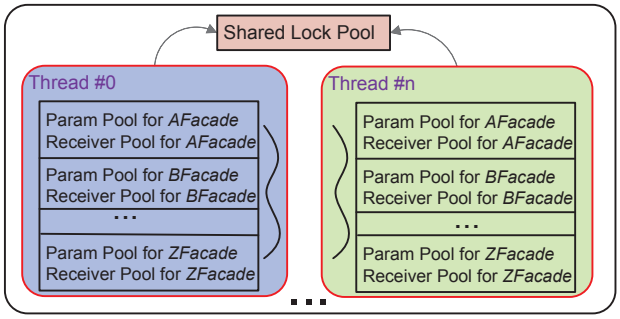



**Figure 3.** A graphical representation of threads and pools, where $AFacade$, $BFacade$, ..., and $ZFacade$ are facade types.

### 3.4 Supporting Concurrency

Naïvely transforming a multi-threaded 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 type, the receiver pool and the regular pool are maintained for each thread. We implement this by associating one instance of class `Pools` with each thread;

the `init` method (discussed in §3.3) 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` $(o)\{\ldots\}$ 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` $(o')\{\ldots\}$. 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 3) 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 field 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 zeroing 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 type. Since the number of facades needed for a data type is a compile-time constant, the total number of facades in the system is $O(t * n)$, where $t$ and $n$ are the numbers of threads and data types, respectively. Considering the additional objects created to represent native pages, the number of heap objects needed in $P'$ is $O(t * n + p)$, 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 reentrant, 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 $O(t * n + p)$.

### 3.5 Data Conversion Functions

For each IP that involves a data type $D$, 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 type 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.

### 3.6 Memory Allocation and Page Management

The FACADE runtime system maintains a list of pages, each of which has a 32K space (i.e., a common practice in the database design [31]). 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) continuous allocation requests get contiguous space (to maximize locality); (2) large arrays are allocated on empty pages: allocating them on non-empty 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.

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 Big Data 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 Big Data 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 determine when to recycle pages. 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 logic. For example, in GraphChi [41], a single-machine graph processing framework, iteration-start and iteration-end are the callbacks explicitly defined by the framework. Although we 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 *sub-iteration* 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 run time. 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.

We have transformed all data classes in the JDK including various collection classes and array-based utility classes. Commonly-used native methods such as `System.arraycopy` and `Unsafe.compareAndSwap` are manually modeled. We have also implemented a set of optimization techniques, including (1) inlining of large arrays, primitive type wrappers (e.g., `Integer`), and objects that can be determined immutable statically; (2) static resolution of virtual calls based on a points-to analysis; and (3) use of a special "oversize" class to allocate large arrays whose size is bigger than 32K; pages on this class can be deallocated earlier when they are no longer needed (e.g., upon the resizing of a data structure). Details of these optimizations are omitted from this paper.

### 3.7 Correctness Argument

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

***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 coverts 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 [22]) to verify whether data items can escape. A real-world Big Data application is often *framework-intensive* 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.

The memory management correctness thus relies on the user's correct specification of data classes. Admittedly, a considerable amount of user effort is needed to understand the program and perform specifications. §4 reports our own experiences with finding data classes for real-world programs that we have never studied.

## 4. Evaluation

The implementation of FACADE is based on the Soot Java compiler infrastructure and consists of approximately 40,000 lines of Java code. We selected 3 well-designed Big Data frameworks and used FACADE to transform their data paths. Our evaluation on 7 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.

### 4.1 GraphChi

***Transformation*** GraphChi [41] 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 types 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 3 classes, FACADE detected 18 *boundary classes* that interact

with data classes but do not have many instances themselves. Boundary classes have both data and non-data 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-hour 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 (7753 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 sub-iterations.

***Test setup*** We tested the generated code and compared its performance with that of the original GraphChi code. The experiments were performed on a 4-core server with 4 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 [40], 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 GC was used for memory reclamation. This GC combines parallel Scavenge (i.e., copying) for the young generation and parallel Mark-Sweep-Compact for the old generation to quickly reclaim unreachable objects. GraphChi uses a parallel sliding windows algorithm that partitions data into shards. Since the number of shards has only little impact on performance (as reported in Figure 8(c) in [41] 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.

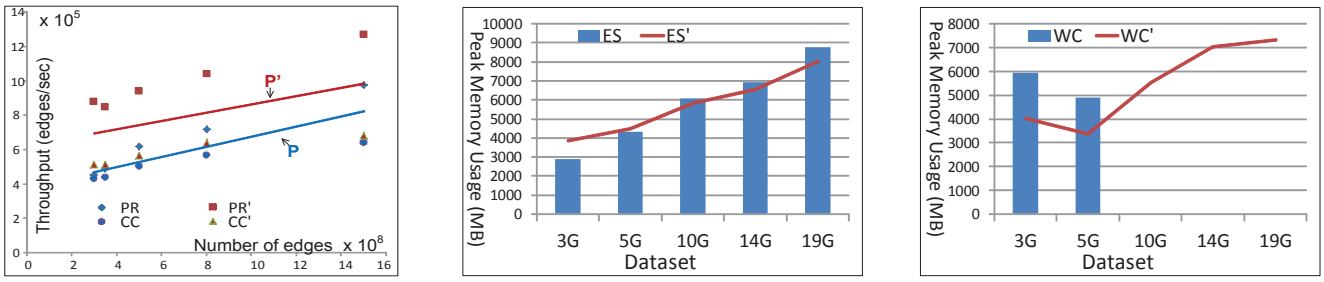
Table 2 shows the detailed performance comparisons. Note that our performance numbers may look different from those reported in [41], because their experiments used SSD and a C++ version of GraphChi. In Table 2, $P'$ outperforms $P$ for all configurations. The performance improvements FACADE has achieved for PR and CC over *twitter-2010* are, respectively, **26.8%** and **5.8%**; larger gains were seen when we experimented with smaller graphs (discussed shortly). The generated program $P'$ not only has much less GC time (i.e., an average 5.1× reduction); data load and engine update time has also been reduced primarily due to inlining and direct memory accesses.

| *App* | *ET(s)* | *UT(s)* | *LT(s)* | *GT(s)* | *PM(M)* |
|---|---|---|---|---|---|
| PR-8g | 1540.8 | 675.5 | 786.6 | 317.1 | 8469.8 |
| PR'-8g | **1180.7** | **515.3** | **584.8** | **50.2** | **6135.4** |
| PR-6g | 1561.2 | 694.0 | 785.2 | 270.1 | 6566.5 |
| PR'-6g | **1146.2** | **518.8** | **545.6** | **49.3** | **6152.6** |
| PR-4g | 1663.7 | 761.6 | 811.5 | 380.7 | **4448.7** |
| PR'-4g | **1159.2** | **499.2** | **580.0** | **50.6** | 6127.4 |
| CC-8g | 2338.1 | 1051.2 | 722.7 | 218.5 | 8398.3 |
| CC'-8g | **2207.8** | **984.3** | **661.0** | **50.3** | **6051.6** |
| CC-6g | 2245.8 | 1005.4 | 698.2 | 179.5 | 6557.8 |
| CC'-6g | **2143.4** | **951.6** | **628.2** | **49.3** | **6045.3** |
| CC-4g | 2288.5 | 1029.8 | 713.7 | 197.4 | **4427.4** |
| CC'-4g | **2120.9** | **932.7** | **630.4** | **50.6** | 6057.0 |

**Table 2. GraphChi performance comparisons on *twitter-2010***: reported are the total execution times (*ET*), engine update times (*UT*), data load times (*LT*), garbage collection times (*GT*), and peak memory consumptions (*PM*); each application has two runs (e.g., original run PR and facade run PR') under three different memory budgets (e.g., 8GB, 6GB, and 4GB); time and memory are measured in seconds and megabytes, respectively; *PM* 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; numbers that show better performance are highlighted.

For PR, the number of objects for its data classes has been reduced from $14,257,280,923$ to $1,000$ memory pages $+11 * (16 * 2 + 1) = 1,363$—other than the main thread, GraphChi uses two thread pools, each containing 16 threads, and each thread has a pool of 11 facades—which leads 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 5 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 control path, whose numbers are very small; the off-heap data storage 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, 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




(a) (b) (c)

**Figure 4.** (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**.

| *Data* | *ES* | *ES'* | *WC* | *WC'* |
|---|---|---|---|---|
| 3GB | 95.5 | **89.3** | **48.9** | 57.4 |
| 5GB | 178.2 | **167.1** | **72.5** | 180.8 |
| 10GB | 326.3 | **302.5** | OME(683.1) | **1887.1** |
| 14GB | 459.0 | **426.0** | OME(943.2) | **2693.0** |
| 19GB | 806.4 | **607.5** | OME(772.4) | **3160.2** |

**Table 3. Hyracks performance comparisons on different datasets**: reported are the total execution times of ES, ES', WC, and WC' measured in seconds; OME($n$) means the program runs out of memory in $n$ seconds.

allows dead data records to accumulate until the end of a (sub-)iteration (i.e., trades off space for time).

***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 4 (a). An 8GB heap was used to run $P$ and $P'$. 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 between $P$ and $P'$ is even larger than what is reported in Table 2. For example, on a graph with 300M edges, PR' and CC' are **48%** and **17%** faster than PR and CC, respectively.

### 4.2 Hyracks

Hyracks [2, 16] 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, the 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 the 8 (data and boundary) classes in 15 seconds, resulting in a speed of 990 instructions per second. 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.

***Test setup*** We ran Hyracks on a 10-slave-node (c3.2x large) Amazon EC2 cluster. Each machine has 2 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 [5] 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 update starts. We ran both $P$ and $P'$ with an 8GB heap. When the heap is exhausted in $P$, JVM terminates immediately with out-of-memory errors. Naïvely comparing scalability would create unfairness for $P$, because $P'$ uses much 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 3 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 dataset, 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 3160.2 seconds for the 19GB dataset. Although both ES and ES' can scale to 19GB, ES' is about **24.7%** faster than ES.

Figure 4 (b) and (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 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).

### 4.3 GPS

GPS [58] is a distributed graph processing system developed for scalable processing of large graphs. We profiled the execution and identified a total number of 4 (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 10691 Jimple instructions) in 9.7 seconds, yielding a 1102 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. Due to space limitations, here we only briefly describe our experimental results.

GPS is overall less scalable than GraphChi and Hyracks due to its object array-based representation of an input graph. However, its extensive use of primitive arrays, which is similar in spirit to what FACADE intends to achieve, leads to relatively small GC effort. For example, GC accounts for only 1–17% of the running time. The set of inputs we used includes the *twitter-2010* graph, the *LiveJournal* graph, and 5 synthetic supergraphs of *LiveJournal* (e.g., the largest supergraph has 120M vertices and 1.7B edges). Compared to the original implementation $P$, the generated version $P'$ has achieved a 3–15.4% running time reduction, a 10–39.8% GC time reduction, as well as 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 $P'$.

### 4.4 Summary

Although we had never studied any of these frameworks before, 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. 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.

## 5. Related Work

***Optimizations of Big Data applications*** While there exists a large body of work on optimizing Big Data applications, these existing efforts focus on domain-specific optimizations, including, for example, data pipeline optimizations [7, 16, 19, 21, 33, 38, 41, 55, 73, 76], query optimizations [23, 25, 45, 52, 54, 56], and Map-Reduce-related optimizations [6, 24, 47, 57, 60, 61, 72]. Despite the commendable accomplishments of these optimizations, Big Data performance is fundamentally limited by memory inefficiencies inherent with the underlying programming systems. Zing [1] is a commercial system developed by Azul that can lower the latency for Java-based Big Data applications by making larger in-memory indexes. This paper attempts to solve this problem by limiting the number of objects used to represent data records, an approach that is orthogonal to, and will provide benefit for, all existing optimization techniques.

***Region-based memory management*** Region-based memory management was first used in the implementations of functional languages [8, 62] such as Standard ML [35], and then was extended to Prolog [50], C [29, 30, 32, 37], and real-time Java [13, 17, 39]. More recently, some mark-region hybrid methods such as Immix [14] 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 [13, 17]) 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 non-intrusive 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 objects via program analysis*** Object inlining [26, 46] is a technique that statically inlines objects in a data structure into its root to reduce the number of point-

ers and headers. Free-Me [34] adds compiler-inserted frees to a GC-based system. Pool-based allocation proposed by Lattner et al. [42–44] 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 [28] 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.

Shuft et al. [59] 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 GC (analogous to to a nursery in a generation collector); objects with non-prolific 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 types 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 profile.

*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 take much 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 [12] provides a hierarchical mechanism for enforcing limits on resources, especially the CPU. HiStar [75] 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 [11, 36] has gone towards 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 [27, 51]), 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. One 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 [3] 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 letting developers rewrite programs.

## 6. Conclusions

Growing datasets require efficiency on all levels of the processing stack. This paper targets the performance problem caused by excessive object creation in a managed Big Data system, and proposes a compiler and runtime FACADE that achieves high efficiency by performing a semantics-preserving transformation of the data path of a Big Data 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.

## Acknowledgments

We would like to thank Michael Bond, David Liu, Kathryn McKinley, Feng Qin, and Dacong Yan for their helpful comments on an early draft of the paper. We also thank the ASPLOS reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under grant CNS-1321179 and CCF-1409829, and by the Office of Naval Research under grant N00014-14-1-0549.

## References


[1] Zing: Java for the real time business. http://www.azulsystems.com/products/zing/whatisit.

[2] Hyracks: A data parallel platform. http://code.google.com/p/hyracks/, 2014.

[3] The Rust programming language. http://www.rust-lang.org/, 2014.

[4] Soot framework. http://www.sable.mcgill.ca/soot/, 2014.

[5] Yahoo! webscope program. http://webscope.sandbox.yahoo.com/, 2014.

[6] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *International Conference on Extending Database Technology (EDBT)*, pages 99–110, 2010.

[7] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, 2008.

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

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

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

[11] G. Back and W. C. Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[12] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 45–58, 1999.

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

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

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

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

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

[18] Y. Bu, V. Borkar, G. Xu, and M. J. Carey. A bloat-aware design for big data applications. In *ACM SIGNPLAN International Symposium on Memory Management (ISMM)*, pages 119–130, 2013.

[19] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

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

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

[22] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[23] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 21–21, 2010.

[24] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[25] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3:515–529, 2010.

[26] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 345–357, 2000.

[27] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 2–15, 2006.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

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

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

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

[33] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 121–133, 2012.

[34] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-Me: a static analysis for automatic individual object reclamation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 364–375, 2006.

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

[36] C. Hawblitzel and T. von Eicken. Luna: A flexible Java protection system. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 391–403, 2002.

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

[38] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, 2007.

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

[40] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *International World Wide Web Conference (WWW)*, pages 591–600, 2010.

[41] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, 2012.

[42] C. Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[43] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 129–142, 2005.

[44] C. Lattner, A. Lenharth, and V. Adve. 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)*, pages 278–289, 2007.

[45] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He, and X. Zhang. Ysmart: Yet another SQL-to-MapReduce translator. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 25–36, 2011.

[46] O. Lhotak and L. Hendren. Run-time evaluation of opportunities for object inlining in Java. *Concurrency and Computation: Practice and Experience*, 17(5-6):515–537, 2005.

[47] J. Liu, N. Ravi, S. Chakradhar, and M. Kandemir. Panacea: Towards holistic optimization of MapReduce applications. In *International Symposium on Code Generation and Optimization (CGO)*, pages 33–43, 2012.

[48] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 340–349, 2010.

[49] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A framework for machine learning in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

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

[51] Y. Mandelbaum, K. Fisher, D. Walker, M. F. Fernández, and A. Gleyzer. PADS/ML: a functional data description language. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 77–83, 2007.

[52] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 121–131, 2011.

[53] K. Nguyen and G. Xu. Cachetor: Detecting cacheable data to remove bloat. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 268–278, 2013.

[54] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, 2010.

[55] C. Olston, B. Reed, A. Silberstein, and U. Srivastava. Automatic optimization of parallel dataflow programs. In *USENIX USENIX Annual Technical Conference (ATC)*, pages 267–273, 2008.

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

[57] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Sci. Program.*, 13(4):277–298, 2005.

[58] S. Salihoglu and J. Widom. GPS: A graph processing system. In *Scientific and Statistical Database Management*, July 2013.

[59] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 295–306, 2002.

[60] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[61] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using hadoop. In *International Conference on Data Engineering (ICDE)*, pages 996–1005, 2010.

[62] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lamda-calculus using a stack of regions. In *ACM*

*SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 188–201, 1994.

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

[64] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction (CC)*, pages 18–34, 2000.

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

[66] G. Xu. 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)*, pages 111–130, 2013.

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

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

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

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

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

[72] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1029–1040, 2007.

[73] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. 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)*, pages 1–14, 2008.

[74] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX conference on Hot topics in cloud computing (HotCloud)*, page 10, Berkeley, CA, USA, 2010.

[75] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, 2006.

[76] J. Zhou, P.-Å. Larson, and R. Chaiken. Incorporating partitioning and parallel plans into the SCOPE optimizer. In *International Conference on Data Engineering (ICDE)*, pages 1060–1071, 2010.