Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen
Iowa State University, USA
{rdyer,hoan,hridesh,tien}@iastate.edu

***Abstract*—In today's software-centric world, ultra-large-scale software repositories, e.g. SourceForge (350,000+ projects), GitHub (250,000+ projects), and Google Code (250,000+ projects) are the new library of Alexandria. They contain an enormous corpus of software and information about software. Scientists and engineers alike are interested in analyzing this wealth of information both for curiosity as well as for testing important hypotheses. However, systematic extraction of relevant data from these repositories and analysis of such data for testing hypotheses is hard, and best left for mining software repository (MSR) experts! The goal of *Boa*, a domain-specific language and infrastructure described here, is to ease testing MSR-related hypotheses. We have implemented *Boa* and provide a web-based interface to *Boa*'s infrastructure. Our evaluation demonstrates that *Boa* substantially reduces programming efforts, thus lowering the barrier to entry. We also see drastic improvements in scalability. Last but not least, reproducing an experiment conducted using *Boa* is just a matter of re-running small *Boa* programs provided by previous researchers.**

***Index Terms*—mining, software, repository, reproducible, scalable, ease of use, lower barrier to entry**

I. Introduction

Ultra-large-scale software repositories, e.g. SourceForge (350,000+ projects), GitHub (250,000+ projects), and Google Code (250,000+ projects) contain an enormous collection of software and information about software. Assuming only a meagre 1K lines of code (LOC) per project, these big-3 repositories amount to at least 8.61 billion LOC alone. Scientists and engineers alike are interested in analyzing this wealth of information both for curiosity as well as for testing such important hypotheses as: "how people perceive and consider the potential impacts of their own and others' edits as they write together? [1]"; "what is the most widely used open source license? [2]"; "how many projects continue to use DES (considered insecure) encryption standards? [3]"; "how many open source projects have a restricted export control policy? [4]"; "how many projects on an average start with an existing code base from another project instead of scratch? [5]"; "how often do practitioners use dynamic features of Javascript, e.g. `eval`? [6]; "What is the average time to resolve a bug reported as critical? [7]".

However, the current barrier to entry could be prohibitive. For example, to answer the questions above, a research team would need to (a) develop expertise in programmatically accessing version control systems, (b) establish an infrastructure for downloading and storing the data from software repositories since running experiments by directly accessing this data is often time prohibitive, (c) program an infrastructure in a full-fledged programming language like C++, Java, C#, or Python to access this local data and answer the hypothesis, and (d) improve the scalability of the analysis infrastructure to be able to process ultra-large-scale data in a reasonable time.

These four requirements substantially increase the cost of scientific research. There are four additional problems. First, experiments are often unreproducible because replicating an experimental setup requires a mammoth effort. Second, reusability of experimental infrastructure is typically low because analysis infrastructure is not designed in a reusable manner. After all, the focus of the original researcher is on the result of the analysis and not on reusability of the analysis infrastructure. Thus, researchers commonly have to replicate each other's efforts. Third, data associated and produced by such experiments is often lost and becomes inaccessible and obsolete, because there is no systematic curation. Last but not least, building analysis infrastructure to process ultra-large-scale data efficiently can be very hard [8]–[10].

To solve these problems, we have designed a domain-specific programming language for analyzing ultra-large-scale software repositories, which we call *Boa*. In a nutshell, *Boa* aims to be for open source-related research what Mathematica is to numerical computing, R is for statistical computing, and Verilog and VHDL is for hardware description. We have implemented *Boa* and provide a web-based interface to *Boa*'s infrastructure [11].

To evaluate *Boa*'s design and effectiveness of its infrastructure we have written programs to answer 21 different research questions in four different categories: questions related to the use of programming languages, project management, legal, and those that relate to platform/environment. Our results show that *Boa* substantially decreases the efforts of researchers analyzing human and technical aspects of open source software development allowing them to focus on their essential tasks. We also see ease of use, substantial improvements in scalability, and lower complexity and size of analysis programs (see Figure 4). Last but not least, reproducing an experiment conducted using *Boa* is just a matter of re-running, often small, *Boa* programs provided by previous researchers.

We now describe *Boa* and explore its advantages. First, we present the language (Section II) and describe its infrastructure (Sections III–IV). Section V presents studies of applicability, scalability, and reproducibility. Section VI positions our work in the broader research area and Section VII concludes.

Java

```
... // imports

public class GetChurnRates {
  public static void main(String[] args) {
    new GetChurnRates().getRates(args[0]);
  }
  public void getRates(String cachePath) {
    for (File file : (File []) FileIO.readObjectFromFile(cachePath)) {
      String url = getSVNUrl(file);
      if (url != null && !url.isEmpty())
        System.out.println(url + "," + getChurnRateForProject(url));
    }
  }
  private double getChurnRateForProject(String url) {
    double rate = 0;
    SVNURL svnUrl;
     ... // connect to SVN and compute churn rate

    return rate;
  }
  private String getSVNUrl(File file) {
    String jsonTxt = "";
     ... // read the file contents into jsonTxt

    JSONObject json = null, jsonProj = null;
     ... // parse the text, get the project data

     if (!jsonProj.has("programming−languages")) return "";
     if (!jsonProj.has("SVNRepository")) return "";
    boolean hasJava = false;
     ... // is the project a Java project?

     if (!hasJava) return "";
    JSONObject svnRep = jsonProj.getJSONObject("SVNRepository");
     if (!svnRep.has("location")) return "";
    return svnRep.getString("location");
  }
}
```

Boa

```
1 rates: output mean[string] of int;
2 p: Project = input;
3 foreach (i: int; p.code_repositories[i].kind == RepositoryKind.SVN && len(p.
          code_repositories[i].revisions) > 10)
4   exists (j: int; match(`^java$`, lowercase(p.programming_languages[j])))
5     foreach (k: int; len(p.code_repositories[i].revisions[k].files) < 100)
6       rates[p.id] << len(p.code_repositories[i].revisions[k].files);
```

Boa's Advantages

- Easy to use - simple programs, as small as 5-10 lines
- Better abstractions - hides specifics of mining software repositories
- Efficient and scalable - Results in as little as one minute
- Enhances reproducibility - Researchers can publish their small *Boa* programs and the dataset used so that others may reproduce the results

Performance Results

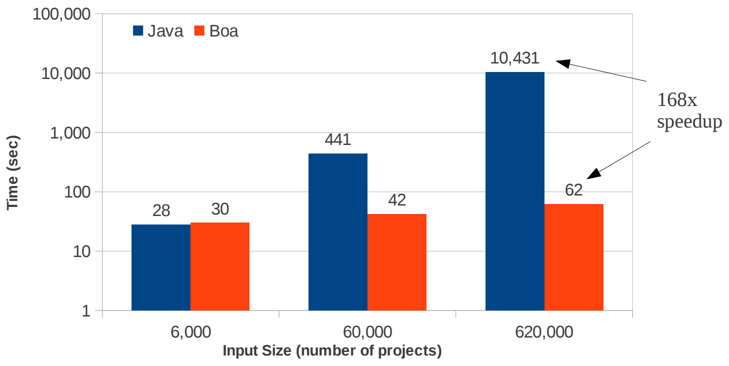

Fig. 1. Programs for answering "What are the churn rates for all Java projects that use SVN?" and performance results on three input sizes.

II. Motivation

Creating experimental infrastructure to analyze the wealth of information available in open source repositories is difficult [12]–[16]. Creating an infrastructure that scales well is even harder [15], [16]. To illustrate, consider a question such as "what are the average numbers of changed files per revision (churn rates) for all Java projects that use SVN?" Answering this question would require knowledge of (at a minimum): reading project metadata and mining code repository locations, how to access those code repositories, additional filtering code, controller logic, etc. Writing such a program in Java for example, would take upwards of 70 lines of code and require knowledge of at least 2 complex libraries. A heavily elided example of such a program is shown in Figure 1, left column.

This program assumes that the user has manually downloaded all project metadata, available as JSON files, and SVN repositories from SourceForge. It then processes the data using a JSON library and collects a list of Subversion URLs. A SVN library is then used to connect to each cached repository in that list and calculate the churn rate for the project. Notice that this code required use of 2 complex, external libraries in addition to standard Java classes and resulted in almost 70 lines of code. It is also sequential, so it will not scale as the data size grows. One could write a concurrent version, but this would add additional complexity.

A. *Boa*: Enabling Data Intensive Open Source Research

We designed and implemented a domain-specific programming language that we call *Boa* to solve these problems. *Boa* aims to lower the barrier to entry and thus enable a larger, more ambitious line of data intensive scientific discovery in open source software development-related research. The main features of *Boa* are inspired from existing languages for data-intensive computing [8], [9], [17], [18]. To these we add built-in types that are specifically designed to ease analysis tasks common in open source software mining research.

To illustrate the features of *Boa*, consider the same question "what are the churn rates for all Java projects that use SVN?". A *Boa* program to answer this question is shown in Figure 1, right column. On line 1, this program declares an output called `rates`, which collects integer values and produces a final result by aggregating the input values for each project (indexed by a string) using the function `mean`. On line 2, it declares that the input to this program will be a project, e.g. Apache OpenOffice. *Boa*'s infrastructure manages the details of downloading projects and their associated information. For each project, the code on lines 3–6 runs. If a repository contains 600k projects, the code on lines 3–6 runs for each.

On line 3, this program says to run code on lines 4–6 for each of the input project's code repositories that are Subversion and contain more than 10 revisions (to filter out

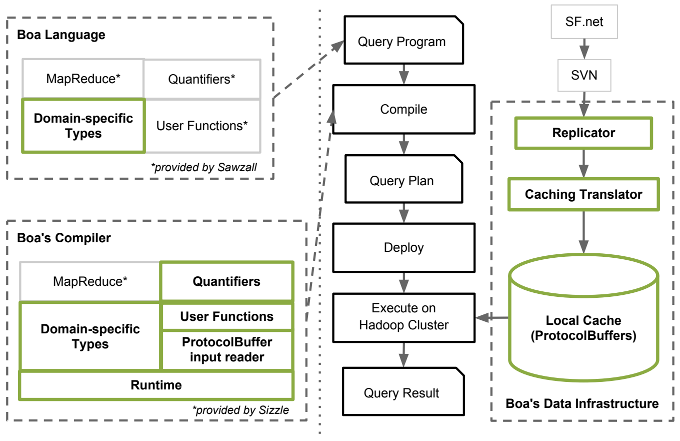

Fig. 2. An Overview of *Boa*'s Infrastructure. New components are marked with green boxes and bold text.

new or abandoned projects). On line 4, this program says to run code on lines 5–6, if and only if for the input project at least one of the programming languages used is Java. Line 5 selects only revisions from such repositories that have less than 100 files changed (to filter out extremely large commits, such as the first commit of a project). Finally, on line 6, this program says to send the length of the array that contains the changed files in the revision to the aggregator `rates`, indexed by the project's unique identifier string. This aggregator produces the final answer to our question.

These 6 lines of code not only answer the question of interest, but run on a distributed cluster potentially saving hours of execution time. Note that writing this small program required no intimate knowledge of how to find/access the project metadata, how to access the repository information, or any mention of parallelization. All of these concepts are abstracted from the user, providing instead simple primitives such as the `Project` type which contains attributes related to software projects such as the name, programming languages used, repository locations, etc. These abstractions substantially ease common analysis tasks.

Since this program runs on a cluster, it also scales extremely well compared to the (sequential) version written in Java. The time taken to run this program on varying input sizes is shown in the lower right of Figure 1. Note that the y-axis is in logarithmic scale. The time to execute the Java program increases roughly linearly with the size of the input while the *Boa* program sees minimal increase in execution time.

We have built an infrastructure for the *Boa* programming language. An overview of this infrastructure is presented in Figure 2. Components are shown inside dotted boxes on the left, the flow of a *Boa* program is shown in the middle, and the input data sources are shown on the right.

The three main components are: the *Boa* language, compiler and runtime, and supporting data infrastructure. First, an analysis task is phrased as a *Boa* program, e.g. that in Figure 1 (see Section III). This program is fed to our compiler (see Section IV-A) via our web-based interface (see Section IV-C). The *Boa* compiler produces a query plan. Our infrastructure then deploys this query plan onto a Hadoop [19] cluster, where it executes. The cluster makes use of a locally cached copy of the source code repositories (see Section IV-B) and based on the query plan creates tasks to produce the final query result. This is the answer to the user's analysis task. We now describe these components in detail.

III. Design of the *Boa* Language

The top left portion of Figure 2 shows the main features of the *Boa* language. We have four main kinds of features at the moment: domain-specific types to ease analysis of open source software repository mining, MapReduce [8] support for scalable analysis of ultra-large-scale repositories, quantifiers for easily expressing loops, and the ability to define functions.

A. Domain-Specific Types in Boa

The *Boa* language provides several domain-specific types for mining software repositories. Figure 3 gives an overview of these types. Each type provides several attributes that can be thought of as read-only fields.

Type	Attributes
Project	id, name, created_date, code_repositories, . . .
Repository	url, kind, revisions
Revision	id, log, committer, commit_date, files
Person	username, real_name, email
File	name, kind

Fig. 3. Some of the domain-specific types provided in *Boa*.

The `Project` type provides metadata about an open source project in the repository, including its name, url, some descriptions, who maintains and develops it, and any code repository. This type is used as input to programs in the *Boa* language.

The `Repository` type provides all of the `Revisions` committed into that repository. A revision represents a group of artifact changes and provides relevant information such as the revision id, commit log and time, the `Person` who committed the revision, and the `Files` committed.

B. MapReduce Support in Boa

In MapReduce [8] frameworks, computations are specified via two user-defined functions: a *mapper* that takes key-value pairs as input and produces key-value pairs as output, and a *reducer* that consumes those key-value pairs and aggregates data based on individual keys. Syntactically, *Boa* is reminiscent of Sawzall [9], a language designed for analyzing log files. In *Boa*, like Sawzall, users write the mapper functions directly and use built-in aggregators as the reduce function. Users declare output tables, process the input, and then send values to the tables. Output declarations specify aggregation functions and the language provides several built in aggregators, such as summing, min/max, mean, etc.

For example, we could write an output declaration for the table `rates` (as shown in Figure 1, line 1). For this table we want to index it by `strings` and give it values of type `int`. We would also like to use the aggregation function `mean`, which produces the mean of each integer emitted to the aggregator. Thus the final result of our output table is a list of string keys, each of which has the mean of all integers indexed by that key.

The plan generated from this code creates one logical process for each project in the corpus. Each process then analyzes a single project's revisions, emitting to the project's table the number of changed files for each revision. The aggregation process then reduces the values sent to it and computes the means.

C. Quantifiers in Boa

Boa defines he quantifiers `exists`, `foreach`, and `ifall`. Their semantics is the similar to when statements with quantifiers as in Sawzall. Quantifiers represent an extremely useful sugar that appears frequently in mining tasks. The sugared form makes programs much easier to write and comprehend.

For example, the `foreach` quantifier on line 3 of Figure 1, is a syntactic sugar for a loop. The statement says each time, when the boolean condition after the semicolon evaluates to true, execute the code on lines 4–6. The `exists` quantifier on line 4 is similar, however the code on lines 5–6 should execute exactly once if there *exists* some (non-deterministically selected) value of `j` where the boolean condition holds.

Not shown is the `ifall` quantifier. This quantifier states the boolean condition must hold for all values. If this is the case, then the associated code executes exactly once.

D. User-Defined Functions in Boa

The *Boa* language provides the ability for users to write their own functions directly in the language. To ease certain common mining tasks, we added built-in functions. Since we can't anticipate all needs of the users, or since our choice of a particular algorithm may not match what the user needs, having the ability to add user-defined functions was important.

The syntax, as inspired by Sawzall, requires declaring the parameters for the function and return type and assigning it to a variable. Functions can be passed as a parameter to other functions or assigned to different variables (if the function types are identical). A concrete example of a user-defined function (`HasJavaFile`) is shown later in Figure 6.

IV. *Boa*'s Supporting Infrastructure

The bottom left portion of Figure 2 shows the various parts of the *Boa* compiler and runtime.

A. Compiler and Runtime

For our initial implementation, we started with code for the Sizzle [20] compiler and framework. Sizzle is an open-source Java implementation of the Sawzall language. Unlike the original Sawzall compiler, Sizzle provides support for generating programs that run on the Hadoop [19] open-source MapReduce framework.

Our main implementation efforts were in adding user-defined functions in the *Boa* compiler, adding support for quantifiers, and supporting the protocol buffer format as input. These efforts were in addition to adding support for our domain-specific types and custom runtime model.

1) User-Defined Functions: The initial code generation strategy for user functions uses a pattern similar to the Java `Runnable` interface. A generic interface is provided by the runtime, which requires specifying the return type of the function as a type argument. Each user-defined function then has an anonymous class generated which implements this interface and provides the body of the function as the body of the interface's `invoke` method. This strategy allows easily modeling the semantics of user-defined functions, including being able to pass them as arguments to other functions and assigning them to (similarly typed) variables.

2) Quantifiers: We modified the compiler to desugar quantifiers into `for` loops. This process requires the compiler to analyze the boolean conditions to automatically infer valid ranges for the loop. The range is determined based on the boolean condition's use of the declared quantifier variable. Currently, quantifiers must be used as indexers to array attributes in our custom types and the range of the loop is the length of the array. We plan to extend support to any array variable in the future.

3) Protocol Buffers: Protocol buffers are a data description format developed by Google that are stored as binary messages. This format was designed to be compact and relatively fast to parse, compared to other formats such as XML. Messages are defined using a struct-like syntax and a compiler is provided which generates Java classes to read and write messages in that format. The *Boa* compiler was modified to use these generated classes when generating code, by mapping them to the domain-specific types provided.

The *Boa* compiler accepts Hadoop `SequenceFiles` as input, which is a special file format similar to a map. It stores key/value pairs, where the key is the project and the value is the binary representation of the protocol buffer message containing that project's data. This format was chosen due to its ease in splitting the input across map tasks.

B. Data Infrastructure

While the semantic model we provide with the *Boa* language and infrastructure states that queries are performed against the source repository in its current state, actually performing such queries over the internet on the live dataset would be prohibitive. Instead, we locally cache the repository information on our cluster and provide monthly snapshots of the data. The right portion of Figure 2 shows the components and steps required for this caching.

The first step is to locally replicate the data. For SourceForge, there are 2 public APIs we make use of. The first is a JSON API that provides information about projects, including various metadata on the project and information about which repositories the project contains. We simply download and cache the JSON objects for each project. The second API is the public Subversion (SVN) urls for code repositories. We make use of a Java SVN library to locally clone these repositories.

	LOC			RTime (sec)		
Task	**Java**	**Boa**	**Diff**	**Java**	**Boa**	**Speedup**
A. Programming Languages						
1. What are the ten most used programming languages?	61	4	15x	602	59	10x
2. How many projects use more than one programming language?	32	4	8x	603	54	11x
3. In which year was Java added to SVN projects the most?	89	10	9x	6,998	41	171x
B. Project Management						
1. How many projects are created each year?	43	3	14x	651	42	16x
2. How many projects self-classify into each topic provided by SourceForge?	45	4	11x	556	46	12x
3. How many Java projects using SVN were active in 2011?	66	6	11x	5,053	56	90x
4. In which year was SVN added to Java projects the most?	107	6	18x	4,880	48	13x
5. How many revisions are there in all Java projects using SVN?	60	5	12x	4,636	59	79x
6. How many revisions fix bugs in all Java projects using SVN?	76	6	13x	10,750	45	239x
7. How many committers are there for each Java project using SVN?	69	6	12x	10,821	50	216x
8. How many Java projects using SVN does each committer work on?	72	4	18x	10,435	58	180x
9. What are the churn rates for all Java projects that use SVN?	68	5	14x	10,431	62	168x
10. How did the no. of commits for Java projects using SVN change over years?	79	6	13x	10,489	43	244x
11. For all Java projects using SVN, what is the distribution of commit log length?	82	6	14x	10,518	44	239x
C. Legal						
1. What are the five most used licenses?	63	4	16x	474	44	11x
2. How many projects use more than one license?	32	4	8x	522	57	9x
D. Platform/Environment						
1. What are the five most supported operating systems?	61	4	15x	469	57	8x
2. What are the projects that support multiple operating systems?	33	4	8x	597	41	15x
3. What are the five most popular databases?	61	4	15x	498	47	11x
4. What are the projects that support multiple databases?	32	4	8x	558	64	9x
5. How often is each database used in each programming language?	71	5	14x	598	49	12x

Fig. 4. Several example mining tasks, with lines of code and execution times (in seconds) for both Java and *Boa* programs solving the tasks.

Once the information is stored locally on our cluster, we run our caching translator to convert the data into the format required by our framework. The input to the translator is the JSON files and SVN repositories and the output is a Hadoop `SequenceFile` containing protocol buffer messages which store all the relevant data.

C. Web-Based Interface

We provide a web-based interface for submitting *Boa* programs, compiling and running those programs on our cluster, and obtaining the output from those programs. Users submit programs to the interface using our syntax-highlighting text editor. Each submission creates a job in the system, so the user can see the status of the compilation and execution, request the results (if available), and resubmit or delete the job.

A daemon running on the cluster identifies jobs needing compiled and submits the code to the compiler framework. If the source compiles successfully, then the resulting JAR file is deployed on our Hadoop cluster and the program executes. If the program finishes without error, the resulting output is made available to the user to download (as a text file).

V. Evaluation

This section presents our empirical evaluation on the scalability and the usefulness of our language and infrastructure. The dataset used in this section contains all metadata about all SourceForge projects (620k+[1]) and Subversion repository metadata for only the Java projects that use Subversion (23k+). Programs were executed on a standard Hadoop [19] 1.0.3 install with 1 name node, 1 job tracker node, and 6 compute nodes. The cluster was not tuned for performance, except for setting the maximum number of map tasks for each compute node equal to the number of cores on that node and increasing the VM heap size to use the available memory on each node.

[1]This includes "user" projects, which aren't listed.

A. Applicability

Our main claim is that *Boa* is applicable for researchers wishing to analyze ultra-large-scale software repositories. In this section we investigate this claim.

Research Question 1: *Does Boa help researchers analyze ultra-large-scale software repositories?*

To answer this question, we examined a set of tasks (see Figure 4) that cover a range of different categories. For each task, we implemented a *Boa* program to solve the task. We also implemented small Java programs to solve the same tasks. The Java programs were written by an expert in mining software repositories and then reviewed by a second person who is an expert in programming languages. The second person performed a code review and also simplified and condensed the programs to decrease the total lines of code as much as reasonably possible without impacting performance. This process substantially reduced (almost by half) the lines of code for the Java versions.

The Java programs were not written as Hadoop programs. Writing the programs in Hadoop would have added substantial additional complexity and lines of code to these programs.

We were interested in investigating how *Boa* helps researchers along three directions: 1) are programs easier to write, 2) do those programs take (substantially) less time to collect the data, and 3) is the language expressive enough to solve such tasks. For each task, we collected two metrics:

- Lines of code (LOC)[2]: the amount of code written
- Running time (RTime): the time to collect the data

All results are shown in Figure 4. The lines of code give an indication of how much effort was required to solve the tasks using each approach. For Java, the tasks required writing 32–107 lines of code and on average required 62 lines of code. Performing the same tasks in *Boa* required at most 10 lines of code and on average less than 5 lines of code. Thus there were 8–18 times fewer lines of code when using *Boa*.

Not shown in the table was the fact the Java programs also required using several libraries (for accessing SVN, parsing JSON data, etc). The *Boa* programs abstracted away the details of how to mine the data and thus the user was not required to use these additional, complex libraries.

The table also lists the time required to run each program and collect the desired data for the tasks. Note the Java programs accessed all JSON and SVN data from a local cache and the times do not include any network access. For the Java programs, there are three distinct groups of running times. The smallest times (A.1, A.2, B.1, B.2, and all of C and D) are tasks that only require parsing the project metadata and did not access any SVN data. The medium times (A.3, B.3, B.4, and B.5) accessed the SVN repositories but only required mining one (or very few) revisions. The largest times (B.6–B.11) all accessed the SVN repositories and mined most of the revisions to answer the task and thus required substantially more time. Note that for the *Boa* programs, all tasks finish on average in 50 seconds, regardless of the type of task. We see minimum speedups of 8 times but in the best case the *Boa* program solves the task almost 250 times faster!

Task	Java (cached)	Java (remote SVNs)	Boa	Speedup
A.3	6,998	45,793	41	1,117x
B.3	5,053	25,690	56	459x
B.4	4,880	18,700	48	390x
B.5	4,636	17,888	59	303x
B.6	10,750	95,404	45	2,120x
B.7	10,821	85,265	50	1,705x
B.8	10,435	95,755	58	1,651x
B.9	10,431	88,440	62	1,426x
B.10	10,489	100,883	43	2,346x
B.11	10,518	88,279	44	2,006x

Fig. 5. Time (in seconds) if Java tasks do not cache SVN repositories first.

While the times in Figure 4 utilize local caches for all data, including SVN repositories, researchers implementing such tasks might not first cache the SVN data. As such, we again present the times for all tasks that access SVN in Figure 5 with the difference being the Java programs now access the SVN repositories remotely. Compared to this strategy, *Boa* programs run over 2,000 times faster!

1) Detailed Examples: Figures 6–9 show four interesting *Boa* programs used to solve some of the tasks. These programs highlight several useful features of the language.

Figure 6 answers task A.3 and demonstrates the use of a user-defined functions. The function `HasJavaFile` (line 4) takes a single `Revision` as argument and determines if it

```
1  counts: output sum[int] of int;
2  p: Project = input;
3
4  HasJavaFile := function(rev: Revision): bool {
5    exists (i: int; match(`.java$`, rev.files[i].name))
6      return true;
7    return false;
8  }
9
10 foreach (i: int; def(p.code_repositories[i]))
11   exists (j: int; HasJavaFile(p.code_repositories[i].revisions[j]))
12     counts[yearof(p.code_repositories[i].revisions[j].commit_date)] << 1;
```

Fig. 6. Task A.3: Querying years when Java files were first added the most.

contains any files with the extension ".java". If the revision contains at least one such file it returns `true`. This function is used in the `when` statement (line 11) as the boolean condition.

```
1  counts: output sum of int;
2  p: Project = input;
3
4  exists (i: int; match(`^java$`, lowercase(p.programming_languages[i])))
5    foreach (j: int; p.code_repositories[j].url.kind == RepositoryKind.SVN)
6      foreach (k: int; isfixingrevision (p.code_repositories[j].revisions[k].log))
7        counts << 1;
```

Fig. 7. Task B.6: Querying number of bug-fixing revisions in Java projects using SVN.

Figure 7 answers task B.6 and makes use of the built-in function `isfixingrevision` (line 6). The function uses a list of regular expressions to match against the revision's log. If there is a match, then the function returns true indicating the log most likely was for a revision fixing a bug.

```
1  counts: output top(5) of string weight int;
2  p: Project = input;
3
4  foreach (i: int; def(p.licenses[i]))
5    counts << p.licenses[i] weight 1;
```

Fig. 8. Task C.1: Querying the five most used licenses.

Figure 8 answers task C.1 and makes use of a *top aggregator* (line 1). The emit statement (line 5) now takes additional arguments giving a weight for the value being emitted. The top aggregator then selects the top N results that have the highest total weight and gives those as output.

```
1  counts: output sum[string][string] of int;
2  p: Project = input;
3
4  foreach (i: int; def(p.programming_languages[i]))
5    foreach (j: int; def(p.databases[j]))
6      counts[p.programming_languages[i]][p.databases[j]] << 1;
```

Fig. 9. Task D.5: Querying pairs of how often each database is used in each programming language.

Figure 9 answers task D.5 and makes use of a multi-dimensional aggregator (line 1) to output pairs of results. Again, the `emit` statement (line 6) is modified. This time, the statement requires providing multiple indexes for the table.

[2]Ignores comments and blank lines. http://reasoning.com/downloads.html

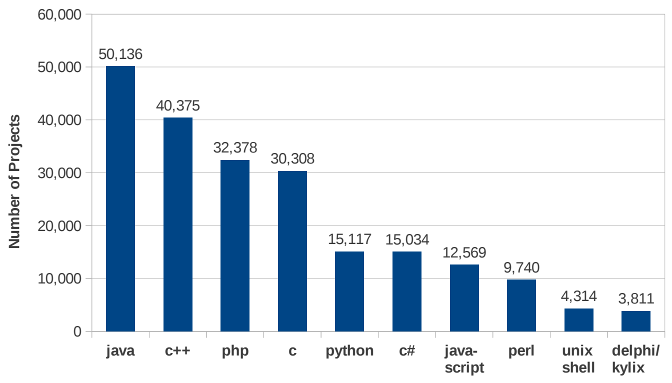

Fig. 10. Task A.1: Popularity of programming languages on SourceForge.

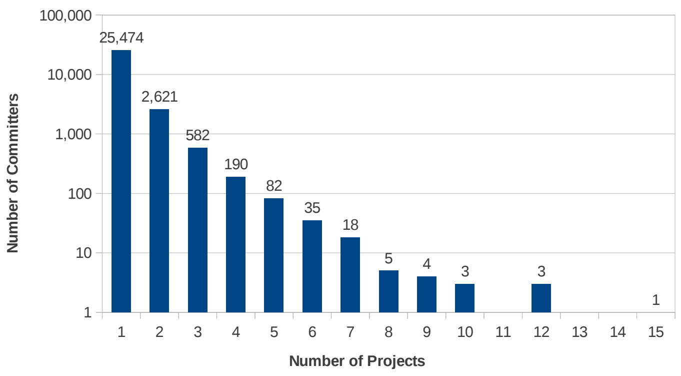

Fig. 12. Task B.8: no. of Java projects each SVN committer works on. NOTE: y-axis is in logarithmic scale.

2) *Results Analysis:* We also show some interesting and potentially useful results from four of the tasks. For example, Figure 10 shows the results of Task A.1 and charts the ten most used programming languages on SourceForge. 9 of the 10 languages appear in the top-12 of the TIOBE Index [21]. Languages such as Visual Basic did not appear in our results despite being #6 on the TIOBE index. This demonstrates that while the language is popular in general, it is not popular in open source. Similarly Objective-C did not appear in our results, as most programs written in Objective-C are for iOS and are (most likely) commercial, closed-source programs, or not typically hosted on SourceForge.

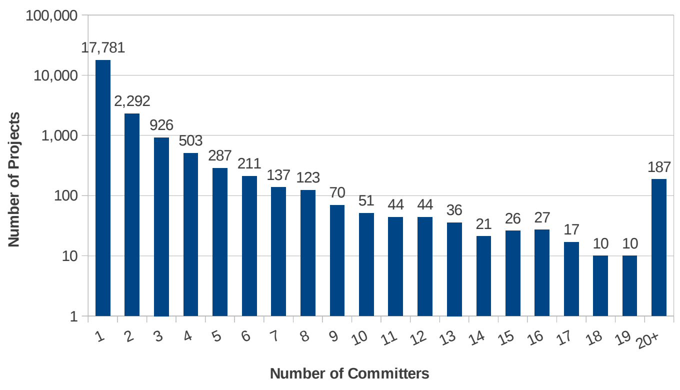

Fig. 11. Task B.7: number of committers in each Java project using SVN. NOTE: y-axis is in logarithmic scale.

The results of Task B.7 are shown in Figure 11. Note that the y-axis is in logarithmic scale. These results show that a large number of open-source projects have only a single committer. Generally, open-source projects are small and have very few committers and thus problems affecting large development teams may not show when analyzing open-source software.

Task B.8 looks at this data from the other angle. Figure 12 shows the number of projects each unique committer works on. Again, the vast majority of open-source developers only work on a single project. Only about 1% of committers work on more than three projects!

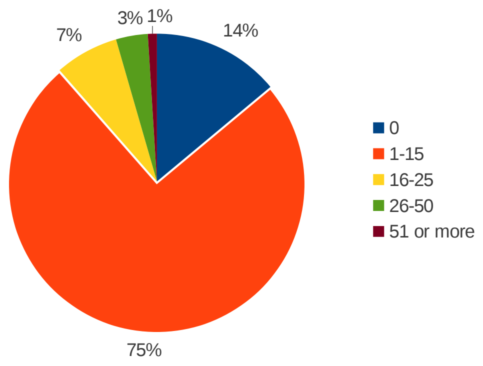

Fig. 13. Task B.11: no. of words in SVN commit logs for Java projects.

Another interesting result came from Task B.11 and is shown in Figure 13. This task examines how many words appear in log messages. First, around 14% of all log messages were completely empty. We do not investigate the reason for this phenomenom but simply point out how prevalent it is. Second, over two thirds of the messages contained 1–15 words, which is less than the average length of a sentence in English. A normal length sentence in English is 15–20 words (according to various results in Google) and thus we see that very few logs (10%) contained descriptive messages.

B. *Scalability*

One of our claims is that our approach is scalable. We investigate this claim in terms of scaling the size of the cluster and scaling the size of the input.

Research Question 2: *Does our approach scale to the size of the cluster?*

To answer this question, we run each of the sample programs listed in Figures 6–9 using our SourceForge.net dataset. We fix the size of the input to 620k projects and vary the number of available map slots in the system from 1–32. Figure 14 shows the results of this analysis where each group represents one of the sample programs, the y-axis is the total time taken in seconds to run the program, and the x-axis is the number of available map slots in the cluster. Each value is the average of 10 executions.

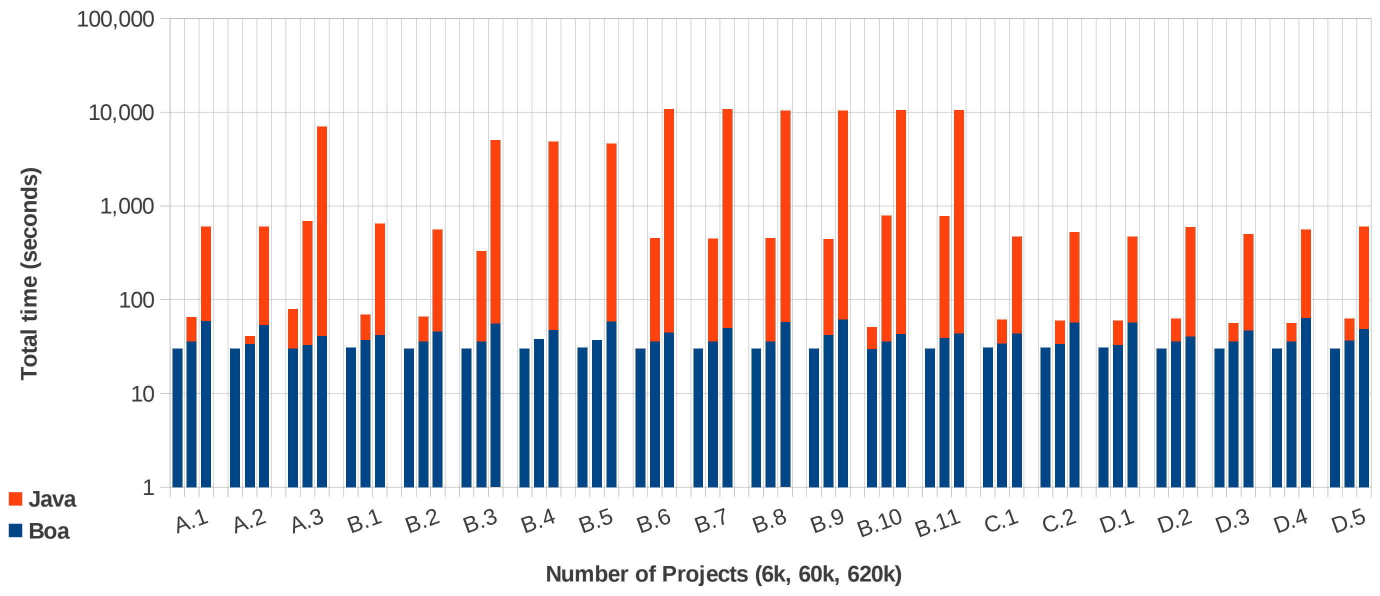

Fig. 15. Scalability of input. Y-axis is total time taken. X-axis is the size of the input in number of projects. NOTE: y-axis is in logarithmic scale.

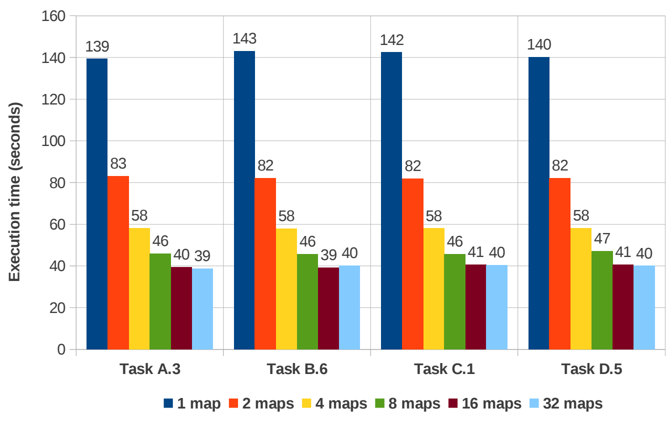

Fig. 14. Scalability of sample programs. Y-axis is total time taken. X-axis is the number of available map slots in the cluster.

As one might expect, the Hadoop framework works well with this large dataset. As the maximum number of map slots increases, we see substantial decreases in execution time as more parallel map slots are being utilized.

Note that with our current input size of 620k projects, the maximum number of map slots needed is 10. Thus we don't generally see any benefit when increasing the maximum map slots past that. As we increase the size of our input however, we would expect to see differences in these data points indicating scaling past 10 map slots.

Research Question 3: *Does our approach scale with the size of the input?*

To answer this question, we fix the number of compute nodes to 6 (with a total of 44 map slots available) and then vary the size of the input (6k, 60k, and 620k projects). The results for all tasks in Figure 4 are shown in Figure 15. We compare against the programs written in Java to answer the same questions. All programs access only locally cached data. Note that the y-axis is in logarithmic scale.

For the smallest input size (6k) on certain tasks, the Java program runs in around 10 seconds while the *Boa* program runs in 30 seconds. At this size *Boa* only uses one map task and thus the overhead of Hadoop dominates the execution time. For the larger input sizes, *Boa* always runs in (substantially) less time than the Java version.

The results also show that the hand written Java programs do not scale based on input size. As the input size increases, the running time for the Java programs also increases (roughly linearly). The *Boa* programs however demonstrate scalability. For the two smallest input sizes, the *Boa* programs take roughly the same amount of time. For the largest input size the *Boa* programs, despite having to process an input 100 times larger than the smallest input size, only take around twice as long. This shows that the *Boa* infrastructure scales well as the input size increases. Note that with our current maximum input size, *Boa* only utilizes 10 (out of 44) map slots and thus we expect more scalability should the input size increase further.

C. Reproducibility

One important claim we make is that if researchers publish results obtained from our infrastructure other researchers can easily reproduce the same results.

Research Question 4: *Using our infrastructure, can researchers easily reproduce previously published results?*

To answer this question, we performed a small controlled experiment. We selected a group of 8 researchers: 1 graduate student and 1 post-doc who are experts in software mining and 5 graduate and 1 undergraduate students who are not experts. Each student was given a short tutorial on how to use our infrastructure as well as the location of *Boa* source code for

18 tasks.[3] This source code represents what a researcher would publish in their paper, along with the dataset they used.

For each of the 18 tasks, results files were provided. This represents the data the previous researchers produced. Each student chose 3 tasks they were interested in reproducing and were given a maximum of 1 hour per task. We measured the length of time required to reproduce each task as well as the number of tries (in case they failed to reproduce the results).

		Intro	Task 1		Task 2		Task 3	
Expert	**Education**	**Time**	**Task**	**Time**	**Task**	**Time**	**Task**	**Time**
Yes	Post-doc	6	B.1	1	B.6	4	B.9	3
Yes	PhD	5	A.1	3	B.6	2	B.7	6
No	PhD	4	B.6	1	B.10	4	B.9	4
No	PhD	4	A.2	2	B.6	2	D.5	4
No	MS	4	A.1	4	B.6	1	D.3	2
No	MS	3	B.6	2	C.1	2	D.4	10
No	MS	6	A.1	2	B.7	3	B.10	3
No	BS	2	A.2	2	D.1	2	D.3	2

Fig. 16. Study results. All times given in minutes.

The results are given in Figure 16 and clearly show that all students were able to reproduce the previously published results in (substantially) less than one hour. Note that all students were also able to reproduce the results on their first try. Thus we assert that using only previously published source code and which dataset was used, other researchers are able to easily reproduce the results.

VI. Related Work

Despite the popularity of Mining Software Repositories (MSR), only a few research groups have attempted to address the problem of mining large-scale software repositories. In this section we discuss some of these efforts and programming languages similar to *Boa*.

A. Mining Software Repositories

Bevan *et al.* [12] proposed a centralized approach in which they define database schemas for metadata and source code in software repositories and such data is downloaded into a centralized database, called *Kenyon*. The data can be accessed from Kenyon via SQL commands with their predefined data schemas. Unlike our infrastructure, which is aimed to support ultra large data in software repositories, Kenyon was not designed for ultra large data with hundred thousands of projects and billions lines of code. Additionally, our language and infrastructure can easily support new metadata from repositories as a newly defined type in the language.

In 2007, Boetticher, Menzies and Ostrand introduced the PROMISE Repository [13], an online data repository for empirical software engineering data, mainly for defect prediction research. They make the repository publicly available and encourage the authors of research papers on defect prediction to upload data. The data in PROMISE are the post-processed data, i.e. the data that were already processed to be suited with each individual research problem in each research paper.

[3] At the start of the study we only had 18 tasks (A.3, B.8, and B.11 missing). For consistency, all participants used the same set of tasks.

For example, the authors of a new bug prediction model using Weka as their machine learning tool would upload the data files in Weka format. This hinders the applicability and usability of the data if other researchers would like to use the original data for a different tool set, a different approach, or even a different problem. PROMISE data is also limited to defect prediction. Additionally, since the data is uploaded for individual research PROMISE potentially contains duplicate data and inconsistencies.

Sourcerer [22] provides an SQL database of metadata and source code on over 18k projects. Queries are performed using standard SQL statements. Thus their approach easily supports joins on the data, where ours does not. However, being built on MapReduce allows easier scalability for our approach. Their approach also does not contain history information (revisions).

Supporting for the reproducibility of research papers published in the MSR area, Gregorio Robles [14] and his team advocated for the construction of open-access data repositories for MSR research. Their goal was to build "a web page with the additional information, most desirably a Sourceforge-like site that acts as a repository for this type of data and tools, and that frees researchers from maintaining infrastructure and links". Their vision is similar to PROMISE but with more general types of data. We focus more on the raw data of open-source projects that can be utilized in any MSR research.

Aiming to improve the scalability and speed of MSR tasks, Hassan *et al.* [15] and Gabel *et al.* [16] use parallel algorithms and infrastructures. They have shown that using map-reduce and other parallel computing infrastructure could achieve that goal. In comparison, they focus only on specific mining tasks (e.g. finding uniqueness and cloned code), while our infrastructure supports a wide range of mining tasks. Additionally, the details of using map-reduce are not exposed to the programmers when using *Boa*.

B. Programming Languages

Martin *et al.* define a program query language (PQL) [23] to allow easily analyzing source code. Their language models programs as certain events, such as the call or return of a method or reading/writing a field, and allow users to write query patterns to match sub-sequences of these events. To match, PQL performs a static analysis that is flow-sensitive and performs a pointer analysis to determine all possible matches to the query. It also provides an online checker that instruments the program and dynamically matches. Each instance of PQL however is limited to matching against a single program and has a limited set of events provided by the language. Our approach is designed to perform queries efficiently against a large corpus of data instead of single programs.

Dean and Ghemawat proposed a computing paradigm called MapReduce [8] in which users easily process large amounts of data in a highly parallel fashion by providing functions for filtering and grouping data, called *mappers*, and additional functions for aggregating the output, called *reducers*. Programs that are heavily data-parallel and written in MapReduce can be executed in parallel on large clusters, without the user worrying about explicitly writing parallel code. Over the years, a large number of languages that directly or indirectly support MapReduce or MapReduce-like paradigms were proposed. Here we discuss some of these languages.

Sawzall [9] is a language developed at Google to ease processing of large datasets, particularly logfiles. The language is intended to run on top of Google's distributed filesystem and map-reduce framework, allowing users to write queries against or process large amounts of log data. Our framework, while syntactically similar to Sawzall, provides several key benefits. First, we provide domain-specific types to ease the writing of software mining tasks. These types represent a lot of cached data and provide convenient ways to access this data, without having to know specifics about how to access code repositories or parse the data contained in them. Second, our framework runs on Hadoop clusters whereas Sawzall only runs on a single machine or on Google's proprietary map-reduce framework.

Apache Pig Latin [17] aims to provide both a procedural style map-reduce framework as well as a more higher-level, declarative style language somewhat similar to standard SQL. Unlike pure map-reduce frameworks or implementations such as Sawzall, Pig Latin provides the ability to easily perform joins on large datasets. The language was also designed to ease the framework's ability to optimize queries. Since our approach is based on Sawzall, we do not directly provide support for joins. Unlike *Boa* however, Pig Latin does not directly provide support for software mining tasks.

Dryad [18] is a framework to allow parallel processing of large-scale data. Dryad programs are expressed as directed, acyclic graphs and thus are more general than standard map-reduce. A high-level procedural language, DryadLINQ [24], is provided that compiles down to Dryad. This language is based on .Net's language integrated query (LINQ) and provides a syntax somewhat similar to a procedural version of SQL and thus is relatively similar to Pig Latin. Also similar to Pig Latin, Dryad does not directly aim to support easing software mining tasks. Microsoft no longer supports Dryad/DryadLINQ.

VII. Future Work and Conclusion

Ultra-large-scale software repositories contain an enormous corpus of software and information about that software. Scientists and engineers alike are interested in analyzing this wealth of information, however systematic extraction of relevant data from these repositories and analysis of such data for testing hypotheses is difficult. In this work, we present *Boa*, a domain-specific language and infrastructure to ease testing MSR-related hypotheses. We implemented *Boa* and provide a web-based interface to *Boa*'s infrastructure. Our evaluation demonstrated that *Boa* substantially reduces programming efforts, thus lowering the barrier to entry. *Boa* also shows drastic improvements in scalability without requiring programmers to explicitly parallelize code. We also demonstrate that experiments conducted using *Boa* are easily reproduced simply by re-running *Boa* programs provided by the previous researchers.

In the future, we plan to support additional version control systems and source repositories. A key challenge in this process will be to reconcile terminological differences between these systems to be able to provide a unified interface.

Acknowledgment

This work was supported in part by NSF grants CCF-11-17937, CCF-10-17334, CCF-10-18600, and CNS-12-23828.

References

[1] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *CSCW*, 1992, pp. 107–114.

[2] J. Lerner and J. Tirole, "Some simple economics of open source," *The Journal of Industrial Economics*, vol. 50, pp. 197–234, 2002.

[3] S. Landau, "Standing the test of time: The data encryption standard," *Notices of the American Mathematical Society*, vol. 47, no. 3, p. 341, March 2000.

[4] S. Goodman, P. Wolcott, and G. Burkhart, *Building on the Basics: An Examination of High-Performance Computing Export Control Policy in the 1990s*. Center for International Security & Cooperation, 1995.

[5] E. Raymond, "The cathedral and the bazaar," *Knowledge, Technology & Policy*, vol. 12, pp. 23–49, 1999.

[6] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *ECOOP*, 2011, pp. 52–78.

[7] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *MSR*, 2007.

[8] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *OSDI*, 2004.

[9] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with Sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, 2005.

[10] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: easy, efficient data-parallel pipelines," in *PLDI*, 2010, pp. 363–375.

[11] H. Rajan, T. N. Nguyen, R. Dyer, and H. A. Nguyen, "Boa website," http://boa.cs.iastate.edu/, 2012.

[12] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating software evolution research with Kenyon," in *ESEC/FSE*, 2005, pp. 177–186.

[13] "Promise 2009," http://promisedata.org/2009/datasets.html.

[14] J. M. González-Barahona and G. Robles, "On the reproducibility of empirical software engineering studies based on data retrieved from development repositories," *Empirical Software Engineering*, vol. 17, no. 1-2, pp. 75–89, 2012.

[15] W. Shang, B. Adams, and A. E. Hassan, "An experience report on scaling tools for mining software repositories using mapreduce," in *ASE*, 2010, pp. 275–284.

[16] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *FSE*, 2010, pp. 147–156.

[17] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *the ACM SIGMOD international conference on Management of data*, 2008, pp. 1099–1110.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *the ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007, pp. 59–72.

[19] Apache Software Foundation, "Hadoop: Open source implementation of MapReduce," http://hadoop.apache.org/, 2012.

[20] A. Urso, "Sizzle: A compiler and runtime for Sawzall, optimized for Hadoop," https://github.com/anthonyu/Sizzle, 2012.

[21] "TIOBE Programming Community Index for July 2012," TIOBE Software BV, Tech. Rep., 2012. [Online]. Available: http://www.tiobe.com/tpci.htm

[22] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, pp. 300–336, April 2009.

[23] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using PQL: a program query language," in *OOPSLA*, 2005, pp. 365–383.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, ï. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language." in *OSDI*, 2008, pp. 1–14.