Boa: Ultra-Large-Scale Software Repository and Source Code Mining

ROBERT DYER, Bowling Green State University
HOAN ANH NGUYEN, Iowa State University
HRIDESH RAJAN, Iowa State University
TIEN N. NGUYEN, Iowa State University

In today's software-centric world, ultra-large-scale software repositories, e.g. SourceForge, GitHub, and Google Code, are the new library of Alexandria. They contain an enormous corpus of software and related information. Scientists and engineers alike are interested in analyzing this wealth of information. However, systematic extraction and analysis of relevant data from these repositories for testing hypotheses is hard, and best left for mining software repository (MSR) experts! Specifically, mining source code yields significant insights into software development artifacts and processes. Unfortunately, mining source code at a large-scale remains a difficult task. Previous approaches had to either limit the scope of the projects studied, limit the scope of the mining task to be more coarse-grained, or sacrifice studying the history of the code. In this paper we address mining source code: a) at a very large scale; b) at a fine-grained level of detail; and c) with full history information. To address these challenges, we present domain-specific language features for source code mining in our language and infrastructure called *Boa*. The goal of *Boa* is to ease testing MSR-related hypotheses. Our evaluation demonstrates that *Boa* substantially reduces programming efforts, thus lowering the barrier to entry. We also show drastic improvements in scalability.

CCS Concepts: •**Software and its engineering** → *Patterns; Concurrent programming structures;*

Additional Key Words and Phrases: Boa; mining software repositories; domain-specific language; scalable; ease of use; lower barrier to entry

ACM Reference Format:
Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen, 2015. Boa: Ultra-Large-Scale Software Repository and Source Code Mining. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (July 2015), 32 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Ultra-large-scale software repositories, e.g. SourceForge (430,000+ projects), GitHub (23,000,000+ projects), and Google Code (250,000+ projects) contain an enormous collection of software and information about software. Assuming only a meager 1K lines of code (LOC) per project, these big-3 repositories amount to at least 25+ 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? [Dourish and Bellotti 1992]";
— "what is the most widely used open source license? [Lerner and Tirole 2002]";

The work described in this article is the revised and extended version of articles in the proceedings of the 35th International Conference on Software Engineering (ICSE'13) [Dyer et al. 2013] and the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE'13) [Dyer et al. 2013b].
Author's addresses: R. Dyer, Department of Computer Science, Bowling Green State University; H. Rajan and H. Nguyen, Department of Computer Science, Iowa State University; T. Nguyen, Department of Electrical and Computer Engineering, Iowa State University.
This work was supported in part by the US National Science Foundation (NSF) under grants CNS-15-13263, CNS-15-12947, CCF-14-23370, CCF-13-49153, CCF-13-20578, TWC-12-23828, CCF-11-17937, CCF-10-17334, and CCF-10-18600.

— "how many projects continue to use DES (considered insecure) encryption standards? [Landau 2000]";
— "how many open source projects have a restricted export control policy? [Goodman et al. 1995]";
— "how many projects on an average start with an existing code base from another project instead of scratch? [Raymond 1999]";
— "how often do practitioners use dynamic features of JavaScript, e.g. `eval`? [Richards et al. 2011]"; and
— "what is the average time to resolve a bug reported as critical? [Weiss et al. 2007]".

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 not replicable 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 [González-Barahona and Robles 2012], because there is no systematic curation. Last but not least, building analysis infrastructure to process ultra-large-scale data efficiently can be very hard [Dean and Ghemawat 2004; Pike et al. 2005; Chambers et al. 2010].

To solve these problems, we 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 are for hardware description. We have implemented *Boa* and provide a web-based interface to *Boa*'s infrastructure [Rajan et al. 2015]. Potential users request access to the system and are typically granted it within a few hours or days. Our eventual goal is to open-source all of the *Boa* infrastructure.

Boa provides domain-specific language features for mining source code. These features are inspired by the rich body of literature on object-oriented visitor patterns [Gamma et al. 1994; Di Falco 2011; Oliveira et al. 2008; Orleans and Lieberherr 2001; Visser 2001]. A key difference from previous work is that we do not require the host language to contain object-oriented features. Our *visitor types* provide a default depth-first search (DFS) traversal strategy, while still maintaining the flexibility to allow custom traversal strategies. Visitor types allow specifying the behavior that executes for a given node type, *before* or *after* visiting the node's children. *Boa* also provides abstractions for dealing with mining of source code history, such as the ability to retrieve specific snapshots based on date. We also show several useful patterns for source code mining that utilize these domain specific language features.

To evaluate *Boa*'s design and effectiveness of its infrastructure we wrote programs to answer 23 different research questions in five different categories: questions related to the use of programming languages, project management, legal, platform/environment, and source code. 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 12). Last but not least, replicating 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 further motivate the need for a new language and infrastructure to perform software mining at an ultra-large scale. Next we present the language (Section 3) and describe its infrastructure (Section 4). Section 5 presents studies of applicability and scalability. Section 7 positions our work in the broader research area and Section 8 concludes.

2. MOTIVATION

Creating experimental infrastructure to analyze the wealth of information available in open source repositories is difficult [Bevan et al. 2005; Promise dataset 2009; González-Barahona and Robles 2012; Shang et al. 2010; Gabel and Su 2010]. Creating an infrastructure that scales well is even harder [Shang et al. 2010; Gabel and Su 2010]. 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.

```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(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");
  }
}
```

Fig. 1. Java program that answers the question "what are the churn rates for all Java projects that use SVN?"

This program assumes that the user has manually downloaded all project metadata, available as JSON files, and SVN repositories from SourceForge, a forge for making open-source projects available [SourceForge 2015]. 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 complexity.

2.1. *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 [Dean and Ghemawat 2004; Pike et al. 2005; Olston et al. 2008; Isard et al. 2007]. 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 2. 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 700k projects, the code on lines 3–6 runs for each.

```
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; lowercase(p.programming_languages[j]) == "java")
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);
```

Fig. 2. Boa program that answers the question “what are the churn rates for all Java projects that use SVN?”

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 new or toy 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.

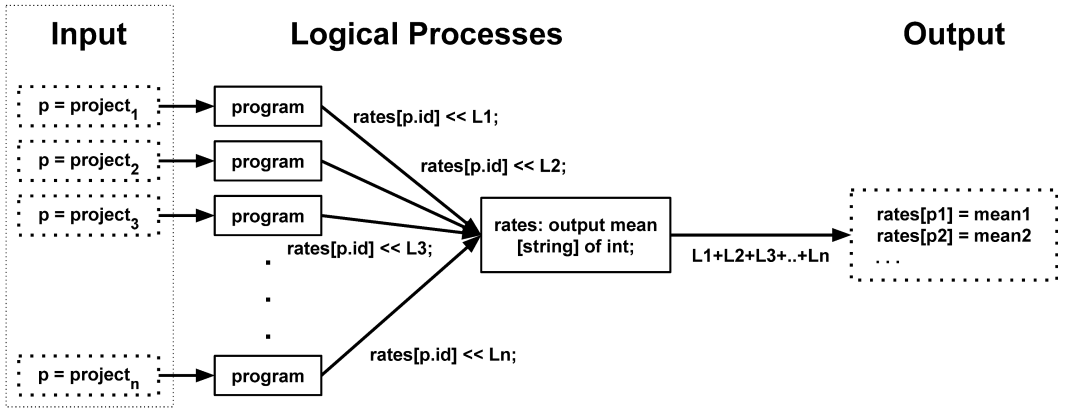

Fig. 3. Overview of the semantic model provided by *Boa* for the query in Figure 2. Each project is a single input and fed to a single process. Each process sends messages to the output process. The output process produces the final result. Each solid box represents a logical process.

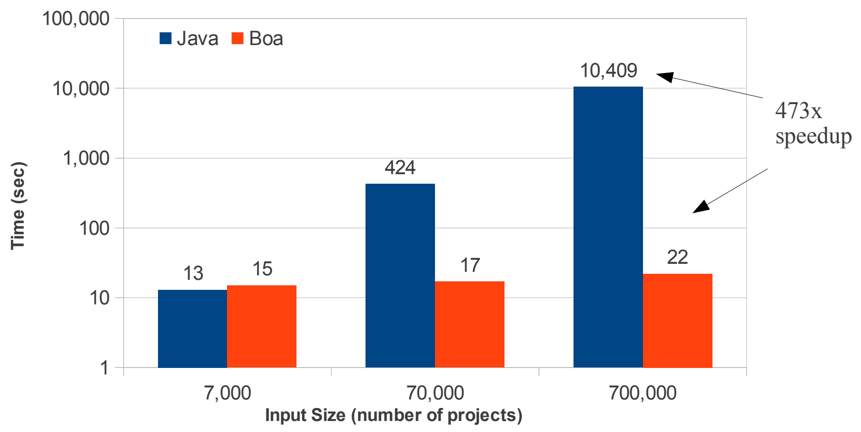

Fig. 4. Performance results for programs in Figures 1 and 2.

Figure 3 shows an example of the semantic model provided by *Boa*. On the left-hand side, each project from the dataset becomes a single input to a process. The program is instantiated once for each input (middle of figure). Each instantiation will process a single project, computing the churn rate for that one project. For each revision in the project, the number of files in that revision is sent to the output process (right-hand side) which aggregates all the values, from each input process, and computes the final result.

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 4. 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 5. 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 2 (see Section 3). This program is fed to our compiler (see Section 4.1) via our web-based interface (see Section 4.2).

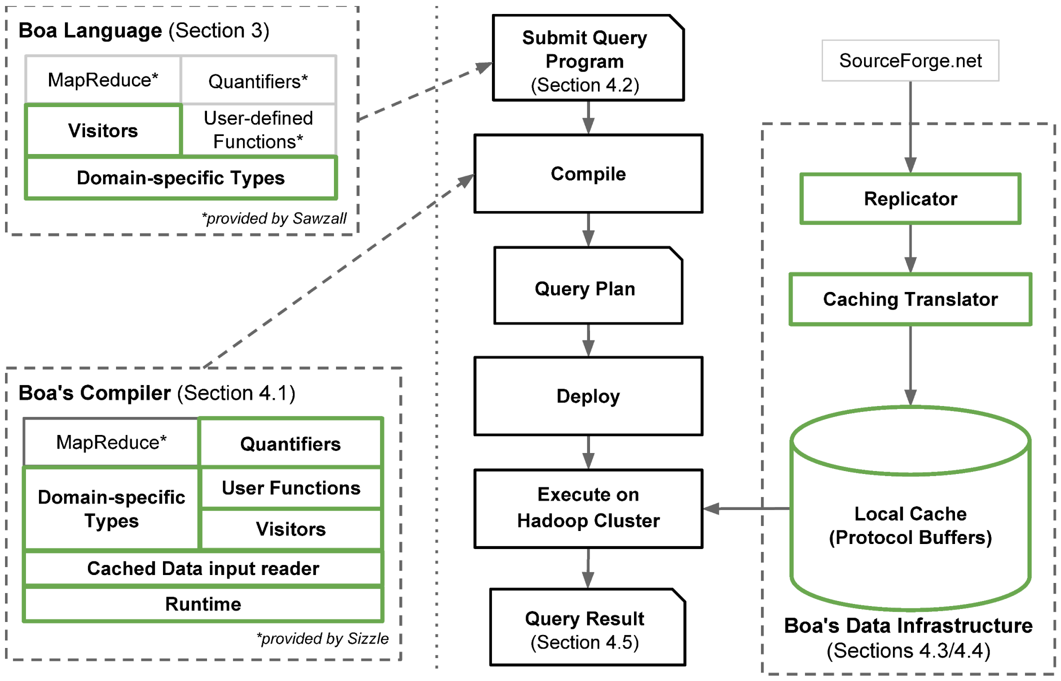

Fig. 5. An Overview of *Boa*'s Infrastructure. New components are marked with green boxes and bold text.

The *Boa* compiler produces a query plan. Our infrastructure then deploys this query plan onto a Hadoop [Apache Software Foundation 2015a] cluster, where it executes. The cluster makes use of a locally cached copy of the source code repositories (see Sections 4.3–4.4) and based on the query plan creates tasks to produce the final query result (see Section 4.5). This is the answer to the user's analysis task. We now describe these components in detail.

3. DESIGN OF THE *BOA* LANGUAGE

The top left portion of Figure 5 shows the main features of the *Boa* language. We have five main kinds of features at the moment: domain-specific types to ease analysis of open source software repository mining, declarative visitors to ease source code mining, MapReduce [Dean and Ghemawat 2004] support for scalable analysis of ultra-large-scale repositories, quantifiers for easily expressing loops, and the ability to define functions.

3.1. Domain-Specific Types in *Boa*

The *Boa* language provides several domain-specific types for mining software repositories. Figure 6 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, . . .
CodeRepository	url, kind, revisions
Revision	id, log, committer, commit_date, files
ChangedFile	name, kind, change
Person	username, real_name, email

Fig. 6. Domain-specific types provided in *Boa* for mining software repositories.

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 `CodeRepository` 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 `ChangedFiles` committed.

Type	**Attributes**
ASTRoot	imports, namespaces
Namespace	name, modifiers, declarations
Declaration	name, kind, modifiers, parents, fields, methods, ...
Type	name, kind
Method	name, modifiers, return_type, statements, ...
Variable	name, modifiers, initializer, variable_type
Statement	kind, condition, expression, statements, ...
Expression	kind, literal, method, is_postfix, ...
Modifier	kind, visibility, other, ...

Fig. 7. Domain-specific types for mining source code.

The types *Boa* provides for representing source code are shown in Figure 7 and include: `ASTRoot`, `Namespace`, `Declaration`, `Method`, `Variable`, `Type`, `Statement`, `Expression`, and `Modifier`. The declaration, type, statement, expression, and modifier types are discriminated types, meaning they actually represent the union of different record structures.

For example, consider the type `Statement`. This type has an attribute `kind`, which is an enumerated value. Based on the kind of statement, specific additional attributes in the record will be set. For example, if the kind is `TYPEDECL` then the `type_decl` attribute is defined. However if the kind is `CATCH` then the `type_decl` is undefined.

Representing these types as discriminated types allows *Boa* to keep the number of types as small as possible. This makes supporting future languages easier by only needing to provide a mapping from the new language to the small set of types in *Boa*. Existing mining tasks would immediately be able to mine source code from the new language.

3.1.1. Mapping Java to Boa's Custom AST. Currently, we have fully mapped the Java language to *Boa*'s schema, attempting to simplify the schema as much as possible. This gives a simple, yet flexible, schema capable of supporting the entire Java language (through Java 7). This allows *Boa* to represent Java source files in the dataset and allows *Boa* programs to query those Java source files.

The top-level symbol in Java's grammar is a `CompilationUnit`. In *Boa*, the top-level type for source code is `ASTRoot`. For each Java `CompilationUnit`, we create one `ASTRoot`. The imports from the `CompilationUnit` directly map to the imports attribute in `ASTRoot`. Everything between the import keyword and the semicolon is stored as a single string in the imports array. We then create one `Namespace` type in *Boa*, which contains the `PackageName` (or an empty string if the default package) and any modifiers.

Next, each Java type is transformed into a `Declaration`. The declaration's `kind` attribute indicates if the type was a class, interface, enum, annotation, etc. Methods are transformed into `Methods` and fields transformed into `Variables`. Finally each statement and expression are transformed.

3.1.2. Extending the AST to Support New Language Features. While *Boa* keeps these types as simple as possible, they are still flexible enough to support more complex language features. As additional support for other source languages is added, if the schema is not capable of directly supporting a particular language feature the `StatementKind` or `ExpressionKind` enumerations can be easily extended. For example, consider the enhanced-for loop in Java:

```
for (String s : iter)
  body;
```

which says to iterate over the expression `iter` and for each string value `s`, run the `body`. *Boa*'s types do not directly contain an `ENHANCEDFOR` kind for this language feature.

Despite this design decision, an enhanced-for statement can be easily represented in *Boa*'s schema without having to extend it. First, *Boa* generates a `Statement` of kind `FOR`. Inside that statement, *Boa* sets `expression` to `iter`. *Boa* also sets the `variable_declaration` for `String s` in the statement. Thus, if a statement of kind `FOR` has its `variable_declaration` attribute set it is a for-each statement. If that attribute is not defined, then it is a standard for-loop.

Using a similar strategy, we plan to support additional languages. If we are unable to map a particular language feature to the existing types and kinds, we can extend them. For example, supporting a null coalescing operator in C# would require extending the ExpressionKind with a new kind. Extending the enumeration is backwards compatible, so previous mining tasks will continue to work as expected.

3.2. Declarative Visitors to Ease Source Code Mining

Users must be able to easily express source code mining tasks. For users who are intimately familiar with compilers and interpreters, the visitor style is well understood. However, other users may find two aspects of visitor-style traversals daunting. First, it generally requires writing a significant amount of boiler-plate code whose length is proportional to the complexity of the programming language being visited. Second, this strategy requires intimate familiarity with the structure of that programming language.

To make source code mining more accessible to all users, we investigated the design of more declarative features for mining source code. In this section, we describe our proposed syntax for writing source code mining tasks. The syntax was inspired by previous language features, such as the before and after visit methods in DJ [Orleans and Lieberherr 2001] and case expressions in Haskell [Jones 2003].

visitor ::= `visitor` `{`*visitClause** `}`
visitClause ::= *beforeClause* | *afterClause*
beforeClause ::= `before`*typeList* `->`*beforeClauseStmt*
afterClause ::= `after`*typeList* `->`*stmt*
typeList ::= _| *identifier* `:`*type* | *type* (, *type*)*
beforeClauseStmt ::= *stmt* | *stopStmt* | `visit` `(`*identifier* `)` `;`
stopStmt ::= `stop`;

Fig. 8. Proposed syntax for easing source code mining.

The new syntax is shown in Figure 8. The top-level syntax for a mining task is a *visitor type*. Visitor types take zero or more *visit clauses*. A visit clause can be a *before* or an *after* clause. During traversal of the tree, a before clause is executed when visiting a node of the specified type. If the default traversal strategy is used, then the node's children will be visited. After all the children are visited, any matching after clause executes.

Before and after clauses take a *type list*. A type list can be a single type with an optional identifier, a list of types, or an underscore wildcard. The underscore wildcard provides default behavior for a visitor clause. This default executes for a node of type T if no other clause specifies T in its type list. Thus, the following code:

```
1 id := visitor {
2   before Project, CodeRepository, Revision -> { }
3   before _ -> counter++;
4 };
```

will execute the clause's body on line 2 when traversing nodes of type `Project`, `CodeRepository`, or `Revision`. When traversing a node of any other type, the default clause's body on line 3 executes. The result of this code is thus a count of all nodes, excluding those of the types listed. Thus we count only the source code AST nodes for a project.

Note that unlike pattern matching and case expressions in functional languages like Haskell, the order of the before and after clauses do not matter. A type may appear in at most one before clause and at most one after clause.

To begin a mining task, users write a `visit` statement:

```
visit(n, v);
```

that has two parts: the node to visit and a visitor. When this statement executes, a traversal starts at the node represented by `n` using visitor `v`.

3.2.1. Supporting Custom Traversals. To allow users the ability to override the default traversal strategy, two additional statements are provided inside `before` clauses. The first is the *stop statement*:

```
stop;
```

which when executed will stop the visitor from traversing the children of the current node. This is useful in cases where the mining task never needs to visit specific types further down the tree, allowing to stop at a certain depth. Note that stop acts similar to a return, so no statements after it are reachable.

If the default traversal is stopped, users may provide a custom traversal of the children with a *visit statement*:

```
visit(child);
```

which says to visit the node's `child` tree once. This statement can be called on any subset of the children and in any order. This also allows for visiting a child more than once, if needed.

Figure 9 illustrates a custom traversal strategy from one of our case studies [Dyer et al. 2014]. This program answers the question *how many fields that use a generic type parameter are declared in each project?* To answer this question, the program declares a single visitor. This visitor looks for `Type` nodes where the name contains a generic type parameter (line 5). This visit clause by itself is not sufficient to answer the question, as generic type parameters might occur in other locations, such as the declaration of a class/interface, method parameters, locals, etc. Instead, a custom traversal strategy (lines 10–34) is needed to ensure only field declarations are included.

The traversal strategy first ensures all fields of `Declaration` are visited (lines 12–13). Since declarations can be nested (e.g. in Java, inside other types and in method declarations) we also must manually traverse to find nested declarations (lines 15–32). Finally, we don't want to visit nodes of type `Expression` or `Modifier` (line 34), as these node types can't possibly contain a field declaration but may contain a `Type` node.

Complex mining tasks can be simplified by using multiple visitors. For example, perhaps we only want to look for certain expressions inside of an if statement's condition. We can write a visitor to find if statements, and then use a second sub-visitor to look for the specific expression by visiting the if statement's children. We could perform this mining task with one visitor, however then we need to have flags set to track if we are in the tree underneath an if statement. Using multiple visitors keeps these two mining tasks separate and avoids using flags to keep it simple.

3.2.2. Mining Snapshots in Time. While our infrastructure contains data for the full revision history of each file, some mining tasks may wish to operate on a single snapshot. We provide several helper functions to ease this use case. For example, the function:

```
p: Project = input;
GenFields: output sum[string] of int;

genVisitor := visitor {
  before t: Type ->
    if (strfind("<", t.name) > -1)
      GenFields[p.id] << 1;

  # traversal strategy ensures we only reach Type
  # if the parent is a Variable, and
  # we only include Variable paths that are fields
  before d: Declaration -> {
  ######## check each field declaration ########
    foreach (i: int; d.fields[i])
      visit(d.fields[i]);

  ########### look for nested types ############
    foreach (i: int; d.methods[i])
      visit(d.methods[i]);
    foreach (i: int; d.nested_declarations[i])
      visit(d.nested_declarations[i]);
    stop;
  }
  before m: Method -> {
    foreach (i: int; m.statements[i])
      visit(m.statements[i]);
    stop;
  }
  before s: Statement -> {
    foreach (i: int; s.statements[i])
      visit(s.statements[i]);
    if (def(s.type_declaration))
      visit(s.type_declaration);
    stop;
  }

  ####### stop at expressions/modifiers ########
  before Expression, Modifier -> stop;
};
visit(p, genVisitor);
```

Fig. 9. Using a custom traversal strategy to find uses of generics in field declarations.

```
getsnapshot(CodeRepository [, time] [, string...])
```

takes a `CodeRepository` as its first argument. It optionally takes a time argument, specifying the time of the snapshot which defaults to the last time in the repository. The function also optionally takes a list of strings. If provided, these strings are used to filter files while generating the snapshot. The file's kind is checked to see if it matches at least one of the patterns specified. For example:

```
getsnapshot(CodeRepository, "SOURCE_JAVA_JLS")
```

says to get the latest snapshot and filter any file that is not a valid Java source file.

A useful pattern is to write a visitor with a before clause for `CodeRepository` that gets a specific snapshot, visits the nodes in the snapshot, and then stops the default traversal:

```
visitor {
  before n: CodeRepository -> {
    snapshot := getsnapshot(n);
    foreach (i: int; def(snapshot[i]))
      visit(snapshot[i]);
    stop;
  }
  ...
}
```

This visitor will visit all code repositories for a project, obtain the last snapshot of the files in that repository, and then visit the source code of those files. This pattern is useful for mining the *current version* of a software repository.

3.2.3. Mining Revision Pairs. Often a mining task might want to locate certain revisions and compare files at that revision to their previous state. For example, our motivating example looks for revisions that fixed bugs and then compares the files at that revision to their previous snapshot. To accomplish this task, one can use the following pattern:

```
1 files: map[string] of ChangedFile;

2 v := visitor {
3   before f: ChangedFile -> {
4     if (def(files[f.name])) {
5       ... # task comparing f and files[f.name]
6     }
7     files[f.name] = f;
8   }
9 };
```

which declares a map of files, indexed by their path. The code on line 4 checks if a previous version of the file was cached. If it was, the code on line 5 executes where `f` refers to the current version of the file being visited and the expression `files[f.name]` refers to the previous version of the file. Finally, the code on line 7 updates the map, storing the current version of the file.

3.2.4. Bringing It All Together. Consider the hypothesis “a large number of bug fixes add checks for `null`”. In this section, we describe a solution to support that hypothesis.

Consider the Boa program in Figure 10, which implements the entire mining task. This program takes a single project as input. It then passes the program’s data tree to a visitor (line 13). This visitor keeps track if the last `Revision` seen was a fixing revision (line 16). When it sees a `ChangedFile` it looks at the current revision’s log message and if it is a fixing revision it will get snapshots of the current file and the previous version of the file and visit their AST nodes (lines 21 and 25).

When visiting the AST nodes for these snapshots, if it encounters a `Statement` of kind `IF` (line 34), it then uses a sub-visitor to check if the statement’s expression contains a null check (lines 35 and 7–12) and increments a counter (line 11). Thus we will know the number of null checks in each snapshot and can compare (line 27) to see if there are more null checks. Note that this analysis is conservative and may not find all fixing revisions that add null checks, as the revision may also *remove* a null check from another location and thus give the same count.

This task illustrates several features mentioned earlier in this section. First, the second visitor shows use of a custom traversal strategy by utilizing a stop statement. Second, it makes use of a sub-visitor (`nullCheckVisitor`). Third, it uses the revision pair pattern to check several versions of a file.

Finally, writing this task required no explicit mention of parallelizing the query. Writing the same task in Hadoop would require a lot of boilerplate code to manually parallelize the task, whereas the Boa version is automatically parallelized.

3.3. MapReduce Support in *Boa*

In MapReduce [Dean and Ghemawat 2004] 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 [Pike et al. 2005], a language designed for analyzing log files. In *Boa*, like Sawzall, users write the mapper functions directly and use built-in

```
# STEP 1 - candidate projects as input
p: Project = input;
results: output collection[string] of string;

fixing := false;
count := 0;
files: map[string] of ChangedFile;

nullCheckVisitor := visitor {
  before e: Expression ->
    if (e.kind == ExpressionKind.EQ || e.kind == ExpressionKind.NEQ)
      exists (i: int; isliteral(e.expressions[i], "null"))
        count++;
};

visit(p, visitor {
  before r: Revision ->
    # STEP 2 - potential revisions that fix bugs
    fixing = isfixingrevision(r.log);

  before f: ChangedFile -> {
    if (fixing && haskey(files, f.name)) {
      count = 0;
      # STEP 3a - check out source from revision
      visit(getast(files[f.name]));
      last := count;

      count = 0;
      # STEP 3b - source from previous revision
      visit(getast(f));

      # STEP 4 - determine if null checks increased
      if (count > last)
        results[p.id] << string(f);
    }
    files[f.name] = f;
    stop;
  }

  before s: Statement ->
    if (s.kind == StatementKind.IF)
      visit(s.expression, nullCheckVisitor);
});
```

Fig. 10. Finding in Boa fixing revisions that add null checks.

aggregators as the reduce function. Users declare output variables, 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 sum, minimum/maximum, mean, etc.

For example, we could declare an output variable `rates` (as shown in Figure 2, line 1). For this output we want to index it by `string`s 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 (see Figure 3). Each process represents a mapper function in the MapReduce program and analyzes a single project's revisions. The output variable `rates` creates a reducer process. The map processes emit the number of changed files for each revision in the project being analyzed. This value is sent to the reducer process, which then aggregates values from all map processes and computes the final mean values.

3.4. Quantifiers in *Boa*

Boa defines the quantifiers `exists`, `foreach`, and `ifall`. Their semantics is similar to *when statements* with quantifiers, as in Sawzall. Quantifiers represent an extremely useful syntactic 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 2, 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.

3.5. 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 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 14.

4. *BOA*'S SUPPORTING INFRASTRUCTURE

The bottom left portion of Figure 5 shows the various parts of the *Boa* compiler and runtime.

4.1. Compiler and Runtime

For our initial implementation, we started with code for the Sizzle [Urso 2013] 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 [Apache Software Foundation 2015a] 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.

4.1.1. 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 (compatible) variables.

4.1.2. 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.

4.1.3. 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.

4.2. 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 compilation 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).

Users also have the option of marking their job as public. This allows anyone (even without a user account on the site) to access details of the job, including the source code query, the job's status, and output of the job. This public access is read-only and provides an archive of the job, suitable for referencing in published research artifacts to ease replicability.

4.3. 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 5 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.

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.

4.4. Storage Strategy

All of the data for a single project is processed inside of one Hadoop map task. This implies that the data for a project must fit in the memory of one map task (which on our cluster, is 1GB). Some projects are extremely large and can not fit their entire object tree in memory at one time (the largest project has over 6.5GB of data). To solve this problem, we split the object tree into a forest of disconnected trees.

Figure 11 shows a portion of an object tree on the left side. This is the sub-tree for one revision of a project, which contains two changed files. To split this tree, we simply turn the `ChangedFiles` into leaves. This produces the forest on the right side of the figure.

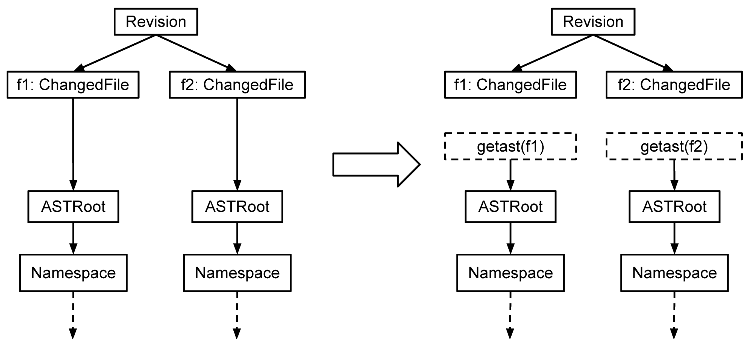

Fig. 11. Splitting an object tree into a forest.

When users wish to access the AST nodes for the changed file `f1` in the language, instead of reading an attribute of the `ChangedFile` users make a call to `getast(f1)`. This call then retrieves and returns that changed file's AST nodes. Once no references exist to any nodes in this sub-tree, they are free to be garbage collected. For most tasks, this solves the problem of fitting a project's data into a map task's process.

Since the AST trees are loaded on demand, we needed a storage strategy that allowed for random access reads. Our first choice was a distributed database named HBase [Apache Software Foundation 2015b], which is an open-source implementation of Google's Bigtable [Chang et al. 2008]. We designed a table format for the AST objects:

Key	**File$_1$**	**File$_2$**	**..**	**File$_n$**
URL1:R1	AST$_1$	AST$_2$	..	...
...	...	...	..	...
URL$_n$:R$_n$	...	...	..	AST$_n$

where each revision is a row in the table, indexed by a unique string containing the repository's URL and the revision number. Each file in that revision is then stored in a column, using the file's path as the column name. This was possible because the design of HBase allows creating columns on demand and empty cells take no space in the filesystem.

This design also allows for easily and incrementally updating the data. As our local cache is updated with new data from the remote repositories, we can simply insert rows for any new revisions.

HBase provides Bloom filters [Bloom 1970] for more efficient random lookups, which we enabled. Despite this optimization, our initial performance tests indicated that reads were much slower than we expected (described in detail in Section 5.3). Thus we designed a second storage strategy, this time using a flat-file datatype called `MapFile`, provided by Hadoop.

A `MapFile` is actually two separate files. The first file is a list of key-value pairs called a `SequenceFile`. This file is sorted by the keys. In this new design, the previous HBase table is essentially linearized into a sorted `SequenceFile`:

Key	**Value**	**Key**	**Value**	**..**	**Key**	**Value**
URL1:R1:F1	AST$_1$	URL1:R1:F2	AST$_2$	..	URL$_n$:R$_n$:F$_n$	AST$_n$

giving each cell a unique key by taking the HBase row key and concatenating the HBase column name.

The `MapFile` data-structure also generates a second file, which is an index. For each block on the filesystem, it will store each block offset of the first file and the first key in each block. A random read becomes finding the block and scanning to find the key. As we show later, this new storage strategy performs substantially better.

Despite the performance benefit, using a `MapFile` comes with a cost of the inability to perform incremental updates to the data. This is a restriction of the underlying distributed filesystem used by Hadoop, which states that files may only be appended. HBase circumvents this restriction by storing updates in memory and occasionally rewriting the underlying stores and merging in the new updates. With a `MapFile` we would have to read and rewrite the entire file for a single, incremental update.

Our final storage strategy thus attempts to take the best of both worlds. First, all data is populated into HBase tables. This provides the easy incremental update of the data. From these tables we then generate a `MapFile`. Generating these files for use as input to mining tasks only takes a few hours and can be routinely scheduled.

4.5. Query Output Format

The output from a Boa program is a text file. The format of that file is described in this section. Consider the output variable declared in Figure 2:

```
rates : output mean[string] of int;
```

which declares an output variable named `rates`. This output variable collects `int` values and computes their `mean`. The variable is indexed by a `string`, meaning that values are grouped by the index strings and for each unique index, a mean is computed.

For this example, the index is a project identifier. We expect to see in the output pairs of all project IDs and a single (mean) value. For example, the output from the program in Figure 2 is:

```
rates[100007] = 4.016949152542373
rates[100009] = 6.583333333333333
rates[100018] = 17.0
rates[100028] = 4.328990228013029
rates[100045] = 7.076923076923077
rates[100050] = 8.276806526806526
rates[100057] = 4.12
rates[100064] = 2.8446697996537225
rates[100081] = 1.0
rates[100083] = 5.2153846153846155
...
```

In this output, each line represents a single project's churn rate. The project's unique identifier is the index (between the brackets) and the churn rate is on the right-hand side. Notice the variable's name (`rates`) appears in the output. This is so if there is more than one output variable, you can distinguish them in the file.

Output lines are also sorted, lexicographically. Sorting is done from left to right by first sorting the output variable name and then by each index.

Finally, if an output variable takes a weight (such as top/bottom and minimum/maximum) then the weight value will show in the output. In this case, the output variable accepts values with weights, groups the output by the value, and then summarizes all the weights for each value. The output shows both the values and the (total) weights. For example, the output for the top-10 programming languages (task A1, Section 5.1) is:

```
counts[] = java, 50692
counts[] = c++, 40934
counts[] = php, 32696
counts[] = c, 30580
counts[] = python, 15352
counts[] = c#, 15305
counts[] = javascript, 12748
counts[] = perl, 9783
counts[] = unix shell, 4379
counts[] = delphi/kylix, 3842
```

In this case, the values are the programming languages and the weights are the number of projects using that language. The output only contains the top-10 highest weighted values.

5. EVALUATION

This section presents our empirical evaluation of the scalability and the usefulness of our language and infrastructure. The dataset used in this section contains all metadata about all SourceForge projects (700k+[1]) and repository metadata for only the Subversion or CVS repositories.

Programs were executed on a Hadoop [Apache Software Foundation 2015a] 1.2.1 install with 1 name node, 1 job tracker node, and 9 compute nodes. The compute nodes have a total of 116 CPU cores and 2GB memory per core. All machines run Ubuntu 12.04LTS. The cluster has been tuned for performance, including setting the maximum number of map tasks for each compute node equal to the number of cores on that node, increasing the VM heap size to 1GB per task, and enabling short-circuit local reads in the distributed filesystem.

5.1. 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 12) that cover a range of different categories. For each task, we implemented a *Boa* program to solve the task. We also implemented pure Java and Hadoop [Apache Software Foundation 2015a] programs to solve the same tasks. The Java programs were written by an expert in mining software repositories and the Hadoop programs written by an expert in mining software repositories and Hadoop. They were 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 and Hadoop versions.

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 12. 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 32–180 lines of code and on average required 70 lines of code. Performing the same tasks in *Boa* required at most 30 lines of code and on average less than 6 lines of code. Thus there were 6–22 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. This was done for fairness, as *Boa* queries also have a local cache of the JSON and SVN data. For the Java programs, there are three distinct groups of running times for programs that finished. 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. Finally note several tasks did

[1]This includes "user" projects, which aren't listed.
[2]Ignores comments and blank lines. http://reasoning.com/downloads.html

	LOC			RTime (sec)		
Task	**Java**	**Boa**	**Diff**	**Java**	**Boa**	**Speedup**
A. Programming Languages						
1. What are the ten most used programming languages?	61	3	20.33x	706	26	27.15x
2. How many projects use more than one programming language?	32	3	10.67x	691	25	27.64x
3. In which year was Java added to SVN projects the most?	89	9	9.89x	4,931	29	170.03x
B. Project Management						
1. How many projects are created each year?	43	2	21.50x	725	25	29.00x
2. How many projects self-classify into each topic provided by SourceForge?	45	3	15.00x	658	27	24.37x
3. How many Java projects using SVN were active in 2011?	66	5	13.20x	4,627	24	192.79x
4. In which year was SVN added to Java projects the most?	107	5	21.40x	1,836	24	76.50x
5. How many revisions are there in all Java projects using SVN?	60	4	15.00x	1,297	26	49.88x
6. How many revisions fix bugs in all Java projects using SVN?	76	5	15.20x	14,213	25	568.52x
7. How many committers are there for each Java project using SVN?	69	5	13.80x	14,220	46	309.13x
8. How many Java projects using SVN does each committer work on?	72	8	9.00x	13,789	27	510.70x
9. What are the churn rates for all Java projects that use SVN?	68	4	17.00x	13,457	30	448.57x
10. How did the no. of commits for Java projects using SVN change over years?	79	5	15.80x	14,062	29	484.90x
11. For all Java projects using SVN, what is the distribution of commit log length?	82	5	16.40x	14,397	27	533.22x
C. Legal						
1. What are the five most used licenses?	63	3	21.00x	673	26	25.88x
2. How many projects use more than one license?	32	3	10.67x	669	24	27.88x
D. Platform/Environment						
1. What are the five most supported operating systems?	61	3	20.33x	639	26	24.58x
2. What are the projects that support multiple operating systems?	33	3	11.00x	723	26	27.81x
3. What are the five most popular databases?	61	3	20.33x	609	25	24.36x
4. What are the projects that support multiple databases?	32	3	10.67x	678	26	26.08x
5. How often is each database used in each programming language?	71	4	17.75x	655	27	24.26x
E. Source Code						
1. How many fixing revisions added a null check?	180	30	6.00x	30,315	484	62.63x
2. What are the number of public methods (NPM), per project and per type?	137	13	10.54x	>24h	753	>114.74x

Fig. 12. Several example mining tasks, with lines of code and execution times (in seconds) for Java and *Boa* programs solving the tasks.

	LOC			RTime (sec)		
Task	**Hadoop**	**Boa**	**Diff**	**Hadoop**	**Boa**	**Speedup**
A. Programming Languages						
1. What are the ten most used programming languages?	88	3	29.33x	24	26	0.92x
2. How many projects use more than one programming language?	43	3	14.33x	26	25	1.04x
3. In which year was Java added to SVN projects the most?	59	9	6.56x	27	29	0.93x
B. Project Management						
1. How many projects are created each year?	46	2	23.00x	25	25	1.00x
2. How many projects self-classify into each topic provided by SourceForge?	44	3	14.67x	25	27	0.93x
3. How many Java projects using SVN were active in 2011?	61	5	12.20x	25	24	1.04x
4. In which year was SVN added to Java projects the most?	71	5	14.20x	22	24	0.92x
5. How many revisions are there in all Java projects using SVN?	53	4	13.25x	24	26	0.92x
6. How many revisions fix bugs in all Java projects using SVN?	69	5	13.80x	24	25	0.96x
7. How many committers are there for each Java project using SVN?	49	5	9.80x	30	46	0.65x
8. How many Java projects using SVN does each committer work on?	48	8	6.00x	24	27	0.89x
9. What are the churn rates for all Java projects that use SVN?	60	4	15.00x	26	30	0.87x
10. How did the no. of commits for Java projects using SVN change over years?	46	5	9.20x	25	29	0.86x
11. For all Java projects using SVN, what is the distribution of commit log length?	46	5	9.20x	25	27	0.93x
C. Legal						
1. What are the five most used licenses?	88	3	29.33x	24	26	0.92x
2. How many projects use more than one license?	43	3	14.33x	24	24	1.00x
D. Platform/Environment						
1. What are the five most supported operating systems?	88	3	29.33x	25	26	0.96x
2. What are the projects that support multiple operating systems?	35	3	11.67x	24	26	0.92x
3. What are the five most popular databases?	88	3	29.33x	24	25	0.96x
4. What are the projects that support multiple databases?	35	3	11.67x	25	26	0.96x
5. How often is each database used in each programming language?	46	4	11.50x	26	27	0.96x
E. Source Code						
1. How many fixing revisions added a null check?	226	30	7.53x	496	484	1.02x
2. What are the number of public methods (NPM), per project and per type?	220	13	16.92x	749	753	0.99x

Fig. 13. The same example mining tasks, with lines of code and execution times (in seconds) for Hadoop and *Boa* programs solving the tasks.

not even finish within 24 hours (all of E). The *Boa* programs run in considerably less time. We see minimum speedups of 24 times but in the best case the *Boa* program solves the task over 569 times faster!

Research Question 2: *Do Boa's abstractions help researchers beyond general-purpose distributed computing frameworks?*

Comparing the Java and *Boa* versions shows a clear advantage to *Boa*, in terms of both lines of code and execution time. These Java versions are what most researchers would likely produce if trying to answer those questions. Some researchers might take the next step of trying to parallelize their analyses using a general-purpose distributed computing framework, such as Hadoop.

To answer this question, we created optimized Hadoop versions of each task. This step re-uses the exact same input data that *Boa* uses. As such, unlike the Java versions that required many libraries for processing JSON, SVN, etc, the Hadoop versions benefited from the pre-processing we did for *Boa*.

The results, shown in Figure 13, show that the performance of the hand optimized Hadoop versions is about on par with that of *Boa*, with *Boa* being on average 2% slower then the hand optimized Hadoop versions. The Hadoop versions have on average 13 times more lines of code more compared to the *Boa* versions. This shows that users can easily produce fast, parallel code using *Boa*, but with many fewer lines of code and without having to learn how to write Hadoop programs.

5.1.1. Detailed Examples. Figures 14–17 show four interesting *Boa* programs used to solve some of the tasks. These programs highlight several useful features of the language.

```
1  counts: output sum[int] of int;
2  p: Project = input;

4  HasJavaFile := function(rev: Revision): bool {
5    exists (i: int; match(`.java$`, rev.files[i].name))
6      return true;
7    return false;
8  }

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

Figure 14 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 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.

```
counts: output sum of int;
p: Project = input;

exists (i: int; match(`^java$`, lowercase(p.programming_languages[i])))
  foreach (j: int; p.code_repositories[j].url.kind == RepositoryKind.SVN)
    foreach (k: int;isfixingrevision(p.code_repositories[j].revisions[k].log))
      counts << 1;
```

Fig. 15. Task B.6: Querying number of bug-fixing revisions in Java projects using SVN.

Figure 15 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;

4 foreach (i: int; def(p.licenses[i]))
5   counts << p.licenses[i] weight 1;
```

Fig. 16. Task C.1: Querying the five most used licenses.

Figure 16 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;

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

Figure 17 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.

5.1.2. Results Analysis. We also show some interesting and potentially useful results from four of the tasks. For example, Figure 18 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 [BV 2012]. 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.

The results of Task B.7 are shown in Figure 19. 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 20 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!

Another interesting result came from Task B.11 and is shown in Figure 21. 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 phenomenon 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.

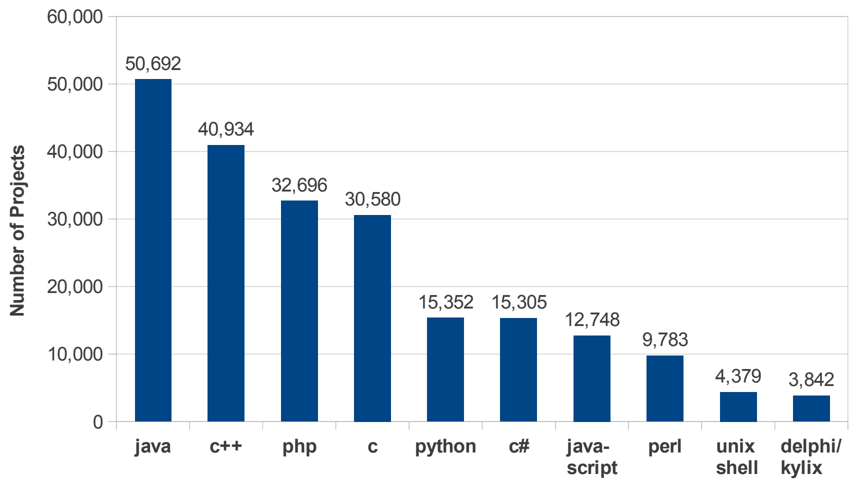

Fig. 18. Task A.1: Popularity of programming languages on SourceForge.

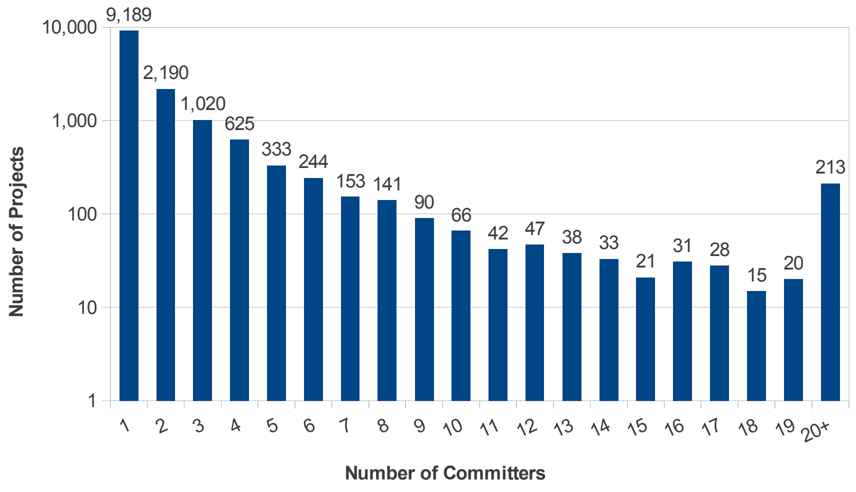

Fig. 19. Task B.7: number of committers in each Java project using SVN. NOTE: y-axis is in logarithmic scale.

5.2. 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 3: *Does our approach scale to the size of the cluster?*

To answer this question, we run one sample program from each category listed in Figure 12 using our SourceForge.net dataset. We fix the size of the input to 700k projects and vary the number of available map slots in the system from 1–116 (note: our current cluster only has 116 cores). Figure 22 shows the results of this analysis where each group represents one of the sample programs, the y-axis (in logarithmic scale) 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.

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.

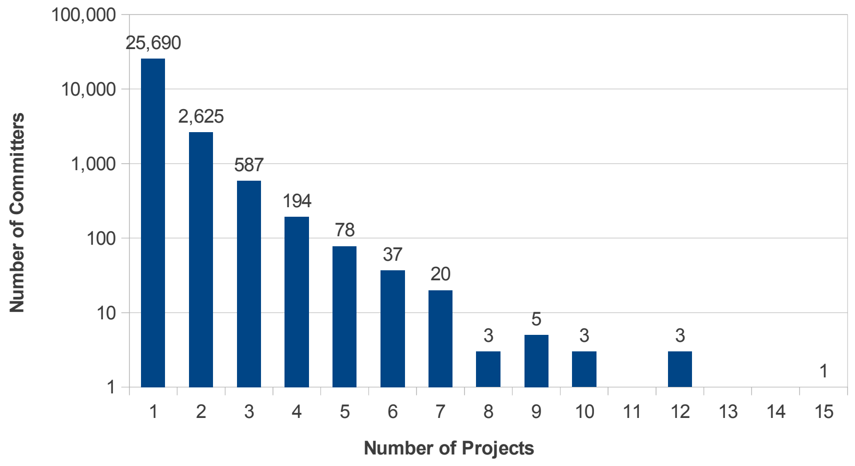

Fig. 20. Task B.8: number of Java projects each SVN committer works on. NOTE: y-axis is in logarithmic scale.

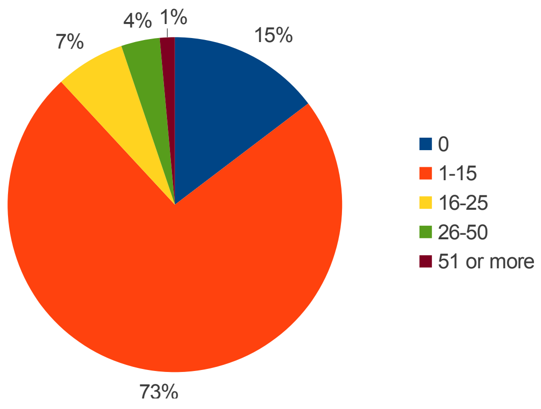

Fig. 21. Task B.11: number of words in SVN commit logs for Java projects.

Note that with our current input size of 700k projects, the maximum number of map slots needed for programs not analyzing source code is 34. 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 34 map slots. We do see scaling past that for the last program, which analyzes source code.

Research Question 4: *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 (7k, 70k, and 700k projects). The results for all tasks in Figure 12 are shown in Figure 23. 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 (7k) 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

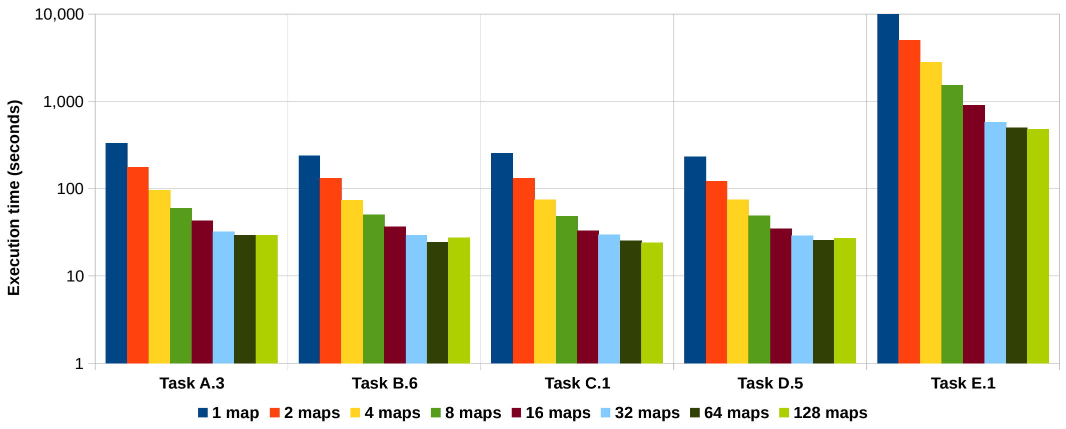

Fig. 22. Scalability of sample programs. Y-axis is total time taken. X-axis is the number of available map slots in the cluster. NOTE: y-axis is in logarithmic scale.

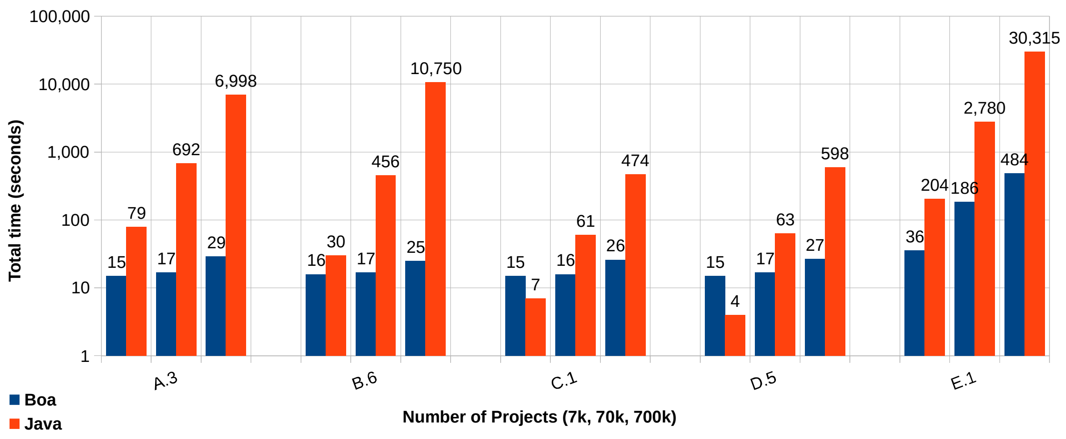

Fig. 23. Scalability of input. Y-axis is total time taken. X-axis is the size of the input in number of projects. NOTE: y-axis is in logarithmic scale and the chart should be interpreted as grouped, not stacked.

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 2–10x as long. This shows that the *Boa* infrastructure scales well as the input size increases.

5.3. Storage Strategy Evaluation

As we mentioned earlier, storing the vast amount of data analyzed by Boa is a non-trivial task. In this section we evaluate the possible storage strategies. For these experiments, the cluster was configured with HBase 0.94.5. Each compute node in the cluster is an HBase region server and the name node master also doubles as the HBase master.

Figure 24 shows the size of the dataset used in our evaluation. This dataset contains project metadata and source code repositories cloned from SourceForge [SourceForge 2015]. While the dataset itself contains metadata on over 700k projects, for the purposes of this section we only look at projects that have at least one valid Java source file. This leaves over 31k projects with over 4 million revisions, 28 million snapshots of Java source files, and over 18 billion AST nodes.

Metric	Total	Mean	Max	Min
Projects	31,432	-	-	-
Revisions	4,298,309	136.75	47,384	1
Java snapshots	28,747,948	6.69	16,062	1
AST nodes	18,323,905,323	637.40	1,072,343	1

Fig. 24. Size of the dataset used for evaluation.

Research Question 5: *What is the best storage strategy for map input data and AST data?*
To answer this question, we evaluate the performance of four different storage strategies. As previously mentioned, due to memory constraints in the map tasks, we needed to split each project's metadata into a forest of trees. This splitting resulted in two different read patterns in our system: sequential reading of the project metadata for use as input to the map tasks and random access reads to mine the ASTs of individual files.

For each read pattern we have a choice of where to store the data, either storing them in a flat file or creating a table in HBase. For input to map tasks we either use a flat `SequenceFile` or an HBase table where rows are projects and columns are their metadata. For reading ASTs, we either use a flat `MapFile` or an HBase table where rows are a single revision and columns are ASTs, one per file in the revision, as was described earlier.

We ran a sample of four mining tasks, including the motivating example (Task E.1), two tasks written for our other study [Dyer et al. 2014] (AnnotUse and SafeVarargs), and a task to reproduce another group's study [Grechanik et al. 2010] (Treasure). For each task, we ran on four different storage strategies:

(1) **HBase+MapFile** represents using HBase for map task input and a `MapFile` for ASTs.
(2) **HBase+HBase** represents using HBase for both map task input and ASTs.
(3) **Seq+MapFile** represents using a `SequenceFile` as map input and `MapFile` for ASTs.
(4) **Seq+HBase** uses a `SequenceFile` for map input and HBase for ASTs.

The results are shown in Figure 25 and are normalized to the first strategy, HBase+MapFile, where each bar represents the geometric mean of five runs. The results clearly show that the first strategy (Seq+MapFile) performs the best. The results also show that using HBase for random access to read the ASTs is substantially slower than using a `MapFile`. For insights into why this is the case, we present two figures that were taken from the cluster's monitoring framework.

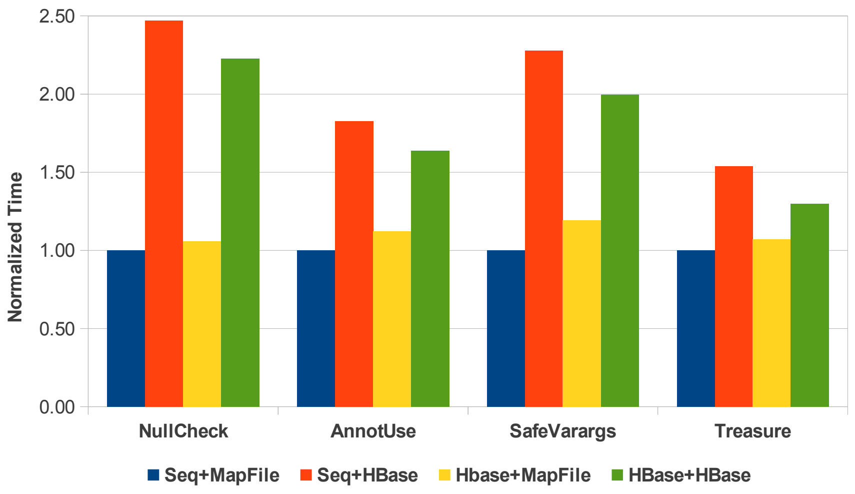

Fig. 25. Performance comparison of MapFile and HBase stores. Results normalized to Seq+MapFile. Smaller is better.

Figure 26 shows the network utilization on the cluster while running two programs. The first program used a `MapFile` for ASTs and the second program used HBase. Both programs used HBase for their map input. As can be clearly seen in the graph, the `MapFile` version has very little network utilization. This is because the data is replicated to each compute node and entirely local for each map task. In contrast, the HBase version clearly shows a lot of network utilization due to the fact that data must be read from HBase's daemons, which are not guaranteed to be local to the map tasks. In fact, even if the data that HBase reads is actually replicated on the local machine, if the daemon controlling that data is on a remote machine then the data must be read remotely. This is part of HBase's architecture and can not be avoided.

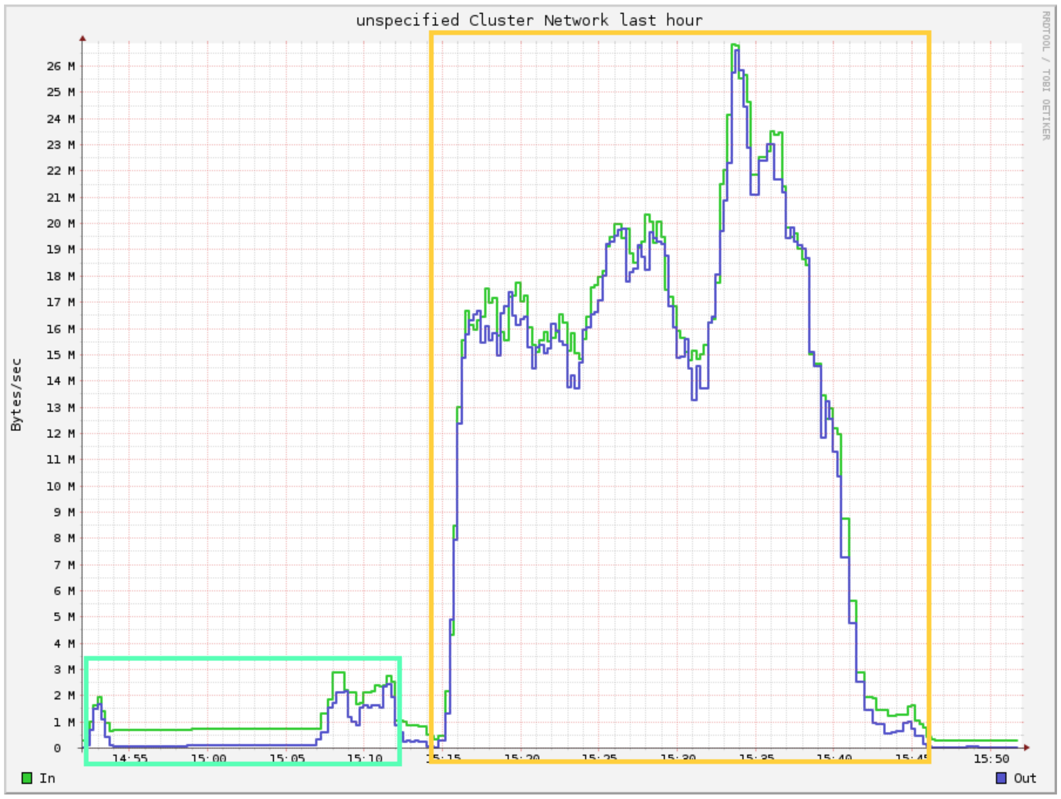

Fig. 26. Network utilization. Note the minimal use by the MapFile store (left) compared to the HBase store (right).

Figure 27 shows the CPU usage across the cluster for the same time-frame. Notice how much higher the CPU use is for the `MapFile` based version. The CPU use for the HBase version is much lower, as the CPU must wait for data to arrive from other nodes. This results in an overall longer running time, as was shown in Figure 25.

In summary, our performance evaluation clearly demonstrates the need to use a `MapFile` for the random access to ASTs. It also demonstrates that using a `SequenceFile` for sequential reads of project metadata is superior. Based on this information, *Boa* now uses HBase tables when processing and storing the data and from those tables generates the flat-files for use when querying.

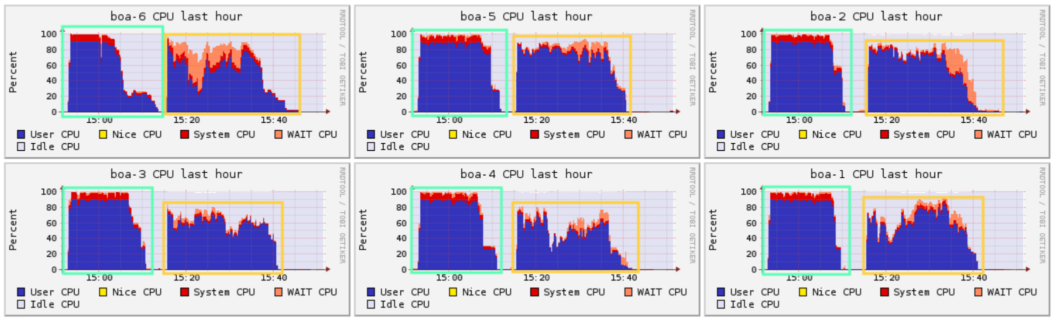

Fig. 27. CPU usage across cluster. Left-most group used the MapFile store. Right-most group used the HBase store.

6. DISCUSSION

6.1. Debugging and Validating Boa Queries

Boa provides different sizes of data for the current dataset: small, medium, and large (where small and medium are random samples 1% and 10% of the full dataset). This allows users to help debug their code faster by selecting less data to process. It can also aid in validation of the results.

Currently, validating the results of a *Boa* query requires some manual effort. The approach we used in previous papers [Dyer et al. 2014; Dyer et al. 2013b] is to look at outliers or randomly sample the results and verify against the live data directly on SourceForge's website. To do this, we augment the program to also output the project name and revision and file path in the version control system. We can then manually inspect the source code to verify the results.

6.2. Analysis Pipelines in Boa

The output of a *Boa* query is a text file (see Section 4.5). *Boa* currently does not allow using the output of one query as input to a second *Boa* query. Instead, the output is structured so that it is easily parseable and can be used as the input to other analysis tasks. For example, in a previous study on the use of Java language features [Dyer et al. 2014], we used *Boa* queries to generate raw indicators and then post-processed the output (using Java) to perform a time series analysis on it. This can easily be accomplished using the client API provided that allows executing queries from Java or C#.

6.3. Extending Boa with Additional Data

While *Boa*'s current dataset contains data from SourceForge, there are a several limitations to the dataset. First, for source code data we only currently support Java source files. Extending *Boa* to support additional source languages requires two things: 1) a robust parser for the newly added source language, capable of gracefully handling different versions of the language as well as erroneous code; and 2) a mapping from the language's AST into *Boa*'s custom AST. In general, we believe the first requirement is the most difficult (for object-oriented and procedural languages, #2 should be relatively straight-forward). Our plan is to investigate adding the more popular languages first.

The second limitation for the dataset is support for version control systems (VCSes). Currently, *Boa* only supports CVS and SVN (and CVS support is provided by first converting the repositories into SVN). Adding support for additional VCSes, such as Git, requires additional engineering effort on the back-end processing scripts to programmatically access the repositories and convert the data into the types provided by *Boa*. We have already begun this process for Git and expect to be able to support most VCSes in the future.

The last limitation for the dataset is the lack of additional artifacts from the forges, such as bug reports, forum posts, social data, etc. Adding these additional pieces of data similarly requires providing the proper types in *Boa*, a mapping from the raw data into those types, and additional engineering effort on the back-end to process the data and perform the conversion.

7. 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*.

7.1. Mining Software Repositories

Bevan *et al.* [Bevan et al. 2005] 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 [Promise dataset 2009], 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. 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 [Linstead et al. 2009] 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, González-Barahona and Robles [González-Barahona and Robles 2012] 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.* [Shang et al. 2010] and Gabel *et al.* [Gabel and Su 2010] 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*.

Hindle and German propose SCQL [Hindle and German 2005], which is a query language for source control repositories. The query language is a temporal logic-based language that queries their general model of source control repositories. As such, temporal based queries (e.g., all files before/after some condition) should be simpler to express than in Boa, which lacks direct support for such temporal queries. Their example implementation only contains data for five projects and may not scale as easily as Boa's.

The Black Duck OpenHub (formerly Ohloh) [Black Duck Software 2015] is a website containing statistics on a large number of open source software projects, regardless of where that project is hosted. The website allows users to search for projects, search for contributors, and search source code. Unlike *Boa*, the granularity of the code search does not currently include statements or expressions.

GHTorrent [Gousios 2013; Gousios and Spinellis 2012] is a website and dataset for querying the event stream on GitHub. Every two months, the collected data is released and available for download. Data for certain projects extends back to 2008. The live data can also be queried online via either MySQL or MongoDB queries. Alitheia Core [Gousios and Spinellis 2009b; Gousios and Spinellis 2009a] provides a highly extensible framework for analyzing software product and process metrics on a large database of open source projects' source code, bug records and mailing lists. Researchers write Java programs using built-in plug-ins and/or creating new plug-ins to compute their desired metrics. Alitheia Core also abstracts raw data into object-relational entities and includes source code revisions as in our framework. However, the main purpose of GHTorrent and Alitheia Core is different from *Boa*. While Alitheia Core focuses on software metrics and GHTorrent focuses on event streams, *Boa* provides fine-grained program elements, i.e. AST nodes and mechanisms for source code traversal.

Flossmetrics [Herraiz et al. 2009] runs analysis tools on a large set of software repositories and publishes various metrics on the data. The metric data is then made available to researchers for further analysis. While both projects aim to provide a dataset for researchers, one of *Boa*'s main goals is to allow easy and efficient querying of that data via a procedural language that maps to a map/reduce architecture.

The SourceForge Research Data Archive (SRDA) [Gao et al. 2007] is a shared archive of monthly dumps directly from SourceForge. The dumps can be downloaded in SQL format or queried online. The archive contains most of the data visible on the website, but lacks the source code repositories.

7.2. Metadata Models

Similar to our goals of providing a language-independent model for mining source code, FAMIX [Tichelaar et al. 2000] is a model for object-oriented languages that is language-agnostic. The model provides various entities and relationships between them. We did not use FAMIX and provided our own AST model as FAMIX does not currently support fine-grained expressions. Our model however lacks relationships and leaves it up to the user to link entities.

The M3 [Izmaylova et al. 2013] source code model is a part of Rascal [Klint et al. 2009]'s standard library. M3 models source code via two different layers. While M3's AST layer is meant to be language-specific, *Boa*'s AST is meant to be language-agnostic. M3 does provide a more abstract relational layer. M3 does not model project or repository metadata.

7.3. Programming Languages

Martin *et al.* define a program query language (PQL) [Martin et al. 2005] 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 [Dean and Ghemawat 2004] 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 [Pike et al. 2005] 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 [Olston et al. 2008] 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 [Isard et al. 2007] 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 [Yu et al. 2008], 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.

8. 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.

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. We also plan to support semantic differencing [Laski and Szermer 1992; Gall et al. 2009] of ChangedFiles to allow easily determining changes in each revision of a file. We would also like to investigate the ability to build analysis pipelines in *Boa*, similar to structuring queries as general directed acyclic graphs as is possible in frameworks like Dryad [Isard et al. 2007].

ACKNOWLEDGMENTS

Authors would like to thank the anonymous TOSEM, ICSE 2013, and GPCE 2013 reviewers for valuable comments on this work. This work significantly benefited from comments obtained during talks and informal conversations at both of these conferences. Authors would also like to express our deepest gratitude for early users and supporters of the Boa infrastructure including but not limited to: Bram Adams, Jonathan Aldrich, Gogul Balakrishnan, Don Batory, Christoph Bockisch, Paulo Borba, Yuanfang Cai, Carl Chang, Myra Cohen, Prem Devanbu, Stephan Diehl, Jamie Douglass, Harald C. Gall, Alessandro Garcia, Jeff Gray, William Griswold, Ahmed Hassan, Hadi Hemmati, Katsuro Inoue, Miryung Kim, Tim Klinger, Doug Lea, Gary Leavens, David Lo, Yuheng Long, Christina Lopes, Hidehiko Masuhara, Massimiliano di Penta, Denys Poshyvanyk, Awais Rashid, Martin Rinard, Gregg Rothermel, Eli Tilevich, Cesare Tinelli, Jan Vitek, Hongyu Zhang, Thomas Zimmermann. Hridesh Rajan would like to thank the audience for the 2nd Workshop on Assessment of Contemporary Modularization Techniques (ACoM 2008) for valuable comments on his position that mining software repositories is an important path forward for evaluating SE properties of PL designs [Rajan 2008], which eventually led to the work on the Boa infrastructure.

Robert Dyer and Hridesh Rajan would like to thank the anonymous reviewers of AOSD 2012 and TAOSD for valuable comments on empirical work that inspired Boa [Dyer et al. 2013a; Dyer et al. 2012].

This work was supported in part by the US National Science Foundation (NSF) under grants CNS-15-13263, CNS-15-12947, CCF-14-23370, CCF-13-49153, CCF-13-20578, TWC-12-23828, CCF-11-17937, CCF-10-17334, and CCF-10-18600.

REFERENCES

Apache Software Foundation. 2015a. Hadoop: open source implementation of MapReduce. http://hadoop.apache.org/. (2015).

Apache Software Foundation. 2015b. HBase: open source implementation of Bigtable. http://hbase.apache.org/. (2015).

Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. 2005. Facilitating software evolution research with Kenyon. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering (ESEC/FSE)*. 177–186.

Black Duck Software. 2015. Black Duck Open HUB. https://www.openhub.net/. (2015).

Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (July 1970), 422–426.

TIOBE Software BV. 2012. *TIOBE Programming Community Index for July 2012*. Technical Report. TIOBE Software BV. http://www.tiobe.com/tpci.htm

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

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2008. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computer Systems* 26, 2, Article 4 (June 2008), 26 pages.

Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In *Proceedings of the 4th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 107–113.

Robert Di Falco. 2011. Hierarchical Visitor Pattern, C2 Pattern Repository. http://c2.com/cgi/wiki?HierarchicalVisitorPattern. (2011).

Paul Dourish and Victoria Bellotti. 1992. Awareness and coordination in shared workspaces. In *Proceedings of the ACM conference on Computer-Supported Cooperative Work (CSCW)*. 107–114.

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 422–431.

Robert Dyer, Hridesh Rajan, and Yuanfang Cai. 2012. An Exploratory Study of the Design Impact of Language Features for Aspect-oriented Interfaces. In *AOSD '12: 11th International Conference on Aspect-Oriented Software Development*.

Robert Dyer, Hridesh Rajan, and Yuanfang Cai. 2013a. Language Features for Software Evolution and Aspect-oriented Interfaces: An Exploratory Study. *Transactions on Aspect-Oriented Software Development (TAOSD): Special issue, best papers of AOSD 2012* 10 (2013), 148–183.

Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *36th International Conference on Software Engineering (ICSE'14)*. 779–790.

Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. 2013b. Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. 23–32.

Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE)*. 147–156.

Harald C. Gall, Beat Fluri, and Martin Pinzger. 2009. Change Analysis with Evolizer and ChangeDistiller. *IEEE Softw.* 26, 1 (2009), 26–33.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Yongqin Gao, Matthew Van Antwerp, Scott Christley, and Greg Madey. 2007. A Research Collaboratory for Open Source Software Research. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS '07)*. IEEE Computer Society, Washington, DC, USA, 4–.

Jesús M. González-Barahona and Gregorio Robles. 2012. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering* 17, 1-2 (2012), 75–89.

Seymour Goodman, Peter Wolcott, and Grey Burkhart. 1995. *Building on the basics: an examination of high-performance computing export control policy in the 1990s*. Center for International Security & Cooperation.

Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, 233–236.

Georgios Gousios and Diomidis Spinellis. 2009a. Alitheia Core: An Extensible Software Quality Monitoring Platform. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, 579–582.

Georgios Gousios and Diomidis Spinellis. 2009b. A platform for software engineering research. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR'09)*. 31–40.

Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's Data from a Firehose. In *MSR '12: Proceedings of the 9th Working Conference on Mining Software Repositories*. IEEE, 12–21.

Mark Grechanik, Collin McMillan, Luca DeFerrari, Marco Comi, Stefano Crespi, Denys Poshyvanyk, Chen Fu, Qing Xie, and Carlo Ghezzi. 2010. An empirical investigation into a large-scale Java open source code repository. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 11:1–11:10.

Israel Herraiz, Daniel Izquierdo-Cortazar, and Francisco Rivas-Hernández. 2009. FLOSSMetrics: Free/Libre/Open Source Software Metrics. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering (CSMR '09)*. IEEE Computer Society, Washington, DC, USA, 281–284.

Abram Hindle and Daniel M. German. 2005. SCQL: a formal model and a query language for source control repositories. In *Proceedings of the 2005 international workshop on Mining Software Repositories (MSR)*. 1–5.

Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *the ACM SIGOPS/EuroSys European Conference on Computer Systems*. 59–72.

Anastasia Izmaylova, Paul Klint, Ashim Shahi, and Jurgen J. Vinju. 2013. M3: An Open Model for Measuring Code Artifacts. *CoRR* abs/1312.1188 (2013).

Simon Peyton Jones. 2003. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. IEEE Computer Society, Washington, DC, USA, 168–177.

Susan Landau. 2000. Standing the test of time: the data encryption standard. *Notices of the American Mathematical Society* 47, 3 (March 2000), 341.

J. Laski and W. Szermer. 1992. Identification of program modifications and its applications in software maintenance. In *Software Maintenance, 1992. Proceerdings., Conference on*. 282–290.

J. Lerner and J. Tirole. 2002. Some simple economics of open source. *The Journal of Industrial Economics* 50 (2002), 197–234. DOI:http://dx.doi.org/10.1111/1467-6451.00174

Erik Linstead, Sushil Bajracharya, Trung Ngo, Paul Rigor, Cristina Lopes, and Pierre Baldi. 2009. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 18 (April 2009), 300–336. Issue 2.

Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 365–383.

Bruno C. d. S. Oliveira, Meng Wang, and Jeremy Gibbons. 2008. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA'08: 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 439–456.

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

Doug Orleans and Karl J. Lieberherr. 2001. DJ: Dynamic Adaptive Programming in Java. In *REFLECTION'01: 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. 73–80.

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

Promise dataset 2009. Promise 2009. http://promisedata.org/2009/datasets.html. (2009).

Hridesh Rajan. 2008. Mining Software Repositories for Evaluating Software Engineering Properties of Language Designs. In *2nd Workshop on Assessment of Contemporary Modularization Techniques (ACoM.08)*.

Hridesh Rajan, Tien N. Nguyen, Robert Dyer, and Hoan Anh Nguyen. 2015. Boa website. http://boa.cs.iastate.edu/. (2015).

Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12 (1999), 23–49. Issue 3.

Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The eval that men do: a large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*. 52–78.

Weiyi Shang, Bram Adams, and Ahmed E. Hassan. 2010. An experience report on scaling tools for mining software repositories using MapReduce. In *Proceedings of the IEEE/ACM international conference on Automated Software Engineering (ASE)*. 275–284.

SourceForge. 2015. SourceForge website. http://sourceforge.net/. (2015).

Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. 2000. FAMIX and XMI. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00) (WCRE '00)*. IEEE Computer Society, Washington, DC, USA, 296–.

Anthony Urso. 2013. Sizzle: a compiler and runtime for Sawzall, optimized for Hadoop. https://github.com/anthonyu/Sizzle. (2013).

Joost Visser. 2001. Visitor combination and traversal control. In *OOPSLA'01: 16th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. 270–282.

Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How long will it take to fix this bug?. In *Proceedings of the 4th international workshop on Mining Software Repositories (MSR)*.

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. 2008. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language.. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*. 1–14.

Received June 2014; revised June 2015; accepted July 2015