Bringing ultra-large-scale software repository mining to the masses with Boa

by

Robert Dyer

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Hridesh Rajan, Major Professor

Samik Basu

Vasant Honovar

Robyn Lutz

Tien N. Nguyen

Iowa State University

Ames, Iowa

2013

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

Throughout my entire research career Hridesh Rajan was there supporting me, encouraging me, and guiding me along the right path. When he gave advice, I listened closely. He knew exactly what I was capable of and never accepted anything less than my best. He has shown incredible patience over these many years and always made himself available to me. I aspire to one day be as excellent a mentor as him.

I may have only met Tien Nguyen one and a half years ago, but in that short time he has become like a second advisor. His enthusiasm for research is unparalleled and a constant inspiration for me. He has also co-authored all of the work this dissertation builds on. His insights and suggestions directly impacted the success of this project and I am truly grateful for the opportunity to work with him.

This dissertation would not have been possible without the hard work and dedication of my colleague Hoan Nguyen. Hoan is one of the hardest working people I know. I could bounce an idea off him in a meeting on a Wednesday, and have empirically validated results to look at by Friday - truly impressive. His attention to detail and insights into this work are one of the key reasons it was so successful. With his help building the infrastructure, we went from problem formation to paper submission in only a single summer!

As an undergradate, I took several programming language courses from Gary Leavens (who is now at UCF). His courses showed me just how little I actually knew about programming language theory and inspired me to continue my education. Whether he realizes it or not, he was directly responsible for me applying to grad school.

I'd also like to thank my colleagues in the Laboratory for Software design: Mehdi Bagherzadeh, Youssef Hanna, Yuheng Long, Sean Mooney, Harish Narayanappa, Rakesh Setty, and Tyler Sondag. Many have co-authored papers with me, but all have provided wonderful feedback and discussion on my ideas over the years as well as a tremendous amount of moral support and comradery.

My family has been extremely supportive over these many years. I could never have made it this

far without the love and support of my parents Bill and Lois. Words can't begin to express my level of gratitude.

My research over the past few years and specifically the research described in this dissertation was supported in part by grants from the US National Science Foundation (NSF) under grants CCF-13-49153, CCF-10-17334, and CNS-07-09217. These grants have ensured I have a warm home and plenty of food for the past few years.

I am sure there are numerous other people and organizations who have helped me over the years, and while your names may not be on this list I still extend my most heartfelt gratitude to you as well.

The majority of content of this dissertation can be found in prior publications. Chapters 1–3 and Sections 5.2–5.3, 5.7, and 7.1–7.3 are based on our ICSE publication introducing *Boa* [31]. Sections 2.5, 3.1.4, 5.8, 6.2, and 7.5 are based on our GPCE publication describing the visitor syntax [33]. Sections 4.3 and 5.6 are vastly extended versions of my student research work on Task Fusion [30]. Sections 6.1 and 7.6 are based on our technical report of our Java feature study [32], which is also currently under submission.

ABSTRACT

Mining software repositories provides developers and researchers a chance to learn from previous development activities and apply that knowledge to the future. Ultra-large-scale open source repositories (e.g., SourceForge with 350,000+ projects, GitHub with 250,000+ projects, and Google Code with 250,000+ projects) provide an extremely large corpus to perform such mining tasks on. This large corpus allows researchers the opportunity to test new mining techniques and empirically validate new approaches on real-world data. However, the barrier to entry is often extremely high. Researchers interested in mining must know a large number of techniques, languages, tools, etc, each of which is often complex. Additionally, performing mining at the scale proposed above adds additional complexity and often is difficult to achieve.

The *Boa* language and infrastructure was developed to solve these problems. We provide users a domain-specific language tailored for software repository mining and allow them to submit queries via our web-based interface. These queries are then automatically parallelized and executed on a cluster, analyzing a dataset containing almost 700,000 projects, history information from millions of revisions, millions of Java source files, and billions of AST nodes. The language also provides an easy to comprehend visitor syntax to ease writing source code mining queries. The underlying infrastructure contains several optimizations, including query optimizations to make single queries faster as well as a fusion optimization to group queries from multiple users into a single query. The latter optimization is important as *Boa* is intended to be a shared, community resource. Finally, we show the potential benefit of *Boa* to the community by reproducing a previously published case study and performing a new case study on the adoption of Java language features.

CHAPTER 1. INTRODUCTION

Ultra-large-scale software repositories, e.g. SourceForge (350,000+ projects), GitHub (250,000+ projects), and Google Code (250,000+ projects) contain an enormous collection of software and information about software. Assuming only a meagre 1K lines of code (LOC) per project, these big-3 repositories amount to at least 8.61 billion LOC alone. Scientists and engineers alike are interested in analyzing this wealth of information both for curiosity as well as for testing such important hypotheses as: "how people perceive and consider the potential impacts of their own and others' edits as they write together? [29]"; "what is the most widely used open source license? [56]"; "how many projects continue to use DES (considered insecure) encryption standards? [54]"; "how many open source projects have a restricted export control policy? [38]"; "how many projects on average start with an existing code base from another project instead of from scratch? [77]"; "how often do practitioners use dynamic features of Javascript, e.g. `eval`? [78]; or "what is the average time to resolve a bug reported as critical? [92]".

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

These four requirements substantially increase the cost of scientific research. There are four additional problems. First, experiments are often unreproducible because replicating an experimental setup requires a mammoth effort. Second, reusability of experimental infrastructure is typically low because analysis infrastructure is not designed in a reusable manner. After all, the focus of the original researcher is on the result of the analysis and not on reusability of the analysis infrastructure. Thus,

researchers commonly have to replicate each other's efforts. Third, data associated and produced by such experiments is often lost and becomes inaccessible and obsolete, because there is no systematic curation. Last but not least, building analysis infrastructure to process ultra-large-scale data efficiently can be very hard [25, 28, 75].

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/VHDL is for hardware description. We implemented *Boa* and provide a web-based interface to *Boa*'s infrastructure [5].

To evaluate *Boa*'s design and effectiveness of its infrastructure we wrote programs to answer 21 different research questions in four different categories: questions related to the use of programming languages, project management, legal, and those that relate to platform/environment. We also implemented several case studies on the Java language, including a reproduction of a prior study [44], which in total contain over 40 additional programs.

Our results show that *Boa* substantially decreases the efforts of scientists and engineers 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 Table 5.2). Last but not least, reproducing an experiment conducted using *Boa* is just a matter of re-running, often small, *Boa* programs provided by previous researchers.

1.1 *Boa*: Enabling Data Intensive Open-source Research

Creating experimental infrastructure to analyze the wealth of information available in open source repositories is difficult [1, 19, 35, 37, 81]. Creating an infrastructure that scales well is even harder [35, 81]. 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 Subversion (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 at the top of Figure 1.1.

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(url + "," + getChurnRateForProject(url));
    }
  }
  private double getChurnRateForProject(String url) {
    double rate = 0;
    SVNURL svnUrl;
    ... // connect to SVN and compute churn rate

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

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

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

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

Boa

```
rates: output mean[string] of int;
p: Project = input;

exists (i: int; lowercase(p.programming_languages[i]) == "java")
  foreach (j: int; p.code_repositories[j].kind == RepositoryKind.SVN
                   && len(p.code_repositories[j].revisions) > 10)
    foreach (k: int; len(p.code_repositories[j].revisions[k].files) < 100)
      rates[p.id] << len(p.code_repositories[j].revisions[k].files);
```

Figure 1.1: Programs for answering “What are the churn rates for all Java projects that use SVN?” in Java and in *Boa*.

This program assumes that the user has manually downloaded all project metadata, available as JSON files, and SVN repositories from SourceForge. It then processes the data using a JSON library and collects a list of Subversion URLs. A SVN library is then used to connect to each cached repository in that list and calculate the churn rate for the project. Notice that this code required use of 2 complex, external libraries in addition to standard Java classes and resulted in almost 70 lines of code. It is also

sequential, so it will not scale as the data size grows. One could write a concurrent version, but this would add additional complexity.

We designed and implemented a domain-specific programming language that we call *Boa* [5, 31] 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 [28, 49, 68, 75]. To these we add built-in types and functions 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 at the bottom of Figure 1.1. 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 4–6 runs. If a repository contains 700k projects, the code on lines 4–6 runs 700k times.

On line 3, this program says to run the code on lines 4–6, if and only if for the input project at least one of the programming languages used is Java. On line 4, this program says to run the code on lines 5–6 for each of the input project's code repositories that are Subversion and contain at least 10 revisions (to filter out new or abandoned projects). Line 5 selects only revisions from such repositories that have less than 100 files changed (to filter out extremely large commits, such as the first commit of a project). Finally, on line 6, this program says to send the length of the array that contains the changed files in the revision to the aggregator `rates`, indexed by the project's unique identifier string. This aggregator produces the final answer to our question.

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

name, programming languages used, repository locations, etc. These abstractions substantially ease common analysis tasks.

Since this program runs on a cluster, it also scales extremely well compared to the (sequential) version written in Java. The time taken to run this program on varying input sizes is shown in Figure 1.2. 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.

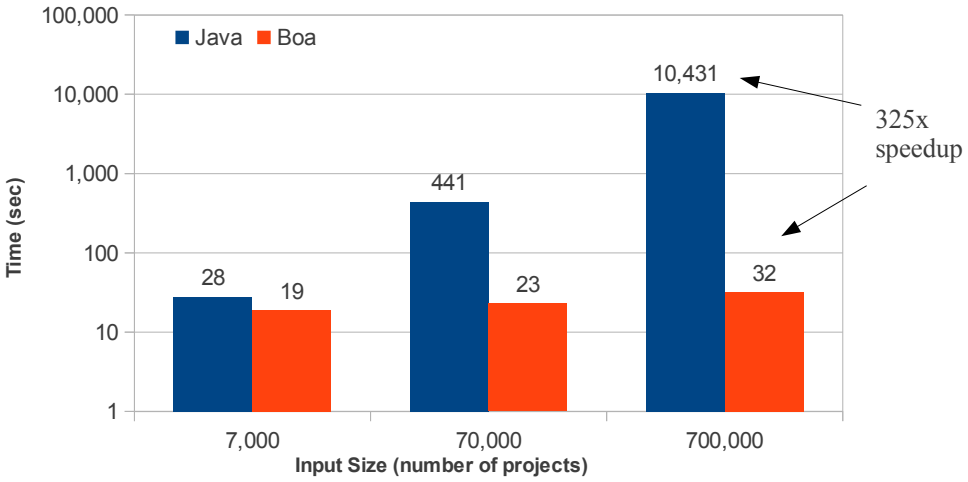


Figure 1.2: Performance results for Java and *Boa* programs

We also built an infrastructure for the *Boa* programming language. An overview of this infrastructure is presented in Figure 1.3. Components are shown inside dotted boxes on the left, the flow of a *Boa* program is shown in the middle, and the input data sources are shown on the right.

The three main components are: the *Boa* language, compiler and runtime, and supporting data infrastructure. First, an analysis task is phrased as a *Boa* program, e.g. that in Figure 1.1 (see Chapter 2). This program is fed to our compiler (see Section 3.1) via our web-based interface (see Section 3.3). The *Boa* compiler produces a query plan. Our infrastructure then deploys this query plan onto a Hadoop [9] cluster, where it executes. The cluster makes use of a locally cached copy of the source code repositories (see Section 3.2) and based on the query plan creates tasks to produce the final query result. This is the answer to the user's analysis task.

In the next chapter we outline the *Boa* language. In Chapter 3 we detail the implementation strategy and framework for the language. In Chapter 4 we describe several optimization strategies to ensure the infrastructure can support many users. We evaluate the language and infrastructure in Chapter 5 and

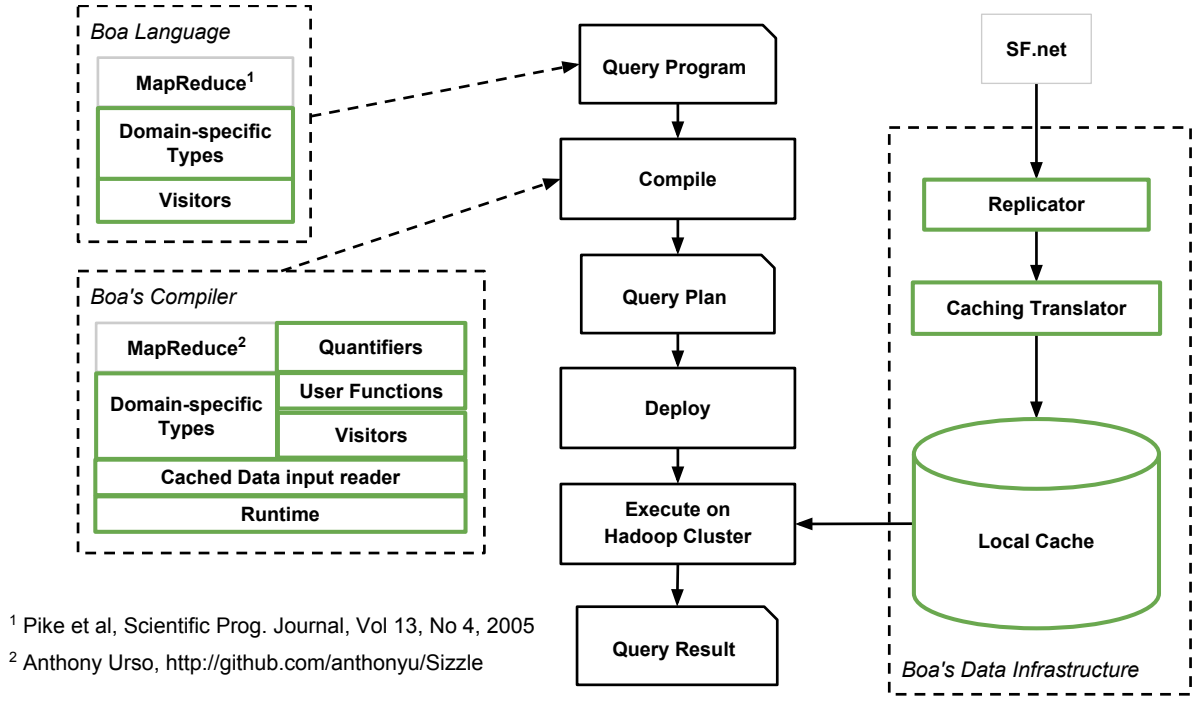


Figure 1.3: An Overview of *Boa*'s Infrastructure. New components are marked with green boxes and bold text.

provide several case studies utilizing *Boa* in Chapter 6. Then in Chapter 7 we discuss works related to *Boa*. In Chapter 8 we outline some ideas on future work. Finally we conclude in Chapter 9.

CHAPTER 2. THE BOA LANGUAGE

One of the first goals of *Boa* is to be easy to use. This means developing a language that is powerful, yet simple to use. The *Boa* language abstracts away many of the details of mining software repositories and also abstracts away details of how to parallelize such analyses.

The left side of Figure 1.3 shows the main kinds of features of the *Boa* language: domain-specific types and functions to ease analysis of open source software repository mining, MapReduce [28] support for scalable analysis of ultra-large-scale repositories, quantifiers for easily expressing loops, the ability to define functions, and an easy to comprehend visitor syntax to ease source code mining tasks. The remainder of this chapter discusses these features in detail. For a full description of the *Boa* language, including the EBNF grammar, please see Appendix A.

2.1 Domain-specific Types in *Boa*

To make mining software repositories as easy as possible, the *Boa* language provides several domain-specific types. Table 2.1 gives an overview of these types (a full list can be found in Appendix B). 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

Table 2.1: Some of the domain-specific types provided in *Boa*.

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

The `CodeRepository` type provides an abstraction of source code versioning systems (such as CVS, SVN, etc). This type contains all of the `Revisions` committed into the 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.

The types provided for representing source code are: `Namespace`, `Declaration`, `Method`, `Variable`, `Type`, `Statement`, `Expression`, and `Modifier`. The declaration, statement, and expression types are discriminated types, meaning they actually represent the union of many different record structures.

For example, consider the type `Statement` that has an attribute `kind`, which is an enumerated value. Based on the kind of statement, different 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 these new languages.

While *Boa* keeps these types as simple as possible, they are still flexible enough to support more complex language features. For example, consider the enhanced-for loop in Java:

```
1 for (String s : iter)
2   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 the `expression` attribute to `iter`. *Boa* also sets the `variable_declaration` attribute for `String s` in the statement. Thus, if a statement of kind `FOR` has its `variable_declaration` attribute set it is an enhanced-for statement. If that attribute is not defined it is a standard for-statement.

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). 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.

2.2 MapReduce Support in *Boa*

In MapReduce [28] 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 [75], a language designed for analyzing log files. In *Boa*, like Sawzall, users write the mapper functions directly and use built-in aggregators as the reduce function. Users declare output tables, process the input, and then send values to the tables. Output declarations specify aggregation functions and the language provides several built in aggregators, such as summing, min/max, mean, etc.

For example, we could write an output declaration for the table `rates` (as shown in Figure 1.1, line 1). For this table 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. Each process then analyzes a single project's revisions, emitting to the project's table the number of changed files for each revision. The aggregator then reduces the values sent to it and computes the means.

2.3 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 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 4 of Figure 1.1, is a syntactic sugar for a loop. The statement says each time, when the boolean condition after the semicolon evaluates to true, execute the code on lines 5–6. The `exists` quantifier on line 3 is similar, however the code on lines 4–6 should execute exactly once if there *exists* some value of `i` 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.

2.4 User-Defined Functions in *Boa*

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

The syntax 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). Several example domain-specific functions are shown in Appendix C.

2.5 Supporting Source Code Analysis with Visitors in *Boa*

Users must also be able to easily express source code mining tasks. For users who are intimately familiar with compilers and interpreters, the visitor pattern [36] 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 [70] and case expressions in Haskell [51].

The new syntax is shown in Figure 2.1. 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 depth-first traversal strategy is used, then the node's children will be visited. After all the children are visited, any matching after clause executes.

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` `;`

Figure 2.1: Proposed syntax for easing source code mining.

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

will execute the clause's body on line 2 when traversing nodes of type `Project`, `CodeRepository`, `Revision`, or `Person`. 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`.

2.5.1 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 zero or more *visit statements*:

```
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. Note this form of the visit statement only has one argument, as the current visitor is assumed.

Figure 2.2 illustrates a custom traversal strategy from one of our case studies in Section 6.1. 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.

```
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);
```

Figure 2.2: Using a custom traversal strategy to find uses of generics in field declarations.

2.5.2 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:

```
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.

2.5.3 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, a task might wish to look for revisions that fixed bugs and then compare 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      if (f.change == ChangeKind.DELETED)
8        remove(files, f.name);
9      else
10       files[f.name] = f;
11   }
12 };
```

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 lines 7–10 updates the map, storing the current version of the file.

2.5.4 Bringing It All Together: Motivating Example

Consider testing a simple hypothesis: a large number of bug fixes add checks for `null`. Null-pointer exceptions are a common source of bugs in object-oriented programs. A possible fix for some of these bugs may be to simply guard access to the variable with a check to ensure it is non-null. In this section, we describe a solution that answers the proposed hypothesis.

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

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

7  nullCheckVisitor := visitor {
8    before e: Expression ->
9      if (e.kind == ExpressionKind.EQ || e.kind == ExpressionKind.NEQ)
10       exists (i: int; isliteral(e.expressions[i], "null"))
11         count++;
12 };

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

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

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

26       # STEP 4 - determine if null checks increased
27       if (count > last)
28         results[p.id] << string(f);
29     }
30     files[f.name] = f;
31     stop;
32   }

33   before s: Statement ->
34     if (s.kind == StatementKind.IF)
35       visit(s.expression, nullCheckVisitor);
36 });
```

Figure 2.3: Finding in Boa fixing revisions that add null checks.

Consider the *Boa* program in Figure 2.3, 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 (step 2) it will get snapshots of the current file and the previous version of the file (step 3) 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 (step 4). 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.

CHAPTER 3. THE BOA INFRASTRUCTURE

The bottom left portion of Figure 1.3 shows the various parts of the *Boa* compiler and runtime. In this chapter we describe each in detail.

3.1 Compiler and Runtime

For our initial implementation, we started with code for the Sizzle [88] 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 [9] open-source MapReduce framework.

```
1  p: Project = input;
2  TopPL: output top(5) of string weight int;

3  foreach (i: int; def(p.programming_languages[i]))
4    TopPL << p.programming_languages[i] weight 1;
```

Figure 3.1: Example program to compute top-5 used programming languages.

Consider the code example in Figure 3.1, which answers the question of what the top-5 programming languages used are. The generated Hadoop program for this 4 line *Boa* program is shown in Figure 3.2. The generated Hadoop program, named `TopFive`, contains code for both the mapper (`TopFiveMapper`) and the reducer (`TopFiveReducer`). It also contains code to start the Hadoop process, by creating a Hadoop job, setting various configuration settings, and then submitting the job.

The majority of the body of the *Boa* program is in the mapper's `map()` method. The only code placed in the reducer are the output variables used, which are stored as a table mapping the output variable's name to a specific aggregator provided by *Boa*'s runtime.

Our main implementation efforts were in supporting the protocol buffer format as input, adding support for quantifiers adding user-defined functions, and supporting the visitor syntax. These efforts

```
public class TopFive extends BoaRunner {
  public static void main(String[] args) throws Exception {
    System.exit(ToolRunner.run(new TopFive(), args));
  }

  public int run(String[] args) throws Exception {
    .. // parse command line

    org.apache.hadoop.mapreduce.Job jb = job(args);
    .. // setup job options
    jb.submit();

    return 0;
  }

  static class TopFiveMapper extends BoaMapper {
    // generate fields for each Boa variable
    private Project $_p;

    protected void map(Text key, BytesWritable value, Mapper<Text, BytesWritable, EmitKey,
  EmitValue>.Context context) throws IOException {
      // read and de-serialize input project
      Project input = Project.parseFrom(com.google.protobuf.CodedInputStream.newInstance(
  value.getBytes(), 0, value.getLength()));

      // body of Boa program (without output variable declarations)
      $_p = input;

      for (int $_i = 0; $_i < $_p.getProgrammingLanguagesList().size(); $_i++)
        if ($_p.getProgrammingLanguagesList().get($_i) != null)
          context.write("TopPL", new boa.io.EmitValue($_p.getProgrammingLanguagesList().get
  ($_i), 1));
    }
  }

  static class TopFiveReducer extends BoaReducer {
    public TopFiveReducer() {
      // output variable declarations
      this.tables.put("TopPL", new TopAggregator(5l));
    }
  }
}
```

Figure 3.2: Generated Hadoop program for example program in Figure 3.1.

were in addition to adding support for our domain-specific types and custom runtime model.

3.1.1 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 generates Hadoop programs that use `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. More details on the storage strategy are given in Section 3.2.1.

The generated Hadoop program gives a single project as input, which is represented as the serialized form of the protocol buffer. The code then deserializes the raw bytes, giving an object-oriented view of the data. Attributes are read from these objects via method calls, e.g. "`_p.getProgrammingLanguagesList()`" in Figure 3.2.

3.1.2 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. For example, the quantifier statement:

```
ifall (i: int; hasfiletype(rev.files[i], "java"))
  counts << 1;
```

generates a for-loop such as:

```
{
  length := len(rev.files);
  stop := false;
  for (i := 0; i < length; i++)
    if (!hasfiletype(rev.files[i], "java")) {
      stop = true;
      break;
    }
  if (!stop) {
    counts << 1;
  }
}
```

which iterates over all the files in the revision. If it finds any file that does not hold the condition, it marks it and breaks out of the loop. If the marker was not set, then the condition held for all values and the original body can execute.

3.1.3 User-Defined Functions

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

3.1.4 Visitors

In this section we outline the code generation strategy for supporting visitor types. For ease of illustration, we omit all code related to MapReduce to allow readers to focus on visitor types. The key to our strategy involves a default visitor (Figure 3.3) that we added to the *Boa* runtime.

```
1public abstract class DeafultVisitor {
2  public final void visit(Project node) {
3    if (preVisit(node)) {
4      ... // call visit() on each of node's children

8      postVisit(node);
9    }
10 }
11 ... // similar visit() for each node type

205 ////////////////////////////////////////
206 // methods for before clauses

207 protected boolean defaultPreVisit() {
208   return true;
209 }

210 protected boolean preVisit(Project node) {
211   return defaultPreVisit();
212 }
213 ... // similar preVisit() for each node type

250 ////////////////////////////////////////
251 // methods for after clauses

252 protected void defaultPostVisit() { }

253 protected void postVisit(Project node) {
254   defaultPostVisit();
255 }
256 ... // similar postVisit() for each node type

295}
```

Figure 3.3: Outline of the abstract default visitor.

The `DefaultVisitor` class contains a public `visit` method for each node type in the language. These methods contain a single if-statement which calls a `preVisit` method in the condition. If that method returns `true`, then the pre-visit method did not execute a stop statement and the children of the current node are each visited followed by a `postVisit` method call.

A `preVisit` and `postVisit` method is also generated for each node type in the language. The bodies of these methods simply call the `defaultPreVisit`/`defaultPostVisit` methods which implements the functionality of the wildcard, which overrides those default methods. These methods are virtual methods and are (possibly) overridden by the concrete visitor sub-classes.

3.1.4.1 Generating Visitors

All visitors in the language:

```
var := visit { .. };
```

inherit from the `DefaultVisitor` (Figure 3.3):

```
var = new DefaultVisitor() { .. };
```

This inheritance provides the visitor with a default depth-first traversal strategy that will visit all nodes in the tree. The actions taken when visiting specific nodes are specified via the before and after visit clauses.

3.1.4.2 Generating Visit Clauses

A before visit clause generates one or more method overrides for the `preVisit` methods. There are three possibilities for a before visit clause's type list. First, it may specify a specific type and an identifier:

```
before id: T -> body;
```

which is translated into:

```
1 protected boolean preVisit(T id) {
2   body;
3   [return true;] // if necessary
4 }
```

Since the method must return a value, the `body` is analyzed to determine if a `stop` statement occurs on all exit paths. If it does not, then a return statement is generated with a value of `true`.

The second form for a visit clause's type list is a list of types:

```
before T1, T2, .. -> body;
```

which is translated similar to before, where each type has its own `preVisit` method generated and the id is a fresh name.

The third form is an underscore wildcard:

```
before _ -> body;
```

which is translated into:

```
1 protected boolean defaultPreVisit() {
2   body;
3   [return true;] // if necessary
4 }
```

similar to the previous translation strategy.

Generation of after visit clauses is almost identical to before clauses, with two slight differences. First, the name of the generated method is changed to `postVisit`/`defaultPostVisit`. Second, since the method has a void return type no return statements are generated.

3.1.4.3 Generating Stop Statements

Before visit clauses return a boolean value to indicate if the `DefaultVisitor` should visit the children of the node. Since `stop` statements can only appear in before visit clauses, they are transformed into:

```
return false;
```

which makes the if condition (Figure 3.3, line 3) `false` and stops the default traversal of the node's children. It also stops the execution of the before visitor.

3.1.4.4 Generating Nested Visit Calls

There is no need to transform a nested `visit` call, as both the method name and arguments are identical in the generated code.

3.2 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 snapshots of the data. The right portion of Figure 1.3 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.

3.2.1 Storage Strategy

All of the data for a single project is processed inside of one 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 (over 6.5GB!) and can not fit their entire object tree in memory at one time. To solve this problem, we split the object tree into a forest of disconnected trees.

Figure 3.4 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.

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

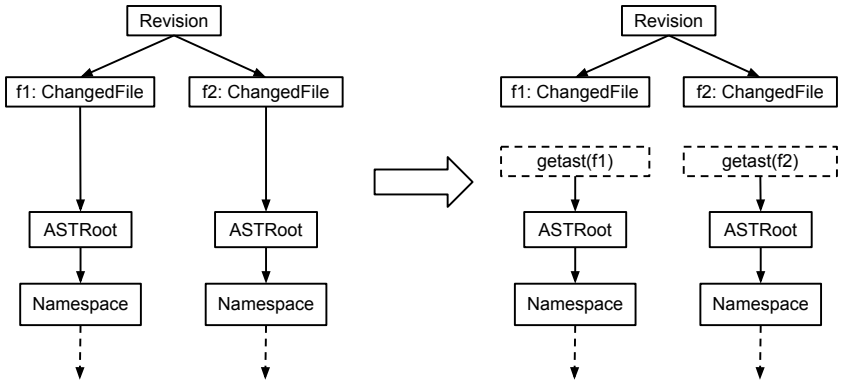


Figure 3.4: Splitting an object tree into a forest.

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 [12], which is an open-source implementation of Google's Bigtable [26]. 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 [20] for more efficient random lookups, which we enabled. Despite this optimization, our initial performance tests indicated that reads were much slower than we expected. 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 our 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 file on the filesystem, it will store the offset of the blocks 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.

3.3 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. The interface utilizes the Drupal open source content managemen system (CMS) [6], which provides user registration and management, theming, easy form generation/validation, etc.

Users submit programs to the interface using our syntax-highlighting text editor (see Figure 3.5). Each submission creates a job in the system (see Figure 3.6), 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 webserver identifies jobs needing compiled and submits the code to the

Figure 3.5: Submitting a query via the web interface.

Figure 3.6: Job created after submitting a query.

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 (see Figure 3.7) to view online or download (as a text file).

3.4 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 1.1:

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

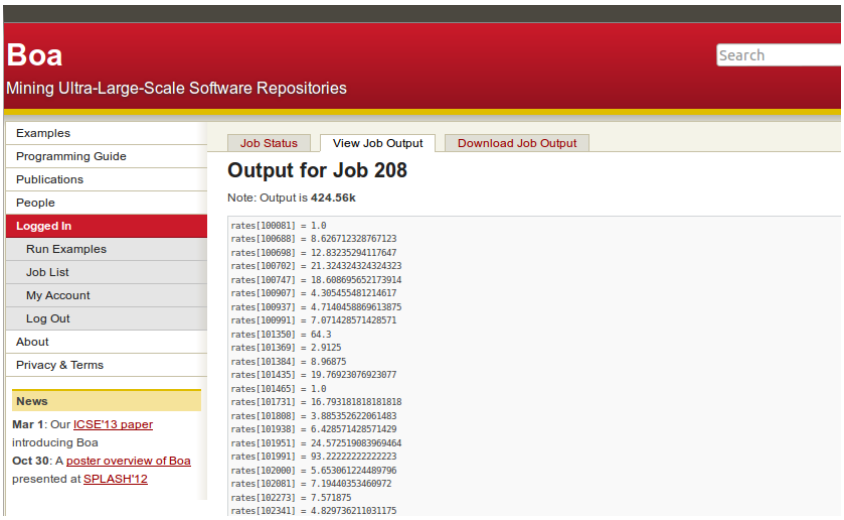


Figure 3.7: Viewing output online (first 64k only). Users can also download the output as a text file.

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 1.1 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 5.2) 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.

CHAPTER 4. OPTIMIZATIONS

In this chapter we investigate how to optimize queries written in *Boa*. First we look at optimizing visitors by ensuring they don't visit unnecessary subtrees. Next we investigate a method for optimizing single programs by rewriting the program to locally combine data prior to emitting it. Then we investigate how to optimize more than one program, possibly from more than one user, by automatically fusing them together into single programs.

4.1 Optimizing Visitor Traversals

By default, the generated code for visitors traverses every node in the tree. For some visitors, this may not be optimal. By analyzing the types in the visit clauses we can determine which subtrees will never be visited and rewrite the visitor to explicitly stop the standard traversal at those points.

For every type in the language, we produce the set of types reachable via traversal starting at a node of that type. For example, every `Revision` is contained inside a `CodeRepository` and thus the first is reachable via the latter. Note that some types, such as `Declaration`, are reachable from themselves (in Java for example, declarations can be nested).

Once we have the sets of reachable types, we then analyze each type in the given visitor's clauses. For each type t, if there is no visit clause for any of its reachable types, then we consider this to be a *lowest type*. For each lowest type, we ensure the traversal stops at that point.

Stopping is accomplished by adding a `stop` statement to the end of the before clause for the type t. There are three cases:

1. The type t has only a before clause in the visitor. A `stop` statement is added to the end of the body of this clause.

2. The type t has only an after clause in the visitor. A before clause is generated with a body containing only a `stop` statement.

3. The type t has both a before and an after clause in the visitor. For example, consider the code:

```
before id1: T -> {
  statement1;
  statement2;
  ..
}
after id2: T -> {
  statement3;
  statement4;
  ..
}
```

In this case, the body of the after clause is merged into the end of the before clause's body:

```
before idFresh: T -> {
  statement1;
  statement2;
  ..
  statement3;
  statement4;
  ..
  stop;
}
```

While doing this, the variables in the after clause are alpha-renamed to avoid name collisions. The identifiers (`id1` and `id2`) are also renamed to a fresh name, ensure that all statements have access to the node. Then a stop statement is added to the end.

The result of this optimization ensures that the visitor will visit all required node types, but stop at the lowest node type and thus avoid traversing the sub-trees below that type (which the visitor is not interested in).

4.2 Task-level Combiners

The form of a query in *Boa* can significantly affect the runtime performance. The underlying MapReduce architecture, and specifically the Hadoop implementation used by *Boa*, transfers data out-

put from maps to the reducers for aggregating. Depending on the reduce function, this data can be optimized prior to transfer. For example, in Hadoop there is an optional class called `Combiner` which, if present, will take spilled records from a map task and locally combine them together before sending the records to the reducer. This minimizes the amount of data sent to the reducer, lowers network congestion, and eases the task of the reducer itself. If the reduce function is both commutative and associative it can be directly re-used as the combiner. Otherwise a custom combiner function is required.

In *Boa*, the runtime code for output aggregators indicate if they can be used as combiners. Each aggregator then defines a specific combine function or re-uses the reduce function (if it is associative and commutative). For example, the sum aggregator simply sums the values locally. The mean aggregator however, sums all the values and emits both the local sum and the local count of the number of values seen.

To demonstrate how the combiner works, consider the following *Boa* program:

```
1 p: Project = input;
2 count: output sum of int;

3 visit(p, visitor {
4     before _ -> count << 1;
5     before Project, CodeRepository, Revision, ASTRoot, Person -> ;
6 });
```

that computes the total number of AST nodes in the system. Since the visitor starts at the `Project` level, any non-AST tree node visited (such as `Project`, `CodeRepository`, `Revision`, `ASTRoot`, and `Person`) do not contribute to the count. All other nodes contribute the value 1 to the total. If each project in the input data has 1000 AST nodes, then each map call will output the value 1 a total of 1000 times. If there were 1000 projects in the input data, then the reducer would see a total of 1 million values of 1, which it would then sum to produce the final result.

Boa however will use Hadoop's combiner functionality. When Hadoop sees a sufficient number of values output on a node, it will invoke the aggregator's combine function. In this case, it would sum the values locally and send the sum to the reducer. The result is substantially less data sent to the reducer over the cluster's network.

Although the combiner operates as expected and indeed improves the efficiency of the system, it highlights one place for improving the performance of Hadoop: a large amount of output data slows down the map tasks. This is because a lot of output data will cause more spilled records, thus invoking

the combiner code. While the combiner is running, the map task is paused and thus useful computation slows down for that compute slot. Additionally, when using a combiner the output from maps is held in memory until either the combiner can be called or the buffer fills (and data is spilled to disk). This adds additional strain on the already congested heap.

While the program above is relatively simple, a minor change:

```
1 p: Project = input;
2 count: output sum of int;

3 count_local := 0;

4 visit(p, visitor {
5     before _ -> count_local++;
6     before Project, CodeRepository, Revision, ASTRoot, Person -> ;
7 });

8 if (count_local > 0) count << count_local;
```

to compute the total AST nodes for each project locally and then output the project's total count to the output variable `len` actually improves performance.

This simple change is mimicking the functionality of the Combiner, however the performance of this version of the program is almost 300% improved (see Section 5.5). One of the reasons for this better performance is that the combiner would need to first sort the data by key (even if there are no indexes on the output variable, there is an implicit key in the MapReduce program).

The other reason for improved performance is the lack of spilled records. In Hadoop, map records are collected in memory and when the buffer fills, it spills the records to disk. When several spill files exist the framework calls the Combiner. By locally aggregating the data in *Boa* first, we avoid generating a large number of output records and thus avoid spilling records and incurring the extra I/O.

Here we outline an algorithm to perform such query optimizations. We call this optimization *task-level combiners*. The algorithm is shown in Algorithm 1. The idea is to take queries of the form shown in the first program and automatically transform them to the second form, thereby performing task-local aggregation on the data and minimizing how often the combiner is triggered.

The algorithm finds output variables using a sum aggregator and inserts a local combiner variable, initialized to 0. It then rewrites all emit statements for the output variable to instead increment the local counter. At the end of the program it then emits that local counter.

The current algorithm is specific to the sum aggregator and only works on output variables without

Algorithm 1 Task-level Combiner Optimization

1. For each output variable v using a sum aggregator
   (a) Insert a local combiner variable (with a fresh name) at the start of the program
   ```
   _local_v := 0;
   ```
   (b) Re-write all emit statements (`v << expr;`) to use the local combiner variable
   ```
   _local_v = _local_v + (expr);
   ```
   (c) Output the local combiner variable at the end of the program
   ```
   if (_local_v > 0) v << _local_v;
   ```

indices. It is relatively straight-forward to extend the algorithm to other aggregators. For example, a top (or bottom) aggregator would require a local list of N top/bottom elements, emit statements would simply update that list, and at the end of the program emit the list's elements. Extending to support indices would require the caching variable be a map.

4.3 Task Fusion

While the previous optimizations focused on single programs, *Boa* is a multi-user environment. We can take the notion of multi-query optimization [13–15,80,83,84] and extend it to the MapReduce world by providing several optimizations in this context. Our first optimization strategy for tasks executing in a multi-user cluster is to perform *task fusion*. Task fusion takes its name from *loop fusion*, which is a compiler optimization for joining the bodies of two or more loops together into a single loop. This is done to increase data locality. We extend this notion to the MapReduce [28] world, specifically for Hadoop [9] implementations.

Task fusion takes two or more Hadoop tasks and fuses them into a single task. This is done to increase data re-use and avoid having to re-read the input data for each individual task. By fusing tasks together, we only have to pay the cost of reading the input data once. For example, in *Boa* this input data is the almost 700k software projects from SourceForge, which on disk (in compressed form) is over 2GB of data. Reading input data represents a large overhead such systems.

At a high level, task fusion works by taking two or more MapReduce tasks and merging them together. Each MapReduce task has one mapper and one reducer (left side of Figure 4.1). What is fused together are the individual mappers, thus giving a program with a single mapper and multiple reducers

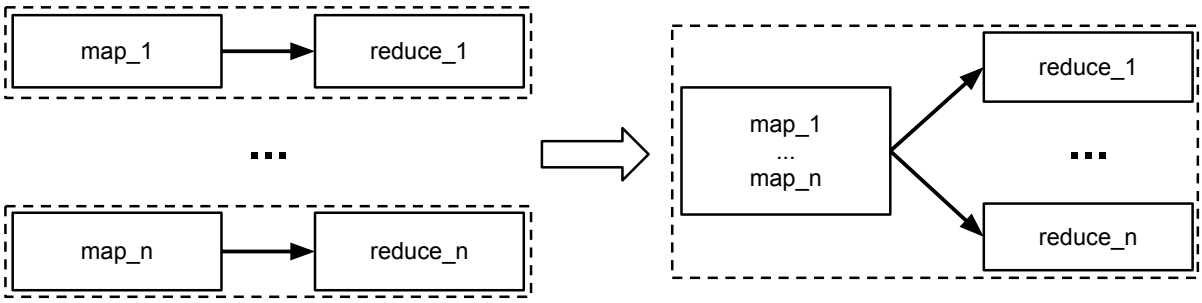


Figure 4.1: Overview of *task fusion*. Dashed boxes represent a single MapReduce job. Solid boxes are individual mapper/reducer tasks.

(right side of Figure 4.1).

```
class FusedTasks extends Mapper {
  class Task1 extends Mapper {
    void map(Text key, BytesWritable val, Context ctx) {
      .. // original code, with output vars renamed
    }
  }
  ...
  class TaskN extends Mapper {
    void map(Text key, BytesWritable val, Context ctx) {
      .. // original code, with output vars renamed
    }
  }
  Task1 t1 = new Task1();
  ...
  TaskN tN = new TaskN();
  void map(Text key, BytesWritable val, Context ctx) {
    try { t1.map(key, val, ctx); } catch ...
    ...
    try { tN.map(key, val, ctx); } catch ...
  }
}
```

Figure 4.2: Code generated when fusing tasks

To fuse the maps together, code similar to Figure 4.2 can be generated. In this figure, we generate the class `FusedMaps` which is declared as a Hadoop `Mapper`. This class contains nested classes, one for each of the tasks being fused. Instances of each class are created, and then the map method contains calls to the map methods of each nested class. These calls are wrapped in try/catch blocks so runtime errors from one task can not impede the execution of another task.

In order to generate code to perform task fusion, several assumptions must hold:

1. The same input data for all tasks

2. No shared state (shared classes, static fields, etc)

3. No side-effects (writing to files, output from mappers, etc)

4. No dependency conflicts (different versions of libraries)

Boa satisfies all of these assumptions by design. All *Boa* programs have the same input (of type `Project`) so assumption 1 holds. *Boa*'s source language has no notion of class, so assumption 2 holds as long as the code generation is careful to avoid using shared classes and static fields (which it does). The language has no capability of writing to files and the output to reducers can be controlled to avoid conflict when fusing, thus assumption 3 holds. There is also no notion of library in *Boa* and all generated programs have the exact same set of dependencies, thus assumption 4 holds.

The assumption on having mappers generate no output can be relaxed, but requires special handling when automatically generating code for task fusion. Consider *Boa*, where the only output allowed is to *output variables*. An output variable is a special variable designed to have values emitted to it. It then aggregates those values together using the function specified in its variable declaration. Example functions include *sum, top, set,* and *collection*. For example, consider the output variable declaration

```
counts: output sum of int;
```

which declares the output variable `counts`. This variable aggregates using the `sum` function and accepts values of type `int`. Values can be emitted to the variable

```
counts << 1;
```

and the variable will sum all emitted values and generate their sum as output.

Boa's compiler transforms these variables into Hadoop reducers. The emit statement shown before generates Hadoop code that writes to the map's context

```
context.write("counts", 1);
```

giving both the output key (`"counts"`) and the value (`1`).

With task fusion however, a rewrite is needed to avoid potential conflicts of two tasks using the same output variable name. Thus the output variables are renamed with a prefix of the task number

```
context.write("1:counts", 1);
```

which ensures all values are grouped by the proper key. A custom partitioner class (see Figure 4.3) uses the task prefix in the key to then send all output for a task to the same reducer. Each reducer in Hadoop

writes output to its own file, and thus this strategy ensures that each original task also has its output in its own file which ensures the process is transparent to the users.

```
public class TaskFusionPartitioner extends Partitioner {
  public int getPartition(Text key, BytesWritable value, int n) {
    String str = key.toString();
    int splitPos = str.indexOf(":");
    return Integer.parseInt(str.substring(0, splitPos));
  }
}
```

Figure 4.3: A custom partitioner for fused tasks

This special handling of output variables allows *Boa* to satisfy all assumptions previously mentioned and *Boa*'s compiler can perform task fusion on multiple source programs from different users.

4.4 Visitor Fusion

In *Boa*, there are actually two forms of input: the Project metadata, which is given as input to the map tasks, and optionally AST nodes from source files in the code repositories. Reading this AST data is an additional opportunity for optimization. In this section, we outline *visitor fusion* which aims to reduce the overhead of reading the AST nodes on disk multiple times, by merging more than one source code analysis together.

Visitor fusion is complicated by the presence of the stop statements. A *stop statement* may appear in before clauses and is similar to a return statement. Unlike return statements, stop statements also tell the visitor to stop the traversal of the tree and to not visit any of the current node's children.

The basic algorithm for visitor fusion is given in Algorithm 2. First, each visitor must be alpha-renamed to ensure there are no duplicate variable names after fusing. Then each visitor that will be fused needs transformed to remove stop statements (see Algorithm 3). Then type lists and wildcards are desugared into single type visits. Finally for each node type in the language, if a visit before or after clause matches that type in more than one visitor, those clauses are merged together.

Transforming the stop statements requires a bit more effort. First, a counter must be added. This counter keeps track of the depth at which a stop statement occurred, so that we can continue traversing down the tree and know when we come back to where the stop statement occurred. This allows fused visitors to visit the children, which may be necessary in one visitor but not the other.

Algorithm 2 Visitor Fusion

1. For each visitor to be fused
   (a) Alpha-rename all variables to fresh names
   (b) Transform all `stop` statements (see Algorithm 3)
   (c) Transform type lists and wildcards to single types
2. For each node type in the language
   (a) fuse all matching `before` clauses in all visitors
   (b) fuse all matching `after` clauses in all visitors

Algorithm 3 Stop Statement Transformer

1. add `_stop: int = 0;` to the start of the program
2. for all `before` clauses containing a `stop` statement
   (a) lift the body into a new function in the containing scope
      i. add `_stop++;` before each `_stop;` statement
      ii. replace `_stop;` statement with `return;`
   (b) replace the original body with
      i. `if (_stop > 0) _stop++;`
      ii. a call to the new function
3. for each node type T matched by a `before` clause in step 2, for each node type T' reachable via type T
   (a) wrap body in a guard: `if (_stop == 0) { body }`
4. for each node type T matched by a `before` clause in step 2
   (a) if there is a matching `after T -> body` replace with:

```
after T -> {
  if (_stop == 0) { body }
  if (_stop > 0) stop--;
}
```

   (b) otherwise, add a new `after` clause

```
after T -> if (_stop > 0) _stop--;
```

The next step is to identify the before clauses that contain stop statements. The bodies of these clauses are lifted into a new function and replaced with a guarded increment of the counter and a call

to that function. The stop statements in the function are then transformed into incrementing the counter and a return statement.

As an example consider the following code:

```
before node: ChangedFile ->
  if (!iskind("SOURCE_JAVA_JLS", node.kind))
    stop;
```

which has a single before clause with a stop statement. This code would be transformed into:

```
stop: int = 0;

stopfunc := function(node: ChangedFile) {
  if (!iskind("SOURCE_JAVA_JLS", node.kind)) {
    stop++;
    return;
  }
};

visit(p, visitor {
  before node: ChangedFile -> {
    if (stop > 0)
      stop++;
    stopFunc(node);
  }
  after node: ChangedFile ->
    if (stop > 0)
      stop--;
});
```

where a new counter has been added, the before clause's body was lifted into a function where the stop statement was transformed, and a new after clause was added to handle decreasing the counter.

If the original code also had another clause which is reachable via a previously transformed clause (it is possibly in the subtree of the previous clause) it must be guarded. For example, a `Method` might possibly be below a `ChangedFile`, so the clause:

```
before node: Method ->
  if (len(node.arguments) > 0)
    MethodsWithArgs << 1;
```

must now be guarded. The guard will check if the counter is 0. If it is not, then we know we are visiting a node below a point where the traversal was stopped and thus the code should not execute.

Finally we have to modify any after clause with a type that matched a previously transformed before clause. The body of the after clause must be guarded, in case we previously stopped traversal. The after clause also updates the counter by decreasing to indicate we are moving up the tree.

4.5 Limitations

Currently, the project combiner optimization only works for sum aggregators. Additionally, the output variable can not contain any indices. In the future we plan to extend the optimization to these

additional use cases.

Task fusion currently requires the following assumptions:

1. No shared state (shared classes, static fields, etc)
2. No side-effects (writing to files, output from mappers, etc)
3. No dependency conflicts (different versions of libraries)

We plan to relax some of these assumptions in the future, by investigating how frameworks like OSGi use separate class spaces to allow modules to load conflicting dependencies and shared state. We also plan to add rewrites for file outputs, similar to how we rewrite map outputs.

There are several limitations to visitor fusion:

Input tasks can not have compile errors This limitation comes from the need to map error messages back to the original task generating that error. If fusion is applied to several tasks and they contain errors, then we need to notify the original authors of each task of just their own errors.

Our implementation in *Boa* avoids this limitation by simply compiling each task individually once, immediately after it is submitted, and only queuing tasks for fusion if they compiled without error. This also gives immediate feedback on compile errors, so users do not need to wait in the queue before seeing them.

Tasks can not cause runtime errors Unlike the previous limitation, this limitation can not be statically checked by the compiler. Since visitor fusion actually merges the bodies of several tasks together, if one of them causes a runtime exception then it will stop the execution of all tasks fused with it. Our current implementation in *Boa* handles this limitation in a simple manner: if a fused execution finishes with any runtime error, we simply re-run each task individually.

In the future, we plan to modify the code generation such that each task gets a boolean variable to indicate if an error has occurred. Then each block from a task will be wrapped in a try/catch, setting the variable to true on exception. These try blocks will be guarded, and only run if there was no previous error. This would allow individual tasks to produce an error while allowing other tasks merged in with them to continue.

All visits must start at the same point We need to ensure that each node is visited exactly the same number of times as in the non-fused tasks. This means that if two visitors started visiting from different points and they were fused, certain nodes would have the wrong number of visits. The simplest solution is to ensure that all visitors being merged start at the same point - the root node `Project`.

In the future, this limitation could be relaxed by augmenting the visited code with a boolean guard. This boolean would only be set to true once the fused visitor reaches the original starting point. Each fused visitor would then have its own boolean and its visit clauses would all be guarded by it.

All fused visitors must be used exactly once This limitation is similar to the previous limitation. To ensure nodes are visited the correct number of times, merged visitors must be used exactly once. For example, if a visitor is inside a loop or a conditional block we don't fuse it. Likewise, if more than one visit is started using a visitor, we don't fuse it.

In the future, more program analysis can be performed to perhaps generate both fused and unfused visitors and choose at runtime which to use. For example, it may make sense to use a fused visitor inside a loop on the first iteration, and then switch to unfused for the remainder.

CHAPTER 5. EVALUATION

This section presents our empirical evaluation of both the *Boa* language and infrastructure. First we look at how applicable the language is for research by asking and answering several mining tasks across multiple categories. We then evaluate if the infrastructure scales as the size of the data and number of nodes increases. We evaluate our chosen storage strategy against alternatives. Next we evaluate the optimizations shown in the previous chapter. Finally we evaluate both the reproducibility of the framework and the readability of the language via small controlled experiments.

5.1 Setup

All programs were executed on a Hadoop [9] 1.2.1 install with 1 name node, 1 job tracker node, and 6 compute nodes. The compute nodes are five Core2 Quad 2.66GHz machines with 8GB ram and a single 24-core 1.9GHz machine with 64GB ram. 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.

Metric	Count
Projects	*699,331*
Repositories	*494,158*
Revisions	**15,063,073**
Files	*69,863,970*
File Snapshots	*147,074,540*
AST Nodes	**18,651,043,238**

Table 5.1: Metrics for the SourceForge-based dataset in *Boa*.

The dataset used for evaluation contains all metadata about all SourceForge projects (700k+[1]) and

[1]This includes "user" projects, which aren't listed on the main website.

repository metadata for only the Subversion and CVS repositories. The dataset also contains information on Java source files (files with the extension ".java") that parse without error. Note this may include semantically-invalid source files (as long as they parse).

Table 5.1 shows the metrics for this dataset. The dataset contains almost 700k projects, almost 500k code repositories with over 15 million revisions, information on almost 70 million unique files with over 140 million snapshots of those files, and over 18 billion AST nodes!

5.2 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 Table 5.2) that cover a range of different categories. For each task, we implemented a *Boa* program to solve the task. We also implemented small Java programs to solve the same tasks. The Java programs were written by an expert in mining software repositories and then reviewed by a second person who is an expert in programming languages. The second person performed a code review and also simplified and condensed the programs to decrease the total lines of code as much as reasonably possible without impacting performance. This process substantially reduced (almost by half) the lines of code for the Java versions.

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

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

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

All results are shown in Table 5.2. The lines of code give an indication of how much effort was required to solve the tasks using each approach. For Java, the tasks required writing 32–107 lines of

[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	4	15.25x	602	21	28.67x
2. How many projects use more than one programming language?	32	4	8.00x	603	22	27.41x
3. In which year was Java added to SVN projects the most?	89	10	8.90x	6,998	21	333.24x
B. Project Management						
1. How many projects are created each year?	43	3	14.33x	651	21	31.00x
2. How many projects self-classify into each topic provided by SourceForge?	45	4	11.25x	556	23	24.17x
3. How many Java projects using SVN were active in 2011?	66	6	11.00x	5,053	24	210.54x
4. In which year was SVN added to Java projects the most?	107	6	17.83x	4,880	22	221.82x
5. How many revisions are there in all Java projects using SVN?	60	5	12.00x	4,636	22	210.73x
6. How many revisions fix bugs in all Java projects using SVN?	76	6	12.67x	10,750	22	488.64x
7. How many committers are there for each Java project using SVN?	69	6	11.50x	10,821	24	450.88x
8. How many Java projects using SVN does each committer work on?	72	9	8.00x	10,435	22	474.32x
9. What are the churn rates for all Java projects that use SVN?	68	5	13.60x	10,431	22	474.14x
10. How did the no. of commits for Java projects using SVN change over years?	79	6	13.17x	10,489	27	388.48x
11. For all Java projects using SVN, what is the distribution of commit log length?	82	6	13.67x	10,518	28	375.64x
C. Legal						
1. What are the five most used licenses?	63	4	15.75x	474	22	21.55x
2. How many projects use more than one license?	32	4	8.00x	522	21	24.86x
D. Platform/Environment						
1. What are the five most supported operating systems?	61	4	15.25x	469	23	20.39x
2. What are the projects that support multiple operating systems?	33	4	8.25x	597	22	27.14x
3. What are the five most popular databases?	61	4	15.25x	498	22	22.64x
4. What are the projects that support multiple databases?	32	4	8.00x	558	22	25.36x
5. How often is each database used in each programming language?	71	5	14.20x	598	21	28.48x

Table 5.2: Several example mining tasks, with lines of code and execution times (in seconds) for both Java and *Boa* programs solving the tasks.

code and on average required 62 lines of code. Performing the same tasks in *Boa* required at most 10 lines of code and on average less than 5 lines of code. Thus there were 8–18 times fewer lines of code when using *Boa*.

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

The table also lists the time required to run each program and collect the desired data for the tasks. Note the Java programs accessed all JSON and SVN data from a local cache and the times do not include any network access. For the Java programs, there are three distinct groups of running times. The smallest times (A.1, A.2, B.1, B.2, and all of C and D) are tasks that only require parsing the project metadata and did not access any SVN data. The medium times (A.3, B.3, B.4, and B.5) accessed the SVN repositories but only required mining one (or very few) revisions. The largest times (B.6–B.11) all accessed the SVN repositories and mined most of the revisions to answer the task and thus required substantially more time. Note that for the *Boa* programs, all tasks finish on average in 24 seconds, regardless of the type of task. We see minimum speedups of 20 times but in the best case the *Boa* program solves the task over 450 times faster!

Task	Java (cached)	Java (remote SVNs)	Boa	Speedup
A.3	6,998	45,793	21	2,180.62x
B.3	5,053	25,690	24	1,070.42x
B.4	4,880	18,700	22	850.00x
B.5	4,636	17,888	22	813.09x
B.6	10,750	95,404	22	4,336.55x
B.7	10,821	85,265	24	3,552.71x
B.8	10,435	95,755	22	4,352.50x
B.9	10,431	88,440	22	4,020.00x
B.10	10,489	100,883	27	3,736.41x
B.11	10,518	88,279	28	3,152.82x

Table 5.3: Time (in seconds) if Java tasks do not cache SVN repositories first.

While the times in Table 5.2 utilize local caches for all data, including SVN repositories, researchers implementing such tasks might not first cache the SVN data. As such, we again present the times for all tasks that access SVN in Table 5.3 with the difference being the Java programs now access the SVN repositories remotely. Compared to this strategy, *Boa* programs run up to 4,000 times faster! The results

clearly show the caching saved significant time, resulting in 10x speedups compared to accessing the remote SVNs.

5.2.1 Detailed Examples

Figures 5.1–5.4 show four interesting *Boa* programs used to solve some of the tasks. These programs highlight several useful features of the language. Code for the remaining programs are shown in Appendix D.

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

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

8  foreach (i: int; def(p.code_repositories[i]))
9    exists (j: int; HasJavaFile(p.code_repositories[i].revisions[j]))
10     counts[yearof(p.code_repositories[i].revisions[j].commit_date)] << 1;
```

Figure 5.1: Task A.3: Querying years when Java files were first added the most.

Figure 5.1 answers task A.3 and demonstrates the use of a user-defined functions. The function `HasJavaFile` (line 3) 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 `exists` statement (line 9) as the boolean condition.

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

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

Figure 5.2: Task B.6: Querying number of bug-fixing revisions in Java projects using SVN.

Figure 5.2 answers task B.6 and makes use of the built-in function `isfixingrevision` (line 5). 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.

Figure 5.3 answers task C.1 and makes use of a *top aggregator* (line 1). The emit statement (line 4) now takes additional arguments giving a weight for the value being emitted. The top aggregator then

```
1 counts: output top(5) of string weight int;
2 p: Project = input;

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

Figure 5.3: Task C.1: Querying the five most used licenses.

selects the top N results that have the highest total weight and gives those as output.

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

3 foreach (i: int; def(p.programming_languages[i]))
4   foreach (j: int; def(p.databases[j]))
5     counts[p.programming_languages[i]][p.databases[j]] << 1;
```

Figure 5.4: Task D.5: Querying pairs of how often each database is used in each programming language.

Figure 5.4 answers task D.5 and makes use of a multi-dimensional aggregator (line 1) to output pairs of results. Again, the `emit` statement (line 5) is modified. This time, the statement requires providing multiple indexes for the table.

5.2.2 Results Analysis

We also show some interesting and potentially useful results from four of the tasks. For example, Figure 5.5 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 [22]. 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 5.6. 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 5.7 shows the number of projects each unique committer works on. Again, the vast majority of open-source developers only work on a single

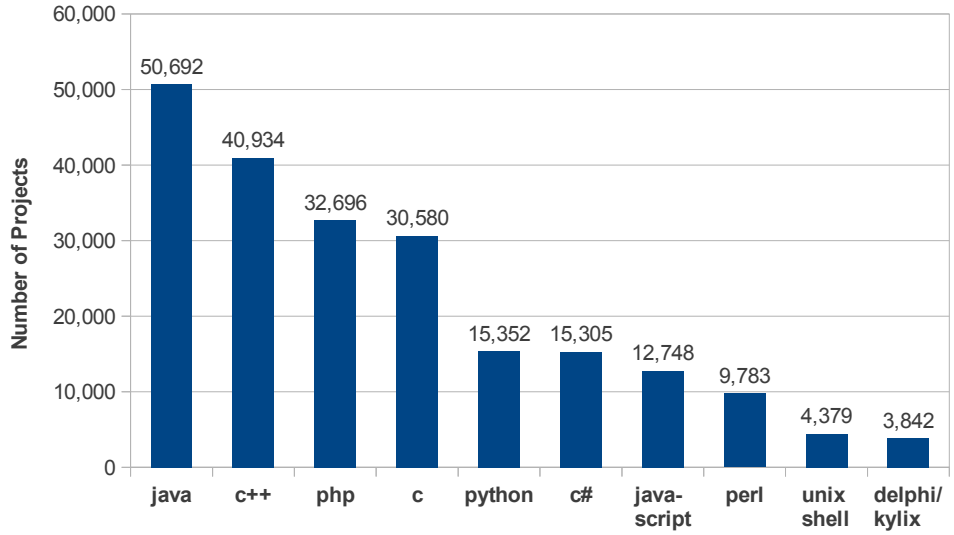


Figure 5.5: Task A.1: Popularity of programming languages on SourceForge.

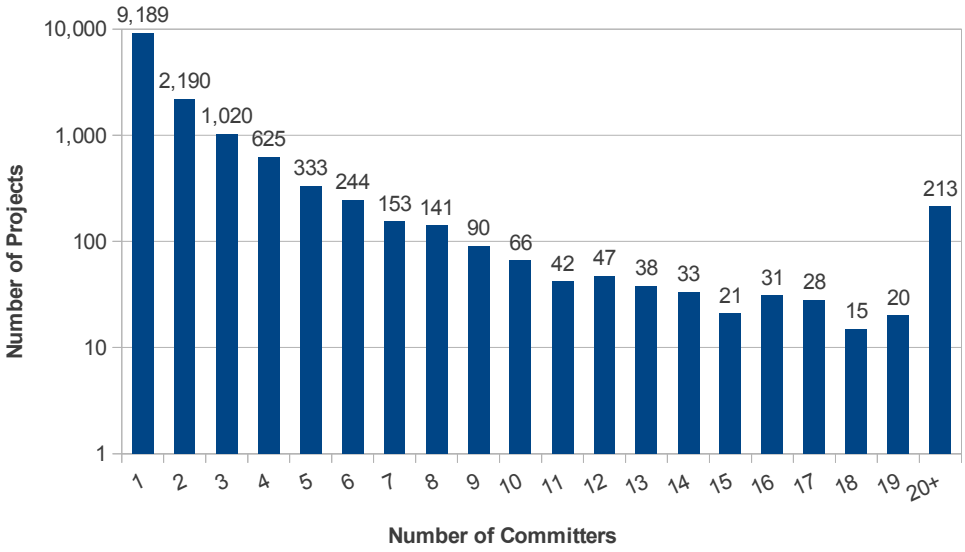


Figure 5.6: Task B.7: number of committers in each Java project using SVN. NOTE: y-axis is in logarithmic scale.

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 5.8. This task examines how many words appear in log messages. First, around 15% 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

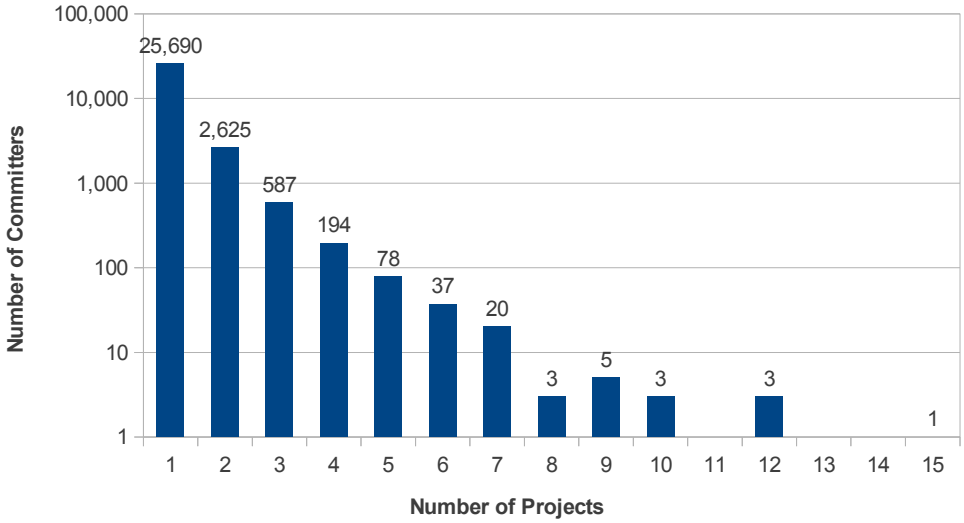


Figure 5.7: Task B.8: number of Java projects each SVN committer works on. NOTE: y-axis is in logarithmic scale.

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 (12%) contained descriptive messages.

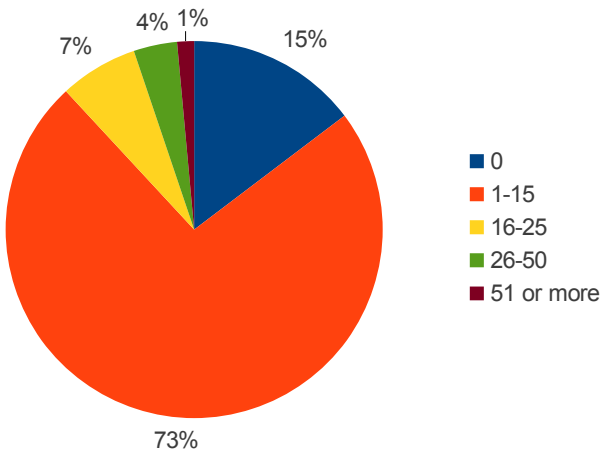


Figure 5.8: Task B.11: number of words in SVN commit logs for Java projects.

5.3 Scalability

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

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

To answer this question, we run each of the sample programs listed in Figures 5.1–5.4 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 1–32. Figure 5.9 shows the results of this analysis where each group represents one of the sample programs, the y-axis is the total time taken in seconds to run the program, and the x-axis is the number of available map slots in the cluster. Each value is the average of 10 executions.

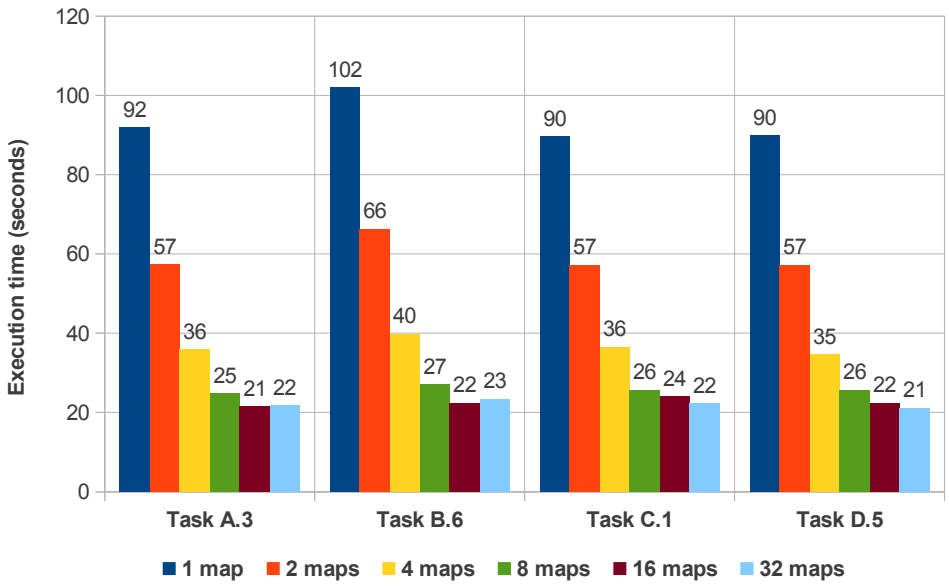


Figure 5.9: Scalability of sample programs. Y-axis is total time taken. X-axis is the number of available map slots in the cluster.

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

Note that with our current input size of 700k projects, the maximum number of map slots needed is 14. Thus we don't generally see any benefit when increasing the maximum map slots past that. As we

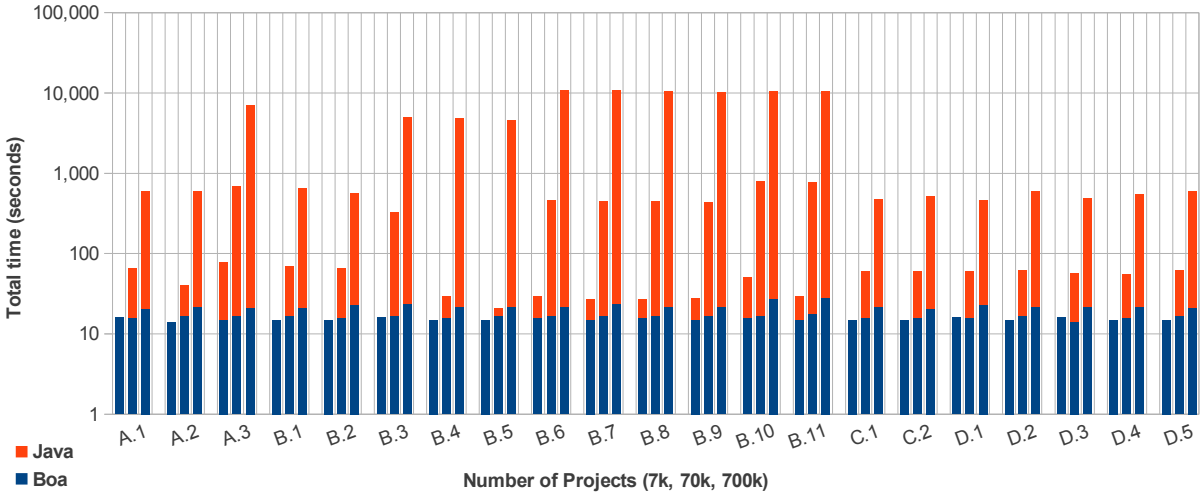


Figure 5.10: Scalability of input. Y-axis is total time taken. X-axis is the size of the input in number of projects. NOTE: y-axis is in logarithmic scale.

increase the size of our input dataset however, we would expect to see differences in these data points indicating scaling past 14 map slots.

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

To answer this question, we fix the number of compute nodes to 6 (with a total of 44 map slots available) and then vary the size of the input (7k, 70k, and 700k projects). The results for all tasks in Figure 5.2 are shown in Figure 5.10. 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 15 seconds. At this size *Boa* only uses one map task and thus the overhead of Hadoop dominates the execution time. For the larger input sizes, *Boa* always runs in (substantially) less time than the Java version.

The results also show that the hand written Java programs do not scale based on input size. As the input size increases, the running time for the Java programs also increases (roughly linearly). The *Boa* programs however demonstrate scalability. For the two smallest input sizes, the *Boa* programs take roughly the same amount of time. For the largest input size the *Boa* programs, despite having to process an input 100 times larger than the smallest input size, only take around twice as long. This shows that

the *Boa* infrastructure scales well as the input size increases.

5.4 Storage Strategy

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.

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

Table 5.4: Size of the dataset used for evaluation.

Table 5.4 shows the size of the dataset used in our evaluation. This dataset contains project metadata and source code repositories cloned from SourceForge [3]. While the dataset itself contains metadata 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.

5.4.1 Evaluation

In this section, 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 NullCheck source code example, two tasks written for our other study [32] (AnnotUse and SafeVarargs), and a task to reproduce another group's study [44] (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 5.11 and are normalized to the first strategy, HBase+MapFile. Each bar represents a 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.

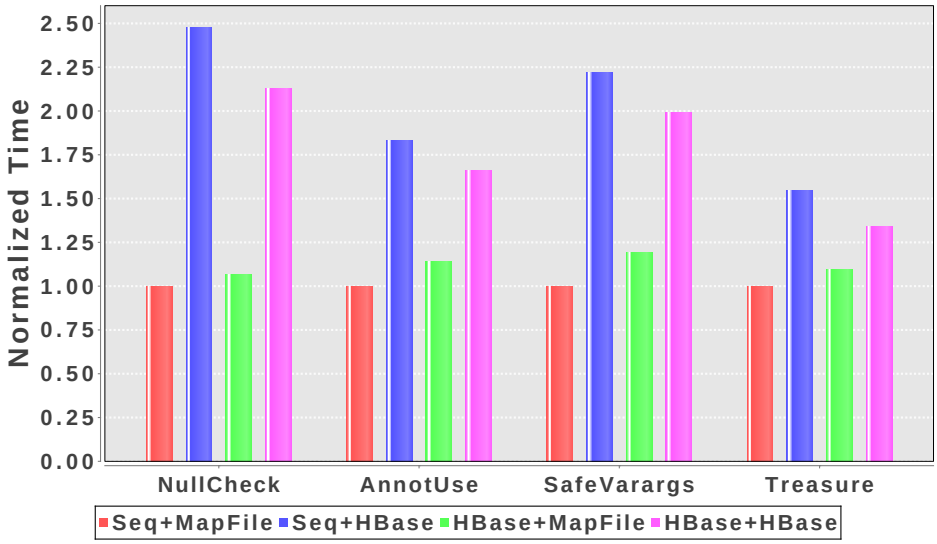


Figure 5.11: Performance comparison of MapFile and HBase stores. Results normalized to Seq+MapFile. Smaller is better.

Figure 5.12 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 a

SequenceFile 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.

Figure 5.12: Network utilization. Note the minimal use by the MapFile store (left) compared to the HBase store (right).

Figure 5.13 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 5.11.

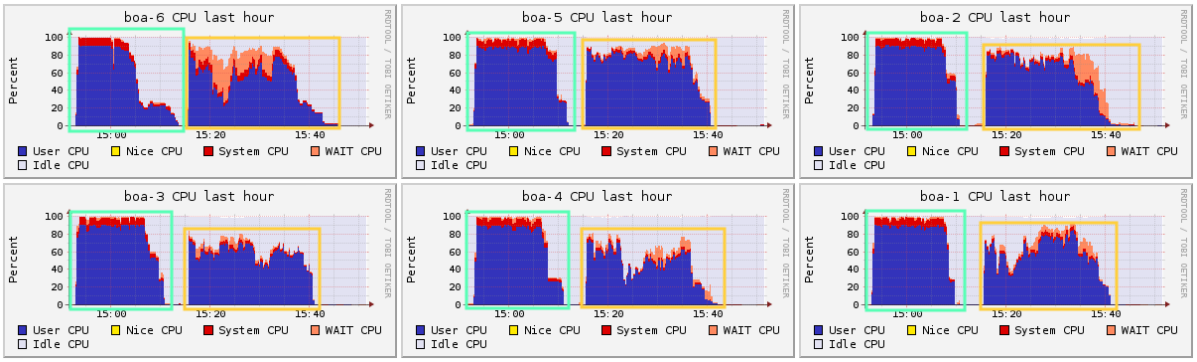


Figure 5.13: CPU usage across cluster. Left-most group used the MapFile store. Right-most group used the HBase store.

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.

5.5 Task-level Combiners Performance

In Section 4.2 we outlined an optimization strategy called *task-level combiners*. In this section we evaluate the performance benefit of this optimization. Recall the example program to count AST nodes:

```
1  p: Project = input;
2  count: output sum of int;

3  visit(p, visitor {
4      before _ -> count << 1;
5      before Project, CodeRepository, Revision, ASTRoot, Person -> ;
6  });
```

which we will use in this section to evaluate the performance. We run this program three times: once with Hadoop's combiners disabled, once with combiners enabled, and once with task-level combiners enabled (which implies Hadoop combiners are enabled). The results are shown in Table 5.5.

Optimization	Time
No Combiner	43h 48m 58s
Combiner	1h 8m 13s
Task-level Combiner	15m 25s

Table 5.5: Time to execute task without combiners, with combiners, and with task-level combiners.

The results clearly show the benefit of Hadoop's combiners. With no combiners enabled, the task takes almost 44 hours! Once Hadoop's combiners are enabled, the task time drops to a little over 1 hour, a 44x improvement. Task-level combiners improved even further, taking only 15 minutes (or 4.5x faster than combiners only)! To investigate further why these approaches are faster, we look at what the I/O in the system is doing.

Table 5.6 shows some of the I/O characteristics on the cluster. The first row shows the number of outputs coming from all map tasks. The second row shows the number of bytes read and written on the local filesystems of each compute node. The third row shows the number of records sent to the combiner and output from the combiner.

	No Combiner	**Combiner**	**Task-level Combiner**
Map Outputs	18.6B	18.6B	65K
Bytes Read / Written	92G / 109G	3.5M / 16.6M	2K / 13M
Combiner Input / Output	- / -	18.6B / 71K	65K / 214

Table 5.6: Number of map outputs, number of local filesystem bytes read/written without combiners, and number of inputs/outputs to the combiner, with combiners, and with task-level combiners.

The table clearly shows why the combiner is 44x faster: the amount of bytes read/written is substantially lower. Without combiners there is almost 100GB read/written, while with combiners that drops to a few MB. Notice however the number of outputs from the maps is the same (18.6 billion) without combiners as with combiners. Task-level combiners however have only 65K map outputs. This is similar to the number of outputs leaving the combiner (71K) in the previous optimization. Notice that with task-level combiners, the total number of outputs from the combiner is 214 meaning the final reducer only has 214 inputs! This means very little data is transmitted and the reducer takes less time.

5.6 Task and Visitor Fusion Performance

In this section we evaluate how our two optimization strategies, task fusion and visitor fusion, perform by comparing three large studies. Each study has around 20 different tasks and we show the execution times for each task in Tables 5.7, 5.9, and 5.11. Each row in the result tables represents 10 runs. We provide basic statistical analysis on the data, including max/min, mean, median, variance, etc.

In total there are over 60 tasks which vary in execution time from around 50 seconds up to almost

30 minutes. For each study we note how long it takes to execute all tasks in that study sequentially. We then manually merge all tasks into a single, monolithic task which we attempt to hand optimize. The performance of this manually merged task is then compared to our two optimization strategies, which are both fully automatic.

5.6.1 Performance Study I: *Boa* Examples

For our first study, we investigate the example *Boa* tasks from the original paper [31]. These tasks only analyze the project and repository metadata and do not analyze any source code from the repositories [3]. The results for this study are shown in Table 5.8. Note that all tasks individually take under one minute to execute (see Table 5.7).

Example Boa tasks [31]						
Task	**Mean**	**Median**	**Max**	**Min**	**StdDev**	**Variance**
A1. Top-10 Programming Languages	52.10	52	56	48	2.92	8.54
A2. Projects Using Multiple Languages	51.10	50	56	47	2.73	7.43
A3. Year Java Added to Projects the Most	50.50	50	55	45	2.72	7.39
B1. Projects Created Each Year	53.60	54	57	49	2.80	7.82
B2. Topics of Each Project	52.30	53	54	50	1.64	2.68
B3. Active Java Projects in 2011	52.40	52	56	50	2.12	4.49
B4. Most Popular Year Adding SVN	50.80	51	53	49	1.40	1.96
B5. Revisions in SVN Java Projects	50.00	50	59	44	4.62	21.33
B6. Number of Fixing Revisions	54.40	55	57	51	2.17	4.71
B7. Number of Committers	55.20	56	58	49	2.82	7.96
B8. Number of Projects for Each Committer	52.60	53	56	47	2.22	4.93
B9. Average Churn Rate	55.50	55.5	62	50	4.14	17.17
B10. Change in Number of Committers	56.40	57.5	62	50	3.95	15.60
B11. Commit Log Lengths	54.40	54.5	58	50	2.72	7.38
C1. Top-5 Licenses	50.90	50.5	53	50	1.20	1.43
C2. Projects Using Mulitple Licenses	50.10	50	53	47	1.79	3.21
D1. Top-5 OSes	52.20	51	56	50	2.25	5.07
D2. Projects Supporting Multiple OSes	52.60	52	57	48	2.88	8.27
D3. Top-5 Databases	49.90	50	53	47	2.23	4.99
D4. Projects Supporting Multiple DBs	50.00	49	54	47	2.75	7.56
D5. DB use in each PL	52.00	53	55	47	2.21	4.89

Table 5.7: Execution times (in seconds) for example Boa tasks

For this study, the time to run all tasks sequentially (shown in the first row) is over 18 minutes. This

[3] At that time, *Boa* did not yet have source code analysis capabilities.

Task	Mean	Median	Max	Min	StdDev	Variance
Sequential	1108.33	1109	1179	1042	44.52	1982.00
Manually Merged	50.80	52	53	47	2.39	5.73
Task Fusion	69.20	69	75	64	3.58	12.84
Task Fusion + Visitor Fusion	71.80	72.5	77	66	3.46	11.96

Table 5.8: Execution times (in seconds) for example Boa tasks [31]

time represents how long the system would take to execute all of the tasks without any of the optimizations described in this paper. As long as our approaches are less than this time we have improved the overall system performance.

Next we list the manually merged time. This is a special program, hand-written to merge all of the tasks together. While writing this task, we made sure to optimize as much as possible. For example, several tasks iterate over the same data so we manually joined those loops. This task thus represents a best-case scenario, and provides a lower goal for our optimizations. For this study, the manually merged task took 50 seconds.

Finally, we show the execution time of the tasks when compiled using automated task fusion. Note that this required no manual merging, but rather simply passing each individual source file to the compiler. This version executes in 69 seconds, which is 16 times faster than the sequential version!

Adding visitor fusion to this work load does not help as these particular tasks contain no visitors. The visitor fusion algorithm thus simply merges each tasks individual map functions together into one monolithic function and the running time is comparable to only using task fusion.

5.6.2 Performance Study II: Java Feature Use

For our second study, we investigate tasks to measure Java language feature use [32]. These tasks analyze the Java source code to identify how various language features are used over time. The results for this study are shown in Table 5.10. Note that the individual tasks for this study take 15–30 minutes each (see Table 5.9), which is substantially longer than the previous study's tasks.

Running this study's 18 tasks sequentially requires over 4.5 hours to complete. The manually merged version is 7 times faster than the sequential, taking just under 40 minutes. Note that when making the manual version for this study, we were not able to fully optimize the visitors by hand. One of the visitors (from J6) was extremely complex and contained multiple stop statements and custom

Java feature use study [32]						
Task	**Mean**	**Median**	**Max**	**Min**	**StdDev**	**Variance**
J1. Assert	857.00	865	896	806	32.59	1062.00
J2. Annotation Declaration	827.10	828	863	742	36.20	1310.32
J3. Annotation Use	1474.80	1491.5	1513	1408	38.23	1461.29
J4. Enhanced For Loop	1770.20	1693.5	2153	1624	189.89	36056.40
J5. Enums	835.40	844.5	867	735	38.27	1464.93
J6. Generic Variable	954.10	962	1004	857	46.15	2129.88
J7. Generic Method	817.30	829.5	865	740	40.12	1610.01
J8. Generic Type	837.50	840	873	783	30.16	909.83
J9. Extends Wildcard	876.60	884	917	818	33.22	1103.38
J10. Super Wildcard	863.40	866.5	904	824	25.71	661.16
J11. Other Wildcard	901.50	910	931	812	34.06	1160.28
J12. Varargs	842.10	844	908	757	47.59	2264.77
J13. Binary Literals	857.50	865.5	880	827	20.26	410.50
J14. Diamond	848.60	860	885	801	31.07	965.16
J15. MultiCatch	805.90	814.5	857	722	55.19	3045.43
J16. Safe Varargs	837.30	844	871	788	25.96	673.79
J17. Try with Resources	834.40	833.5	861	807	20.99	440.49
J18. Underscore Literals	1367.10	1340	1454	1301	60.79	3695.43

Table 5.9: Execution times (in seconds) for Java feature study tasks

Task	**Mean**	**Median**	**Max**	**Min**	**StdDev**	**Variance**
Sequential	17547.33	17487	18502	16628	617.22	380959.50
Manually Merged	2389.20	2361.5	2543	2291	84.43	7128.84
Task Fusion	14260.70	14209	14758	13657	354.01	125319.57
Task Fusion + Visitor Fusion	1891.80	1918.5	2100	1618	156.73	24563.51

Table 5.10: Execution times (in seconds) for the Java feature use study [32]

visit calls. For this visitor, we simply left it un-merged and called it from inside the first visitor. Thus the running time is close to that of J6 plus the longest running time, J4.

The automated task fusion version shows improvement over the sequential, with a time that is 18% lower. However when compared to the manual version, there is still room for improvement. Since every task in this study uses visitors, we also run the visitor fusion. The results of using both task and visitor fusion on this study actually beat the manual version! The visitor fusion time is 9 times faster than the sequential and 20% lower than the manual version.

To understand why the automated version is fastest in this case, it is important to realize how much data is generated as output. These 18 tasks generate over 17GB of output data. With the automated

task and visitor fusion algorithms, each individual task has its own reducer generated. Thus for the automated runs, there were 18 reducers available to process and generate this data. For the manual version however, it is considered by the compiler as a single task and only gets a single reducer. This single reducer obviously takes longer than 18 parallel reducers, and thus the manual version is slower.

5.6.3 Performance Study III: Treasure Study

Our next study is on 22 tasks that analyze Java source code [33]. These tasks are a reproduction of a previous study [44] on the Java language. The results for this study are shown in Table 5.12. Individual tasks in this study take 6–7 minutes each (see Table 5.11).

Treasure study [44] reproduction [33]						
Task	**Mean**	**Median**	**Max**	**Min**	**StdDev**	**Variance**
T1. Classes	373.80	377.5	412	337	22.95	526.84
T2. Static Classes	399.90	411.5	438	339	35.77	1279.43
T3. Anonymous Classes	386.60	372	435	344	31.73	1006.71
T4. Nested Classes	375.10	380.5	407	337	26.22	687.43
T5. Assert Statements	438.20	427	494	406	32.18	1035.73
T6. Methods	391.40	388	428	358	26.93	725.38
T7. Static Methods	389.20	384	444	354	30.88	953.51
T8. Methods (interfaces)	393.20	392	429	345	29.12	847.73
T9. Method Arities	404.20	399.5	431	393	12.28	150.84
T10. `void` return Methods	386.70	382	428	361	22.12	489.12
T11. Methods Returning Arrays	386.30	386.5	424	359	23.48	551.34
T12. non-`void` return Methods	386.30	389	428	341	30.46	927.57
T13. Fields	440.90	429	490	416	27.17	738.10
T14. `this` Expressions	403.20	405	431	353	23.98	575.07
T15. Static Fields	390.70	384.5	442	371	21.53	463.57
T16. Volatile Fields	404.10	407	442	357	28.98	839.66
T17. Conditional Statements	384.20	371.5	436	349	28.54	814.40
T18. `String` Fields	430.40	429.5	478	404	20.26	410.27
T19. `try` Statements	432.60	427.5	449	419	11.46	131.38
T20. Exceptions Thrown From `catch`	447.60	440.5	495	414	31.94	1020.27
T21. Exceptions	385.10	379.5	426	347	29.34	860.99
T22. Local Variables	434.60	427.5	491	413	22.62	511.82

Table 5.11: Execution times (in seconds) for Treasure study tasks

The sequential run of these tasks takes almost 2.5 hours to complete. The manually merged version takes around 11 minutes to complete, which is 12 times faster.

Task	Mean	Median	Max	Min	StdDev	Variance
Sequential	8947.33	8860	9778	8257	508.58	258656.50
Manually Merged	719.60	727	780	631	49.21	2421.60
Task Fusion	5916.80	5890.5	6100	5818	98.07	9617.51
Task Fusion + Visitor Fusion	6387.40	6432	6813	5632	382.54	146336.49

Table 5.12: Execution times (in seconds) for the Treasure study [44] reproduction [33]

Running these tasks with automated task fusion takes a bit over 1.5 hours to complete. It is important to note that even though task fusion is slower than the manually merged version, its running time is still about 33% lower than the sequential running time.

Unfortunately, for these tasks adding visitor fusion not only doesn't help, but makes it take longer than using task fusion alone. Note however that it is still faster than the sequential version, and thus it is still an improvement.

The reason visitor fusion fails here is due to the particular tasks. These tasks all get a snapshot (the last version of each file in a repository) and then manually visit the files from that snapshot. When N of these are merged, the merged visitor winds up getting N copies of the same snapshot, and then visiting the snapshot N times. The visitor fusion algorithm is not able to optimize here.

We can however detect when a visitor is manually visiting nodes in such a manner and avoid attempting visitor fusion on them. In this way, we can avoid the slight overhead incurred when merging such visitors.

5.6.4 Performance Study IV: Mixed Workloads

Since each of the previous studies contained relatively homogenous tasks, in this study we investigate a mixed workload. We create three different workloads (M1, M2, M3) with different mixtures of tasks from the previous studies. The first workload is mostly fast tasks from the first study. The second workload is three tasks from each study. The third workload is mostly slow tasks from the second study. The results are shown in Table 5.13.

There are several interesting results here. First, note the running time for M1. Despite having mostly fast tasks, the longer tasks dominate the running time. This implies that if it can be avoided, fast and slow tasks should not be fused together.

Second, notice that in all three workloads our optimizations manage to bring the running time down

M1. Mostly Fast (A3 / B4 / B5 / C1 / C2 / D3 / D4 / T20 / J4)						
Task	**Mean**	**Median**	**Max**	**Min**	**StdDev**	**Variance**
Sequential	2592.56	2495	3028	2404	234.15	54824.78
Manually Merged	1757.80	1713.5	2111	1628	154.95	24010.84
Task Fusion	1908.90	1974.5	2100	1595	159.45	25424.99
Task Fusion + Visitor Fusion	1884.30	1965.5	2073	1569	179.86	32350.01

M2. Even Distribution (A1 / B9 / D3 / T1 / T16 / T20 / J4 / J15 / J18)						
Task	**Mean**	**Median**	**Max**	**Min**	**StdDev**	**Variance**
Sequential	5373.56	5262	5984	4982	356.94	127406.78
Manually Merged	2100.70	2072	2404	1919	146.87	21571.34
Task Fusion	3753.00	3728.5	4005	3338	220.46	48601.11
Task Fusion + Visitor Fusion	2182.30	2193	2320	1981	97.25	9457.34

M3. Mostly Slow (D3 / T1 / T16 / T20 / J4 / J6 / J11 / J15 / J18)						
Task	**Mean**	**Median**	**Max**	**Min**	**StdDev**	**Variance**
Sequential	7141.22	7032	7801	6681	391.68	153410.69
Manually Merged	1924.00	1923.5	1984	1811	46.13	2127.56
Task Fusion	5224.40	5211	5621	4794	269.36	72554.27
Task Fusion + Visitor Fusion	2359.20	2362.5	2466	2225	80.73	6517.51

Table 5.13: Execution times (in seconds) for mixed workloads

very close to that of the manually merged version. In one case (M1), task fusion alone is enough. For the other two workloads however, visitor fusion was needed. For example with M3, task+visitor fusion is 2.2 times faster than task fusion alone.

5.6.5 Summary

To summarize, task fusion shows substantial improvements over running tasks individually in sequence, taking 33% to 93% less time. Additionally, visitor fusion can give further improvements of up to 7 times faster than using task fusion alone. On a shared, multi-user cluster these optimizations could allow executing at least three times more tasks without providing additional hardware.

5.7 Reproducibility

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

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

To answer this question, we performed a small controlled experiment. We selected a group of 8 researchers: 1 graduate student and 1 post-doc who are experts in software mining and 5 graduate and 1 undergraduate students who are not experts. Each student was given a short tutorial on how to use our infrastructure as well as the location of *Boa* source code for 18 tasks.[4] This source code represents what a researcher would publish in their paper, along with the dataset they used.

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

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

Table 5.14: Study results. All times given in minutes.

The results are given in Figure 5.14 and clearly show that all students were able to reproduce the previously published results in (substantially) less than one hour. Note that all students were also able to reproduce the results on their first try. Thus we assert that using only previously published source code and which dataset was used, other researchers are able to easily reproduce the results.

5.8 Language Comprehension

In this section, we outline a small controlled experiment to determine if our proposed framework and language extensions make it easier to understand source code mining tasks. Each participant was

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

given a short survey to indicate their education level, years of professional software development (if any), and their self-rated experience with the several topics.

- Object-oriented languages, Java, etc
- MapReduce, Hadoop, *Boa*, etc
- Writing compilers, source code analysis
- Design patterns, visitor pattern, etc

Each participant was then shown, one at a time, a set of 5 source code mining tasks written in *Boa*. For each task, they were asked to describe in their own words what the task does. They were given up to five minutes to study each task and forced to move on if no answer was given after five minutes. The five tasks were:

Q1 Count AST nodes

Q2 Assert use over time

Q3 Annotation use, by name

Q4 Type name collector, by project and file

Q5 Finding null checks

These answers were graded on a fixed set of criteria. For each question, we determined a list of criteria that must all be mentioned in order for the answer to be marked correct. For example, for Q1 they had to mention counting only AST nodes (not all nodes) and grouping the count by project.

Then they were shown the same set of 5 tasks again in a random order, only this time instead of a free-form entry they were given a choice of four descriptions and asked to choose the one that best fit. Only one of the four descriptions was accurate while the other four varied slightly (to make them inaccurate). For example, for Q1 only half the responses mention grouping by project. Also only two responses mention counting only AST nodes.

The results are shown in Table 5.15. A 'Y' indicates the participant answered correctly, both in the free-form and the multiple choice. Similarly a 'N' indicates they answered incorrectly in both.

An entry marked '-Y' indicates their free-form answer was incorrect while their multiple choice was correct. Conversely, a '+N' indicates their free-form answer was correct and multiple choice answer incorrect. For the multiple choice, they were also given a choice of 'I am not sure what this task does' which is indicated in the table as '?'. We count this as an incorrect answer.

Q1	Q2	Q3	Q4	Q5	Total (multiple-choice)	Total (free-form)	Time
N	Y	Y	Y	Y	80%	80%	12m32s
-Y	Y	Y	Y	Y	100%	80%	11m22s
?	Y	Y	Y	Y	80%	80%	19m22s
-Y	Y	Y	Y	Y	100%	80%	18m21s
?	+N	Y	Y	N	40%	60%	11m40s
N	Y	Y	Y	-Y	80%	60%	23m01s
N	-Y	Y	Y	Y	80%	60%	16m10s
N	+N	-Y	-Y	Y	60%	40%	14m50s
				Mean	**77.5%**	**67.5%**	**15m55s**

Table 5.15: Controlled experiment on comprehensibility of source code mining tasks in Boa.

On average it took 16 minutes to study these five tasks, or around 3 minutes to comprehend a mining task in *Boa*. There are two ways to grade these responses: by considering the multiple choice answers or by considering the free-form answers. First we grade using the multiple choice responses. The accuracy of the comprehension was at 77.5% on average. Note however that one of the tasks in particular (Q1) seemed to give difficulty. Feedback suggested they failed to understand the semantics of the wildcard. Excluding that task, the accuracy jumps to over 90%.

When grading using free-form answers, five participants scored lower and one scored higher. The overall average dropped to 67.5%, which is still better than two-thirds. Again, excluding the first task the accuracy is very high: 80%.

We repeated this experiment with the same participants six months later. In the repeated experiment, the same five mining tasks were used as in the previous experiment, but this time the source code implementing the tasks was Java+Hadoop code. The results are shown in Table 5.16. Note that all results were anonymized so rows do not correlate to rows in Figure 5.15.

Again, participants spent 16 minutes on average to study these tasks. This time however, the accuracy of comprehension when grading by multiple-choice was lower at 62.5%, almost 15% lower than the *Boa* survey! Note that the participant who scored 100% also marked they had prior experience using Hadoop.

Q1	Q2	Q3	Q4	Q5	Total (multiple-choice)	Total (free-form)	Time
-Y	-Y	N	-Y	-Y	80%	0%	23m44s
?	-Y	-Y	-Y	N	60%	0%	10m50s
-Y	Y	+N	Y	-Y	80%	60%	23m48s
N	Y	N	-Y	N	40%	20%	12m07s
N	-Y	N	N	N	20%	0%	12m08s
-Y	Y	Y	Y	Y	100%	80%	15m52s
N	N	Y	-Y	-Y	60%	20%	18m14s
-Y	+N	Y	N	Y	60%	60%	11m17s
				Mean	**62.5%**	**30%**	**16m**

Table 5.16: Controlled experiment on comprehensibility of source code mining tasks in Java+Hadoop.

Another interesting result was the number of '-Y' responses. There were 15 such responses in the second survey compared to only 6 in the *Boa* survey. This may indicate more guessing or possibly a memory effect where they recalled the answer from taking the *Boa* survey six months earlier. Due to this, when grading by free-form responses the participants only scored an average of 30% accuracy. That is 37.5% lower than the *Boa* survey!

The results from these studies are extremely promising for two reasons. First, it gives insight that in only a few minutes most people can comprehend a source code mining task using our approach. Second, this comprehension comes with **no training at all** in the new language features! Based on the feedback, we believe that even a short training session on *Boa*'s language features would have helped the participants understand Q1 better.

5.8.1 Threats to Validity

Our comprehension study suffers from selection bias as all participants were graduate students. We try to offset this bias by selecting participants from several sub-fields of SE/PL. The study also suffers from testing effects, since each task is given to the participants twice. We offset this effect by randomizing the presentation order the second time tasks were shown. There is also possible construct bias as we chose which tasks to present and might inadvertently select only simple tasks. To counter this, we chose what we considered to be a range from easy to difficult tasks. Finally, there are additional testing effects since the Java+Hadoop portion of the survey was performed after the *Boa* portion. This could actually bias the results in favor of the Java+Hadoop approach.

CHAPTER 6. CASE STUDIES

In this section we present two large case studies to demonstrate the usability of *Boa*. The first study investigates how Java language features have been adopted over time. The second study is a reproduction of a Java feature study, but at a much larger scale.

6.1 Java Feature Usage

The Java Language Specification (JLS) [40–43] is the official specification for Java. New editions of the specification (JLS2–JLS4) are released as the language evolves to add new features. The official Java platforms (Java Runtime Environment (JRE) and Java Development Kit (JDK); Standard (SE), Mobile (ME), and Enterprise Editions (EE)) all implement the language based on this official specification.

Changes to the specification are driven by needs from the community. This need often comes in the form of an official request (a Java Specification Request (JSR)) using the Java Community Process (JCP). The JSR formally defines what the need is, why the current specification is lacking, and proposes a solution. Each new language feature has an accompanying JSR and each new edition of the language has an umbrella JSR to identify the new features.

Currently however, there is little quantitative evidence demonstrating how most of these new language features are used in practice. Previous studies have investigated the use of certain Java language features, e.g. [44] investigated the use of several object-oriented features in Java, such as class, interface, and method usage and [74] investigated the use of generics in Java. Similarly, [58], [23], and [27] investigated the use of non-language features such as reflection (which in Java, is supported by the runtime and not the language). However, these studies looked at a relatively small number of Java projects (around 20), investigated a very small subset of features, or did not investigate their adoption over time.

In this section, we utilize the *Boa* language and infrastructure [5, 31, 33] to study Java feature adop-

tion over time for 18 language features and on a large corpus of projects. The dataset we query is over 31k projects from SourceForge [3], representing over 9 million unique Java source files, with over 28 million snapshots of those files, which when parsed contain over 18 billion AST nodes.

From this dataset we investigate if features were indeed anticipated by the community, by looking for their uses before their release dates. Our results show this is true: every feature is used prior to release. We then investigate how those features are adopted over time along three dimensions: *projects*, *source code files*, and *committers using the features*. Our results show that while some features are widely used, many see only limited use.

We then investigate if these features aren't being used due to lack of opportunity, by defining a set of mining tasks to locate source code that could potentially use these new features. We find millions of such cases, both in files existing before the feature's release date and in new files created after the feature's release. This suggests there is room for better tool support to recommend the use of these new language features or to refactor code to use these features. It also suggests there may be a need for better training and advertisement of new features. Some of our interesting results include:

- All language features were used prior to their official release, indicating anticipation of such features.
- All studied features are used, however a few features are clearly the most popular, including: annotation use, enhanced-for loops, and variables with generic types. Several features saw minimal use.
- Developers do refactor existing code to utilize new language features after their release. Thus, tool support for such refactoring operations and recommendation of code locations to refactor is important for the community.
- We found many instances where features could have been used, but were not, indicating a need for better training or IDE support. In fact, some missed opportunities could actually lead to erroneous behavior.
- Committers tend to adopt new features on an individual basis rather than in a team. This result is consistent with a previous study [74], but with 100 times more committers.

- Most committers use only a small number of new features. A small number of committers account for the majority of new language feature uses.

Next, we give background on each edition of the JLS and the new language features. Then in Section 6.1.2 we pose the research questions our study aims to answer. We describe the approach used in our study in Section 6.1.3 and give the study itself in Section 6.1.4.

6.1.1 Background: Java Language Specifications (JLS)

Since the original edition of the Java Language Specification (JLS) [40], there have been three updates. In this section we outline some of the changes to the language for each edition. The full list of features is shown in Table 6.1. Note that new language features are purely additive - each edition is fully backwards compatible with previous editions.

6.1.1.1 JLS2 New Language Features

The Java Language Specification, edition 2 (JLS2) [41] was a relatively minor update in terms of new language features. This edition added one new language feature: *assert statements*.

6.1.1.2 JLS3 New Language Features

The Java Language Specification, edition 3 (JLS3) [42] added several significant language features, including: *annotation types*, *enhanced-for loops*, *type-safe enumerations* (enums), *generic types*, and *variable-argument methods* (varargs).

6.1.1.3 JLS4 New Language Features

The Java Language Specification, Java SE 7 edition (JLS4) [43] made several changes, including: *binary literals*, a *diamond operator* for generic type inference, *allowing catching multiple exception types*, *suppression of varargs warnings*, *automatic resource management*, and *underscores in literals*.

As these features are not as widely known, we detail some of them in this section.

JLS2	
Assert	`assert i > 0;`
JLS3	
Annotation Declaration	`@interface Test { }`
Annotation Use	`@Test void m() { .. }`
Enhanced-For Loop	`for (T val : items) ..`
Enums	`enum E { N1, ..; }`
Generic Variable	`List<T> l;`
Generic Method	`<T> void m(T a) { .. }`
Generic Type	`interface List<T> { .. }`
Extends Wildcard	`Class<? extends E> c;`
Super Wildcard	`Class<? super S> c;`
Other Wildcard	`Class<?> c;`
Varargs	`void m(T... arg) { .. }`
JLS4	
Binary Literals	`int FIVE = 0b101;`
Diamond	`Map<K, V> m = new HashMap<>();`
MultiCatch	`catch (E1 | E2 e) { .. }`
Safe Varargs	`@SafeVarargs`
Try with Resources	`try (File f = new ..) { .. }`
Underscore Literals	`int MILLION = 1_000_000;`

Figure 6.1: Studied Java language features, with examples.

Type Inference for Generic Instance Creation (Diamond) As previously mentioned, the language allows generic types. When declaring a variable of a generic type however, the generic type arguments must be repeated. For example:

```
Map<K, V> m = new HashMap<K,V>();
```

declares a `HashMap` with keys of type `K` and values of type `V`. Note that the generic type arguments were repeated both in the variable declaration (left) and the object instantiation (right). This edition allows omitting the repeated generic type arguments in the instantiation (the so called *diamond operator*), thus changing the previous example to:

```
Map<K, V> m = new HashMap<>();
```

This new diamond operator can be used anywhere the compiler is able to infer the generic type arguments.

Catching Multiple Exception Types (MultiCatch) This edition allows specifying more than one exception type inside a catch clause. The catch clause's body is then executed when either exception type is caught. For example, the statement:

```
try { .. } catch (E1 | E2 e) { .. }
```

executes the catch statement's body if the try statement throws an exception of type `E1` or type `E2`. This helps avoid code duplication.

Safe Varargs Warning Suppression The variable number of arguments in methods feature added in JLS2 can lead to a large number of compile-time warnings when combined with generics. Often however the programmer knows that these warnings can safely be ignored, so the ability to disable those warnings was added:

```
@SafeVarargs
@SuppressWarnings({"unchecked", "varargs"})
static <T> List<T> asList(T... elems) { .. }
```

The use of either of these annotations will suppress compiler warnings at this location.

Try with Resources Certain resources, such as files, require manually releasing them when finished. This by itself is easy to forget, however even when programmers remember to close the resource, errors can still creep in [91]. To ease the management of these resources, a new statement was introduced:

```
try (File f = new ..) { .. }
```

This try statement declares a resource `f` which is available within the try statement's body. Upon exiting the try statement (either through normal or exceptional program flow) the resource is automatically released.

6.1.2 Questions Regarding Language Feature Use

The focus of our study is the usage of Java language features by open-source developers. In this section, we outline the specific research questions (RQ) we wish to answer.

RQ1: *Do projects use new language features before the features are released?* Often, especially with Java, an implementation of a requested feature is available before its release. This can take the form of an official beta/pre-release or an unofficial compiler.

We are interested in how often new language features are used prior to their official release. Such data can give an indication if a particular feature was anticipated and if providing implementations prior to release may be useful to the community.

RQ2: *How frequently is each language feature used?* The next question deals with feature usage. The addition of language features is driven by needs from the community, yet to date there has been no study to see how most of Java's language features are being adopted by developers.

This question examines language features introduced in JLS2–JLS4. For each language feature, its use across our entire dataset is tracked. This data gives insight into how each feature was, and is, being used.

RQ3: *How did committers adopt and use language features?* Once a new set of language features is available, it takes time for developers to learn how and where to use them. Some developers may be excited and try using them as often as possible. Other developers may be content with solving problems with the old set of features, as that is what they are accustomed to. We wish to investigate to see how language feature adoption occurs for individual developers.

RQ4: *Were there missed opportunities to use language features?* Although a new language feature may be available, developers might chose to not use it. We are interested in knowing how often such missed opportunities exist.

RQ5: *Was old code refactored to use new language features?* We also wish to investigate to see if code using older language features is ever updated to use the newer language features.

6.1.3 Approach: Dataset

In this section, we describe our approach for answering the previously identified research questions. Our approach relies on *Boa* [5, 31, 33] and its dataset from SourceForge [3].

6.1.3.1 Dataset Used in Our Study

The dataset used for our study is the September 2013 dataset from *Boa*. This dataset includes all Java projects on SourceForge with at least one CVS or Subversion repository. The dataset does not include Java projects with only Mercurial, Git, or Bazaar repositories. The total number of Java projects is over 35k (see Table 6.1).

Metric	Count
All Projects	*699,331*
Java Projects	*35,341*
Studied Projects	**31,432**
Repositories	*32,555*
Revisions	**9,557,448**
Files	*41,733,495*
File Snapshots	*86,411,272*
Java Files	**9,093,216**
Java File Snapshots	**28,747,948**
AST Nodes	**18,323,905,323**

Table 6.1: Metrics for the SourceForge-based dataset in Boa.

However, not all of these projects are useful. We identified almost 4k projects that did not have at least one Java source file that parsed without error. We filtered those projects out, leaving over 31k projects in the dataset for use in our study.

The dataset contains widely-used Java projects, including: Azureus/Vuze, Weka, Hibernate, JHotDraw, JabRef, JUnit, iText, FindBugs, JML, TightVNC, etc. This dataset represents over 9 million revisions by more than 50k developers. It contains over 9 million unique Java files and over 28 million snapshots of those files. This represents (to the best of our knowledge) the largest empirical dataset to date for Java projects that contains both full history information of the source repositories with over a decade of history and the full AST information from the Java source files.

For our research questions, the size of the Java projects (whether 1 or 1k files) is irrelevant, as we are interested in investigating Java language features used by developers without constraining the study to any specific kind of developer. Thus we include small projects (perhaps written by novice developers) as well as large projects (perhaps written by experts). However for RQ3, smaller projects could affect our results and thus as we mention later, for this research question, we filtered projects with

few developers.

6.1.4 Study: Analyzing Java Language Feature Adoption

In this section we investigate Java language feature usage.

RQ1: Do projects use new language features before the features are released? If a feature is requested by the community, then most likely people will be excited to use it prior to its release. To see if this is true, first we needed to know the release dates of official implementations for each language specification. We show these release dates, based on each specification's JSR, in Table 6.2.

JLS2 (JSR 59) - Released 09 May 2002			
Feature	**Earliest Use**	**Projects**	**Files**
Assert	09 Feb 1998	114	1,068
JLS3 (JSR 176) - Released 30 Sep 2004			
Feature	**Earliest Use**	**Projects**	**Files**
Annotation Declaration	11 Nov 2003	7	130
Annotation Use	05 Jan 2002	12	1,165
Enhanced For	20 Jan 2002	44	634
Enums	05 Jan 2002	20	173
Generic Variable	11 Jul 1998	59	2,311
Generic Method	04 May 1999	22	919
Generic Type	01 Jul 1998	31	2,047
Extends Wildcard	02 Jan 2002	18	587
Super Wildcard	24 Jul 2003	3	426
Other Wildcard	10 Feb 2002	23	649
Varargs	23 Jul 2003	10	76
JLS4 (JSR 366) - Released 20 Jul 2011			
Feature	**Earliest Use**	**Projects**	**Files**
Binary Literals	04 Nov 2010	2	4
Diamond	01 Aug 2010	12	399
MultiCatch	01 Aug 2010	9	95
SafeVarargs	30 Apr 2011	3	17
Try with Resources	04 Nov 2010	8	109
Underscore Literal	04 Nov 2010	2	2

Table 6.2: Language features are used before their release. (Note: cutoff times were midnight UTC on release date)

Using the release dates in this table, we then analyzed each valid Java file to see if it used a particular feature. We filtered out any Java file containing a parse error. Then we collected the timestamps of each

file using each language feature and then filtered based on the particular language feature's release date. The results are shown in Table 6.2 and include the list of features, the date of the first mined use of the feature, the number of projects that used the feature prior to its release, and the total number of files that used the feature prior to its release.

For the earliest uses, we manually investigated to verify the identified files actually used the particular feature and that the commit date matched our results. Based on this analysis we identified one project with clearly erroneous commit dates[1] and we removed that project from this analysis. Interestingly, for one project that made heavy use of generics in 1998, the commit log referenced "switch[ing] to GJ", which is the language extension proposed by Bracha *et al.* [21] that eventually became the basis of Java's generics.

The results in the table clearly show that every language feature was used prior to its official release date. Next we investigate how each language feature was adopted over time.

RQ2: How frequently is each language feature used? The addition of language features is driven by needs from the community. In this section, we quantitatively investigate how developers use these new features by looking at each unique Java source-file path in the system and taking the last existing snapshot of each. We then analyze that set of snapshots and count feature usage. For each file, we generate a mapping between features and the total uses in the file.

We show the results in Table 6.3, first by total number of uses across the entire dataset, then by percent of Java files using the feature, and finally by percent of projects using the feature. The table clearly shows every feature is being used at least once. One trend that becomes readily apparent is that JLS4 features are not used very often, compared to JLS3 features. This is despite the fact there were over a million revisions and 3k Java projects active since the release of JLS4.

Also observe the trends for the ratios of uses to files. For example, the Annotation Declaration feature has a ratio close to one[2]; there is roughly one annotation declaration per file. This is similar for Enums and Generic Type. These features represent types in Java and thus one generally expects to see one type per file. The ratios for the other features are higher (2–6) since they are expressions and

[1] http://goo.gl/Hqn6qZ

[2] This feature appears in 0.28% of files, or around 25k files, and is used around 29k times total.

JLS2	Assert
Uses	408,802
File	1.04%
Project	12.72%

JLS3	Annotation Declaration	Annotation Use
Uses	29,415	11,692,911
File	0.28%	21.98%
Project	6.35%	57.07%

JLS3	Enhanced For Loop	Enums	Generic Variable	Generic Method
Uses	2,666,411	162,445	2,473,581	257,921
File	8.4%	1.47%	9.24%	0.9%
Project	48.61%	29.42%	57.15%	13.52%

JLS3	Generic Type	Extends Wildcard	Super Wildcard	Other Wildcard	Varargs
Uses	214,012	411,940	84,602	936,546	221,322
File	1.89%	1.37%	0.15%	2.45%	0.95%
Project	19.87%	15.67%	2.71%	22.52%	15.43%

JLS4	Binary Literals	Diamond	MultiCatch	Safe Varargs	Try with Resources	Underscore Literals
Uses	90	22,473	1,920	192	1,597	889
File	0%	0.08%	0.01%	0%	0.01%	0%
Project	0.02%	0.4%	0.27%	0.06%	0.21%	0.02%

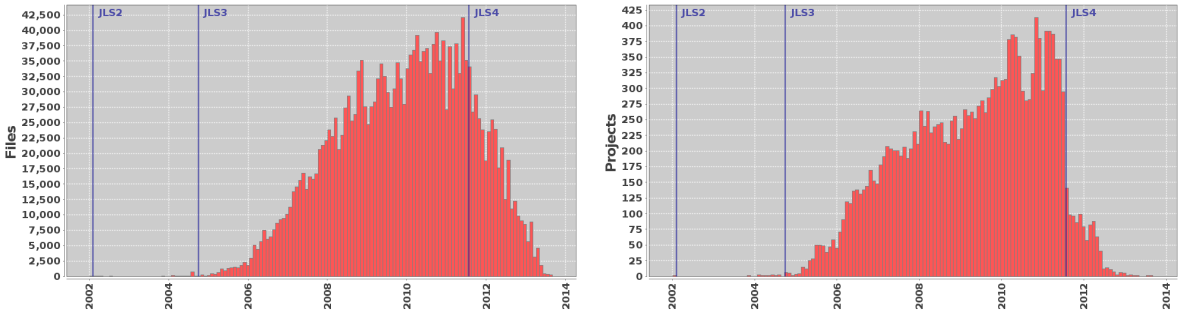
Table 6.3: Java language feature usage by total number of uses, by percent of all files, and by percent of all projects.

statements. For example, the ratio of enhanced-for loops is three[3] meaning files using the feature use it around three times.

To see how the features were adopted over time, we plotted histograms of each feature's use, both by number of files and by number of projects. The histograms contain bins with 30-day time ranges. The first time a feature appears, it is added to the respective bin. See Figures 6.2a–6.2b and Figures 6.3a–6.3b. The plots also contain marker lines to indicate the release date of each JLS.

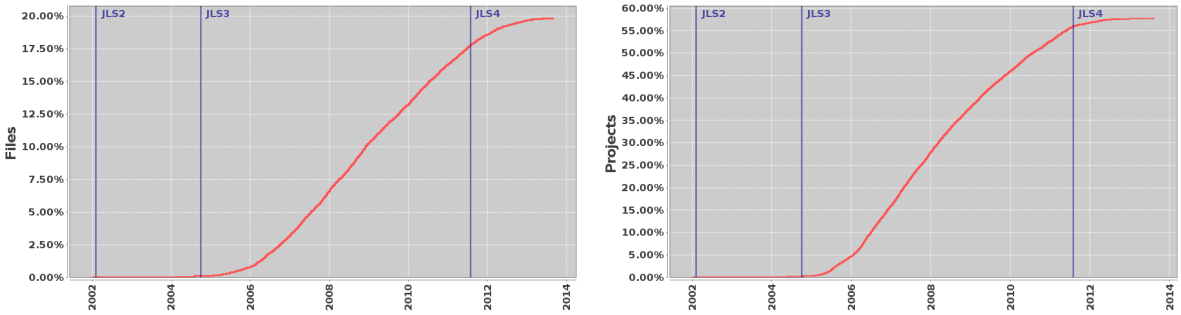
We also plotted densities of each feature's use, both by number of files and by number of projects. Points in these charts represent the number of files/projects using a feature at that time, divided by the

[3]This feature appears in 8.4% of files, or around 763k files, and is used around 2.6m times total.



(a) First uses, by File

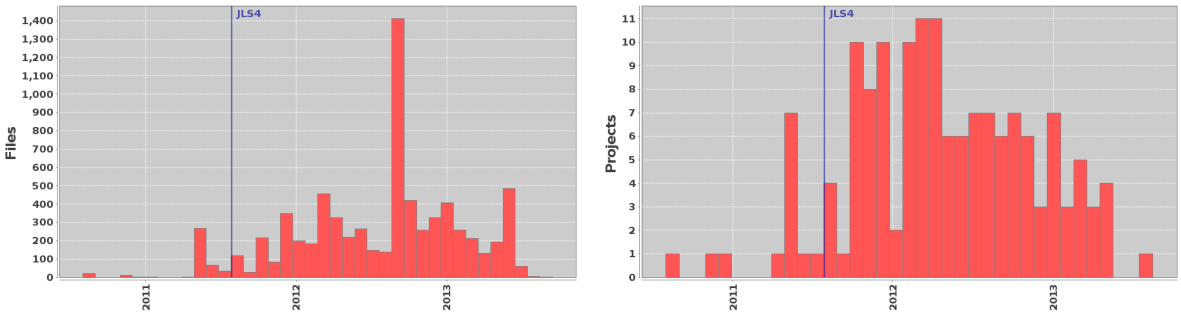
(b) First uses, by Project



(c) Use Density, by File

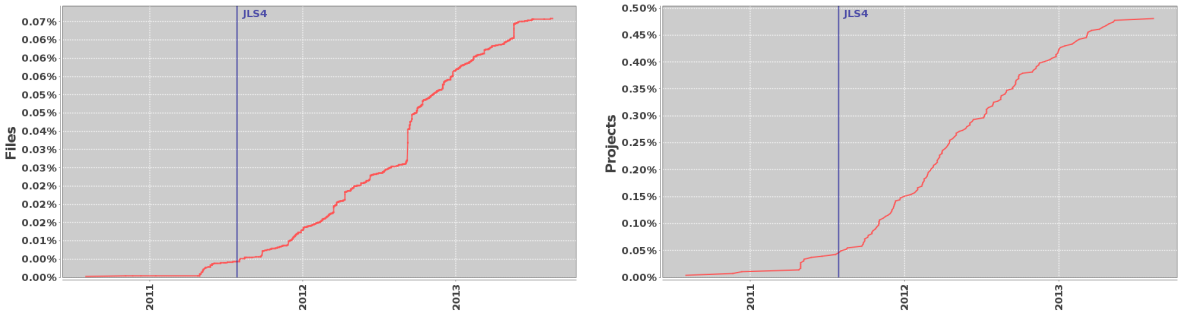
(d) Use Density, by Project

Figure 6.2: Use of the *Annotation Use* language feature.



(a) First uses, by File

(b) First uses, by Project



(c) Use Density, by File

(d) Use Density, by Project

Figure 6.3: Use of the *Diamond* language feature.

total number of Java files/projects at that time, to account for growing repositories. See Figures 6.2c–6.2d and Figures 6.3c–6.3d.

After examining these plots for each feature, we noticed similar trends among the features. They fell into two categories: JLS4 features and non-JLS4 features. Since the trends are similar across features, we picked representatives from each category.

For example, Figure 6.2 shows a non-JLS4 feature, Annotation Use. The histograms all show increasing adoption of the features after release with peaks around 2011. Then the number of files/projects adopting the feature for the first time starts decreasing. To better understand this decrease, we investigate the density plots.

As can be expected, the files and projects in the system were increasing over time. The density plots remove this variable from our analysis, by computing the percent of feature use at each time. For example, when we look at Figures 6.2c–6.2d we can see that even as the total number of files and projects in the system increases, the relative percent is increasing too. Thus we can see that over time, the use of the feature is increasing. This trend is apparent for all features studied.

Notice that Figure 6.3, a JLS4 feature, doesn't show as strong of trends as the previous two features discussed. In this chart, the histograms have less of an obvious trend to them, due to the relatively low number of total uses for this new feature. While the density graphs still show the same general trend of increasing use, both by files and by projects, there is less of a defined curve in these graphs.

Investigating Frequently Used Features As seen in Table 6.3, most language features are used in a very small number of files (2% or less). The exceptions are Annotation Use, Enhanced For, and Generic Variable declarations. We further investigate some of these popular language features.

First let's look into the use of annotations, by collecting the annotation types named at each use. Table 6.4 shows the top-ten frequently used annotation types and the number of uses for each. As can be seen, almost half of the annotation uses were the `@Override` annotation. Such widespread use of this annotation makes sense as IDEs such as Eclipse typically automatically add this annotation. The second most used annotation, `@Test`, is used by unit testing frameworks. In fact, other than `@Test` and `@SubL`, the annotations listed are all JDK or J2EE provided annotations. We anticipated high use of JDK annotations, as the Annotation Declaration language feature has less than 0.3% use across all

Java files, but the clear domination of those annotations was surprising.

Annotation Name	Uses	Percent
@Override	5,534,089	47.33%
@Test	981,737	8.40%
@SuppressWarnings	634,697	5.43%
@Column	246,467	2.11%
@XmlElement	140,754	1.20%
@SubL	134,990	1.15%
@Generated	131,759	1.13%
@XmlAttribute	101,156	0.87%
@XmlAccessorType	81,140	0.69%
@Deprecated	80,217	0.69%

Table 6.4: Annotation uses. Percents are out of all annotation uses.

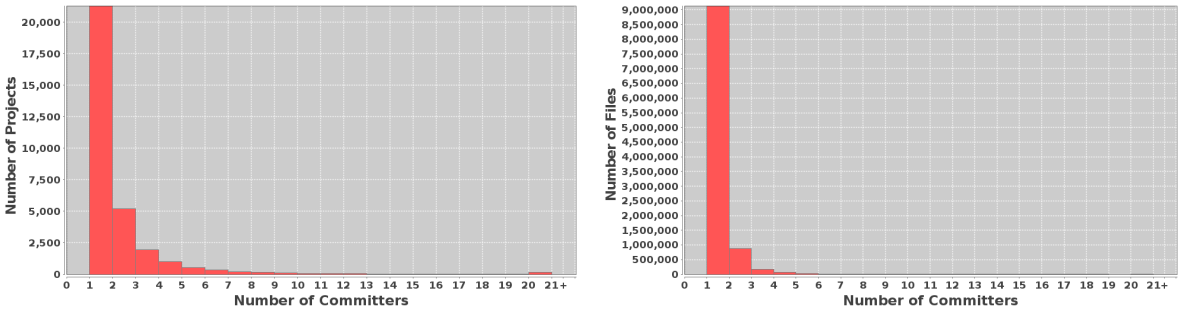
Generic Type	Uses	Percent	Generic Type	Uses	Percent
List	3,628,998	32.31%	List<String>	514,339	22.68%
ArrayList	2,145,612	19.10%	ArrayList<String>	416,306	18.35%
Map	1,156,480	10.30%	Class<?>	295,554	13.03%
HashMap	842,934	7.50%	Map<String, String>	208,195	9.18%
Set	811,990	7.23%	Map<String, Object>	177,048	7.81%
Collection	643,047	5.73%	Set<String>	170,727	7.53%
Vector	570,016	5.07%	HashMap<String, String>	148,861	6.56%
Class	547,628	4.88%	Vector<String>	137,706	6.07%
Iterator	500,887	4.46%	HashSet<String>	110,424	4.87%
HashSet	384,408	3.42%	HashMap<String, Object>	89,088	3.93%

Table 6.5: Variables declared with generic types.

Next let's look into the generic variable declarations, by collecting the counts of each declared generic variable's type. Table 6.5 shows the top-ten frequent generic types used (top) and the top-ten parameterized types (bottom). The results clearly show that the majority of generics are from collection types, the most common being `List<String>`. These results are consistent with the previously published study on generics use by Parnin *et al.* [74], although our study was on a thousand times more projects.

RQ3: How did committers adopt and use language features? While in RQ1 we showed that all features are adopted before their release, and in RQ2 we showed how features are adopted over time, so far we have only evaluated feature adoption in terms of files and projects. In this section, we wish

to evaluate if similar adoption trends also apply in terms of committers. Specifically, we also wish to study the adoption behavior of individual committers.



(a) Number of committers in a project

(b) Number of committers editing a file

Figure 6.4: Number of committers per-project and per-file in SourceForge.

To do that, for each changed or added file that was recognized as containing a feature, we collected its commit time and author. For each commit that has changed files containing the use of a feature for the first time, the corresponding author is counted as one committer using that feature. The number of committed files containing the new features are also recorded and counted toward the number of uses for the corresponding committer. The threat to this method of counting is that if a committer uses a feature in a file which has already contained that feature (introduced by some other committer), they would not be counted. However, in this dataset a file is usually owned and edited by one or a few committers (as shown in Figure 6.4b), thus our results are not affected much.

RQ3.1: How many committers adopted and used new features over time? Figures 6.5 and 6.6 show the result for the number of committers using two different features over time. Each bar shows the number of users in the corresponding month. Even though the features appeared at different times, both show the same trend of adoption: a few committers used the feature before its release, then the number of users increases to a peak, and finally decreases. This is consistent with the adoption trend for projects and files seen in RQ2.

Among the committers using a feature, we counted the ones who used that feature for the first time (the lower area, in red) and the ones who had used that feature before (the upper area, in blue). As seen in the figures, after the release date more committers adopted the new features. Once a feature is used for the first time, many committers kept using it in later commits (in blue). After a while, the number

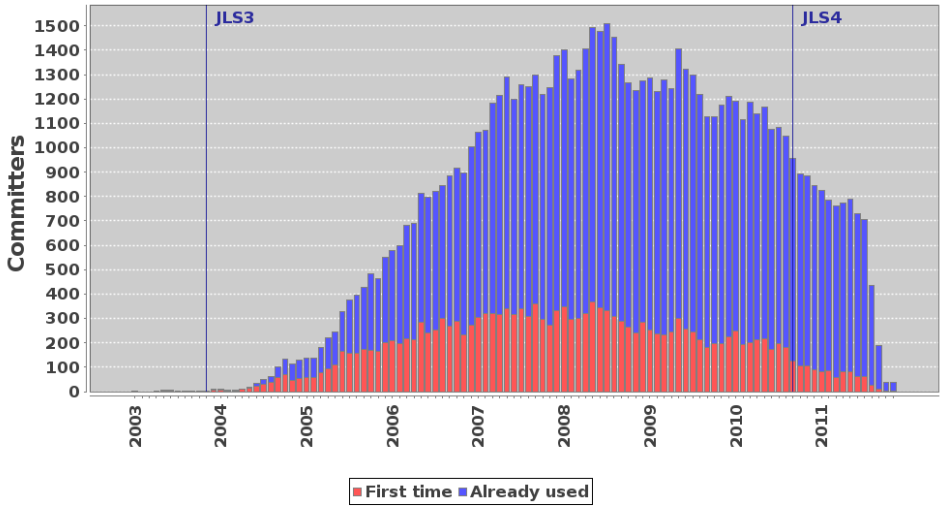


Figure 6.5: Committers use of Annotations over time.

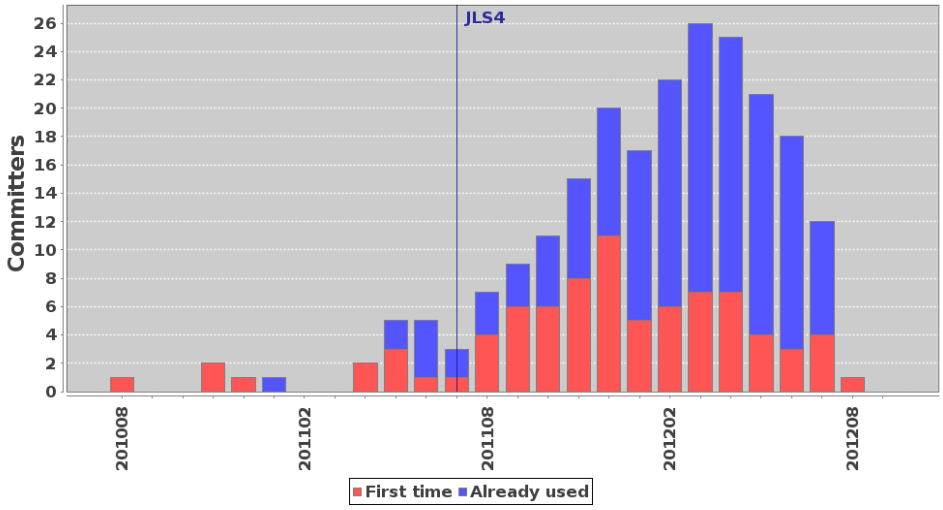
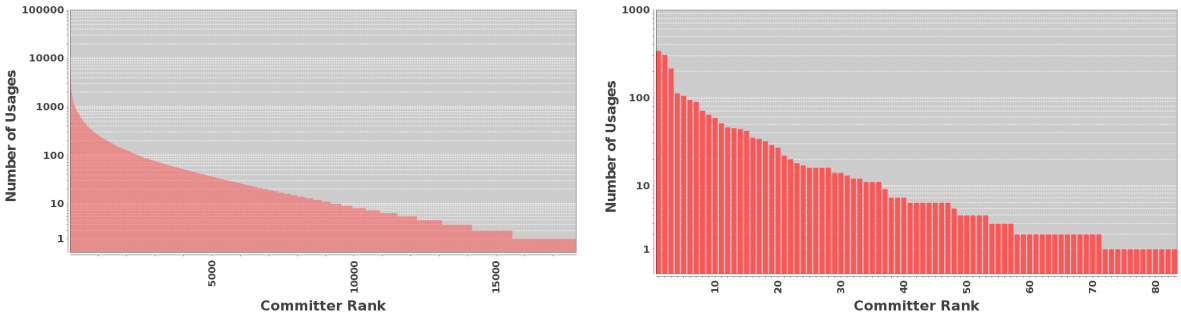


Figure 6.6: Committers use of Diamond over time.

of first-time users (in red) decreases. This trend is the same for all the features in our study.

Comparing the charts, the number of committers using Annotations is much higher than that for Diamond. This result is expected and is consistent with RQ2.

RQ3.2: How much did committers use each feature? To answer this question, we count the number of uses of each feature for each committer. Since different features are used at different levels of granularity in the source code, e.g. generic fields can only be declared at the type level while enhanced-for loops can be used multiple times in the body of the method, we used the granularity of files to compute the number of uses. That is, the number of uses for a committer is the number of files to which



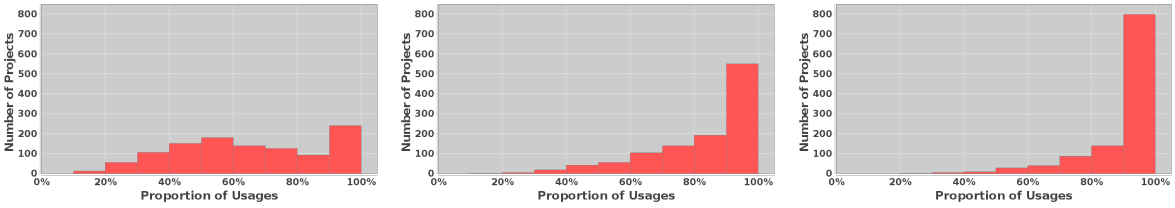
(a) Annotation Use (b) Diamond

Figure 6.7: Use of language features by committers.

that committer was the first one introducing that feature.

Figure 6.7 shows the result for two features: Annotation Use (6.7a) and Diamond (6.7b). In each chart, the x-axis represents the committers ranked by their number of uses and the y-axis (in logarithmic scale) is the number of uses. Each bar represents the number of uses for a single committer. The charts consistently show a power-law trend and the number of uses is highly skewed. A small number of committers accounts for a large number of feature uses. About half of the number of committers introduced a feature to less than 10 files, while a few committers used the feature in tens, hundreds, to thousands files. This trend holds for all features.

Comparing the charts, we can see that the number of committers are quite different: about 18,000 for Annotation Use (6.7a) and about 85 for Diamond (6.7b). In addition, the number of committers with the same number of uses varies among features. For example, at 10 uses, there are about 9,000 and 34 committers, respectively. This suggests that there are some popular feature(s) which are widely-used (e.g. Annotation).



(a) Top-1 Annotation Use committer (b) Top-2 Annotation Use committers (c) Top-3 Annotation Use committers

Figure 6.8: Proportion of feature uses in projects.

6.1.4.1 RQ3.3: Did committers adopt features on an individual basis or as a team?

To answer this question, we investigated how many team members adopted features in each project. We first collected the set of committers for each project, identified how many times each committer used a feature, and ranked the committers per-project based on their number of uses. Then, for the top-k committers (k=1,2,3), we computed the proportion of their uses over the total number of uses in the whole project.

In Figure 6.4a, we can see that the distribution of the number of committers in a project is right-skewed. That is, many projects have only a few committers. In those projects, only one or two committers contribute almost 100% of the uses. To avoid that bias and to study the team culture, we filtered out all projects having less than six committers.

The result is shown in Figure 6.8. Each chart shows the histogram of the proportion of feature usage in projects adopting that feature. The bins are the ranges 1-10%, 11-20%, ..., and 91-100%. Figure 6.8a shows the result for the top-1 user. One committer contributes 100% uses in almost 250 projects and 80% uses in about 100 projects. In Figure 6.8c, when considering the top-3 users, the number of 100% uses increases to almost 800 projects and are almost all the projects using that feature. The other features (not shown) follow similar trends.

This result indicates that a feature is not widely adopted by all members of the team, but instead are mainly championed by a small number of members. This is also consistent with the finding by Parnin *et al.* [74] even though they studied only 20 projects while we studied almost 1,800 projects (with at least 6 committers each).

RQ3.4: Did committers use all new features? For this question, we track the feature uses of a group of "active" committers, who routinely committed code over a long enough period. Since JLS3 had the most new features, we used the set of committers at the release time of JLS3. We kept all committers that had routinely committed code at least every 6 months in the time between releases of JLS3 and JLS4. Filtering for committers that used at least one language feature in our study, the remaining set contained 61 committers. The scatter graph in Figure 6.9 shows their uses over time.

For better visualization, we group related features from the same edition, i.e. Annotation Declaration/Use into Annotation, all generics features into Generics, Binary/Underscore Literals into Literal,

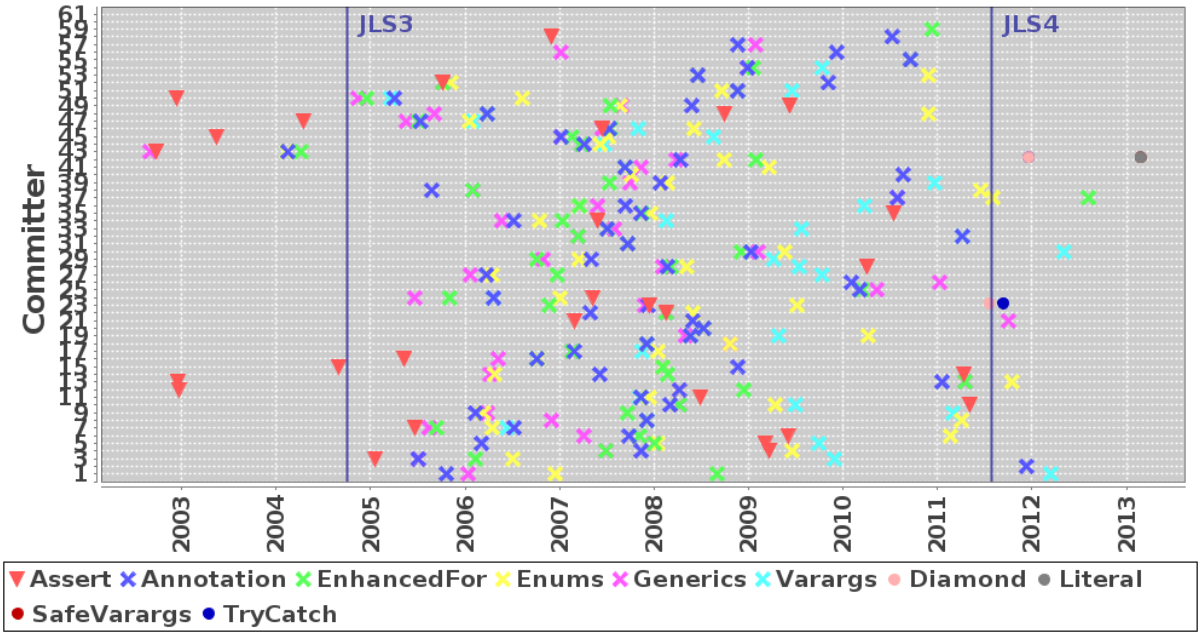


Figure 6.9: Tracking features used by committers.

and Try with Resources and MultiCatch into TryCatch. Each horizontal line shows the use over time for a single committer.

As seen from the graph, among the 61 committers only committer #24 adopted features from all three editions. Most committers used features from JLS2 and JLS3. JLS4 was only used by committers 42 and 24. Most of the committers used Assert, the only new feature in JLS2, however, they started late after its release. Meanwhile, the committers adopted JLS3 quite early and most of them used several different features. In terms of individual feature uses, up to now, no committer has used all studied features. Committers have used at most 7 out of 10 different grouped features.

RQ4: Were there missed opportunities to use language features? In this section we investigate missed opportunities to use new language features, by mining the latest snapshot of source code to find locations where new language features could potentially be used. For example, we mined to find integer literals with 7 or more characters that did not use underscores. We also mined methods that have as their first statement an if condition that if true throws an `IllegalArgumentException` (which could potentially be turned into an assert statement), methods that take an array as last argument instead of a varargs argument, expressions where the literal '1' was shifted left (which could use binary literals),

generic instantiations that don't use the diamond pattern, try statements with more than one catch block having the same body, and try statements with a call to a `close()` method in the finally block.

The results are shown in Table 6.6. In the first row, we list the number of mined potential uses in files that existed prior to the feature's release. These represent places where maintainers could refactor code to use new language features. We found tens of thousands (to millions) of potential uses.

	Assert	Varargs	Binary Literals	Diamond	MultiCatch	Try with Resources	Underscore Literals
Old	89K	612K	56K	3.3M	341K	489K	22.2M
New	291K	1.6M	5K	414K	24K	33K	2.3M
All	380K	2.2M	61K	3.7M	365K	522K	24.5M
Files	1.39%	12.74%	0.11%	12.25%	2.28%	1.85%	20.17%
Projects	18.18%	88.78%	5.9%	59.08%	49.75%	37.27%	88.86%

Table 6.6: Potential language feature uses, in old files (before feature release) and new files (after feature release).

The second line of the table shows potential uses in files that were added after the release of the feature. These are locations that developers had the option to use a language feature, but did not. Again, we found thousands of potential uses for each feature and even millions for two features.

While some of this unused potential has small impact, such as underscore and binary literals making code more readable, other missed opportunities could actually lead to erroneous behavior. Specifically, we investigate regarding the try with resources language feature which aims to properly close resources. As Weimer and Necula [91] point out, this is a common source of bugs in programs. For example, the code:

```
BufferedReader br = ...;
String s = br.readLine();
br.close();
```

wouldn't call `close` if the call to `readLine` throws an exception.

While we found over 500k potential uses for this language feature, we were interested in how many of those might lead to buggy behavior. We narrowed the results of the algorithm to only include methods that throw `IOException`, do not catch that exception anywhere in the body, and contain a call to a `close()` method. We found 193,768 instances of potential resource handling bugs!

RQ5: Was old code refactored to use new language features? As we showed in the last section, when new language features are released there is potentially a lot of existing code that could have used the new feature. In this section we investigate if developers refactor old code to update it to the new language features.

Unlike the last section where we used only the latest snapshot, in this analysis we mine each version of a file and compute the number of potential and actual uses of a language feature. We then compute those values on the previous version of the file. If the number of potential uses decreases by **exactly** the amount the actual uses increased, we consider it a potential refactoring. This analysis is extremely conservative and may miss a lot of refactorings, but it should give a low number of false positives and make verification easier, and allows us to confirm the existence of refactoring activities to use new features. We show the results in Table 6.7.

	Assert	Varargs	Diamond	MultiCatch	Try with Resources	Underscore Literals
Count	180	2.1K	8.5K	162	154	2
Files	105	1.6K	3.8K	125	99	1
Projects	37	488	72	23	17	1

Table 6.7: Detected refactorings to use new features.

We verified the results by manually checking 30 detected files from each feature, grouping the results by project and systematically sampling from a random starting point in the list. When verifying a file, if other files from the **same revision** were in the dataset we also verified those. In total we verified 2,598 out of 5,694 files as direct refactorings, 13 as not refactorings, and 4 as more complex refactorings that also added the new feature.

During this process we found several commit logs mentioning upgrades to JDK7 or specifically refactoring for one feature (such as Diamond, MultiCatch, or Assert). One even stated "Reviewing locations where 'throw' appears and substituting by 'assert' when convenient."[4]

As we showed, developers do refactor existing code to utilize new language features after their release. Thus, tool support for refactoring operations to use new language features and recommendation of code locations to refactor is important for developers.

[4] http://goo.gl/5pyR0T

6.1.5 Threats to Validity

We identified a threat regarding who commits code versus who actually wrote that code. Someone may commit a file they did not write, perhaps adding a file from another library so it is local in their own repository. Our analysis would attribute the source of that file to the person who committed it which is why we focused on committers, not developers.

A similar threat relates to the timestamps of committed code. If someone commits a file they did not write, the timestamp of the commit may be wrong. It is possible that features were actually used earlier than identified in RQ1.

We identified an external threat to our study regarding the generalizability of our results. Since we only studied open-source software, the results may not necessarily represent Java language feature usage by non-open source developers, such as those in industry. We also do not know the experience level of committers, which may vary greatly and limits our ability to generalize. We avoid generalizing our results and instead focus on if the trends we observed are similar to the trends the previous study by [74] observed.

6.2 Treasure Study Reproduction

Grechanik *et al.* performed a large-scale empirical study on Java source code from 2,080 open-source projects [44]. The dataset used in their study were randomly selected projects from SourceForge. For their study they built an SQL database containing tables and attributes for storing (non)terminals from Java's grammar. They posed 32 different research questions and queried their database to answer them. To show the usefulness of our approach, we reproduced a portion of this study using *Boa*.

As the actual queries used are not available, we had to make a few assumptions about their study. First, we assumed that none of the projects in their study were empty and all contained at least one valid Java source file. This assumption was made on the basis that their minimum value for number of classes per application is 1. Thus, we filter our dataset to exclude any projects without at least one valid Java source file. This left 23,510 projects in our study.

Second, although their paper only mentions parsing the source code, we assume that since they were only working with releases of each project that they also had type resolution and bindings. This

	Total		Mean		Median		Max		Min	
Question	**Boa**	**Treasure**	**Boa**	**Treasure**	**Boa**	**Treasure**	**Boa**	**Treasure**	**Boa**	**Treasure**
Classes	11,822,321	270,973	503.68	96.8	89	33	139,668	2,071	1	1
Static Classes	569,501	7,368	24.25	6.7	0	0	23,744	1,035	0	0
Anonymous Classes	3,772,130	29,237	0.05	0.04	0	0	724	136	0	0
Nested Classes	1,218,213	14,270	51.86	0.06	3	0	30,576	61	0	0
`assert` Statements	612,166	2,047	0.01	0	0	0	374	9	0	0
Methods	68,062,962	938,779	5.89	3.5	2	4	32,774	1,175	1	1
Static Methods	5,696,065	231,647	0.48	0.36	0	0	4,853	289	0	0
Methods (interfaces)	4,712,116	84,130	6.13	3.4	3	3	10,000	558	1	1
Method Arities	66,778,747	544,324	1.59	1.5	1	1	262	30	1	1
`void` return Methods	35,988,971	414,953	3.54	5.1	2	3	32,772	1,172	1	1
Methods Returning Arrays	1,334,259	24,744	1.87	2	1	1	383	137	1	1
non-`void` return Methods	32,073,991	523,826	4.93	5.8	2	3	4,854	888	1	1
Fields	31,682,721	448,898	2.68	1.9	0	0	10,000	1,457	0	0
`this` Expressions	51,933,214	840,937	0.72	2.2	0	1	6,294	785	0	0
Static Fields	10,949,191	154,067	0.93	0.7	0	0	10,000	1,457	0	0
Volatile Fields	48,471	492	0	0	0	0	97	9	0	0
Conditional Statements	118,557,128	620,419	1.63	0.76	0	0	5,294	750	0	0
`String` Fields	6,425,161	231,647	0.54	0.3	0	0	3,473	432	0	0
`try` Statements	14,080,420	93,714	0.19	0.11	0	0	1,722	90	0	0
Exceptions Thrown From `catch`	4,559,274	110,740	0.3	0.26	0	0	34	5	0	0
Exceptions	12,631,996	818,358	0.17	0.9	0	0	1,086	40	0	0
Local Variables	79,057,404	818,358	1.09	0.87	0	0	7,005	1,055	0	0

Table 6.8: Reproducing a portion of the Treasure study [44], at a much larger scale.

information is not yet available in *Boa*, so we do not reproduce the six tasks that rely on that information for accuracy.

Finally, we assume that the versions of each project used in their study were the latest versions. As *Boa* contains all revisions for projects but does not currently know what revision(s) map to specific releases, for our version of the study we simply take the latest snapshot of each project. We also filter out any obvious branches (in SVN, branches typically are rooted in the 'branches' folder). This gave a total of 8,360,673 changed files in this snapshot, or about one third of the total dataset.

The results of our study, as well as the values from the previous study, are shown in Table 6.8. The statistical values (mean, median, max, and min) are computed using the most logical container for each question. For example, the container for classes are projects, the container for methods are classes, etc. As can be seen, most values differ between the studies. This is to be expected, as there are over 11 times more projects in our study. However, note that the general trends are similar and in particular the order of magnitude between rows is maintained.

For our version of the study, some of the values in the max column seemed like they might be too high. We manually verified[5] these values to be correct. Some interesting results:

- Despite the Java VM having a limit of 255 arguments for a method, we located a class constructor with 262 arguments!
- We located a test class with over 32k (hopefully generated) `void` methods in it to exhaustively test a method's 16-bit integer argument.
- A compiler-generated X10 file with over 7k local variables.
- A class with 10k static fields as constant strings.

The one task where we differ substantially is for nested classes. Note the mean value is almost 1k times higher. We believe this is because their study averages nested classes by number of methods. However we disagree with this, as the most common container for a nested class is another class. Thus we opted to compute this value slightly different.

[5] http://goo.gl/bwGGC http://goo.gl/jf0Fy http://goo.gl/zuYoh http://goo.gl/ZgamQ

6.2.1 Threats to Validity

The results of our reproduction of the Treasure study in may not generalize to Java development practices in industry, as all of the code in our study comes from open-source. This same threat applies to the original study [44]. We avoid generalizing our results and instead focus on if the trends we observe are similar to the trends the previous study observed.

CHAPTER 7. RELATED WORK

In this section we discuss works related to *Boa*. First we discuss other mining software repository frameworks. Then we discuss works that provide data-parallel frameworks. Then we discuss languages intended to ease data-parallel computing. We compare our optimizations to existing MapReduce optimizations. We also discuss related approaches utilizing the visitor pattern and finally discuss language studies related to our case studies.

7.1 Mining Software Repository Frameworks

Sourcerer [4,16,57,71] 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 full history information (revisions).

Bevan *et al.* 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* [19]. 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 Kenyon, with its database solution, data schema evolution is not easily supported.

In 2007, Boetticher, Menzies and Ostrand introduced the PROMISE Repository [1], 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.

Supporting for the reproducibility of research papers published in the MSR area, Gregorio Robles [37] and his team advocated for the construction of open-access data repositories for MSR research. Their goal was to build "a web page with the additional information, most desirably a Sourceforge-like site that acts as a repository for this type of data and tools, and that frees researchers from maintaining infrastructure and links". Their vision is similar to PROMISE but with more general types of data. We focus more on the raw data of open-source projects that can be utilized in any MSR research.

Aiming to improve the scalability and speed of MSR tasks, Hassan *et al.* [81] and Gabel *et al.* [35] 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 describe SCQL [46], 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.

7.2 Data-Parallel Frameworks

Dean and Ghemawat proposed a computing paradigm called MapReduce [28] 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, frameworks, and implementations that directly or indirectly support MapReduce or MapReduce-like paradigms were proposed [9, 25, 49, 68, 75, 94, 95]. *Boa* currently uses MapReduce, via Apache Hadoop [9], as it's execution model. Apache Hadoop is an open-source implementation of MapReduce.

Map-Reduce-Merge [94] adds a new phase to MapReduce called *merge*, which runs after the reduce phase. The merge phase takes the already sorted data from the reducers and performs a merge on it. This new computation model provides the ability to easily express several relational algebra operators as well as perform joins.

Dryad [49] 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 MapReduce. The framework then handles the details of assigning each computation node to servers, handling transfer of data, etc. The framework also directly supports MapReduce programs.

Similar to Dryad, newer versions of Apache Hadoop (MapReduce v2) provide a general computing framework called YARN (Yet Another Resource Negotiator). This framework allows custom application masters that handles scheduling and coordination of the compute tasks. The framework directly supports MapReduce, allows any form of directed, acyclic graph of MapReduces, and is general enough to allow any custom graph of tasks (that don't necessarily have to be directed acyclic graphs). In the future, we plan to migrate *Boa* to the YARN framework. Immediately this should give some performance improvements and in the long term the new framework opens opportunities to investigate further computational models in *Boa*, such as the ability to chain multiple *Boa* programs together.

7.3 Data-Parallel Languages

Sawzall [75] 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 large amount of specially 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 MapReduce framework.

Apache Pig Latin [68] aims to provide both a procedural style MapReduce 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.

SCOPE [24, 97] is a data-parallel processing language from Microsoft. The language is SQL-like and abstracts away any notion of parallelizing the queries. The compiler generates a query plan consisting of a directed, acyclic graph of computations and the execution engine handles automatically scheduling and coordinating tasks on a cluster. Although SCOPE provides support for user defined data, it does not contain any types or direct support for mining software repositories.

A high-level procedural language, DryadLINQ [95], 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.

Hive [87] is a query engine built on top of Hadoop. Hive provides an SQL-like query language called HiveQL. These queries are transformed by a rule-based query optimizer into a MapReduce jobs which then execute on Hadoop. Hive also provides the ability to define custom datatypes, import and export data, and several standard relational operators such as select, filter, and joins. Hive does not provide data types or support for mining software repositories.

FlumeJava [25] is a Java library that provides several abstractions for performing data-parallel computations. These abstractions include a `PCollection`, similar to Java `Collections`, and sets of

functions for grouping, filtering, and aggregating data. These abstractions are at a higher level than the MapReduce programs executed by FlumeJava's framework. FlumeJava's execution engine handles generation of a query plan to run the query in a distributed fashion and provides several optimizations for the queries. FlumeJava also does not aim to ease software mining tasks like *Boa*.

As none of these languages were designed for mining software repositories, they lack the domain-specific types and functions provided by *Boa*.

7.4 MapReduce Optimizations

There are a lot of existing compiler optimizations. Loop fusion is an optimization that takes two or more loops, and fuses them into a single loop. This is done to increase data locality. This technique however is applied only to single programs, and is not typically used to merge loops from unrelated programs together. Note however that any existing program optimization applicable to Java programs can still be applied to individual *Boa* programs.

Chain folding and job merging [63] are two design patterns for optimizing MapReduce pipelines. *Chain folding* takes a single task, represented by a chain of maps and reduces and folds them together, such that the end result is a single map and reduce. The Hadoop API provides ChainMapper [8] and ChainReducer [7] classes for a specific kind of chains of the form: one or more maps, followed by exactly one reduce, followed by zero or more maps. *Job merging* is similar to our task fusion, in that it takes two separate MapReduce jobs and merges them together, by manually combining the map and reduce tasks. The general advice however is to do this merging by hand [82], whereas our approach is fully automatic. Also note that job merging is intended for single users to merge their own jobs together, whereas our approach works for jobs from multiple users in a transparent fashion.

FlumeJava [25] is a Java library that provides several abstractions for performing data-parallel computations. FlumeJava also provides a set of optimizations, including what they call *sibling fusion* and *MSCR fusion*. These two forms of fusion are very similar to our task fusion, in that they attempt to automatically merge nodes in the execution plan together and generate a simpler graph. This merging results in executing fewer MapReduce processes.

Pig latin [68] abstracts the details of MapReduce by providing a data-flow language somewhat

similar to standard SQL. The underlying framework was designed to ease optimizations and makes extensive use of relational database optimizations such as re-ordering joins and pushing selections and projections before joins.

While all of these languages aim to abstract away some or all of the details of distributed programming with MapReduce and many provide optimizations for programs written using them, the optimizations provided are for single programs. Our approach in this paper takes it a step further and provides optimizations for sets of unrelated programs. Note that the optimizations of previous approaches are still applicable in our setting, as they would improve the individual tasks prior to fusing them together.

7.5 Analyzing Source Code

Source code analysis is often performed using a visitor-style pattern [36]. The visitor pattern is intended to allow easily adding additional functionality to a hierarchy of types, without having to modify each type. This is typically accomplished via a double-dispatch where each type to be traversed contains an `accept` method and the new analysis contains `visit` methods. By default visitors perform a depth-first traversal of the tree. There are other forms of the pattern, such as hierarchical visitors [2] which allow controlling the traversal and visitor combinators [90] to compose more complex visitors. There are also reusable visitor pattern libraries [67]. Other approaches make use of visitors, such as Ovlinger and Wand who define a language for recursive traversals [72].

Our language is similar to many of these approaches, however while these approaches are typically for object-oriented languages our host language has no notion of object (only simple record types). Visitors make use of dynamic dispatch in the underlying language, which is not available in procedural languages like Boa. Also, since there is no notion of inheritance, the number of types in the language are fixed, making the analysis in our compiler implementation much simpler and allowed for the optimization mentioned in Section 4.1.

Orleans and Lieberherr provide the language DJ [70], a purely Java-based library implementation of Demeter/Java [69]. In DJ, users provide a traversal strategy and declare visitors with before and after visit methods, similar to our approach. DJ's implementation uses reflection to implement traversals, while our implementation uses a `DefaultVisitor` and has no reflection in the generated code.

Both the work on DJ [70] and recursive traversals [72] provide syntax for specifying traversals separate from the visitor code. Our approach provides a default depth-first traversal and if users need a custom traversal strategy they must specify it intermixed with the visitor code by using `stop` statements and `visit` calls. In the future we may investigate syntax for separating custom traversal strategies from the visitor syntax.

Martin *et al.* describe a program query language (PQL) [61] for easily analyzing source code. They provide a fixed set of events in the language, such as method call or field access and allow queries on those events with static and dynamic matching algorithms. The query language lacks a visitor syntax.

There are also interactive tools for querying source code using natural language queries [53] and custom languages such as JQuery [50]. Since these tools are interactive, they are designed for searching a single codebase and not for mining source code across a large number of projects.

The Sourcerer project [57] provides project metadata source code for over 18k Java projects. Their data is stored in a SQL database, allowing for standard SQL queries on that data. They provide data on single snapshots of projects, including source code information which is represented in the database as entities and relationships. Entities include declarations, type references, and local variables. Relationships include full type resolution and binding of the entities, which our approach does not currently support. The Treasure study [44] built a database containing source code for over 2k projects. They take source code from releases of each project, and map it into their database schema. This schema is capable of representing the entire source code, down to the expression level. Bevan *et al.* proposed a centralized database and data schema called Kenyon [19] for storing mined software repository information. They provide an SQL interface for querying this dataset. All three of these approaches use SQL for mining source code, which gives the benefit of easily performing joins. However source code queries often require recursion (over the graph structure of the data), which is cumbersome to express in SQL [45].

Hajiyev *et al.* describe CodeQuest [45], which uses safe Datalog to query source code information. They map the Datalog queries to standard SQL and query a relational database containing source code information. Unlike SQL, Datalog allows easily specifying recursive-style queries but lacks the visitor pattern familiar to researchers who have worked on or studied compilers and source code analysis previously.

7.6 Language Feature Studies

Grechanik *et al.* [44] performed a large-scale study on Java features on 2k projects from SourceForge. They provided a relational database and studied features such as: classes (abstract, nested, etc), methods (arities, return types, etc), fields, conditional statements, etc. The majority of the features studied are object-oriented language features available since JLS1. They did not study newer language features in JLS3/JLS4. Their study also focused on releases of projects and not the full revision history.

Parnin *et al.* [74] mined the history of 20 open-source Java projects to evaluate how Java generics were integrated and adopted into open source software. As we already showed, our finding on the most popular generic types is consistent with their empirical result. It is also true for the finding that generics are usually adopted by individuals championing for the features, rather than all committers in the team. Hoppe and Hanenberg [48] performed a small empirical study to determine if generic types in Java provide benefit to developers. Basit *et al.* [18] performed an empirical study on two projects regarding how Java generics and C++ templates can help in code refactoring.

Livshits *et al.* [58] focused on the reflection feature in Java. They introduced a static-analysis based reflection resolution algorithm that uses points-to analysis to approximate the targets of reflective calls as part of the call graph. Callaú *et al.* [23] studied the reflection feature in Smalltalk. They reported that such a feature is mostly used in specific kinds of projects: core system libraries, development tools, and tests, rather than in regular applications. Some uses of dynamic features are statically tractable and unproblematic for static analysis and other tools. Christensen *et al.* [27] analyze Java reflection to improve the analysis of string expressions. Dynamic features in Python have been reported to be used more in the initialization than in the computation process [47]. Pankratius *et al.* [73] reported on transactional memory actually helping programmers write concurrent code.

Richards *et al.* [78] performed a large-scale study on the use of `eval` in JavaScript applications. `eval` is used to transform text into executable code, allowing programmers the ability to dynamically extend applications. They studied large-scale execution traces with 550k calls to the `eval` function exercised in over 10k websites. They found that it is often misused and many uses were unnecessary and could be replaced with equivalent and safer code. Earlier, Richards *et al.* [79] analyzed a smaller set of JavaScript programs and concluded the popular usage of `eval` and reported the degree of dynamism

in those programs. Ratanaworabhan *et al.* [76] reported on an existing benchmark for JavaScript and focused on two aspects of JavaScript runtime behavior 1) functions and code and 2) events and handlers. Yue and Wang [96] performed an empirical study on almost 7k websites regarding insecure practices of JavaScript inclusion and dynamic generation. They reported that over 40% of the websites dangerously use `eval`.

Gorschek *et al.* [39] performed a large-scale study on how developers use object-oriented concepts. Tempero [85] studied how fields are used in Java and reported that it is common for developers to declare non-private fields, but then not take advantage of that access. Tempero *et al.* [86] found higher use of inheritance than expected and variation in the use of inheritance between interfaces and classes. Malayeri and Aldrich [59] performed a study on 29 Java programs and reported that nominally typed programs could benefit from structural subtyping. Muschevici *et al.* [64] studied multiple dispatch in several languages and compared its uses.

Meyerovich and Rabkin [62] studied how programming languages are adopted by users, via several large surveys. Their study was focused on which languages were adopted and did not go into detail of specific language features.

The Sourcerer project [57] provides a relational database of mined software artifacts. Their dataset contains over 18k Java projects from SourceForge and Apache. The data is modeled as entities, such as classes, methods, or fields, and relationships among those entities. The dataset contains the source code from the latest snapshot of each project. Baldi *et al.* use the Sourcerer project and topic modeling to empirically validate the theory that aspects are latent topics with a high scattering entropy [17].

While these previous studies have looked at various language features, most are limited to studying a few features, looked at a relatively small number of projects, or did not look at the full history of the software studied. Our study looks at most of Java's new language features, studies over 31k Java projects, and uses each file's full history.

CHAPTER 8. FUTURE WORK

The techniques developed for *Boa* were in the domain of software repository mining, but we believe the idea can be generalized to other domains. Many other domains have large datasets shared by multiple researchers. For example, MERRA [65] is a dataset and framework for processing climate data and models. Similarly, there are shared datasets for genomics, astronomy, geography, etc. In this chapter we outline future directions we plan to investigate to generalize the approach, bringing the benefits of *Boa* to many other domains.

The main research question here is: can we provide a way for domain-experts to describe their domain and from that automatically generate a *Boa*-like framework? Answering this question is a very difficult task, so we break it up into two parts. First, we investigate more generalized methods for handling large amounts of data from any domain. Second, we investigate ways to provide domain experts the ability to specify the form of queries in their domain. Finally, we investigate additional future work in the context of *Boa*.

8.1 Data Description, Transformation, and Storage

In this thesis, we presented the dataset (software repositories), our translation of that data into a custom format using Protocol Buffers, and a storage strategy that ensured the data did not overwhelm the runtime system while still maintaining the ability to incrementally update. The first step to generalizing our approach is figuring out how to handle any arbitrary domain's data.

There has been a lot of work already done on data description languages (DDL). Simply describing the form of the domain's data may not be a difficult task. However, translating that data into a form ammenable to large-scale processing is a non-trivial task. As we showed in Chapter 3, the naive method of representing our data did not work, as the input data for a single project could easily out-grow the

memory available to process it. Thus, an important research question is: can we automatically ensure the data representation strategy will work in the generated infrastructure, and if not can we automatically transform the data representation into a form that will?

8.2 Domain-specific Query Languages

The language we presented in this thesis was specifically designed for the domain of software repository mining. For example, the visitor syntax is a very common approach to performing static analysis, writing compilers, etc which most software engineering researchers are familiar with. Thus it was a good fit for this particular domain. For other domains, forming queries using this language may not be a good fit. For example, if the domain is social networks (which are graphs) it may make sense to provide a query syntax based on the bulk synchronous parallel (BSP) model [89]. Many existing graph-processing frameworks use this model, such as Apache's Hama [11] and Giraph [10] projects and Google's Pregel [60]. An important research question is: how do we allow domain experts to describe the form of queries in their domain?

As a first step, we plan to investigate providing a parameterized compiler-generation framework, where domain experts write their own compiler and we provide hooks to allow for common tasks, such as reading the input data, writing output, the web-based interface (similar to that in Chapter 3), etc. Such a framework would be similar to the Spoofax language workbench [52], where users define their language's grammar and the framework automatically generates various services such as syntax highlighting, code folding, etc. One important research question would be: given the user's grammar definition, how do we automatically infer the proper runtime model, e.g. MapReduce or BSP? As a first step, the domain-expert can indicate the model to use however in the future the proper model could be inferred by analyzing the query language's grammar.

8.3 Future Work on *Boa*

In this section we investigate future directions specific to the *Boa* language and framework.

8.3.1 Language Extensions

Forges such as SourceForge contain a vast wealth of information. Currently *Boa* only contains a subset of that information, such as the project metadata, repository metadata (revisions, logs, etc), and source code. Software mining researchers however are often interested in the other artifacts on such forges, such as bug or issue reports, forum posts, email lists, and additional file formats in the repositories such as design and artifact documentation, logs, etc. While it may be fairly straight-forward to clone this data and provide it in *Boa* as domain-specific types, doing so may not properly abstract the details of mining this data.

One avenue of future work is to examine these additional artifacts and data types and see if the *Boa* language requires additional domain-specific features for mining them. This is similar to how we provide domain-specific visitor features in our language for performing source-code analysis, which is a syntax familiar to people already doing such analyses. For example, we may need to provide domain-specific features for mining bug reports.

Another possible future direction for the language is to modify the existing domain-specific features for source code mining. While the visitor pattern is very familiar to existing developers and flexible enough to perform any traversal, many queries may be interested in matching specific language syntax in the source code being mined. As such, a question that arises is: can we provide a simple way of matching complex patterns of source code, without requiring a visitor pattern and maintaining some state while visiting the source code? A possible related work in this direction are existing tools such as Coccinelle [55], Coverity [34], and Flawfinder [93], which provide pattern matching frameworks where users write queries that look like the source code they wish to match.

8.3.2 Infrastructure Extensions

At the moment, the data analyzed by *Boa* queries is all from SourceForge. There are many additional forges (GitHub, CodePlex, Google Code, etc), some of which (such as GitHub) are even more popular than SourceForge. Supporting these additional forges requires additional support on the back-end for cloning and updating their data and also requires some changes to the datatypes. Each forge has a different set of metadata. For example, SourceForge provides a lot of information, such as topics,

environment, etc which is not available on other forges like GitHub. We would like to provide as uniform an interface as possible to the metadata provided by these forges. As such, one interesting research question in this area is: would it be possible to use data mining and automatically infer some of the missing information? For example, perhaps we can use topic modeling to automatically infer the topics of GitHub projects.

Supporting GitHub would also require that we support the Git repository format, whereas currently we only support Subversion and CVS (by converting to SVN format). We need to update our domain-specific types to support the Git format, which is a distributed version control system (VCS). One research problem that remains is determining a proper data representation that can unify the representation for such distributed VCSs as well as the non-distributed form, such as SVN and CVS which we already handle. Once a flexible model is developed we can utilize existing tools to handle mirroring repositories in the various formats format, as some already exist and are open-sourced [66].

Additional effort is needed to support more source code languages (other than Java). We designed our custom AST to be general enough to support most object-oriented features and easily extensible. As such, adding additional OO languages (such as C# or C++) should be relatively straight-forward. At this time we have not attempted to support non-OO languages (such as Scheme/Lisp). Support for other programming paradigms may require modifications to our types. This work will focus on answering the question: is there a (relatively small?) set of language features that are capable of forming the basis of all (or the majority of) programming languages, even from different paradigms?

8.3.3 Improving Adoption and Usability of *Boa*

One important question that still remains is how easy is the *Boa* language to learn and use? We have only recently opened the infrastructure to other users and early feedback seems to indicate a broad range of answers from very simple to "incredibly difficult." Our comprehension study showed the code is easier to comprehend when compared to Hadoop programs solving the same tasks, but in the future we would like a proper usability study to see how writing queries in the language compares to other approaches, such as Hadoop or SQL queries.

One possible avenue to help ease the adoption of *Boa* is to provide an API interface to the infrastructure. Such an API would allow queries to be embedded into other languages. This API can operate

similar to how SQL queries are used in languages such as Java, although asynchronously due to the time required to execute queries and the (potential) size of the resulting data. Developing this API may pose several interesting research questions, such as when and how do we cache data, what sort of consistency guarantees do we provide, etc.

CHAPTER 9. 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 have 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 write parallelizing code. *Boa* is already expressive enough to allow us to reproduce a previous study on the Java language [44] as well as conduct our own large-scale case study on the adoption of Java language features.

Analysis tasks written in *Boa* are easy to comprehend, when compared to similar tasks written in Hadoop. We also demonstrated that experiments conducted using *Boa* are easily reproduced simply by re-running *Boa* programs provided by the previous researchers.

APPENDIX A. GRAMMAR OF THE BOA LANGUAGE

This section contains the full grammar for the *Boa* language. The grammar uses the following BNF-style conventions:

- *Symbol* is a non-terminal symbol
- "**symbol**" is a terminal symbol
- [x] denotes zero or one occurrences of x
- (x)* denotes zero or more occurrences of x
- $(x)^+$ denotes one or more occurrences of x
- x | y means one of either x or y

A *Boa* `Program` is a sequence of one or more `Declaration`s and `Statement`s.

Program ::= (*Declaration* | *Statement*)$^+$

Declarations

`Declaration`s introduce names and include variable declarations (`VarDeclaration`), static variable declarations (`StaticVarDeclaration`), and `TypeDeclaration`s.

Declaration ::= *VarDeclaration* | *StaticVarDeclaration* | *TypeDeclaration*

VarDeclaration ::= *identifier* "**:**" [*Type*] ["**=**" *Expression*] "**;**"

StaticVarDeclaration ::= "**static**" *VarDeclaration*

TypeDeclaration ::= "**type**" *identifier* "**=**" *Type* "**;**"

A variable declaration declares a new variable with a name given by an identifier. By default, the variable will have an undefined value. If the variable declaration contains an initializer expression then the variable will be assigned the expression, possibly with an automatic cast (assuming the type is also specified). If there is no type specified, it is inferred from the type of the initializer expression. In this case, the colon and equals are often placed together without a space (:=).

Variables can also be declared as static. Static variables are evaluated once per program (not once per input value). All output variables are implicitly static (and the static keyword may be omitted).

A `TypeDeclaration` binds a name given by an identifier to a `Type`.

Types

Boa is a statically-typed language. `Types` can be an identifier referring to a named type previously declared in scope.

Type ::= *identifier* | *ArrayType* | *MapType* | *StackType* | *VisitorType* | *OutputType* | *FunctionType*

Types can also be an `ArrayType`, `MapType`, or `StackType`. Array types expect the type of values stored in the array. Values are stored into a array using `Assignment` statements where the left-hand side is a `Factor` with an `Index`.

Map types expect the type of keys and values stored in the map. Values are stored into a map using `Assignment` statements where the left-hand side is a `Factor` with an `Index`. A function call $haskey(m, k)$ is provided to test if a map m contains a key k. Maps can be cleared using the $clear(m)$ function call.

Stack types expect the type of values stored on the stack. A value v may be pushed onto the stack s by calling $push(s, v)$ and a value may be popped off the stack by calling $pop(s)$. Stacks can be cleared using the $clear(s)$ function call.

ArrayType ::= "**array**" "**of**" *Type*

MapType ::= "**map**" "**[**" *Type* "**]**" "**of**" *Type*

StackType ::= "**stack**" "**of**" *Type*

Types may also be a `VisitorType`. All visitors have this type.

VisitorType ::= "**visitor**"

Output is generated using variables with `OutputType`. Output types have an identifier naming an aggregator function. Depending on the aggregator function, it may also take an expression list or parameters. Output types have a sequence of zero or more index types. They also specify the type of the expected values emitted to the output variable. Depending on the aggregator, the output type also takes an optional weight type and an optional format expression list.

ExprList ::= *Expression* ("**,**" *Expression*)*

OutputType ::= "**output**" *identifier* ["**(**" *ExprList* "**)**"] ("**[**" *Type* "**]**")* "**of**" *Type*
["**weight**" *Type*] ["**format**" "**(**" *ExprList* "**)**"]

`FunctionTypes` specify a signature for a function, which includes the ordered list of any `parameters` and an optional return type.

parameter ::= *identifier* "**:**" *Type*

FunctionType ::= "**function**" "**(**" [*parameter* ("**,**" *parameter*)*] "**)**" ["**:**" *Type*]

Statements

The language provides several different `Statements`, including an `EmptyStatement` (which is just a semi-colon and does nothing).

Statement ::= *Assignment* | *Block* | *BreakStatement* | *ContinueStatement* | *DoStatement*
| *EmitStatement* | *EmptyStatement* | *ExprStatement* | *ForStatement* | *IfStatement*
| *QuantifierStatement* | *ResultStatement* | *ReturnStatement* | *SwitchStatement*
| *WhileStatement*

EmptyStatement ::= "**;**"

An `Assignment` statement takes a `Factor` on the left-hand side representing the location to store an `Expression` (on the right-hand side).

Assignment ::= *Factor* "**=**" *Expression* "**;**"

An expression statement takes an `Expression` and either increments or decrements its value.

ExprStatement ::= *Expression* [“**++**” | “**--**”] “**;**”

The language also provides `Block`s containing a sequence of zero or more `Declaration`s and `Statement`s. Blocks allow scoping declarations.

Block ::= “**{**” (*Declaration* | *Statement*)* “**}**”

An `IfStatement` is provided that takes an `Expression` and if the value of that expression is true runs a statement, otherwise runs an optional else statement. The `SwitchStatement` is similar to other languages, with the exception that several cases may be provided at once and a default case is always required.

IfStatement ::= “**if**” “**(**” *Expression* “**)**” *Statement* [“**else**” *Statement*]

SwitchStatement ::= “**switch**” “**(**” *Expression* “**)**” “**{**”
(“**case**” *Expression* (“**,**” *Expression*)* “**:**” *Statement* (*Statement*)*)*
“**default**” “**:**” *Statement* (*Statement*)* “**}**”

Break and continue statements are provided for exiting loops and control structures. These behave similarly to other languages, such as Java. The `ReturnStatement` allows returning expressions from inside functions.

BreakStatement ::= “**break**” “**;**”

ContinueStatement ::= “**continue**” “**;**”

ReturnStatement ::= “**return**” [*Expression*] “**;**”

The language provides several common looping statements, such as do, for, and while. Note that the for-loops require local rules for `ExprStatement` and `VarDecl` so that they can be omitted (by providing only a semi-colon).

DoStatement ::= “**do**” *Statement* “**while**” “**(**” *Expression* “**)**” “**;**”

ForExprStatement ::= *Expression* [“**++**” | “**--**”]

ForVarDecl ::= *identifier* “**:**” [*Type*] [“**=**” *Expression* | *Block*]

ForStatement ::= “**for**” “**(**” [*ForVarDecl* | *ForExprStatement*] “**;**” [*Expression*] “**;**”
[*ForVarDecl* | *ForExprStatement*] “**)**” *Statement*

WhileStatement ::= “**while**” “**(**” *Expression* “**)**” *Statement*

The language provides a new looping construct called a `QuantifierStatement`. There are three forms of quantifiers: foreach, ifall, and exists. All expect a parameter, which is a variable declaration without an initializer. The variable is declared in the scope of the statement.

QuantifierStatement ::= (“**foreach**” | “**ifall**” | “**exists**”) “**(**” *parameter* “**;**” *Expression* “**)**” *Statement*

An `EmitStatement` allows emitting an `Expression` to an identifier naming an output variable. If the output variable has indices, the statement provides expressions for them. Similarly, if the output variable's aggregator requires a weight, the emit statement provides it.

EmitStatement ::= *identifier* (“**[**” *Expression* “**]**”)* “**<<**” *Expression* [“**weight**” *Expression*] “**;**”

Expressions

Expressions apply operations on `Factor`s to compute values. `Factor`s are `Operand`s followed by a sequence of zero or more `Selector`s, `Index`es, or `Call`s.

Operators	Associativity
(unary) + - ~ ! not	right to left
* / % << >> &	left to right
+ - | ^	left to right
== != < <= > >=	left to right
&& and	left to right
|| or	left to right

Expression ::= *Conjunction* ((“||” | “**or**”) *Conjunction*)*

Conjunction ::= *Comparison* ((“**&&**” | “**and**”) *Comparison*)*

Comparison ::= *SimpleExpr* [("**==**" | "**!=**" | "**<**" | "**<=**" | "**>**" | "**>=**") *SimpleExpr*]

SimpleExpr ::= *Term* (("**+**" | "**-**" | "|" | "**^**") *Term*)*

Term ::= *Factor* (("*****" | "/" | "**%**" | "**<<**" | "**>>**" | "**&**") *Factor*)*

Factor ::= *Operand* (*Selector* | *Index* | *Call*)*

Selector ::= "**.**" *identifier*

Index ::= "**[**" *Expression* ["**:**" *Expression*] "**]**"

Call ::= "**(**" [*ExprList*] "**)**"

Operand ::= *identifier* | *StringLiteral* | *IntegerLiteral* | *FloatingPointLiteral* | *CharacterLiteral*
| *TimeLiteral* | *Composite* | *VisitorExpr* | *Function* | ("**+**" | "**-**" | "**~**" | "**!**" | "**not**") *Factor*
| "**(**" *Expression* "**)**"

Pair ::= *Expression* "**:**" *Expression*

PairList ::= *Pair* ("**,**" *Pair*)*

Composite ::= "**{**" [*PairList* | *ExprList* | "**:**"] "**}**"

Function ::= *FunctionType Block*

Visitors

Visitor bodies are declared as `VisitorExpr`, which contains a visitor type and a body with a sequence of zero or more `VisitClause`s.

VisitorExpr ::= *VisitorType* "**{**" (*VisitClause*)* "**}**"

A `VisitClause` is either a *before* visit or an *after* visit. Before visits run when a type matched by the `IdentifierList` is visited, before its children are visited. After visits run for matching types after visiting its children. Since before visits have the ability to control traversal, statements in before visits include two additional statements.

VisitClause ::= "**before**" *IdentifierList* "**->**" *BeforeClauseStatement*
| "**after**" *IdentifierList* "**->**" *Statement*

Before visit bodies can be a normal statement, a stop statement which stops the standard traversal strategy at that point in the tree, or a visit statement to manually visit a child node.

BeforeClauseStatement ::= *Statement*

| "**stop**" "**;**"

| "**visit**" "**(**" *identifier* "**)**" "**;**"

An `IdentifierList` can be a wildcard, in which case it provides default behavior for the visitor and runs if no other clause matches the type being visited. It can also be a specific type with a named identifier or a list of types.

IdentifierList ::= "**_**"

| *identifier* "**:**" *identifier*

| *identifier* ("**,**" *identifier*)*

Literals

Literals are composed of `digits` and `letters`.

digit ::= "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**"

letter ::= "**a**" | "**b**" | "**c**" | "**d**" | "**e**" | "**f**" | "**g**" | "**h**" | "**i**" | "**j**" | "**k**" | "**l**" | "**m**" | "**n**" | "**o**" | "**p**"

| "**q**" | "**r**" | "**s**" | "**t**" | "**u**" | "**v**" | "**w**" | "**x**" | "**y**" | "**z**"

| "**A**" | "**B**" | "**C**" | "**D**" | "**E**" | "**F**" | "**G**" | "**H**" | "**I**" | "**J**" | "**K**" | "**L**" | "**M**" | "**N**" | "**O**" | "**P**"

| "**Q**" | "**R**" | "**S**" | "**T**" | "**U**" | "**V**" | "**W**" | "**X**" | "**Y**" | "**Z**"

An `identifier` must start with a letter and may be followed by a sequence of zero or more letters, digits, and underscores.

identifier ::= *letter* (*letter* | *digit* | "**_**")*

`IntegerLiterals` can be specified with either binary, decimal, hex, or octal syntax any may be negative. `BinaryLiterals` start with a digit 0 and a lowercase or uppercase letter B and are followed by a sequence of one or more digits 0 or 1. `DecimalLiterals` start with any digit other than 0 and

are followed by zero or more digits. `HexLiterals` start with a digit 0 and an uppercase or lowercase letter X and are followed by one or more digits or letters A to F. `OctalLiterals` start with the digit 0 and are followed by zero or more digits 0 to 7.

IntegerLiteral ::= [“**-**”] (*BinaryLiteral* | *DecimalLiteral* | *HexLiteral*) | *OctalLiteral*

BinaryLiteral ::= “**0**” (“**b**” | “**B**”) (“**0**” | “**1**”)$^+$

DecimalLiteral ::= (“**1**” | “**2**” | “**3**” | “**4**” | “**5**” | “**6**” | “**7**” | “**8**” | “**9**”) (*digit*)*

HexLiteral ::= “**0**” (“**x**” | “**X**”) (*digit* | “**a**” | “**b**” | “**c**” | “**d**” | “**e**” | “**f**” | “**A**” | “**B**” | “**C**” | “**D**” | “**E**” | “**F**”)$^+$

OctalLiteral ::= “**0**” (“**0**” | “**1**” | “**2**” | “**3**” | “**4**” | “**5**” | “**6**” | “**7**”)*

A `FloatingPointLiteral` can be negative. It has decimals, a period, more decimals, and an optional scale. Either the first or second group of digits can be missing (but not both). Additionally, either the period or the scale may be missing (but not both).

scale ::= (“**e**” | “**E**”) [“**+**” | “**-**”] (*digit*)$^+$

FloatingPointLiteral ::= [“**-**”] “**.**” (*digit*)$^+$ [*scale*]

| [“**-**”] (*digit*)$^+$ *scale*

| [“**-**”] (*digit*)$^+$ “**.**” (*digit*)* [*scale*]

A `CharacterLiteral` is a single quote followed by one `character` followed by a single quote.

character ::= ~(“**n**” | “**r**” | “**f**” | “**t**” | “**b**” | “\” | “””)

| “\” (“**n**” | “**r**” | “**f**” | “**t**” | “**b**” | “\” | “””)

CharacterLiteral ::= “’” *character* “’”

A `StringLiteral` is a double quote followed by a sequence of zero or more `characters` followed by a double quote. A second form of string literals uses the backtick instead of double quotes. This form is designed to make writing regular expressions easier, by not having the notion of an escape character (meaning the backslash does not need to be doubled).

StringLiteral ::= “”” (*character*)* “”””

| “`” ~(“\**n**” | “\**r**” | “`”) “`”

A `TimeLiteral` defines a notion of date and time. It can be an integer, followed by a lowercase or uppercase letter T. In this case the literal is interpreted to be the number of microseconds since Dec 31 16:00:00 PST 1969. A time literal may also be an uppercase letter T followed a string literal, where the string gives the date and time.

TimeLiteral ::= *IntegerLiteral* (“**t**” | “**T**”)
| “**T**” *StringLiteral*

APPENDIX B. DOMAIN-SPECIFIC TYPES IN BOA

In this chapter we list all of the domain-specific types provided by *Boa*. We also list all attributes contained in the types.

Person A unique person's information	
Attribute : Type	**Description**
email : string	The person's email address, if known
real_name : string	The person's real name, if known, otherwise the same as username
username : string	The person's username

Project Metadata Types

Project Top-level type, represents a single project on the forge	
Attribute : Type	**Description**
audiences : array of string	A list of the target audiences for the project
code_repositories : array of CodeRepository	A list of all code repositories associated with this project
created_date : time	The time the project was created
databases : array of string	A list of all databases used by the project
description : string	A description of the project
developers : Person	A list of all software developers currently on the project
donations : bool	If true, this project explicitly states it accepts donations
homepage_url : string	A URL to the project's homepage
id : string	Unique identifier for the project
interfaces : array of string	A list of all interfaces supported by the project
licenses : array of string	A list of all licenses used by the project
maintainers : Person	A list of all people currently maintaining the project
name : string	The name of the project
operating_systems : array of string	A list of all OSes supported by the project
programming_languages : array of string	A list of all programming languages used by the project
project_url : string	A URL to the project's page on the forge
topics : array of string	A list of self-categorized topics the project belongs to

Source Control Metadata Types

CodeRepository A source code repository (SVN, CVS, Git, etc)	
Attribute : Type	**Description**
kind : RepositoryKind	The kind of code repository (SVN, GIT, etc)
revisions : array of Revision	All of the revisions contained in the code repository
url : string	The URL to access the code repository

enum **RepositoryKind** Describes the kind of code repository	
Attribute	**Description**
BZR	For Bazaar code repositories
CVS	For CVS code repositories
GIT	For Git code repositories
HG	For Mercurial code repositories
SVN	For Subversion code repositories
UNKNOWN	The code repository's kind was not known

Revision A single revision inside a CodeRepository	
Attribute : Type	**Description**
author : Person	The person who authored the revision, if known, otherwise the same as committer
commit_date : time	The time the revision was committed
committer : Person	The person who committed the revision
files : array of ChangedFile	A list of all files committed in the revision
id : int	A unique identifier for the revision
log : string	The log message attached to the revision

ChangedFile A file committed in a Revision	
Attribute : Type	**Description**
change : ChangeKind	The kind of change for this file
kind : FileKind	The kind of file
name : string	The full name and path of the file

enum **ChangeKind** Describes the kind of change for the file	
Attribute	**Description**
ADDED	The file did not already exist and was added
DELETED	The file was deleted
MODIFIED	The file already existed and was modified

enum **FileKind** Describes the kind of the file	
Attribute	**Description**
BINARY	The file represents a binary file
SOURCE_JAVA_ERROR	The file represents a Java source file that had a parse error
SOURCE_JAVA_JLS2	The file represents a Java source file that parsed without error as JLS2
SOURCE_JAVA_JLS3	The file represents a Java source file that parsed without error as JLS3
SOURCE_JAVA_JLS4	The file represents a Java source file that parsed without error as JLS4
TEXT	The file represents a text file
UNKNOWN	The file's type was unknown
XML	The file represents an XML file

Source Code AST Types

ASTRoot Container class that holds a file's parsed AST	
Attribute : Type	**Description**
imports : array of string	The imported namespaces and types
namespaces : array of Namespace	The top-level namespaces in the file

Namespace A namespace (aka, package) in a source file	
Attribute : Type	**Description**
declarations : array of Declaration	Declarations contained in this namespace
modifiers : array of Modifier	Any modifiers/annotations on the namespace
name : string	The name of the namespace

Declaration A type declaration, such as a class or interface	
Attribute : Type	**Description**
fields : array of Variable	The fields in the declaration
generic_parameters : array of Type	Any generic parameters to this declaration
kind : TypeKind	The kind of this declaration
methods : array of Method	The methods in the declaration
modifiers : array of Modifier	The modifiers/annotations on this declaration
name : string	The name of this declaration
nested_declarations : array of Declaration	Any nested declarations
parents : array of Type	The explicitly named parent type(s) of this declaration

Type A type in an AST	
Attribute : Type	**Description**
kind : TypeKind	The kind of the type
name : string	The name of the type

enum **TypeKind** The kinds of types in an AST	
Attribute	**Description**
ANNOTATION	An annotation type
ANONYMOUS	An anonymous type
CLASS	A class type
DELEGATE	A delegate type
ENUM	An enumerated type
GENERIC	A generic type
INTERFACE	An interface type
OTHER	Any other kind of Type
STRUCT	A struct

Method A method declaration	
Attribute : Type	**Description**
arguments : array of Variable	The arguments the method takes
exception_types : array of Type	The list of exceptions thrown by this method
generic_parameters : array of Type	The list of generic parameters for this method
modifiers : array of Modifier	A list of all modifiers on the variable
name : string	The name of the method
return_type : Type	The type returned from the method; if the method returns nothing, this type will be void
statements : array of Statement	The statements in the method body

Variable A variable declaration - can be a field, local, parameter, etc	
Attribute : Type	**Description**
initializer : Expression?	If the variable has an initial assignment, the expression is stored here
modifiers : array of Modifier	A list of all modifiers on the variable
name : string	The name of the variable
variable_type : Type	The type of the variable

Statement A single statement	
Attribute : Type	**Description**
expression : Expression?	A sub-expression
initializations : array of Expression	Used in for loops
kind : StatementKind	The kind of statement
statements : array of Statement	A list of sub-statements
type_declaration : Declaration?	A type declaration
updates : array of Expression	Used in for loops
variable_declaration : Variable?	A variable declaration

enum **StatementKind** The kind of statement	
Attribute	**Defined Sub-Attributes**
ASSERT	expression
BLOCK	statements
BREAK	expression (if to a label)
CASE	expression
CATCH	variable_declaration, statements
CONTINUE	expression (if to a label)
DO	expression, statements
EMPTY	
EXPRESSION	expression
FOR	initializations, updates, expression, statements, variable_declaration (for enhanced-for loops)
IF	expression, statements
LABEL	expression, statements
OTHER	
RETURN	expression
SWITCH	expression, statements
SYNCHRONIZED	expression, statements
THROW	expression
TRY	statements, initializations (for try with resources)
TYPEDECL	type_declaration
WHILE	expression, statements

Expression A single expression	
Attribute : Type	**Description**
annotation : Modifier?	An annotation
anon_declaration : Declaration?	An anonymous declaration
expressions : array of Expression	A list of sub-expressions
generic_parameters : array of Type	A list of generic type parameters
is_postfix : bool?	If an operator is postfix
kind : ExpressionKind	The kind of expression
literal : string?	A literal value
method : string?	A method name
method_args : array of Expression	A list of method arguments
new_type : Type?	A type
variable : string?	A variable name
variable_decls : array of Variable	A list of variable declarations

enum **ExpressionKind** The kind of expression	
Attribute	**Defined Sub-Attributes**
ANNOTATION	annotation
ARRAYINDEX	expressions
ARRAYINIT	expressions
ASSIGN	expressions
ASSIGN_ADD	expressions
ASSIGN_BITAND	expressions
ASSIGN_BITOR	expressions
ASSIGN_BITXOR	expressions
ASSIGN_DIV	expressions
ASSIGN_LSHIFT	expressions
ASSIGN_MOD	expressions
ASSIGN_MULT	expressions
ASSIGN_RSHIFT	expressions
ASSIGN_SUB	expressions
ASSIGN_UNSIGNEDRSHIFT	expressions
BIT_AND	expressions
BIT_LSHIFT	expressions
BIT_NOT	expressions
BIT_OR	expressions
BIT_RSHIFT	expressions
BIT_UNSIGNEDRSHIFT	expressions
BIT_XOR	expressions
CAST	expressions, new_type
CONDITIONAL	expressions
EQ	expressions

enum **ExpressionKind** The kind of expression	
Attribute	**Defined Sub-Attributes**
GT	expressions
GTEQ	expressions
LITERAL	literal
LOGICAL_AND	expressions
LOGICAL_NOT	expressions
LOGICAL_OR	expressions
LT	expressions
LTEQ	expressions
METHODCALL	method, expressions, generic_parameters
NEQ	expressions
NEW	expressions, new_type, generic_parameters, anon_declaration
NEWARRAY	expressions, new_type
NULLCOALESCE	expressions
OP_ADD	expressions
OP_DEC	expressions, is_postfix
OP_DIV	expressions
OP_INC	expressions, is_postfix
OP_MOD	expressions
OP_MULT	expressions
OP_SUB	expressions
OTHER	
TYPECOMPARE	expressions, new_type
VARACCESS	expressions, variable
VARDECL	variable_decl

Modifier A single modifier	
Attribute : Type	**Description**
annotation_members : array of string	If the kind is ANNOTATION, then a list of all members explicitly assigned values
annotation_name : string?	If the kind is ANNOTATION, then the name of the annotation
annotation_values : array of Expression	If the kind is ANNOTATION, then a list of all values that were assigned to members
kind : ModifierKind	The kind of modifier
other : string?	If the modifier kind is OTHER, the modifier string from the source code
visibility : Visibility	A kind of visibility modifier

enum **ModifierKind** The kind of modifier	
Attribute	**Description**
ABSTRACT	An abstract modifier
ANNOTATION	An annotation modifier
FINAL	A final modifier
OTHER	Any other modifier - the value is in the Modifier's other attribute
STATIC	A static modifier
SYNCHRONIZED	A synchronized modifier
VISIBILITY	A Visibility modifier - the value is in the Modifier's visibility attribute

enum **Visibility** A visibility modifier	
Attribute	**Description**
NAMESPACE	A namespace (aka, default, aka package) visibility modifier
PRIVATE	A private modifier
PROTECTED	A protected modifier
PUBLIC	A public modifier

APPENDIX C. DOMAIN-SPECIFIC FUNCTIONS IN BOA

In this chapter, we describe some of the domain-specific functions provided by the *Boa* runtime. Where possible, we also show the *Boa* code implementing the function.

```
getast := function(name: ChangedFile) : ASTRoot;
```

Returns the `ASTRoot` of the specified `file`, if it exists. Otherwise returns `undef`.

```
getsnapshot := function(cr: CodeRepository, t: time, filters: array of string) : array of
  ChangedFile {
  snapshot: map[string] of ChangedFile;

  visit(cr, visitor {
    before node: Revision ->
      if (node.commit_date > t)
        stop;
    before node: ChangedFile -> {
      filter := len(filters) > 0;

      exists (i: int; iskind(filters[i], node.kind))
        filter = false;

      if (!filter) {
        if (node.change == ChangeKind.DELETED)
          remove(snapshot, node.name);
        else
          snapshot[node.name] = node;
      }
    }
  });

  return values(snapshot);
};
```

Returns a snapshot of `ChangedFiles`. A snapshot is the last version of a file before a given time `t` (if no time is given, `NOW` is used). If any `filters` are given, they are used to filter out files. The file kind is checked against each string and must one or more filters. Matches are performed by comparing the filter against the start of the file kind.

```
isliteral := function(e: Expression, s: string) : bool {
  return e.kind == ExpressionKind.LITERAL && def(e.literal) && e.literal == s;
};
```

Returns `true` if the expression `e` is of kind LITERAL and the literal matches the string `s`.

```
hasfiletype := function(data: Revision, ext: string) : bool {
  exists (i: int; match(format(`\.%s$`, lowercase(ext)), lowercase(data.files[i].name)))
    return true;
  return false;
};
```

Does the `data` contain a file of the specified type? This compares based on the given file `ext`ension. The comparison is case insensitive. Valid `dsl_types` are: `Project`, `CodeRepository`, and `Revision`.

```
isfixingrevision := function(log: string) : bool {
  if (match(`\bfix(s|es|ing|ed)?\b`, log))    return true;
  if (match(`\b(error|bug|issue)(s)\b`, log)) return true;
  return false;
};
```

Is the given `log` message indicating it is a fixing revision? A message is considered indicating a bug fix if it matches a set of regular expressions.

```
iskind := function(s: string, k: dsl_type) : bool {
  return match(format("^%s", s), string(k));
};
```

Returns `true` if the kind `k` starts with the string `s`. Valid `dsl_types` are: `FileKind`.

APPENDIX D. ADDITIONAL EXAMPLE BOA PROGRAMS

In this chapter, we show example *Boa* queries from several different categories.

Programming Languages

```
counts: output top(10) of string weight int;
p: Project = input;
foreach (i: int; def(p.programming_languages[i]))
    counts << p.programming_languages[i] weight 1;
```

Figure D.1: A1. What are the ten most used programming languages?

```
counts: output sum of int;
p: Project = input;
if (len(p.programming_languages) > 1)
    counts << 1;
```

Figure D.2: A2. How many projects use more than one programming language?

```
counts: output sum of int;
p: Project = input;
foreach (i: int; match(`^scheme$`, lowercase(p.programming_languages[i])))
    counts << 1;
```

Figure D.3: How many projects use the Scheme programming language?

Project Management

```
counts: output sum[int] of int;
p: Project = input;
counts[yearof(p.created_date)] << 1;
```

Figure D.4: B1. How many projects are created each year?

```
values: output sum[string] of int;
p: Project = input;
foreach (i: int; def(p.topics[i]))
    values[lowercase(p.topics[i])] << 1;
```

Figure D.5: B2. How many projects self-classify into each topic provided by SourceForge?

```
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].kind == RepositoryKind.SVN)
        exists (k: int; yearof(p.code_repositories[j].revisions[k].commit_date) == 2011)
            counts << 1;
```

Figure D.6: B3. How many Java projects using SVN were active in 2011?

```
counts: output top(1) of int weight int;
p: Project = input;
exists (j: int; match('^java$', lowercase(p.programming_languages[j])))
    foreach (i: int; p.code_repositories[i].kind == RepositoryKind.SVN)
        if (len(p.code_repositories[i].revisions) > 0)
            counts << yearof(p.code_repositories[i].revisions[0].commit_date) weight 1;
```

Figure D.7: B4. In which year was SVN added to Java projects the most?

```
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].kind == RepositoryKind.SVN)
        counts << len(p.code_repositories[j].revisions);
```

Figure D.8: B5. How many revisions are there in all Java projects using SVN?

```
counts: output sum[string] of int;
p: Project = input;
committers: map[string] of bool;
foreach (i: int; def(p.code_repositories[i]))
    foreach (j: int; def(p.code_repositories[i].revisions[j]))
        committers[p.code_repositories[i].revisions[j].committer.username] = true;
if (len(committers) > 0)
    counts[p.id] << len(committers);
```

Figure D.9: B7. How many committers are there for each project?

```
p: Project = input;
counts: output mean[string] of int;
foreach (i: int; def(p.code_repositories[i]))
    foreach (j: int; def(p.code_repositories[i].revisions[j]))
        counts[p.id] << len(p.code_repositories[i].revisions[j].files);
```

Figure D.10: B9. What are the churn rates for all projects?

```
counts: output sum[int] of int;
p: Project = input;
exists (i: int; match('^java$', lowercase(p.programming_languages[i])))
    foreach (j: int; p.code_repositories[j].kind == RepositoryKind.SVN)
        foreach (k: int; def(p.code_repositories[j].revisions[k]))
            counts[yearof(p.code_repositories[j].revisions[k].commit_date)] << 1;
```

Figure D.11: B10. How did the number of commits for Java projects using SVN change over years?

Legal

```
counts: output sum of int;
p: Project = input;
if (len(p.licenses) > 1)
    counts << 1;
```

Figure D.12: C2. How many projects use more than one license?

Platform/Environment

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

Figure D.13: D1. What are the five most supported operating systems?

```
counts: output collection[string] of string;
p: Project = input;
if (len(p.operating_systems) > 1)
    counts[p.id] << p.project_url;
```

Figure D.14: D2. Which projects support multiple operating systems?

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

Figure D.15: D3. What are the five most popular databases?

```
counts: output collection[string] of string;
p: Project = input;
if (len(p.databases) > 1)
    counts[p.id] << p.name;
```

Figure D.16: D4. What are the projects that support multiple databases?

Source Code

```
p: Project = input;
top5: output top(5) of string weight int;
astCount := 0;
visit(p, visitor {
    # only look at the latest snapshot
    before n: CodeRepository -> {
        snapshot := getsnapshot(n);
        foreach (i: int; def(snapshot[i]))
            visit(snapshot[i]);
        stop;
    }
    # by default, count all visited nodes
    before _ -> astCount++;
    # these nodes are not part of the AST, so do nothing when visiting
    before Project, ChangedFile -> ;
});
# Output is in Millions of AST nodes.
top5 << p.project_url weight astCount / 1000000;
```

Figure D.17: E1. What are the five largest projects, in terms of AST nodes?

```
counts: output sum of int;
p: Project = input;
visit(p, visitor {
    before node: CodeRepository ->
        counts << len(getsnapshot(node, "SOURCE_JAVA_JLS"));
});
```

Figure D.18: E2. How many valid Java files in latest snapshot?

```
AddedNullCheck: output sum of int;
p: Project = input;
isfixing := false;
count := 0;
# map of file names to the last revision of that file
files: map[string] of ChangedFile;
visit(p, visitor {
    before node: Revision -> isfixing = isfixingrevision(node.log);
    before node: ChangedFile -> {
        # if this is a fixing revision and
        # there was a previous version of the file
        if (isfixing && haskey(files, node.name)) {
            # count how many null checks were previously in the file
            count = 0;
            visit(getast(files[node.name]));
            last := count;
            # count how many null checks are currently in the file
            count = 0;
            visit(getast(node));
            # if there are more null checks, output
            if (count > last)
                AddedNullCheck << 1;
        }
        if (node.change == ChangeKind.DELETED)
            remove(files, node.name);
        else
            files[node.name] = node;
        stop;
    }
    before node: Statement ->
        # increase the counter if there is an IF statement
        # where the boolean condition is of the form:
        #   null == expr OR expr == null OR null != expr OR expr != null
        if (node.kind == StatementKind.IF)
            visit(node.expression, visitor {
                before node: Expression ->
                    if (node.kind == ExpressionKind.EQ || node.kind == ExpressionKind.NEQ)
                        exists (i: int; isliteral(node.expressions[i], "null"))
                            count++;
            });
});
```

Figure D.19: E3. How many fixing revisions added null checks?

```
p: Project = input;
GenericFields: output sum[string] of int;
visit(p, visitor {
    before node: Type ->
        if (strfind("<", node.name) > -1)
            GenericFields[p.project_url] << 1;
    before node: Declaration -> {
        # check all fields
        foreach (i: int; node.fields[i])
            visit(node.fields[i]);
        # also look at nested declarations
        foreach (i: int; node.methods[i])
            visit(node.methods[i]);
        foreach (i: int; node.nested_declarations[i])
            visit(node.nested_declarations[i]);
        stop;
    }
    before node: Method -> {
        foreach (i: int; node.statements[i])
            visit(node.statements[i]);
        stop;
    }
    before node: Statement -> {
        foreach (i: int; node.statements[i])
            visit(node.statements[i]);
        if (def(node.type_declaration))
            visit(node.type_declaration);
        stop;
    }
    # fields cant be below expressions or modifiers
    before Expression, Modifier -> stop;
});
```

Figure D.20: E4. How many generic fields are declared in each project?

```
p: Project = input;
Varargs: output collection[string][string][time] of int;
file_name: string;
commit_date: time;
visit(p, visitor {
    before node: ChangedFile -> file_name = node.name;
    before node: Revision -> commit_date = node.commit_date;
    before node: Method ->
        if (len(node.arguments) > 0
                && strfind("...", node.arguments[len(node.arguments) - 1].variable_type.
  name) > -1)
            Varargs[p.project_url][file_name][commit_date] << 1;
});
```

Figure D.21: E5. How is varargs used over time?

```
p: Project = input;
TransientTotal: output sum of int;
TransientMax: output maximum(1) of string weight int;
TransientMin: output minimum(1) of string weight int;
TransientMean: output mean of int;
count := 0;
s: stack of int;
visit(p, visitor {
    before node: CodeRepository -> {
        # only look at the latest snapshot
        # and only include Java files
        snapshot := getsnapshot(node, "SOURCE_JAVA_JLS");
        foreach (i: int; def(snapshot[i]))
            visit(snapshot[i]);
        stop;
    }
    before node: Declaration -> {
        # only interested in fields, which only occur inside (anonymous) classes
        if (node.kind == TypeKind.CLASS || node.kind == TypeKind.ANONYMOUS) {
            # store old value
            push(s, count);
            count = 0;
            # find uses and increment counter
            foreach (i: int; def(node.fields[i]))
                foreach (j: int; node.fields[i].modifiers[j].kind == ModifierKind.OTHER
                                 && node.fields[i].modifiers[j].other == "transient")
                    count++;
        } else
            stop;
    }
    after node: Declaration -> {
        # output result
        TransientTotal << count;
        TransientMax << p.id weight count;
        TransientMin << p.id weight count;
        TransientMean << count;
        # restore previous value
        count = pop(s);
    }
});
```

Figure D.22: E6. How is transient keyword used in Java?

```
# Computes Number of Attributes (NOA) for each project, per-type
# Output is: NOA[ProjectID][TypeName] = NOA value
p: Project = input;
NOA: output sum[string][string] of int;

visit(p, visitor {
  # only look at the latest snapshot
  before n: CodeRepository -> {
    snapshot := getsnapshot(n);
    foreach (i: int; def(snapshot[i]))
      visit(snapshot[i]);
    stop;
  }
  before node: Declaration ->
    if (node.kind == TypeKind.CLASS)
      NOA[p.id][node.name] << len(node.fields);
});
```

Figure D.23: F1. What are the number of attributes (NOA), per-project and per-type?

```
# Computes Number of Public Methods (NPM) for each project, per-type
# Output is: NPM[ProjectID][TypeName] = NPM value
p: Project = input;
NPM: output sum[string][string] of int;

visit(p, visitor {
  # only look at the latest snapshot
  before n: CodeRepository -> {
    snapshot := getsnapshot(n);
    foreach (i: int; def(snapshot[i]))
      visit(snapshot[i]);
    stop;
  }
  before node: Declaration ->
    if (node.kind == TypeKind.CLASS)
      foreach (i: int; has_modifier_public(node.methods[i]))
        NPM[p.id][node.name] << 1;
});
```

Figure D.24: F2. What are the number of public methods (NPM), per-project and per-type?

BIBLIOGRAPHY

[1] Promise 2009 dataset. http://promisedata.org/2009/datasets.html.

[2] Hierarchical visitor pattern, c2 pattern repository. http://c2.com/cgi/wiki?HierarchicalVisitorPattern, 2012.

[3] Sourceforge website. http://sourceforge.net/, 2012.

[4] Sourcerer website. http://sourcerer.ics.uci.edu/, 2012.

[5] Boa website. http://boa.cs.iastate.edu/, 2013.

[6] Drupal website. http://www.drupal.org/, 2013.

[7] Apache Hadoop. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/lib/ChainReducer.html, 2013.

[8] Apache Hadoop. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/mapred/lib/ChainMapper.html, 2013.

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

[10] Apache Software Foundation. Giraph: iterative graph processing. http://giraph.apache.org/, 2013.

[11] Apache Software Foundation. Hama project. http://hama.apache.org/, 2013.

[12] Apache Software Foundation. Hbase: Open source implementation of Bigtable. http://hbase.apache.org/, 2013.

[13] A. M. Ayad and J. F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 419–430, New York, NY, USA, 2004. ACM.

[14] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, Dec. 2004.

[15] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[16] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. In *WASDeTT-3: 3rd International Workshop on Academic Software Development Tools and Techniques*, pages 1–35, 2010.

[17] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA, pages 543–562, 2008.

[18] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. An empirical study on limits of clone unification using generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, SEKE, pages 109–114, 2005.

[19] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Facilitating software evolution research with Kenyon. In *ESEC/FSE'05: 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 177–186, 2005.

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

[21] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. *SIGPLAN Not.*, 33(10), Oct. 1998.

[22] T. S. BV. TIOBE Programming Community Index for July 2012. Technical report, TIOBE Software BV, 2012.

[23] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: the case of Smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR, pages 23–32, 2011.

[24] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *VLDB'08: Proceedings of the VLDB Endowment*, 1(2):1265–1276, Aug. 2008.

[25] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *PLDI'10: 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–375, 2010.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008.

[27] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th international conference on Static Analysis*, SAS, pages 1–18, 2003.

[28] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI'04: 6th Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[29] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *CSCW*, pages 107–114, 1992.

[30] R. Dyer. Task fusion: improving utilization of multi-user clusters. In *Student research competition at the 4th ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH, 2013.

[31] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE'13: 35th International Conference on Software Engineering*, pages 422–431, 2013.

[32] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. A large-scale empirical study of Java language feature usage. Technical Report 13-02, Iowa State University, 2013.

[33] R. Dyer, H. Rajan, and T. N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of AST nodes. In *GPCE'13: 12th International Conference on Generative Programming: Concepts & Experiences*, page (to appear), 2013.

[34] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, 2000.

[35] M. Gabel and Z. Su. A study of the uniqueness of source code. In *FSE'10: 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 147–156, 2010.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

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

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

[39] T. Gorschek, E. Tempero, and L. Angelis. A large-scale empirical study of practitioners' use of object-oriented concepts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ICSE, pages 115–124, 2010.

[40] J. Gosling, B. Joy, and G. Steele. *Java(TM) Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1996.

[41] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2000.

[42] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*. Addison-Wesley Professional, 3rd edition, 2005.

[43] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *Java(TM) Language Specification*. Prentice Hall, Java SE 7 edition, 2013.

[44] M. Grechanik, C. McMillan, L. DeFerrari, M. Comi, S. Crespi, D. Poshyvanyk, C. Fu, Q. Xie, and C. Ghezzi. An empirical investigation into a large-scale Java open source code repository. In *International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 11:1–11:10, 2010.

[45] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: scalable source code queries with datalog. In *ECOOP'06: 20th European conference on Object-Oriented Programming*, pages 2–27, 2006.

[46] A. Hindle and D. M. German. SCQL: a formal model and a query language for source control repositories. In *Proceedings of the 2005 international workshop on Mining Software Repositories*, MSR, pages 1–5, 2005.

[47] A. Holkner and J. Harland. Evaluating the dynamic behaviour of Python applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91*, ACSC, pages 19–28, 2009.

[48] M. Hoppe and S. Hanenberg. Do developers benefit from generic types? An empirical comparison of generic and raw types in Java. In *4th ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH, 2013.

[49] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Eurosys'07: 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 59–72, 2007.

[50] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD'03: 2nd international conference on Aspect-oriented software development*, pages 178–187, 2003.

[51] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[52] L. C. L. Kats and E. Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, pages 444–463, 2010.

[53] M. Kimmig, M. Monperrus, and M. Mezini. Querying source code with natural language. In *ASE'11: 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 376–379, 2011.

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

[55] J. Lawall, J. Brunel, N. Palix, R. Hansen, H. Stuart, and G. Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in linux code. In *IEEE/IFIP International Conference on Dependable Systems Networks*, DSN, pages 43–52, 2009.

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

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

[58] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS, pages 139–160, 2005.

[59] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software*, ESOP, pages 95–111, 2009.

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

[61] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA'05: 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, 2005.

[62] L. Meyerovich and A. Rabkin. Empirical analysis of programming language adoption. In *4th ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH, 2013.

[63] D. Miner and A. Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, 2012.

[64] R. Muschevici, A. Potanin, E. Tempero, and J. Noble. Multiple dispatch in practice. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA, pages 563–582, 2008.

[65] NASA. MERRA analytic services: orchestrating big data with climate analytics-as-a-service. http://www.nas.nasa.gov/SC13/demos/demo24.html, 2013.

[66] Ohloh. Ohloh source control management library. https://github.com/blackducksw/ohloh_scm, 2012.

[67] B. C. d. S. Oliveira, M. Wang, and J. Gibbons. The visitor pattern as a reusable, generic, type-safe component. In *OOPSLA'08: 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 439–456, 2008.

[68] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD/PODS'08: 35th ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[69] D. Orleans and K. J. Lieberherr. DemeterJ. Technical report, Northeastern University, 2001.

[70] D. Orleans and K. J. Lieberherr. DJ: Dynamic adaptive programming in Java. In *REFLECTION'01: 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 73–80, 2001.

[71] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. SourcererDB: an aggregated repository of statically analyzed and cross-linked open source Java projects. In *MSR'09: 6th IEEE International Working Conference on Mining Software Repositories*, pages 183–186, 2009.

[72] J. Ovlinger and M. Wand. A language for specifying recursive traversals of object structures. In *OOPSLA'99: 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–81, 1999.

[73] V. Pankratius and A.-R. Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA, pages 43–52, 2011.

[74] C. Parnin, C. Bird, and E. R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *8th IEEE International Working Conference on Mining Software Repositories*, MSR, 2011.

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

[76] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps, 2010.

[77] E. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12:23–49, 1999.

[78] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP, pages 52–78, 2011.

[79] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI, 2010.

[80] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.

[81] W. Shang, B. Adams, and A. E. Hassan. An experience report on scaling tools for mining software repositories using MapReduce. In *ASE'10: 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 275–284, 2010.

[82] Stack Overflow. http://stackoverflow.com/questions/3143484/, 2010.

[83] Y. Tao, M. L. Yiu, D. Papadias, M. Hadjieleftheriou, and N. Mamoulis. Rpj: producing fast join results on streams through rate-based optimization. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 371–382, New York, NY, USA, 2005. ACM.

[84] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 799–810. VLDB Endowment, 2006.

[85] E. Tempero. How fields are used in Java: An empirical study. In *Proceedings of the 20th Australian Software Engineering Conference*, ASWEC, pages 91–100, 2009.

[86] E. Tempero, J. Noble, and H. Melton. How do Java programs use inheritance? An empirical study of inheritance in Java software. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP, pages 667–691, 2008.

[87] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a warehousing solution over a map-reduce framework. *VLDB'09: Proceedings of the VLDB Endowment*, 2(2):1626–1629, Aug. 2009.

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

[89] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

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

[91] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA, pages 419–431, 2004.

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

[93] D. Wheeler. Flawfinder. http://www.dwheeler.com/flawfinder/.

[94] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD/PODS'07: 34th ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, 2007.

[95] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI'08: 10th Symposium on Operating System Design and Implementation*, pages 1–14, 2008.

[96] C. Yue and H. Wang. Characterizing insecure JavaScript practices on the web. In *Proceedings of the 18th international conference on World Wide Web*, WWW, pages 961–970, 2009.

[97] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *The VLDB Journal*, 21(5):611–636, Oct. 2012.